Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Mathematics

Finite Element Methods with Applications in Mathematical Finance

Jonathan Kelly

Supervised by Kirk M. Soodhalter

April 7, 2024

Submitted in partial fulfilment of the requirements for the degree of
B.A. Mathematics

Declaration

I hereby declare that this Capstone Project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Signed:  Date: 7/4/24

Abstract

In this paper we will discuss the Finite Element Method, a technique primarily used in Engineering to numerically solve a range of Partial Differential Equations. The aim of this project is to explore the theory behind the method, examine in detail each of the steps in the numerical method. We shall then demonstrate the FEM on different types of differential equation and perform error analysis on the numerical solution, and lastly apply the approach to option pricing problems in mathematical finance.

Acknowledgements

I would like to thank Kirk Soodhalter for the support and guidance he provided over the duration of the project.

I would also like to thank my friends from the maths course for the feedback they gave on my project.

Contents

1 | Introduction

There are many different numerical methods that can be used to solve Partial Differential Equations. Since many PDE's do not have a closed form solution, a popular method with practical use in areas such as Engineering and Physics is the **Finite Element Method**, which is used to **approximate complex continuous functions as discrete models.**

For example, take a particular PDE problem, such as the 2-dimensional Poisson equation, $-\nabla^2 u = f(x, y)$. To demonstrate the basics of the finite element method this PDE is often used as a model problem.

The first step involves converting our PDE into its weak form by multiplying with a test function $v(x, y)$ and integrating over the domain to get $\int_\Omega \nabla u \cdot \nabla v \; dx \; dy = \int_\Omega f \cdot v \; dx \; dy$. We then take the problem's domain Ω which would be broken into subdomains Ω_n, or **elements**. From there each element would be approximated using an interpolation function $\phi_n(x, y)$. Once we apply the discretisation to the weak form above, we obtain a set of integral equations which can be represented in matrix form $A\mathbf{u} = \mathbf{f}$. Lastly, we can finally solve for $\mathbf{u}$ and perform post-processing to visualise our results.

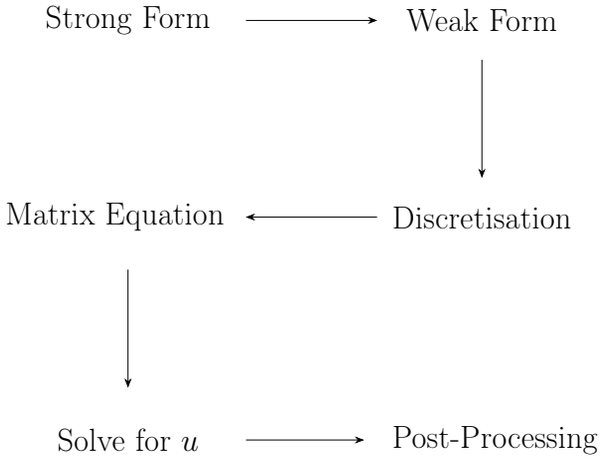

Figure 1.1: General outline of steps for the Finite Element Method

The above is just one example of where the finite element method can be applied, and there are many other examples where it can be demonstrated. In this project we shall explore the reasons why this method has practical use, and how it is applied to solve different types of Differential Equations, ranging from ODE's to Parabolic PDE's. Given this is a numerical method, we shall demonstrate the FEM through programming on MATLAB, analyse our results and also perform error analysis on example problems. Finally, we shall apply the FEM to a real world problem; option pricing in mathematical finance. Our final goal is to apply the FEM to the Black-Scholes PDE and to return numerical solutions using the outlined methods researched in this paper.

2 | The Finite Element Method: An Introduction

In this chapter we shall achieve the following objectives

- How the finite element method works and how it is implemented on a model problem for a 1D ODE and 2D PDE
- Briefly discuss the mathematical theory which allows the finite element method to be implemented

2.1 FEM on a 1D Model Problem

Since the finite element method varies based on the type of differential equation and boundary conditions, we shall introduce all the parts to the method via. a model problem.

This will be the **Poisson equation with homogeneous Dirichlet boundary conditions**

$$-\nabla^2 u = f\,, \quad u = 0 \text{ on } \partial\Omega$$

In the case of our 1-dimension formulation we take the problem from [1].

$$-u''(x) = f(x), \ \ \Omega = [0,1] \ \text{ with boundary condition } \ u(x) = 0 \ \ for \ \ x = 0, 1.$$

2.1.1 The Weak Formulation

In certain cases, depending on the value of $f(x)$, there does not exist a sufficiently smooth function u satisfying the above equation. Therefore, we must formulate an alternative description to the boundary value problem, that allows for solutions that may not be smooth or continuous. This involves determining a new formulation for our problem so we can determine solutions for it. Here we shall cite from [1] and [2].

The formulation we will use is the *weak formulation*, which is "less restrictive". In order to change to this form, we must multiply the equation by a test function v and integrate over the domain Ω:

$$\int_\Omega (u'' + f)v = 0.$$

Given **v** will be sufficiently smooth, the "smoothness" required of **u** can be reduced by using integration by parts:

$$\left(\int_\Omega u'' \cdot v\right) + \left(\int_\Omega f \cdot v\right) = \left(\int_\Omega u' \cdot v'\right) \ - \underbrace{\left(\int_{\partial\Omega} v \cdot \partial_n u\right)}_{=0} + \left(\int_\Omega f \cdot v\right) = 0.$$

Therefore we obtain the following equation which is our **weak formulation**:

$$\int_\Omega u' \cdot v' = \int_\Omega f \cdot v.$$

We now introduce short hand notations for our integrals below for simplicity from [2]:

$$(f, v) = \int_\Omega f \cdot v \ , \tag{2.1}$$

$$a(u, v) = \int_\Omega u' \cdot v' \ . \tag{2.2}$$

The notation for (2.2) will be primarily used in the higher dimension cases.

We therefore arrive at the **weak formulation of the model problem:**

$$\text{Find u} \in V \text{ such that } a(u, v) = (f, v) \ \ \forall v \in V. \tag{2.3}$$

As we can see, the new formulation 2.3 doesn't require second derivatives, only first derivatives, which means we have weaker restrictions on our model problem.

2.1.2 Solution Spaces

The next question arises: In what circumstances is a solution u to this equation and the test functions v **meaningful?** To answer this, we must choose an **appropriate function space**.

So what function space V should we choose? If we look at the boundary conditions $u(0) = u(1) = 0$, we ideally want the test functions v to be well behaved so that the integrals exist and are well defined. We also need v and v' to be *square integrable* on the domain and for v to vanish at $x = 0, 1$. We can therefore begin by defining the space of "Square Integrable Functions" L_2 below (or Lebesgue space),

$$L_2(\Omega) = \left\{ v : \Omega \mapsto \mathbb{R} \mid \int_\Omega v^2 < \infty \right\}.$$

The integrals in the equation will now be well-defined. As we mentioned above, we also want all first derivative to also be in $L_2(\Omega)$ so we introduce the Sobolev Space $\mathcal{H}^1(\Omega)$:

$$H^1(\Omega) = \left\{ v : \Omega \mapsto \mathbb{R} \mid v,\, v' \in L_2(\Omega) \right\}.$$

Lastly, since we want to abide by the boundary conditions of the problem, we shall use the space $H_0^1(\Omega)$

$$H_0^1(\Omega) := \left\{ v \in H^1(\Omega) : v(0) = v(1) = 0 \right\}.$$

Now we arrive at the new formulation of the problem:

$$\text{Find u} \in H_0^1(\Omega) \text{ such that } a(u, v) = (f, v) \ \ \forall v \in H_0^1(\Omega)\,. \tag{2.4}$$

This new space is larger than the space of continuous functions with piece-wise continuous derivatives, and it is the largest space where the solutions to the differential equation will be of value and meaningful. The method described in (2.4) is often called a **Galerkin Method**.

Remark: The function spaces L_2, H^1, H_0^1 are all examples of **Hilbert Spaces**, which means they are all linear spaces with a scalar product and corresponding norm $||\cdot||$ and are complete (every Cauchy sequence with respect to $||\cdot||$ is convergent) . The new formulation of problem above (2.4) is of interest since we will be able to calculate a basic error estimate for the FEM when using this norm. (See Chapter 4)

2.1.3 Finite Elements: Discretisation of the domain

Now that we have reformulated our problem it is time to introduce the method in which we shall find an approximate solution: the **finite element method.**

For our model problem, the domain is the real line $\Omega = [0, 1]$. The mesh is simply *the domain split into discrete intervals*, which we call "elements". Let us start by dividing our domain into n sub-intervals:

$$x_0 = 0 < x_1 < ... < x_n = 1.$$

We want to have a uniform mesh; in other words, each node (or x_i) should be equidistant from each other:

$$x_i = ih, \;\; h = \frac{1}{n}, \;\; 0 \leq i \leq n, \;\; h = x_{i+1} - x_i \; .$$

The interval $K_i = [x_i, x_{i+1}]$ will be our *elements*, and $\bigcup_i K_i = T_h$ will be our *mesh* which equals our domain.

When we consider each element individually we want to find a way of *approximating the function inside that element.* An obvious option would be to approximate our function using polynomials. So taking reference from [2] and [3], for our 1-dimensional case we will use the space of linear polynomials:

$$P_1 := \{a_o + a_1 x | a_0, a_1 \in \mathbb{R}\}.$$

Now, we can convert our infinite dimensional space V down to a finite dimensional space using our mesh parameter h:

$$V_h := \{v \in \mathcal{C}[0, 1] \;\mid\; v|_{K_i} \in P_i \, , K_i := [x_i, x_{i+1}], \, 0 \leq i \leq n, \;\; v(0) = v(1) = 0\}.$$

This finite dimensional space is where we shall find the finite element approximation for our model problem.

We mentioned in the weak formulation how we need to use *test functions* in our search for the solution. In V_h our test functions are usually called *shape functions* and can be represented by hat functions ϕ_i

$$\phi_i(x) = \begin{cases} 0 & \text{if} \quad x \notin [x_{i-1}, x_{i+1}], \\ \frac{x - x_{i-1}}{x_i - x_{i-1}} & \text{if} \quad x \in [x_{i-1}, x_i], \\ \frac{x_{i+1} - x}{x_{i+1} - x_i} & \text{if} \quad x \in [x_i, x_{i+1}]. \end{cases}$$

with the property that

$$\phi_i(x_j) = \begin{cases} 1 & \text{if} \quad i = j, \\ 0 & \text{if} \quad i \neq j. \end{cases}$$

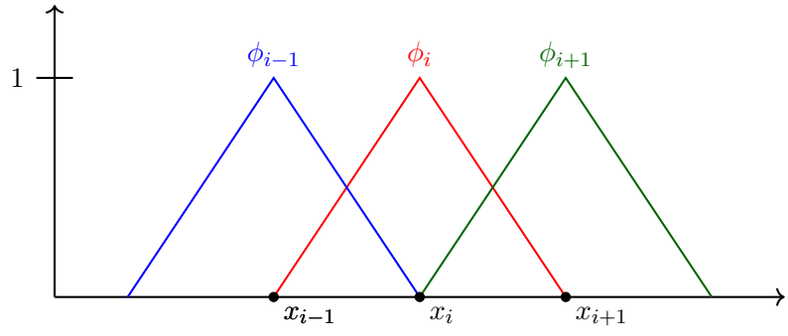

Figure 2.1: A visual representation of how the hat functions ϕ_i look on a given interval

These shape functions are essentially **basis functions** for the finite dimensional space V_h, and what we can see happening in Figure (2.1) is that every function $u_h \in V_h$ that approximates $u \in V$ is a **linear combination** of the basis functions:

$$u_h(x) = \sum_{i=1}^{n} u_{h,i}\phi_i(x) \;\; \forall x \in [0,1], \;\; u_{h,i} \in \mathbb{R} \; .$$

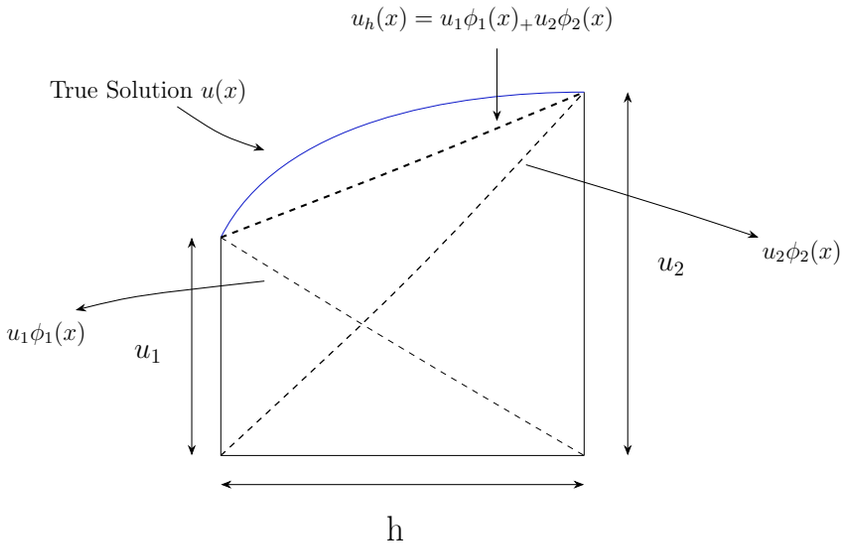

Figure 2.2: A visual representation of how Galerkin Discretisation u_h approximates a given function u

2.1.4 Converting from V to V_h to obtain a matrix equation

Next we need to convert from our initial infinite dimensional space V to a finite dimensional space V_h . (Here we shall cite from [3])

Our infinite dimensional problem is:

$$\text{Find } u \in V \text{ such that } \int_\Omega u' \cdot \phi' \, dx = \int_\Omega f \cdot \phi \, dx \quad \forall \phi \in V \; .$$

When we change to V_h we now change to a finite dimensional problem where we can find the finite element solution u_h:

$$\text{Find } u_h \in V_h \text{ such that } \int_\Omega u_h' \cdot \phi_j' \;\; dx = \int_\Omega f \cdot \phi_j \;\; dx \quad \forall \phi_j \in V_h \; . \tag{2.5}$$

However since $u_h(x) = \sum_{i=1}^{n} u_i\phi_i(x)$, we can compute individually for each j the components of this integral:

$$\int_\Omega u_h' \cdot \phi_j' \, dx = \int_\Omega f \cdot \phi_j \, dx$$

$$\implies \int_\Omega \left(\sum_{i=1}^{n} u_i\phi_i(x) \right)' \cdot \phi_j' \, dx = \int_\Omega f \cdot \phi_j \, dx \;\; \forall j = 1, ..., n \; .$$

Since the coefficients of u_h are constant, we can take the summation out of the integral to create a linear equation system:

$$\int_\Omega \left(\sum_{i=1}^{n} u_i \phi_i'(x) \right) \cdot \phi_j' \, dx = \int_\Omega f \cdot \phi_j \, dx \implies \sum_{i=1}^{n} u_i \int_\Omega \phi_i' \cdot \phi_j' \, dx = \int_\Omega f \cdot \phi_j \, dx \, .$$

- Our integral $\int_\Omega \phi_i' \cdot \phi_j'$ becomes a matrix $A = a_{ij}$.
- The right hand side $\int_\Omega f \cdot \phi_j$ is a vector which we will call B.
- The components u_i make up the vector U.

Thus we obtain the linear system equation:

$$AU = B \tag{2.6}$$

where A is called the *stiffness matrix* and B is called the *load vector (or force vector).*

Visually this will look like (while referencing [4]),

$$\begin{bmatrix} a(\phi_1, \phi_1) & a(\phi_1, \phi_2) & ... & a(\phi_1, \phi_{n-1}) \\ a(\phi_2, \phi_1) & a(\phi_2, \phi_2) & ... & a(\phi_2, \phi_{n-1}) \\ \vdots & \vdots & \vdots & \vdots \\ a(\phi_{n-1}, \phi_1) & a(\phi_{n-1}, \phi_2) & ... & a(\phi_{n-1}, \phi_{n-1}) \end{bmatrix} \begin{bmatrix} u_{h,1} \\ u_{h,2} \\ \vdots \\ u_{h,n-1} \end{bmatrix} = \begin{bmatrix} (f, \phi_1) \\ (f, \phi_2) \\ \vdots \\ (f, \phi_{n-1}) \end{bmatrix}, \tag{2.7}$$

From there we can refer to the next chapter to see how the Finite Element Method for the model problem can be implemented on MATLAB so we can solve (2.6).

2.2 FEM on 2D PDE's

2.2.1 Weak Formulation in 2D

So far we have covered the concepts of the Finite Element Method with respect to to the 1-dimensional Poisson Equation. When we switch to a 2-dimensional problem the method is similar, however there are a few parts that need to be reformulated as we shall see below.

Below we shall use the 2D Poisson equation as our model problem, whilst referencing ([1]) and ([3]).

Find u such that:

$$-\Delta u = f, \ \text{ in } \ \Omega \tag{2.8}$$

$$u = 0, \ \text{ on } \ \partial\Omega \tag{2.9}$$

where

$$\Delta u = \frac{\partial^2 x_1}{\partial x_1^2} + \frac{\partial^2 x_2}{\partial x_2^2}$$

is the *Laplacian Operator* , f is a function in $L_2(\Omega)$ and $\Omega \subset \mathbb{R}^2$.

Before we can progress any further we need to introduce some preliminaries. The first one is one of *Green's Formulae.*

To derive it, we first start with the divergence theorem for two dimensions:

$$\int_\Omega \left(\frac{\partial f_1}{\partial x_1} + \frac{\partial f_2}{\partial x_2} \right) dx = \int_{\partial\Omega} f \cdot n \, ds$$

where:

- $f = (f_1, f_2)$ is a vector-valued function defined on the domain Ω.
- $n = (n_1, n_2)$: the outward unit normal to the boundary of the domain.
- dx : the element of area in $\mathbb{R}^2$.
- ds : the element of arc length along $\partial\Omega$.

If we apply the divergence theorem to $f = (vw, 0)$ and $f = (0, vw)$, we get the following:

$$\int_\Omega \frac{\partial v}{\partial x_i} w \; dx + \int_\Omega v \frac{\partial w}{\partial x_i} \; dx = \int_{\partial\Omega} v\, w\, n_i \; ds, \quad i = 1, 2 \; .$$

We let $\nabla v = \left(\frac{\partial v}{\partial x_1}, \frac{\partial v}{\partial x_2}\right)$ be the gradient of v , and we obtain the following:

$$\int_\Omega \nabla v \cdot \nabla w \; dx = \int_\Omega \left[\frac{\partial v}{\partial x_1}\frac{\partial w}{\partial x_1} + \frac{\partial v}{\partial x_2}\frac{\partial w}{\partial x_2} \right] \; dx$$

$$= \int_\Omega \left[v \, \frac{\partial w}{\partial x_1} \, n_1 + v \, \frac{\partial w}{\partial x_2} \, n_2 \right] ds \;\; - \int_\Omega v \left[\frac{\partial^2 w}{\partial x_1^2} + \frac{\partial^2 w}{\partial x_2^2} \right] ds \;\; \leftarrow (IBP)$$

$$= \int_{\partial\Omega} v \frac{\partial w}{\partial n} ds - \int_\Omega v \; \Delta w \; \; dx$$

We therefore obtain Green's Formula:

$$\int_\Omega \nabla v \cdot \nabla w \; dx = \int_{\partial\Omega} v \frac{\partial w}{\partial n} ds - \int_\Omega v \; \Delta w \; \; dx$$

where $\frac{\partial w}{\partial n}$ is our *normal derivative* :

$$\frac{\partial w}{\partial n} = \frac{\partial w}{\partial x_1} n_1 + \frac{\partial w}{\partial x_2} \, n_2 \, .$$

We can now describe the **weak formulation for the 2D Poisson problem.**

Find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \int_\Omega f \, v \; dx \quad \forall \, v \in V \tag{2.10}$$

where the LHS of (2.10) is

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \int_\Omega \left[\frac{\partial u}{\partial x_1}\frac{\partial v}{\partial x_1} + \frac{\partial u}{\partial x_2}\frac{\partial v}{\partial x_2} \right] dx$$

and

$$V \; = \left\{ v : v \, \text{is continuous on } \Omega, \frac{\partial v}{\partial x_1} \text{ and } \frac{\partial v}{\partial x_2} \text{ are piecewise continuous on } \Omega \text{ and } v = 0 \text{ on } \partial\Omega \right\} .$$

To see how (2.10) follows from (2.8), we multiply the LHS of (2.8) by a test function $v \in V$ and integrate over the domain Ω. Then by Green's Formula we have:

$$\int_\Omega f \, v \; dx = - \int_\Omega v \; \Delta u = \int_\Omega \nabla v \cdot \nabla u \; dx - \underbrace{\int_{\partial\Omega} v \frac{\partial u}{\partial n} ds}_{= \, 0} \; .$$

Therefore we obtain the same formulation as the 1D model problem

$$a(u, v) = (f, v) \quad \forall v \in V \, ,$$

where we introduce a similar space for V as we did in the 1D model problem:

$$V = H_0^1(\Omega) = \{ v : v, \nabla v \in L_2(\Omega) \, , v|_{\partial\Omega} = 0 \}$$

We can now discuss the conversion from V to V_h below.

