Mimetic Finite Difference Discretization of the Laplace Equation

Ludwig Lahmeyer

November 8, 2016

Abstract

This report explores the construction of Mimetic Finite Difference Methods for the Laplace equation. It details and compares two different methods, namely the Primal and Mixed form methods. As opposed to the reference material that this report is based on, we focus on the 2D case. As such, a large part of this report is focused on the derivation of the discretization scheme. The last section is dedicated to the attached Matlab code that was written using the discretizations and to comparing the pros and cons of the respective methods.

Contents

1 Introduction **3**

2 Derivation of the Discretization Schemes **4**

2.1 Important Notation 4

2.1.1 Polygonal Meshes 4

2.1.2 Edge Normals and Orientation 4

2.1.3 Lebesgue Measure 5

2.1.4 Fields and Spaces 5

2.1.5 The Inner Product 5

2.1.6 Projection Operator 5

2.1.7 Reconstruction Operator 5

2.2 The Mixed Formulation MFDM Discretization Scheme for the Laplace Equation . . . 6

2.3 The MFDM for the Primal Form of the Laplace Equation 9

2.3.1 Derivation Process of $M_{\mathscr{V}}$ 10

2.3.2 Derivation Process of $M_{\mathscr{E}}$ 11

2.3.3 Assembling the Final System 12

3 Matlab Code **14**

4 Experimental Results **17**

Appendices **22**

A Matlab Codes: Mixed Formulation Approach **23**

B Matlab Codes: Primal Form Approach **27**

List of Figures

2.1 Square Polygonal Mesh 4

4.1 Unit Square with exact solution: $u(x, y) = e^x \sin y$ 18
4.2 Unit Circle with exact solution: $u(x, y) = e^x \sin y$ 19
4.3 Exact solution: $u(x, y) = e^x \sin y$, Unit Square 19
4.4 Exact solution: $u(x, y) = e^x \sin y$, Unit Square 20
4.5 Wrench Polygonal Mesh 21
4.6 Circle Polygonal Mesh 21

Chapter 1

Introduction

In the course of trying to understand physical processes we nowadays have several tools at our disposal, one of these being the creation of mathematical models for computer simulations of the process in question. For example when constructing a bridge, it might be useful to construct a mathematical model of the bridge in question first in order to ensure that it can sustain the required amount of strain and will not collapse at the slightest puff of wind. While we will not be dealing with such a complicated example in this report, we will be explaining one of the methods that exists for the construction of such mathematical models, the Mimetic Finite Difference Method (MFDM), and construct a MFDM code in Matlab for the Laplace boundary value problem, which can be applied to gravimetric problems.

$$\begin{aligned} -\Delta p &= b \text{ in } \Omega \\ p &= g^D \text{ on } \Gamma_D \\ \nabla p \cdot \boldsymbol{n} &= g^N \text{ on } \Gamma_N \end{aligned} \tag{1.1}$$

The advantage of the MFDM Discretization method is that it mimics the properties of the model it is applied to, including symmetry, duality, self-adjointness to name a few. It does this by ascertaining that the solution p and its flux $\boldsymbol{u}$ satisfy the Green formula[4]:

$$\int_\Omega \nabla p \boldsymbol{u} dx = -\int_\Omega \text{div} \boldsymbol{u} p dx + \int_{\partial\Omega} pn \cdot u \tag{1.2}$$

.

Flux is the rate of flow through the surface in the form of a 2 dimensional vector.

In the following section we will be describing how to derive a discretization scheme for (1.1) in 2 dimensions using a mixed formulation MFDM approach. We will during the course of this report be working with polygonal meshes, generated by the code provided in [5].

Chapter 2

Derivation of the Discretization Schemes

2.1 Important Notation

2.1.1 Polygonal Meshes

Prior to discretizing we need to define some terms and introduce the meshes we will be working with, an example of such a polygonal mesh can be seen in figure 2.1. A mesh is a way of representing our domain as the union of smaller domains, a very simple example being a rectangular domain being represented as the union of several smaller square domains.

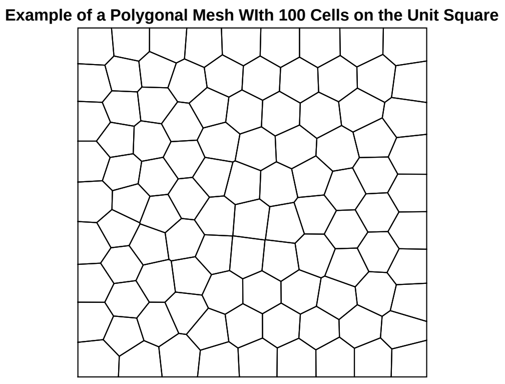
Example of a Polygonal Mesh WIth 100 Cells on the Unit Square

Figure 2.1: Square Polygonal Mesh

This mesh is constructed by the function PolyMesher [5], which takes a given domain (for example the unit square domain in Figure 2.1) and returns a representation of it as the union of smaller, polygonal domains. From this point onwards we will refer to the polygons in our mesh as cells, the nodes of our mesh as vertices and the edges being the lines connecting the vertices. Another important concept that needs introduction is restriction to a cell. When we restrict to a cell we focus solely on the information (vertices, edges, edge centers, barycenters, etc.) that is relevant to the individual cell. For example if we have a polygon U, the edges and vertices restricted to U would be the edges and vertices that define and shape U. The same applies for restriction to edges and vertices. If we restrict our mesh to a certain edge we are only interested in the information relevant to that specific edges, so vertex co-ordinates, center of the edge etc.

2.1.2 Edge Normals and Orientation

Edge normals refer to the two orthonormal unit vectors of an edge in the 2D plane, and for the purpose of the discretization it is important to keep track of which of the two possibilities you are using. This is due to the fact that when we restrict to a cell, we assign values to the orientation of the edge normal, 1 for outward facing normals with respect to the cell and -1 for inward facing normals. Then orientation of the edge normal for edge e, restricted to cell P is referred to as $\alpha_{\mathrm{e,P}}$ in the rest of the report.

We have a similar concept for the orientation of the edges. If we have and edge e that spans vertices v_1 and v_2 we have to decide on the orientation of the edge, does it go from v_1 to v_2 or vice versa. In the attached code the orientation of the edges is decided in the stage where the unique edges are determined. For the Primal Form method we also need to introduce $\beta_{\mathrm{e,P}}$ which denotes whether the edge e restricted to cell P is orientated counter-clockwise: $\beta_{\mathrm{e,P}}$ has the value 1 if it is counter clockwise, -1 if it is clockwise.

2.1.3 Lebesgue Measure

For the rest of this report $|\cdot|$ will be used to denote the Lebesgue measure. Depending on the subset we are taking the Lebesgue measure of, the means of calculating the measure differs. The Lebesgue measure of an edge is defined as its length, whereas the Lebesgue measure of a cell is the area of the relevant cell.

2.1.4 Fields and Spaces

The discretization schemes that we will be working with require that we employ both scalar and vector fields. A scalar/vector field takes certain elements of the mesh and associates scalar/vector variables with the elements.

The spaces we will be working with are the vertex, edge, face and cell spaces. Respectively these spaces are associated with the set of mesh vertices, edges, faces and cells, and are denoted by $\mathscr{V}, \mathscr{E}, \mathscr{F}, \mathscr{P}$. Generically, we will denote them with $\mathscr{I}$.

2.1.5 The Inner Product

$[\cdot,\cdot]_{\mathscr{I}}$ will be used to denote the inner product over the space $\mathscr{I}$. We will define the inner product for the individual spaces as necessary in the discretization sections of the report.

2.1.6 Projection Operator

The projection operator used in the derivation of the discretization schemes will be denoted by $\Pi^{\mathscr{I}}$. It projects the spaces of sufficiently smooth scalar or vector-valued functions into the discrete spaces described above.

It will on occasion be useful to restrict this projection operator to certain elements of the mesh. When the projection operator is restricted, it will be denoted as follows[2, 3.1]:

$$\Pi^{\mathscr{I}}_{Q} : X_{|\mathrm{Q}} \to \mathscr{I}_{h,\mathrm{Q}},$$

where Q is a single mesh element, for example the vertex, edge, face or cell.

2.1.7 Reconstruction Operator

The reconstruction operator is the inverse of the projection operator and is of particular interest in the Primal Method case, as the reconstruction operators for the 2D case differ from the 3D case. This means that instead of being able to employ the reconstruction operators listed in [2], we have to derive our own. It is denoted in the rest of the report as:

$$R^{\mathscr{I}}_{\mathrm{P}} : \mathscr{I}_{h,\mathrm{P}} \to X_{|\mathrm{P}}.$$

Where the space $X_{|\mathrm{P}}$ contains the trial space $\mathscr{T}_{\mathrm{P}}$, which is the space of the trial functions. As in the finite element method, a trial function $\boldsymbol{u}$ is a function that satisfies the weak formulation of our system for all test functions, $\boldsymbol{v} \in X_0$.

$$\int_\Omega \boldsymbol{u} \cdot \boldsymbol{v} dV - \int_\Omega p \text{div} \boldsymbol{v} dV = -\int_{\Gamma_\text{D}} g^D \boldsymbol{v} \cdot \boldsymbol{n} \ \forall \boldsymbol{v} \in X_0 \tag{2.1}$$
$$\int_\Omega q \text{div} \boldsymbol{u} dV = \int_\Omega bq dV \ \forall q \in L^2(\Omega)$$

2.2 The Mixed Formulation MFDM Discretization Scheme for the Laplace Equation

Our first step is to re-write (1) in its mixed form, in order to introduce the variable $\boldsymbol{u}$, the flux of the solution p:

$$\boldsymbol{u} + \nabla p = 0 \text{ in } \Omega \tag{2.2}$$
$$\text{div}\boldsymbol{u} = b \text{ in } \Omega \tag{2.3}$$
$$p = g^D \text{ on } \Gamma_D \tag{2.4}$$
$$u \cdot \boldsymbol{n} = g^N \text{ on } \Gamma_N \tag{2.5}$$

We now start the discretization process by equipping the scalar and vector fields with suitable degrees of freedom[3].

- We denote the space of discrete scalar fields as $\mathscr{P}_h$, with its degrees of freedom being attached to every mesh cell $\text{P} \in \Omega_h$, where Ω is the mesh. The value associated with cell P is stored as q_P in the vector $q_h \in \mathscr{P}_h$.

$$q_h = (q_P)_{P \in \Omega_h}$$

.

- Similarly, the space of discrete flux fields is denoted by $\mathscr{F}_h$, attaching one degree of freedom to each mesh edge $\text{e} \in \mathscr{F}$. We denote the value associated with each mesh edge e as u_e, and store it in the vector $\boldsymbol{u}_h \in \mathscr{F}_h$.

$$\boldsymbol{u}_h = (u_e)_{e \in \mathscr{F}}$$

.

