A Two-Level Method For The Quasi-Geostrophic Equations

David Wells

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Mathematics

Traian Iliescu, Chair
Slimane Adjerid
Tao Lin

29 April 2013
Blacksburg, Virginia

Keywords: Quasi-geostrophic equations, Finite Element Method, Argyris Element,
Two-Level Method

A Two-Level Method For The Quasi-Geostrophic Equations

David Wells

(ABSTRACT)

The quasi-geostrophic equations (QGE) are a model of large-scale ocean flows. We consider a pure stream function formulation and cite results for optimal error estimates for finding approximate solutions with the finite element method. We examine both the time dependent and steady-state versions of the equations. Numerical experiments verify the error estimates.

We examine the *Argyris finite element* and derive the transformation matrix necessary to perform calculations on the reference triangle. We use the Argyris element because it is a high-order, conforming finite element for fourth order problems.

In order to increase computational efficiency, we consider a *two-level* method to linearize the system of equations. This allows us to solve a small, nonlinear system and then use the result to linearize a larger system.

Acknowledgments

I would like to thank my family for their continued support of my studies. I would not have gotten this far without their constant encouragement and wanting the best for me.

I would also like to thank my academic family at Virginia Tech. This work would not be possible without my advisor and fellow students Erich Foster and Zhu Wang. More generally, I would like to thank the Mathematics department for making Blacksburg feel like home.

Finally, on a technical note, I would like to thank the GMSH team for their fantastic meshing software and the SAGE project for advancing the use of Python for mathematics.

Contents

List of Tables

List of Figures

Chapter 1

The Quasi-Geostrophic Equations

1.1 Stream Function & Vorticity

It is common in geophysical flows to work with the *stream function* and *vorticity* rather than the velocity fields. Let v_1 be the x-component of the velocity field and v_2 be the y-component of the velocity field. We define the vorticity to be the curl of the two-dimensional flow, where [11]

$$\omega = \operatorname{curl} \vec{v} = \frac{\partial v_2}{\partial x} - \frac{\partial v_1}{\partial y}. \tag{1.1}$$

The use the term curl is slightly abusive, since we are in two dimensions. It is equivalent to the usual curl definition in three dimensions with no z derivative and no z velocity; that is,

$$\omega = \det \begin{pmatrix} \vec{i} & \vec{j} & \vec{k} \\ \dfrac{\partial}{\partial x} & \dfrac{\partial}{\partial y} & 0 \\ v_1 & v_2 & 1 \end{pmatrix} = \left(0, \quad 0, \quad \frac{\partial v_2}{\partial x} - \frac{\partial v_1}{\partial y}\right). \tag{1.2}$$

Therefore the convention is to drop the vector notation and consider the vorticity to be purely a scalar quantity. We will use the notation $\vec{w}$ to indicate the vector form and w to indicate the scalar form where appropriate.

The stream function ψ is defined so that we recover the velocity field from its derivatives; that is,

$$\left(v_1, \quad v_2\right) = \left(\frac{\partial \psi}{\partial y}, \quad -\frac{\partial \psi}{\partial x}\right). \tag{1.3}$$

Some fluid mechanics books [12] use $-\psi$ instead of ψ; we will use the version above, which is more common in geophysical work.

There are several useful properties of both the stream function and vorticity that make

them applicable to fluid flow problems. The two quantities are related by

$$\begin{aligned}\frac{\partial^2}{\partial x^2}\psi + \frac{\partial^2}{\partial y^2}\psi &= -\frac{\partial}{\partial x}v_2 + \frac{\partial}{\partial y}v_1 \\ &= -\omega.\end{aligned} \tag{1.4}$$

Additionally, the stream function was chosen in such a way that it automatically conserves mass; that is, if we wish to satisfy the incompressibility condition $\nabla \cdot \vec{v} = 0$ we have

$$\begin{aligned}\left(\frac{\partial}{\partial x}, \quad \frac{\partial}{\partial y}\right) \cdot \begin{pmatrix} v_1, & v_2 \end{pmatrix} &= \left(\frac{\partial}{\partial x}, \quad \frac{\partial}{\partial y}\right) \cdot \left(-\frac{\partial \psi}{\partial y}, \quad \frac{\partial \psi}{\partial x}\right) \\ &= -\frac{\partial^2 \psi}{\partial x \partial y} + \frac{\partial^2 \psi}{\partial y \partial x} \\ &= 0.\end{aligned} \tag{1.5}$$

1.2 Application to 2D Navier Stokes Equations

The *Navier Stokes equations* are a PDE model of fluid flow. They effectively couple conservation of momentum with conservation of mass. In their nondimensionalized form they are [13]

$$\begin{aligned}\vec{v}_t - \frac{1}{Re}\Delta\vec{v} + \vec{v} \cdot \nabla\vec{v} + \nabla p &= \vec{f} \\ \nabla \cdot \vec{v} &= 0.\end{aligned} \tag{1.6}$$

Here we use Re to be the *Reynolds number*, defined as

$$Re = \frac{LU\rho}{\mu}, \tag{1.7}$$

where L and U are the length and velocity scales and ρ and μ are the density and molecular viscosity of the fluid. The Reynolds number is dimensionless and is the ratio between the convective (that is, bulk flow) and diffusive (that is, molecular diffusion) transfers of momentum.

The stream function-vorticity formulation of the Navier Stokes equations comes from taking the two-dimensional curl of the first equation. Since $\nabla \times \nabla p = 0$, this eliminates the pressure terms. By the identity [16]

$$\vec{v} \cdot \nabla\vec{v} = \frac{1}{2}\nabla(\vec{v} \cdot \vec{v}) - (\vec{v} \times (\nabla \times \vec{v})), \tag{1.8}$$

for the momentum conservation term we have that

$$\begin{aligned}\nabla \times \vec{f} &= \nabla \times \left(\vec{v}_t - \frac{1}{Re}\Delta\vec{v} + \vec{v}\cdot\nabla\vec{v} + \nabla p\right) \\ &= w_t - \frac{1}{Re}\Delta w + \nabla \times (\vec{v}\cdot\nabla\vec{v}) + 0 \\ &= w_t - \frac{1}{Re}\Delta w + \nabla \times (\vec{v}\times\vec{w}) \\ &= w_t - \frac{1}{Re}\Delta w - \nabla\cdot(w\vec{v}) \\ &= w_t - \frac{1}{Re}\Delta w - (\nabla w)\cdot\vec{v}. \end{aligned} \tag{1.9}$$

Noting that the convection term may be rewritten in terms of stream function by

$$\begin{aligned}\begin{pmatrix} v_1, & v_2 \end{pmatrix}\cdot\nabla w &= v_1 w_x + v_2 w_y \\ &= -\psi_y w_x + \psi_x w_y \\ &= J(\psi, w) \end{aligned} \tag{1.10}$$

we have the *stream function-vorticity* form of the Navier Stokes equations, or

$$w_t - \frac{1}{Re}\Delta w - J(\psi, w) = \nabla \times \vec{f}. \tag{1.11}$$

Substituting in Equation 1.4 we obtain

$$(-\Delta\psi)_t + \frac{1}{Re}\Delta^2\psi + J(\psi, \Delta\psi) = \nabla \times \vec{f}. \tag{1.12}$$

Therefore the two-dimensional Navier Stokes equations are reducible to a single fourth-order scalar equation.

1.3 The Quasi-Geostrophic Equations

The *quasi-geostrophic equations* are a model of large-scale ocean currents which resemble the two-dimensional Navier Stokes equations written in stream function-vorticity form. Written in terms of the *barotropic vorticity* q and stream function ψ, they are

$$\begin{aligned} q_t + \frac{1}{Re}\Delta q + J(\psi, q) &= F \\ q &= -Ro\Delta\psi + y \end{aligned} \tag{1.13}$$

where Re is the Reynolds number (which conventionally uses an *eddy viscosity* to account for the larger length scales present in ocean currents), [17] and Ro is the *Rossby number*. A full derivation from scaling arguments is available in [18]. The Rossby number is defined as

$$Ro = \frac{U}{\beta L^2}, \tag{1.14}$$

where U and L are again the velocity and length scales, while β is a term resulting from the linearization of the Coriolis force.

The difference between the quasi-geostrophic equations and the Navier Stokes equations is the addition of the Coriolis force, which accounts for the rotation of the earth. This force is given by the *β-plane approximation*, given by

$$f = f_0 + \beta y. \tag{1.15}$$

Put another way, the force is approximately linear to the distance in the y coordinate, which corresponds to distance away from the equator. The nondimensionalization of this results in the reduction to just $f = y$.