2.2.2 Discretisation in 2D: Building a mesh

Our next aim is to construct a finite dimensional space V_h of V . In order to do this, we want to construct a space of piece-wise polynomial functions that a computer is able to represent. For this we shall reference [4] and [3].

The main method that is generally used is to partition the domain into a set of non-overlapping triangles, known as a *triangulation*.

For instance, take domain $\Omega \subset \mathbb{R}^2$, and let $T_h = K_1, ..., K_m$ be a set of non-overlapping triangles. We let $\Omega = \bigcup_{K \in T_h} K$, where the intersection of two triangles is an edge, a corner, or empty. We also want to define the nodes $N_i,\ i = 1, ..., M$ of the triangles. To help measure the size of a triangle, we introduce the mesh parameter $h = \max_{K_j \in T_h} h_j$, where $h_j =$ Longest side of triangle K_j . Typically we also introduce a way of measuring the quality of a triangulation. Given $\rho_j =$ Diameter of circle inscribed in K_j we define the inequality

$$1 \geq \frac{\rho_j}{h_j} \geq \beta > 0,$$

where the constant β will measure the quality of our triangulation. The quality of the triangulation increases as we increase the value of β.

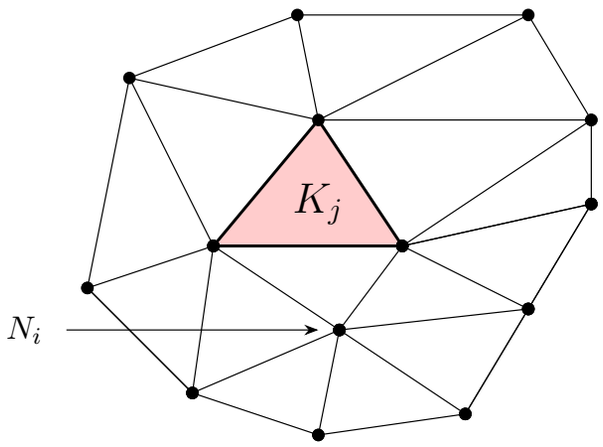

Figure 2.3: Here we take a domain Ω and create a triangulation mesh.

We can now introduce the piece-wise polynomial space where the test functions shall be used.

Let K be a triangle. Then the space of linear functions on K is

$$P_1(K) = \{v : v = c_0 + c_1 x_1 + c_2 x_2,\ (x_1, x_2) \in K,\ c_0, c_1, c_2 \in \mathbb{R}\}$$

If we want appropriate parameters to describe a function $v \in V_h$ we choose **the value of v at the nodes N_i of T_h** but exclude the nodes on the boundary since $v = 0$ on $\partial\Omega$.

2.2.3 Converting from V to V_h in 2D

Our next step is to convert our infinite dimensional space V to a finite dimensional space V_h , which will be

$$V_h = \left\{v : v \in C^0(\Omega),\ \ v|_K \in P_1(K)\ \ \forall K \in T_h,\ \ v|\partial\Omega = 0\right\},$$

where $C^0(\Omega)$ is the space of all continuous functions on Ω. Below we have a theorem which helps us define our basis functions for V_h .

Theorem 2.2.1 (Local DoF's). *Let $K \in \mathcal{T}_h$ be a triangle with vertices $a^i,\ i = 1, 2, 3$ which are its local degrees of freedom. Then, a linear function $p_1(x, y) = \alpha + \beta x + \gamma y$ defined on K is uniquely determined by its local degrees of freedom*

Proof: We have $p_1(x, y) = \alpha + \beta x + \gamma y$. We know our vertices will be equal to some coordinates $a^i = (x_i, y_i),\ i = 1, 2, 3$. We obtain the following set of equations

$$\begin{aligned}\alpha + \beta x_1 + \gamma y_1 &= a^1,\\ \alpha + \beta x_2 + \gamma y_2 &= a^2,\\ \alpha + \beta x_3 + \gamma y_3 &= a^3.\end{aligned}$$

This becomes a linear system of equations

$$\begin{bmatrix}1 & x_1 & y_1\\ 1 & x_2 & y_2\\ 1 & x_3 & y_3\end{bmatrix}\begin{bmatrix}\alpha\\ \beta\\ \gamma\end{bmatrix} = \begin{bmatrix}v_1\\ v_2\\ v_3\end{bmatrix} \rightarrow Ba = b.$$

It can be easily checked that $det(B) = 2(\text{Area of triangle}) \neq 0$ since $\frac{\rho_j}{h_j} \geq \beta > 0$, which implies the linear system has a unique solution. □

The theorem above allows us to use the nodal points of a given triangle as parameters for functions in V_h . Next, we define the following basis functions: Let $\phi_j \in V_h,\ j = 1, ..., M$ such that

$$\phi_j(N_i) = \begin{cases} 1 & \text{if} \quad i = j\\ 0 & \text{if} \quad i \neq j\end{cases} \quad , \quad i, j = 1, ..., M.$$

By using these basis functions as our test functions we can now apply the finite element method to a 2D problem. The graph below demonstrates how the basis functions work.

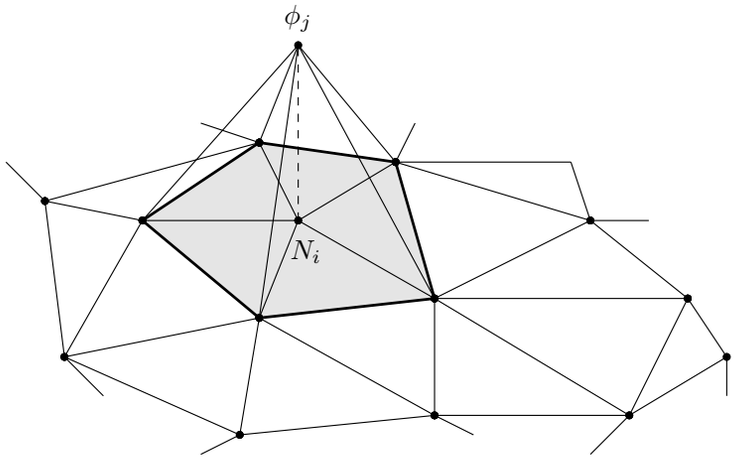

Figure 2.4: Here we see the basis function ϕ_j
in action and how it equals 1 when it is vertically above the corresponding node N_j

We can therefore represent a function $v \in V_h$ as:

$$v(x) = \sum_{j=1}^{n_p} \eta_j \phi_j(x),\ \ \eta_j = v(N_j),\ \ x \in \Omega. \tag{2.11}$$

where η_j represents the nodal values of v and n_p is the number of nodes in our triangulation.

We can now move onto the derivation for the finite element solution. Let $\{\phi_i\}_{i=1}^{n_i}$ be the basis for V_h, where n_i are the interior nodes of the mesh (the hat functions of V_h vanish at the boundary). We replace V with V_h to get the following finite element method: Find $u_h \in V_h$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v \, dx = \int_\Omega f \, v \, dx, \quad \forall v \in V_h \, .$$

We can also use our short-hand notation we introduced in the 1D case: Find $u_h \in V_h$ such that

$$a(u_h, v) = (f, v) \quad \forall v \in V_h. \tag{2.12}$$

We use the same methods as before in the previous 1D model problem to obtain the linear system of equations:

$$AU = B$$

where we have

$$A_{ij} = \int_\Omega \nabla \phi_i \cdot \nabla \phi_j \, dx, \quad i, j = 1, 2, ..., n_i \, ,$$

$$b_i = \int_\Omega f \phi_i \, dx \quad i = 1, 2, ..., n_i$$

as our stiffness matrix and load vector respectively. In the next chapter we can go into detail about how we can solve this system of linear equations.

3 | Coding the FEM: Implementation on MATLAB

Our next goal is to implement our method on a coding language. In this project, our language of choice is MATLAB, a popular mathematical coding language. In this chapter, we address the following objectives:

- Describe the algorithms that assemble the matrix equations on MATLAB
- Discuss how numerical integration can be implemented for the FEM.
- Demonstrate the use of reference elements to allow for quicker computation.

3.1 Solving the 1D Model Problem on MATLAB

3.1.1 Outline of main steps

So far we completed the following, taking reference from [2].

- Creating a domain Ω and decomposition of this domain into elements.
- Assemble the linear system of equations $Au = b$ from the weak formulation.

Next we need to solve the above equation using MATLAB. Below is a rough overview of the steps we shall implement in MATLAB.

- We compute *locally on each mesh element* the matrix entries and RHS entries on a *master cell (reference element)* and insert the respective values at their global places in the matrix A .
- In the assembly, the integrals are evaluated using (typically) numerical quadrature in order to allow for more general expressions (This is a common problem when higher-order polynomials need to be evaluated).
- Finally, obtain numerical solution to the linear system $Au = b$.
- **Postprocessing**: Here we write the solution data $(U_j, 1 \leq j \leq n)$ into a file that can be read from a graphic visualisation program.

3.1.2 Local and Global Assembly

We refer back to (2.7).

$$\begin{bmatrix} a(\phi_1,\phi_1) & a(\phi_1,\phi_2) & \dots & a(\phi_1,\phi_{n-1}) \\ a(\phi_2,\phi_1) & a(\phi_2,\phi_2) & \dots & a(\phi_2,\phi_{n-1}) \\ \vdots & \vdots & \vdots & \vdots \\ a(\phi_{n-1},\phi_1) & a(\phi_{n-1},\phi_2) & \dots & a(\phi_{n-1},\phi_{n-1}) \end{bmatrix} \begin{bmatrix} u_{h,1} \\ u_{h,2} \\ \vdots \\ u_{h,n-1} \end{bmatrix} = \begin{bmatrix} (f,\phi_1) \\ (f,\phi_2) \\ \vdots \\ (f,\phi_{n-1}) \end{bmatrix},$$

Given that $AU = \sum_{i=1}^{n} u_i \int_\Omega \nabla\phi_i(x) \cdot \nabla\phi_j$, we have that the integrated entries in the stiffness matrix can be **split up element by element**. In other words , for any given entry $a_{ij} = \int_\Omega \phi_i'(x)\phi_j'(x)dx$ we obtain the following:

$$\int_\Omega \phi_i'(x)\phi_j'(x)dx = \int_{x_0}^{x_1} \phi_i'(x)\phi_j'(x)dx + \int_{x_1}^{x_2} \phi_i'(x)\phi_j'(x)dx + \ ...+ \ + \int_{x_N-1}^{x_N} \phi_i'(x)\phi_j'(x)dx\,. \quad (3.1)$$

One crucial observation to make is that **each interval has only two nonzero basis functions**. What this means is that most of the integrals in (3.1) are zero. Therefore our matrix will look like the following: (taking from [4])

$$A = \begin{bmatrix} \int_{x_0}^{x_1} \phi_1'\phi_1'\,dx & 0 & ... & 0 \\ 0 & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & 0 \end{bmatrix} + \begin{bmatrix} \int_{x_1}^{x_2} \phi_1'\phi_1'\,dx & \int_{x_1}^{x_2} \phi_1'\phi_2'\,dx & ... & 0 \\ \int_{x_1}^{x_2} \phi_2'\phi_1'\,dx & \int_{x_1}^{x_2} \phi_2'\phi_2'\,dx & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & 0 \end{bmatrix}$$

$$+ \begin{bmatrix} 0 & 0 & 0 & ... & 0 \\ 0 & \int_{x_2}^{x_3} \phi_2'\phi_2'\,dx & \int_{x_2}^{x_3} \phi_2'\phi_3'\,dx & ... & 0 \\ 0 & \int_{x_2}^{x_3} \phi_3'\phi_2'\,dx & \int_{x_2}^{x_3} \phi_3'\phi_3'\,dx & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & ... & 0 \end{bmatrix} + \cdots + \begin{bmatrix} 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & \int_{x_N-1}^{x_N} \phi_{N-1}'\phi_{N-1}'\,dx \end{bmatrix}.$$

What we now do is highlight the contribution from each element:

$$K_i = \begin{bmatrix} \int_{x_i}^{x_{i+1}} \phi_i'\phi_i'\,dx & \int_{x_i}^{x_{i+1}} \phi_i'\phi_{i+1}'\,dx \\ \int_{x_i}^{x_{i+1}} \phi_{i+1}'\phi_i'\,dx & \int_{x_1}^{x_{i+1}} \phi_{i+1}'\phi_{i+1}'\,dx \end{bmatrix} \quad (3.2)$$

We shall call (3.2) the *local stiffness matrix* while A will be the *global stiffness matrix.* Below we demonstrate how the local stiffness matrix is calculated and then used in the assembly of the global stiffness matrix.

3.1.3 Gaussian Quadrature and Reference Element

We begin by providing the details for evaluating the stiffness matrix A and load vector b. In general, our stiffness matrix A will be too complicated to solve using general methods.

Let us return to our stiffness matrix $A = a_{ij} = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i = \sum_{K_i} \int_{K_i} \nabla\phi_j \cdot \nabla\phi_i$, where $K_i := (x_i, x_{i+1})$. As we explained earlier, we have split our domain into "elements" and approximate the function using hat elements. In our implementation, there are two issues that need to be considered:

1. We are calculating each integral individually (for each basis function) and this will take up time especially as we increase the dimensions.
2. The integrals that must be calculated will become very complex (especially in higher dimensions).

There are ways to overcome the problems mentioned above and we shall explore them while taking reference from [2] and [4].

To address problem 1, it is universal practise to introduce a method that means we don't have to define each hat function individually, especially since most of these local integrals are zero thanks to the property of the hat function.

We therefore introduce a *reference element* (also called a master element) for the domain Ω and transform the integrals into the reference element. See the equation below to get an idea of how this will be implemented for our stiffness matrix A and load vector b.

$$\int_{x_s}^{x_{s+1}} f(x)\,dx = \int_{-1}^{1} \tilde{f}(\xi)\,d\xi\,.$$

Without loss of generality, consider the physical element K_s which is one of s elements in our integral we need to evaluate. We have

$$K_s^{(h_s)} = (x_s, x_{s+1}), s = 0, ..., n \quad , h_s = x_{s+1} - x_s.$$

We define our master element $K^{(1)} = [-1, 1]$ and we define a local variable $\xi \in [-1, 1]$ which will be used to help transform our physical element into our master element.

We define the following linear transformation $T_s : K^{(1)} \to K_s^{h_s}$:

$$T_s(\xi) = x = x_s + \frac{x_{s+1} - x_s}{2}\,(1 + \xi)$$
$$\implies dx = \frac{x_{s+1} - x_s}{2} d\xi\,.$$

We then define the inverse of the transformation $T_s^{-1} : K_s^{h_s} \to K^{(1)}$:

$$T_s^{-1}(x) = \xi = \frac{x - x_s}{x_{s+1} - x_s} + \frac{x - x_{s+1}}{x_{s+1} - x_s},$$
$$\implies d\xi = \frac{2}{x_{s+1} - x_s} dx\,.$$

We then apply these transformations to our basis functions in the integral

$$\psi_i^h \text{ on } K_i^{(h_i)} \;:\; \psi_i^h(x) := \psi_i^1(T_s^{-1}(x)) = \psi_i^1(\xi)$$

with the following chain rule property

$$\partial_x \psi_i^h(x) = \partial_\xi \psi_i^1(\xi) \frac{d\xi}{dx}\,.$$

We can now apply the formulae above to our equations for A and B. For the load vector $B = (f, \phi_i)$ we have:

$$\int_{K_h} f(x)\psi_i^h(x)dx = \int_{K^{(1)}} f(T_h(\xi)) \cdot \psi_i^1(\xi) \cdot J \cdot d\xi,$$

and then for the stiffness matrix $A = a(\phi_i, \phi_j)$ we have

$$\int_{K_h} \partial_x \psi_i^h(x) \cdot \partial_x \psi_j^h(x)dx = \int_{K^{(1)}} \big(\partial_\xi \psi_i^1(\xi)\big) \cdot \xi_x \cdot \big(\partial_\xi \psi_j^1(\xi)\big) \cdot \xi_x \cdot J \cdot d\xi$$

where $J = det(\nabla T_s(\xi))$.

To solve problem 2, since the integrals will be become increasingly complex, it is usually not practical to find the exact integrals, which means we can introduce a way to *approximate* the integrals. The method we shall use is *Gaussian Quadrature*, and we have the formula described below.

$$\int_{x_s}^{x_{s+1}} f(x)\,dx \;= \int_{-1}^{1} \tilde{f}(\xi)\,d\xi \;\approx \sum_{i=0}^{n_q} \omega_i f(\xi_i)$$

Above we have

- ω_i is our quadrature weights
- ξ_i are the quadrature points ($n_q + 1$ in total).

We now combine this with the numerical quadrature we defined in the previous section to obtain the following equations for calculating A and B numerically:

$$\int_{K_s} \partial_x \psi_i^h(x) \cdot \partial_x \psi_j^h(x) dx \approx \sum_{k=0}^{n_q} \omega_k \left(\partial_\xi \psi_j^1(\xi_k) \cdot \xi_x\right) \cdot \left(\partial_\xi \psi_i^1(\xi_k) \cdot \xi_x\right) \cdot J \tag{3.3}$$

$$\int_{K_s} f(x)\psi_i^h(x) dx \approx \sum_{k=1}^{n_q} \omega_k f(T_h(\xi_k))\psi_i^1(\xi_k) \cdot J. \tag{3.4}$$

Finally, we need to determine what our Gaussian weights and points will be. Since we are integrating, we want the weights and points ω_i and ξ_i to be chosen such that the quadrature formula is as accurate as possible. This is done by assuming that our formula gives the same results for integrating a $2N-1$ polynomial (where N is the order of our gaussian quadrature). That way our points and weights are chosen so that it has the **highest algebraic precision** [5].

Below we give detail as to how we can derive the weights and points for **Gaussian Quadrature of Order 2.** As described above, we want $\omega_1\,, \omega_2\,, \xi_1$ and ξ_2 to be chosen so that the quadrature gives the same results as integrating a $2N-1$ polynomial ($N = 2$ in this case).

$$\int_{-1}^{1} g(\xi)d\xi = \omega_1 g(\xi_1) + \omega_2 g(\xi_2) = \int_{-1}^{1} \left(a_0 + a_1 x + a_2 x^2 + a_3 x^3\right) dx$$

The RHS simplifies down to

$$a_0(1-(-1)) + a_1\left(\frac{1^2-(-1)^2}{2}\right) + a_2\left(\frac{1^3-(-1)^3}{3}\right) + a_3\left(\frac{1^4-(-1)^4}{4}\right)$$

However we have that

$$\omega_1 g(\xi_1) + \omega_2 g(\xi_2) = \omega_1\left(a_0 + a_1\xi_1 + a_2\xi_1^2 + a_3\xi_1^3\right) + \omega_2\left(a_0 + a_1\xi_2 + a_2\xi_2^2 + a_3\xi_2^3\right)$$

When we rearrange we obtain the following equations:

$$1-(-1) = \omega_1 + \omega_2\,, \quad \left(\frac{1^2-(-1)^2}{2}\right) = \omega_1\xi_1 + \omega_2\xi_2\,,$$

$$\left(\frac{1^3-(-1)^3}{3}\right) = \omega_1\xi_1^2 + \omega_2\xi_2^2\,, \quad \left(\frac{1^4-(-1)^4}{4}\right) = \omega_1\xi_1^3 + \omega_2\xi_2^3\,.$$

This is a system of non-linear equations and they can be solved to give:

$$\omega_1 = \omega_2 = \frac{1}{2}, \;\; \xi_1 = -\frac{1}{\sqrt{3}}, \; \xi_2 = \frac{1}{\sqrt{3}}.$$

which are our respective quadrature weights and points.