$u_\text{e} = \frac{1}{|\text{e}|} \int_\text{e} \boldsymbol{u} \cdot \boldsymbol{n}_\text{e}$ represents the average flux $\boldsymbol{u}$ across the mesh edge e in the direction of the normal vector of u_e, denoted from this point as $\boldsymbol{n}_\text{e}$. We will also define $u_{\text{P,e}}$ as the average flux of $\boldsymbol{u}$ across edge e restricted to cell P in the direction $\boldsymbol{n}_{\text{P,e}}$, where $\boldsymbol{n}_{\text{P,e}}$ is the normal vector of the cell oriented outward of cell P. From this it follows that $u_{\text{P,e}} = \alpha_{\text{P,e}} u_\text{e}$, where $\alpha_{\text{P,e}} = \boldsymbol{n}_{\text{P,e}} \cdot \boldsymbol{n}_\text{e}$.

Now that we have our 2 spaces $\mathscr{P}_h$ and $\mathscr{F}_h$ we need to endow them with mimetic inner products. We start with $\mathscr{P}_h$, and define the inner product[3] as:

$$[p_h, v_h]_{\mathscr{P}_h} = \sum_{\text{P} \in \Omega_h} |\text{P}| p_\text{P} v_\text{P}, \tag{2.6}$$

with $|\text{P}|$ defined as the area of cell P, and the mimetic inner product on $\mathscr{F}_h$ as:

$$[\boldsymbol{u}_h, \boldsymbol{v}_h]_{\mathscr{F}_h} = \sum_{\text{P} \in \Omega_h} [\boldsymbol{u}_\text{P}, \boldsymbol{v}_\text{P}]_{\mathscr{F}_h, \text{P}}, \tag{2.7}$$

where $[\boldsymbol{u}_{\mathrm{P}}, \boldsymbol{v}_{\mathrm{P}}]_{\mathscr{F}_h}$ is assembled from the local inner products $[\boldsymbol{u}_{\mathrm{P}}, \boldsymbol{v}_{\mathrm{P}}]_{\mathscr{F}_h,\mathrm{P}}$. We first define the following terms with $\mathbf{e}_i$ denoting basis vector number i for $\mathbb{R}$. For $\mathbb{R}^2$ we have the following:

$$\mathbf{e}_1 = (1,0) \text{ and } \mathbf{e}_2 = (0,1)$$
$$t_i^1(\mathbf{x}) = \mathbf{e}_i \cdot (\mathbf{x} - \mathbf{x}_{\mathrm{P}})$$
$$\mathbf{e}_i = \nabla t_i^1$$

.

We then perform integration by parts to arrive at our final expression:

$$\begin{aligned}
[\varphi, \Pi_{\mathrm{P}}^{\mathscr{F}}(\mathbf{e}_i)]_{\mathscr{F}_h,\mathrm{P}} &= \int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{F}}(\varphi) \cdot \mathrm{e}_i dV \\
&= \int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{F}}(\varphi) \cdot \nabla t_i^1 dV \\
&= -\int_{\mathrm{P}} t_i^1 \mathrm{div} R_{\mathrm{P}}^{\mathscr{F}}(\varphi) dV + \sum_{\mathrm{f}\in\partial\mathrm{P}} \int_{\mathrm{f}} \mathbf{n}_{\mathrm{P,f}} \cdot R_{\mathrm{P}}^{\mathscr{F}}(\varphi) t_i^1 dS
\end{aligned}$$

The first integral on the right hand side is equal to zero, by first using the commuting property to rewrite $\mathrm{div} R_{\mathrm{P}}^{\mathscr{F}}(\varphi)$ as $R_{\mathrm{P}}^{\mathscr{P}}(\mathrm{div}_h \varphi)$. As this is constant on the cell P, the remaining integral $\int_{\mathrm{P}} t_i^1 dV = 0$, we then make use of the data locality property which tells us that $\mathbf{n}_{\mathrm{f}} \cdot R_{\mathrm{P}}^{\mathscr{F}}(\varphi) = R_{\mathrm{f}}^{\mathscr{F}}(\varphi_{|\mathrm{f}}) = \varphi_{\mathrm{f}}$ [2, R4 & R5 P. 72-73]. Since we are working in the 2D case, our edge and face spaces, $\mathscr{E}$ and $\mathscr{F}$, are equivalent. We make use of that equivalence to show that $\mathbf{n}_{\mathrm{f}} \cdot R_{\mathrm{P}}^{\mathscr{F}}(\varphi) = \mathbf{n}_{\mathrm{e}} \cdot R_{\mathrm{P}}^{\mathscr{E}}(\varphi) = \varphi_{\mathrm{e}}$.

$$\begin{aligned}
[\varphi, \Pi_{\mathrm{P}}^{\mathscr{F}}(\mathbf{e}_i)]_{\mathscr{F}_h,\mathrm{P}} &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \int_{\mathrm{e}} \mathbf{n}_{\mathrm{P,e}} \cdot R_{\mathrm{P}}^{\mathscr{E}}(\varphi) t_i^1 dS \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \alpha_{\mathrm{P,e}} \varphi_{\mathrm{e}} \int_{\mathrm{e}} t_i^1 dS \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \alpha_{\mathrm{P,e}} \varphi_{\mathrm{e}} \mathbf{e}_i \cdot (\mathbf{x}_{\mathrm{e}} - \mathbf{x}_{\mathrm{P}}) |\mathrm{e}| \\
[\varphi, \Pi_{\mathrm{P}}^{\mathscr{F}}(\mathbf{e}_i)]_{\mathscr{F}_h,\mathrm{P}} &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \varphi_{\mathrm{e}} R_{\mathrm{e}} \qquad (2.8)
\end{aligned}$$

Here $\alpha_{\mathrm{P,e}}$ denotes the dot product of the edge normal and outward cell normal, having a value of ± 1. We know that there exists a matrix, denoted by $M_{\mathscr{P}}$, that (2.6) can be written in the following form:

$$[p_h, v_h]_{\mathscr{P}_h} = p_h^T M_{\mathscr{P}} v_h = \sum_{\mathrm{P}\in\Omega_h} |\mathrm{P}| p_{\mathrm{P}} v_{\mathrm{P}}. \qquad (2.9)$$

From this we can deduce that $M_{\mathscr{P}}$ would be a diagonal matrix where the element $M_{\mathscr{P}}(i,i) = |\mathrm{P}_i|$, in other words the area of cell i. We will similarly construct $M_{\mathscr{F}}$, with the change that we firstly construct $M_{\mathscr{F},P}$ for every cell P, and then construct the global matrix $M_{\mathscr{F}}$ using these.

$$[\boldsymbol{u}_h, \boldsymbol{v}_h]_{\mathscr{F}_h,P} = \boldsymbol{u}_{h,P}^T M_{\mathscr{F},P} \boldsymbol{v}_{h,P},$$

where $M_{\mathscr{F},P}$ is restricted to cell P, and satisfies the equation $M_{\mathscr{F},P} N_{\mathscr{F},P} = R_{\mathscr{F},P}$, and $N_{\mathscr{F},P}$, $R_{\mathscr{F},P}$ are rectangular and of the form [2, Section: 3.4.3]:

$$N_{\mathscr{F},P} = \begin{pmatrix} \boldsymbol{n}_{\mathrm{e}_1}^T \\ \boldsymbol{n}_{\mathrm{e}_2}^T \\ \vdots \\ \boldsymbol{n}_{\mathrm{e}_{N_{\mathrm{P}}^E}}^T \end{pmatrix}, R_{\mathscr{F},P} = \begin{pmatrix} \alpha_{\mathrm{P,e}_1} |\mathrm{e}_1| (\boldsymbol{x}_{\mathrm{e}_1} - \boldsymbol{x}_{\mathrm{P}})^T \\ \alpha_{\mathrm{P,e}_2} |\mathrm{e}_2| (\boldsymbol{x}_{\mathrm{e}_2} - \boldsymbol{x}_{\mathrm{P}})^T \\ \vdots \\ \alpha_{\mathrm{P,e}_{N_{\mathrm{P}}^F}} |\mathrm{e}_{N_{\mathrm{P}}^F}| (\boldsymbol{x}_{\mathrm{e}_{N_{\mathrm{P}}^F}} - \boldsymbol{x}_{\mathrm{P}})^T \end{pmatrix} \qquad (2.10)$$

,

where x_{P} is the barycenter or centroid[6] of cell P, and x_{e} is the center of the edge in question. Matrix $\mathrm{R}_{\mathscr{F},P}$ is assembled from (2.8), and $\mathrm{N}_{\mathscr{F},P}$ is the face projection operator applied to the basis vectors in $\mathbb{R}^2, \Pi_{\mathrm{P}}^{\mathscr{F}}(\mathbf{e}_i) = \left(\frac{1}{|\mathrm{e}|}\int_{\mathrm{e}} \boldsymbol{e}_i \cdot \boldsymbol{n}_{\mathrm{e}}\right)_{\mathrm{e}\in\mathscr{F}} = \left(\frac{1}{|\mathrm{e}|}\int_{\mathrm{e}} \boldsymbol{e}_i \cdot \boldsymbol{n}_{\mathrm{e}}\right)_{\mathrm{e}\in\mathscr{E}}$. As we are able to calculate these matrices, we are now able to form $M_{\mathscr{F},P}$ from the equation $M_{\mathscr{F},P}N_{\mathscr{F},P} = R_{\mathscr{F},P}$. The global mass matrix $M_{\mathscr{F}}$ is assembled in the same way as in the finite element method[1].

$$M_{\mathscr{F}} = \sum_{\mathrm{P}\in\Omega_h} A_{\mathrm{P}}^T M_{\mathscr{F},P} A_{\mathrm{P}} \tag{2.11}$$

Where element $A_{\mathrm{P}}(i,j) = 1$ if the local edge $\mathrm{e}_{i,P}$ is the global edge e_j. In other words A_{P} is the assembly matrix with number of rows equal to the number of edges the cell P has, and number of columns the same as the number of edges in the mesh.

