23MAC300: A Computational Investigation into a Refined Method for Recovering the Gradient of a Finite Element Solution to an Elliptic Boundary Value Problem

James Jarman
supervised by Dr Marco Discacciati

May 9, 2024

Abstract

Computing the velocity and pressure of fluid flow within a porous medium poses a significant real-world challenge, typically addressed through modelling as an elliptic boundary value problem. While analytical solutions for this problem remain elusive, numerical methods become essential for recovering the gradient of the pressure, representing the velocity field. Traditional finite element methods, while useful, exhibit limitations in accurately recovering the desired gradient. Thus, this dissertation aims to investigate a refined finite element approach proposed by Loula et al. tailored to this specific problem domain. The study involves deriving the necessary theoretical framework and then implementing this in MATLAB. Analytical testing will validate a proposition made by Loula regarding the rate of convergence for this technique. Improvements in the convergence rates for computed solutions will be observed with the new technique compared to an in-built MATLAB method. Finally, potential extensions of the project will be highlighted in the concluding remarks.

Contents

1 Introduction - the aim of the project

The overarching aim of this report is to explore how we can recover the gradient of u (i.e. ∇u), where u denotes a solution to an elliptic PDE [1], which has been evaluated beforehand. Recovering the gradient is particularly useful, especially within the real-world context of computing the velocity and pressure of a fluid in a porous medium [2]. Typically, a computation of this nature would begin by determining u, the pressure. Subsequently, the gradient of u, denoting the velocity, is computed during a post-processing phase. In fact, the pressure, $u(x, y)$, is usually determined as the solution of the following elliptic boundary value problem:

$$-\Delta u = f \quad \text{in } \Omega, \tag{1}$$

$$u = 0 \quad \text{on } \Gamma. \tag{2}$$

Here, Ω is a bounded open subset of $\mathbb{R}^2$ with smooth boundary Γ and f is an assigned function. Let $\boldsymbol{\sigma}$ denote the velocity, which is calculated as the gradient of u. Thus, recalling the definitions of mathematical operators, *div* and ∇, the above two equations can be rewritten as:

$$-div\, \boldsymbol{\sigma} = f \quad \text{in } \Omega, \tag{3}$$

$$\boldsymbol{\sigma} = \nabla u \quad \text{in } \Omega, \tag{4}$$

$$u = 0 \quad \text{on } \Gamma. \tag{5}$$

The three equations above explicitly include the gradient field of u (namely $\boldsymbol{\sigma} = \nabla u$), and thereby provides a framework for the calculation of $\boldsymbol{\sigma} = \boldsymbol{\sigma}(x, y)$. In Section 2 we will introduce all essential definitions and mathematical background required to fully interpret this set of equations and subsequently solve the problem.

In general, the equations introduced above can not be solved analytically, and so some numerical method must be used to feasibly find an approximate solution. The finite element method (FEM) is a common and widely used method for numerically solving differential equations, in particular the method can provide approximate solutions for partial differential equations in two or three variables. In this report, we explore how finite element methods can be implemented to achieve our main objective of numerically calculating the gradient of a given finite element solution.

The solution of equations (1) and (2), u, will be computed by the finite element method. This method provides a representation of u as a linear combination of basis functions of a finite dimensional space. This representation will resemble $u_h = \sum_j u_j \varphi_j$, where φ_j are basis functions of the finite dimensional space which will be defined in Section 2.

The obvious approach to finding the gradient of this given finite element solution, u_h, would be to simply calculate the derivative of each element.

However, it can be shown by a clear visual example that in most cases this rudimentary approach produces a poor approximation. Considering an arbitrary 1-dimensional example, illustrated in Figure 1, it becomes apparent that the finite element solution is continuous (as it is composed of continuous straight elements that have the same value at each internal boundary of the mesh), however taking the simple derivative of each element results in a discontinuous function (shown in red). Consequently, the computed approximation is poor.

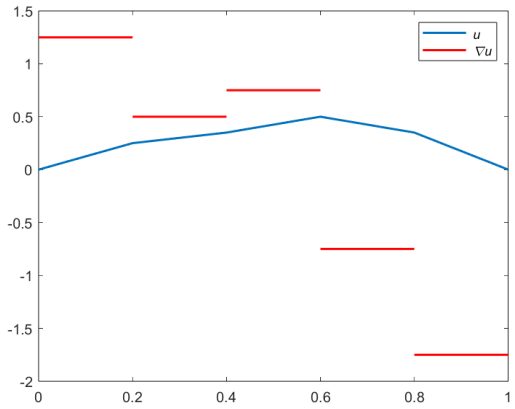


Figure 1: Illustrating the discontinuity of a rudimentary approximation for ∇u.

Fortunately, there are more refined techniques that can be constructed which compute gradient field approximations with a greater level of accuracy. In this report we will discuss and explore a particular global post-processing technique proposed by Loula et al [3]. The basic idea of Loula's method (which will be described in detail later in this report) is to compute u using a general displacement method and subsequently obtain $\boldsymbol{\sigma}$ from a variational formulation [4] constructed by combining the residual of the balance equation (3), and a weak formulation of the constitutive equation (4) (all these terms will be described in detail later in this report).

An overview of the report is as follows. All necessary mathematical background required will be introduced in Section 2 alongside an outline of the derivation of the variational formulation described in Loula's paper. In Section 3 we summarise the derivations of the algebraic formulation for the corresponding components of our variational formulation. Sections 4 and 5 showcase the most notable original contributions of this project: the numerical implementation of the method in MATLAB is presented and then the observed results are subsequently analysed. We will conclude this report in Section 6 by summarising what we have discovered and list possible future extensions for this project.

2 Mathematical Formulation

As stated in the introduction, the central goal of this report is to explore a developed variant of the FEM to recover the gradient of u (i.e. ∇u), where u is a previously determined finite element solution of a second-order elliptic boundary value problem. This section describes the necessary mathematical background and definitions required to derive the variational formulation presented in Loula's method that is proposed to solve our problem.

Recall that $u = u(x, y)$ is the solution of equations (1) and (2). It was prefaced in the introduction that these two equations can be manipulated in such a way to derive an equivalent set of equations (3)-(5). Equivalence can easily be shown in one line, however first it is important to define a few mathematical operators which will be used throughout the report.

2.1 Mathematical Operators

Definition 1. The ***Laplace operator***, Δ, is a second-order differential operator. For a two-dimensional Euclidean space, if $u = u(x, y)$ is a twice-differentiable real-valued function, then the ***Laplacian*** of u is defined as:

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \tag{6}$$

where $\frac{\partial^2}{\partial x^2}$ and $\frac{\partial^2}{\partial y^2}$ represent second-order partial derivatives with respect to the coordinates x and y, respectively.

Using this definition, we can immediately rewrite equation (1) as

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f. \tag{7}$$

Now, let us recall the set of equations (3)-(5) that are claimed to be equivalent to equations (1) and (2). To re-emphasise, after the introduction of the variable $\boldsymbol{\sigma}$ constrained by equations (3) and (4), our original aim of recovering ∇u is equivalent to recovering the gradient field $\boldsymbol{\sigma} = \boldsymbol{\sigma}(x, y)$. In this project we will be working entirely in two dimensions so it will be important to note that $\boldsymbol{\sigma}$ is a two-dimensional vector field and we can define its components thus

$$\boldsymbol{\sigma}(x, y) = \begin{pmatrix} \sigma_1(x, y) \\ \sigma_2(x, y) \end{pmatrix}.$$

Let us now define the divergence operator, div, and gradient operator, ∇, which appear in equations (3) and (4), respectively.

Definition 2. The ***divergence operator***, $div\,\mathbf{F}$, for a two-dimensional vector field $\mathbf{F}$ = (P, Q) is defined as:

$$div\,\mathbf{F} = \frac{\partial P}{\partial x} + \frac{\partial Q}{\partial y}.$$

Definition 3. The ***gradient operator*** of a two-dimensional function $f(x, y)$, denoted as ∇f, is defined as:

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}.$$

It is important to note that the gradient operator transforms the scalar function, $f(x, y)$, to a vector function, $\nabla f(x, y)$.

Now that we have defined all of the required operators, let us proceed with showing the equivalence of the two sets of equations.

Firstly, let us begin by rewriting equation (4) acknowledging how we have defined $\boldsymbol{\sigma}$ and the gradient operator. Taking

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_1 \\ \sigma_2 \end{pmatrix}$$

and

$$\nabla u = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{pmatrix},$$

and substitute both of these expressions into equation (4). As a result, we achieve the following equivalence:

$$\begin{pmatrix} \sigma_1 \\ \sigma_2 \end{pmatrix} = \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{pmatrix}.$$

Thereafter, utilising the aforementioned equivalences, let us consider equation (3). Applying the definition of divergence where required, we can formulate the following equivalence:

$$\begin{aligned} -div\,(\boldsymbol{\sigma}) = -div\,(\nabla u) = -div \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{pmatrix} &= -\left(\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x} \right) + \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial y} \right) \right) \\ &= -\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = f. \end{aligned}$$

Indeed, the second line above is equivalent to equation (7), demonstrating that equations (3)-(5) are entirely equivalent to equations (1) and (2).

Now that we have established the equivalence of our sets of equations, we can proceed to describe Loula's proposed method based on this set of three equations. However, before we begin the initial step of deriving the variational formulation, let us introduce some standard notation and relevant function spaces that will be used throughout the report.

2.2 Introducing Standard Notation and Function Spaces

To start with, let $L^2(\Omega)$ be the space of square integrable scalar real-valued functions defined on Ω:

$$L^2(\Omega) = \{f : \Omega \to \mathbb{R} : \int_\Omega \left|f^2\right| d\Omega < +\infty\}. \tag{8}$$

The $L^2(\Omega)$ inner product is defined as

$$(f, g) = \int_\Omega fg \, d\Omega \qquad \forall f, g \in L^2(\Omega), \tag{9}$$

along with the norm of $L^2(\Omega)$, used for analysis purposes in Section 5, which is defined as

$$\|f\|_{L^2(\Omega)} = (f, f)^{\frac{1}{2}} \qquad \forall f \in L^2(\Omega). \tag{10}$$

Now, let us define two subspaces of $L^2(\Omega)$ which will be crucial for specifying the space in which u will exist. Firstly, we define $H^1(\Omega) \subset L^2(\Omega)$ as:

$$H^1(\Omega) = \{f \in L^2(\Omega) : \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \in L^2(\Omega)\}. \tag{11}$$

Furthermore, let us define $H_0^1(\Omega) \subset H^1(\Omega)$ as

$$H_0^1(\Omega) = \{f \in H^1(\Omega) : f = 0 \text{ on } \Gamma\}, \tag{12}$$

and we will denote this space as V (i.e. $V = H_0^1(\Omega)$).

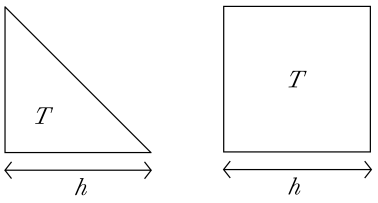
The above spaces serve as the foundation for the spaces in which our solutions will be defined. However, before proceeding with further space definitions, let us introduce a discrete mesh.

To discretise the domain Ω, we will introduce a mesh with elements of size h. Depending on the chosen element geometry, h will denote different lengths. Refer to figure 2 to observe how h will be defined on a triangle and square element, respectively. Let T denote the domain of a single element in Ω, either a triangle or square.

We want to construct a space of piecewise polynomials that are continuous both within and at the boundaries between the elements of our domain. The function space also needs to be built such that a triangle or square domain mesh can be used. Let us introduce a function space, S_h^k, that, by definition, will meet our criteria. S_h^k is the subset of $L^2(\Omega)$ consisting of piecewise polynomials of degree k (i.e. each polynomial is separately defined within each element of the mesh) and are continuous at the mesh boundaries i.e. $S_h^k \subset C^0(\Omega)$.

Let any polynomial of this form be denoted as $p \in S_h^k$. We should now take p and restrict it in the following manner:

$$p_{|T} \in \begin{cases} \mathbb{P}_k & \text{if T is a triangle} \\ \mathbb{Q}_k & \text{if T is a square} \end{cases}$$



(a) The size of h on triangle element T.

(b) The size of h on square element T.

Figure 2: Sizes of elements, T.

where $\mathbb{P}_k$ and $\mathbb{Q}_k$ are spaces of polynomials of degree $\leq k$.

Now, taking the intersection between V and S_h^k, gives us

$$S_h^k \cap V = V_h. \tag{13}$$

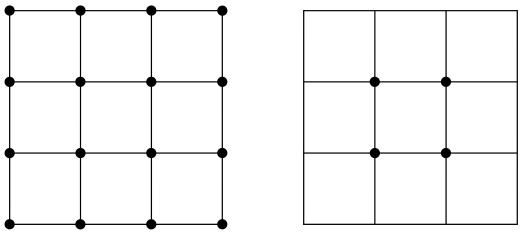
Our finite element solution u_h is given as a linear combination of the basis functions in this space i.e. $u_h \in V_h$. This intersection ensures that all the functions of V_h are piecewise polynomial and so constructible by the FEM.

Finally, let us define the space in which our solution, $\boldsymbol{\sigma}$, will be defined. X_h is a finite element subspace defined as:

$$X_h = S_h^l \times S_h^l.$$

In other words, X_h is the C^0 Lagrangian space of vectorial piecewise polynomials of degree l.

The dimension of function spaces V_h and X_h relate to the number of points used in the mesh chosen to divide up and cover the given domain. Let us consider the explicit example illustrated in figure 3. In figure 3(a), $\mathbb{Q}^1$ elements are defined for the S_h^1 function space on the domain Ω, while in figure 3(b), they are defined for the V_h functions on the same domain Ω (i.e. $V \cap S_h^1$).



(a) Nodes for S_h^1.

(b) Nodes for $V_h (= V \cap S_h^1)$.

Figure 3: Nodes for forming the bases of the function spaces (a) S_h^1, and (b) $V_h (= V \cap S_h^1)$ on Ω.

Given that V is the space where functions are, by definition, equal to 0 on the boundary, to determine the dimension of the space V_h, we must remove all boundary nodes. Thus, despite originating from same domain, the dimension of the function space depicted in figure 3(a) is greater, totalling 16, compared to the dimension of the function space in figure 3(b), which encompasses only the 4 internal nodes.

We can generalise this by stating that when the same degrees of polynomials l and k are used, the dimensions of any arbitrary domain satisfy:

$$dim V_h < dim S_h^l.$$

2.2.1 Constructing a Polynomial Basis for V_h and X_h

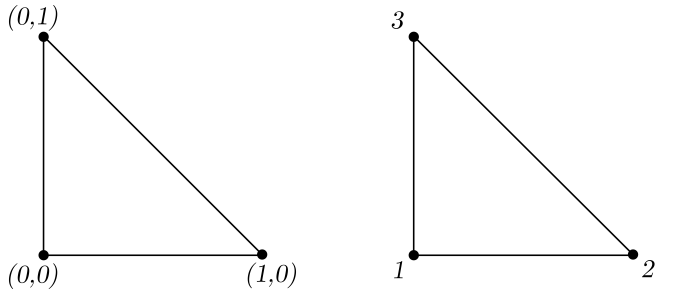
Given that both S_h^l (the component of X_h that constitutes a finite dimensional space) and V_h are finite dimensional spaces, we can assign each space a basis. Let the basis of S_h^l (or X_h) be denoted $\{\varphi_i\}_{i=1,\dots,N}$, and the basis of V_h be denoted $\{\psi_i\}_{i=1,\dots,M}$.