3.1.4 Main Algorithm for solving the Model Problem

We shall define the basic algorithm (taking reference from [3]) that is used to solve the model problem below using MATLAB.

Algorithm 1

1. Create a mesh with n elements on the domain Ω and define the corresponding space of continuous piece-wise linear functions V_h
2. Compute the $(n-1) \times (n-1)$ stiffness matrix A and the $(n-1) \times 1$ load vector B with entries
$$A_{ij} = \int_\Omega \nabla\phi_i \cdot \nabla\phi_j, \;\; B_i = \int_\Omega f \cdot \nabla\phi_i$$
3. Solve the linear system
$$AU = B$$
and obtain U
4. We solve for u_h by setting
$$u_h(x) = \sum_{j=1}^{n-1} u_j(x)\phi_j(x)$$
and plot our solution.

Allow us to focus on Step 2 of the process. How can we efficiently solve A and B? Since direct integration of the entries may not be possible in certain cases, we shall create an algorithm that uses numerical quadrature techniques that we explored above.

We shall focus primarily on the model problem as our reference. We shall use the algorithm explored in [2], where we assemble using Gaussian Quadrature and compute all entries of A and B. Below is the algorithm for locally assembling the stiffness matrix and load vector.

Algorithm 1 Calculate Local Stiffness Matrix A_e and Load Vector B_e

for Element K_s with $s = 0, ..., n$ **do**
  $dx = (x_{s+1} - x_s)/2$
  for Quad. Points l with $l = 0, ..., n_q$ **do**
    for DoF's i with $i = 1, 2$ **do**
      $B_e(i) \mathrel{+}= f(\xi_l) \cdot \psi_i(\xi_l) \cdot \omega(l) \cdot dx$
      for DoF's j with $j = 1, 2$ **do**
        $A_e(i,j) \mathrel{+}= \left(\psi_i'(\xi_l) \cdot \psi_j'(\xi_l) \cdot \omega(l)\right) / dx$
      end for
    end for
  end for
end for
return A_e, B_e

Once we have obtained A_e and B_e , we can simply assemble our global matrices A and B by inputting the local values per element.

3.1.5 Example: 1D Poisson with Dirichlet Boundary Conditions

Below we demonstrate the implementation of the Finite Element Method for the following 1D problem.

$$\text{Find u such that:} \quad -u'' = 12x^2 - 36x + 18, \quad u(0) = u(3) = 0 \tag{3.5}$$

One can use regular ODE methods to show that the analytical solution is $u(x) = (x-3)^2x^2$, but we shall use the finite element method to approximate the true solution. The MATLAB program we use to solve this problem is FEM1DOfficial.m (See Appendix 1 for MATLAB code)

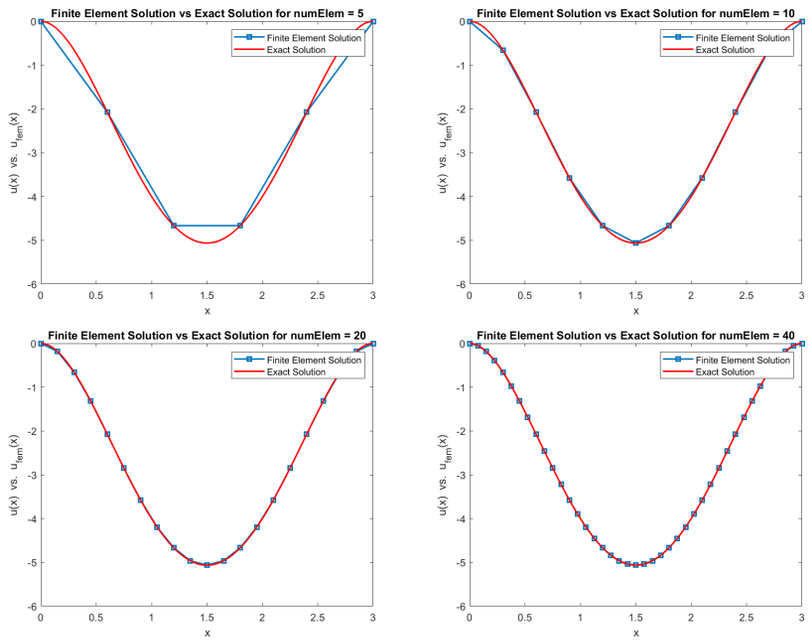

Figure 3.1: Here we can see the FEM solution $u_{fem}(x)$ approximating the analytical solution $u(x)$. As $numElem$ increases, we attain a better approximation of the true solution.

Note: In our results, we let u_{fem} be the FEM solution, instead of using u_h . As for the analytical solution, for some examples we write u_{exact} instead of u.

3.2 Solving the 2D Model Problem on MATLAB

When implementing the Finite Element Method for the 2D Poisson equation, the steps are very similar to the 1D case. We shall explain below in detail the steps that are slightly different. We shall reference from [2] and [4] in this section.

3.2.1 Implementing the Triangulation

We describe how MATLAB will implement the triangulation.

1. We define the nodal points of the triangulation, and we assume we have a total of $nnode$ nodal points: $(x,y_1), (x_2, y_2), ..., (x_{nnode}, y_{nnode})$
2. We define the elements of the triangulation $K_1, K_2, ..., K_{nelem}$, where we assume there are $nelem$ in total. We then use a $3 \times (nelem)$ matrix $nodes$ to describe the relationship between each element and their nodal points.

3.2.2 Stiffness Matrix and Assembly

To give an idea as to how the stiffness matrix will be assembled, we first start out with getting a visual of how it will look. For this case, we will take the 2D Poisson Equation as our model problem: Find u such that:

$$-\nabla u = f(x,y), \quad \Omega = (0,a)\times(0,b), \qquad u = 0 \ \text{ on } \ \partial\Omega. \tag{3.6}$$

To visualise how this will look, let us give a straightforward example of a uniform mesh $\Omega = (0,0)\times(a,b)$. In this case, we let $a = b$ and we obtain a square domain.

To obtain a uniform triangulation, we can use **row wise natural ordering for the nodal points.** In other words, for nodal coordinates (x_i, y_j) in the domain

$$x_i = ih\,, \quad y_j = jh\,, \quad h = \frac{1}{n}\,, \quad i = 1,2,...,m-1\,, \ \ j = 1,2,...,n-1$$

In the figure below, we can see our domain will be discretised by our triangulation. On the right hand side, we can see that we will store each triangular element in *increasing order* starting from the bottom left (by their nodal points on MATLAB).

When we evaluate each nodal point i using the global basis functions, in our stiffness matrix the evaluation for node i will be a **linear combination of the values of** u_h at the 5 nodes circled in red (on the LHS).

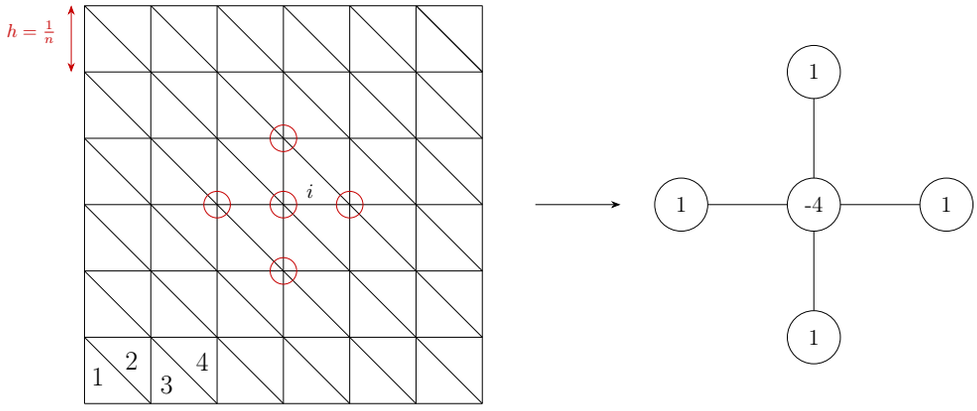

Figure 3.2: LHS: Our uniform triangulation on a square domain. RHS: For each node i, in our stiffness matrix the evaluation for node i will be a linear combination of the values of u_h at the 5 nodes circled in red on the LHS

This will result in our stiffness matrix A being *block tridiagonal*. Our matrix equation $AU = b$ will look like the following:

$$\begin{bmatrix} B & -I & 0 & \cdots & & 0 \\ -I & B & -I & & & 0 \\ 0 & -I & \cdots & & & \vdots \\ \vdots & & & \cdots & & \\ & & & & B & -I \\ 0 & \cdots & & & -I & B \end{bmatrix} \begin{bmatrix} u_{h,1} \\ u_{h,2} \\ \\ \vdots \\ \\ u_{h,n} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \\ \vdots \\ \\ b_n \end{bmatrix}, \quad \text{where } B = \begin{bmatrix} 4 & -1 & 0 & & & \\ -1 & 4 & -1 & & & \\ 0 & & \cdots & \cdots & & \\ & & & \cdots & \cdots & \\ & & & -1 & 4 & -1 \\ & & & & -1 & 4 \end{bmatrix}$$

As for the local and global assembly of A and b, the procedure for assembly is very similar to the 1D case (3.1.2), except we will have 3×3 element stiffness matrices.

$$K_i = \begin{bmatrix} \int_{K_i} \nabla\phi_i \cdot \nabla\phi_i \, dx & \int_{K_i} \nabla\phi_i \cdot \nabla\phi_j \, dx & \int_{K_i} \nabla\phi_i \cdot \nabla\phi_k \, dx \\ \int_{K_i} \nabla\phi_j \cdot \nabla\phi_i \, dx & \int_{K_i} \nabla\phi_j \cdot \nabla\phi_j \, dx & \int_{K_i} \nabla\phi_j \cdot \nabla\phi_k \, dx \\ \int_{K_i} \nabla\phi_k \cdot \nabla\phi_i \, dx & \int_{K_i} \nabla\phi_k \cdot \nabla\phi_j \, dx & \int_{K_i} \nabla\phi_k \cdot \nabla\phi_k \, dx \end{bmatrix}$$

Letting N_i, N_j and N_k be the vertices of triangle K, we note that $\int_K \nabla\phi_i \nabla\phi_j = a_K(\phi_i, \phi_j) = 0$ except in the case that the vertices of K are N_i and N_j . We can then use the formula in (2.11) to represent the triangle K_i and compute the local elements.

Similarly speaking we will compute the right hand side b by calculating the local 3×1 load vectors. These can then be assembled into the global stiffness matrix A and load vector b.

3.2.3 Implementation of Master Element and Quadrature Formulas

We begin by defining how we can transform from a general triangle to a right-angled master triangle for the 2D implementation. Our master triangle (or element) will be a **right angled triangle** with vertices $(0,0), (1,0)$ and $(0,1)$ on a 2D plane with coordinates (ξ, η). We define the three basis functions we shall use on this plane,

$$\varphi_1(\xi,\eta) = 1 - \xi - \eta \,, \quad \varphi_2(\xi,\eta) = \xi \,, \quad \varphi_3(\xi,\eta) = \eta \,. \tag{3.7}$$

We then define the transformation from a triangle with vertices $(x_1, y_1), (x_2, y_2)$ and (x_3, y_3) to the master triangle:

$$x = \sum_{j=1}^{3} x_j \varphi_j(\xi,\eta), \quad y = \sum_{j=1}^{3} y_j \varphi_j(\xi,\eta) \tag{3.8}$$

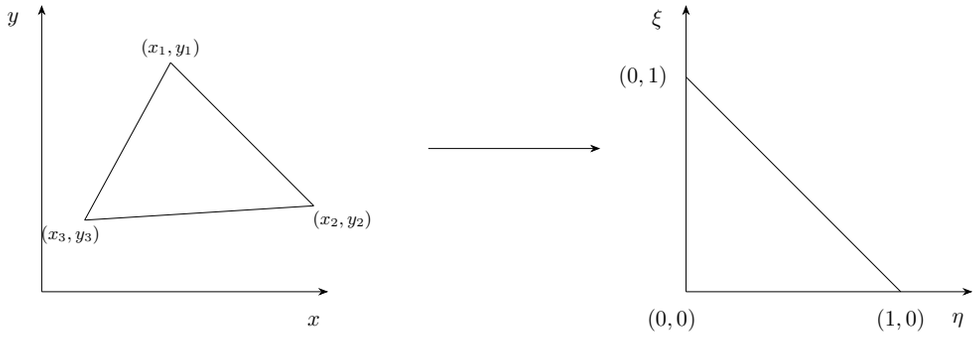

Figure 3.3: A 2D visual of the transformation to the reference triangle

Our next step is to determine how the stiffness matrix A and load vector B is calculated. Our integrals are the following:

$$\int\int_{\Omega} f(x,y)\phi_j(x,y)dxdy = \int\int_{\triangle} f(\xi,\eta)\,\varphi_j\,|J|\,d\xi d\eta \,, \tag{3.9}$$

$$\int\int_{\Omega} \nabla\phi_i(x,y)\nabla\phi_j(x,y)dxdy = \int\int_{\triangle} \left(\frac{\partial\varphi_i}{\partial x}\frac{\partial\varphi_j}{\partial x} + \frac{\partial\varphi_i}{\partial y}\frac{\partial\varphi_j}{\partial y} \right) |J|\,d\xi d\eta \tag{3.10}$$

Our first priority will be calculating the Jacobian:

$$J = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

Using (3.8), we can deduce that

$$\frac{\partial x}{\partial \xi} = x_2 - x_1, \quad \frac{\partial y}{\partial \xi} = y_2 - y_1,$$
$$\frac{\partial x}{\partial \eta} = x_3 - x_1, \quad \frac{\partial y}{\partial \eta} = y_3 - y_1.$$

Since the stiffness matrix involved partial derivatives of x and y, we need to use the chain rule to differentiate (x, y) with respect to (ξ, η).

$$\frac{\partial \varphi_i}{\partial x} = \frac{\partial \varphi_i}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \varphi_i}{\partial \eta}\frac{\partial \eta}{\partial x},$$
$$\frac{\partial \varphi_i}{\partial y} = \frac{\partial \varphi_i}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial \varphi_i}{\partial \eta}\frac{\partial \eta}{\partial y}.$$

We can rearrange this into a linear system of equations:

$$\begin{bmatrix} \frac{\partial \varphi_i}{\partial x} \\ \frac{\partial \varphi_i}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x} \\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial \varphi_i}{\partial \xi} \\ \frac{\partial \varphi_i}{\partial y} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial \varphi_i}{\partial \xi} \\ \frac{\partial \varphi_i}{\partial y}. \end{bmatrix}$$

Lastly, $\frac{\partial \varphi_i}{\partial \xi}$ and $\frac{\partial \varphi_i}{\partial \xi}$ can be easily calculated from (3.7). It also important to note the relationship between the area of a triangle and the determinant of the Jacobian.

$$det(J) = 2 \times (\text{Area of Triangle})$$

Lastly, we define the quadrature formula we shall implement for the equations (3.9) and (3.10):

$$\int\int_{\triangle} f(\xi, \eta) \cdot \varphi_j \cdot |J| \cdot d\xi d\eta \approx \sum_{k=0}^{n_q} \omega_k \, f(\xi_k, \eta_k) \cdot \varphi_j(\xi_k, \eta_k) \cdot |J|$$

$$\int\int_{\triangle} \left(\frac{\partial \varphi_i}{\partial x}\frac{\partial \varphi_j}{\partial x} + \frac{\partial \varphi_i}{\partial y}\frac{\partial \varphi_j}{\partial y} \right) \cdot |J| \cdot d\xi d\eta \approx \sum_{k=0}^{n_q} \omega_k \left(\frac{\partial \varphi_i(\xi_k, \eta_k)}{\partial x}\frac{\partial \varphi_j(\xi_k, \eta_k)}{\partial x} + \frac{\partial \varphi_i(\xi_k, \eta_k)}{\partial y}\frac{\partial \varphi_j(\xi_k, \eta_k)}{\partial y} \right) \cdot |J|$$

With regards to quadrature points, we decide to go with the four point implementation.

$$1.\ (\xi_1, \eta_1) = \left(\frac{1}{3}, \frac{1}{3}\right), \quad w_1 = -\frac{27}{96} \qquad 2.\ (\xi_2, \eta_2) = \left(\frac{2}{15}, \frac{11}{15}\right), \quad w_2 = \frac{25}{96}$$
$$3.\ (\xi_3, \eta_3) = \left(\frac{2}{15}, \frac{2}{15}\right), \quad w_3 = \frac{25}{96} \qquad 4.\ (\xi_4, \eta_4) = \left(\frac{11}{15}, \frac{2}{15}\right), \quad w_4 = \frac{25}{96}$$

3.3 Main Algorithm for solving the Model Problem in 2D

We shall focus on the local assembly of the stiffness matrix and load vector in the algorithm below:

Algorithm 2 Calculate Local Stiffness Matrix A_e and Load Vector B_e

for Element K_s with $s = 0, ..., n$ **do**
  for Quad. Points l with $l = 0, ..., n_q$ **do**
    for DoF's i with $i = 1, 2, 3$ **do**
      $B_e(i) += f(\xi_l, \eta_l) \cdot \psi_i(\xi_l, \eta_l) \cdot \omega(l) \cdot |J|$
      for DoF's j with $j = 1, 2, 3$ **do**
        $A_e(i, j) += \omega_l \left(\partial_x \varphi_i(\xi_k, \eta_k) \partial_x \varphi_j(\xi_k, \eta_k) + \partial_y \varphi_i(\xi_k, \eta_k) \partial_y \varphi_j(\xi_k, \eta_k)\right) \cdot |J|$
      end for
    end for
  end for
end for
return A_e, B_e

3.4 Example Case: FEM for 2D Poisson on MATLAB

We define the following 2D Poisson Equation

$$\text{Find u such that } -\nabla u = f = 2\pi^2 sin(\pi x) sin(\pi y) \text{ on } \Omega = [0, 1]^2$$

Like with the 1D case, we can use standard PDE methods to obtain the analytical solution $u = sin(\pi x) sin(\pi y)$, but here we shall use the methods described above to approximate the true solution. The MATLAB program we use to solve this problem is FEM2DOfficial.m (See Appendix 1 for MATLAB code)

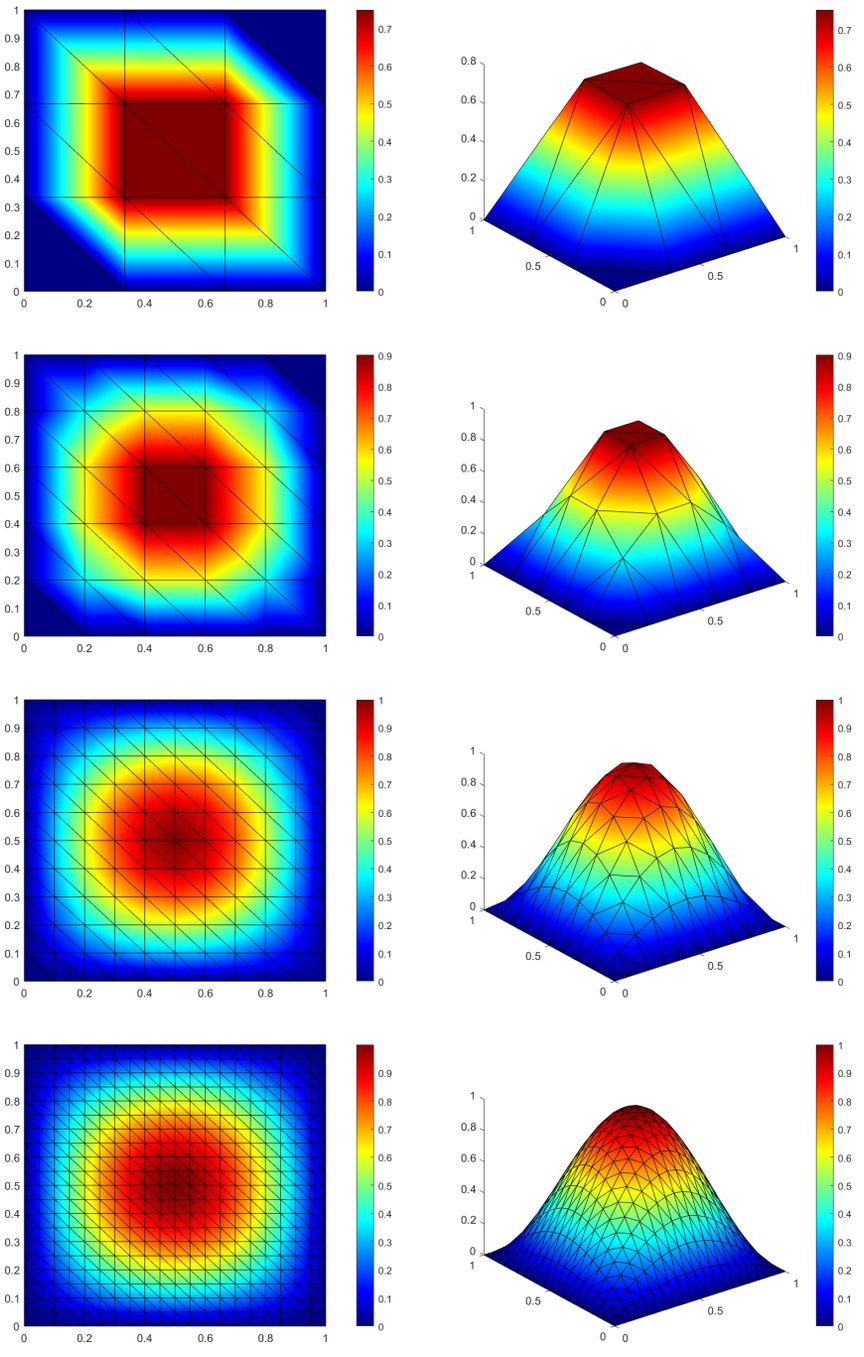

Figure 3.4: Here we can see the Finite Element Method approximating the solution. As our value of $h = 1/(numElem)$ decreases, we obtain a more accurate approximation of analytical solution.