Now that we have defined the inner products, we have to choose a discrete approximation for the divergence operator div_h. One such discrete approximation can be found in [2, Equation 2.23]:

$$(\mathrm{div}_h \boldsymbol{u}_h)_{\mathrm{P}} = \frac{1}{|\mathrm{P}|}\sum_{\mathrm{e}\in\partial\mathrm{P}} \alpha_{\mathrm{P},\mathrm{e}}|\mathrm{e}|u_{\mathrm{e}} \tag{2.12}$$

Here, div_h is a matrix with # of rows equal to the number of cells in the mesh and # of columns equal to the number of edges on the mesh. We arrive at this equation by starting with the Stokes theorem on the cell P:

$$\int_{\mathrm{P}} \mathrm{div}\mathbf{u}dV = \int_{\partial\mathrm{P}} \mathbf{u}\cdot\mathbf{n}_{\mathrm{P},\mathrm{e}}dS$$

And then using the face and cell projection operators defined in [2, Eq. (2.16)&(2.17)] to arrive at our definition of div_h applied to a face-based discrete field $\mathbf{u}_h$ (2.12).

$$\begin{aligned}|\mathrm{P}|(\mathrm{div}_h \boldsymbol{u}_h)_{\mathrm{P}} &= \int_{\partial\mathrm{P}} \mathbf{u}\cdot\mathbf{n}_{\mathrm{P},\mathrm{e}}dS \\ &= \sum_{\mathrm{e}\in\partial\mathrm{P}}\int_{\mathrm{e}} \mathbf{u}\cdot\alpha_{\mathrm{P},\mathrm{e}}\mathbf{n}_{\mathrm{e}}dS\end{aligned}$$

Rearranging the definition of the face projection operator $u_{\mathrm{e}}|\mathrm{e}| = \int_{\mathrm{e}} \mathbf{u}\cdot\mathbf{n}_{\mathrm{e}}dS$:

$$\begin{aligned}|\mathrm{P}|(\mathrm{div}_h \boldsymbol{u}_h)_{\mathrm{P}} &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \alpha_{\mathrm{P},\mathrm{e}}u_{\mathrm{e}}|\mathrm{e}| \\ (\mathrm{div}_h \boldsymbol{u}_h)_{\mathrm{P}} &= \frac{1}{|\mathrm{P}|}\sum_{\mathrm{e}\in\partial\mathrm{P}} \alpha_{\mathrm{P},\mathrm{e}}u_{\mathrm{e}}|\mathrm{e}|\end{aligned}$$

$$\mathrm{div}_h(i,j) = \begin{cases} \frac{|\mathrm{e}_j|}{|\mathrm{P}_i|}\alpha_{\mathrm{P}_i,\mathrm{e}_j}, \text{if } \mathrm{e}_j \text{ is and edge of } \mathrm{P}_i \\ 0, \text{otherwise}\end{cases} \tag{2.13}$$

Similarly we have to find a discrete approximation of the gradient operator $\widetilde{\nabla}_h$. But we cannot just choose any discrete approximation, we need to ensure that div_h and $\widetilde{\nabla}_h$ satisfy the Green theorem. We start by writing (1.2) in its discrete form:

$$\begin{aligned}\left[\boldsymbol{u}_h, \widetilde{\nabla}_h p_h\right]_{\mathscr{F}_h} &= -\left[\mathrm{div}_h\boldsymbol{u}_h, p_h\right]_{\mathscr{P}_h} \\ \boldsymbol{u}_h^T M_{\mathscr{F}}\widetilde{\nabla}_h p_h &= -\boldsymbol{u}_h^T \mathrm{div}_h^T M_{\mathscr{P}} p_h \\ M_{\mathscr{F}}\widetilde{\nabla}_h &= -\mathrm{div}_h^T M_{\mathscr{P}} \\ \widetilde{\nabla}_h &= -M_{\mathscr{F}}^{-1}\mathrm{div}_h^T M_{\mathscr{P}}\end{aligned} \tag{2.14}$$

All that is left to do, is to construct the RHS of the system, and in order to do this it is easier to discretize the weak form of our equation. Doing this we get the following system[2, (1.18)]:

$$\begin{aligned}\int_\Omega \boldsymbol{u}\cdot\boldsymbol{v}dV - \int_\Omega p\mathrm{div}\boldsymbol{v}dV &= -\int_{\Gamma_\mathrm{D}} g^D\boldsymbol{v}\cdot\boldsymbol{n}\ \forall\boldsymbol{v}\in X_0 \\ \int_\Omega q\mathrm{div}\boldsymbol{u}dV &= \int_\Omega bqdV\ \forall q\in L^2(\Omega)\end{aligned} \tag{2.15}$$

$$\begin{aligned}[\boldsymbol{u}_h,\boldsymbol{v}_h]_{\mathscr{F}_h} - [p_h,\mathrm{div}_h\boldsymbol{v}_h]_{\mathscr{P}_h} &= -\boldsymbol{v}_h^T\Gamma \\ [\mathrm{div}_h\boldsymbol{u}_h, q_h]_{\mathscr{P}_h} &= [\Pi(b), q_h]_{\mathscr{P}_h}\end{aligned}$$

$$\begin{aligned}\boldsymbol{v}_h^T M_{\mathscr{F}}\boldsymbol{u}_h - \boldsymbol{v}_h^T\mathrm{div}_h^T M_{\mathscr{P}} p_h &= -\boldsymbol{v}_h^T\Gamma \\ q_h^T M_{\mathscr{P}}\mathrm{div}_h\boldsymbol{u}_h &= q_h^T\tilde{b}\end{aligned}$$

$$\begin{aligned}M_{\mathscr{F}}\boldsymbol{u}_h - \mathrm{div}_h^T M_{\mathscr{P}} p_h &= -\Gamma \\ M_{\mathscr{P}}\mathrm{div}_h\boldsymbol{u}_h &= \tilde{b}\end{aligned}$$

$$\begin{pmatrix} M_{\mathscr{F}} & -\mathrm{div}_h^T M_{\mathscr{P}} \\ -M_{\mathscr{P}}\mathrm{div}_h & 0\end{pmatrix}\begin{pmatrix}\boldsymbol{u}_h \\ p_h\end{pmatrix} = \begin{pmatrix}-\Gamma \\ -\tilde{b}\end{pmatrix}. \tag{2.16}$$

Where $\int_{\Gamma_\mathrm{D}} g^D\boldsymbol{v}\cdot\boldsymbol{n}$ comes from the boundary conditions at the external edges and $\int_\Omega bqdV$ as a result of integration by parts. If the boundary conditions are given by the function g^D, then the vector of boundary conditions Γ is given by:

$$\Gamma(j) = \Pi^{\mathscr{F}_h}(g^D) = \begin{cases} |\mathrm{e}_j| * g^D(\boldsymbol{x}_{e_j}), & \text{if edge } \mathrm{e}_j \text{is an external edge} \\ 0, & \text{otherwise.}\end{cases}$$

Likewise, if the RHS of Laplace is given by the function $b(x,y)$ the RHS of the final system $\tilde{b}$ is given as:

$$\tilde{b}(i) = |\mathrm{P}_i| * \Pi^{\mathscr{P}_h}(b) = |\mathrm{P}_i| * b(\boldsymbol{x}_{P_i}), \forall\ \mathrm{P}_i\in\Omega_h. \tag{2.17}$$

2.3 The MFDM for the Primal Form of the Laplace Equation

Using [2], it is relatively straightforward to derive a 2D discretization scheme for the mixed formulation approach from the 3D formulas given, as all the formulas are the same. When we are working with the Primal Form approach, the formulas for the mass matrices differ, meaning we have to derive a new discretization scheme.

As a reminder we are still working with the Laplace equation (1.1), but we will be attaching different degrees of freedom. The D.O.F.'s we will be using are as follows[2, Chapter 6.1.1]:

- The node-based discrete field $p_h\in\mathscr{V}_h$, the components of which approximate the nodal values of p. In other words $p_h = (p_\mathrm{v})_{\mathrm{v}\in\mathscr{V}}$
- The edge-based discrete field $\mathrm{F}_h\in\mathscr{E}_h$ where the components F_e approximate the tangential components of ∇p on mesh edges e. $\mathrm{F}_h = (F_\mathrm{e})_{\mathrm{e}\in\mathscr{E}}$

From our choice of D.O.F.'s we now need to define our primary mimetic operator. From [2, Lemma 2.2] it becomes clear that as we have the spaces $\mathscr{V}_h$ and $\mathscr{E}_h$, our primary operator should be the mimetic gradient operator $\nabla_h : \mathscr{V}_h \to \mathscr{E}_h$. And as we are working with the Laplacian ($div(\nabla)$), our derived mimetic operator is the divergence operator $\widetilde{div}_h$.

We now need to define a discrete approximation of the gradient operator. We will be using the following [2, Equation (2.19)]:

$$(\nabla_h p_h)_\mathrm{e} = \frac{p_{\mathrm{v}_2} - p_{\mathrm{v}_1}}{|\mathrm{e}|} \tag{2.18}$$

Where e is the edge orientated from v_1 to v_2. Given the principles explored in [2, Chapter 2], we know that the our derived divergence operator is dual to the primary gradient operator with respect to the inner products in $\mathscr{V}_h$ and $\mathscr{E}_h$. Thus giving us the equation [2, Equation (2.32)]:

$$\widetilde{div}_h = -M_{\mathscr{V}}^{-1} \nabla_h^T M_{\mathscr{E}} \tag{2.19}$$

The process we employ to arrive at this definition is the same we used in the Mixed Formulation method, which can be seen in equation (2.14). As we are working in slightly different spaces than in the Mixed Formulation we have to derive some new expressions for our mass matrices. These derivation processes are detailed in the following two sections.

2.3.1 Derivation Process of $M_{\mathscr{V}}$

$$[\varphi, \Pi_\mathrm{P}^{\mathscr{V}}(\mathrm{c})]_{\mathscr{V}_h,\mathrm{P}} = \int_\mathrm{P} R_\mathrm{P}^{\mathscr{V}}(\varphi) c dV,$$

where φ is a scalar field and c is a constant function, which we will set to $c = 1$. We have the identity that in 2D $2 = \mathrm{div}(\mathrm{x} - \mathrm{x}_\mathrm{P})$, where x_P is the barycenter of the cell, and by then performing integration by parts, we end up with the following derivation:

$$\begin{aligned} 2\int_\mathrm{P} R_\mathrm{P}^{\mathscr{V}}(\varphi) dV &= \int_\mathrm{P} R_\mathrm{P}^{\mathscr{V}}(\varphi)\mathrm{div}(\mathrm{x} - \mathrm{x}_\mathrm{P}) dV \\ &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \int_\mathrm{e} R_\mathrm{e}^{\mathscr{V}}(\varphi_{|\mathrm{e}})(\mathrm{x} - \mathrm{x}_\mathrm{P}) \cdot \mathrm{n}_{\mathrm{P,e}} dL - \int_\mathrm{P} \nabla R_\mathrm{P}^{\mathscr{V}}(\varphi) \cdot (\mathrm{x} - \mathrm{x}_\mathrm{P}) dV \end{aligned}$$

We make use of the orthogonality and commutative properties, R4 and R3 respectively[2, Chapter 3.3.2.1], from this we see that the second integral on the right hand side must be equal to 0. We first use the commuting property to make the following change: $\nabla R_\mathrm{P}^{\mathscr{V}} = R_\mathrm{P}^{\mathscr{E}} \nabla$, and then make use of the orthogonal property to show that $R_\mathrm{P}^{\mathscr{E}}(\nabla_h \varphi) \cdot \int_\mathrm{P} (\mathrm{x} - \mathrm{x}_\mathrm{P}) dV = R_\mathrm{P}^{\mathscr{E}}(\nabla_h \varphi) \cdot 0 = 0$ [2, Page 72]. Using the midpoint quadrature rule, where $\boldsymbol{x}_\mathrm{e}$ is the midpoint of the edge e, we can also show that:

$$\int_\mathrm{e} R_\mathrm{e}^{\mathscr{V}}(\varphi_{|\mathrm{e}}) dL = R_\mathrm{e}^{\mathscr{V}}(\varphi)(\boldsymbol{x}_\mathrm{e})|\mathrm{e}| = \frac{\varphi_{\mathrm{v}_1} + \varphi_{\mathrm{v}_2}}{2}|\mathrm{e}|.$$