We can describe the set of basis functions of V_h or X_h as a collection of polynomial functions that have been constructed on a reference element, spanning our space and adhering to the criteria outlined above. The polynomial degree of the basis functions for V_h and X_h are k and l, respectively. Given the definitions we have established for V_h and X_h, it follows that the set of basis functions for any polynomial degree will be consistent across both spaces. Consequently, the method for constructing these basis functions will be the same. For the remainder of this section, we will analyse a general set of basis functions as $\{\varphi_i\}_{i=1,\dots,N}$.

Note. For visualisation purposes, throughout this section we will consider an explicit example and illustrate the points made using figures. Let us choose the simplest example: the functions constituting the polynomial basis for $\mathbb{P}^1$ polynomials on the reference triangle.

First, let us examine the reference triangle, which is a right-angled triangle comprising of vertices with Cartesian coordinates: $(0,0)$, $(1,0)$, and $(0,1)$, as illustrated in Figure 4(a). In our example, for a degree one polynomial basis, we select 3 nodes on the reference triangle, numbered in a particular manner. The numbering can be visualised in figure 4(b).

Note that the numbering of the nodes is crucial, as it corresponds to the indexing of our basis functions $\varphi_i(x,y)$. We define basis functions $\varphi_i(x,y)_{i=1,\dots,N}$, where N is the number of nodes on the reference triangle and hence also the number of coefficients to be determined in the general form of the polynomial being used. In other words, the number of basis functions we will have will be equal to the number of nodes on the reference triangle, which depends on the degree of polynomial we are using. To determine the basis functions corresponding to each polynomial degree we use



(a) The Cartesian coordinates of the nodes on the reference triangle.

(b) The node numbering convention used when constructing degree-one polynomial basis functions, $\mathbb{P}^1$.

Figure 4: The reference triangle Cartesian coordinates and $\mathbb{P}^1$ node numbering convention.

Lagrangian polynomials and the Kronecker delta function. Specifically, we define:

$$\varphi_i(x_i, y_j) = \delta_{ij} = \begin{cases} 0, & \text{for } i \neq j \\ 1, & \text{for } i = j. \end{cases} \tag{14}$$

In simpler terms, each basis function will be zero at all nodes except a single node where it will have the value one. This approach allows us to compute a basis on the reference triangle.

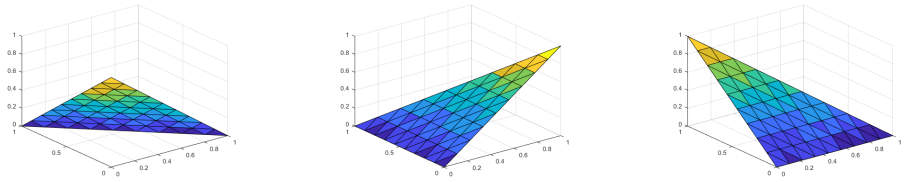
Let us defer computational details until later and present the basis functions for our example (details on implementing these computations in MATLAB will be highlighted in section 4.1). We determine that the basis of first-degree polynomials on the reference triangle, denoted as $\mathbb{P}^1$, is given by:

$$\{\varphi_i(x, y)\}_{i=1,\dots,3} = \begin{cases} \varphi_1(x, y) = -x - y + 1, \\ \varphi_2(x, y) = x, \\ \varphi_3(x, y) = y, \end{cases} \tag{15}$$

and these can be visualised on a series of three-dimensional plots, as seen in figure 5.

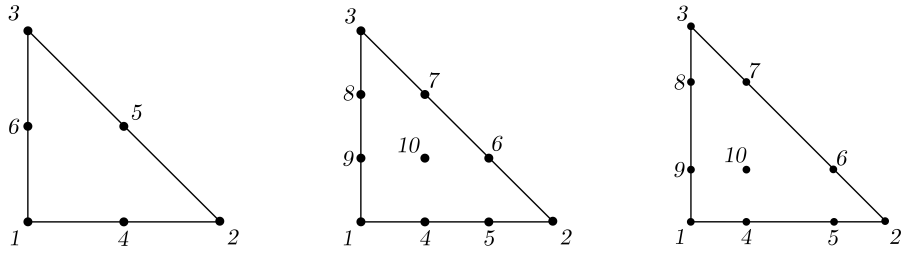
By employing the same methodology, polynomial bases, $\mathbb{P}^2$ and $\mathbb{P}^3$, on the reference triangle can also be computed. The number of nodes and their positioning for each of these polynomial bases on the reference triangle is illustrated in figure 6(a) and 6(b), respectively.

Remark. We will also consider an alternative method of positioning our nodes for a degree three polynomial construction. The nodes will be assigned according to Gauss-Lobatto's quadrature theory [5], and we will denote this



(a) $\varphi_1(x, y)$ (b) $\varphi_2(x, y)$. (c) $\varphi_3(x, y)$.

Figure 5: Visualisation of $\{\varphi_i(x, y)\}_{i=1,\ldots,3}$ for $\mathbb{P}^1$. The (x, y) plane is horizontal and the value of $\varphi_i(x, y)$ is the coloured vertical height above the plane.



(a) $\mathbb{P}^2$ (b) $\mathbb{P}^3$ (c) $\mathbb{P}^3$gl

Figure 6: The node numbering convention used when constructing the polynomial bases: (a) $\mathbb{P}^2$, (b) $\mathbb{P}^3$ and (c) $\mathbb{P}^3$gl.

basis as $\mathbb{P}^3$gl. This node assignment scheme has been illustrated in figure 6(c).

By configuring the placement and numbering of nodes accordingly, we can similarly construct polynomial bases $\mathbb{Q}^1$, $\mathbb{Q}^2$, $\mathbb{Q}^3$, and $\mathbb{Q}^3$gl on a reference square, where the reference square is defined with vertices at $(-1, -1)$, $(1, -1)$, $(1, 1)$, and $(-1, 1)$, as illustrated in figure 7.

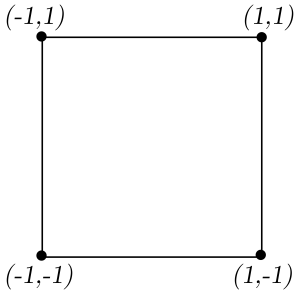
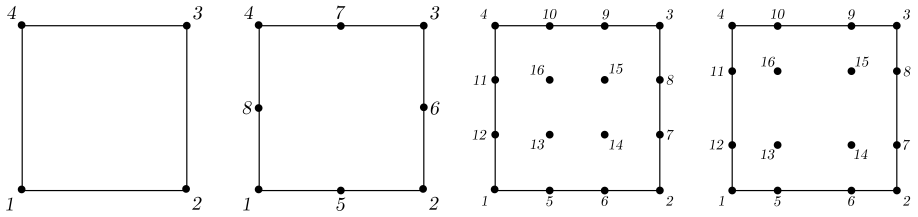


Figure 7: The Cartesian coordinates of the nodes on the reference square.

The node schemes for each of these square bases have been illustrated in figure 8(a)-(d). Our MATLAB implementation will accommodate both

types of element geometries.



(a) $\mathbb{Q}^1$ (b) $\mathbb{Q}^2$ (c) $\mathbb{Q}^3$ (d) $\mathbb{Q}^3$gl

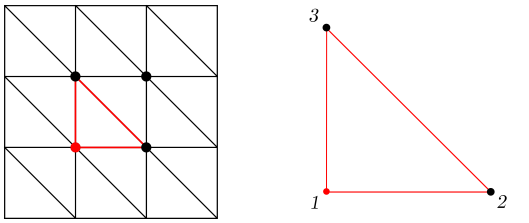
Figure 8: The node numbering convention used when constructing the polynomial bases: (a) $\mathbb{Q}^1$, (b) $\mathbb{Q}^2$, (c) $\mathbb{Q}^3$ and (d) $\mathbb{Q}^3$gl.

2.2.2 Extending Basis Functions from a Reference Element to the Whole Mesh

In the above section, our focus was solely on a reference element and how to construct basis functions on this reference element. Now, we need to extend our understanding to define these functions on a whole mesh.

The simplest way to explain this concept will be to go through a visual step by step example. Let us use the same domain upon which we defined V_h on, in figure 3(a), but now we will construct it so that the mesh covering the domain is made up of arbitrary triangular elements. See figure 9(a) for this illustration. We want to consider how we can define basis functions on this example mesh, using the basis functions we defined on the reference triangle in section 2.2.1.

First, consider the red node and three red lines connected to each other creating the red triangle in figure 9(a). We will be building a set of basis functions for the red node.

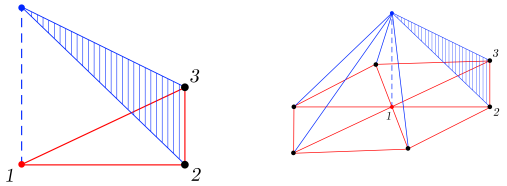


(a) Nodes for V_h on domain Ω, now divided into triangular elements.

(b) Single triangular element extracted from the mesh in (a).

Figure 9: Extracting an arbitrary element from a mesh on Ω.

The red triangle of figure 9(a) is also depicted separately in figure 9(b). This is not the reference triangle, but if we number the the vertices in a similar way to that in the reference triangle, we can create a map from the reference triangle to this triangle within our mesh. Let us assume that the set of three basis functions $\varphi_i(x,y)_{i=1,\dots,3}$ we defined on the reference triangle T, are mapped to this new triangle which we will denote $\hat{T}$. These new basis functions will be denoted as $\hat{\varphi}_i(x,y)_{i=1,\dots,3}$. We can subsequently determine functions similar to those in figure 5, with these new, $\hat{\varphi}_i(x,y)$, basis functions. Using the same node numbering as in figure 9(b), we can illustrate a plot of a mapped basis function relating to the red node in figure 10(a).



(a) A sketch of a single mapped basis function.

(b) A sketch of all six mapped basis functions.

Figure 10: A three-dimensional sketch of (a) a mapped basis function, and (b) all the mapped basis functions, for all the elements containing the red node in figure 9(b).

Now, considering the other five elements surrounding the red node in figure 9(a), we can formulate similar functions for each of these elements. Such that we have six separate functions that can be combined together and we can visualise this on a plot in figure 10(b).

Thus, we have built a function around the red node. We have defined functions on the six triangles surrounding our red node, as depicted in figure 10(b), but since we want a continuous function on the whole domain, and we have defined the edges of the six triangle elements to equal 0, we can extend by 0 across all the other elements on the rest of the domain. This will produce the desired continuous function on the entire domain.

Similar sets of basis functions can be constructed for all of the other interior points in the mesh in figure 9(a).

We have now defined all the necessary spaces and properties of our spaces to proceed with deriving the theoretical framework required for Loula's method. For more details on the finite element method in the context we have just discussed, refer to to [6].

2.3 Deriving the Variational Formulation

With all the subspaces and their corresponding bases that we will use throughout this report now suitably defined, we can proceed through the derivation of Loula's proposed method. We will first loosely describe the steps of the derivation, and then more carefully navigate through a rigorous derivation.

Note that we will now be working wholly in the discrete finite element subspaces V_h and X_h. Let us start by constructing a weak formulation [7] of the constitutive equation (4). Then, in order to force equilibrium in a stronger sense, Loula proposes the introduction of a residual term.

Let us consider the two equations: the constitutive equation (4) and the balance equation (3). The constitutive equation appears sufficient to compute $\boldsymbol{\sigma}$ as it seems to be implying that given u, to compute $\boldsymbol{\sigma}$ you just take the gradient of u. However, we should note that this is only one of the two equations that originate from equation (1) when rewritten equivalently. When we rewrite equation (1), we not only have $\boldsymbol{\sigma} = \nabla u$, but we also have the constraint that $\boldsymbol{\sigma}$ must satisfy the divergence condition embedded in the balancing equation (3) i.e. $-div\,\boldsymbol{\sigma} = f$.

Thus, if we solely focus on the constitutive equation (4), we essentially overlook the fact that the balance equation (3) must also be satisfied. Therefore, once we derive the weak formulation of the constitutive equation, we need to incorporate the constraint that the divergence of $\boldsymbol{\sigma}$ must equal $-f$. This can be achieved by constructing a residual term and combining it with the above weak formulation, and by doing so, we enforce equilibrium in a stronger sense.

This residual term will be constructed in a similar way to how the weak form of the constitutive equation is derived in the previous step, essentially forming a weak formulation of the balance equation (3). However, this term will subsequently be penalised with a carefully chosen coefficient, and this is because this term is only included to subsidise the information derived from the weak form of the constitutive equation. Further explanation regarding the residual term will be included later in this section as it is easier to explain once we have started the explicit derivation.

Now, let us begin the explicit derivation by constructing a weak formulation of the constitutive equation (4). Firstly, let us rearrange the discretised version of (4) to equal zero, i.e.:

$$\boldsymbol{\sigma}_h - \nabla u_h = \mathbf{0}.$$

Next, let us introduce an arbitrary function $\boldsymbol{\tau}_h = \boldsymbol{\tau}_h(x, y)$ from X_h by taking the dot product with both sides of the equation, then integrating over the domain Ω. Thus, we obtain:

$$\int_\Omega (\boldsymbol{\sigma}_h - \nabla u_h) \cdot \boldsymbol{\tau}_h \, d\Omega = \int_\Omega \mathbf{0} \cdot \boldsymbol{\tau}_h \, d\Omega, \quad \forall \boldsymbol{\tau}_h \in X_h.$$

Finally, recognising that the right-hand side is equal to zero, we can then use the definition of our inner product (9) to establish our weak formulation:

$$\int_{\Omega} (\boldsymbol{\sigma}_h - \nabla u_h) \cdot \boldsymbol{\tau}_h \, d\Omega = 0, \quad \forall \boldsymbol{\tau}_h \in X_h$$

$$\implies (\boldsymbol{\sigma}_h - \nabla u_h, \boldsymbol{\tau}_h) = 0, \quad \forall \boldsymbol{\tau}_h \in X_h. \tag{16}$$

Note. The left hand side of both equations above are equivalent, but we choose to express the integral as the inner product form for the sake of tidiness in our final variational formulation, i.e. we will write the derivation of the final result as a combination of inner product forms.