4 | Theory of Finite Elements and Error Analysis

In this section we address the following objectives:

- Examine why the Finite Element Method is a viable option for obtaining numerical solutions to differential equations from a mathematical perspective.
- Describe FEM error results for 1D and 2D differential equations and the influence of numerical quadrature on these results.
- Demonstrate results of important error inequalities for the 1D Poisson Equation in MATLAB

4.1 Mathematical Theory of Finite Elements

Thus far we have loosely introduced many of the key components of the Finite Element Method, such as weak derivatives, Sobolev Spaces, bilinear forms, etc. Our intention in this section is to **rigorously define** these topics so that we can introduce a theorem that gives the Finite Element Method its validity as an appropriate numerical method: the Lax-Milgram Theorem.

Here we shall reference from [6] and [7] to help us.

4.1.1 Weak Derivatives

We start with the following formal definition of a weak derivative:

Definition 4.1.1. *Let $\Omega \in \mathbb{R}^n$ and $f \in L^1_{loc}$, where*
$L^1_{loc} := \{f \ : \ f \in L^1(K) \ \forall \text{ compact } K \ \subset \ \text{interior } \Omega\}$. *We say that a given function*
$f \in L^1_{loc}(\Omega)$ *has a* ***weak derivative****, D^α , provided there exists a function $g \in L^1_{loc}(\Omega)$ such that*

$$\int_\Omega g(x)\varphi(x)\,dx = (-1)^{|\alpha|}\int_\Omega f(x)\varphi^{(\alpha)}(x)\,dx \quad \forall \varphi \in \mathcal{D}(\Omega)\,.$$

If such a g exists, we let $D^\alpha f = g$.

We provide some details of the notation below:

- $\alpha = (\alpha_1, ..., \alpha_n)$ is our multi-index where $|\alpha| = \sum_{i=1}^n \alpha_i$
- We usually denote $\mathcal{D}(\Omega) := C_0^\infty$, which is the set of C^∞ functions with compact support in Ω.
- $D^\alpha\varphi$ is defined as our partial derivative for $\varphi \in C^\infty$, with

$$D^\alpha\varphi = \left(\frac{\partial}{\partial x_1}\right)^{\alpha_1}\left(\frac{\partial}{\partial x_2}\right)^{\alpha_2}\cdots\left(\frac{\partial}{\partial x_n}\right)^{\alpha_n}\varphi.$$

4.1.2 Sobolev Spaces and Norms

Definition 4.1.2. *Let $k \in \mathbb{N}_0$, $f \in L^1_{loc}(\Omega)$. Suppose the weak derivatives $D^\alpha f$ exists $\forall |\alpha| \leq k$. For $1 \leq p \leq \infty$, we define the* ***Sobolev Norm*** *as follows:*

$$||f||_{W^p_k(\Omega)} := \left(\sum_{|\alpha| \leq k} ||D^\alpha f||^p_{L^p(\Omega)} \right)^{\frac{1}{p}} .$$

From there, we define the ***Sobolev Space:***

$$W^p_k(\Omega) := \left\{ f \in L^1_{loc}(\Omega) \ : \ ||f||_{W^k_p(\Omega)} < \infty \right\} .$$

Below we provide a brief definition of the **Hilbert Space**

Definition 4.1.3. *Let $(V, (\cdot, \cdot))$ be an inner-product space. If the associated normed linear space $(V, ||\cdot||)$ (such that $||v|| := \sqrt{(v, v)}$) is complete, then $(V, (\cdot, \cdot))$ is called a* ***Hilbert space.***

We have that Sobolev spaces with $p = 2$ are an example of a Hilbert space and are denoted by

$$H^k(\Omega) = W^k_2(\Omega)$$

as described earlier in Section 2.

We also need to define the **space of bounded linear functionals** on H, which we will denote as H' . This is known as the **dual** of a Hilbert space, and the dual norm is defined as

$$||L||_{H'} := \sup_{0 \neq v \in H} \frac{L(v)}{||v||_J}$$

4.1.3 Bilinear Forms

So far we have been working with the bilinear form $a(u, v) := \int_\Omega \nabla u \cdot \nabla v \, dx$ for the 1D and 2D Poisson Equations, and now we need to introduce definitions for the bilinear form to allow us to progress to the Lax-Milgram Theorem.

Definition 4.1.4. *Let H be a Hilbert space with norm $||\cdot||$ and let $V \subset H$. Then a bilinear form $a : H \times H \to \mathbb{R}$ is said to be* ***continuous*** *if $\exists\, C < \infty$ such that*

$$|a(u, v)| \leq C||u||_H||v||_H \quad \forall u, v \in H$$

and ***coercive*** *on $V \subset H$ if $\exists\, \alpha > 0$ such that*

$$a(u, v) \geq \alpha||v||^2_H \quad \forall v \in V .$$

4.1.4 Lax-Milgram Theorem

Let us look back at the original variational problem we were trying to solve when we began;

$$\text{Given } F \in V' \text{, find } u \in V \text{ such that } a(u, v) = F(v) \ \ \forall v \in V. \tag{4.1}$$

We introduced the approximation problem to help introduce the Finite Element Method

$$\text{Given } V_h \subset V \text{ and } F \in V' \text{, find } u_h \in V_h \text{ such that } a(u_h, v) = F(v) \ \ \forall v \in V_h . \tag{4.2}$$

We can now state the fundamental theorem which proves the **existence and uniqueness** of the solution to the variational problem, and consequently the approximation problem aswell.

Theorem 4.1.5 (Lax-Milgram)**.** *Let $(V, (\cdot,\cdot))$ be a Hilbert space, let $a(\cdot,\cdot)$ be a continuous and coercive bilinear form and let $F(\cdot)$ be a continuous linear functional on V'. Then there exists a unique $u \in V$ such that*

$$a(u, v) = F(v) \quad \forall v \in V\,.$$

Using the theorem above, we can conclude that the variational problem (4.1) has a unique solution.

Moreover, since V_h is a closed subspace of V (and thus also a Hilbert Space), this means the variational problem (4.2) also has a unique solution.

4.2 Error Analysis of the Finite Element Method

In this subsection we will explore the accuracy of the FEM solution by analysing the discretisation error $e_h = u - u_h$.

4.2.1 Galerkin Orthogonality and Best Approximation Result

One question that may arise when researching our numerical method is the following: is the FEM is the best approximation to the true solution? To explore this, let us first introduce Galerkin Orthogonality (while taking reference from [3]).

Note: Without loss of generality, we can assume the theorems below are applied to the FEM in 2D, unless stated otherwise. The theorems below still hold in for the FEM in 1D.

Theorem 4.2.1 (Galerkin Orthogonality)**.** *The finite element approximation u_h, defined by (2.5), satisfies the following equality:*

$$a(u - u_h, \phi_h) = 0, \quad \forall \phi_h \in V_h\,.$$

Proof: We have the following variational formulation from our previous workings:

$$a(u, v) = (f, v) \quad \forall v \in V.$$

We then have the discretised version of this formulation:

$$a(u_h, v_h) = (f, v_h) \quad \forall v_h \in V_h\,.$$

It's important to observe that $V_h \subset V$. Using this fact we perform the following:

$$(f, v_h) - (f, v_h) = a(u, v_h) - a(u_h, v_h) = 0 \quad \forall v_h \in V_h\,.$$

The result immediately follows due to the properties of bilinear forms: □

Theorem 4.2.2. *The finite element solution u_h to (2.8) satisfies the best approximation result:*

$$||\nabla(u - u_h)||_{L^2(\Omega)} \leq ||\nabla(u - v)||_{L^2(\Omega)} \quad \forall v \in V_h\,.$$

Proof: Let $v \in V_h$. We do some equation manipulation:

$$||\nabla(u - u_h)||^2_{L^2(\Omega)} = \int_\Omega \nabla(u - u_h) \cdot \nabla(u - v + v - u_h)\, dx$$

$$= \int_\Omega \nabla(u - u_h) \cdot \nabla(u - v)\, dx + \int_\Omega \nabla(u - u_h) \cdot \nabla(v - u_h)\, dx\,.$$

We can then use Galerkin Orthogonality:

$$\int_\Omega \nabla(u - u_h) \cdot \nabla(v - u_h)\, dx = 0$$

since $v - u_h \in V_h$. We can then finish the equation

$$\begin{aligned} &\int_\Omega \nabla(u - u_h) \cdot \nabla(u - v)\, dx \\ &\leq ||\nabla(u - u_h)||_{L^2(\Omega)} ||\nabla(u - v)||_{L^2(\Omega)}\,. \end{aligned}$$

Our results follows immediately. □

To give a visual of the results (4.2.1) and (4.2.2), see the graph below for a geometric interpretation of the FEM.

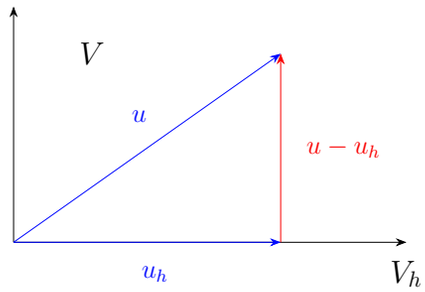

Figure 4.1: A visual of how Galerkin Orthogonality looks in the space V

Since the error stands orthogonal for all $\phi_h \in V_h$, the solution u_h has the smallest distance to u , which verifies that u_h is the best approximation to u.

4.2.2 Error Estimates for the Finite Element Method

Below we shall explore the order of convergence of the error estimate, which is a useful measure for how well our numerical method approximates a given differential equation. The following error estimates also hold in 2D, which is why we shall only explore the 1D case.

We will now briefly introduce **interpolation functions,** which are useful when deriving error estimates.

Definition 4.2.3. *We define our interval $I = [x_0, x_1]$, and we have our have our basis functions ϕ_i as defined form the 1D case in section 2 such that $V_h = \{\phi_1, \phi_2, ..., \phi_{n+1}\}$. Then the linear 1D piecewise interpolation function $u_I : H^1 \to V_h$ is defined as follows*

$$u_I(x) = \sum_{j=0}^{n+1} u(x_j)\phi_j(x)$$

or more simply

$$u_I(x) = u(x_{i-1})\frac{x - x_i}{x_{i-1} - x_i} + u(x_i)\frac{x - x_{i-1}}{x_i - x_{i-1}}, \quad x_{i-1} \leq x \leq x_i$$

If we use our best approximation result that we obtained earlier, we obtain an *upper bound for the error of interpolation functions.*

$$||u - u_h||_a \leq ||u - u_I||_a\,.$$

We have two important error estimates for our 1D case (taken from [2]):

Theorem 4.2.4. *Let $u \in H_0^1$ and $u_h \in V_h$ be the solutions of the continuous and discrete Poisson problems. Then*

$$\lim_{h \to 0} ||u - u_h||_{H^1} = 0$$

Moreover, if $u \in H^2$ (if $f \in L_2$ or 'twicely differentiable') we have

$$||u - u_h||_{H^1} \leq Ch||u''||_{L_2} = Ch||f||_{L_2} = O(h) \ .$$

We thus have linear convergence *of the FEM for the H^1 norm.*

Theorem 4.2.5. *Let $u \in H_0^1$ and $u_h \in V_h$ be the solutions of the continuous/discrete Poisson problems. Then*

$$||u - u_h||_{L_2} \leq C_1 h^2 ||u''||_{L_2} = C_1 h^2 ||f||_{L_2} = O(h^2) \ .$$

We thus have quadratic convergence *of the FEM for the L_2 norm.*

Lastly, we investigate the infinity norm error [4]. Note that the infinity norm is simply

$$||u - u_h||_\infty := \max_i |(u - u_h)_i|$$

Theorem 4.2.6. *Let $u \in H_0^1$ and $u_h \in V_h$ be the solutions of the continuous/discrete Poisson problems. Then*

$$||u - u_h||_\infty \leq C_2 h^2 ||u''||_\infty = C_2 h^2 ||f||_\infty = O(h^2) \ .$$

We thus have quadratic convergence *of the FEM for the $\infty-$norm.*

These three inequalities are important as they give us an idea as to how fast our error converges, which we see below with our graph and table below (see Table 4.1).

4.2.3 Error Estimation for Numerical Quadrature

In Section 3, when calculating the stiffness matrix and load vector, we chose to use numerical quadrature (Gaussian quadrature) to reduce the computational cost. What this gives us is **an approximation $\tilde{u}_h$ of the finite element solution** u_h , which in turn creates another error. Below we describe the new error inequality as follows (for the $H^1(\Omega)$ norm):

$$||u - \tilde{u}_h||_{H^1(\Omega)} \leq ||u - u_h||_{H^1(\Omega)} + ||u_h - \tilde{u}_h||_{H^1(\Omega)} \ . \tag{4.3}$$

On the RHS the first error is known as the **error of approximation** and the second error is known as the **error of consistence.**

To help us find the error of consistence, we shall introduce a lemma that allows us to derive an error estimate [2].

Lemma 4.2.7 (Strang's First Lemma)**.** *Let the approximate bilinear form $\tilde{a}$ and linear functional $\tilde{l}$ satisfy the condition*

$$||v_h||_{H^1}^2 \leq \gamma \tilde{a}(v_h)$$

uniformly in $0 < h < h_0$ and also holds the estimate

$$|(a - \tilde{a})(u_h, v_h)| + |(f, v_h) - \tilde{l}(v_h)| \leq ch^\tau \cdot ||v_h||_{H^1}, \ v_h \in V_h = S_m^h$$

where S_h^m is the space of finite elements of degree m. Then the quantitative error estimate for the approximation of $u_h \in V_h$ is given by

$$||u_h - \tilde{u}_h||_{H^1} \leq c\gamma h^\tau \ .$$

The next theorem below applies the lemma above to allow us to derive an error estimation of the FEM for numerical quadrature formulae.

Theorem 4.2.8. *Assume the quadrature formula, of order $r \geq 3$,*

$$Q_E(g) = \sum_{i=1}^{L} \hat{\omega}_i g(\hat{\xi}_i)$$

on E, which is admissible for $P_{m-1}(E)$, where $P_{m-1}(E)$ is the space of polynomials with degree less than or equal to $m-1$ on domain E. Let $r \geq m-1, m-2$. The degree of the finite elements is $m-1$ that is $u_h, v_h \in S_h^{m-1}$. The coefficient functions $a_{\nu\mu} \in L^\infty(E)$ are sufficiently smooth. Then the following inequalities hold:

$$||v_h||_{H^1}^2 \leq \gamma \tilde{a}(v_h) \,,$$

$$|(a - \tilde{a})(u_h, v_h)| \leq c h^{r-m+2} \cdot ||v_h||_{H^1}, \; v_h \in S_h^m \,,$$

$$|(f, v_h) - \tilde{l}(v_h)| \leq c h^{r-m+2} \cdot ||v_h||_{H^1}, \; v_h \in S_h^m \,.$$

By applying the Strang Lemma (4.2.7) above, we obtain an error estimate for the error of consistence, with $\tau = r - m + 2$.

$$||u_h - \tilde{u}_h||_{H^1} = O(h^{r-m+2}) \,.$$

In conclusion, we obtain the total error to be as follows:

$$||u - \tilde{u}_h||_{H^1} \leq ||u - u_h||_{H^1} + ||u_h - \tilde{u}_h||_{H^1} = O(h^{m-1} + h^{r-m+2}) \,.$$

In this case $m - 1 := m'$ denotes the degree of the finite elements and $r - m + 2 := r'$ denotes the degree of the quadrature rule.

In conclusion, if we want **optimal convergence** (the rate of convergence will not be worsened by the implementation of numerical quadrature) then we need $r' \geq m'$. This in turn allows us to neglect the effects of numerical quadrature on the FEM error.

For example, in the case that we let $r = m - 1$ we obtain the same rate of convergence as the error of approximation for the H^1 norm

$$||u - \tilde{u}_h||_{H^1} = O(h) \,.$$

4.2.4 Example: Computing Error Norms for MATLAB

When computing the error estimates on MATLAB, we know that our error will be $e_h = u - u_h$, so we can evaluate the error norms $||u - u_h||$. Since we must evaluate the error norm **by element,** we use the element wise error norm (taken from [2]). Given a triangulation $\mathcal{T} = \{K_1, K_2, ..., K_n\}$, we have the following:

For L_2:

$$||u - u_h||_{L_2(\Omega)} = \sqrt{\int_\Omega (u - u_h)^2 dx} = \sqrt{\sum_{K_j} \int_{K_j} (u - u_h)^2 dx} \tag{4.4}$$

For H^1:

$$||u-u_h||_{H^1(\Omega)} = \sqrt{\int_\Omega ((u-u_h)^2 + (\nabla u - \nabla u_h)^2)\,dx} = \sqrt{\sum_{K_j} \int_{K_j} ((u-u_h)^2 + (\nabla u - \nabla u_h)^2)\,dx} \tag{4.5}$$

4.2.5 Example: Error Analysis for 1D Poisson Example

Earlier we applied the FEM to the 1D problem (3.5) and now we decide to explore the nature of the errors of our approximation.

To apply (4.4) and (4.5), we choose to **interpolate over each element** and take the average of the interval for both the analytical solution and the FEM approximation and then insert our values into the formulas above. We use the MATLAB program FEM1DOfficial.m to get our results.

Below is a table and a graph showing our findings for the error obtained.

Table 4.1: The FEM Solution accuracy for different values of h

numElem	**h**	**L2Error**	**H1Error**
5	0.600000	5.382548e-01	1.505992e+00
10	0.300000	1.445885e-01	7.089334e-01
20	0.150000	3.676572e-02	3.458153e-01
40	0.075000	9.229978e-03	1.716111e-01
80	0.037500	2.309902e-03	8.563178e-02
160	0.018750	5.776260e-04	4.279351e-02
320	0.009375	1.444159e-04	2.139392e-02
640	0.004687	3.610456e-05	1.069660e-02

And then we have the following graph to visualise the results.