$$\begin{aligned} \int_\mathrm{P} R_\mathrm{P}^{\mathscr{V}}(\varphi) dV &= \frac{1}{2} \sum_{\mathrm{e}\in\delta\mathrm{P}} \int_\mathrm{e} R_\mathrm{e}^{\mathscr{V}}(\varphi_{|\mathrm{e}})(\mathrm{x} - \mathrm{x}_\mathrm{P}) \cdot \mathrm{n}_{\mathrm{P,e}} dL \\ &= \frac{1}{2} \sum_{\mathrm{e}\in\delta\mathrm{P}} (\mathrm{x} - \mathrm{x}_\mathrm{P}) \cdot \mathrm{n}_{\mathrm{P,e}} \int_\mathrm{e} R_\mathrm{e}^{\mathscr{V}}(\varphi_{|\mathrm{e}}) dL \\ &= \frac{1}{2} \sum_{\mathrm{e}\in\partial\mathrm{P}} (\mathrm{x} - \mathrm{x}_\mathrm{P}) \cdot \mathrm{n}_{\mathrm{P,e}} \frac{\varphi_{\mathrm{v}_1} + \varphi_{\mathrm{v}_2}}{2}|\mathrm{e}| \\ &= \frac{1}{4} \sum_{\mathrm{e}\in\partial\mathrm{P}} (\mathrm{x} - \mathrm{x}_\mathrm{P}) \cdot \mathrm{n}_{\mathrm{P,e}}|\mathrm{e}|(\varphi_{\mathrm{v}_1} + \varphi_{\mathrm{v}_2}) \\ [\varphi, \Pi_\mathrm{P}^{\mathscr{V}}(\mathrm{c})]_{\mathscr{V}_h,\mathrm{P}} &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \mathrm{R}_\mathrm{v}\varphi_\mathrm{v}, \end{aligned} \tag{2.20}$$

where e is the edge between the vertices v_1 and v_2. We collect all of these $\mathrm{R_v}$ in the vector $\mathrm{R}= (\mathrm{R_v})_{\mathrm{v}\in\delta\mathrm{P}}$, and let $\mathrm{N} = \prod_{\mathrm{P}}^{\mathscr{V}}(c) = (c(x_v))_{v\in\mathscr{V}} = (1)_{v\in\mathscr{V}}$. This implies that all elements of N are equal to 1. We use [2, Equation (3.52)] to rewrite our (2.20) in it's matrix representation:

$$\begin{aligned}\varphi^T MN &= \varphi^T R\\ MN &= R\end{aligned} \tag{2.21}$$

Where M denotes the local mass matrix for the cell P. All of these local matrices will then be added to the global mass matrix $M_{\mathscr{V}}$. For methods of solving system (2.21), once again see [2, Chapter 3.4.5]. As with the Mixed Formulation method, the global matrix is assembled in the same way it is in the finite element method[1].

2.3.2 Derivation Process of $M_{\mathscr{E}}$

$$[\varphi, \Pi_{\mathrm{P}}^{\mathscr{E}}(\mathrm{e}_i)]_{\mathscr{E}_h,\mathrm{P}} = \int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{E}}(\varphi)\cdot \mathrm{e}_i dS \tag{2.22}$$

Before we can start our derivation process, we need to define the cross product and curl in two dimensions, as they are generally only definable in three dimensions, and our variable p_i^1 (Once again, e_i denotes the basis vectors in $\mathbb{R}^2$):

Cross Product: Given u= (u_x, u_y) and v= (v_x, v_y), then the cross product is given by[8]:

$$\mathrm{u}\times\mathrm{v} = \det(\mathrm{u}\ \mathrm{v}) = u_x v_y - u_y v_x$$

.

Curl: The curl in 2D is calculated in 2 different ways, depending on whether we take the curl of a scalar or a vector in 2D, we define it as follows:

$$curl(c) = \left[\frac{\partial c}{\partial y}, -\frac{\partial c}{\partial x}\right] \text{ ,where c is a scalar}$$

$$curl\left(\begin{bmatrix} a\\ b\end{bmatrix}\right) = \frac{\partial b}{\partial x} - \frac{\partial a}{\partial y}$$

$$\begin{aligned}\mathrm{p}_i^1 &= \mathrm{e}_i \times (\mathrm{x} - \mathrm{x_P})\\ \mathrm{e}_i &= curl\mathrm{p}_i^1.\end{aligned}$$

Where we once again have that $\mathrm{x_P}$ is the barycenter of the cell. We will also define $\boldsymbol{\tau}_{\mathrm{e}}$ to be the tangential unit vector of edge e, and similarly to the normal vector, when it is restricted to cell P, it is noted as $\boldsymbol{\tau}_{\mathrm{e,P}}$, the tangential unit vector oriented counter clockwise with respect to the cell P. The product of these two is denoted as $\boldsymbol{\tau}_{\mathrm{e}}\cdot\boldsymbol{\tau}_{\mathrm{e,P}} = \beta_{\mathrm{P,e}}$, and is either positive or negative 1 depending on the orientation of the tangential unit vector when restricted to cell P.

Using [2, (R3)&(R4) P.72] and integration by parts, we can simplify the above equation:

$$\begin{aligned}\int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{E}}(\varphi)\cdot\mathrm{e}_i dS &= \int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{E}}(\varphi)\cdot curl\mathrm{p}_i^1 dS\\ &= -\int_{\mathrm{P}} \mathrm{p}_i^1 curl R_{\mathrm{P}}^{\mathscr{E}}(\varphi) dS + \sum_{\mathrm{e}\in\partial\mathrm{P}}\int_{\mathrm{e}} \boldsymbol{\tau}_{\mathrm{e}}\cdot R_{\mathrm{P}}^{\mathscr{E}}(\varphi)\mathrm{p}_i^1 dL\end{aligned}$$

The first integral on the RHS is equal to 0 [2, Orthogonality Property R4, P. 72]:

$$-\int_{\mathrm{P}} \mathrm{p}_i^1 curl R_{\mathrm{P}}^{\mathscr{E}}(\varphi)dS = -\int_{\mathrm{P}} R_{\mathrm{P}}^{\mathscr{F}}(curl_h\varphi)\mathrm{p}_i^1 dS = -R_{\mathrm{P}}^{\mathscr{F}}(curl_h\varphi)\int_{\mathrm{P}} \mathrm{e}_i\times(\mathrm{x}-\mathrm{x_P})dS = -R_{\mathrm{P}}^{\mathscr{F}}(curl_h\varphi)\cdot 0 = 0$$

Again, we make use of data locality property, and the fact that in this method our cell space is equal to our face space, $\mathscr{P} = \mathscr{F}$ to make the following equivalence [2, R5, P.73]:

$$\boldsymbol{\tau}_{\mathrm{e}} \cdot R^{\mathscr{E}}_{\mathrm{P}}(\varphi) = \boldsymbol{\tau}_{\mathrm{e}} \cdot R^{\mathscr{E}}_{\mathrm{f}}(\varphi) = R^{\mathscr{E}}_{\mathrm{e}}(\varphi_{|\mathrm{e}}) = \varphi_{\mathrm{e}}$$

$$\begin{aligned}
\int_{\mathrm{P}} R^{\mathscr{E}}_{\mathrm{P}}(\varphi) \cdot \mathrm{e}_i dS &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \int_{\mathrm{e}} \boldsymbol{\tau}_{\mathrm{e}} \cdot R^{\mathscr{E}}_{\mathrm{P}}(\varphi)\mathrm{p}^1_i dL \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \int_{\mathrm{e}} \beta_{\mathrm{P,e}} R^{\mathscr{E}}_{\mathrm{e}}(\varphi_{|\mathrm{e}})\mathrm{p}^1_i dL \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \beta_{\mathrm{P,e}}\varphi_{\mathrm{e}} \int_{\mathrm{e}} \mathrm{p}^1_i dL \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \beta_{\mathrm{P,e}}\varphi_{\mathrm{e}} \int_{\mathrm{e}} \mathbf{p}^1_i dL \\
&= \sum_{\mathrm{e}\in\partial\mathrm{P}} \beta_{\mathrm{P,e}}\varphi_{\mathrm{e}}|\mathrm{e}|\mathbf{p}^1_i \\
[\varphi, \Pi^{\mathscr{E}}_{\mathrm{P}}(\mathrm{e}_i)]_{\mathscr{E}_h,\mathrm{P}} &= \sum_{\mathrm{e}\in\partial\mathrm{P}} \mathrm{R}_{\mathrm{e}}\varphi_{\mathrm{e}}
\end{aligned}$$

When applying [2, Lemma 3.2] we have to make a slight adjustment because we can no longer just simply assume that the orientation of the local edge e of cell P is the same as the orientation of cell e on the global mesh. Therefore we multiply by the orientation constant $\beta_{\mathrm{P,e}} = \pm 1$, depending on whether the edge locally has the same orientation as in the global mesh.

We now define our $\mathrm{N} = \prod^{\mathscr{E}}_{\mathrm{P}}(\mathrm{e}_i) = \left(\frac{1}{|\mathrm{e}|}\int_{\mathrm{e}} e_i \cdot \boldsymbol{\tau}_{\mathrm{e}}\right)_{e\in\mathscr{E}}$ (which is the equivalent of an array containing all of the normalized tangential vectors of the local edges), and collect all of our R_{e} in the local vector $\mathrm{R} = (\mathrm{R}_{\mathrm{e}})_{\mathrm{e}\in\delta\mathrm{P}}$. Once again we make use of [2, Equation (3.52)] to rewrite (2.22) into its matrix representation:

$$\begin{aligned}
\varphi^T MN &= \varphi^T R \\
MN &= R
\end{aligned}$$

Where M denotes the local mass matrix for cell P, and as each of these are calculated for the individual cells they are added to the global mass matrix $M_{\mathscr{E}}$. This is achieved in the same manner as for $M_{\mathscr{V}}$, by making use of the assembly method of the finite element method and using [2, Chapter 3.4.5] to solve the local systems for M.

2.3.3 Assembling the Final System

Now that we have assembled our two mass matrices, we can set up our final system, much like we did with the mixed form method. Making use of the mimetic operators, we first re-write (2.2) and (2.3) into the following form:

$$\begin{aligned}
\boldsymbol{u}_h - \nabla_h p_h &= 0 \\
-\widetilde{\mathrm{div}}_h \boldsymbol{u}_h &= \Pi^{\mathscr{V}}(b)
\end{aligned} \tag{2.23}$$

.

We also know due to the Green Formula that:

$$\widetilde{\mathrm{div}}_h = -M_{\mathscr{V}}^{-1}\nabla_h^T M_{\mathscr{E}}$$

.

We can now re-arrange (2.23) to get the following final linear system which we can solve using the calculated mass matrices:

$$\nabla_h^T M_{\mathscr{E}} \nabla_h p_h = M_{\mathscr{V}} \Pi^{\mathscr{V}}(b) \tag{2.24}$$

Chapter 3

Matlab Code

While the entirety of the code will be included as an appendix, this section will be dedicated to explaining some of the less self-explanatory code sections.

The final discretizations used in the methods are not necessarily complex in and of themselves, however they present a small challenge when it comes to programming them, specifically in terms of indexing and shape of the mesh. The Polymesher function outputs the location of the nodes and which nodes belong to which cell in a counter clockwise rotation, but this tells us nothing about the edges and the normal vectors, etc.. As such it was necessary to create several arrays in order to determine the structure of the mesh, most importantly the unique edges of the mesh and cell-edge adjacency. As that code can be somewhat confusing, what follows is a very short explanation of the code along with the code itself.