We are interested in the *stream function formulation of the quasi-geostrophic equations*, which results from combining equations in 1.13 in to a single term

$$(-\Delta\psi)_t + \frac{1}{Re}\Delta^2\psi + J(\psi, \Delta\psi) - \frac{1}{Ro}\psi_x = \frac{1}{Ro}F. \tag{1.16}$$

1.4 Boundary Conditions

For our work, we will use the boundary conditions

$$\psi\Big|_{\partial\Omega} = 0, \quad \frac{\partial\psi}{\partial\vec{n}}\Big|_{\partial\Omega} = 0. \tag{1.17}$$

These correspond to the no-slip boundary condition from Navier Stokes [10]. Let (t_1, t_2) be the unit tangent vector at some point along $\partial\Omega$. For the no-slip boundary conditions we have that

$$v_1 t_1 + v_2 t_2 = 0. \tag{1.18}$$

Using the definition of stream function, this is equivalent to

$$\frac{\partial\psi}{\partial y}t_1 - \frac{\partial\psi}{\partial x}t_2 = 0 \tag{1.19}$$

or

$$\nabla\psi \cdot \Big(-t_2, \quad t_1\Big) = 0 \tag{1.20}$$

which is just the normal derivative of the stream function. Similarly, if we enforce the no-normal-flow condition, we have that

$$v_1 t_1 - v_2 t_2 = 0 \tag{1.21}$$

or

$$\frac{\partial\psi}{\partial y}t_1 + \frac{\partial\psi}{\partial x}t_2 = \nabla\psi \cdot \Big(t_1, \quad t_2\Big) = 0 \tag{1.22}$$

so the tangential derivative of ψ around a boundary is zero. Therefore ψ is constant along boundaries; the convention is to express this as $\psi = 0$.

Chapter 2

Implementation of the Argyris Polynomials on Physical Elements

2.1 Introduction

Our work involves finding *conforming* solutions to a fourth-order problem. Hence, we need to use a C^1 finite element. We opted to use the Argyris finite element because of its high rate of spatial convergence ($O(h^6)$ in the L^2 norm for sufficiently smooth problems) and simplicity of implementation compared to hierarchical elements. The most important calculation with the Argyris finite element is the mapping between the reference element and the physical element, which we discuss in detail below.

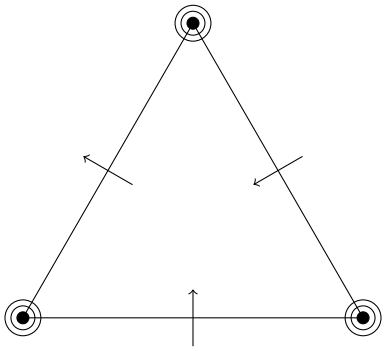

Figure 2.1: Usual depiction of the Argyris element.
Each vertex corresponds to six basis polynomials (all of the two-dimensional first and second derivatives as well as the function value) while the midpoints have the value of the normal derivatives. Guaranteeing uniqueness of the choice of direction for normal derivatives is discussed in Chapter 4.

2.2 Calculating the Basis Polynomials from Constraints

One way to describe the Argyris basis polynomials is as a set of constraints. Without loss of generality, we will work on the usual reference simplex $\{(0,0),(1,0),(0,1)\}$, but these calculations are similar for any given element. The constraints we wish to impose on each basis polynomial are

1. Like Lagrange elements, only one basis polynomial should be nonzero at each vertex. This yields three constraints.
2. At each vertex, only one basis polynomial should have a nonzero value for the first and second (mixed) derivatives. This yields 15 constraints.
3. At the midpoint of each edge, only one basis polynomial should have a nonzero normal derivative. This yields three constraints.

Using ArgyrisPack [5] and SAGE [19], we can calculate the matrix of constraints on the reference element as follows:

```
import ap.symbolic.symbolic as symb
import operator as op
import sympy

var('x, y')
xs = [0, 1, 0]
ys = [0, 0, 1]
# This is taken from symbolic.py; we actually want the matrix for
# demonstration purposes.
constants = [var('c' + str(num)) for num in range(1,22)]
monic_basis = [x**m * y**(n-m) for n in range(6)
               for m in [n-k for k in range(n+1)]]
test_polynomial = sum(map(op.mul, constants, monic_basis))
constraints = symb.constraint_system(test_polynomial, xs, ys)

constraint_matrix = matrix([symb.get_coefficients(constraint, constants)
                            for constraint in constraints])
```

Each constraint corresponds to a single basis polynomial. To get one line in the matrix of constraints, we will take the 'test polynomial'

$$t(x) = c_{16}x^5 + c_{17}x^4y + c_{18}x^3y^2 + c_{19}x^2y^3 + c_{20}xy^4 + c_{21}y^5 + c_{11}x^4 + c_{12}x^3y + c_{13}x^2y^2 + c_{14}xy^3 + c_{15}y^4 + c_{10}y^3 + c_7x^3 + c_8x^2y + c_9xy^2 + c_4x^2 + c_5xy + c_6y^2 + c_2x + c_3y + c_1$$

differentiate it appropriately, and evaluate it at the correct node. For example, the sixth constraint (the first x derivative at the second node, $(1,0)$) corresponds to the expression

$$4\,c_{11} + 5\,c_{16} + c_2 + 2\,c_4 + 3\,c_7, \tag{2.1}$$

which corresponds to the entry

$$[0,1,0,2,0,0,3,0,0,0,4,0,0,0,0,5,0,0,0,0,0] \tag{2.2}$$

in the constraint matrix.

The first 18 constraints are implemented in this manner. The normal derivatives require orientation information as well as a point. We follow Dominguez's derivation here [3]. Consider the three vertices $\vec{x}_1, \vec{x}_2$, and $\vec{x}_3$: we define the *edge vectors* to be

$$\vec{v}_1 = \vec{x}_2 - \vec{x}_1, \quad \vec{v}_2 = \vec{x}_3 - \vec{x}_1, \quad \vec{v}_3 = \vec{x}_3 - \vec{x}_2. \tag{2.3}$$

This choice guarantees uniqueness of orientation between elements as long as the node numbers are presented in increasing order (so given some element with three vertex nodes, we know that the edge vector always points from the lower-number node to the higher-number node). This is important in maintaining orientation of the normal derivatives on physical elements and will be discussed more in Section 4.2. Each normal derivative is defined to be the derivative in the direction of $\vec{v}_i$ (for $i = 1, 2, 3$) rotated $\pi/2$ radians counter-clockwise. Hence for some midpoint (x_i, y_i) we have a constraint

$$\left(\frac{d}{dx}t(x,y), \frac{d}{dy}t(x,y)\right)\Bigg|_{(x_i,y_i)} \cdot \frac{1}{|\vec{v}_i|}\vec{v}_i. \tag{2.4}$$

Therefore we have twenty one constraints, which matches the number of polynomials needed to span $\mathcal{P}^5$ on the reference element. The system of constraints is included in figure 2.2 and has a determinant of $-\frac{\sqrt{2}}{64}$. Let A be the constraint matrix. Due to the choice of constraints, we have that the coefficients of the Argyris basis polynomials are the columns of $A^{-1}I = A^{-1}$. Therefore the Argyris basis polynomials are linearly independent and therefore span $\mathcal{P}^5$ on the reference element.

2.3 C^1 Continuity of the Argyris Basis Polynomials

Before we show the C^1 continuity of the Argyris element, we need a lemma from interpolation theory. We will use this lemma again to build the change of coordinates matrix.