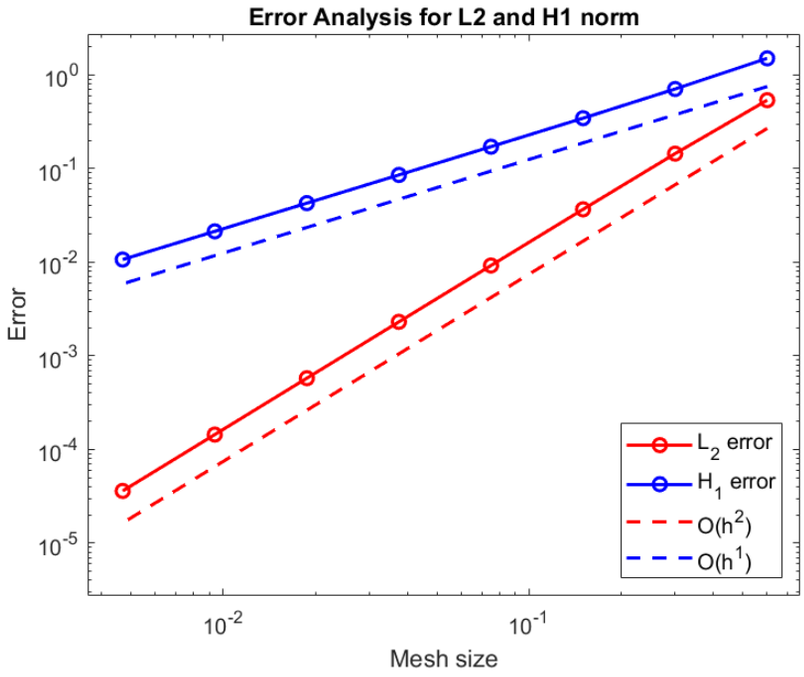

Figure 4.2: Here we can see the convergence rate for each error norm, with L_2 having a faster convergence rate compared to H^1

4.2.6 Example: Error Analysis for 2D Poisson Example

Earlier we applied the FEM to the 2D problem (3.4). We shall now explore the errors in our numerical solution.

We shall derive results for the infinity norm error for different levels of discretisation and use the MATLAB program FEM2DOfficial.m to get our results.

Below is a table and a graph showing our findings for the error obtained. We also calculate the **rate of convergence** [7], which is simply

$$r = log_2 \left| \frac{u - u_h}{u - u_{h/2}} \right|$$

Table 4.2: The FEM error for different values of numElem

numElem (on each axis)	**InfNormError**	**Rate of Convergence**
4	5.842106e-03	-
8	2.086806e-03	1.485192
16	5.763899e-04	1.856179
32	1.471472e-04	1.969785
64	3.702571e-05	1.990661

And then we have the following graph to visualise the results.

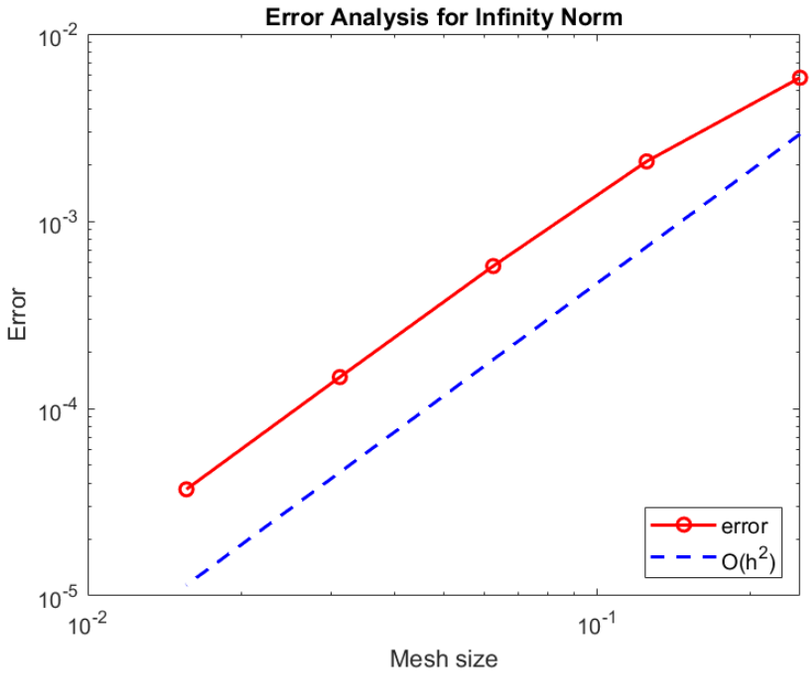

Figure 4.3: Here we can see the infinity norm error has quadratic convergence

5 | Application of FEM to Option Pricing

In this section we shall apply our knowledge of the Finite Element Method to an important topic in mathematical finance: Option Pricing. We shall complete the following objectives:

- Outline how the FEM can be applied to parabolic PDE's and how to discretise the time domain using a discrete time stepping scheme.
- Learn what options are and the important terminology related to the field.
- Derive the Black-Scholes Equation, an important PDE used to price options, and outline how to transform the BSE into a 1D heat equation.
- Provide the details as to how the FEM can be used to obtain a numerical solution to the B-S Equation and provide examples computed on MATLAB.

5.1 The Finite Element Method for Parabolic PDE's

So far in the course of this paper, we have gone into depth on the theory of FEM applied to elliptic PDE's (Poisson Equation in 1D and 2D). In the real world however,, many problems often have a time dependent constraint; these are called **parabolic PDE's**.

In order to apply the Finite Element method to these types of PDE's, our approach needs to slightly different.

We start by considering the following model problem for a 1 dimensional linear parabolic problem from [1]. This will be the heat equation on a 2-dimensional object with spatial domain Ω.

$$\dot{u} - \Delta u = f \quad \text{in } \Omega \times I, \tag{5.1}$$

$$u = 0 \quad \text{on } \partial\Omega \times I, \tag{5.2}$$

$$u(\cdot, 0) = u^0(x)\,. \tag{5.3}$$

where:

- $u(x,t)$: Temperature at $x \in \Omega \subset \mathbb{R}^2$ at time $t \in I = (0,T)$
- u^0 : Initial temperature
- f : Given heat production
- $\dot{u} = \partial u / \partial t$

To apply the FEM to this equation, we must consider a *semi-discrete* analogue of (5.1) where we *discretise in* ***space*** using the FEM. Afterwards we can discretise time to obtain a fully discrete problem.

5.1.1 Space Discretisation

We take our model problem (5.1) and we shall modify it in a similar way to our original model Poisson problem. We let $V = H_0^1$ be our function space, introduce test function v to the equation and perform integration over our domain (via. Green's formula). We now have the following formulation of the model problem.

Find $u(t) \in V$ such that:

$$\int_I \dot{u}(t)v + \int_I \nabla u(t) \cdot \nabla v = \int_I f(t)v$$

...or in shorthand notation,

$$(\dot{u}(t), v) + a(u(t), v) = (f(t), v) \quad \forall v \in V, t \in I, \tag{5.4}$$

$$u(0) = u^0. \tag{5.5}$$

We can now attain the semi-discrete analogue of 5.4 by replacing V with V_h .

Find $u_h \in V_h$, $t \in I$, such that:

$$(\dot{u_h}(t), v) + a(u_h(t), v) = (f(t), v) \quad \forall v \in V_h, t \in I, \tag{5.6}$$

$$(u_h(0), v) = (u^0, v) \quad \forall v \in V_h\,. \tag{5.7}$$

We must find an appropriate representation for $u_h(t, x)$. A common approach is to use a combination of shape functions that individually represent the time and spatial dimensions. Here we can use the following representation for $u_h(t, x)$:

$$u_h(t, x) = \sum_{i=1}^{M} \xi_i(t)\phi_i(x) \quad t \in I \tag{5.8}$$

with time dependent coefficients $\xi_i(t) \in \mathbb{R}$.

We can now use (5.8) and let $v = \phi_j,\ j = 1, ..., M$ in (5.6) and we get the following:

$$\int_\Omega \left(\sum_{i=1}^{M} \dot{\xi}_i(t)\phi_i(x) \right) \phi_j(x) + \int_\Omega \left(\sum_{i=1}^{M} \xi_i(t)\nabla\phi_i(x) \right) \cdot \nabla\phi_j(x) = \int_I f(t)\phi_j(x)$$

which can be simplified down to:

$$\sum_{i=1}^{M} \dot{\xi}_i(t)(\phi_i, \phi_j) + \sum_{i=1}^{M} \xi_i(t)a(\phi_i, \phi_j) = (f(t), \phi_j) \quad j = 1, ..., M\,,$$

$$\sum_{i=1}^{M} \xi_i(0)(\phi_i, \phi_j) = (u^0, \phi_j) \quad j = 1, ..., M\,.$$

We can simplify by letting

- $B = b_{ij} = (\phi_i, \phi_j)$
- $A = a_{ij} = a(\phi_i, \phi_j)$

- $F = F_i = (f(t), \phi_i)$
- $U_i^0 = (u^0, \phi_i)$

where B is the *mass matrix* and A is the *stiffness matrix* (which we have seen before). We can now simply the equations down to the following:

$$B\dot{\xi}(t) + A\xi(t) = F(t), \quad t \in I, \tag{5.9}$$

$$B\xi(0) = U^0 \,. \tag{5.10}$$

This linear system of differential equations can be solved using similar methods carried out in MATLAB.

5.1.2 Time Discretisation

Now that we have a semi-discrete problem, the next step is to discretise the time variable. The most popular method is by using a finite difference method called the **Crank-Nicolson Method.** But first, let us briefly provide background for the method [3].

We start by discretising the time domain $0 = t_0 < t_1 < ... < t_M = T$ and the idea is to define the time steps to be $\Delta t = t_k - t_{k-1}\ \forall k = 1, 2, ..., M$. We then change the time derivative in (5.9) to

$$\dot{\xi}(t) \approx \frac{\xi_l - \xi_{l-1}}{\Delta t}$$

The idea is that given our initial condition $\xi(0)$, we can compute the solution $\xi(t+1)$ at the time $t^{k+1} = t^k + \Delta t$ for $k = 0, 1, 2, ...$

We then utilise the **Euler Methods,** of which there are two; the Backwards Euler method,

$$B\frac{\xi_l - \xi_{l-1}}{\Delta t} + A\xi_l = F_l \,,$$

and the Forward Euler method,

$$B\frac{\xi_l - \xi_{l-1}}{\Delta t} + A\xi_{l-1} = F_{l-1} \,.$$

The Crank-Nicolson method is a **combination** of the two above methods. The method is implemented as follows

$$B\frac{\xi_l - \xi_{l-1}}{\Delta t} + A\frac{\xi_l - \xi_{l-1}}{2} = \frac{F_l - F_{l-1}}{2} \,.$$

We can manipulate this equation to get

$$\left(B + \frac{\Delta t}{2}A\right)\xi_l = \left(B - \frac{\Delta t}{2}A\right)\xi_{l-1} - \frac{\Delta t}{2}\left(F_l + F_{l-1}\right) \,. \tag{5.11}$$

The reason we choose this method, and more specifically a Finite Difference scheme, is because of its desirable properties:

- C-N Method has 2^{nd} order accuracy, meaning the error of the numerical solution decreases at a rate proportional to $\left(\Delta t^2\right)$.
- C-N Method is unconditionally stable for (linear) heat equation (meaning the numerical solution will not grow without bound regardless of the choice of Δt).
- The C-N method is computationally efficient and relatively simple to implement.

One we have obtained (5.11) above, this equation can then be implemented on a coding language to obtain a solution for each ξ_l .

5.2 Introduction to Option Pricing

We are now at the point in the project where we shall apply the Finite Element Method to a real life application. This will be to option pricing, which is a very important topic in mathematical finance. But first, we have outlined below some background information on options.

5.2.1 Background Terminology

What is an option?

- Option - An option is a contract offered by a person that gives the buyer of this contract *the right, but not the obligation,* to buy/sell a particular underlying asset at a specified price (strike price) on or before a particular date (expiry date).

There are two different types of options: Call and Put Options

- Call Option - This is an option where the buyer has the "option" to *buy the asset.*
- Put Option - This is an option where the buyer has the "option" to *sell the asset.*

It is also important to mention that when the buyer of the option decides to buy/sell the underlying asset, they *exercise* the option.

There are many different types of options, but for this paper we will focus on European options. These are options that can *only* be exercised *on the expiry date.*

The important question that arises here is **how can we price this option?** We don't know *how the underlying asset's value is going to change* over time, and the price (and potential price) of the asset directly influences the price of the option.

This is where mathematical models for option pricing come in. **Continuous time stochastic models** help us derive an equation for the price of options. One famous example is the *Black-Scholes* model, developed by Fischer Black and Myron Scholes in the 60-70's (which was further developed by Robert C. Merton in the 70's).

5.3 How to Price Options: The Black-Scholes Model

First, we shall layout all the (relevant) factors that can affect the price of an option:

- S_0 - The current price of our underlying stock (asset) on the market.
- K - The strike price of the option (The price we pay if we exercise the option).
- T - The time until the expiration date.
- σ - The volatility of the price of the asset.
- r - The risk free interest rate.

We shall now present the Black-Scholes(-Merton) formula for estimating European call and put options. We shall refer to [8] and [9] for formulating the equation.

The Black-Scholes model assumes that a stock price follows a **Geometric Brownian motion**. The model is defined by the following equation:

$$\Delta S = \mu S \Delta t + \sigma S \Delta W \tag{5.12}$$

and the variables are defined below:

- S - Stock Price
- r - risk free interest rate
- σ - volatility of the stock
- μ - Expected rate of return of the stock per year
- W - Weiner Process, where $dW \sim \mathcal{N}(0, dt)$.

On the RHS the first term is used to describe the "drift" of the stock price (and is sometimes called the *deterministic* component of the stock), while the second term describes the "randomness" in the movement of the stock price.

We then apply Itô's Lemma to the equation (taken from [10]), which is:

Theorem 5.3.1 (Itô's Lemma)**.** *Assume that a process X has a stochastic differential given by*

$$dX(t) = \mu(t)dt + \sigma(t)dW(t)$$

where μ and σ represent adapted processes (in this case, they represent the drift and volatility respectively) and W is a Weiner process. Let $f(t, X(t))$ be a $C^{1,2}$ function. Then $Z = f(t, X(t))$ is also a stochastic process, and has a stochastic differential given by

$$df(t, X(t)) = \left(\frac{\partial f}{\partial t} + \mu\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial x^2}\right)dt + \sigma\frac{\partial f}{\partial x}dW(t)\,.$$

We can let $f(t, S(t))$ be our function for the price of our option. We then substitute into (5.12) and get the following

$$\Delta f = \left(\mu S\frac{\partial f}{\partial S} + \frac{\partial f}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 f}{\partial S^2}\right)\Delta t + \frac{\partial f}{\partial S}\sigma S\Delta W. \tag{5.13}$$

Our next step is to build our Black-Scholes differential equation. How do we do this?

What we notice above is that (5.12) and (5.13) both follow the same Itô process. This means **we can build a portfolio to eliminate this process** which will allow us to price an option.

In our portfolio, the action we undertake will be the process of *selling an option and buying $\frac{\partial f}{\partial S}$ shares of an underlying asset.* We define the portfolio Π below

$$\Pi = -f + \frac{\partial f}{\partial S}S$$

with the discrete version being

$$\Delta\Pi = -\Delta f + \frac{\partial f}{\partial S}\Delta S\,.$$

Substituting in our previous components for Δf and ΔS from (5.12) and (5.13) we get

$$\Delta\Pi = \left(\frac{\partial f}{\partial t} - \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta t\,.$$

Over the time period Δt we have eliminated ΔW , so the portfolio is *riskless* in this time period and takes a *riskless interest rate.* In other words, the portfolio will earn instantaneous rates of return over time period Δt. Therefore

$$\Delta\Pi = r\Pi\Delta t \implies \left(\frac{\partial f}{\partial t} - \frac{1}{2}\frac{\partial^2 f}{\partial S^2}\sigma^2 S^2\right)\Delta t = r\left(f - \frac{\partial f}{\partial S}S\right)\Delta t\,.$$

We can then finally rearrange to get

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf = 0 \tag{5.14}$$

which is our **Black-Scholes differential equation**. This PDE is solvable for some boundary conditions and unsolvable analytically for others.

5.4 Solving the Black-Scholes Equation for European Options

5.4.1 Transformation to a Parabolic PDE

How can we solve the PDE (5.14) above? Let us start by considering the boundary conditions of our PDE, which are

- $V(0,t) = 0$ for $0 \leq t \leq T$
- $V(S,T) = max(S-K, 0)$ for $0 \leq S \leq \infty$
- $V(S,t) \to S - Ke^{-r(T-t)}$ as $S \to \infty$.

The general approach is to apply a transformation of variables [11]

$$S = Ke^x, \quad \tau = \frac{\sigma^2}{2}(T-t), \quad k_1 = \frac{2r}{\sigma^2}, \quad V(S,t) = Kv(x,\tau)\,.$$

We can then formulate the following transformation (See Appendix 2 for detailed derivation of transformation):

$$v(x,\tau) = \frac{1}{K} exp\left(\left(\frac{1}{2}(k_1-1)x\right) + \left(\frac{1}{4}(k_1+1)^2\tau\right)\right) V(S,t)\,.$$

Using this transformation, we convert the original PDE (5.14) to a standard heat conduction equation, which is equivalent to the following Initial Value Problem with domain $x \in (-\infty, \infty)$ and $\tau \in [0, \infty)$

$$\frac{\partial v}{\partial \tau} = \frac{\partial^2 v}{\partial x^2}\,.$$

This is a 1-dimensional heat conduction equation and it can be interpreted as the distribution of temperature $v(x,\tau)$ for a beam at time τ.

The initial and boundary conditions are as follows:

Call Option:

$$v_c(x,0) = max\left\{e^{\frac{1}{2}(k_1+1)x} - e^{\frac{1}{2}(k_1-1)x},\ 0\right\} = \gamma_c(x)\,.$$

$$v_c(x_{-\infty},\tau) = 0$$

$$v_c(x_{+\infty},\tau) = e^{\frac{1}{2}(k_1+1)x_{+\infty} + \frac{1}{4}(k_1+1)^2\tau}$$

Put Option:

$$v_p(x,0) = max\left\{e^{\frac{1}{2}(k_1-1)x} - e^{\frac{1}{2}(k_1+1)x},\ 0\right\} = \gamma_p(x)\,.$$

$$v_p(x_{+\infty},\tau) = 0$$

$$v_p(x_{-\infty},\tau) = e^{\frac{1}{2}(k_1-1)x_{-\infty} + \frac{1}{4}(k_1-1)^2\tau}$$

In this case $x_{-\infty} = x_{min}$ and $x_{\infty} = x_{max}$.

5.4.2 Numerical Solution using FEM

Now we can solve our 1D transformed Black-Scholes equation numerically using the methods we have researched [12],[13]. In this case, we solve the PDE with respect to initial/boundary conditions for the *call option* (the steps for the *put option* are the same).