Matlab Code 3.1: Determine Unique Edges and Which Edges Create Which Cells

```matlab
function [Edges] = UniqueEdges(NumEle,CelltoVertix)
Edges = [0,0]; %Initialize
for i = 1:NumEle %For all cells
    temp = vertcat(CelltoVertix{i},CelltoVertix{i}(1));
    for j = 1:length(temp)-1 %For all edges of cell
        if ~ismember([temp(j),temp(j+1)],Edges,'rows') &&...
        ~ismember([temp(j+1),temp(j)],Edges,'rows')
            Edges(end+1,:) = [temp(j),temp(j+1)];
        end
    end
    clear temp %Not clearing causes issues for loop
end
Edges = Edges(2:end,:); %Remove the initialization
end
```

As stated, the function of this code snippet is to identify the unique edges of the mesh, ensuring that each edge appears exactly once in the list. This is achieved in the following fashion: for every cell we create the temporary matrix temp, where column 1 contains the start vertices of the edges of P, and column 2 contains the end vertices of the edges of P. Then we compare each vertex pair in temp to the vertex pairs (i.e. edges) in the array Edges. If the edge has already been added to the array, we ignore it, else the new edge is added to the edges array. This is all achieved by the inbuilt Matlab function ismember. The final step involves removing the initial values we populated the edges array with, as it would otherwise be empty causing an error as ismember would have nothing to compare.

Matlab Code 3.2: Code for Global $M_{\mathscr{F}}$ Matrix

```matlab
function [Mf] = GMM(NumEle, Vertices, Cell2Edge, EdgeNormals, Alphas, EdgeCenters,
barycenters, EdgeLengths, Edges)

Mf = sparse(zeros(length(Edges)));

for i = 1:NumEle
```

```
7           R = zeros(length(Vertices{i}),2);
8           N = zeros(length(Vertices{i}),2);
9           temp = find(Cell2Edge(i,:));
10          for j = 1: length(temp)
11              R(j,:) = Alphas{i}(j)*(EdgeCenters(temp(j),:)
12              - barycenters(i,:))*EdgeLengths(temp(j));
13              N(j,:) = EdgeNormals(temp(j),:);
14          end
15          Size = 1/(length(temp)^2);
16          M0 = R*((R'*N)\R');
17          M = M0 + Size*trace(M0)*(eye(length(temp)) - N*((N'*N)\N'));
18          M = sparse(M);
19
20          Assembly = zeros(length(temp),length(Edges));
21
22          for k =         1:length(temp)
23              Assembly(k,temp(k)) = 1;
24          end
25
26          Assembly = sparse(Assembly);
27
28          Mf = Mf + Assembly' * M * Assembly;
29
30          clear temp
31      end
32
33      end
```

GMM constructs the global mass matrix Mf from the local mass matrices M at each cell P. M at each cell P is defined as described in the section following (2.10), R & N for each P being calculated in line 11 and 13 respectively. The local matrix M is then calculated in line 15-18. The Assembly matrix is constructed in line 22-24, in the form of a matrix indicating what the local edge's position is in the global system.

Matlab Code 3.3: Unique Edges V2

```
function [EdgesofP,Edges] = UniqueEdges(CelltoVertix)
alledges = [];
EdgesofP = cell(length(CelltoVertix),1);
for i = 1:length(CelltoVertix)
        temp = sort(horzcat(CelltoVertix{i},[CelltoVertix{i}(2:end);CelltoVertix{i}(1)]),2);
        EdgesofP{i} = temp;
        alledges = vertcat(alledges,temp);
end
Edges = unique(sort(alledges,2),'rows','stable');
end
```

The algorithm above performs the same function as Matlab Code 1, however Matlab Code 1 makes use of the inbuilt Matlab function ismember, which is far from efficient. Due to this we had to find a more efficient method which is shown in V2. As opposed to checking if a certain vertex pair has already been added to the unique vertex pair matrix as we do in Matlab Code 1, in V2 we instead construct the alledges matrix which consists of all vertex pairs for all cells P and then use the unique function in Matlab to determine the unique vertex pairs which is a lot quicker than the ismember command.

For reasons I have thus far been unable to determine however, this improved edge finding algorithm is not easily adapted to the Mixed Form method, causing the error to increase drastically independent of the approach attempted. This will later on in the experimental results section lead to some discrepancies in the assemble time as the Primal Form implementation has the more efficient code, and outperforms the Mixed Form method for coarse meshes. But for fine meshes, the Mixed Form implementation has a final system that is simpler to solve, causing it to outperform the Primal Form implementation for fine meshes. One explanation for this discrepancy is the fact that the improved edge finding algorithm can give the Primal Form Method an edge for coarse meshes, but can not make up for the difficult and slow solution of the final system in fine meshes.

Chapter 4

Experimental Results

The Matlab code for both implementations was then applied to a variety of domains and test problems. Both methods were applied to three domains, Unit Square/Circle and Wrench domain, and three test problems, as detailed in (4.1)-(4.3). The graphs and results detailed in this section focus on (4.1), and the Unit Circle/Square domain. The two methods employ different unknowns as detailed in the theory section and as such the size of the system that is being solved by the two methods varies for the same problem. This is something to keep in mind when analysing the results, as a larger number of unknowns might lead to a more accurate result, but will most likely also take longer to solve. The number of unknowns for the Mixed Form method is equal to the number of cells plus the number of edges, and for the Primal method, is equal to the number of internal nodes.

$$\begin{aligned} u_{exact}(x,y) &= e^x sin(y) \\ \Delta u_{exact}(x,y) &= 0 \end{aligned} \tag{4.1}$$

$$\begin{aligned} u_{exact}(x,y) &= \cos(2\pi x)\cos(\pi y) \\ \Delta u_{exact}(x,y) &= -5\pi^2 u_{exact}(x,y) \end{aligned} \tag{4.2}$$

$$\begin{aligned} u_{exact}(x,y) &= \sin(2\pi x)\sin(\pi y) \\ \Delta u_{exact}(x,y) &= -5\pi^2 u_{exact}(x,y) \end{aligned} \tag{4.3}$$

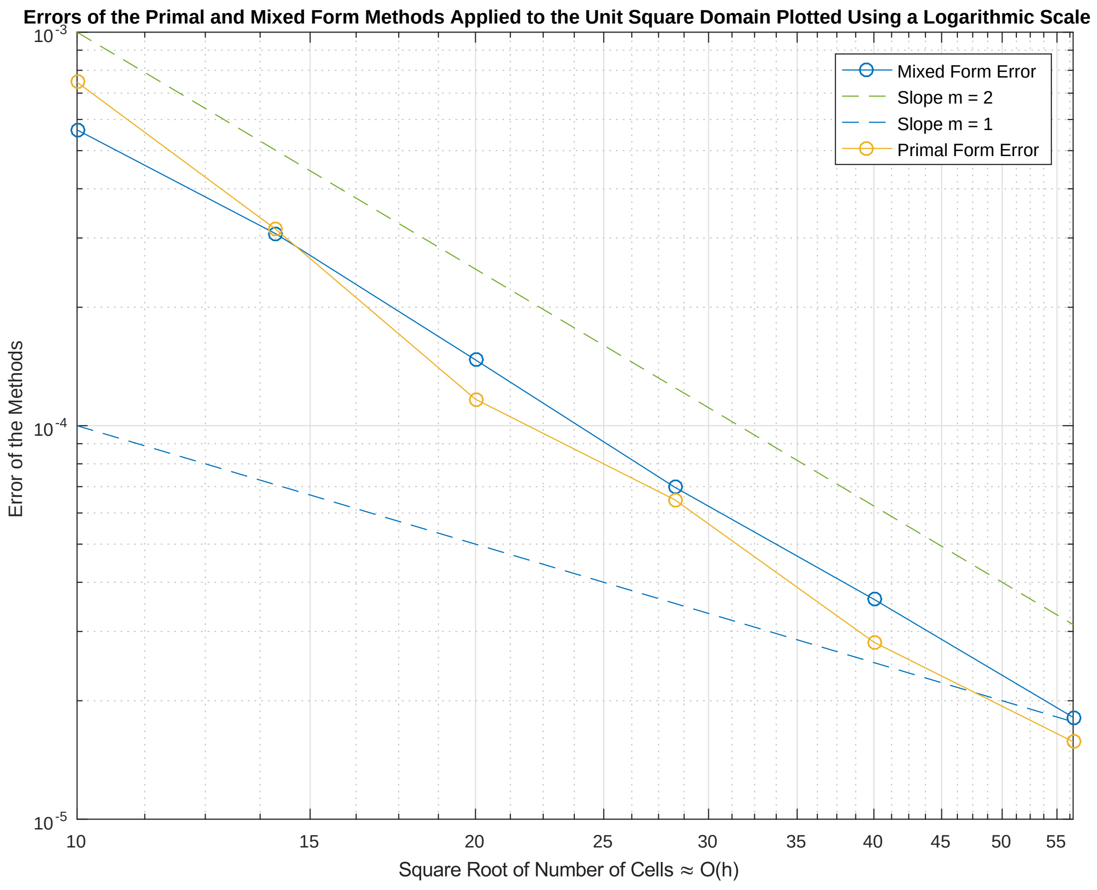

Figure 4.1: Unit Square with exact solution: $u(x, y) = e^x \sin y$

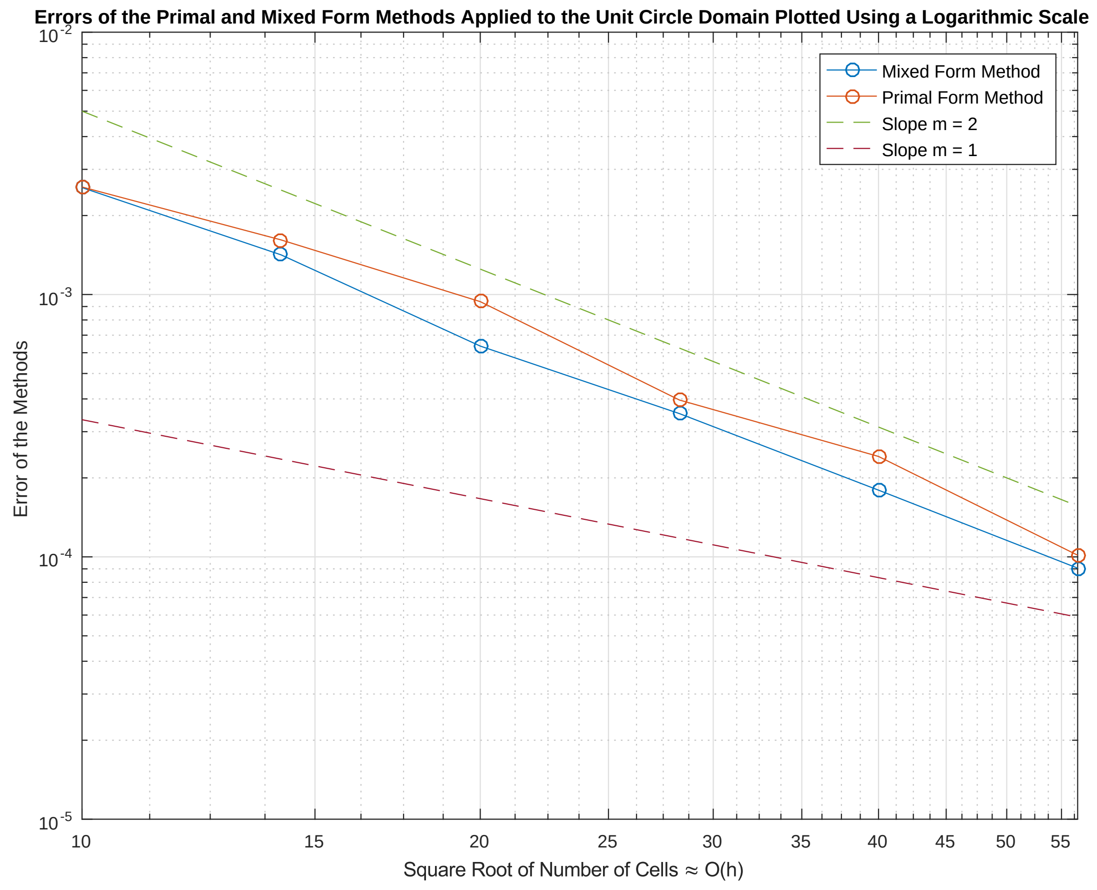

Figure 4.2: Unit Circle with exact solution: $u(x, y) = e^x \sin y$

Figures 4.1 & 4.2 demonstrate the errors generated by the two methods when applied to the unit square and circle domains. From these 2 figures we can see that the error seems to behave very similarly for both methods in both domains, not showing that either has a clear advantage over the other. To give the reader an impression of which meshes were tested, see Figures 2.1, 4.5 & 4.6 which demonstrate the meshes tested.