We can now proceed by constructing the weak formulation of our balance equation (3). Similar to the previous construction, we introduce an arbitrary $\boldsymbol{\tau}_h \in X_h$ and derive:

$$\int_{\Omega} (div\, \boldsymbol{\sigma}_h + f) \cdot div\, \boldsymbol{\tau}_h \, d\Omega = \int_{\Omega} 0 \cdot div\, \boldsymbol{\tau} \, d\Omega, \quad \forall \boldsymbol{\tau}_h \in X_h,$$

and using inner product notation, we can express the weak formulation as:

$$(div\, \boldsymbol{\sigma}_h + f, div\, \boldsymbol{\tau}_h) = 0, \quad \forall \boldsymbol{\tau}_h \in X_h. \tag{17}$$

Now, we need to combine the weak formulations of the constitutive and balance equations. In essence, we are dealing with a linear combination of the equations.

At the discrete level, it's not correct to assume that $\boldsymbol{\sigma}$ is exactly equal to the gradient of u. Errors arise due to the approximation of $\boldsymbol{\sigma}$ and u on the mesh being used.

As h tends to 0, theoretically this formulation should converge to the exact $\boldsymbol{\sigma}$, as u_h converges to the exact solution, thereby yielding the exact gradient of u. Therefore, to ensure this is the case, by construction we will pre-multiply the residual term by a coefficient incorporating h, to ensure that as h approaches 0, this term diminishes in significance. Penalizing the weak formulation (17) with a term that tends towards zero ensures the desired outcome is achieved, while still considering this type of information.

Hence, let us multiply both sides of the weak formulation (17) by $(\delta h)^{\alpha}$,

$$(\delta h)^{\alpha}(div\, \boldsymbol{\sigma}_h + f, div\, \boldsymbol{\tau}_h) = 0, \quad \forall \boldsymbol{\tau}_h \in X_h, \tag{18}$$

thus constructing a consistent correction term, i.e. the residual term, where δ and α are positive real parameters.

In conclusion, considering all the derivations above, we combine the weak formulation of equation (4) with the residual term of equation (3) to reformulate our problem as follows. Given $u_h \in V_h$, find $\boldsymbol{\sigma}_h \in X_h$ such that the following is satisfied:

$$(\boldsymbol{\sigma}_h - \nabla u_h, \boldsymbol{\tau}_h) + (\delta h)^{\alpha}(div\, \boldsymbol{\sigma}_h + f, div\, \boldsymbol{\tau}_h) = 0 \quad \forall \boldsymbol{\tau}_h \in X_h. \tag{19}$$

Using basic properties of the inner product, let us expand the brackets and rearrange so that we have only terms involving $\boldsymbol{\sigma}$ on the left-hand side. Hence, the updated problem statement is as follows: Given $u_h \in V_h$, find $\boldsymbol{\sigma}_h \in X_h$ such that the following holds:

$$(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) + (\delta h)^\alpha (div\, \boldsymbol{\sigma}_h, div\, \boldsymbol{\tau}_h) = (\nabla u_h, \boldsymbol{\tau}_h) - (\delta h)^\alpha (f, div\, \boldsymbol{\tau}_h) \quad \forall \boldsymbol{\tau} \in X_h. \tag{20}$$

Remark. A potential extension to Loula's work involves multiplying ∇u by a 2×2 matrix, κ, which could be used to describe physical properties of the problem we introduced in the first paragraph of our introduction. An example of such physical properties may include the permeability of the porous medium mentioned earlier.

The original problem would be adjusted such that $-div\,(k\nabla u) = f$, and we could proceed with a similar construction of a variational formulation arriving at the equivalent problem. Given $u_h \in V_h$, find $\boldsymbol{\sigma}_h \in X_h$ such that the following holds:

$$(\boldsymbol{\sigma}_h, \boldsymbol{\tau}_h) + (\delta h)^\alpha (div\, \boldsymbol{\sigma}_h, div\, \boldsymbol{\tau}_h) = (\kappa \nabla u_h, \boldsymbol{\tau}_h) - (\delta h)^\alpha (f, div\, \boldsymbol{\tau}_h) \quad \forall \boldsymbol{\tau} \in X_h. \tag{21}$$

In the forthcoming section, we will conduct the derivations of the algebraic counterpart to this equation, incorporating κ. However, this aspect has not been integrated into the numerical implementation and testing in MATLAB thus far.

3 Algebraic Formulation

In this section, we will derive an algebraic formulation corresponding to each of the four components of equation (21), considering the explicit definitions of $\boldsymbol{\sigma}_h$, $\boldsymbol{\tau}_h$, ∇u_h and f. We Once the algebraic formulations have been laid out, we will present the resulting linear equation, which can be readily solved for $\boldsymbol{\sigma}_h$.

Note. To simplify notation, we will omit the subscript h throughout the remainder of Section 3. However, it is essential to note that all variables previously defined as discrete will retain their discrete nature post-derivation. Everything discussed in this section must be understood within the context of the discrete setting.

3.1 Algebraic Representation of Terms in the Variational Formulation

Each term of equation (21): $(\boldsymbol{\sigma}, \boldsymbol{\tau})$, $(div\, \boldsymbol{\sigma},\, div\, \boldsymbol{\tau})$, $(\kappa \nabla u, \boldsymbol{\tau})$ and $(f, div\, \boldsymbol{\tau})$, can be formulated as a product of a matrix with integral elements, and a vector containing real coefficients. While the derivation steps for each term are similar, there are some important differences noted where necessary.

The derivation of the algebraic formulation is precisely what we need to implement Loula's method in MATLAB later in 4. Therefore, the manner in which we conducted these derivations was specifically to ensure a clear definition of each matrix. These clear definitions serve as the foundation for effectively and accurately implementing our method in section 4.

3.1.1 Deriving the Algebraic Form of $(\boldsymbol{\sigma}, \boldsymbol{\tau})$

Starting with the first term of equation (21), $(\boldsymbol{\sigma}, \boldsymbol{\tau})$, we will go through a step-by-step formulation of the corresponding algebraic formulation.

Note. Although we have presented the variational formulation as a combination of terms in inner product form, it is actually much clearer to derive algebraic formulations from the integral form of these terms. Hence, the initial step of each derivation will be to express the inner product term as an explicit integral expression.

Using the inner product definition provided in definition 9, let us express $(\boldsymbol{\sigma},\, \boldsymbol{\tau})$ equivalently as:

$$(\boldsymbol{\sigma},\, \boldsymbol{\tau}) = \int_{\Omega} \boldsymbol{\sigma} \cdot \boldsymbol{\tau}\, d\Omega. \tag{22}$$

As previously mentioned, we are exclusively working in two-dimensions, so we can assume, without loss of generality, that $\boldsymbol{\sigma} = (\sigma_1, \sigma_2)^T$ and $\boldsymbol{\tau} = (\tau_1, \tau_2)^T$. Substituting the two-dimensional vector form of $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ into equation (22), we obtain:

$$\int_{\Omega} \begin{pmatrix} \sigma_1 \\ \sigma_2 \end{pmatrix} \cdot \begin{pmatrix} \tau_1 \\ \tau_2 \end{pmatrix} d\Omega = \int_{\Omega} (\sigma_1\tau_1 + \sigma_2\tau_2)\, d\Omega. \tag{23}$$

For brevity, we will only derive the formulation of the integral

$$\int_{\Omega} \sigma_1\tau_1\, d\Omega, \tag{24}$$

and note that the formulation for $\int_\Omega \sigma_2\tau_2\, d\Omega$ can be obtained in an identical manner. The two terms will subsequently be summed together to form our algebraic formulation of $(\boldsymbol{\sigma}, \boldsymbol{\tau})$.

Working with the integral (24) we can start manipulating the terms to construct an explicit form of the global matrix. Since $\sigma_1 = \sigma_1(x, y) \in X_h$, σ_1 can be expressed as a linear combination of basis functions, such that:

$$\sigma_1 = \sum_{j=1}^{N} \sigma_{1j}\varphi_j. \tag{25}$$

Here $\sigma_{1j} \in \mathbb{R}$ are the unknown coefficients we are looking to approximate, and $\varphi_j = \varphi_j(x, y) \in X_h$ are the global basis functions introduced in section 2.2.1.

As $\boldsymbol{\tau} \in X_h$ (and therefore $\tau_1 \in X_h$), is any arbitrary function in X_h, we can choose τ_1 to be equal to any function from the same function space. We will choose that $\tau_1 = \varphi_i(x, y) \in X_h$, referring to the same basis of functions for X_h introduced in section 2.2.1. Substituting the linear combination, (25), into our integral term, (24), and also introducing our chosen τ_1 functions, we find an expression for our integral to be:

$$\int_{\Omega} (\sum_{j=1}^{N} \sigma_{1j}\varphi_j)\varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N.$$

Let us now use standard integral manipulation techniques, and the fact that $\sigma_{1j} \in \mathbb{R}$ are constants. We can move the sum and σ_{1j} outside the integral to obtain:

$$\sum_{j=1}^{N} \sigma_{1j} \int_{\Omega} \varphi_j\varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N. \tag{26}$$

Upon inspection of this expression, it becomes evident that we can represent it as the result of multiplying a matrix, with elements $\int_\Omega \varphi_j\varphi_i \, d\Omega$, by a vector containing real coefficients, σ_{1j}.

Remark. It is essential to emphasize that the indices i and j adhere to the conventional matrix notation, i.e. i will represent the row index, and j will represent the column index. Special attention should be given to organising the matrix properly to ensure that φ_i and φ_j are correctly positioned.

Given that both indices i and j range from 1 to N, we will need to formulate an $N \times N$ matrix. We will denote this matrix as $M_m^{(1)}$ and its elements will take the form:

$$m_{m_{ij}}^{(1)} = \int_{\Omega} \varphi_j\varphi_i \, d\Omega, \qquad \forall i, j = 1, \ldots, N \tag{27}$$

such that the matrix $M_m^{(1)}$ will appear as:

$$M_m^{(1)} = \begin{bmatrix} \int_\Omega \varphi_1\varphi_1 \, d\Omega & \int_\Omega \varphi_2\varphi_1 \, d\Omega & \cdots & \int_\Omega \varphi_N\varphi_1 \, d\Omega \\ \int_\Omega \varphi_1\varphi_2 \, d\Omega & \int_\Omega \varphi_2\varphi_2 \, d\Omega & & \vdots \\ \vdots & & \ddots & \\ \int_\Omega \varphi_1\varphi_N \, d\Omega & \cdots & & \int_\Omega \varphi_N\varphi_N \, d\Omega \end{bmatrix}. \tag{28}$$

The σ_{1j} coefficients will form a vector of size $N \times 1$, denoted as $\boldsymbol{\sigma}_1$:

$$\boldsymbol{\sigma}_1 = \begin{bmatrix} \sigma_{11} \\ \sigma_{12} \\ \cdots \\ \sigma_{1N} \end{bmatrix}. \tag{29}$$

Note. $\boldsymbol{\sigma}_1 \neq \sigma_1(x, y)$. The vector of coefficients is named $\boldsymbol{\sigma}_1$ to prevent confusion later in the report. It important to note that $\boldsymbol{\sigma}_1$ is distinct from the non-boldface σ_1.

Multiplying (28) by (29), we can express the algebraic form of $\int_\Omega \sigma_1 \tau_1 \, d\Omega$ as:

$$M_m^{(1)} \boldsymbol{\sigma}_1. \tag{30}$$

We can proceed, analogously to the approach outlined above, to formulate the algebraic representation of $\int_\Omega \sigma_2 \tau_2, d\Omega$. The $N \times 1$ vector of unknown coefficients, $\boldsymbol{\sigma}_2$, can be introduced in a similar manner as in the previous derivation, representing σ_2 as a linear combination of basis functions. This results in the following algebraic formulation:

$$M_m^{(2)} \boldsymbol{\sigma}_2. \tag{31}$$

Furthermore, since φ_i and φ_j have been chosen from the same function space, X_h, in both formulations, by comparing the elements of $M_m^{(1)}$ and $M_m^{(2)}$, it becomes evident that they are identical. Therefore, we can deduce that $M_m^{(1)} = M_m^{(2)}$ and hence, we will denote $M_m^{(1)} = M_m = M_m^{(2)}$.

Remark. Further to the earlier remark, it's worth noting that while the order of indices i and j isn't critical in this case due to the symmetry of M_m ($M_m = M_m^\mathsf{T}$), for non-symmetric matrices, the positioning of our integral elements is essential. This will become apparent in later formulations.

In conclusion, we have derived and determined the algebraic formulation of $(\boldsymbol{\sigma}, \boldsymbol{\tau})$, comprising of two components: an $N \times N$ matrix of known definite integrals and two $N \times 1$ vectors of unknowns, $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$.

3.1.2 Deriving the Algebraic Form of $(div\, \boldsymbol{\sigma}, div\, \boldsymbol{\tau})$

Now let us consider the second term of equation (21), $(div\, \boldsymbol{\sigma}, div\, \boldsymbol{\tau})$. The formulation will follow similar steps to that of $(\boldsymbol{\sigma}, \boldsymbol{\tau})$. However, one key difference is that $div, \boldsymbol{\sigma}$ and $div, \boldsymbol{\tau}$ are scalars, while $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ were vectors. This difference arises due to the application of the divergence operator, as defined in definition 2.

Using definition 2, and assuming again that $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$ are both two-dimensional vectors, we can express $div\, \boldsymbol{\sigma}$ as:

$$div\, \boldsymbol{\sigma} = \frac{\partial \sigma_1}{\partial x} + \frac{\partial \sigma_2}{\partial y},$$

and similarly express $div\,\boldsymbol{\tau}$ as:

$$div\,\boldsymbol{\tau} = \frac{\partial \tau_1}{\partial x} + \frac{\partial \tau_2}{\partial y}.$$

Let us start by converting the inner product form to an explicit integral from. Therefore, we can express $(div\,\boldsymbol{\sigma}, div\,\boldsymbol{\tau})$ as:

$$(div\,\boldsymbol{\sigma}, div\,\boldsymbol{\tau}) = \int_{\Omega} div\,\boldsymbol{\sigma} \cdot div\,\boldsymbol{\tau}\, d\Omega = \int_{\Omega} (\frac{\partial \sigma_1}{\partial x} + \frac{\partial \sigma_2}{\partial y}) \cdot (\frac{\partial \tau_1}{\partial x} + \frac{\partial \tau_2}{\partial y})\, d\Omega$$

$$= \int_{\Omega} (\frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x} + \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_2}{\partial y} + \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_1}{\partial x} + \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_2}{\partial y})\, d\Omega. \tag{32}$$

Similar to the previous derivation, we will first focus on the first term of the above integral, $\int_{\Omega} \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x}\, d\Omega$. The other terms can be derived in a similar way and then subsequently summed together.

We express σ_1 as the same linear combination as in (25), and once again, we choose τ_1 such that $\tau_1 = \varphi_i(x, y)$. Substituting these equivalent expressions for σ_1 and τ_1 into $\int_{\Omega} \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x}\, d\Omega$, and performing similar integral manipulations as we did to derive (26), we obtain the following:

$$\int_{\Omega} \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x}\, d\Omega = \int_{\Omega} \frac{\partial (\sum_{j=1}^{N} \sigma_{1j}\varphi_j)}{\partial x}\frac{\partial \varphi_i}{\partial x}\, d\Omega$$

$$= \sum_{j=1}^{N} \sigma_{1j} \int_{\Omega} \frac{\partial \varphi_j}{\partial x}\frac{\partial \varphi_i}{\partial x}\, d\Omega \qquad \forall i = 1, \ldots, N. \tag{33}$$

Equation (33) closely resembles equation (26), with the only difference being that both basis functions, φ_i and φ_j, have been partially differentiated with respect to x.