We start by deriving the weak form of our PDE

$$\int_\Omega \left(\frac{\partial v}{\partial \tau} - \frac{\partial^2 v}{\partial x^2} \right) N(x) dx = 0$$

which converts down to

$$\int_\Omega \frac{\partial v}{\partial \tau} N(x) dx - \int_\Omega \frac{\partial^2 v}{\partial x^2} N(x) dx = 0 \;\rightarrow\; (IBP) \;\rightarrow \int_\Omega \frac{\partial v}{\partial \tau} N(x) dx + \int_\Omega \frac{\partial v}{\partial \tau} \frac{\partial N(x)}{\partial \tau} dx = 0\,.$$

It is important to consider the **boundary conditions** of our model. We decide to incorporate an additional component that will satisfy the boundary conditions. We introduce the following equation that will approximate our solution

$$v(x,\tau) = \sum_{i=1}^{N} \xi_i(\tau)\phi_i(x) + \phi_0(x,\tau)\;.$$

In this case, $\phi_0(x,\tau)$ is implemented to **satisfy the boundary conditions,** and as a result is considered to be known. We establish $\phi_0(x,\tau)$ to be following:

$$\phi_0(x,t) = (\beta(\tau) - \alpha(\tau)) \left(\frac{x - x_{-\infty}}{x_\infty - x_{-\infty}} \right) + \alpha(\tau)$$

where $\alpha(\tau) = v_c(x_{-\infty},\tau)$ and $\beta(\tau) = v_c(x_{+\infty},\tau)$ as defined above. This can be also be seen as

$$\phi_0(x,\tau) = \begin{cases} \alpha(\tau) & \text{if} \quad x = x_{-\infty}, \\ \beta(\tau) & \text{if} \quad x = x_\infty\;. \end{cases}$$

So we can now integrate by parts,

$$\int_{x_0}^{x_m} \left[\sum_{i=1}^{m-1} \dot{\xi}_i \phi_i + \dot{\phi}_0 \right] \phi_j dx = \int_{x_0}^{x_m} \left[\sum_{i=1}^{m-1} \xi_i \phi_i'' + \phi_0'' \right] \phi_j dx$$

we can then reformulate to get the following:

$$B\xi_\tau + b = -A\xi - a\,.$$

where B is our mass matrix and A is our stiffness matrix, and

$$a(\tau) = \begin{bmatrix} \int \phi_0''(x,\tau)\phi_1(x)dx \\ \vdots \\ \int \phi_0''(x,\tau)\phi_{m-1}(x)dx \end{bmatrix}, \quad b(\tau) = \begin{bmatrix} \int \dot{\phi}_0(x,\tau)\phi_1(x)dx \\ \vdots \\ \int \dot{\phi}_0(x,\tau)\phi_{m-1}(x)dx \end{bmatrix}.$$

For simplicity, we can manually calculate A and B to get the following

$$A = \begin{bmatrix} \frac{2}{h} & \frac{-1}{h} & \cdots & & 0 \\ \frac{-1}{h} & \frac{2}{h} & & & \\ \vdots & & \ddots & & \\ & & & \frac{2}{h} & \frac{-1}{h} \\ 0 & & & \frac{-1}{h} & \frac{2}{h} \end{bmatrix}, \quad B = \begin{bmatrix} \frac{2h}{3} & \frac{h}{6} & \cdots & & 0 \\ \frac{h}{6} & \frac{2h}{3} & & & \\ \vdots & & \ddots & & \\ & & & \frac{2h}{3} & \frac{h}{6} \\ 0 & & & \frac{h}{6} & \frac{2h}{3} \end{bmatrix}.$$

where $h = x_i - x_{i-1}$. In the case of the chosen ϕ_0, we have that $\phi_0(x,\tau)'' = 0$ and therefore $a(\tau) = 0$. We also will establish the initial conditions for τ and at points $x = x_j$ by using the fact that $v_c(x,0) = \gamma(x)$,

$$\xi_j(0) = \gamma(x_j) - \phi_0(x_j, 0) \ .$$

Lastly, we perform time discretisation by using the Crank Nicholson scheme as we discussed earlier. By letting $\dot{\xi}_l = (\xi_l - \xi_{l-1})/(\Delta\tau)$ we get the following:

$$\left(B + \frac{\Delta\tau}{2}A\right)\xi_l = \left(B - \frac{\Delta\tau}{2}A\right)\xi_{l-1} - \frac{\Delta\tau}{2}\left(b_l + b_{l-1}\right)$$

5.4.3 Example 1: European Call Option FEM Approximation

We consider the **call option price of Amazon stock** with the following parameters:

- Time to Maturity $T = 0.5$ (6 months)
- Strike Price $K = 120\,\$$ (The price we pay for the stock if we choose to buy it)
- Volatility $\sigma = 35\%$ (variance)
- risk-free interest rate $r = 2\%$
- Min/Max price of stock $S_{min} = 20,\ S_{max} = 1500$.

For the implementation on MATLAB we choose $N = 100\,, M = 50$ for the number of spatial and temporal points, respectively. The MATLAB program we use to solve this problem is BSFEMCall.m (See Appendix 1 for MATLAB code). Below we graph the results of our solution.

Our first graph describes the numerical solution for the option price for different values of (S,t). We can observe that **the option value increases as the stock increases in value**

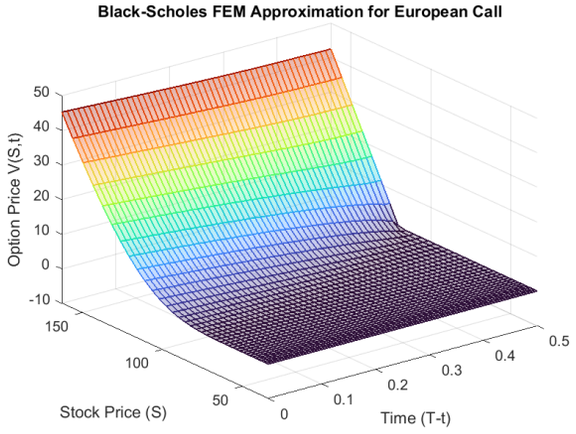

Figure 5.1: 3D plot of FEM solution for call option

We can see that as $t \to T$, our equation for ξ_l **"sharpens"** around the the strike price K. This is because as we are close to the expiry date, *there is less opportunity for the variance to drastically change the price of the option.* Below we have a 2D plot which helps us visualise how this "intrinsic value" for the option becomes more defined as we approach the expiry date.

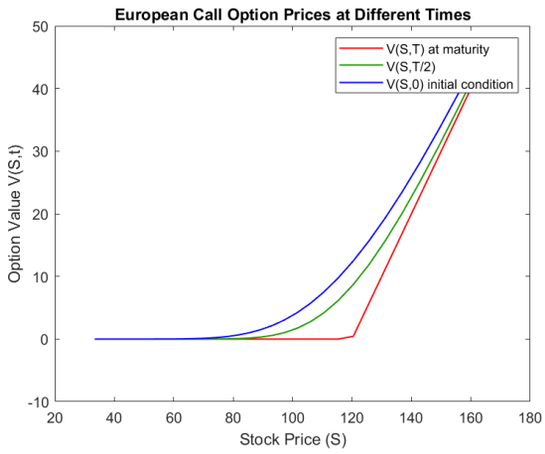

Figure 5.2: Numerical Solution for $t = 0,\ T/2,\ T$

To check the accuracy of the solution, we perform error analysis in the graph below by calculating $e = V_{exact} - V_{fem}$ (NOTE: V_{exact} is the analytical solution of the option price given (S, t) and it calculated on MATLAB using the blsprice() function.) We can observe below that the numerical solution is slightly inaccurate for values around the strike price $S = K$.

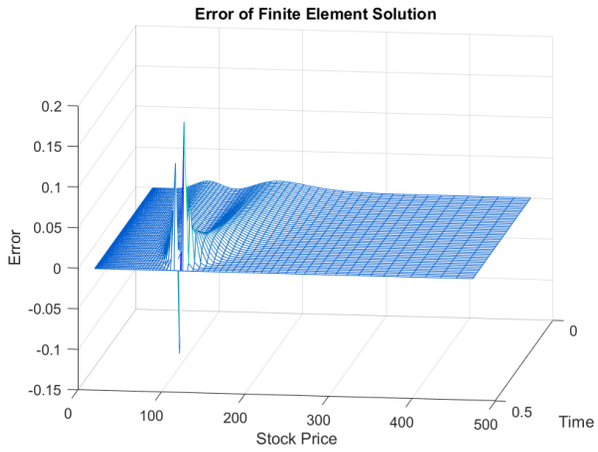

Figure 5.3: The error for the call option $V_{exact} - V_{fem}$

Note: In each graph, we don't include the boundaries of the domain for the stock price since our main interest is in the FEM approximation around the value of the strike price K.

5.4.4 Example 2: European Put Option FEM Approximation

We consider the **put option price of Microsoft stock** with the following parameters:

- Time to Maturity $T = 0.25$ (3 months)
- Strike Price $K = 175\,\$$ (The price we pay for the stock if we choose to buy it)
- Volatility $\sigma = 30\%$ (variance)
- risk-free interest rate $r = 4\%$
- Min/Max price of stock $S_{min} = 25,\ S_{max} = 500$.

For the implementation on MATLAB we choose $N = 100\,, M = 50$ for the number of spatial and temporal points, respectively. The MATLAB program we use to solve this problem is BSFEMPut.m (See Appendix 1 for MATLAB code). Below we graph the results of our solution.

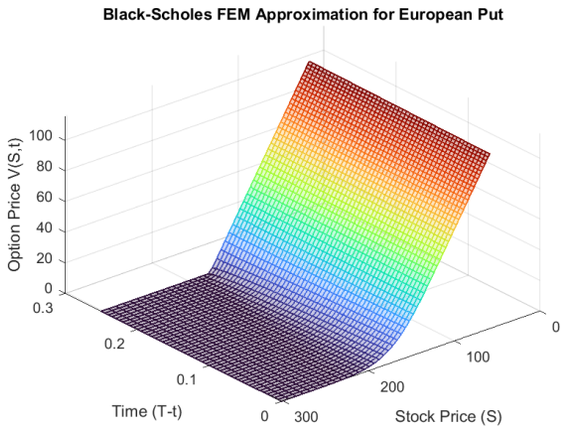

Figure 5.4: 3D plot of FEM solution for put option

As we explained before, we can see that as $t \to T$, our equation for ξ_l "sharpens" around the the strike price K. The only difference in this example is that we are working with a put option, which means **the value of the option will increase as the stock price decreases.**

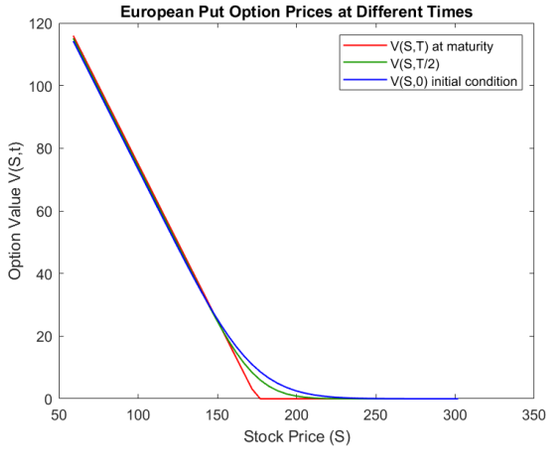

Figure 5.5: Numerical Solution for $t = 0,\, T/2,\, T$

To check the accuracy of the solution, we perform error analysis in the graph below by calculating $e = V_{exact} - V_{fem}$. For values around the strike price $S = K$ we again observe inaccuracies in the numerical solution.

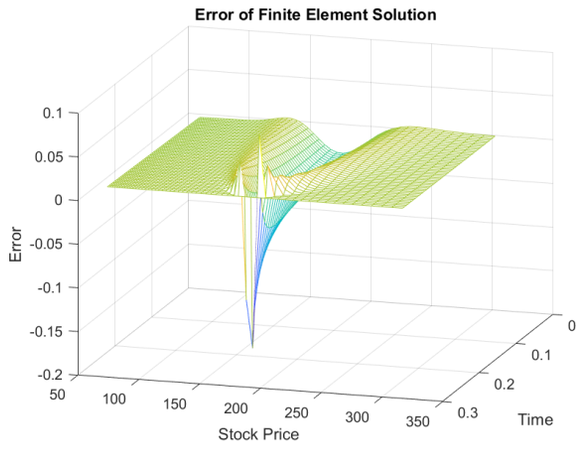

Figure 5.6: The error for the put option $V_{exact} - V_{fem}$

Note: In each graph, we don't include the boundaries of the domain for the stock price since our main interest is in the FEM approximation around the value of the strike price K.

6 | Conclusions and Future Work

At this point, we have now explored the ways in which the Finite Element Method can be applied to several types of differential equations. From exploring the mathematical theory that acts as a foundation for its implementation to its application for real world problems, we have built a solid understanding of a numerical method that has many advantages, such as:

- Weak restriction requirements for differential equations.
- Strong theoretical foundation (using Lax-Milgram).
- Accuracy: A highly accurate method with linear (or faster) convergence for the error.
- Versatility: The method can be applied to a wide range of differential equations.

With the knowledge we have gained, we can explore many other areas where the FEM can be extended for future work, such as

- Applying the FEM to Elliptic PDE's with *Neumann or Robin boundary conditions.*
- Implementing an *adaptive* Finite Element Method (whereby the triangulation is refined in specific areas of the domain where the solution has quicker variance)
- Implementing the FEM for American options, which can be formulated as a *free boundary problem.*

A1 | MATLAB Codes for the FEM

A1.1 FEM1DOfficial.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    1D Finite Element Method Solver
%  This code solves the model problem -u''(x) = f for Omega = [x1,x2] with
%  Dirichlet boundary conditions u(x1) = u(x2) = 0
%
%  This program solves the method for different values of h (step size) and
%  provides error analysis by producing the error for the infinity norm,
%  L2 norm and H1 norm.
%
%  x1, x2 - interval bounds
%
%  node_no - number of nodes
%  element_no - number of elements
%  x - our array which contains the coordinates of our nodes
%  h - length of each element
%
%  [xi, w] - Our gaussian points and gaussian weights (respectively)
%
%  nodes - this will be our 2 x (element_no) array which contains the
%  numbering for each pair of nodes
%
%  A_global, B_global - This will be our global matrices for our stiffness
%  matrix and load vector, respectively
%
%  u - Our finite element solution
%
%
%  REFERENCE:  Numerical Solution of Differential Equations -- Introduction to
%              Finite Difference and Finite Element Methods, Cambridge
    University Press, 2017


%%%%%%%%%%%%%%%%%%%%%%%%%      Main Section      %%%%%%%%%%%%%%%%%%%%%%%%%

close all
clear all

L2_errors = [];
H1_errors = [];
h_values = [];

fprintf('numElem\t\th\t\t\tL2Error\t\t\t\tH1Error\n');

for numNodes = [6 11 21 41 81 161 321 641]
    numElem = numNodes - 1;
    x1 = 0;
```

```
    x2 = 3;
    h = (x2 - x1)/numElem;
    [L2err, H1err, A_global] = main(numNodes);

    % Store errors and corresponding step sizes
    L2_errors = [L2_errors, L2err];
    H1_errors = [H1_errors, H1err];
    h_values = [h_values, h];
    fprintf('%d\t\t%.6f\t\t%.6e\t\t%.6e\n', numElem ,h, L2err, H1err);
end

% Plot errors vs. step sizes
loglog(h_values, L2_errors, 'ro-', h_values, H1_errors, 'bo-', 'LineWidth', 1.5)
hold on
loglog(h_values, 0.5*(L2_errors(1)/(h_values(1)^2))*h_values.^2, 'r--', ...
       h_values, 0.5*(H1_errors(1)/h_values(1))*h_values, 'b--', 'Linewidth', 1.5)
lgd = legend("L_2 error", "H_1 error", 'O(h^2)', 'O(h^1)', 'Location', 'southeast');
xlabel('Mesh size')  % x-axis label
ylabel('Error')  % y-axis label
title('Error Analysis for L2 and H1 norm')
lgd.FontSize = 10;


function [L2err, H1err, A_global] = main(node_no)

    % Define bounds of the domain
    x1 = 0;
    x2 = 3;

    % Define step size h
    element_no = node_no - 1;
    h = (x2-x1)/element_no;

    x = zeros(node_no,1);
    [xi, w] = quadrature();

    % Setup x vector for coordinates of nodes on the domain
    for i = 1:node_no
        x(i) = x1 + (i-1)*h;
    end

    % Setup matrix for storing the node pairs for each element
    nodes = zeros(2, element_no);
    for i = 1:element_no
        for j = 1:2
            nodes(j,i) = j + (i-1);
        end
    end

    % Initialise global stiffness matrix and load vector
    A_global = zeros(node_no, node_no);
    B_global = zeros(node_no, 1);

    % Assemble stiffness matrix and load vector
    [A_global, B_global] = form_matrices(nodes, x, element_no, xi, w, A_global, B_global, node_no);

    % Solve for u
    u = A_global\B_global;

```

```
106
107
108     %%%%%%%%%%%%%%%%%%%%%%     Exact Solutions to u     %%%%%%%%%%%%%%%%%%%%%%
109
110
111     uexact = @(x) -(x-3)^2*x^2;
112     uderivative = @(x) -4*x^3 + 18*x^2 - 18*x;
113
114
115     %%%%%%%%%%%%%%%%%%%%%%      Calculate Errors     %%%%%%%%%%%%%%%%%%%%%%%%
116
117     [L2err, H1err] = error(uexact, uderivative, x, xi, w, nodes, u, element_no)
        ;
118
119     x3 = linspace(x1, x2, 1000);
120     u_analytical = u_exact(x3);
121
122     % Plotting
123     figure;
124     plot(x, u, 's-', "LineWidth", 1.5);
125     hold on;
126     plot(x3, u_analytical, 'r-', 'LineWidth', 1.5);
127     hold on;
128     xlabel('x');
129     ylabel('u(x)  vs.  u_{fem}(x)');
130     title(sprintf('Finite Element Solution vs Exact Solution for numElem = %d',
         node_no-1));
131     legend('Finite Element Solution', 'Exact Solution');
132     hold off;
133
134
135     %%%%%%%%%%%%%%%%%%%%%%      Post-Processing      %%%%%%%%%%%%%%%%%%%%%%%%
136
137     x3 = linspace(x1, x2, 1000);
138     for i = 1:length(x3)
139         u_analytical(i) = uexact(x3(i));
140     end
141
142 end
143
144
145 %%%%%%%%%%%     Numerical Quadrature and Reference Element     %%%%%%%%%%%
146
147
148 function [xi, w] = quadrature()
149     xi = [-1/sqrt(3); 1/sqrt(3)];
150     w = [1; 1];
151 end
152
153 function [N, dN] = basis(xi)
154     N = [(1-xi)/2; (1+xi)/2];
155     dN = [-1/2; 1/2];
156 end
157
158
159 %%%%%%%%%%%%%%%%%%%%     Assemble Matrices A, B     %%%%%%%%%%%%%%%%%%%%%%%
160
161
162 function [A_global, B_global] = form_matrices(nodes, x, element_no, ...
163     xi, w, A_global, B_global, node_no)
164     for nel = 1:element_no
165         i1 = nodes(1, nel);
166         i2 = nodes(2, nel);
```

```
167
168         [A_local, B_local] = local_mat(x(i1), x(i2), xi, w);
169         [A_global, B_global] = global_mat(A_local, B_local, nel, nodes,
        A_global, B_global);
170     end
171
172
173     % Apply Dirichlet boundary conditions
174     A_global(1, :) = 0;  % Zero out the first row
175     A_global(:, 1) = 0;  % Zero out the first column
176     A_global(1, 1) = 1;  % Set diagonal element to 1
177     B_global(1) = 0;  % Set the value at the first node
178
179     A_global(node_no, :) = 0;  % Zero out the last row
180     A_global(:, node_no) = 0;  % Zero out the last column
181     A_global(node_no, node_no) = 1;  % Set diagonal element to 1
182     B_global(node_no) = 0;  % Set the value at the last node
183 end
184
185
186
187 %%%%%%%%%%%%%%%%%%%%      Calculate Local Matrices      %%%%%%%%%%%%%%%%%%%%
188
189
190 function [A_local, B_local] = local_mat(x1, x2, xi, w)
191     dx = (x2-x1)/2;
192     xk = 1;
193
194
195     A_local = zeros(2,2);
196     B_local = zeros(2,1);
197
198     for l=1:2
199         x = x1 + (x2-x1)/2 * (1+xi(l));
200         xf = f(x);
201         [N,dN] = basis(xi(l));
202         for i =1:2
203             B_local(i) = B_local(i) + N(i)*xf*w(l)*dx;
204             for j =1:2
205                 A_local(i,j) = A_local(i,j) + (xk*dN(i)*dN(j)/(dx*dx))*w(l)*dx;
206             end
207         end
208     end
209 end
210
211
212 %%%%%%%%%%%%%%%%%%%%%%     Assemble Global Matrices    %%%%%%%%%%%%%%%%%%%%%%
213
214
215 function [A_global, B_global] = global_mat(A_local, B_local, nel, nodes,
        A_global, B_global)
216     for i = 1:2
217         iglobal = nodes(i, nel);
218         B_global(iglobal) = B_global(iglobal) + B_local(i);
219         for j = 1:2
220             jglobal = nodes(j, nel);
221             A_global(iglobal, jglobal) = A_global(iglobal, jglobal) + A_local(i
        ,j);
222         end
223     end
224 end
225
226
```