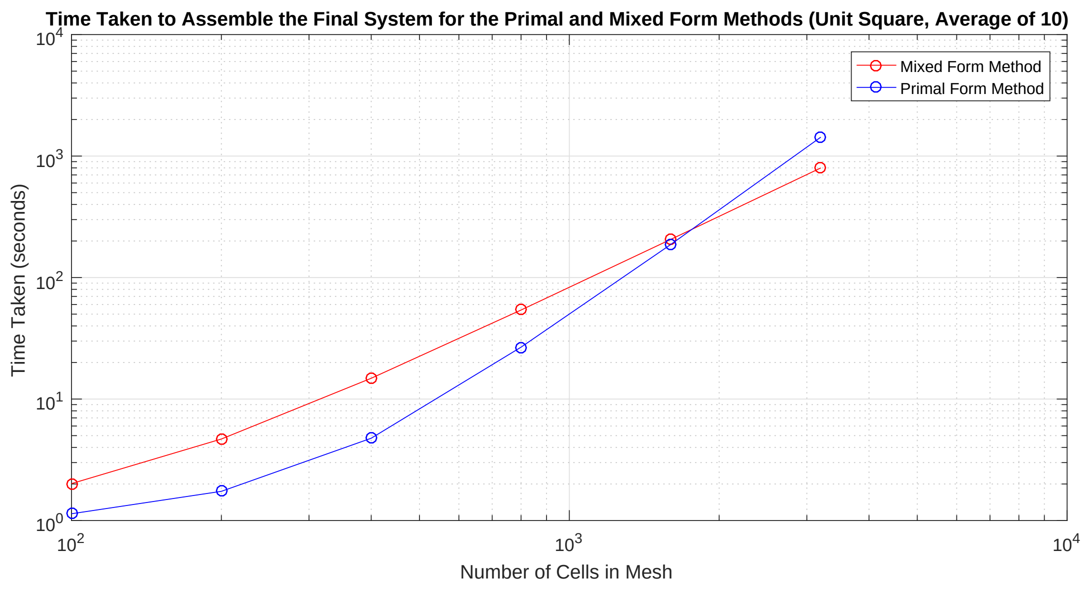

Figure 4.3: Exact solution: $u(x, y) = e^x \sin y$, Unit Square

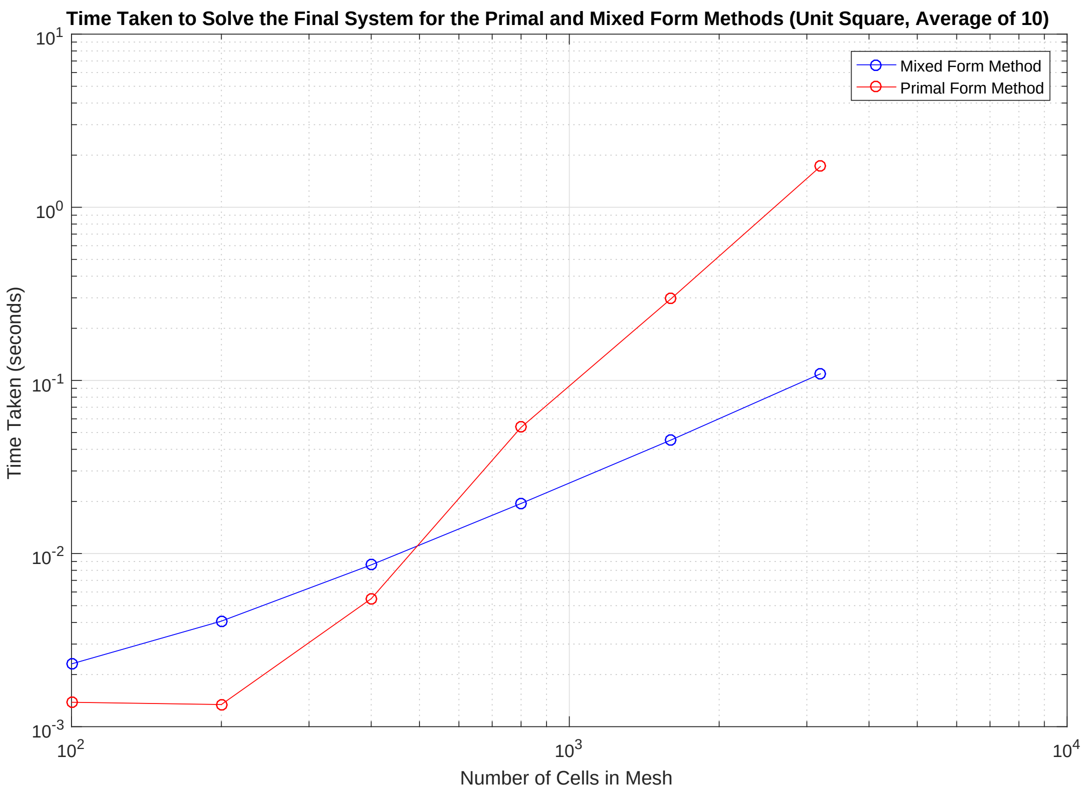

Figure 4.4: Exact solution: $u(x, y) = e^x \sin y$, Unit Square

The other factor to consider when choosing a discretization would be the time it takes for the methods to assemble and then solve the system. As mentioned in the Matlab Code section, the Primal Form code as a slight edge over the mixed form method, as I was able to circumvent the ismember function of Matlab, and instead employ the unique function. From Figures 4.3 and 4.4 we can see that, even with the Primal Form method having this edge, as the number of cells in the mesh increases, the Mixed Form method turns out to be the faster of the two methods by a slight margin when it comes to assembly time and a significant margin with respect to solve time. This goes for both the time taken to assemble the final system and to solve the system. From both of these figures it also appears that the difference in the solve and assembly time would increase the finer the mesh becomes, due to the fact that time for the Mixed Form method appears to increase linearly, while the Primal Form method increases exponentially.

From the error graphs 4.2 & 4.1 we can also tell that both methods appear to be second order accurate, which is exactly what we were expecting and attempting to create.

Combining the results for time taken and accuracy, it would appear that the Mixed Form method is the more suitable of the two when working with fine meshes, providing a quicker result than the Primal Form method, with comparable accuracy.