Let us denote the corresponding $N \times N$ matrix as M_{xx} (the subscript xx denotes each function being differentiated with respect to x, and similar nomenclature will be used for further matrices). The $N \times 1$ vector is the same $\boldsymbol{\sigma}_1$ as in equation (29). The elements of M_{xx} can be expressed as:

$$m_{xx_{ij}} = \int_{\Omega} \frac{\partial \varphi_j}{\partial x}\frac{\partial \varphi_i}{\partial x}\, d\Omega \qquad \forall i, j = 1, \ldots, N, \tag{34}$$

and hence, $\int_{\Omega} \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x}\, d\Omega$ can be formulated in algebraic form as:

$$M_{xx}\boldsymbol{\sigma}_1. \tag{35}$$

To derive the algebraic form of the remaining three terms in equation (32) we will follow the same steps as above. The formulation of the $N \times N$ matrix

corresponding to $\int_\Omega \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_2}{\partial y}, d\Omega$ closely resembles that of $\int_\Omega \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_1}{\partial x}, d\Omega$, with the only distinction lying in the differentiation of each φ_i and φ_j with respect to y rather than x. We will denote this matrix as M_{yy}, and its elements can be expressed as follows:

$$m_{yy_{ij}} = \int_\Omega \frac{\partial \varphi_j}{\partial y}\frac{\partial \varphi_i}{\partial y}\, d\Omega \qquad \forall i, j = 1, \ldots, N \tag{36}$$

The algebraic form of $\int_\Omega \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_2}{\partial y}\, d\Omega$ is:

$$M_{yy}\boldsymbol{\sigma}_2. \tag{37}$$

To derive the algebraic form of the remaining two terms of (32), we can adopt the same procedure as above. Ensuring to carefully match the subscript indices of σ_1, τ_2, and σ_2, τ_1, respectively, and correctly partial differentiating each function with respect to the matching variable, we find the corresponding $N \times N$ matrix, M_{xy}, for $\int_\Omega \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_2}{\partial y}, d\Omega$ has elements:

$$m_{xy_{ij}} = \int_\Omega \frac{\partial \varphi_j}{\partial x}\frac{\partial \varphi_i}{\partial y}\, d\Omega \qquad \forall i, j = 1, \ldots, N, \tag{38}$$

and the $N \times N$ matrix, M_{yx}, corresponding to $\int_\Omega \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_1}{\partial x}, d\Omega$ consists of elements:

$$m_{yx_{ij}} = \int_\Omega \frac{\partial \varphi_j}{\partial y}\frac{\partial \varphi_i}{\partial x}\, d\Omega \qquad \forall i, j = 1, \ldots, N. \tag{39}$$

Remark. If the indices i and j in (39) are swapped, we find:

$$m_{yx_{ji}} = \int_\Omega \frac{\partial \varphi_i}{\partial y}\frac{\partial \varphi_j}{\partial x}\, d\Omega = \int_\Omega \frac{\partial \varphi_j}{\partial x}\frac{\partial \varphi_i}{\partial y}\, d\Omega = m_{xy_{ij}}$$

concluding $m_{yx_{ji}} = m_{xy_{ij}}$ and hence, $M_{yx} = M_{xy}^{\mathsf{T}}$.

Matching the corresponding vector of coefficients with each of the above matrices, we find the algebraic form of $\int_\Omega \frac{\partial \sigma_1}{\partial x}\frac{\partial \tau_2}{\partial y}\, d\Omega$ is:

$$M_{xy}\boldsymbol{\sigma}_1, \tag{40}$$

and the algebraic form of $\int_\Omega \frac{\partial \sigma_2}{\partial y}\frac{\partial \tau_1}{\partial x}\, d\Omega$ is:

$$M_{xy}^{\mathsf{T}}\boldsymbol{\sigma}_2. \tag{41}$$

In conclusion, we have found the equivalent algebraic forms for each of the four terms of ($div\,\boldsymbol{\sigma}$, $div\,\boldsymbol{\tau}$). Refer to (35), (37), (40) and (41).

3.1.3 Deriving the Algebraic Form of $(\kappa\nabla u, \boldsymbol{\tau})$

The third term of equation (21), $(\kappa\nabla u, \boldsymbol{\tau})$, involves our solution variable, u, belonging to the function space V_h, as defined earlier in section 2.2. This differs from the previous two derivations, where the components of our inner product term, $\boldsymbol{\sigma}$ and $\boldsymbol{\tau}$, were both defined within X_h. In this section, it's important to note that we'll define basis functions from X_h as φ and from V_h as ψ.

As alluded to in the remark at the end of section 2, the inclusion of a matrix κ preceding ∇u is an extension of Loula's work. Given that we are operating in two dimensions, let's represent κ as the 2×2 matrix:

$$\begin{pmatrix} \kappa_{11} & \kappa_{12} \\ \kappa_{21} & \kappa_{22} \end{pmatrix}.$$

We will assume the elements of κ are constant, denoted as $\kappa_{ij} \in \mathbb{R}$ for $i, j = 1, 2$. Another possible extension of this work would involve exploring the use of non-constant κ_{ij}, where $\kappa_{ij} = \kappa_{ij}(x, y)$.

Proceeding with the derivation of $(\kappa\nabla u, \boldsymbol{\tau})$, let's begin by multiplying the 2×2 matrix κ with the 2×1 vector ∇u to obtain:

$$\kappa\nabla u = \begin{pmatrix} \kappa_{11} & \kappa_{12} \\ \kappa_{21} & \kappa_{22} \end{pmatrix} \begin{pmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{pmatrix} = \begin{pmatrix} \kappa_{11}\frac{\partial u}{\partial x} + \kappa_{12}\frac{\partial u}{\partial y} \\ \kappa_{21}\frac{\partial u}{\partial x} + \kappa_{22}\frac{\partial u}{\partial y} \end{pmatrix}.$$

We will once again express our inner product term as an explicit integral. Given that $\boldsymbol{\tau} = (\tau_1, \tau_2)$, we can evaluate the term $(\kappa\nabla u, \boldsymbol{\tau})$ as:

$$\begin{aligned}
(\kappa\nabla u, \boldsymbol{\tau}) &= \int_\Omega \kappa\nabla u \cdot \boldsymbol{\tau}\, d\Omega = \int_\Omega \begin{pmatrix} \kappa_{11}\frac{\partial u}{\partial x} + \kappa_{12}\frac{\partial u}{\partial y} \\ \kappa_{21}\frac{\partial u}{\partial x} + \kappa_{22}\frac{\partial u}{\partial y} \end{pmatrix} \cdot \begin{pmatrix} \tau_1 \\ \tau_2 \end{pmatrix} d\Omega \\
&= \int_\Omega (\kappa_{11}\frac{\partial u}{\partial x} + \kappa_{12}\frac{\partial u}{\partial y})\tau_1 + (\kappa_{21}\frac{\partial u}{\partial x} + \kappa_{22}\frac{\partial u}{\partial y})\tau_2\, d\Omega \\
&= \int_\Omega \kappa_{11}\frac{\partial u}{\partial x}\tau_1 + \kappa_{12}\frac{\partial u}{\partial y}\tau_1 + \kappa_{21}\frac{\partial u}{\partial x}\tau_2 + \kappa_{22}\frac{\partial u}{\partial y}\tau_2\, d\Omega. \qquad (42)
\end{aligned}$$

Just like in Section 3.1.2, we have a sum of four terms, each of which will require similar derivations. Let's thoroughly derive the algebraic formulation for $\int_\Omega \kappa_{11}\frac{\partial u}{\partial x}\tau_1\, d\Omega$, and subsequently, we'll present the algebraic formulations for the three remaining terms. The derivation of all four terms will follow the same methodology.

Since we have chosen to work with a constant κ_{11}, we can move it outside the integral, resulting in $\kappa_{11}\int_\Omega \frac{\partial u}{\partial x}\tau_1\, d\Omega$. Indeed, if κ_{ij} were not constant, the situation would become more complex. The integral would no longer

be separable from the coefficient, and additional calculations or numerical methods would be necessary to handle such cases.

The provided finite element solution, denoted as u, is expressed as a linear combination of basis functions originating from the function space V_h. As previously mentioned, these basis functions will be designated as ψ, distinguishing them from the basis functions φ belonging to the function space X_h. Hence, the representation of u as a linear combination of basis functions is as follows:

$$u = \sum_{j=1}^{M} u_j \psi_j. \tag{43}$$

Choosing $\tau_1 = \varphi_i(x, y)$ and substituting (43) into $\kappa_{11} \int_\Omega \frac{\partial u}{\partial x} \tau_1, d\Omega$, then rearranging the sum, we arrive at:

$$\kappa_{11} \int_\Omega \frac{\partial u}{\partial x} \tau_1 \, d\Omega = \kappa_{11} \int_\Omega \frac{\partial (\sum_{j=1}^{M} u_j \psi_j)}{\partial x} \varphi_i \, d\Omega$$

$$= \kappa_{11} \sum_{j=1}^{M} u_j \int_\Omega \frac{\partial \psi_j}{\partial x} \varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N. \tag{44}$$

Once again, it's evident that the above term can be expressed as a combination of a vector and a matrix. Therefore, we can construct an algebraic form similar to that in the previous two subsections. However, it's important to note that the indices i and j affect this expression slightly differently compared to previous derivations.

Remark. In the previous two subsections, index i was assigned from 1 to N, where N represents the number of nodes (and thus the number of basis functions) used to define the basis of our function space X_h. However now, when choosing our basis functions for ψ_j from the function space V_h, we could be working with a different degree of polynomial approximation, resulting in a different number of nodes required to define our basis functions. We use M to denote the number of basis functions, $\psi_j \in V_h$, and hence the range of our index j.

The value of N and M affects the sizes of both our matrix containing definite integrals and the coefficient vector. Specifically, the matrix will have dimensions $N \times M$, and the vector will have dimensions $M \times 1$.

Denote the $M \times 1$ vector as $\boldsymbol{u}$, which stores the coefficient values of the provided finite element solution u. In this specific derivation, let us denote the $N \times M$ matrix as M_{0X}. The elements of M_{0X} will be:

$$m_{0X_{ij}} = \int_\Omega \frac{\partial \psi_j}{\partial x} \varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N, \quad \forall j = 1, \ldots, M. \tag{45}$$

Thus, the multiplication of κ_{11}, M_{0X}, and $\boldsymbol{u}$, in sequence, constructs the algebraic representation of $\kappa_{11} \int_\Omega \frac{\partial u}{\partial x} \tau_1 \, d\Omega$:

$$\kappa_{11} M_{0X} \boldsymbol{u}. \tag{46}$$

When processing the third term of (42), $\kappa_{21} \int_\Omega \frac{\partial u}{\partial x} \tau_2, d\Omega$, we derive an equation nearly identical to (44), with the only distinction being the constant coefficient (note that τ_2 can be chosen similarly to τ_1, rendering this difference insignificant in the algebraic representation):

$$\kappa_{21} \int_\Omega \frac{\partial u}{\partial x} \tau_2 \, d\Omega = \kappa_{21} \sum_{j=1}^{M} u_j \int_\Omega \frac{\partial \psi_j}{\partial x} \varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N,$$

and thus, the algebraic form of $\kappa_{21} \int_\Omega \frac{\partial u}{\partial x} \tau_2 \, d\Omega$ is:

$$\kappa_{21} M_{0X} \boldsymbol{u}. \tag{47}$$

Like the examples in previous subsections, the only difference in deriving the algebraic form for the remaining integral terms lies in the variable with respect to which u is differentiated. This applies to the remaining terms: $\kappa_{12} \int_\Omega \frac{\partial u}{\partial y} \tau_1 \, d\Omega$ and $\kappa_{22} \int_\Omega \frac{\partial u}{\partial y} \tau_2 \, d\Omega$, which share the same matrix in algebraic form. Denote the matrix as M_{0Y}, and thus it can be easily demonstrated that the integral elements of the corresponding algebraic matrix are:

$$m_{0Y_{ij}} = \int_\Omega \frac{\partial \psi_j}{\partial y} \varphi_i \, d\Omega \qquad \forall i = 1, \ldots, N, \quad \forall j = 1, \ldots, M. \tag{48}$$

By individually multiplying $M_{0Y} \boldsymbol{u}$ by κ_{12} and κ_{22} successively, we obtain the algebraic expressions for $\kappa_{12} \int_\Omega \frac{\partial u}{\partial y} \tau_1 \, d\Omega$ and $\kappa_{22} \int_\Omega \frac{\partial u}{\partial y} \tau_2 \, d\Omega$ as follows:

$$\kappa_{12} M_{0Y} \boldsymbol{u}, \tag{49}$$

and:

$$\kappa_{22} M_{0Y} \boldsymbol{u}, \tag{50}$$

respectively.

3.1.4 Deriving the Algebraic Form of $(f, div\, \boldsymbol{\tau})$

Combining all of the knowledge gained from the previous three derivations, we can readily derive the algebraic formulation of the final term, $(f, div\, \boldsymbol{\tau})$.

Converting our inner product term to integral form and applying the definition of the divergence operator, we find:

$$(f, div\, \boldsymbol{\tau}) = \int_\Omega f \cdot div\, \boldsymbol{\tau} \, d\Omega = \int_\Omega f (\frac{\partial \tau_1}{\partial x} + \frac{\partial \tau_2}{\partial y}) \, d\Omega \tag{51}$$

Let us focus on just the first term, $\int_\Omega f \frac{\partial \tau_1}{\partial x}\, d\Omega$. To begin, we choose to approximate f as a linear combination of basis functions from function space V_h, thus yielding:

$$f \approx \sum_{j=1}^{M} f_j \psi_j. \tag{52}$$

Choosing $\tau_1 = \varphi_i(x, y)$, and substituting the linear combination for f into $\int_\Omega f \frac{\partial \tau_1}{\partial x}, d\Omega$, we obtain the following integral form:

$$\int_\Omega f \frac{\partial \tau_1}{\partial x}\, d\Omega \approx \int_\Omega (\sum_{j=1}^{M} f_j \psi_j) \frac{\partial \varphi_i}{\partial x}\, d\Omega \approx \sum_{j=1}^{M} f_j \int_\Omega \psi_j \frac{\partial \varphi_i}{\partial x}\, d\Omega \qquad \forall i = 1, \ldots, N. \tag{53}$$

For the algebraic formulation of (53), denote the vector of coefficients as $\mathbf{f}$ and the $N \times M$ matrix as M_{X0}. The elements of M_{X0} will be:

$$m_{X0,ij} = \int_\Omega \psi_j \frac{\partial \varphi_i}{\partial x}\, d\Omega \qquad \forall i = 1, \ldots, N, \quad \forall j = 1, \ldots, M, \tag{54}$$

ensuring that the indices i and j are assigned carefully.