Lemma 2.1. *Let $\phi \in \mathcal{P}^5$ and $x_1, x_2 \in \mathbb{R}$. If*

$$\psi(t) = \phi(tx_1 + (1-t)x_2), \tag{2.5}$$

then

$$\psi'(1/2) = \frac{15}{8}(\psi(1) - \psi(0)) - \frac{7}{16}(\psi'(1) + \psi'(0)) + \frac{1}{32}(\psi''(1) - \psi''(0)). \tag{2.6}$$

Proof. Note that $\psi(t)$ is a single-variable fifth-order polynomial

$$\psi(t) = a_6t^6 + a_5t^5 + a_4t^4 + a_3t^3 + a_2t^2 + a_1t + a_0.$$

As

$$\psi'(1/2) = a_1 + a_2 + \frac{3}{4}a_3 + \frac{1}{2}a_4 + \frac{5}{16}a_5 + \frac{3}{16}a_6$$

and

$$\begin{aligned}
\frac{15}{8}(\psi(1)-\psi(0)) &= \frac{15}{8}\,a_1+\frac{15}{8}\,a_2+\frac{15}{8}\,a_3+\frac{15}{8}\,a_4+\frac{15}{8}\,a_5+\frac{15}{8}\,a_6\\
-\frac{7}{16}(\psi'(1)+\psi'(0)) &= -\frac{7}{8}\,a_1-\frac{7}{8}\,a_2-\frac{21}{16}\,a_3-\frac{7}{4}\,a_4-\frac{35}{16}\,a_5-\frac{21}{8}\,a_6\\
\frac{1}{32}(\psi''(1)-\psi''(0)) &= \frac{3}{16}\,a_3+\frac{3}{8}\,a_4+\frac{5}{8}\,a_5+\frac{15}{16}\,a_6,
\end{aligned}$$

we have that the sum of the evaluations of ψ, ψ', and ψ'' on the left is equal to $\psi'(1/2)$. □

Lemma 2.2. *If $p(t) \in \mathcal{P}^5$, then $p(t)$ is uniquely determined by specifying values*

$$p(0), p(1), p'(0), p'(1), p''(0), \text{ and } p''(1).$$

Proof. Let

$$p(t) = c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0.$$

Therefore we may represent the chosen constraints by the system

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1 & 1 & 1\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 1 & 2 & 3 & 4 & 5\\ 0 & 0 & 2 & 0 & 0 & 0\\ 0 & 0 & 2 & 6 & 12 & 20 \end{pmatrix} \begin{pmatrix} c_0\\ c_1\\ c_2\\ c_3\\ c_4\\ c_5 \end{pmatrix} = \begin{pmatrix} p(0)\\ p(1)\\ p'(0)\\ p'(1)\\ p''(0)\\ p''(1) \end{pmatrix},$$

where the matrix has determinant -4. Therefore the given set of constraints corresponds to exactly one polynomial. □

Lemma 2.3. *If $p(t) \in \mathcal{P}^4$, then $p(t)$ is uniquely determined by specifying values*

$$p(0), p(1), p'(0), p'(1), \text{ and } p(1/2).$$

Proof. Let

$$p(t) = c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0.$$

Therefore we may represent the chosen constraints by the system

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0\\ 1 & 1 & 1 & 1 & 1\\ 0 & 1 & 0 & 0 & 0\\ 0 & 1 & 2 & 3 & 4\\ 1 & \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} \end{pmatrix} \begin{pmatrix} c_0\\ c_1\\ c_2\\ c_3\\ c_4 \end{pmatrix} = \begin{pmatrix} p(0)\\ p(1)\\ p'(0)\\ p'(1)\\ p(1/2) \end{pmatrix},$$

where the matrix has determinant $-\frac{1}{16}$. Therefore the given set of constraints corresponds to exactly one polynomial. □

Proposition 2.4. *The Argyris basis polynomials are C^0.*

Proof. By Section 2.2 we have that the system of constraints yields a unique set of polynomials defined on a element. Therefore the one-dimensional polynomial along each edge exists.

Let $p(x, y)$ be a basis polynomial with support including the line segment between (x_1, y_1) and (x_2, y_2), which are two vertex nodes used to interpolate the Argyris basis polynomials. Let

$$\psi(t) = p(tx_1 + (1-t)x_2, ty_1 + (1-t)y_2)$$

so $\psi(t)$ is a fifth-order polynomial along the given edge. To make the notation clearer, let

$$\lambda(t) = (tx_1 + (1-t)x_2, ty_1 + (1-t)y_2) \tag{2.7}$$

so $\psi(t) = p(\lambda(t))$. Taking the first derivative yields

$$\psi'(t) = \Big(x_1 - x_2, \quad y_1 - y_2\Big) \cdot (\nabla p)(tx_1 + (1-t)x_2, ty_1 + (1-t)y_2). \tag{2.8}$$

Taking the second derivative yields

$$\begin{aligned}
\psi''(t) &= \Big(x_1 - x_2, \quad y_1 - y_2\Big) \cdot ((x_1 - x_2)(\nabla p_x)(\lambda(t)) + (y_1 - y_2)(\nabla p_y(\lambda(t)))) \\
&= (x_1 - x_2)^2 p_{xx}(\lambda(t)) + (x_1 - x_2)(y_1 - y_2)p_{xy}(\lambda(t)) \\
&+ (x_1 - x_2)(y_1 - y_2)p_{xy}(\lambda(t)) + (y_1 - y_2)^2 p_{yy}(\lambda(t)) \\
&= (x_1 - x_2)^2 p_{xx}(\lambda(t)) + 2(x_1 - x_2)(y_1 - y_2)p_{xy}(\lambda(t)) + (y_1 - y_2)^2 p_{yy}(\lambda(t)).
\end{aligned} \tag{2.9}$$

Therefore $\psi(0), \psi(1), \psi'(0), \psi'(1), \psi''(0)$, and $\psi''(1)$ are all specified. Therefore by Lemma 2.2 the values of the Argyris basis functions are unique. Therefore the Argyris basis polynomials are in C^0. □

Proposition 2.5. *The Argyris basis polynomials are C^1.*

Proof. It suffices to show that on a given edge that the polynomials interpolating the normal derivative and the tangential derivative are both unique. This is because we may reconstruct the x and y derivatives as linear combinations of the normal and tangential derivatives. Let 2.7, 2.8, and 2.9 be as in the C^0 proof. Consider the derivative in the tangential direction (that is, the direction of increasing t) first. Then by the C^0 proof and Lemma 2.1

$$\psi'(0), \psi'(1/2), \psi'(1), \psi''(0), \text{ and } \psi''(1)$$

are all specified. Therefore by Lemma 2.3 the tangential derivative is unique.

Now consider the derivative in the normal direction. Let $\phi(t)$ be the polynomial interpolating values of the normal derivative along the edge parameterized by t, so

$$\phi(t) = (\nabla p)(\lambda(t)) \cdot \Big(x_2 - x_1, \quad y_1 - y_2\Big).$$

Therefore $\phi(0), \phi(1)$, and $\phi(1/2)$ are all specified. The second derivative is specified by

$$\begin{aligned}
\phi'(t) &= \Big(x_1 - x_2, \quad y_1 - y_2\Big) \cdot ((x_2 - x_1)(\nabla p_x)(\lambda(t)) + (y_1 - y_2)(\nabla p_y(\lambda(t)))) \\
&= (x_2 - x_1)(x_1 - x_2)p_{xx}(\lambda(t)) + (x_2 - x_1)(y_1 - y_2)p_{xy}(\lambda(t)) \\
&+ (x_1 - x_2)(y_1 - y_2)p_{xy}(\lambda(t)) + (y_1 - y_2)^2 p_{yy}(\lambda(t)) \\
&= -(x_1 - x_2)^2 p_{xx}(\lambda(t)) + (y_1 - y_2)^2 p_{yy}(\lambda(t)).
\end{aligned}$$

Therefore

$$\phi'(0) = -(x_1 - x_2)^2 p_{xx}(x_2, y_2) + (y_1 - y_2)^2 p_{yy}(x_2, y_2)$$

and

$$\phi'(1) = -(x_1 - x_2)^2 p_{xx}(x_1, y_1) + (y_1 - y_2)^2 p_{yy}(x_1, y_1).$$

Combining the last few results we have the specified values

$$\phi(0), \phi(1/2), \phi(1), \phi'(0), \text{ and } \phi'(1),$$

so $\phi(t)$ is unique. Therefore the derivatives in both the tangential and normal directions are unique. Therefore the derivatives of the basis polynomials are in C^0. Therefore, together with Proposition 2.4, each basis polynomial is in C^1. □

2.4 Change of Coordinates Matrix

2.4.1 The Dual Space Transformation

This work follows Dominguez et al [2] closely. Let $\phi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Define the linear functionals

$$\begin{aligned}
\mathcal{L}_i^0(\phi) &= \phi(\vec{x}_i) \\
\mathcal{L}_i^x(\phi) &= \phi_x(\vec{x}_i) \\
\mathcal{L}_i^y(\phi) &= \phi_y(\vec{x}_i) \\
\mathcal{L}_i^{xx}(\phi) &= \phi_{xx}(\vec{x}_i) \\
\mathcal{L}_i^{xy}(\phi) &= \phi_{xy}(\vec{x}_i) \\
\mathcal{L}_i^{yy}(\phi) &= \phi_{yy}(\vec{x}_i) \\
\mathcal{L}_i^n(\phi) &= (\nabla\phi)(\vec{m}_i) \cdot \frac{\vec{v}_i}{|\vec{v}_i|}
\end{aligned}$$

for vertices $\vec{x}_i$, midpoints $\vec{m}_i$ and $i = 1, 2, 3$. Note that this form a basis for the dual space of the Argyris polynomials (and hence they are linearly independent) [14]. For convenience, number these functionals as $\mathcal{L}_j$ for $j = 1, 2, \ldots, 21$ with the enumeration

$$\begin{aligned}
&\mathcal{L}_1^0, \mathcal{L}_2^0, \mathcal{L}_3^0 \\
&\mathcal{L}_1^x, \mathcal{L}_1^y, \mathcal{L}_2^x, \mathcal{L}_2^y, \mathcal{L}_3^x, \mathcal{L}_3^y \\
&\mathcal{L}_1^x x, \mathcal{L}_1^x y, \mathcal{L}_1^y y, \mathcal{L}_2^x x, \mathcal{L}_2^x y, \mathcal{L}_2^y y, \mathcal{L}_3^x x, \mathcal{L}_3^x y, \mathcal{L}_3^y y \\
&\mathcal{L}_1^n, \mathcal{L}_2^n, \mathcal{L}_3^n.
\end{aligned}$$