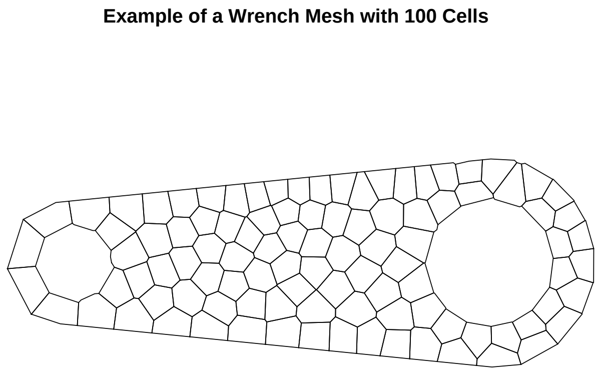

Figure 4.5: Wrench Polygonal Mesh

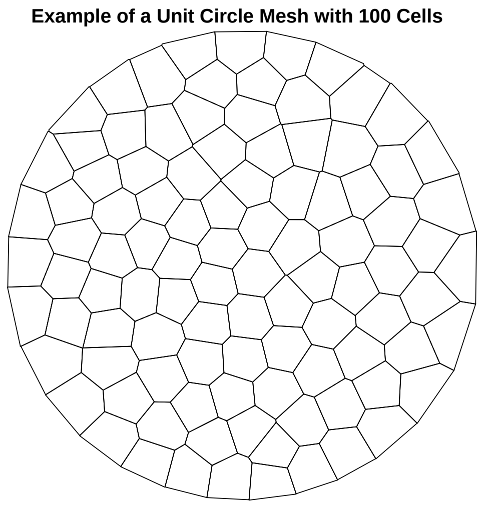

Figure 4.6: Circle Polygonal Mesh

Appendices

Appendix A

Matlab Codes: Mixed Formulation Approach

```
%-----------------------Mixed Formulation Approach------------------------%
%-------------------------------------------------------------------------%
function [uh,ph,U_exact] = Code_MFDM_MFA(Domain,u_exact,lap,NumEle,MaxIter)
[Node, CelltoVertix,~,~,P] =...
PolyMesher(Domain,NumEle,MaxIter); %Generate the mesh. See [5].
Edges = UniqueEdges(NumEle,CelltoVertix); %Determine the edges of mesh
Cell2Edge =...
CellEdgeArray(NumEle,Edges,CelltoVertix); %Determines which edges are
%a part of which cell
ExtEdges = find(mod(sum(Cell2Edge,1),2)); %Determines external edges
EdgeCenters = Centers(Node,Edges); %Find edge centers
TEMP = [Node(Edges(:,1),:),Node(Edges(:,2),:)];
EdgeLengths = sqrt((TEMP(:,1)-TEMP(:,3)).^2 +...
(TEMP(:,2)-TEMP(:,4)).^2); %Find edge lengths
Vertices = Vert(CelltoVertix,Node); %Used in determining areas of cells
AreaofP = Area(NumEle,Vertices); %Fidn area of cells
EdgeNormals = Normals(Node,Edges); %Find edge normals
Alphas = Outward(NumEle, Cell2Edge,...
EdgeNormals, EdgeCenters, P); %Determine if normal is outward to cell i
barycenters = centroid(NumEle, Vertices, P); %Find barycenters of cells
Mf = GMM(NumEle, Vertices, Cell2Edge,...
EdgeNormals, Alphas, EdgeCenters,...
barycenters, EdgeLengths, Edges); %Construct global Mf matrix
divh = divMatrix(NumEle, Cell2Edge,...
Edges, Alphas, EdgeLengths, AreaofP); %Construct divh matrix
Mp = sparse(diag(AreaofP)); %Construct Mp matrix
rhs = RHS(NumEle, ExtEdges, Edges, EdgeLengths,...
AreaofP, u_exact, lap, barycenters, EdgeCenters); %Construct RHS
A = sparse([Mf,-divh'*Mp;Mp*divh,zeros(NumEle)]); %Construct final system
%as in (20)
x = A\rhs; %Solve system
uh = x(1:length(Edges));
ph = x(length(Edges)+1:end); %Seperate solutions into uh and ph
U_exact = u_exact(barycenters(:,1),barycenters(:,2)); %Exact solution
e_h = ph-U_exact;
error = sqrt(e_h' * Mp * e_h); %Calculate error
%-------------------------------------------------------------------------%
% Vizualize the solution                                                  %
% Courtesy of Anton Evgrafov                                              %
%-------------------------------------------------------------------------%
clf; axis equal; axis off; hold on;
MaxNVer = max(cellfun(@numel,CelltoVertix)); %Max. num. of vertices in mesh
PadWNaN = @(E) [E(:)' NaN(1,MaxNVer-numel(E))]; %Pad cells with NaN
ElemMat = cellfun(PadWNaN,CelltoVertix,'UniformOutput',false);
ElemMat = vertcat(ElemMat{:}); %Create padded element matrix
```

```
patch('Faces',ElemMat,'Vertices',Node,'FaceColor',...
'flat','FaceVertexCData',ph);
colorbar;
end
%--------------------------------------------------DETERMINE UNIQUE EDGES%
function [Edges] = UniqueEdges(NumEle,CelltoVertix)
Edges = [0,0]; %Initialize
for i = 1:NumEle %For all cells
temp = vertcat(CelltoVertix{i},CelltoVertix{i}(1));
for j = 1:length(temp)-1 %For all edges of cell
    if ~ismember([temp(j),temp(j+1)],Edges,'rows') &&...
    ~ismember([temp(j+1),temp(j)],Edges,'rows')
        Edges(end+1,:) = [temp(j),temp(j+1)];
    end
end
clear temp %Not clearing causes issues for loop
end
Edges = Edges(2:end,:); %Remove the initialization
end
%-------------------------------------------------ARRAY OF CELLS TO EDGES%
function [Cell2Edge] = CellEdgeArray(NumEle,Edges,CelltoVertix)
Cell2Edge = zeros(NumEle, length(Edges));
for i = 1:NumEle %For all cells
    temp = [CelltoVertix{i},...
    vertcat(CelltoVertix{i}(2:end),CelltoVertix{i}(1))]; %Edges of cell
    for j = 1:length(Edges) %For all edges in cell
        if ismember(Edges(j,:),temp,'rows') ||...
        ismember(fliplr(Edges(j,:)),temp,'rows')
        Cell2Edge(i,j) = 1;
        end
    end
    clear temp
end
end
%------------------------------------------------------------EDGE CENTERS%
function [EdgeCenters] = Centers(Node,Edges)
EdgeCenters = zeros(length(Edges),2);
for i = 1:length(EdgeCenters)
    EdgeCenters(i,:) = 0.5*(Node(Edges(i,1),:)+Node(Edges(i,2),:));
end
end
%-----------------------------------ARRAY EASES NOTATION FOR AREA FUNCTION%
function [Vertices] = Vert(CelltoVertix,Node)
Vertices = cell(length(CelltoVertix),1);
for i = 1:length(CelltoVertix)
    Vertices{i} = Node(CelltoVertix{i},:); %Same as CelltoVertix but with
    %co-ordinates instead of nodes
end
end
%-------------------------------------------------------------AREA OF CELLS%
function [AreaofP] = Area(NumEle,Vertices)
AreaofP = zeros(NumEle,1);
for i = 1:NumEle
    AreaofP(i) = polyarea(Vertices{i}(:,1),Vertices{i}(:,2));
end
end
%---------------------------------------------------------NORMALS OF EDGES%
function [EdgeNormals] = Normals(Node,Edges)
EdgeNormals = zeros(length(Edges),2);
for i = 1:length(Edges) %For every edge
    temp = [Node(Edges(i,1),:);Node(Edges(i,2),:)]; %Co-ordinates of the
    %vertices of the edge
    EdgeNormals(i,:) = [temp(1,2)-temp(2,2),temp(2,1)-temp(1,1)];
    clear temp
end
EdgeNormals = normc(EdgeNormals')'; %Normalize
```

```
end
%-----------------------------------------------------------OUTWARD NORMALS%
function [Alphas] = Outward(NumEle, Cell2Edge, EdgeNormals, EdgeCenters, P)
Alphas = cell(NumEle,1);
for i = 1:NumEle %For every cell
    temp = find(Cell2Edge(i,:)); % Edges of cell i
        for j = 1:length(temp) %For every edge
        if dot(EdgeNormals(temp(j),:),EdgeCenters(temp(j),:)-P(i,:))>=0
            Alphas{i}(j) = 1; %If edge normal is outward to cell i
        else
            Alphas{i}(j) = -1; %Otherwise
        end
    end
end
end
%-----------------------------------------------------------------GLOBAL MF%
function [Mf] = GMM(NumEle, Vertices, Cell2Edge, EdgeNormals,...
Alphas, EdgeCenters, barycenters, EdgeLengths, Edges)
Mf = sparse(zeros(length(Edges))); %Initialize matrix
for i = 1:NumEle %For all cells
    R = zeros(length(Vertices{i}),2); %Initialize R, new for each cell i
    N = zeros(length(Vertices{i}),2); %Initialize N, new for each cell i
    temp = find(Cell2Edge(i,:)); %Edges of cell
    for j = 1: length(temp) %For each edge of cell i
        R(j,:) = Alphas{i}(j)*(EdgeCenters(temp(j),:) -...
        barycenters(i,:))*EdgeLengths(temp(j));
        N(j,:) = EdgeNormals(temp(j),:);
    end
    Size = 1/length(temp);
    M0 = R*((R'*N)\R'); %For ease of notation, precalculate M0 and M1
    M1 = Size*trace(M0)*(eye(length(temp)) - N*((N'*N)\N'));
    M = M0 + M1; %Calculate local M
    M = sparse(M);
    Assembly = zeros(length(temp),length(Edges)); %Assembly matrix
    for k = 1:length(temp)
        Assembly(k,temp(k)) = 1;
    end
    Assembly = sparse(Assembly);
    Mf = Mf + Assembly' * M * Assembly; %Update global Mf with local M
    clear temp
end
end
%---------------------------------------------------------------DIVH MATRIX%
function [divh] = divMatrix(NumEle, Cell2Edge, Edges, Alphas,...
EdgeLengths, AreaofP)
divh = (zeros(NumEle,length(Edges))); %Initialize array
for i = 1:NumEle %For every cell
    temp = find(Cell2Edge(i,:)); %Edges of cell i
    for j = 1:length(temp) %For every edge of cell i
        divh(i,temp(j)) = Alphas{i}(j)*EdgeLengths(temp(j))/AreaofP(i);
    end
end
divh = sparse(divh);
end
%-------------------------------------------------------------CALCULATE RHS%
function [rhs] = RHS(NumEle,ExtEdges,Edges,EdgeLengths,AreaofP,...
u_exact,lap,barycenters,EdgeCenters)
Gamma = zeros(length(Edges),1);
for i = ExtEdges
    Gamma(i) = EdgeLengths(i) * u_exact(EdgeCenters(i,1),EdgeCenters(i,2));
end
b = zeros(NumEle,1);
for i = 1:NumEle
    b(i) = AreaofP(i) * lap(barycenters(i,1),barycenters(i,2));
end
rhs = vertcat(Gamma,-b); %RHS of final system (20)
```

```
end
%-------------------------------------------------------------BARYCENTERS%
function [barycenters] = centroid(NumEle, Vertices, P)
barycenters = zeros(NumEle,2);
for i = 1:NumEle %For every cell
    cellarea = 0;
    temp = vertcat(Vertices{i},Vertices{i}(1,:));
    for j = 1:length(temp)-1 %Split cell into triangles to calc barycenter
        areaoftri = polyarea([temp(j:j+1,1);P(i,1)],...
        [temp(j:j+1,2);P(i,2)]);
        tricentx = sum([temp(j:j+1,1);P(i,1)])/3 ;
        tricenty = sum([temp(j:j+1,2);P(i,2)])/3 ;
        barycenters(i,:) = barycenters(i,:)+areaoftri*[tricentx,tricenty];
        cellarea = cellarea + areaoftri;
    end
    barycenters(i,:) = barycenters(i,:)/cellarea;
clear temp
end
end
```