As a result, we arrive at the algebraic formulation of $\int_\Omega f \frac{\partial \tau_1}{\partial x}, d\Omega$ as:

$$M_{X0}\boldsymbol{f}. \tag{55}$$

Following the same derivation steps for $\int_\Omega f \frac{\partial \tau_2}{\partial y}\, d\Omega$, swapping the variable with which we partially differentiate φ_j, we find the algebraic form to be:

$$M_{Y0}\boldsymbol{f}, \tag{56}$$

where the elements of M_{Y0} are:

$$m_{Y0,ij} = \int_\Omega \psi_j \frac{\partial \varphi_i}{\partial y}\, d\Omega \qquad \forall i = 1, \ldots, N, \quad \forall j = 1, \ldots, M. \tag{57}$$

Remark. As we established in section 2.2.1, by construction, any arbitrary set of basis functions, whether it be $\{\varphi_i(x, y)\}_{i=1,\ldots,N}$ or $\{\psi_j(x, y)\}_{i=1,\ldots,M}$, for $N = M$, we have:

$$\varphi_i(x, y) = \psi_j(x, y) \quad \forall\, i = j.$$

Remark. Making a further remark, let us conduct a similar examination to the remark at the bottom of Section 3.1.2, i.e.swapping the indices of the elements of the four new matrices. Interchanging i and j (with both indices still ranging from 1 to N and 1 to M, respectively) in the elements of M_{0X}, we find the following:

$$m_{0X_{ji}} = \int_\Omega \frac{\partial \psi_i}{\partial x} \varphi_j\, d\Omega = \int_\Omega \frac{\partial \varphi_i}{\partial x} \psi_j\, d\Omega = m_{X0_{ij}},$$

for all $i = 1, \ldots, N$ and $j = 1, \ldots, M$. The interpretation of this expression is that the matrix M_{0X} defined with bases of degree k and l, respectively, will be equal to the matrix M_{X0}^{T} defined with bases of degree l and k, respectively. In essence, each of M_{0X} and M_{X0} corresponding to a reverse combination of polynomial bases used will be equal to the transpose of the other one in the pair. We find a similar property for reverse combinations of polynomial bases when constructing M_{0Y} and M_{Y0}.

3.2 Assembling the Algebraic Formulation

In Section 3.1 we derived all the necessary components to build a linear system of equations in matrix form, equivalent to equation (21). Carefully substituting the algebraic formulations into (21) we obtain:

$$\begin{aligned} &\begin{pmatrix} M_m & \mathbf{0} \\ \mathbf{0} & M_m \end{pmatrix} \begin{pmatrix} \boldsymbol{\sigma}_1 \\ \boldsymbol{\sigma}_2 \end{pmatrix} + (\delta h)^\alpha \begin{pmatrix} M_{xx} & M_{xy}^{\mathsf{T}} \\ M_{xy} & M_{yy} \end{pmatrix} \begin{pmatrix} \boldsymbol{\sigma}_1 \\ \boldsymbol{\sigma}_2 \end{pmatrix} \\ &\quad = \begin{pmatrix} \kappa_{11} M_{0X} + \kappa_{12} M_{0Y} \\ \kappa_{21} M_{0X} + \kappa_{22} M_{0Y} \end{pmatrix} \boldsymbol{u} - (\delta h)^\alpha \begin{pmatrix} M_{X0} \\ M_{Y0} \end{pmatrix} \boldsymbol{f}, \end{aligned} \tag{58}$$

which can then be solved using a direct method to find $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2)$, and hence achieve our goal of finding $\boldsymbol{\sigma}$.

Remark. Note that the $\mathbf{0}$ found in both the top right and bottom left of the first matrix in equation (58), are $N \times N$ zero matrices. As a result, we find $\begin{pmatrix} M_m & \mathbf{0} \\ \mathbf{0} & M_m \end{pmatrix}$ is a $2N \times 2N$ matrix, matching the dimension of $\begin{pmatrix} M_{xx} & M_{xy}^{\mathsf{T}} \\ M_{xy} & M_{yy} \end{pmatrix}$. It can also be checked, by multiplying out the matrices and vectors, that both the left and right hand-side have a dimension of $2N \times 1$, which matches the dimension of our aim, $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2)$.

With all the essential theory and derivations laid out, we can now proceed to implement a solver for the linear system (58) in MATLAB and test the numerical results along with their convergence rates.

4 Numerical Implementation using MATLAB

Section 4 marks the initial point in this report where we will begin to explore how the theory derived in sections 2 and 3, and hence also Loula's method, can be implemented within the MATLAB environment. We will outline the steps required to develop codes for constructing all of the matrices necessary for solving the linear system (58).

This section will begin by expanding on the construction of basis functions introduced in section 2.2.1, and outlining an implementation built to construct these bases in MATLAB. Next, we will analyse a rounding error that was discovered in previous attempted implementations and explore

how the `sym` and `syms` functions can be used to eradicate this error. Finally, after detailing the method for constructing all matrices necessary to build our system (58), we will conclude by outlining our plans for testing and analysing our implementation.

4.1 Constructing Basis Functions in MATLAB

In section 2.2.1, it was mentioned that we would elaborate on the details for constructing a polynomial basis of functions. Taking the same example we used in section 2.2.1, let us outline the process for constructing the basis of first-degree polynomials on the reference triangle, (15).

Starting with the Kronecker delta function, defined in (14), we can formulate a system of equations as follows:

$$\begin{cases} \varphi_1(x_1, y_1) = 1, \varphi_1(x_2, y_2) = 0, \varphi_1(x_3, y_3) = 0, \\ \varphi_2(x_1, y_1) = 0, \varphi_2(x_2, y_2) = 1, \varphi_2(x_3, y_3) = 0, \\ \varphi_3(x_1, y_1) = 0, \varphi_3(x_2, y_2) = 0, \varphi_3(x_3, y_3) = 1. \end{cases} \tag{59}$$

Now, let us substitute the corresponding x and y coordinates of the three nodes defined on the reference triangle (as illustrated in figure 4(a)), into (59), yielding:

$$\begin{cases} \varphi_1(0, 0) = 1, \varphi_1(1, 0) = 0, \varphi_1(0, 1) = 0, \\ \varphi_2(0, 0) = 0, \varphi_2(1, 0) = 1, \varphi_2(0, 1) = 0, \\ \varphi_3(0, 0) = 0, \varphi_3(1, 0) = 0, \varphi_3(0, 1) = 1. \end{cases} \tag{60}$$

Note that all polynomials of degree one, and thus all our basis functions, φ_i, follow the general form:

$$\varphi_i(x, y) = c_1 x + c_2 y + c_3, \tag{61}$$

where c_1, c_2 and c_3 represent the coefficients of a basis function to be determined. Let us introduce the notation $c_{i,j}$ and define these as the the ith coefficient of the jth basis function. We can use this notation in (61) for each value of i and j, and then substitute into the system of equations (60), resulting in the following system:

$$\begin{cases} c_{3,1} = 1, c_{1,1} + c_{3,1} = 0, c_{2,1} + c_{3,1} = 0, \\ c_{3,2} = 0, c_{1,2} + c_{3,2} = 1, c_{2,2} + c_{3,2} = 0, \\ c_{3,3} = 0, c_{1,3} + c_{3,3} = 0, c_{2,3} + c_{3,3} = 1. \end{cases} \tag{62}$$

Let us define a 3×3 matrix c to store the coefficient elements c_{ij}, where each c_{ij} is positioned in the ith row and jth column. Then, let us define another 3×3 matrix A constructed such that Ac yields the left-hand side of the system (62). Finally, let b be the right-hand side of the system (62),

which is simply a 3×3 identity matrix. Hence, with these three matrices, we can express system (62) equivalently as the matrix equation $Ac = b$:

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \tag{63}$$

which in turn can be solved using a direct method in MATLAB. We find that

$$c = \begin{pmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

in this example. Assigning the coefficients to each φ_i accordingly, we achieve the set of basis functions presented in (15).

Each of the matrix components A, b and c can easily be constructed in MATLAB. The MATLAB function that constructs these functions is provided in listing 1.

```
% Basis functions expressed as: phi(xi,yi) = c1*xi + c2*yi + c3
% Numbering of the nodes:
% 1 - (0,0), 2 - (1,0), 3 - (0,1).

xi = [0 1 0];
yi = [0 0 1];

A = zeros(3,3);
for i = 1:3
 A(i,:) = [xi(i), yi(i), 1];
end

b = eye(3,3);

c = A\b;

syms x y
phi = sym(zeros(3,1));
for j = 1:3
 phi(j,1) = sym(c(1,j),'r')*x + ...
            sym(c(2,j),'r')*y + ...
            sym(c(3,j),'r');
end
```

Listing 1: Section of `basisfunctionsP1.m` [8] constructing the $\mathbb{P}^1$ basis functions φ_i on the reference triangle.

4.2 Constructing Algebraic Formulation in MATLAB

Let us split the construction of the eight separate matrices in our linear equation, (58), into two subsections: those on the left-hand side and those on the right-hand side. Elements of the left-hand side matrices are constructed entirely from a product of two basis polynomials of equal degree, l, from the set of basis functions $\varphi \in X_h$. Whereas, elements of the right-hand

side matrices are formed by a product of one polynomial from a set of basis functions $\varphi \in X_h$ and another from a set of basis functions $\psi \in V_h$, with polynomial degrees of l and k, respectively.

Previous versions of the code aimed at computing the first four matrices have served as the foundation, in our new version we have refined and introduced enhancements to achieve a higher level of accuracy in computation. The implementation for constructing the second set of four matrices had not been developed previously, and so implementing the second set of matrices is an original contribution to the project. We will indicate, through references or other means, when another person's code is used and when the developed code is an original contribution.

4.2.1 Constructing M_m, M_{xx}, M_{yy} and M_{xy}

In this first subsection, we will solely focus on detailing an explicit MATLAB implementation to compute the first component of the left-hand side of our system, denoted as M_m in equation (58). Refer to section 3.1.1 for the full derivation and details of M_m, but we can simply describe this matrix as an $N \times N$ matrix with elements as expressed in (27).

The easiest way to present the method for computing M_m is to use an explicit example. Let us take the same example we used in sections 2.2.1 and 4.1: the $\mathbb{P}^1$ basis polynomials on the reference triangle and assume that the necessary basis functions, (15), have already been computed. We can then build a matrix by substituting these functions into the expression (27) corresponding to each element of the matrix. Consequently, we can explicitly write the 3×3 matrix, M_m as:

$$\begin{bmatrix} \int_T (-x-y+1)(-x-y+1)\,dT & \int_T x(-x-y+1)\,dT & \int_T y(-x-y+1)\,dT \\ \int_T (-x-y+1)x\,dT & \int_T xx\,dT & \int_T yx\,dT \\ \int_T (-x-y+1)y\,dT & \int_T xy\,dT & \int_T yy\,dT \end{bmatrix},$$

where T is the reference triangle here, i.e.

$$\int_T dT = \int_0^1 \int_0^{-x+1} dy\,dx.$$

Remark. Here, integration is specified over T instead of Ω. This adjustment links back to the discussion in section 2.2.2, where we extended our understanding to define basis functions across the entire mesh. Let us expand further.

We break down our domain, Ω, into a mesh of elements (either triangular or quadrilateral). Then, the integral over the domain becomes the sum of integrals over each element. As depicted in figure 10, our construction ensures

that functions on all elements are well-defined, facilitating straightforward integration as they are polynomials.

We recognise that each element requires integration of the same type of function. We also recognise that every element is different from the reference element just by the coordinates. Hence, for integrating elements within a mesh, we consistently refer to the reference triangle. This approach enables us to integrate once on the reference triangle, then compute integrals for all other elements by adjusting coordinates.

Let us move on finally, by evaluating the definite integrals, we construct the exact M_m for $\mathbb{P}^1$ polynomials on the reference triangle. The remaining section of the MATLAB function `basisfunctionsP1.m` [8], that computes these matrices has been printed in listing 2.

```
1  % A, b and c have already been computed appropriately above
2
3  syms x y
4  phi = sym(zeros(3,1));
5  dphidx = sym(zeros(3,1)); % partial derivative of phi w.r.t. x
6  dphidy = sym(zeros(3,1)); % partial derivative of phi w.r.t. y
7
8  for j = 1:3
9   phi(j,1) = sym(c(1,j),'r')*x + ...
10             sym(c(2,j),'r')*y + ...
11             sym(c(3,j),'r'); % assigning coefficients from c
12   dphidx(j,1) = diff(phi(j,1),x);
13   dphidy(j,1) = diff(phi(j,1),y); % using 'diff' to compute partials
14  end
15
16  symMASS  = sym(zeros(3,3)); % M_m
17  symDERXX = sym(zeros(3,3)); % M_xx
18  symDERYY = sym(zeros(3,3)); % M_yy
19  symDERXY = sym(zeros(3,3)); % M_xy
20  for i = 1:3
21   for j = 1:3
22    symMASS(i,j)  = int(int(phi(i,1)*phi(j,1),y,0,-x+1),x,0,1);
23    symDERXX(i,j) = int(int(dphidx(i,1)*dphidx(j,1),y,0,-x+1),x,0,1);
24    symDERYY(i,j) = int(int(dphidy(i,1)*dphidy(j,1),y,0,-x+1),x,0,1);
25    symDERXY(i,j) = int(int(dphidx(i,1)*dphidy(j,1),y,0,-x+1),x,0,1);
26   end
27  end
```

Listing 2: Section of `basisfunctionsP1.m` computing the first four matrices of linear equation (58) for $\mathbb{P}^1$ polynomials on the reference triangle.

As illustrated in lines 12 and 13 of listing 2, we can compute the relevant partial derivatives of each basis function and hence simultaneously compute M_{xx}, M_{yy}, and M_{xy} in lines 23-25 of listing 2. For each of these three matrices, refer to (34), (36), and (38), respectively, for the explicit expressions of the elements in each matrix.

We can replicate this process for polynomials of degrees 2 and 3 (including the degree 3 Gauss-Lobatto variant) on the reference triangle. It is important to note that the dimension of our matrices for these polynomial

degrees will be 6×6 and 10×10, respectively, corresponding to the number of nodes (and hence functions) required to build a polynomial basis on the reference triangle.

Let us now begin to introduce an original contribution of this project by highlighting a significant improvement that addresses a noticeable error in the results produced by the previous implementation.

4.2.2 Implementing Symbolic Variables in MATLAB

The code shown in section 4.1 and 4.2.1 was already provided in the finite element library [9]. While this served as a solid foundation, a flaw was observed during an initial review of the outputs produced by the previous code. This presented an opportunity for us to improve this and all subsequent codes.