Let $N_j(x, y)$ be a basis polynomial on the physical element and $\hat{N}_j(x, y)$ be a basis polynomial on the reference element. Let $F\vec{x} = B\vec{x} + b$ be the affine transformation between coordinates on the reference element and coordinates on the physical element. By construction $\mathcal{L}_i(N_j) = \delta_{ij}$, so each linear functional corresponds to a row in the constraint matrix from the previous section.

Let $\hat{\mathcal{L}}_j$ be the corresponding functional on the reference element. Let $\tilde{\mathcal{L}}_j(\phi) = \hat{\mathcal{L}}_j(\phi \circ F)$. Hence by definition $\tilde{\mathcal{L}}_j(\phi \circ F^{-1}) = \hat{\mathcal{L}}_j(\phi)$. By construction, both $\{\mathcal{L}_j\}$ and $\left\{\tilde{\mathcal{L}}_j\right\}$ are bases of the dual space of $\mathcal{P}^5$, so there exists some matrix C such that

$$\tilde{\mathcal{L}}_j = \sum_{i=1}^{21} c_{ij}\hat{\mathcal{L}}_i. \tag{2.10}$$

Therefore, converting back to the space of polynomials (note the transposition operation) we wish to show that

$$N_j \circ F = \sum_{i=1}^{21} c_{ji}\hat{N}_i. \tag{2.11}$$

At this stage it is worth noting that we have an expression for a basis function defined on the physical element which takes coordinates on the reference element as an argument; this is exactly what we want to find for efficient computations using precomputed values of basis polynomials.

2.4.2 Calculating the Matrix C

Let R be the 2×2 matrix corresponding to a rotation of $\pi/2$ radians in the counter-clockwise direction. We will decompose the last three functionals on the physical element into perpendicular and normal components. Let $\mathcal{L}_j^* = \mathcal{L}_j$ for $j \leq 18$. Let

$$\mathcal{L}_i^{\perp}(\phi) = (\nabla\phi)(\vec{x}_i) \cdot R\vec{v}_i \tag{2.12}$$

and

$$\mathcal{L}_i^{\parallel}(\phi) = (\nabla\phi)(\vec{x}_i) \cdot \vec{v}_i. \tag{2.13}$$

Let $\mathcal{L}_j^*$ for $j = 19$ to $j = 24$ be $\mathcal{L}_1^{\perp}, \mathcal{L}_2^{\perp}, \mathcal{L}_3^{\perp}, \mathcal{L}_1^{\parallel}, \mathcal{L}_2^{\parallel}$, and $\mathcal{L}_3^{\parallel}$.

Proposition 2.6. *If $\mathcal{P}$ is the space of bivariate polynomials and $\mathcal{P}^*$ is its dual space, then*

$$\tilde{L}_i = \sum_{j=1}^{24} D_{ij}\mathcal{L}_j^*. \tag{2.14}$$

Proof. By the chain rule we have that

$$\nabla(\phi \circ F) = B^T(\nabla\phi) \circ F.$$

Similarly, for the second derivatives (sometimes called the *condensed Hessian*, or $\mathcal{H}(\phi) = (\phi_{xx}, \phi_{xy}, \phi_{yy})^T$), by the chain rule

$$\mathcal{H}(\phi \circ F) = \Theta(\mathcal{H}(\phi) \circ F)$$

where Θ is the second derivative analogue of B^T, so

$$\Theta = \begin{pmatrix} B_{11}^2 & 2B_{11}B_{21} & B_{21}^2 \\ B_{12}B_{11} & B_{12}B_{21} + B_{11}B_{22} & B_{21}B_{22} \\ B_{12}^2 & 2B_{22}B_{12} & B_{22}^2 \end{pmatrix}.$$

Finally, let

$$\begin{pmatrix} \vec{a}_1^T \\ \vec{a}_2^T \\ \vec{a}_3^T \end{pmatrix} = \begin{pmatrix} |\vec{v}_1|^{-2} & 0 & 0 \\ 0 & |\vec{v}_2|^{-2} & 0 \\ 0 & 0 & |\vec{v}_3|^{-2} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \\ -1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix} B^T.$$

Let $\vec{f_i} = \vec{a_i} \cdot R\vec{a}_i$ and $\vec{g}_i = \vec{a_i} \cdot \vec{v}_i$. Let

$$Q = \begin{pmatrix} f_1 & 0 & 0 & g_1 & 0 & 0 \\ 0 & f_2 & 0 & 0 & g_2 & 0 \\ 0 & 0 & f_3 & 0 & 0 & g_3 \end{pmatrix}.$$

Let D be the 21×24 block-diagonal matrix formed by the combination of I_3, B^T, B^T, B^T, Θ, Θ, Θ, and Q.

Recall the original ordering where the subscript i referred to corner number. Note that

$$\tilde{\mathcal{L}}_i^0 = \mathcal{L}_i^0, \quad \begin{pmatrix} \tilde{\mathcal{L}}_i^x \\ \tilde{\mathcal{L}}_i^y \end{pmatrix} = B^T \begin{pmatrix} \mathcal{L}_i^x \\ \mathcal{L}_i^y \end{pmatrix}, \quad \begin{pmatrix} \tilde{\mathcal{L}}_i^{xx} \\ \tilde{\mathcal{L}}_i^{xy} \\ \tilde{\mathcal{L}}_i^{yy} \end{pmatrix} = \Theta \begin{pmatrix} \mathcal{L}_i^{xx} \\ \mathcal{L}_i^{xy} \\ \mathcal{L}_i^{yy} \end{pmatrix}$$

because of the definition of $\tilde{\mathcal{L}}_j$. Similarly, based on the definition of F (and hence B), $\vec{v}_i = B\hat{\vec{v}}_i$. Therefore

$$\begin{aligned} \tilde{\mathcal{L}}_i^n(\phi) &= \hat{\mathcal{L}}_i^n(\phi \circ F) \\ &= \vec{n}_i \cdot \nabla(\phi \circ F)(\hat{m}_i) \\ &= \frac{1}{|\hat{\vec{v}}_i|} R\hat{\vec{v}}_i \nabla(\phi \circ F)(\hat{m}_i) \\ &= \frac{1}{|\hat{\vec{v}}_i|} R\hat{\vec{v}}_i B^T \nabla(\phi)(m_i). \end{aligned}$$

By construction we have (that is, undoing the rotation)

$$\begin{pmatrix} \mathcal{L}_i^\perp \\ \mathcal{L}_i^\parallel \end{pmatrix} = \begin{pmatrix} -v_i^y & v_i^x \\ v_i^x & v_i^y \end{pmatrix} \begin{pmatrix} \mathcal{L}_i^x \\ \mathcal{L}_i^y \end{pmatrix}$$

so

$$\frac{1}{|\hat{\vec{v}}|^2} \begin{pmatrix} -v_i^y & v_i^x \\ v_i^x & v_i^y \end{pmatrix} \begin{pmatrix} \mathcal{L}_i^x \\ \mathcal{L}_i^y \end{pmatrix} = \begin{pmatrix} \mathcal{L}_i^\perp \\ \mathcal{L}_i^\parallel \end{pmatrix}.$$

Combining the last two results we have that

$$\tilde{\mathcal{L}}_i^n = \frac{1}{|\hat{\vec{v}}_i|^3} R\hat{\vec{v}}_i \cdot B^T \begin{pmatrix} -v_i^y & v_i^x \\ v_i^x & v_i^y \end{pmatrix} \begin{pmatrix} \mathcal{L}_i^\perp \\ \mathcal{L}_i^\parallel \end{pmatrix}.$$

Due to how we defined f_i and g_i we have that

$$\tilde{\mathcal{L}}_i^n = f_i \mathcal{L}_i^\perp + g_i \mathcal{L}_i^\parallel$$

so the constructed operator D has the desired property. □

This proves that we have one linear operator, D, that maps $\left\{\mathcal{L}_j^*\right\} \to \left\{\tilde{\mathcal{L}}_j\right\}$. The last step is to map $\mathcal{L}_j \to \mathcal{L}_j^*$ so that we have a complete transformation between the reference functionals (that is, the reference basis polynomials and their derivatives evaluated at quadrature points) and the physical basis polynomials evaluated on the affine transformation of the same quadrature points.