Appendix B

Matlab Codes: Primal Form Approach

```
%---------------------------Primal Form Approach--------------------------%
%-------------------------------------------------------------------------%
function [uh,U_exact]=primalLaplace(Domain,u_exact,lap,NumEle,MaxIter)
[Node, CelltoVertix,~,~,P] =...
PolyMesher(Domain,NumEle,MaxIter); %Generate mesh. See [5]
[EdgesofP,Edges] = UniqueEdges(CelltoVertix); %Determine edges
ENrofP = EdgeNumbersP(EdgesofP,Edges); %Number and orient edges
TEMP = [Node(Edges(:,1),:),Node(Edges(:,2),:)];
EdgeLengths = sqrt((TEMP(:,1)-TEMP(:,3)).^2 +...
(TEMP(:,2)-TEMP(:,4)).^2); %Calculate edge lengths
EdgeNormals = Normals(Node,Edges); %Find edge normals
EdgeCenters = Centers(Node,Edges); %Find edge centers
[OutwardNormals] = Outward(NumEle, ENrofP,...
EdgeNormals, EdgeCenters, P); %Find outward normals of cells
Vertices = Vert(CelltoVertix,Node); %Array for ease of finding barycenters
barycenters = centroid(NumEle, Vertices, P); %Find barycenters
NormEdges = NormalizedEdges(Node,Edges); %Orthonormalize edges
EdgeOrientation = ...
Betas(EdgesofP,CelltoVertix); %Find orientation of edges for each cell
gradh = Gradient(EdgeLengths,Edges,Node); %Find gradh matrix
Mv = GMV(EdgeCenters,barycenters,OutwardNormals,EdgeLengths,ENrofP,...
CelltoVertix,Node); %Calculate Mv
Me = GME(EdgeOrientation,ENrofP,EdgeLengths,EdgeCenters,barycenters,...
NormEdges); %Calculate Me
A = gradh'*Me*gradh; %Form the matrix A for final system
Cell2Edge = CellEdgeArray(NumEle,Edges,ENrofP); %Cell-Edge adjecency
ExtEdges = find(mod(sum(Cell2Edge,1),2)); %External Edges
ExtNodes = unique(Edges(ExtEdges,:)); %External nodes
b = Mv * RHS(ExtNodes,u_exact,lap,Node); %Form RHS of final system
A1=A;
A1(:,ExtNodes)=[];
A2 = A(:,ExtNodes); %Remove known values from system
xint = Node;
xint(ExtNodes,:)=[];
xext = Node(ExtNodes,:);
U_exact = u_exact(xint(:,1),xint(:,2)); %Exact solution for internal nodes
rhs = b - A2*u_exact(xext(:,1),xext(:,2)); %Modify RHS to reflect
%the removal of known values
FinalA = A1;
FinalA(ExtNodes,:)=[];
FinalRHS = rhs;
FinalRHS(ExtNodes,:)=[];
uh = FinalA\FinalRHS; %Solve the system
errorMv = Mv;
errorMv(ExtNodes,:) = []; %Remove known nodes from Mv
errorMv(:,ExtNodes) = [];
e_h = x-U_exact;
error = sqrt(e_h' * errorMv * e_h); %Calculate error
```

```matlab
end
%-------------------------------------------------DETERMINE UNIQUE EDGES%
function [EdgesofP,Edges] = UniqueEdges(CelltoVertix)
alledges = []; %Initialize array
EdgesofP = cell(length(CelltoVertix),1); %Edges of cell P, initialize
for i = 1:length(CelltoVertix)
    temp = sort(horzcat(CelltoVertix{i},...
    [CelltoVertix{i}(2:end);CelltoVertix{i}(1)]),2); %Temp for notation
    EdgesofP{i} = temp;
    alledges = vertcat(alledges,temp); %Allocate values
end
Edges = unique(sort(alledges,2),'rows','stable'); %Determine unique edges
end
%-----------------------------------------------------------NUMBER EDGES%
function [ENrofP] = EdgeNumbersP(EdgesofP,Edges)
ENrofP = cell(length(EdgesofP),1); %Initialize
for i = 1:length(EdgesofP)
    [~,ENrofP{i}] = ismember(EdgesofP{i},Edges,'rows'); %Numbered edges
    %belonging to P
end
end
%-----------------------------------------------------FIND EDGE NORMALS%
function [EdgeNormals] = Normals(Node,Edges)
EdgeNormals = zeros(length(Edges),2); %Initialize
for i = 1:length(Edges)
    temp = [Node(Edges(i,1),:);Node(Edges(i,2),:)];
    EdgeNormals(i,:) = [temp(1,2)-temp(2,2),...
    temp(2,1)-temp(1,1)]; %Find edge normals
    clear temp %Clear for next iteration
end
EdgeNormals = normr(EdgeNormals); %Normalize
end
%-------------------------------------------------------OUTWARD NORMALS%
function [OutwardNormals] = Outward(NumEle, ENrofP, EdgeNormals,...
EdgeCenters, P)
OutwardNormals = cell(NumEle,1); %Initialize
for i = 1:NumEle
    temp = ENrofP{i};
    for j = 1:length(temp)
        if dot(EdgeNormals(temp(j),:),EdgeCenters(temp(j),:)-P(i,:))>=0
            OutwardNormals{i}(j,:) = EdgeNormals(temp(j),:);
        else %Determine if outward or inward normal for cell P
            OutwardNormals{i}(j,:) = -EdgeNormals(temp(j),:);
        end
    end
end
end
%----------------------------------------------------------EDGE CENTERS%
function [EdgeCenters] = Centers(Node,Edges)
EdgeCenters = zeros(length(Edges),2); %Initialize
for i = 1:length(EdgeCenters)
    EdgeCenters(i,:) = 0.5*(Node(Edges(i,1),:)+...
    Node(Edges(i,2),:)); %Find edge centers
end
end
%----------------------------------ARRAY EASES NOTATION FOR AREA FUNCTION%
function [Vertices] = Vert(CelltoVertix,Node)
Vertices = cell(length(CelltoVertix),1); %Initialize
for i = 1:length(CelltoVertix)
    Vertices{i} = Node(CelltoVertix{i},:); %Cell edge adjacency
end
end
%-----------------------------------------------------FIND BARYCENTERS%
function [barycenters] = centroid(NumEle, Vertices, P)
barycenters = zeros(NumEle,2); %Initialize
for i = 1:NumEle
```

```
    cellarea = 0; %Reset after each iteration
    temp = vertcat(Vertices{i},Vertices{i}(1,:));
    for j = 1:length(temp)-1
        areaoftri = polyarea([temp(j:j+1,1);P(i,1)],...
        [temp(j:j+1,2);P(i,2)]); %Calculate area of each triangle that
        %creates the cell P
        tricentx = sum([temp(j:j+1,1);P(i,1)])/3 ;
        tricenty = sum([temp(j:j+1,2);P(i,2)])/3 ;
        barycenters(i,:) = barycenters(i,:)+...
        areaoftri*[tricentx,tricenty]; %Calculate barycenter
        cellarea = cellarea + areaoftri;
    end
    barycenters(i,:) = barycenters(i,:)/cellarea;
    clear temp %Prep for new iteration
end
end
%-------------------------------------------------CALCULATE MV MASS MATRIX%
function [Mv] = GMV(EdgeCenters,barycenters,OutwardNormals,...
EdgeLengths,ENrofP,CelltoVertix,Node)
Mv = zeros(length(Node)); %Initialize
for i = 1:length(ENrofP)
    R = zeros(length(CelltoVertix{i}),1); %Initialize matrix R for cell i
    N = ones(length(CelltoVertix{i}),1); %Form matrix N for cell i
    temp = CelltoVertix{i};
    for j=1:length(temp)
        R(j) = 0.25 * dot( EdgeCenters(ENrofP{i}(j),:) -...
        barycenters(i,:) , OutwardNormals{i}(j,:) ) *...
        EdgeLengths(ENrofP{i}(j)); %Calculate elements of R for cell i
    end
    Size = 1/(length(temp)); %Find size
    M0 = R*((R'*N)\R');
    M = M0 + Size*trace(M0)*(eye(length(temp)) - N*((N'*N)\N'));
    M = sparse(M); %Calculate M and save in sparse
    Assembly = zeros(length(temp),length(Node));
    Assembly = sparse(Assembly); %Initialize assembly array
    for k = 1:length(temp)
        Assembly(k,temp(k)) = 1; %Find values of assembly array
    end
    Mv = Mv + Assembly' * M * Assembly; %Add local mass matrix to global
    %mass matrix
    clear temp
end
Mv = sparse(Mv); %Save as sparse
end
%---------------------------------------------------CALCULATE GRADH MATRIX%
function [gradh] = Gradient(EdgeLengths,Edges,Node)
gradh = zeros(length(Edges),length(Node)); %Initialize
for i = 1:length(Edges)
    gradh(i,Edges(i,1)) = -1/(EdgeLengths(i)); %Calculate elements
    gradh(i,Edges(i,2)) = 1/(EdgeLengths(i));
end
gradh = sparse(gradh); %Save as sparse
end
%-------------------------------------------------CALCULATE ME MASS MATRIX%
function [Me] = GME(EdgeOrientation,ENrofP,EdgeLengths,EdgeCenters,...
barycenters,NormEdges)
Me = zeros(length(EdgeLengths)); %Initialize
for  i = 1:length(ENrofP)
    R = zeros(length(ENrofP{i}),2); %Initialize R & N matrices for cell i
    N = zeros(length(ENrofP{i}),2);
    temp = ENrofP{i};
    for j = 1:length(temp)
        distVec = EdgeCenters(temp(j),:) - barycenters(i,:); %Distance from
        %edge to
        %barycenter
        R(j,:) = EdgeOrientation{i}(j)*[distVec(2),-distVec(1)] *...
```

```
                EdgeLengths(temp(j)); %Calculate elements of R & N for cell i
                N(j,:) = NormEdges(temp(j),:);
            end
            Size = 1/length(temp); %Find size
            M0 = R*((R'*N)\R');
            M = M0 + Size*trace(M0)*(eye(length(temp)) - N*((N'*N)\N'));
            M = sparse(M); %Calculate M and save as sparse
            Assembly = zeros(length(temp),length(EdgeLengths));
            Assembly = sparse(Assembly); %Initialize assembly matrix
            for k = 1:length(temp)
                Assembly(k,temp(k)) = 1; %Find elements of assembly matrix
            end
            Me = Me + Assembly' * M * Assembly; %Add local mass matrix to global
            %mass matrix
            clear temp %Prepare for next iteration
        end
        end
        %-----------------------------------------------------ORTHONORMALIZE EDGES%
        function [NormEdges] = NormalizedEdges(Node,Edges)
        NormEdges = zeros(length(Edges),2); %Initialize
        for i =1:length(Edges)
            start = Node(Edges(i,1),:);
            finish = Node(Edges(i,2),:);
            NormEdges(i,:) = finish-start; %find vector of the edge
        end
        NormEdges = normr(NormEdges); %Normalize the vector
        end
        %--------------------------------------------------ARRAY OF CELLS TO EDGES%
        function [Cell2Edge] = CellEdgeArray(NumEle,Edges,ENrofP)
        Cell2Edge = zeros(NumEle, length(Edges)); %Initialize
        for i = 1:NumEle
            temp = ENrofP{i};
            for j = 1:length(temp)
                Cell2Edge(i,temp(j)) = 1; %Cell edge adjacency
            end
            clear temp %Prepare for next interation
        end
        end
        %-------------------------------------------------------------RHS OF SYSTEM%
        function [rhs] = RHS(ExtNodes,u_exact,lap,Node)
        rhs = zeros(length(Node),1); %Initialize
        for i = 1:length(rhs)
            rhs(i) = lap(Node(i,1),Node(i,2)); %Calculate initial conditions
        end
        for j = ExtNodes
            rhs(j) = rhs(j) + u_exact(Node(i,1),Node(i,2)); %Calculate boundary
            %conditions
        end
        end
        %--------------------------------------------EDGE ORIENTATION FOR EACH CELL%
        function [EdgeOrientation] = Betas(EdgesofP,CelltoVertix)
        EdgeOrientation = cell(length(EdgesofP),1); %Initialize
        for i = 1:length(EdgesofP)
            temp = EdgesofP{i};
            for j = 1:length(temp)
                if CelltoVertix{i}(j) == EdgesofP{i}(j,1) %Determine orientation
                %of edge j in cell i
                    EdgeOrientation{i}(j) = 1;
                else
                    EdgeOrientation{i}(j) = -1;
                end
            end
        end
        end
```

Bibliography

[1] http://www.colorado.edu/engineering/CAS/courses.d/IFEM.d/IFEM.Ch25.d/IFEM.Ch25.pdf.

[2] Lourenco Beirao da Veiga; Konstantin Lipnikov; Gianmarco Manzini. *The Mimetic Finite Difference Method for Elliptic Problems.* Springer, 2014.

[3] Lourenco Beirao da Veiga; Konstantin Lipnikov; Gianmarco Manzini. *The Mimetic Finite Difference Method for Elliptic Problems.* Springer, 2014. Chapter 5: The Diffusion Problem in Mixed Form.

[4] Konstatin Lipnikov. Mimetic finite difference discretizations. http://math.lanl.gov/~lipnikov/Research/MimeticFD/MimeticFD.html. [Accessed April-June 2016].

[5] Cameron Talischi;Glaucio H. Paulino;Anderson Pereira;Ivan F. M. Menezes. Polymesher: A general-purpose mesh generator for polygonal elements written in matlab. *Springer*, 2012.

[6] Wikipedia. Centroid. https://en.wikipedia.org/wiki/Centroid.

[7] Wikipedia. Vector calculus identities.

[8] WolframMathWorld. Cross product.