The error was observed in the output of matrices M_m, M_{xx}, M_{yy} and M_{xy} when working with basis functions of polynomial degree three. Both `basisfunctionsP3.m` [9] and `basisfunctionsP3gl.m` [9] produced similar issues. Although many of the computed elements were accurate, some appeared to be significantly incorrect. Specifically, these problematic elements were fractions with numerators and denominators as integers of magnitude up to 10^{32}. When computing the decimal value of these fractions we would find the output to be a seemingly regular value. For instance, take the element in row 2, column 1 of M_{xx}, computed when using `basisfunctionsP3.m`, which was calculated as:

$$-\frac{851861203353370160620534321119187}{9735556609752801803494680617287680} = -0.0875000\ldots\text{(16 s.f.)} \approx -\frac{7}{80}.$$

This is a clear indication of a minor rounding error at some stage in the code, leading to an anomalous output.

The solution we devised involved the use of symbolic variables, employing the `sym` [10] and `syms` [11] functions in MATLAB. The initial code (printed in listing 2) attempted to incorporate symbolic arrays by introducing symbolic variables `x` and `y` during the construction of the basis functions $\varphi(x, y)$ and the corresponding calculated partial derivatives. However, we discovered that the issue arose before this point and was a result of the coordinates of the nodes being recurring, such as $\frac{1}{3}$ in the case of $\mathbb{P}^3$, or irrational, like $\frac{5-\sqrt{5}}{10}$ in the case of $\mathbb{P}^3$gl.

The error arose because our (x_i, y_i) were initially created as 'double' variables. Consequently, when computing the coefficient matrix, `c`, using a MATLAB matrix solver (e.g., `A\b` or `linsolve(A,b)`), the resulting 'double' matrix would immediately lose precision. This occurs because elements are being stored as decimals, with precision limited to a certain magnitude of significant figures. Due to this initial lack of precision, by the time we

reached the computation of the definite integral elements in our matrices, a noticeable error had accumulated.

After reviewing the MATLAB documentation for symbolic variables (i.e. variables created used `sym` and `syms` functions), we formulated an appropriate solution. This involved creating a symbolic variable for the problematic coordinate and subsequently using this symbolic variable to construct the arrays containing our node coordinates at the initial stage. In simple terms, instead of assigning a decimal value (with limited precision and thus errors would accumulate during subsequent iterative numerical computation) to a variable, we assign a symbolic representation of the number to the variable (which remains the same symbolic representation during computation until the very final stage of evaluation, and so avoiding accumulating arithmetical errors due to lack of precision). Consequently, at each step of the code, the various components we compute will remain in this exact symbolic form.

An example code implementation is provided in listing 3, with comments throughout to explain the steps. This example relates to the $\mathbb{P}^3$gl polynomials on a reference triangle.

```
  sqrt5 = sym(sqrt(5)); % create symbolic variable for sqrt(5)

 % enter the node coordinate values using the symbolic variable:
  xi = [0, 1, 0, (sqrt5-1)/(2*sqrt5), (sqrt5+1)/(2*sqrt5), ...
        (sqrt5+1)/(2*sqrt5), (sqrt5-1)/(2*sqrt5), 0, 0, 1/3];
  yi = [0, 0, 1, 0, 0, (sqrt5-1)/(2*sqrt5), (sqrt5+1)/(2*sqrt5), ...
        (sqrt5+1)/(2*sqrt5), (sqrt5-1)/(2*sqrt5), 1/3];
 % symbolic vectors xi and yi containing the coordinates of nodes

  A = sym(zeros(10,10)); % create a symbolic array of zeroes

  for i = 1:10
   A(i,:) = [xi(i)^3, yi(i)^3, xi(i)^2*yi(i), xi(i)*yi(i)^2, ...
       xi(i)^2, yi(i)^2, xi(i)*yi(i), xi(i), yi(i), 1];
  end

  b = sym(eye(10,10));

  c = linsolve(A,b); % A and b are symbolic so c will also be symbolic


  syms x y % creating symbolic variables x and y (same as previous
    implementation)

  phi = sym(zeros(10,1));
  dphidx = sym(zeros(10,1));
  dphidy = sym(zeros(10,1));

  for j = 1:10
   phi(j,1) = c(1,j)*x^3 + ...
              c(2,j)*y^3 + ...
              c(3,j)*x^2*y + ...
              c(4,j)*x*y^2 + ...
              c(5,j)*x^2 + ...
              c(6,j)*y^2 + ...
              c(7,j)*x*y + ...
              c(8,j)*x + ...
              c(9,j)*y + ...
```

```
38                c(10,j); % we do not need to convert elements of c into
                           symbolic variables as we did in old code
39     dphidx(j,1) = diff(phi(j,1),x);
40     dphidy(j,1) = diff(phi(j,1),y);
41    end
42  % proceed with computing the matrices as normal
```

Listing 3: Section of `basisfunctionsP1_sym.m` [8] constructing the $\mathbb{P}^1$ basis functions φ_i on the reference triangle with symbolic variables.

Note. Introducing a symbolic variable into a regular array (in the case above, substituting `sqrt5 = sym(sqrt(5))` into our coordinate vectors `xi` and `yi`) immediately converts them into a symbolic array.

After adjusting the code as outlined above, we can proceed with the same algorithm to calculate the matrices. The process will mirror the steps highlighted in lines 16-27 of listing 2, (amending appropriately for the correct size matrices, dictated by the polynomial degree). As a consequence of this work, the elements of matrices M_m, M_{xx}, M_{yy}, and M_{xy} will now be in exact symbolic form, ensuring a higher level of accuracy compared to the previous attempts.

All subsequent code for constructing any of the eight matrices has been updated to utilise symbolic variables, improving the precision of computations throughout the system of matrices (58).

4.2.3 Constructing M_{0X}, M_{0Y}, M_{X0} and M_{Y0}

Let us proceed with building and computing the four matrices on the right-hand side of the system (58): M_{0X}, M_{0Y}, M_{X0}, and M_{Y0}. We will outline the general approach for constructing each matrix in MATLAB. Refer to (45), (48), (54), and (57) to see the expressions for the elements of each $N \times M$ matrix, respectively.

As we noted in the theoretical derivation for these four matrices in sections 3.1.3 and 3.1.4, each matrix is constructed by multiplying a combination of basis functions from two different spaces. Therefore, to compute the elements, we need to introduce another set of basis functions into our MATLAB implementation. As indicated in the comment at the beginning of section 3.1.3, this second set of functions in V_h will be denoted ψ_j for $j = 1, \ldots, M$. By construction, we will ensure that the naming convention used in our code aligns with this theoretical framework.

We will implement the construction of the ψ functions following the same structure as that for the φ functions discussed in section 4.1. This means the initial part of our implementation for M_{0X} will include two similar sections: one for constructing the φ_i functions and another for the ψ_j functions. By design, we will first construct φ and then ψ for all matrices.

Note. When the degree of $\varphi(x, y)$ matches the degree of $\psi(x, y)$ (i.e., $l =$

$\deg(\varphi) = \deg(\psi) = k$), the sets of functions will be identical:

$$\varphi_i = \psi_j \text{ for } i = j, \quad \forall i, j = 1, \ldots, M \text{ (or } M \text{ as } M = N).$$

Despite this, in such cases, we will still compute the two sets of functions separately to maintain consistency in the structure across all code implementations for various polynomial degree combinations.

Once we have constructed both sets of basis functions and their partial derivatives, the next step is to evaluate the definite integrals. It is crucial to exercise caution when coding these computations to ensure that the indices align with the theory presented in sections 3.1.3 and 3.1.4. We have made a concerted effort to maintain consistency in variables and index usage with the theory as much as possible. Listing 4 illustrates the final lines of our implementation, which compute and assign the elements of the four matrices (the example is for the combination $l = 2$, $k = 3$).

```
1  for i = 1:6
2   for j = 1:10
3    mat0X(i,j) = int(int(phi(i,1)*dpsidx(j,1),y,0,-x+1),x,0,1);
4    mat0Y(i,j) = int(int(phi(i,1)*dpsidy(j,1),y,0,-x+1),x,0,1);
5    matX0(i,j) = int(int(dphidx(i,1)*psi(j,1),y,0,-x+1),x,0,1);
6    matY0(i,j) = int(int(dphidy(i,1)*psi(j,1),y,0,-x+1),x,0,1);
7   end
8  end
```

Listing 4: Final lines computing the second four matrices of linear equation (58) for $l = 2$, $k = 3$.

Comparing lines 3-6 in listing 4, which are responsible for constructing `mat0X`, `mat0Y`, `matX0`, and `matY0`, with the corresponding theoretical expressions (45), (48), (54), and (57), respectively, we can see that the indices and variables used are consistent with our explicit definitions. We can observe that the dimensions of the matrices created also match our theoretical expectations, i.e., 6×10 ($N \times M$ for $l = 2$, $k = 3$).

In Section 5, we will explore a proposition from Loula's paper concerning the convergence rates of the proposed method for different combinations of polynomials of degree l and k. In accordance with this proposition, when computing the matrices discussed in this section, we will only calculate the following combinations:

$$l = k, \text{ for } k = 1, 2, 3,$$

and for $l \neq k$:

$$\begin{cases} l = k - 1, & \text{for } k = 2, 3, \\ l = k + 1, & \text{for } k = 1, 2. \end{cases}$$

Note. We will also explore combinations involving Gauss-Lobatto numbering. Therefore, wherever we have l or k equal to 3, we will also compute the 3gl variant. This results in a total of 10 combinations that we will construct and test.

We will generate only the essential combinations. As discussed in the remark at the end of section 3.1.4, it is worth noting that we don't need to generate reverse combinations. For instance, we will compute the matrices for $l = 1, k = 2$, but there is no need to compute $l = 2, k = 1$ due to the transposition property.

4.2.4 Constructing Matrices for the Reference Square Geometry

As we alluded to in previous sections, we can also construct similar matrices on the reference square element. The setup to construct our matrices will be very similar, but if T now represents the reference square, bounds of the integral elements would now be:

$$\int_T dT = \int_{-1}^{1}\int_{-1}^{1} dy\, dx.$$

Due to this change in bounds, the final section of our codes evaluating the definite integrals will be slightly different. For instance, see an example line of code in listing 5 that computes the matrix M_m on the reference square. These changes in bounds in MATLAB will be consistent across all eight matrices.

```
1  symMASS(i,j) = int(int(phi(i,1)*phi(j,1),y,-1,1),x,-1,1);
```

Listing 5: Line of code computing the definite integral elements of M_m for an arbitrary polynomial basis φ on the reference square.

4.2.5 Nomenclature of Implementations in MATLAB

We have built separate code implementations for each of our various combinations and we have named these MATLAB files in the following way.

The codes producing M_m, M_{xx}, M_{yy} and M_{xy} are named:

`basisfunctions'T''l'_sym.m`,

where `T` is either `P` or `Q`, corresponding to the reference element used to construct basis functions on, either a triangle or square, respectively, and `l` is the polynomial degree used for constructing φ. This includes the 3gl variant, i.e. `l` can equal `1, 2, 3` or `3gl`.

Similarly, the codes producing M_{0X}, M_{0Y}, M_{X0} and M_{Y0} are named:

`basisfunctions'T''l''T''k'_sym.m`,

where `T` and `l` are the same as before and `k` is defined similarly to `l`, i.e. the polynomial degree used to construct, hence `k` can equal `1, 2, 3` or `3gl`.

4.3 Solving the Linear System

In section 3.2, we detailed how to combine the eight matrices into linear system (58) which could be solved to recover our desired gradient, $\boldsymbol{\sigma}$. With all these matrices now constructed in MATLAB, we can assemble the system and solve it using a built-in MATLAB direct method.

5 Numerical Results and Convergence Study

Now that we've completed the theoretical derivation and implemented the construction of all the necessary components, the next stage of this report will be to evaluate whether Loula's proposed method offers an improvement over an alternative built-in MATLAB method. Section 5 will be structured as follows.

We will begin by introducing Loula's convergence proposition and provide a brief overview of the theory concerning errors and convergence rates to help interpret this proposition. We will outline what we planned to test and the MATLAB method we used to analyse the results, making reference to the code we developed to produce our results.

Next, we will present and analyse the results we obtained, comparing them with the proposition. Finally, we will conclude this section by comparing Loula's method with the in-built MATLAB function `grad`, determining which method is superior and why.

5.1 Loula's Proposition and Convergence Analysis Theory

One of the main characteristics to assess for any method is the rate at which computed solutions converge to the exact solution as we reduce h (and consequently increase the number of elements of our mesh). The following proposition, drawn from Loula's paper, theoretically outlines the convergence rates we should expect to observe during our testing.

5.1.1 Loula's Proposition

Loula's proposition is divided into two parts: the first part considers the case where $\delta = 0$, while the second part encompasses the scenario where $\delta > 0$ and $0 \leq \alpha \leq 2$.

Proposition 1. *(Loula's Convergence Proposition for $\delta = 0$)*

Let $\{u, \boldsymbol{\sigma}\}$ represent the solution of the overarching problem discussed in the report, while $\boldsymbol{\sigma}_h$ denotes the approximate solution obtained from our system, (58)*. Then, for $\delta = 0$, there exists a positive number C, independent of h, such that we have the following result:*

$$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2} \leq C \left(h^{l+1} \|f\| + h^k \|f\| \right).$$

In essence, this implies that the order of convergence rate for the error is equal to $min(l+1, k)$.

One exception is noted: for $k = l = 1$*, super-convergence is numerically observed, i.e.* $\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\| = O(h^2)$.

Note. Throughout section 5, $\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2}$ denotes the error between our computed $\boldsymbol{\sigma}_h$ and the exact $\boldsymbol{\sigma}$ using the L^2 norm, as defined in (10). We will expand on error and convergence rate theory in section 5.1.4, after presenting the remainder of the proposition.

We can explicitly interpret this proposition by creating a table containing the theoretical convergence rate for each combination of k and l. The explicit theoretical results have been printed in table 1.

$\delta = 0$		
k	l	Order of Convergence
1	1	2*
1	2	1
2	1	2
2	2	2
2	3	2
3	2	3
3	3	3

Table 1: Theoretical order of convergence for each polynomial combination with $\delta = 0$. *super-convergence observed with this combination.

Next, let us consider the proposition for values of $\delta > 0$.

Proposition 2. *(Loula's Convergence Proposition for* $\delta > 0$*)*

Considering the same setup as in proposition 1, for $\delta > 0$ *and* $0 \leq \alpha \leq 2$*, there exists a positive number* C*, independent of* h*, such that we have the following result:*

$$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2} \leq C \left(h^{l+\frac{\alpha}{2}} \|f\| + h^{k+1-\frac{\alpha}{2}} \|f\| \right). \tag{64}$$

In essence, this implies that the order of convergence rate for the error is equal to $min(l + \frac{\alpha}{2}, k + 1 - \frac{\alpha}{2})$.

Loula's paper examines two parameter combinations with $\delta > 0$, namely $\delta = 0.1, \alpha = 1$ and $\delta = 1, \alpha = 2$. In order to make direct comparisons, we will also test with these combinations. Let us construct convergence tables 2(a) and 2(b) similar to table 1, containing the explicit rates of convergence that are theorised for each combination of polynomials.

We want to verify if these proposed convergence rates hold for our implemented method, but first, there are a few items and theory that we need to introduce before we can fully interpret these propositions and tables.