Proposition 2.7. *There exists some operator E such that*

$$\mathcal{L}_i^* = \sum_{j=1}^{21} e_{ij}\mathcal{L}_j, \quad i = 1, 2, \ldots, 24. \tag{2.15}$$

Proof. Let E be the 24×21 matrix with structure

$$E = \begin{pmatrix} I_{18} & 0 \\ 0 & L \\ T & 0 \end{pmatrix}$$

where L is the diagonal matrix formed from $|\vec{v}_1|, |\vec{v}_2|$, and $|\vec{v}_3|$, while T is the 3×18 matrix composed of the three subblocks

$$\frac{15}{18}\begin{pmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \\ 0 & -1 & 1 \end{pmatrix}, \quad \frac{-7}{16}\begin{pmatrix} \vec{v_1^T} & \vec{v_1^T} & \vec{0} \\ \vec{2}^t & \vec{0} & \vec{v_2^T} \\ \vec{0} & \vec{v_3^T} & \vec{v_3^T} \end{pmatrix}, \quad \frac{1}{32}\begin{pmatrix} -\vec{w_1^T} & \vec{w_1^T} & \vec{0} \\ -\vec{w_2^T} & \vec{0} & \vec{w_2^T} \\ \vec{0} & -\vec{w_3^T} & \vec{w_1^T} \end{pmatrix}$$

where

$$\vec{w_i^T} = \left((v_i^x)^2, \quad 2v_i^x, \quad v_i^y(v_i^y)^2\right).$$

By definition, $\mathcal{L}_j^* = \mathcal{L}_j$ for $j \leq 18$. By definition of the perpendicular functionals we also have that $\mathcal{L}_i^\perp = |\vec{v}_i|\mathcal{L}_i^n$, which corresponds to the diagonal submatrix L in E.

For some edge $v_\gamma = x_\beta - x_\alpha$, $\alpha < \beta$, by repeated application of the chain rule we have that

$$\begin{aligned}
\psi'(1/2) &= \mathcal{L}_\gamma^\perp(\phi) \\
\psi(0) &= \mathcal{L}_\alpha^0(\phi) \\
\psi(1) &= \mathcal{L}_\beta^0(\phi) \\
\psi'(0) &= v_\gamma^x\mathcal{L}_\alpha^x(\phi) + v_\gamma^y\mathcal{L}_\alpha^y(\phi) \\
\psi'(1) &= v_\gamma^x\mathcal{L}_\beta^x(\phi) + v_\gamma^y\mathcal{L}_\beta^y(\phi) \\
\psi''(0) &= (v_\gamma^x)^2\mathcal{L}_\alpha^{xx}(\phi) + 2v_\gamma^xv_\gamma^y\mathcal{L}_\alpha^{xy}(\phi) + (v_\gamma^y)^2\mathcal{L}_\alpha^{yy}(\phi) \\
\psi''(1) &= (v_\gamma^x)^2\mathcal{L}_\beta^{xx}(\phi) + 2v_\gamma^xv_\gamma^y\mathcal{L}_\beta^{xy}(\phi) + (v_\gamma^y)^2\mathcal{L}_\beta^{yy}(\phi).
\end{aligned}$$

Therefore, by Lemma 2.1

$$\begin{aligned}
\mathcal{L}_\gamma^\parallel = &\frac{15}{8}(-\mathcal{L}_\alpha^0 + \mathcal{L}_\beta^0) - \frac{7}{16}(v_\gamma^x\mathcal{L}_\alpha^x + v_\gamma^y\mathcal{L}_\alpha^y + v_\gamma^x\mathcal{L}_\beta^x + v_\gamma^y\mathcal{L}_\beta^y) \\
&+ \frac{1}{32}((v_\gamma^x)^2\mathcal{L}_\alpha^{xx} + 2v_\gamma^xv_\gamma^y\mathcal{L}_\alpha^{xy} + (v_\gamma^y)^2\mathcal{L}_\alpha^{yy}(v_\gamma^x)^2\mathcal{L}_\beta^{xx} + 2v_\gamma^xv_\gamma^y\mathcal{L}_\beta^{xy} + (v_\gamma^y)^2\mathcal{L}_\beta^{yy})
\end{aligned}$$

which is the same (when written in system form) as the submatrix T in E. □

Combining 2.14 and 2.15 we obtain the transformation

$$DE : \mathcal{L}_j \to \tilde{\mathcal{L}}_j.$$

Let $C = DE$. Then by Theorem 3.23 in [14], as $\{\mathcal{L}_j\}$ is a dual basis for the polynomials defined on the physical element ($\{N_j\}$) and $\left\{\tilde{\mathcal{L}}_j\right\}$ is a dual basis for $\left\{\hat{N}_j \circ F^{-1}\right\}$ we have that (for $\vec{x} = F(\hat{\vec{x}})$)

$$N_i(\vec{x}) = \sum_{j=1}^{21} (C^T)_{ji} \hat{N}_j(F^{-1}(\vec{x})).$$

Therefore, by definition of $\hat{\vec{x}}$

$$N_i(F(\hat{\vec{x}})) = \sum_{j=1}^{21} (C^T)_{ji} \hat{N}_j(\hat{\vec{x}})$$

or simply

$$N_i \circ F = \sum_{j=1}^{21} (C^T)_{ji} \hat{N}_j. \tag{2.16}$$

2.4.3 Practical Computations

Now that we have the transformation C^T, we may work exclusively with precomputed reference values (that is, we may rewrite the values of physical basis polynomials as linear combinations of reference value functions at the quadrature points). Let $\hat{\vec{\phi}}(\hat{x})$be the 21 reference basis polynomials and $\vec{\phi}(x)$ be the 21 physical basis functions. Then

$$\vec{\phi}(x) = C^T \hat{\vec{\phi}}(F^{-1}(x)) \tag{2.17}$$

$$\nabla\vec{\phi}(x) = C^T B^{-1} \left(\nabla\hat{\vec{\phi}}\right)(F^{-1}(x)) \tag{2.18}$$

$$[\vec{\phi}_{xx}, \vec{\phi}_{xy}, \vec{\phi}_{yy}](x) = C^T \Theta^{-1} [\hat{\vec{\phi}}_{xx}, \hat{\vec{\phi}}_{xy}, \hat{\vec{\phi}}_{yy}](F^{-1}(x)) \tag{2.19}$$

In particular, given some set of quadrature points, we only need to compute values of the Argyris basis polynomials once. Given a physical element, we can calculate the values of the basis polynomials by forming the necessary coordinate transformations (C^T, B, and Θ) and multiplying.

We used this structure to write a fast implementation of the necessary translations to form common local matrices called *ArgyrisPack*. The part pertaining to evaluating the polynomials at quadrature points has the following levels:

1. Level 1: local functions; we use the set of Argyris polynomials defined on the reference simplex to compute values of the functions, first derivatives, and second derivatives.

2. Level 2: global functions; we use the transformations detailed above to compute the values of the physical basis polynomials at their appropriate quadrature points.

3. Level 3: We use the level 2 functions to compute common per-element matrices (the mass, stiffness, and 'biharmonic' or 'bending matrices in particular).

2.5 Conclusions

The C^T matrix is described in terms of about a dozen smaller matrices. Forming and multiplying these matrices out is fairly wasteful. We opted to multiply everything symbolically and then translate the resulting symbolic matrix into C code instead. Therefore, in conclusion, the only difference between working on a physical element with Lagrange basis polynomials and one with Argyris basis polynomials is just an extra matrix multiply; assembling the global mass and stiffness matrices is not appreciably slower.

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

Figure 2.2: Constraint matrix for the Argyris basis polynomials.

Chapter 3

Implementation of Boundary Conditions for the Argyris Element

3.1 Introduction

Enforcing boundary conditions with Hermite basis functions requires a more general method than the usual Lagrange formulation.