```
227
228
229
230 %%%%%%%%%%%%%%%%%       Calculate the Error Norms       %%%%%%%%%%%%%%%%%
231
232 function [L2err, H1err] = error(uexact, uderivative, x, xi, w, nodes, u,
    element_no)
233     L2err = 0;
234     H1err = 0;
235     for nel = 1:element_no
236         i1 = nodes(1, nel);
237         i2 = nodes(2, nel);
238
239         h = x(i2) - x(i1);
240
241         errl2 = 0;
242         errh1 = 0;
243         for ig = 1:length(xi)
244             [N, dN] = basis(xi(ig));
245             x_interp = N(ig) * x(i1) + (1 - N(ig)) * x(i2); % Interpolated x
    within the element
246             dx_interp = (dN(ig) * x(i1) + (1 - dN(ig)) * x(i2))/2 ; %
    Interpolated derivative of x within the element
247             u_interp = N(ig) * u(i1) + (1 - N(ig)) * u(i2); % Interpolated u
    within the element
248
249
250
251             u_deriv = uderivative(dx_interp);
252             u_ex = uexact(x_interp); % Exact solution at the interpolated point
253             u_deriv_interp = dN(ig) * u(i1) + (1 - dN(ig)) * u(i2);
254
255             errl2 = errl2 + (u_ex - u_interp)^2 * w(ig);
256             errh1 = errh1 + ((u_ex - u_interp)^2 + ((u_deriv - u_deriv_interp)*
    h).^2) * w(ig);
257         end
258
259         % Scale error by element size and accumulate
260         L2err = L2err + errl2 * h;
261         H1err = H1err + errh1*h;
262     end
263
264     % Take square root to obtain L2, H1 error norm
265     L2err = sqrt(L2err);
266     H1err = sqrt(H1err);
267 end
268
269 %%%%%%%%%%%%%     Exact Solution for u, values for f     %%%%%%%%%%%%%%%%
270
271 function y = u_exact(x)
272
273     y = -(x-3).^2.*x.^2;
274
275 end
276
277 function y = f(x)
278     y = 12*x^2-36*x+18;
279 end
```

A1.2 FEM2DOfficial.m

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%                   2D Finite Element Method Solver
%  This code solves the model problem -\nabla u = f for a sqaure (or
    rectangular)
%  domain \Omega = [xmin,xmax] x [ymin,ymax] (Can be modified to your liking).
%
%  xmin, xmax, ymin, ymax - the bounds of our domain
%  Nx, Ny - Number of elements in the x,y direction resp.
%
%  nele - number of elements. (We multiply by 2 since each 'square' is made up
    of 2 triangles)
%
%  p(1,1), p(1,2), ... , p(1,n) <-- x coordinates of the nodal points
%  p(2,1), p(2,2), ... , p(2,n) <-- y coordinates of the nodal points
%
%  t(1,1), t(1,2), ... , t(1, nele) <-- index of the first node of an element
%  t(2,1), t(2,2), ... , t(2, nele) <-- index of the second node of the element
%  t(3,1), t(3,2), ... , t(3, nele) <-- index of the thrid node of the element
%
%  e(1,1), e(1,2), ... , e(1, nbc) <-- index of the beginning node of a
    boundary edge
%  e(2,1), e(2,2), ... , e(2, nbc) <-- index of the end node of a boundary edge
%
%
%  REFERENCE:  Numerical Solution of Differential Equations -- Introduction to
%              Finite Difference and Finite Element Methods, Cambridge
    University Press, 2017
%

close all
clear all


%%%%%%%%%%%%%%%      Plotting FEM Solution and Error     %%%%%%%%%%%%%%%%
errors = [];
meshSizes = [];

%%%%  State the type of domain here. ('square' or 'circle')  %%%%%%%%
type = "square";

if type == "square"
    fprintf('numElem (on each axis)\t\tInfNormError\t\tRate of Convergence\n');
elseif type == "circle"
    fprintf('h\t\t\t\t\tInfNormError\t\tRate of Convergence\n');
end

i = 1;
for numElem = [4,8,16,32,64]

    [ufem, u_exact, totalerr, p, e, t, err] = main(numElem,numElem, type);
    if type == "square"
        h(i) = 1/numElem;
    elseif type == "circle"
        h(i) = 2/numElem;
    end

    % Store errors and corresponding step sizes
    errors = [errors, totalerr];
    meshSizes = [meshSizes, numElem]; % Update here
    if i == 1
        if type == "square"
            fprintf('%d\t\t\t\t\t\t\t%.6e\t\n', numElem, totalerr);
        elseif type == "circle"
             fprintf('%f\t\t\t%.6e\t\n', 2/numElem, totalerr);
```

```
        end

    else
        rate = log2(errors(i)/errors(i-1));
        if type == "square"
            fprintf('%d\t\t\t\t\t\t\t%.6e\t\t%d\t\n', numElem, totalerr, abs(
    rate));
        elseif type == "circle"
            fprintf('%f\t\t\t%.6e\t\t%d\t\n', 2/numElem, totalerr, abs(rate));
        end
    end
    figure;
    subplot(1, 2, 1);
    pdemesh(p,e,t, 'XYData', ufem, 'ZData', ufem, 'Mesh','on','ColorMap','jet')
    ;

    if type == "square"
        title(['FEM Solution for h = ', num2str(1/numElem)]);
    elseif type == "circle"
        title(['FEM Solution for h = ', num2str(2/numElem)]);
    end
    subplot(1, 2, 2);
    pdemesh(p,e,t,'XYData', err, 'ZData', err, 'Mesh','on','ColorMap','jet');
    title('FEM error u_{exact} - u_{fem}');
    i = i+1;
end

% Plot errors vs. step sizes
figure;
loglog(h, errors, 'ro-', 'LineWidth', 1.5);
hold on
loglog(h, 0.5*(errors(1)/(h(1)^2))*h.^2, 'b--', 'Linewidth', 1.5)
lgd = legend("error", 'O(h^2)', 'Location', 'southeast');
xlabel('Mesh size')  % x-axis label
ylabel('Error')  % y-axis label
title('Error Analysis for Infinity Norm')
lgd.FontSize = 10;



function [ufem, u_exact, totalerr, p, e, t, err] = main(Nx,Ny, type)


%%%%%%%%%%%%%%%%%%      Set up Uniform Triangulation      %%%%%%%%%%%%%%%%

% Define the domain [xmin, xmax] x [ymin, ymax]
xmin = 0;
xmax = 1;
ymin = 0;
ymax = 1;


% Generate nodal points for p
x = linspace(xmin, xmax, Nx+1);
y = linspace(ymin, ymax, Ny+1);
[X, Y] = meshgrid(x, y);
p = [X(:)'; Y(:)']; % This will be our coordinates for each nodal point

% Generate elements and assign to t
nele = Nx*Ny*2; % Total number of elements
t = zeros(4, nele);
count = 1;
for i = 1:Nx
```

```
    for j = 1:Ny
        n1 = (j-1)*(Nx+1) + i;
        n2 = n1 + 1;
        n3 = n1 + Nx + 1;
        n4 = n3 + 1;

        % Assign nodes for indices of triangles
        t(:,count) = [n1; n2; n3; 1]; % First triangle
        t(:,count+1) = [n2; n4; n3; 1]; % Second triangle

        count = count + 2;
    end
end


% Generate boundary edges and assign to e
nbc = 2*(Nx + Ny); % Total number of boundary edges
e = zeros(2, nbc);
count = 1;

% Bottom boundary
for i = 1:Nx
    e(:,count) = [i; i+1];
    count = count + 1;
end

% Right boundary
for j = 1:Ny
    e(:,count) = [(j-1)*(Nx+1) + Nx + 1; j*(Nx+1) + Nx + 1];
    count = count + 1;
end

% Top boundary
for i = Nx:-1:1
    e(:,count) = [Ny*(Nx+1) + i + 1; Ny*(Nx+1) + i];
    count = count + 1;
end

% Left boundary
for j = Ny:-1:1
    e(:,count) = [j*(Nx+1)+1; (j-1)*(Nx+1)];
    count = count + 1;
end



%%%%%%%%%%%%%%%%%%%%     Assembling and Solving     %%%%%%%%%%%%%%%%%%%%


if type == "circle"
    [p, e, t] = initmesh("circleg",'hmax',1/(Nx/2));
end

% Extract the number of elements and nodes
[~, nelem] = size(t);
[~, nnode] = size(p);

% Initialize arrays to store nodal indices for each element
nodes = zeros(3, nelem);

% Loop through each element to extract nodal indices
for i = 1:nelem
    nodes(1,i) = t(1,i);
```

```
    nodes(2,i) = t(2,i);
    nodes(3,i) = t(3,i);
end

% Initialize global stiffness matrix and force vector
A_global = zeros(nnode, nnode);
B_global = zeros(nnode, 1);


% Define out quadrature points and weights

% 3 point quadrature
%xi_eta = [0, 1/2; 1/2, 0; 1/2, 1/2]; % Quadrature points
%weights = [1/6, 1/6, 1/6]; % Weights for each quadrature point

% 1 point quadrature
%xi_eta = [1/3,1/3];
%weights = 1/2;

% 4 point quadrature
xi_eta = [1/3,1/3; 2/15, 11/15; 2/15, 2/15; 11/15, 2/15];
weights = [-27/96, 25/96, 25/96, 25/96];

% Loop through each element to build stiffness matrix and load vector using quadrature
for nel = 1:nelem
    % Get nodal indices and coordinates for the current element
    i1 = nodes(1,nel);
    i2 = nodes(2,nel);
    i3 = nodes(3,nel);

    x1 = p(1,i1); y1 = p(2,i1);
    x2 = p(1,i2); y2 = p(2,i2);
    x3 = p(1,i3); y3 = p(2,i3);


    % Calculate the area of the triangle (for the Jacobian)
    area = abs(det([1,x1,y1;1,x2,y2;1,x3,y3]))/2;

    % Initialize element stiffness matrix and force vector
    A_local = zeros(3, 3);
    B_local = zeros(3, 1);

    % Loop over quadrature points
    for q = 1:length(weights)

        % Transform (  ,  ) to (x,y) using the shape functions
        xi = xi_eta(q, 1);
        eta = xi_eta(q, 2);
        N = [(1 - xi - eta), xi, eta]; % Linear shape functions

        % Calculate derivatives of basis functions
        dNdxi = [-1, 1, 0;
                 -1, 0, 1];

        % Calculation of the Jacobian matrix
        Jac = [x2 - x1, y2 - y1;
             x3 - x1, y3 - y1];


        % Calculation of derivatives of N with respect to x and y using chain rule
        dNdx_dy = 1/2 * (Jac \ dNdxi);
```

```
246
247         dNdx = dNdx_dy(1, :);
248         dNdy = dNdx_dy(2, :);
249
250         x = N(1)*x1 + N(2)*x2 + N(3)*x3;
251         y = N(1)*y1 + N(2)*y2 + N(3)*y3;
252
253         % Compute the Jacobian for the transformation
254         J = area*2; % Since the area of the master element is 1/2
255
256         % Integrate stiffness matrix and force/load vector
257         for i = 1:3
258             B_local(i) = B_local(i) + N(i)*f(x,y)*J*weights(q);
259
260             for j = 1:3
261                 A_local(i,j) = A_local(i,j) + (dNdx(i)*dNdx(j) + dNdy(i)*dNdy(j
    ))*J/(2);
262             end
263         end
264     end
265
266     % Assembly into global stiffness matrix and load vector
267     for i= 1:3
268         iglobal = nodes(i,nel);
269         B_global(iglobal) = B_global(iglobal) + B_local(i);
270         for j=1:3
271             jglobal = nodes(j,nel);
272             A_global(iglobal,jglobal) = A_global(iglobal,jglobal) + A_local(i,j
    );
273         end
274     end
275 end
276
277
278 % Extract the number of prescribed boundary conditions
279 [~, npres] = size(e);
280 g = zeros(npres, 1);
281 % Apply Dirichlet boundary conditions
282 for i = 1:npres
283     xb = p(1, e(1,i));
284     yb = p(2, e(1,i));
285
286     g(i) = uexact(xb,yb);
287 end
288
289
290 % Modify global stiffness matrix and force vector for Dirichlet boundary
    conditions
291 for i = 1:npres
292     nod = e(1,i);
293     for k = 1:nnode
294         B_global(k) = B_global(k) - A_global(k, nod)*g(i);
295         A_global(nod,k) = 0;
296         A_global(k,nod) = 0;
297     end
298     A_global(nod, nod) = 1;
299     B_global(nod) = g(i);
300 end
301
302
303 % Solve equation to obtain FEM solution
304 ufem = A_global\B_global;
305
```

```
306
307
308 %%%%%%%%%%%%%%%%%%%%%%     Calculating Error     %%%%%%%%%%%%%%%%%%%%%%%%%
309
310
311 % Obtain exact solution for u
312 u_exact = zeros(nnode, 1);
313 for i = 1:nnode
314     xt = p(1,i); yt = p(2,i);
315     u_exact(i) = uexact(xt, yt);
316 end
317     % Calculate infinity norm error
318     err = ufem - u_exact;
319     totalerr = abs(norm(err,Inf));
320
321 end
322
323 %%%%%%%%%%%%%     Exact Solution for u, values for f     %%%%%%%%%%%%%%%%
324
325 function yp = uexact(x,y)
326
327     % 1.
328     %yp = (1-x*x - y*y)/(4);
329
330     % 2.
331     %yp = -cos(pi*x);
332
333     % 3.
334     % yp = -sin(pi*x)*cos(2*pi*y);
335
336     % 4.
337     %yp = x*x + y*y;
338
339     % 5.
340     yp = sin(pi*x)*sin(pi*y);
341
342     % 6.
343     %yp = 1/4*(x^2+y^4)*sin(pi*x)*cos(4*pi*y);
344 return
345
346 end
347
348
349 function f = f(x,y)
350
351     % 1.
352     %f = 1;
353
354     % 2.
355     %f = -pi*pi*cos(pi*x);
356
357     % 3.
358     % f = -5*pi^2*cos(2*pi*y)*sin(pi*x);
359
360     % 4.
361     %f = -4;
362
363     % 5.
364      f = 2*pi^2*sin(pi*x)*sin(pi*y);
365
366     % 6.
367     %f = 17*pi^2*cos(4*pi*y)*sin(pi*x)*(x^2/4 + y^4/4) - 3*y^2*cos(4*pi*y)*sin(
    pi*x) - (cos(4*pi*y)*sin(pi*x))/2 - x*pi*cos(pi*x)*cos(4*pi*y) + 8*y^3*pi*
```

```
    sin(pi*x)*sin(4*pi*y);

    return
end
```

A1.3 BSFEMCall.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                  1D Black-Scholes Equation Solver for
%                         European Call Option
%
%  The following code solves the B-S equation as a transformed heat
%  equation using the Finite Element Method. This program solves the
%  equation for different values of N and M (no. of spatial and temporal
%  points resp.) and plots the solution. We also perform basic error
%  analysis by taking the analytical solution and solving for the error.
%
%  T - Time to Maturity
%  K - Strike Price
%  r - risk free interest rate
%  sigma - volatility
%
%  x_min, x_max - stock price bounds (these are implemented so we can
%  define the spatial domain)
%  x, tau - space and time domain resp.
%  x_eff - spatial domain excluding boundaries
%
%  S, k1, r - variables for transformation to heat equation
%
%  b - matric used to describe the boundary conditions as described in
%  Section 5
%
%  xi - matrix for the time variables (calculated using Crank-Nicolson)
%

close all
clear all


%%%%%%%%%%%%%%%%%%%%   Initial Setup of Variables   %%%%%%%%%%%%%%%%%%%%%%%

T = 0.5; K = 120;
r = 0.02; sigma = 0.35;


% Log transformation of stock price bounds
x_min = log(20/K); x_max = log(1500/K);



% Discretise the spatial and time domain

N = 100; M = 50;

x = linspace(x_min, x_max, N+2); dx = x(2) - x(1);
tau = linspace(0, T*(sigma^2)/2, M+1); dtau = tau(2) - tau(1);
x_eff = x(2 : length(x)-1);

xi = zeros(length(x_eff), length(tau));
b=xi;

k1 = 2*r / (sigma^2);
```

```
S = K*exp(x);


%%%%%%%%%%%%%%%%%%  Establish boundary conditions  %%%%%%%%%%%%%%%%%%%%%%


% Initial condition for option value at time T (this is gamma in our notes)
IC = @(x, q) max(0, exp(0.5 * x * (q + 1)) - exp(0.5 * x * (q - 1)));

% Boundary conditions (alpha - value at x_min , beta - value at x_max)
beta = @(tau) exp(1/2*(k1+1)*x_max + 1/4*tau*(k1+1)^2);
dbeta = @(tau) (1/4*tau*(k1+1)^2)*beta(tau);
alpha = @(tau) 0;
phi_boundary = @(x, tau) (beta(tau) - alpha(tau)) .* (x - x_min)/(x_max - x_min
    ) + alpha(tau);

% Set initial condition.

xi(:,1) = IC(x_eff, k1)-phi_boundary(x_eff,0);

% Construct b to incorporate the boundary conditions into the FEM solution.
for i = 1 : length(tau)
    b(:, i) = ((x_eff - x_min)/(x_max - x_min)) * dx * (0.25 * (k1 + 1) ^ 2) *
    beta(tau(i));
end


%%%%%%  Assemble matrices and solve for xi using time stepping scheme  %%%%

A1 = zeros(N,N);
for i=1:N
    if i > 1
        A1(i, i-1) = -1/dx;
    end
    A1(i, i) = 2/dx;
    if i < N
        A1(i, i+1) = -1/dx;
    end
end
A1(1,1:2)=[2/dx,-1/dx]; A1(N,N-1:N)=[-1/dx,2/dx];

B1 = zeros(N,N);
for i=1:N
    if i > 1
        B1(i, i-1) = dx/6;
    end
    B1(i, i) = 2*dx/3;
    if i < N
        B1(i, i+1) = dx/6;
    end
end
B1(1,1:2)=[2*dx/3,dx/6]; B1(N,N-1:N)=[dx/6,2*dx/3];


% Solve for xi using the Crank-Nicolson Scheme
B_final = B1 + (0.5 * dtau) .* A1;
A_final = B1 - (0.5 * dtau) .* A1;

for i = 2 : length(tau)
    xi(:, i) = B_final \ (A_final * xi(:, i - 1) - (dtau/2) * (b(:, i) + b(:, i
    -1)));
end
```

```
116
117 %%%%%%%%%%%%%%%%%%   Option Price Calculation V(S,t)   %%%%%%%%%%%%%%%%%%
118
119 % Get option price at nodes
120 base = zeros(length(x), length(tau));
121 for i = 1 : length(x)
122     for j = 1 : length(tau)
123         base(i, j) = phi_boundary(x(i), tau(j));
124     end
125 end
126
127 nodes = [zeros(1, length(tau)); xi; zeros(1, length(tau))];
128 nodes = nodes + base;
129
130
131 % Transform option prices back to the stock price and time domain
132 for i=1:length(x)
133     for j=1:length(tau)
134         nodes(i,j)=(K*exp((-0.5)*(k1-1)*x(i)+(-0.25)*((k1+1)^2)*tau(j)))*nodes(
    i,j);
135     end
136 end
137
138
139
140 %%%%%%%%%%%%%%%%%%%%%%      Post-Processing      %%%%%%%%%%%%%%%%%%%%%%%%%
141
142
143
144 % Output a 3D mesh of our FEM approximation
145 V = nodes;
146 stock = S;
147 time = T-tau*2/sigma^2;
148 s = mesh(time',stock(round(N/8):round(N/2)),V(round(N/8):round(N/2),:));view(3)
    ;
149 s.FaceColor = 'interp';
150 s.FaceAlpha = 0.3;
151 s.EdgeAlpha = 1;
152 s.LineWidth = 0.75;
153 colormap turbo
154
155 %mesh(time', stock, Nodes_v);view(3);
156 title('Black-Scholes FEM Approximation for European Call');
157 ylabel('Stock Price (S)');xlabel('Time (T-t)');zlabel('Option Price V(S,t)');
158
159
160 % Output a 2D plot of the FEM Approximation for different time values
161 figure(2);
162 plot(stock(round(N/8):round(N/2)), V(round(N/8):round(N/2), 1), 'r-', '
    LineWidth', 1);
163 hold on;
164 plot(stock(round(N/8):round(N/2)), V(round(N/8):round(N/2), M/2), '-', '
    LineWidth', 1, 'Color',[0.1,0.6,0]);
165 plot(stock(round(N/8):round(N/2)), V(round(N/8):round(N/2),M), 'b-', 'LineWidth
    ', 1);
166 legend("V(S,T) at maturity", "V(S,T/2)", "V(S,0) initial condition");
167 xlabel('Stock Price (S)');
168 ylabel('Option Value V(S,t)');
169 title('European Call Option Prices at Different Times');
170
171
172
173 %%%%%%%%%%%%%%%%%%%%%%      Error Analysis      %%%%%%%%%%%%%%%%%%%%%%%%%
```