(a) $\delta = 0.1, \alpha = 1$			(b) $\delta = 1, \alpha = 2$		
k	l	Order of Convergence	k	l	Order of Convergence
1	1	$\frac{3}{2}$	1	1	1
1	2	$\frac{3}{2}$	1	2	1
2	1	$\frac{3}{2}$	2	1	2
2	2	$\frac{5}{2}$	2	2	2
2	3	$\frac{5}{2}$	2	3	2
3	2	$\frac{5}{2}$	3	2	3
3	3	$\frac{7}{2}$	3	3	3

Table 2: Theoretical order of convergence for each polynomial combination with (a) $\delta = 0.1, \alpha = 1$ and (b) $\delta = 1, \alpha = 2$.

5.1.2 Choosing the Assigned Function f

Before proceeding with the error and convergence theory, let us select the assigned function f that we will use for testing. For testing purposes, it is important to choose a function that can be solved analytically. This will allow us to obtain an exact solution, which is necessary for computing exact differences with our approximation method and, consequently, assessing the quality of our implementation. Loula chose $f = \sin(\pi x)\sin(\pi y)$ which can be solved analytically to obtain the following solution to our problem:

$$u = \frac{1}{2\pi^2}\sin(\pi x)sin(\pi y),$$

$$\boldsymbol{\sigma} = \frac{1}{2\pi}\begin{pmatrix}\cos(\pi x)\sin(\pi y)\\ \sin(\pi x)\cos(\pi y)\end{pmatrix}$$

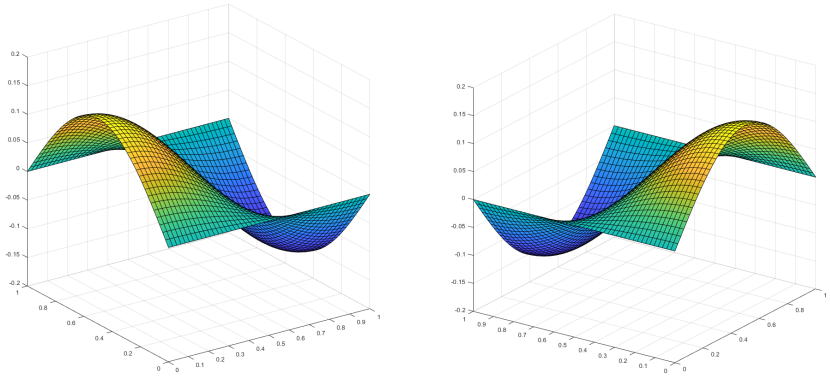
In order to directly compare our results with Loula's, we will also choose this particular f for testing and analysis.

Note. For visualisation purposes let us display this solution on a three-dimensional plot. The individual components $\boldsymbol{\sigma}_1$ and $\boldsymbol{\sigma}_2$ have been depicted in figure 11(a) and (b), respectively.

5.1.3 Testing Combinations

Now that we have designated the function f for testing and error analysis, let us preface the convergence theory by revisiting all the variables and parameters that we can vary during our tests.

Firstly, we can test the various combinations of polynomial degrees we highlighted at the end of section 4.2.3. This alone amounts to 10 combinations.



(a) $\boldsymbol{\sigma}_1$. (b) $\boldsymbol{\sigma}_2$.

Figure 11: The components of our recovered gradient $\boldsymbol{\sigma}$ for the assigned function $f = \sin(\pi x)\sin(\pi y)$, depicted on three-dimensional plots.

Then, for each polynomial combination, we can test any number of different meshes. Let us denote the number of meshes tested as η. Hence, we will define the different values of h for a series of meshes on the $[0,1] \times [0,1]$ domain as:

$$h_i = \left(\frac{1}{2}\right)^i \text{ for } i = 1, \ldots, \eta.$$

Remark. We will conduct all testing on the $[0,1] \times [0,1]$ domain, allowing us to use the formula above to calculate h. Due to computational limitations, we will limit η to 5. However, we expect the forthcoming theory to hold for values greater than 5. We will continue using η during the theoretical stage for generality.

The last variable we can adjust during testing is the combination of the two parameters δ and α in our residual coefficient, $(\delta h)^\alpha$. While there are infinitely many possibilities, we will restrict our testing to the three combinations Loula tested in their paper, as stated in section 5.1.1. However, for generality let us denote the number of parameter combination settings tested as ζ.

As we can see from the above, there will be a considerable number of combinations to test, highlighting the importance of developing a code that can efficiently conduct all these tests. We will introduce this code in section 5.1.5.

Now, let us finally introduce all the convergence analysis theory.

5.1.4 Convergence Analysis Theory

As noted in the remark preceding Loula's proposition, all error calculations will be done using the L^2 norm defined in (10). Throughout the following theory, we will use $\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2}$ to denote the error.

Remember that from our system of equations (58), the solution we recover is $(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2)$, a two-dimensional vector. The method we implemented computes this vector so in order to calculate the error for our objective, $\boldsymbol{\sigma}$, we will calculate the error for each component and combine the values by taking the mean, i.e.:

$$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2} = \frac{\|\boldsymbol{\sigma}_1 - \boldsymbol{\sigma}_{1_h}\|_{L^2} + \|\boldsymbol{\sigma}_2 - \boldsymbol{\sigma}_{2_h}\|_{L^2}}{2}.$$

For each polynomial combination we will compute η approximate solutions and hence η separate error values, one for each of the tested meshes. These values are how we can measure the convergence rate.

Let us illustrate with an explicit example to introduce the values and the theory. Consider the $k = l = 1$ polynomial combination on triangle elements, with $\delta = 0.1$ and $\alpha = 1$. The resulting error values are shown in table 3.

h	$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_h\|_{L^2}$
$\left(\frac{1}{2}\right)^1$	2.9987e-02
$\left(\frac{1}{2}\right)^2$	8.6827e-03
$\left(\frac{1}{2}\right)^3$	2.3702e-03
$\left(\frac{1}{2}\right)^4$	6.3448e-04
$\left(\frac{1}{2}\right)^5$	1.7162e-04

Table 3: Error values for $k = l = 1$ polynomial combination on triangle elements, with $\delta = 0.1$ and $\alpha = 1$.

Using the error values in table 3, we can determine the rate of convergence between each consecutive change of mesh by employing the following formula:

$$q_i = \frac{\log\left(\frac{\left\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_{i+1}}\right\|_{L^2}}{\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2}}\right)}{\log\left(\frac{h_{i+1}}{h_i}\right)} \text{ for } i = 1, \ldots, \eta - 1, \tag{65}$$

where q_i is the rate of convergence between the ith and $i+1$th mesh, and $\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2}$ denotes the L^2 norm error for the ith mesh. In this example, as we are testing consecutive meshes with values of h halving each step, the denominator of our fraction is equal to $\log\left(\frac{1}{2}\right)$ for all i.

Taking this particular example, we can calculate the rates of convergence between each consecutive mesh. These rates are shown in table 4.

h_i	h_{i+1}	$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2}$	$\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_{i+1}}\|_{L^2}$	q_i
$\left(\frac{1}{2}\right)^1$	$\left(\frac{1}{2}\right)^2$	2.9987e-02	8.6827e-03	1.7881
$\left(\frac{1}{2}\right)^2$	$\left(\frac{1}{2}\right)^3$	8.6827e-03	2.3702e-03	1.8731
$\left(\frac{1}{2}\right)^3$	$\left(\frac{1}{2}\right)^4$	2.3702e-03	6.3448e-04	1.9013
$\left(\frac{1}{2}\right)^4$	$\left(\frac{1}{2}\right)^5$	6.3448e-04	1.7162e-04	1.8863

Table 4: Convergence rates between ith and $i+1$th meshes for $k = l = 1$ polynomial combination on triangle elements, with $\delta = 0.1$ and $\alpha = 1$.

We observe from table 4 that the rate of convergence appears to be tending towards 2 as of h decreases. A rate of $q = 2$ suggests quadratic convergence. For future reference, we note that $q = 1$ implies linear convergence, while $q = 3$ or $q = 4$ indicate cubic or quartic convergence, respectively.

5.1.5 MATLAB Code for Comparing Convergence Rates

We have covered all the necessary theory for analysing the convergence rates of our implemented method. Now, let us briefly introduce and explain the code we developed to effectively test and compare convergence rates between different parameter combinations, and also easily change the combination of polynomial bases we employ. The MATLAB function we developed is printed in listing 6.

```
1  polyCombi = 'P1_P1'; % enter polynomial combination
2
3  numberOfMeshes = 5; % number of meshes
4  numberOfSettings = 3; % number of combinations of parameters
5  errorL2 = zeros(numberOfMeshes,numberOfSettings);
6  hSize = zeros(numberOfMeshes,numberOfSettings);
7
8  deltaSettings = [0,0.1,1]; % set the combinations of parameter delta
9  alphaSettings = [1,1,2]; % set the combinations of alpha parameter
10
11 for i = 1:1:numberOfMeshes
12   % load mesh
13   meshElements = 2^i;
14   meshIdentifier = strcat(polyCombi,'_',num2str(meshElements),'_',
       num2str(meshElements));
15   filename = strcat('data/gradient/loula/mesh_',meshIdentifier,'.mat'
       );
16   load(filename);
17   clear filename meshIdentifier;
18
19   for j = 1:1:numberOfSettings
20     % settings for the problem
21     settings.delta     = deltaSettings(j); % value of parameter delta
22     settings.alpha     = alphaSettings(j); % value of parameter alpha
23     settings.plot      = 0;
24     settings.plotExact = 0;
```

```
25      settings.error      = 1;
26
27      % execute main code
28      problemDataFile = @gradientDataLoula;
29      [u,error] = gradientReconstruction (mesh,dof,settings,
        problemDataFile,{});
30
31      hSize(i,j) = 1/meshElements;
32      errorL2(i,j) = mean(error.L2);
33     end
34    end
35
36    errorL2conv = zeros(numberOfMeshes-1,numberOfSettings);
37
38    for i = 1:1:numberOfMeshes-1
39     for j = 1:1:numberOfSettings
40      errorL2conv(i,j) = log(errorL2(i+1,j)/errorL2(i,j))/(log(1/2));
41     end
42    end
```

Listing 6: `executeGradientReconstruction_errorL2.m` [8].

Listing 6 is a MATLAB function that allows us to analyse the convergence rates of our implemented method. Below is a description fo the code:

1. Line 1: This line allows us to select the combination of polynomials, φ and ψ, from the combinations listed at the end of section 4.2.3.

2. Lines 3 and 4: These lines enable us to specify the number of meshes, η, and the number of parameter combinations, ζ, we want to test

3. Lines 8 and 9: Here, `alphaSettings` and `deltaSettings` allow us to explicitly list the different parameter combinations to be tested..

4. The subsequent `for` loop iterates through each mesh, increasing the number of elements by a factor of 2 each time, thereby halving the size of h.

5. Within each mesh iteration, another `for` loop is performed to execute the solver for each of the parameter combinations we wish to test. For each iteration, the code calculates the solution and the L^2 error norm, storing these values in an array `errorL2`. Overall, the solver is executed $\eta \times \zeta$ times for the selected polynomial combination.

6. The final lines of the function compute the rate of convergence between each change of h for all defined parameter combinations.

5.2 Comparing our Results with the Theory

With the code provided in listing 6, we can now begin collecting our results for each of our testing combinations. We outlined in the beginning of section 5.1 a general set of mesh and parameter combinations we planned to test. Let us now specify the exact combinations we will be testing:

- the convergence rates for all previously identified 10 combinations of polynomial degree k and l.
- 5 different meshes on the $[0,1] \times [0,1]$ domain, with $h_i = \frac{1}{2}^i$ for $i = 1, \ldots, 5$.
- both square and triangular elements on the $[0,1] \times [0,1]$ domain, resulting in meshes with $4, 16, 64, 256, 1028$ quadrilateral elements and $8, 32, 128, 512, 2056$ triangular elements, respectively.
- 3 previously identified combination settings of the δ and α parameters.

All results will be presented in the series of tables 5-10, with the various combinations of variables clearly stated. We will now conclude this subsection by analysing these tables.

Note. In some cases, the less refined meshes may not exhibit a specific rate of convergence, and it is only as the mesh is refined that these rates will tend towards a specific value. Therefore, we will calculate the convergence rate between the two most refined meshes and further confirm the value that it appears to tend towards. We will highlight cells of the table in green if they match the proposed theoretical result, red if they do not match, and orange if we cannot determine if they converge to any particular value.

Let us analyse each of the $\mathbb{P}$ and $\mathbb{Q}$ elements together for each of the parameter combinations separately.

$\underline{\delta = 0:}$ Starting with $\delta = 0$, let's first note that when $\delta = 0$, the terms with the residual coefficient in our linear system (58) vanish. According to the theory we discussed in section 2.3, this would suggest that certain constraints are not incorporated into the solution, and we would expect the accuracy to be lower than when $\delta > 0$.

The results for $\delta = 0$ can be found in tables 5 and 6. In general, we find that the results produced by our implemented method match the theoretical results, particularly with the $\mathbb{Q}$ elements. All but one polynomial combination match, and it is interesting to see that our implementation reproduced Loula's 'anomalous' observation for $k = l = 1$, producing a convergence greater than the predicted (noted as super-convergence in Loula's paper).

The one result on $\mathbb{Q}$ elements that does not match the theory, $k = 1, l = 2$, also produced a rate of convergence greater than expected. This might suggest that the super-convergence property is related to polynomial combinations with $k = 1$. This conjecture is supported to some extent by the $\mathbb{P}$ elements, where both combinations with $k = 1$ seem to converge to an order of convergence greater than 1, within the 5 meshes. In fact, these two combinations do not seem to tend towards any 'regular' order of convergence. With more meshes, this might tend towards a particular number, and if so, we would predict them to tend towards 2.

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	2*	1.9982	2
1	2	1	1.9928	2
2	1	2	2.0047	2
2	2	2	1.9925	2
2	3	2	1.9984	2
2	3gl	2	1.9984	2
3	2	3	2.9704	3
3	2	3	3.0144	3
3gl	2	3	2.9704	3
3gl	3gl	3	3.0065	3

Table 5: Observed convergence rates for $\mathbb{Q}$ elements with $\delta = 0$.

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	2*	1.6203	N/A
1	2	1	1.5531	N/A
2	1	2	2.0096	2
2	2	2	1.9837	2
2	3	2	2.0002	2
2	3gl	2	2.0002	2
3	2	3	2.9546	3
3	3	3	3.0928	3
3gl	2	3	2.9538	3
3gl	3gl	3	3.0756	3

Table 6: Observed convergence rates for $\mathbb{P}$ elements with $\delta = 0$.

$\underline{\delta = 0.1, \alpha = 1}$: Moving onto the $\delta = 0.1, \alpha = 1$ parameter combination, this one is more interesting as we include the residual term into our linear system (58), and we can therefore expect equilibrium to be enforced in a stronger sense, i.e., the solutions to be more accurate and converge at a greater rate than $\delta = 0$. As $\alpha = 1$, according to proposition 2, our convergence rates will not be integers. However, we find that, in table 7, for all $\mathbb{Q}$ element combinations, our convergence rates seem to tend towards integers, and integers that are exactly 0.5 greater than the theoretical result. This is not problematic as the exhibited rates are greater than expected, meaning our implementation works better than theorised on $\mathbb{Q}$ elements.