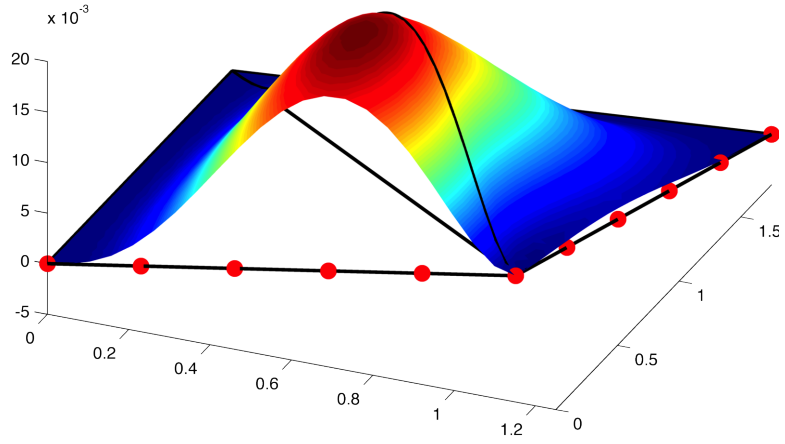

Figure 3.1: Example of a basis function that is zero at the nodes but not along the edges.

Recall that on the Argyris element some basis functions capture the value of the solution at the nodes, while others capture the values of *derivatives* of the solution at the nodes. Therefore the three common choices for boundary conditions (on boundary $\partial\Omega$)

$$u_h\Big|_{\partial\Omega} = 0, \frac{du_h}{d\vec{n}}\Big|_{\partial\Omega} = 0, \text{ or possibly } \Delta u_h\Big|_{\partial\Omega} = 0 \tag{3.1}$$

all act like 'essential' boundary conditions in the sense that they correlate with weights on specific boundary functions at the node points.

3.2 Implementing The Boundary Conditions as Constraints

As mentioned before, we wish to implement the no-slip boundary conditions. Consider the homogeneous boundary condition: we wish that the sum of the basis functions be zero along the boundary, or for N basis functions

$$u_h\Big|_{\partial\Omega} = \sum_{j=1}^{N} \phi_j\Big|_{\partial\Omega} = 0. \tag{3.2}$$

In particular, consider one edge of one element $\partial\Omega_i$. Since we have order five polynomials, they act like one-dimensional order five polynomials when restricted to this edge. Hence, if we force them to interpolate a value of zero at six points, then by uniqueness the polynomial must be zero along the whole edge. This gives us six constraints

$$\sum_{j=1}^{21} \phi_j(x_k, y_k) = 0, k = 1, 2, \ldots, 6, (x_k, y_k) \in \partial\Omega_i. \tag{3.3}$$

Similarly, to enforce the condition on the normal derivative, we have that

$$\sum_{j=1}^{21} \frac{\partial\phi_j}{\partial\vec{n}}(x_k, y_k) = 0, k = 1, 2, \ldots, 6, (x_k, y_k) \in \partial\Omega_i, \tag{3.4}$$

where $\vec{n}$ is the unit normal vector relative to the given edge. This approach yields a matrix of constraints B such that

$$B\vec{u}_h = \vec{0}, \tag{3.5}$$

where $\vec{u}_h$ is the vector weights on each basis function. Suppose that we also have some (singular) matrix A resulting from the finite element formulation such that we wish to impose

$$A\vec{u}_h = l. \tag{3.6}$$

For example, in the case of Poisson's equation, A corresponds to the matrix of inner products of the form $(\nabla\phi_i, \nabla\phi_j)$ (and hence it is singular because it does not contain any boundary data).

3.3 Solving the System

We may rewrite this problem posed by Equation 3.5 and Equation 3.6 in its variational form as

$$L(\vec{u}_h, \lambda) = \frac{1}{2}\vec{u}_h^T A\vec{u}_h - \vec{u}_h^T l + \lambda^T Bu. \tag{3.7}$$

Hence, setting the derivatives of each system equal to zero we obtain

$$\frac{\partial L}{\partial \vec{u}_h} = A\vec{u}_h - l + B^T\lambda = 0$$
$$\frac{\partial L}{\partial \lambda} = Bu = 0.$$

Finding a fixed point (we are not guaranteed to find a minimum because A may not be symmetric positive definite) of L corresponds to satisfying our system and our constraints with an extra penalty term $B^T\lambda$. This yields the system

$$\begin{pmatrix} A & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} \vec{u}_h \\ \lambda \end{pmatrix} = \begin{pmatrix} l \\ 0 \end{pmatrix}. \tag{3.8}$$

The block matrix of zeros in the bottom right causes some issues with numerical stability (particularly with LU factorization algorithms), so in practice we use the penalty method

$$\begin{pmatrix} A & B^T \\ B & \epsilon \end{pmatrix} \begin{pmatrix} \vec{u}_h \\ \lambda \end{pmatrix} = \begin{pmatrix} l \\ 0 \end{pmatrix} \tag{3.9}$$

where ϵ is a diagonal matrix with entries on the order of machine precision.

Chapter 4

Mesh Generation for Argyris Finite Elements

4.1 Introduction

In order to generate a mesh for use in a finite element code, we need to solve the following problems:

1. The elements need to be oriented in the same way (see below) due to the normal derivatives.
2. We must have a way to 'lift' quadratic or linear meshes (which are easy to generate with already-written software) to Argyris meshes.
3. We need a way to extract the boundary information in a way that is relevant to our needs for enforcing the boundary conditions.

The meshing program is also part of ArgyrisPack and is simple to use. The code

```
import ap.mesh.meshes as meshes

argyris_mesh = meshes.mesh_factory('unitsquare.mesh', argyris=True)
argyris_mesh.savetxt()
```

is all that is required to convert a quadratic mesh to an Argyris mesh and save the output in a standard representation.

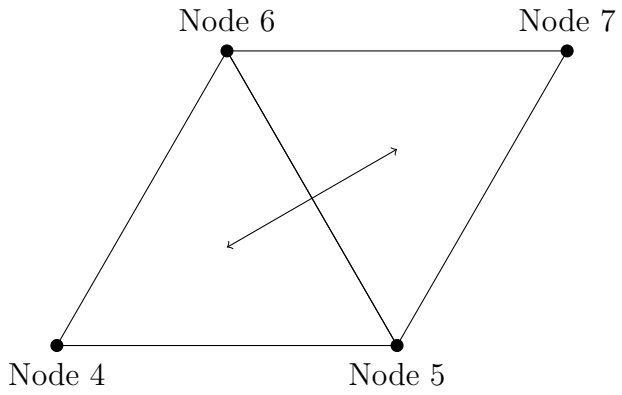

Figure 4.1: Example of incorrectly oriented triangles.
If the first triangle is $(4, 5, 6)$ and the second is $(6, 5, 7)$, then the normal derivatives along the common edge will not point in the same direction.

4.2 Resolving Orientation Issues

Note that the last three basis functions on a quadratic mesh correspond to midpoints. Therefore, we will use these basis functions for the Argyris basis functions corresponding to normal derivatives (which are also located at the midpoints). Recall that we defined the normal derivative to be the derivative in the direction indicated by taking the current edge vector and rotating $\pi/2$ radians. For example, consider the element given by corners $((x_1, y_1), (x_2, y_2), (x_3, y_3))$. This gives us three edge vectors (using the standard Argyris ordering [3])

$$(x_2 - x_1, y_2 - y_1), (x_3 - x_1, y_3 - y_1) \text{ and } (x_3 - x_2, y_3 - y_2).$$

Note that if we picked a different ordering of the *same* triangle then we would have the edge vectors

$$(x_2 - x_3, y_2 - y_3), (x_1 - x_3, y_1 - y_3) \text{ and } (x_1 - x_2, y_1 - y_2)$$

which correspond to normal derivatives in the opposite direction. Hence, we need a consistent method of ordering nodes to guarantee that between elements the normal derivative functions are being computed the same way.

Our solution is as follows: recall the steps used to create the E matrix in the change of coordinates mapping. We picked the normal vectors from edges $x_\beta - x_\alpha$ where $\beta > \alpha$. Hence we can avoid orientation issues if we renumber the corners in increasing order. Note that in the derivation we always have the higher-order node minus the lower order node; if we enforce this globally, then orientations will be unique.

For example, consider the elements with a common edge described by global nodes $(3, 4, 7)$ and $(1, 3, 4)$. In the first element, the first edge is described by $(x_4 - x_3, y_4 - y_3)$ (second node minus first node). In the second element, the last edge is described by the same value (third node minus second node).

4.3 Converting a Quadratic Mesh to a Argyris Mesh

There are many freely available programs that will generate unstructured meshes with linear or quadratic basis functions; in particular, we used GMSH to generate unstructured meshes [9]. Our goal is to build upon this previous work and write a computer program which can take a quadratic mesh as input and return an Argyris mesh as output.