```
174
175 % Calculate the analytical solution to the B-S Equation (using financial
    toolbox for MATLAB)
176 for i = 1:length(x)
177     for j = 1:length(tau)
178         [Call, Put] = blsprice(K*exp(x(i)), K, r, T - (tau(j)*2)/sigma^2, sigma
    );
179         V_exact(i,j) = Call;
180     end
181 end
182
183
184
185 V_exact = flip(V_exact,2);
186 V_exact = V_exact(1:round(3*N/4), :);
187 V = V(1:3*N/4,:);
188 err = V_exact - V;
189 figure(3);
190 mesh(time, stock(1:round(3*N/4)), err);
191 colormap winter
192 title('Error of Finite Element Solution');
193 ylabel('Stock Price');xlabel('Time');zlabel('Error');
```

A1.4 BSFEMPut.m

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %                  1D Black-Scholes Equation Solver for
3 %                         European Put Option
4 %
5 %  The following code solves the B-S equation as a transformed heat
6 %  equation using the Finite Element Method. This program solves the
7 %  equation for different values of N and M (no. of spatial and temporal
8 %  points resp.) and plots the solution. We also perform basic error
9 %  analysis by taking the analytical solution and solving for the error.
10 %
11 %  T - Time to Maturity
12 %  K - Strike Price
13 %  r - risk free interest rate
14 %  sigma - volatility
15 %
16 %  x_min, x_max - stock price bounds (these are implemented so we can
17 %  define the spatial domain)
18 %  x, tau - space and time domain resp.
19 %  x_eff - spatial domain excluding boundaries
20 %
21 %  S, k1, r - variables for transformation to heat equation
22 %
23 %  b - matric used to describe the boundary conditions as described in
24 %  Section 5
25 %
26 %  xi - matrix for the time variables (calculated using Crank-Nicolson)
27 %
28
29 close all
30 clear all
31
32
33 %%%%%%%%%%%%%%%%%%%%   Initial Setup of Variables   %%%%%%%%%%%%%%%%%%%%%%%
34
35 T = 0.25; K = 175;
36 r = 0.04; sigma = 0.30;
37
```

```
38
39 % Log transformation of stock price bounds
40 x_min = log(25/K); x_max = log(500/K);
41
42
43
44 % Discretise the spatial and time domain
45
46 N = 100; M = 50;
47
48 x = linspace(x_min, x_max, N+2); dx = x(2) - x(1);
49 tau = linspace(0, T*(sigma^2)/2, M+1); dtau = tau(2) - tau(1);
50 x_eff = x(2 : length(x)-1);
51
52 xi = zeros(length(x_eff), length(tau));
53 b=xi;
54
55 k1 = 2*r / (sigma^2);
56 S = K*exp(x);
57
58
59 %%%%%%%%%%%%%%%%%%  Establish boundary conditions  %%%%%%%%%%%%%%%%%%%%%%
60
61
62 % Initial condition for option value at time T (this is gamma in our notes)
63 IC = @(x, k1) max(0, exp(0.5 * x * (k1 - 1)) - exp(0.5 * x * (k1 + 1)));
64
65 % Boundary conditions (alpha - value at x_min , beta - value at x_max)
66 alpha = @(tau) exp(1/2*(k1-1)*x_max + 1/4*tau*(k1-1)^2);
67 beta = @(tau) 0;
68 phi_boundary = @(x, tau) (beta(tau) - alpha(tau)) .* (x - x_min)/(x_max - x_min
       ) + alpha(tau);
69
70 % Set initial condition.
71
72 xi(:,1) = IC(x_eff, k1)-phi_boundary(x_eff,0);
73
74 % Construct b to incorporate the boundary conditions into the FEM solution.
75 for i = 1 : length(tau)
76     b(:, i) = ((x_eff - x_min)/(x_max - x_min)) * dx * (0.25 * (k1 + 1) ^ 2) *
       beta(tau(i));
77 end
78
79
80 %%%%%%  Assemble matrices and solve for xi using time stepping scheme  %%%%
81
82 A1 = zeros(N,N);
83 for i=1:N
84     if i > 1
85         A1(i, i-1) = -1/dx;
86     end
87     A1(i, i) = 2/dx;
88     if i < N
89         A1(i, i+1) = -1/dx;
90     end
91 end
92 A1(1,1:2)=[2/dx,-1/dx]; A1(N,N-1:N)=[-1/dx,2/dx];
93
94 B1 = zeros(N,N);
95 for i=1:N
96     if i > 1
97         B1(i, i-1) = dx/6;
98     end
```

```
    B1(i, i) = 2*dx/3;
    if i < N
        B1(i, i+1) = dx/6;
    end
end
B1(1,1:2)=[2*dx/3,dx/6]; B1(N,N-1:N)=[dx/6,2*dx/3];

% Solve for xi using the Crank-Nicolson Scheme
B_final = B1 + (0.5 * dtau) .* A1;
A_final = B1 - (0.5 * dtau) .* A1;

for i = 2 : length(tau)
    xi(:, i) = B_final \ (A_final * xi(:, i - 1) - (dtau/2) * (b(:, i) + b(:, i
    -1)));
end

%%%%%%%%%%%%%%%%%%   Option Price Calculation V(S,t)   %%%%%%%%%%%%%%%%%%

% Get option price at nodes
base = zeros(length(x), length(tau));
for i = 1 : length(x)
    for j = 1 : length(tau)
        base(i, j) = phi_boundary(x(i), tau(j));
    end
end

nodes = [zeros(1,length(tau)); xi; zeros(1,length(tau))];
nodes = nodes + base;


% Transform option prices back to the stock price and time domain
for i=1:length(x)
    for j=1:length(tau)
        nodes(i,j)=(K*exp((-0.5)*(k1-1)*x(i)+(-0.25)*((k1+1)^2)*tau(j)))*nodes(
    i,j);
    end
end



%%%%%%%%%%%%%%%%%%%%%      Post-Processing      %%%%%%%%%%%%%%%%%%%%%%%%%



% Output a 3D mesh of our FEM approximation
V = nodes;
stock = S;
time = T-tau*2/sigma^2;
s = mesh(time',stock(round(3*N/10):round(85*N/100)),V(round(3*N/10):round(85*N
    /100),:));view(3);
%mesh(time', stock, Nodes_v);view(3);
s.FaceColor = 'interp';
s.FaceAlpha = 0.3;
s.EdgeAlpha = 1;
s.LineWidth = 0.75;
colormap turbo
title('Black-Scholes FEM Approximation for European Put');
ylabel('Stock Price (S)');xlabel('Time (T-t)');zlabel('Option Price V(S,t)');


% Output a 2D plot of the FEM Approximation for different time values
figure(2);
plot(stock(round(3*N/10):round(85*N/100)), V(round(3*N/10):round(85*N/100), 1),
```

```
    'r-', 'LineWidth', 1);
hold on;
plot(stock(round(3*N/10):round(85*N/100)), V(round(3*N/10):round(85*N/100), M/2), '-', 'LineWidth', 1, 'Color',[0.1,0.6,0]);
plot(stock(round(3*N/10):round(85*N/100)), V(round(3*N/10):round(85*N/100),M), 'b-', 'LineWidth', 1);
legend("V(S,T) at maturity", "V(S,T/2)", "V(S,0) initial condition");
xlabel('Stock Price (S)');
ylabel('Option Value V(S,t)');
title('European Put Option Prices at Different Times');



%%%%%%%%%%%%%%%%%%%%%%      Error Analysis      %%%%%%%%%%%%%%%%%%%%%%%%%

% Calculate the analytical solution to the B-S Equation (using financial toolbox for MATLAB)
for i = 1:length(x)
    for j = 1:length(tau)
        [Call,Put] = blsprice(K*exp(x(i)), K, r, T - (tau(j)*2)/sigma^2, sigma);
        V_exact(i,j) = Put;
    end
end



V_exact = flip(V_exact,2);
V_exact = V_exact(round(3*N/10):round(85*N/100), :);
V = V(round(3*N/10):round(85*N/100),:);
err = V_exact - V;
figure(3);
mesh(time, stock(round(3*N/10):round(85*N/100)), err);
colormap parula
title('Error of Finite Element Solution');
ylabel('Stock Price');xlabel('Time');zlabel('Error');
```

A2 | Derivation of Heat Equation for Black-Scholes

(Reference: [14])

We start with the Black-Scholes differential equation:

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

for the price of an option $V(S,t)$ where S is the price of the underlying asset and T is the time to maturity. This equation is a "backward type" equation, as we can see from the graph below.

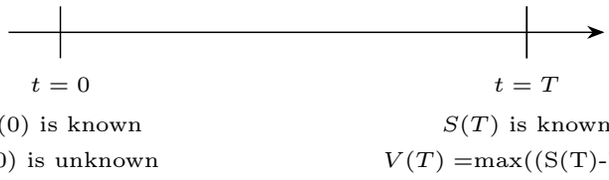

Figure A2.1: Black-Scholes equation without a change of variables

The idea is to make the model a *forward moving* differential equation. This requires a change of variables.

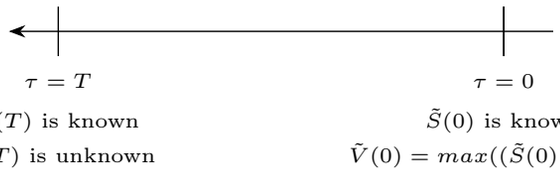

Figure A2.2: Black-Scholes equation with a change of variables

As stated in Chapter 5, let us use the change of variables:

$$S = Ke^x, \quad \tau = \frac{\sigma^2}{2}(T-t), \quad V(S,t) = Ku(x,\tau).$$

We now calculate the relevant derivatives

$$\frac{\partial V}{\partial t} = \frac{\partial u}{\partial \tau}\frac{\partial \tau}{\partial t} = K\frac{\partial u}{\partial \tau}\cdot\frac{-\sigma^2}{2}$$

$$\frac{\partial V}{\partial s} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial S} = K\left(\frac{\partial u}{\partial x}\right)\cdot\frac{1}{S} = e^{-x}\left(\frac{\partial u}{\partial x}\right)$$

and

$$\frac{\partial^2 V}{\partial S^2} = \frac{\partial}{\partial S}\left(e^{-x}\frac{\partial u}{\partial x}\right).$$

We note that $\partial S = \partial(Ke^x) = Ke^x \partial x$. Therefore

$$\frac{\partial^2 V}{\partial S^2} = \frac{e^{-x}}{K}\frac{\partial}{\partial x}\left(e^{-x}\frac{\partial u}{\partial x}\right) = \frac{e^{-x}}{K}\left(-e^{-x}\frac{\partial u}{\partial x} + e^{-x}\frac{\partial^2 u}{\partial x^2}\right)$$

which simplifies down to

$$\frac{\partial^2 V}{\partial S^2} = \frac{e^{-2x}}{K}\left(-\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\right).$$

Now we can substitute these into our equation.

$$\frac{\partial u}{\partial \tau}\cdot\frac{-K\sigma^2}{2} + \frac{1}{2}\sigma^2 K^2 e^{2x}\left(\frac{e^{-2x}}{K}\left(-\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\right)\right) + rKe^x e^{-x}\left(\frac{\partial u}{\partial x}\right) - rKv(x,\tau) = 0$$

We can now simplify down and divide across by $\frac{K\sigma^2}{2}$.

$$-\frac{\partial u}{\partial t} - \frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} + \frac{2r}{\sigma^2}\frac{\partial u}{\partial x} - \frac{2r}{\sigma^2}u(x,\tau) = 0.$$

The next step is to let $k_1 = \frac{2r}{\sigma^2}$,

$$\frac{\partial u}{\partial t} = (k-1)\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2} - ku(x,\tau).$$

We are now close to obtaining the standard heat equation, however we need to remove $\frac{\partial u}{\partial x}$ and $u(x,\tau)$ from our equation. To do this, we will use another transformation,

$$u(x,\tau) = e^{ax+bt}v(x,\tau).$$

We now calculate the partial derivatives,

$$\frac{\partial u}{\partial x} = ae^{ax+bt}v + e^{ax+bt}\frac{\partial v}{\partial x},$$
$$\frac{\partial u}{\partial t} = be^{ax+bt}v + e^{ax+bt}\frac{\partial v}{\partial t},$$
$$\frac{\partial^2 u}{\partial x^2} = a^2 e^{ax+bt}v + 2ae^{ax+bt}\frac{\partial v}{\partial x} + e^{ax+bt}\frac{\partial^2 v}{\partial x^2}.$$

We now substitute these into our equation and divide across by e^{ax+bt},

$$bv(x,\tau) + \frac{\partial v}{\partial t} = (k-1)\left(av(x,\tau) + \frac{\partial v}{\partial x}\right) + a^2 v + 2a\frac{\partial v}{\partial x} + \frac{\partial^2 v}{\partial x^2}$$

$$\rightarrow \frac{\partial v}{\partial t} = \frac{\partial^2 v}{\partial x^2} + v(x,\tau)(a(k_1-1) - k_1 - b + a^2) + \frac{\partial v}{\partial x}(k_1 - 1 + 2a).$$

Immediately we can eliminate $\frac{\partial v}{\partial x}$ if we let $a = \frac{1}{2}(k-1)$

We also need $a(k_1-1) - k_1 - b + a^2 = 0$. We can then calculate what the value of b needs to be,

$$-\frac{1}{2}(k_1-1)^2 - k_1 + \frac{1}{4}(k_1-1)^2 = b \rightarrow b = -\frac{1}{4}(k_1+1)^2.$$

Finally we reach the end with our desired transformation being

$$V(S,t) = Kexp\left(-\frac{1}{2}(k_1-1)x - \frac{1}{4}(k_1+1)^2\tau\right)v(x,\tau)$$

to obtain the heat equation

$$\frac{\partial v}{\partial t} = \frac{\partial^2 v}{\partial x^2}.$$

We shall briefly mention how our boundary conditions have changed.

Originally we had

- $V(0,t) = 0$ for $0 \leq t \leq T$
- $V(S,T) = max(S-K,0)$ for $0 \leq S \leq \infty$
- $V(S,t) \to S - Ke^{-r(T-t)}$ as $S \to \infty$.

With the transformation our boundary conditions are

- $v_c(x,0) = max\left\{e^{\frac{1}{2}(q+1)x} - e^{\frac{1}{2}(q-1)x},\ 0\right\}$
- $v_c(x_{-\infty},\tau) = 0$
- $v_c(x_{+\infty},\tau) = e^{\frac{1}{2}(q+1)x_{+\infty}+\frac{1}{4}(q+1)^2\tau}$

A3 | Applying the FEM to a 2D PDE on a circle domain

In the paper, we only explored the application of the Finite Element method in 2D to a problem defined on a square domain. Let us analyse the following problem where the PDE is defined on a *circle domain.*

Example A3.0.1. *Find u such that $-\nabla u = f = -5\pi^2 cos(2\pi y)sin(\pi x)$ on $\Omega = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 < 1\}$ and $\partial\Omega = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 = 1\}$.*

Instead of using the uniform triangulation, we can use a built in function on MATLAB, called initmesh(), to generate a triangulation for us. From there we apply the code FEM2DOFFICIAL.m to obtain our results.

On the following page we graph the numerical solution and the respective error for different levels of discretisation.

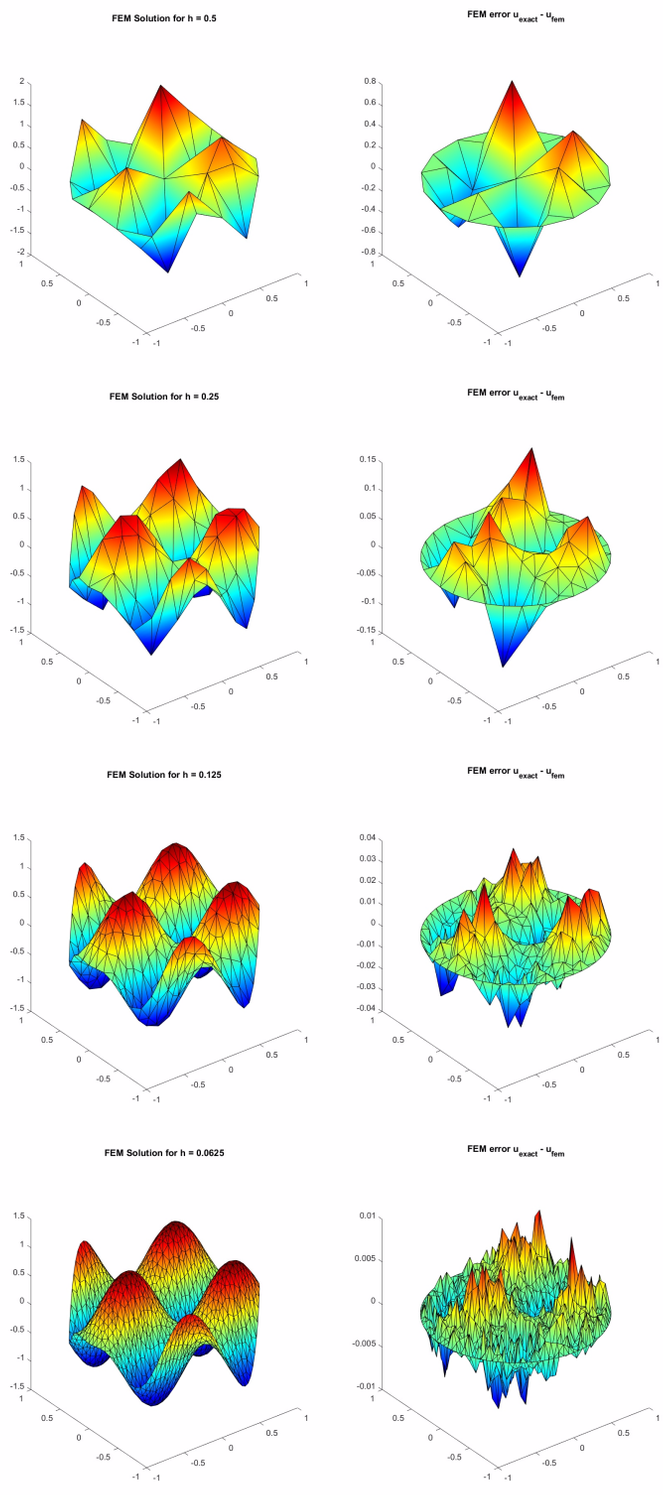

Figure A3.1: In the first column below we have the results for different levels of h (= approximate maximum length for side of a triangle in the triangulation). In the second column we plot the error $u_{exact} - u_{fem}$ at each nodal point.

Bibliography

[1] Claes Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Cambridge University Press, 1987.

[2] Thomas Wick. Numerical methods for partial differential equations, 2020. URL https://www.ifam.uni-hannover.de/wick.html.

[3] Fredik Bengzon Mats G. Larson. *The Finite Element Method: Theory, Implementations and Applications*. Springer, 2013.

[4] Tao Tang Zhilin Li, Zhonghua Qiao. *Numerical Solution to Differential Equations: Introduction to Finite Difference and Finite Element Methods*. Cambridge University Press, 2018.

[5] Autar Kaw. *Numerical Methods with Applications*. Online textbook. URL https://nm.mathforcollege.com/NumericalMethodsTextbookUnabridged/.

[6] L. Ridgeway Scott Susanne C. Brenner. *The Mathematical Theory of Finite Element Methods*. Springer, 2008.

[7] Finite element method for poisson equations with dirchlet boundary conditions, 2018. URL https://github.com/aprilzhizhou/finite_element_method_for_poisson_eqs. Github repository for project on Finite Element Method.

[8] Mehmet Dik Brandon Washburn. Derivation of black-scholes equation using itô's lemma. *Proceedings of International Mathematical Sciences, Volume 3, Issue 1, pp. 38-49*, 2021.

[9] Dexter Edwards. Numerical and analytic methods in option pricing, 2015.

[10] Tomas Björk. *Arbitrage Theory in Continuous Time, 2nd Edition*. Oxford Univerisity Press, 2003.

[11] J. González Salazar A. Andalaft-Chacur, M. Montaz Ali. Real options pricing by the finite element method. *Computers Mathematics with Applications, Volume 61, pp. 2863-2873*, 2011.

[12] Md. Anowar Hossain Md. Kazi Salah Uddin, Md. Noor-A-Alam Siddiki. Numerical solution of a linear black-scholes models: A comparative overview. *IOSR Journal of Engineering, Vol. 05, Issue 08 (August. 2015), ||V3|| PP 45-51*, 2015.

[13] Huang Jiaheng Hu Wei, Yao Wuguannan. The finite element method for option pricing under heston's model, 2016. Final Report for MA6621 Programming and Computing for Finance and Actuarial Science at the City Univeristy of Hong Kong.

[14] Sumardi Erny Rahayu Wijayanti and Nanang Susyanto. European call options pricing numerically using finite element method. *IAENG International Journal of Applied Mathematics, 52:4, IJAM*$_{52_{42}1}$, 2022.