We find similar results for $\mathbb{P}$ in table 8, in that our observed convergence rate exceeds the theoretical for all combinations. However, by the 5th mesh, for 7 out of 10 combinations, the observed rate is closer to the theorised rate compared to the $\mathbb{Q}$ elements, not seeming to tend towards the integer above.

$\underline{\delta = 1, \alpha = 2}$: Finally, let us analyze the results in tables 9 and 10. Once again, we find that the majority of the convergence rates observed here

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	1.5	1.9531	2
1	2	1.5	1.9144	2
2	1	1.5	2.0664	2
2	2	2.5	2.9554	3
2	3	2.5	3.0212	3
2	3gl	2.5	3.0212	3
3	2	2.5	2.9586	3
3	3	3.5	4.0189	4
3gl	2	2.5	2.9581	3
3gl	3gl	3.5	4.0337	4

Table 7: Observed convergence rates for $\mathbb{Q}$ elements with $\delta = 0.1, \alpha = 1$.

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	1.5	1.8863	N/A
1	2	1.5	1.7315	N/A
2	1	1.5	2.0791	2
2	2	2.5	2.6895	N/A
2	3	2.5	3.0351	3
2	3gl	2.5	3.0351	3
3	2	2.5	2.6908	N/A
3	3	3.5	3.8576	N/A
3gl	2	2.5	2.6907	N/A
3gl	3gl	3.5	3.8896	N/A

Table 8: Observed convergence rates for $\mathbb{P}$ elements with $\delta = 0.1, \alpha = 1$.

tend towards integers, with only the rates for $k = 1$ combinations on $\mathbb{P}$ elements not converging to any integer within 5 meshes. Ignoring $k = 1$ combinations, there is an interesting trend observed for both element types where, when $k = l$, the observed rate of convergence is exactly 1 higher than the theoretical rate. All other combinations, i.e., for $k \neq l$, match the theory. It is challenging to determine what has caused this trend, but it is possible something similar to what happened for $k = l = 1$ with $\delta = 0$ may have occurred.

Note. It is evident that across all combinations, employing the Gauss-Lobatto quadrature rule for third-degree polynomials instead of the regular third-degree numbering yields the same convergence rates. Thus, we can conclude that there is no observed advantage to using this alternative node positioning scheme.

In conclusion, we observe that $\mathbb{Q}$ elements tend to exhibit more regular convergence rates, converging to "regular" numbers more quickly than $\mathbb{P}$ elements. However, across all polynomial combinations and parameter settings, and both element types, we consistently find that the convergence

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	1	1.9505	2
1	2	1	1.9163	2
2	1	2	2.0532	2
2	2	2	3.0301	3
2	3	2	2.1755	2
2	3gl	2	2.1755	2
3	2	3	3.2604	3
3	3	3	4.0513	4
3gl	2	3	3.2597	3
3gl	3gl	3	3.9570	4

Table 9: Observed convergence rates for $\mathbb{Q}$ elements with $\delta = 1, \alpha = 2$

k	l	Theoretical Convergence Rate	Observed Convergence Rate Between 4th & 5th Mesh (5 s.f.)	Convergence Rate Prediction
1	1	1	1.7468	N/A
1	2	1	1.5099	N/A
2	1	2	2.1080	2
2	2	2	2.9664	3
2	3	2	2.1033	2
2	3gl	2	2.1033	2
3	2	3	3.0037	3
3	3	3	3.8517	4
3gl	2	3	3.0044	3
3gl	3gl	3	3.9383	4

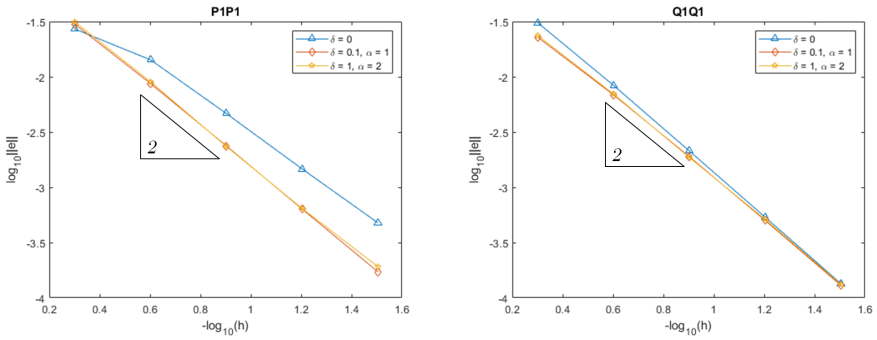
Table 10: Observed convergence rates for $\mathbb{P}$ elements with $\delta = 1, \alpha = 2$

rates observed from our results are greater than or equal to the predicted rates according to propositions 1 and 2. This indicates the success of our implementation.

5.2.1 Illustrating the Influence of Parameter Settings on Convergence Rate

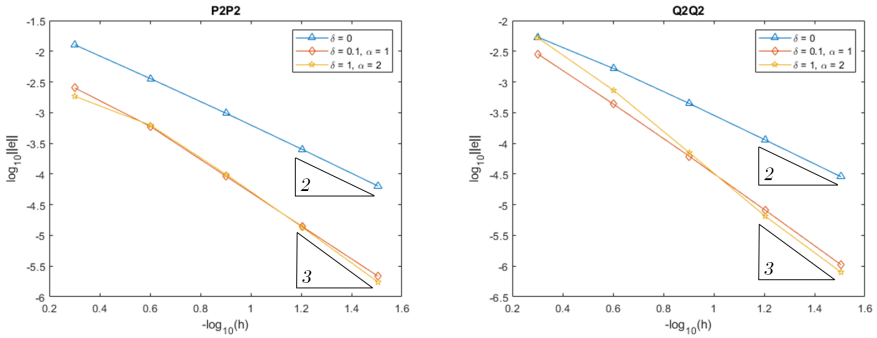
Visualising the convergence rates for each parameter combination by plotting $\log_{10}(\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2})$ against $-\log_{10}(h_i)$ provides an insightful comparison. Figures 12, 13, and 14 depict these plots, with the rate of convergence for each combination clearly shown. These plots will allow us to make observations about the observed rates of convergence between the various parameter combinations.

We noted in previous analysis, but we can now visually observe that for $k = l = 1$, we observe a greater convergence rate for $\delta = 0$ than we would ordinarily expect as suggested by Loula. More interestingly, visualising the results in this way clearly shows that when $\delta > 0$ (i.e., including the residual term in the implemented method), in almost all cases, we observe a greater



(a) Triangle elements. (b) Quadrilateral elements.

Figure 12: Plots of $\log_{10}(\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2})$ against $-\log_{10}(h_i)$ for $k = l = 1$ on (a) triangular elements and (b) quadrilateral elements.



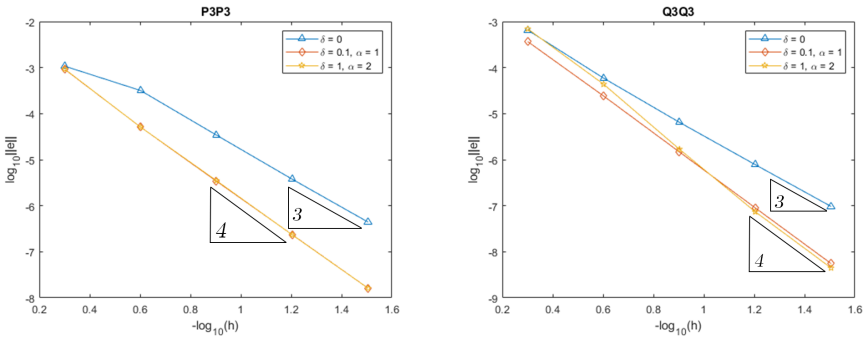
(a) Triangle elements. (b) Quadrilateral elements.

Figure 13: Plots of $\log_{10}(\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2})$ against $-\log_{10}(h_i)$ for $k = l = 2$ on (a) triangular elements and (b) quadrilateral elements.

rate of convergence compared to when $\delta = 0$, where the residual term is not included in the computations. This suggests that by including the residual term, we have enforced equilibrium in a stronger sense, resulting in the method producing better results than a rudimentary method.

5.3 Comparing Loula's Method with In-Built MATLAB Function

As mentioned earlier in Section 5, we will wrap up this section by investigating an in-built MATLAB function capable of computing function gradients. Our exploration of the `gradient` function quickly revealed some major limitations compared to Loula's method. Let us explore the algorithm of



(a) Triangle elements. (b) Quadrilateral elements.

Figure 14: Plots of $\log_{10}(\|\boldsymbol{\sigma} - \boldsymbol{\sigma}_{h_i}\|_{L^2})$ against $-\log_{10}(h_i)$ for $k = l = 3$ on (a) triangular elements and (b) quadrilateral elements.

`gradient` and discuss these limitations.

5.3.1 Understanding the Algorithm of MATLAB's `gradient` Function

The `gradient` function accepts input in the form of a vector, matrix, or multidimensional array, which can resemble a mesh-like structure. It then computes the numerical gradient based on this input and provides the corresponding output.

According to the MATLAB documentation for `gradient` [12], the algorithm operates as follows: It computes the gradient using central differences for interior data points of the input array, while employing a single-sided difference method along the edges. However, during our testing of the convergence rate using this method, we encountered unexpected results. Subsequent investigation revealed a contradiction with the MATLAB documentation.

We discovered that while the MATLAB documentation states that a single-sided difference method is used along the edges, in reality, the `gradient` function employs a central difference method for both interior and exterior points. Specifically, for exterior points, it calculates the gradient using a central difference method by averaging the exterior point with the closest interior point, and then computing gradient values from this midpoint.

5.3.2 Comparison with the `gradient` Function

In accordance with the theory of the central difference method, we anticipated observing quadratic convergence during testing, meaning the convergence rate should be of order 2. After conducting our own tests, we con-

firmed this expectation. We developed `gradient_testing.m` [8] to verify our findings,

Indeed, this comparison highlights the limitations of the `gradient` function compared to our implemented method. While the `gradient` function demonstrates convergence rates of order two, Loula's method exhibited rates of convergence up to order four. Additionally, the `gradient` function is restricted to meshes composed of quadrilateral geometries only, whereas Loula's method can handle meshes with triangular elements and has the potential to be extended to more generic geometries in the future. These factors collectively suggest that Loula's method implemented in MATLAB is superior in versatility and performance.

Remark. It is worth noting that despite the difference in accuracy of solutions, we found that the difference in speed at which MATLAB executes either code was negligible for the meshes we tested. This suggests that while Loula's method may offer superior accuracy, there's no significant sacrifice in computational time taken when compared to using the `gradient` function.

6 Conclusion

This report embarked on an exploration of a specific finite element method proposed by Loula to recover the gradient of u, where u can be determined as the solution to the elliptic boundary value problem in (1) and (2). We started by defining certain finite element function spaces that would be essential for defining our solution. Following the introduction of the polynomial bases for the function spaces, we proceeded to the first piece of original work, deriving the algebraic formulation of the key components of Loula's method.

The implementation of the method in MATLAB constituted the primary original contribution to this project. Also incorporating improvements in the previous version of the code allowed us to improve the level of precision. Using our new code modules, a comparative analysis demonstrated consistency with the accuracy levels of our implementation and the theoretical propositions in Loula's paper. Across all trials, our implemented method consistently demonstrated convergence rates equal to or higher than those theorised.

As a result of all our work on this project, we have developed a MATLAB function that yields results of superior accuracy compared to any built-in function designed for this particular problem. Nonetheless, opportunities for further enhancement and expansion remain. Let us outline a couple of suggested extensions of this project.

By incorporating a matrix κ into equation (1), we produce an interesting expansion of the problem. This matrix κ was included in the theoretical framework in section 3, thereby laying the groundwork for potentially extending our MATLAB implementation to accommodate this additional

variable.

Furthermore, another possibility for extending this project emerges during the construction of the mesh used for computing our finite element solution. While we have successfully implemented quadrilateral and triangular geometries, the framework could be further developed for more complex geometries.

Acknowledgements

I sincerely appreciate the consistent support and valuable feedback provided by Dr. Marco Discacciati, my supervisor, over the past 7 months. He consistently offered guidance, patiently addressed my queries, and helped clarify complex concepts. I'm also thankful to my father for the occasional discussions we had about my project, despite his limited understanding of the subject matter, and for his unwavering support throughout.

References

[1] Pinchover, Y. and Rubinstein, J. (2005) *An Introduction to Partial Differential Equations*. New York: Cambridge University Press, pp. 64-65.

[2] Knabner, P. and Angermann, L. (2003) *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*. New York: Springer-Verlag, Inc., pp. 1-18, 92-113

[3] Loula, A., Rochinha, F. and Murad, M. (1995) 'Higher-order gradient post-processings for second-order elliptic problems', *Computer Methods in Applied Mechanics and Engineering*, 128(1), pp. 361-381.

[4] Alzate, P.P.C., Ve'lez, G.C. and Mesa, F. (2018) 'A Note on the Variational Formulation of PDEs and Solution by Finite Elements', *Advanced Studies in Theoretical Physics*, 12(4), pp. 173-179.

[5] Abramowitz, M. and Stegun, I.A. (1972) *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Tenth Printing. New York: Dover Publications, Inc., pp. 888

[6] Atkinson, K. and Han, W. (2004) *Elementary Numerical Analysis*. Third Edition. New York: John Wiley & Sons, Inc., pp. 451-489

[7] Šolín, P. (2006) *Partial Differential Equations and the Finite Element Method*. New York: John Wiley & Sons, Inc., pp. 14-16

[8] Jarman, J. (2024) *Library of Implemented Codes Relating to this Project*. Available at: `https://github.com/jameslboro/23MAC300-F017951.git` (Accessed: 09 May 2024)

[9] Discacciati, M. (2024) *Finite Element Library*. Available at (*access may need to be approved*): https://lunet-my.sharepoint.com/:f:/r/personal/mamd7_lunet_lboro_ac_uk/Documents/_MarcoDisca/work/teaching/finalYearProjects/2023_2024/James?csf=1&web=1&e=O41Gkf (Accessed: 09 May 2024)

[10] The MathWorks Inc. (2024) *sym - Create symbolic variables, expressions, functions, matrices*. Available at: https://uk.mathworks.com/help/symbolic/sym.html (Accessed: 09 May 2024)

[11] The MathWorks Inc. (2024) *syms - Create symbolic scalar variables and functions, and matrix variables and functions*. Available at: https://uk.mathworks.com/help/symbolic/syms.html (Accessed: 09 May 2024)

[12] The MathWorks Inc. (2024) *gradient - Numerical gradient*. Available at: https://uk.mathworks.com/help/matlab/ref/gradient.html (Accessed: 09 May 2024)