Algorithm 1 Algorithm for adding extra nodes to a quadratic mesh.

Step 1: Examine the quadratic mesh and create a unique, sorted list of the node numbers corresponding to corners. These may not be sequential.

Step 2: Let N be the number of *total* nodes in the quadratic mesh plus one. This is where we will start numbering the new nodes.

Step 3: For the ith entry (starting from zero) in the list of corner nodes, create a new list of five nodes corresponding to the five derivative-information nodes on each corner. Give these new nodes numbers $N + 5i, N + 5i + 1, \ldots, N + 5i + 5$.

In Python, this algorithm is (when storing new entries in a hash table) for an already known Lagrange mesh of quadratic elements

```
# code taken from ap.mesh.meshes.py, lines 267 to 271
max_lagrange_mesh = lagrange_mesh.elements.max() + 1;
self.stacked_nodes = \
{node_number : np.arange(max_lagrange_mesh + 5*count,
                         max_lagrange_mesh + 5*count + 5)
 for count, node_number in enumerate(np.unique(self.elements[:,0:3]))}
```

Hence, we can form the classic 'element connectivity matrix' by taking an element, extracting the corners, and then correlating the corners with the 'stacked nodes' computed above.

4.4 Recording Boundary Information

Finally, we require a way to describe the boundary in terms of elements and not just nodes. Recall from 3.2 that we wish to impose the constraint

$$\sum_{j=1}^{21} \phi_j(x_k, y_k) = 0, k = 1, 2, \ldots, 6, (x_k, y_k) \in \partial\Omega_i.$$

This requires knowledge of the element as well as knowledge of the current edge. Hence, we consider the following three edge types:

1. The first and second nodes coupled with their midpoint (the 19th basis function).
2. The first and third nodes coupled with their midpoint (the 20th basis function).

3. The second and third nodes coupled with their midpoint (the 21st basis function).

Therefore every edge corresponds to the physical element number and the edge type as well as the midpoint. Hence, given an Argyris mesh, we store *all* the edge information available and index the edges by their (unique) midpoints. Therefore on a given boundary (where the nodes are known) we can extract all the edges by retrieving one edge per midpoint. This is sufficient information to determine the values of the basis functions along the edge during construction of the matrix of constraints.

Chapter 5

A Finite Element Formulation for the Quasi-Geostrophic Equations

5.1 Overview

The quasi-geostrophic equations in pure stream function form are written as a single fourth order, nonlinear partial differential equation. We are interested in finding conforming approximations to solutions of the equation by the finite element method.

5.2 Weak Formulation

Let

$$X = H_0^2(\Omega) = \left\{ \psi \in H^2(\Omega) : \psi = \frac{\partial \psi}{\partial \vec{n}} = 0 \,\text{on}\, \partial\Omega \right\} \tag{5.1}$$

where $H^2(\Omega)$ is the space of functions [4] with first and second weak derivatives. Multiplying by a test function $\phi \in H_0^2(\Omega)$ and integrating over the desired domain Ω we have that the quasi-geostrophic equations become

$$\int_\Omega (-\Delta\psi)_t \phi dx + \frac{1}{Re}\int_\Omega \Delta^2\psi\phi dx + \int_\Omega J(\psi, \Delta\psi)\phi dx - \frac{1}{Ro}\int_\Omega \psi_x \phi dx = \frac{1}{Ro}\int_\Omega F\phi dx. \tag{5.2}$$

By integration by parts we have that

$$\begin{aligned}
\int_\Omega (-\Delta\psi)_t \phi dx &= \int_\Omega \nabla \cdot (-\nabla\psi)_t \phi dx \\
&= -\int_\Omega (-\nabla\psi)_t \cdot (\nabla\phi) dx + \int_{\partial\Omega} (-\nabla\psi)\cdot\vec{n}\phi dS \\
&= \int_\Omega (\nabla\psi)_t \cdot (\nabla\phi) dx
\end{aligned}$$

as $\nabla\psi \cdot \vec{n} = 0$ is a given boundary condition. Similarly, for the biharmonic term

$$\begin{aligned}\int_\Omega \Delta^2\psi\phi dx &= \int_\Omega \nabla\cdot\nabla(\Delta\psi)\phi dx \\ &= -\int_\Omega \nabla(\Delta\psi)\cdot\nabla\phi dx + \int_{\partial\Omega}\nabla(\Delta\psi)\cdot\vec{n}\phi dS \\ &= -\int_\Omega \nabla(\Delta\psi)\cdot\nabla\phi dx \\ &= \int_\Omega (\Delta\psi)(\Delta\phi)dx + \int_{\partial\Omega}(\Delta\psi)(\nabla\phi)\cdot\vec{n}dS \\ &= \int_\Omega (\Delta\psi)(\Delta\phi)dx,\end{aligned}$$

where the last integral around the border cancels by the condition that $\phi|_{\partial\Omega} = 0$.

Integrating the Jacobian term by parts requires additional expansion. Recall that

$$J(f,g) = \frac{\partial f}{\partial x}\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\frac{\partial g}{\partial x}.$$

Hence, applying the condition that test functions are zero on $\partial\Omega$, we have that

$$\begin{aligned}\int_\Omega J(\psi,\Delta\psi)\phi dx &= \int_\Omega (\psi_x\Delta\psi_y - \psi_y\Delta\psi_x)\phi dx \\ &= \int_\Omega \psi_x\Delta\psi_y\phi dx - \int_\Omega \psi_y\Delta\psi_x\phi dx \\ &= -\int_\Omega \psi_x\Delta\psi\phi_y dx + \int_{\partial\Omega}\psi_x\Delta\psi\phi dS + \int_\Omega \psi_y\Delta\psi\phi_x dx - \int_{\partial\Omega}\psi_y\Delta\psi\phi dS \\ &= -\int_\Omega \psi_x\Delta\psi\phi_y dx + \int_\Omega \psi_y\Delta\psi\phi_x dx \\ &= \int_\Omega \Delta\psi(\psi_y\phi_x - \psi_x\phi_y)dx \\ &= \int_\Omega \Delta\psi J(\phi,\psi)dx.\end{aligned}$$

Therefore we have the weak form, where $\forall\phi \in H_0^2(\Omega)$ a solution ψ satisfies

$$\int_\Omega (\nabla\psi)_t\cdot(\nabla\phi)dx + \frac{1}{Re}\int_\Omega \Delta\psi\Delta\phi dx + \int \Delta\psi J(\phi,\psi)dx - \frac{1}{Ro}\int_\Omega \psi_x\phi dx = \frac{1}{Ro}\int_\Omega F\phi dx. \quad (5.3)$$

Proposition 5.1. *For the steady-state version of the equation, $(\Delta\psi)_t = 0$, the weak solution ψ obeys the stability estimate[7]*

$$|\psi|_2 \le \frac{Re}{Ro}\,||F||_{-2}\,. \quad (5.4)$$

Proof. Consider the weak form with $\phi = \psi$. Then

$$\frac{1}{Re}\int_\Omega \Delta\psi\Delta\psi dx + \int \Delta\psi J(\psi,\psi)dx - \frac{1}{Ro}\int_\Omega \psi_x\psi dx = \frac{1}{Ro}\int_\Omega F\psi dx.$$

As $J(\psi,\psi) = \psi_x\psi_y - \psi_y\psi_x = 0$ the nonlinear term is zero. By Green's theorem, the Coriolis term becomes

$$\begin{aligned}\int_\Omega \psi_x\psi dx &= \frac{1}{2}\int_\Omega \frac{\partial}{\partial x}\psi^2 dx \\ &= \frac{1}{2}\int_{\partial\Omega} \psi^2 dx + 0dy \\ &= 0.\end{aligned}$$

Therefore we are left with

$$\int_\Omega (\Delta\psi)^2 dx = |\psi|_2^2 = \frac{Re}{Ro}\int F\psi dx \leq \frac{Re}{Ro}\,||F||_{-2}\,|\psi|_2$$

by the definition of the $||\cdot||_{-2}$ norm. □

5.3 Finite Element Formulation

5.3.1 Discrete Formulation

By work in [7] we have that a finite element solution exists to the steady-state quasi-geostrophic equations in pure stream function form, given some limit on the initial data. By [6] the semidiscretization also has a finite element solution. For our case we consider the space X_h to be linear combinations of the Argyris basis polynomials defined on a triangulation. Therefore there exists some ψ_h such that for any $\phi_h \in X_h$,

$$\int_\Omega (\nabla\psi_h)_t\cdot\nabla\phi_h dx + \frac{1}{Re}\int_\Omega \Delta\psi_h\Delta\phi_h + \int_\Omega \Delta\psi_h J(\phi_h,\psi_h)dx - \frac{1}{Ro}\int_\Omega (\psi_h)_x\phi_h dx = \frac{1}{Ro}\int_\Omega F\phi_h dx.$$

For the steady-state problem, the same estimate as in Proposition 5.1 holds in the discrete case. In particular, by [6] we have that (if a small data condition is satisfied) with Argyris finite elements, and maximum element side length h there exists some constant C where

$$||\psi - \psi_h||_{L^2(\Omega)} = Ch^6. \tag{5.5}$$

Hence we may achieve *optimal* error estimates (proportional to interpolation error) by use of a C^1 finite element.

To compute ψ_h, assume that

$$\psi_h = \sum_{i=1}^{N} c_i\phi_i. \tag{5.6}$$

We use a *Galerkin* method, where we "test" our final answer against every test function in the finite dimensional space.

Note that in our context the value c_i may reflect function value, first derivative value, second derivative value, or normal derivative value. This allows us to rewrite the time-dependent problem as a system of nonlinear ODEs, or

$$S\frac{d\vec{c}}{dt} + \frac{1}{Re}L\vec{c} + J(\vec{c})\vec{c} - \frac{1}{Ro}B\vec{c} = F(t) \tag{5.7}$$

where S, L, $J(\vec{c})$, and B correspond to the discretization of the stiffness, biharmonic, nonlinear, and β-plane matrices, and $\vec{c}$ is a column vector of basis function weights.

5.3.2 Steady-State QGE Numerical Example

Based on Theorem 4.3 in [7] we should see order 6 convergence in the L^2 norm, order 5 in the H^1 norm and order 4 in the H^2 norm. Consider the test problem (for $Re = 1$ and $Ro = 1$)

$$\psi = (\sin(5\pi x)\sin(3\pi y))^2, \quad \Omega = [0,1]^2. \tag{5.8}$$

mesh step size	DOF	e_{L^2}	L^2 order	H^1 order	H^2 order
0.6667	106	0.23494	N/A	N/A	N/A
0.4444	174	0.16432	0.8817	1.0905	0.6011
0.2963	242	0.092459	1.4183	0.9558	0.6435
0.1975	558	0.013453	4.7540	3.7925	2.6513
0.1317	946	0.0025536	4.0982	3.2944	2.6755
0.0878	1875	0.00046847	4.1824	3.3596	2.7146
0.0585	4191	2.5667e-05	7.1628	5.8656	4.5458
0.0390	8560	2.5636e-06	5.6819	4.7988	3.8196
0.0260	18669	2.2623e-07	5.9872	4.9761	3.9441
0.0173	40805	1.8889e-08	6.1237	5.1084	4.0570
0.0116	91431	1.5388e-09	6.1845	5.1110	4.0573
0.0077	202634	1.3486e-10	6.0042	5.0220	4.0200
0.0051	454824	4.8121e-11	2.5416	5.0395	4.0202

Table 5.1: Demonstration of optimal orders of convergence with L^2 error (e_{L^2}) for steady-state QGE.

The method exhibits the correct convergence orders. However, the L^2 order of convergence falls abruptly at the end. This is explainable by looking at the absolute error; the average L^2 error per degree of freedom is approximately $1.058 \cdot 10^{-16}$, or on the order of machine precision.

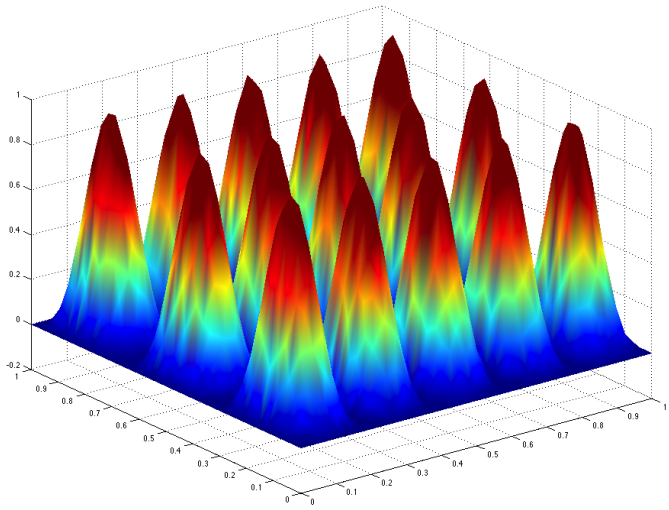

Figure 5.1: Plot of solution of 5.8 with $h = 0.0585$.

5.3.3 Time-dependent QGE Example

While results are known for the semidiscretization [6], rigorous results for convergence in time for the full discretization are not yet proven. However, applying the trapezoid rule yields optimal rates of convergence in time in practice. Consider the test problem (for $Re = 1$ and $Ro = 1$)

$$\psi = (\sin(2\pi x)\sin(2\pi y))^2(1 - \sin(30t))t\exp(3xt), \quad \Omega = [0,1]^2, \quad 0 \le t \le 1/2. \tag{5.9}$$

Experiments were run on the unit square with an unstructured mesh of average length 1/20 and 5029 degrees of freedom.

Time step size	Final e_{L^2}	L^2 order
1/16	0.0045579	N/A
1/32	0.001644	1.3862
1/64	0.00042052	1.9547
1/128	0.00010553	1.9924
1/256	2.641e-05	1.9979
1/512	6.6038e-06	1.9996

Table 5.2: Demonstration of optimal order of convergence with L^2 error (e_{L^2}) for the implicit trapezoid rule.

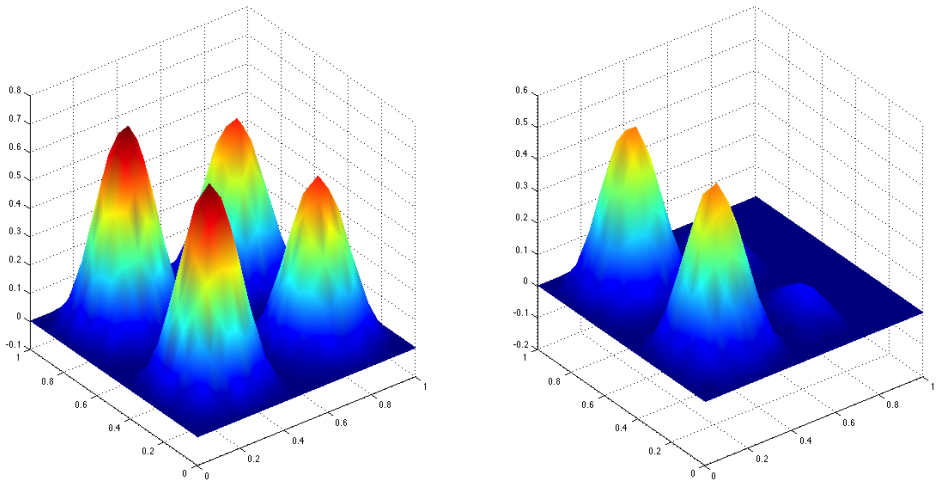

Figure 5.2: Plot of QGE solution for a time step of 1/512 at $t = 1/4$ and $t = 1/2$.

Since we achieve the optimal rate of convergence, the error in the time discretization dominates the error in the space discretization.

Chapter 6

A Two-Level Method for Solving the Quasi-Geostrophic Equations

6.1 Overview

A computational difficulty involved in finding solutions to many finite element problems is solving the resulting nonlinear system. Historically, these algorithms have been used to reduce linear systems to symmetric positive definite ones and nonlinear systems to linear ones [21]. Layton was the first (to our knowledge) to apply a two-level algorithm to the Navier Stokes equations [15]. Our interest in the two-level method is to lower total solution time. In particular, since the quasi-geostrophic equations model the ocean, we are ultimately interested in solving long time integrations more efficiently.

Consider the pure stream function form of the quasi-geostrophic equations. Denote the bilinear and trilinear forms by

$$\begin{aligned} a(\psi,\phi) &= \frac{1}{Re}\int_\Omega \Delta\psi\Delta\phi dx, \\ b(\zeta;\psi,\phi) &= \int_\Omega \Delta\zeta J(\phi,\psi)dx, \\ c(\psi,\phi) &= -\frac{1}{Ro}\int_\Omega \psi_x\phi dx, \\ l(\phi) &= \frac{1}{Ro}\int_\Omega F\phi dx. \end{aligned}$$

Therefore the finite element formulation of the steady-state QGE is to find some $\psi \in X$ such that

$$a(\psi,\phi) + b(\psi;\psi,\phi) + c(\psi,\phi) = l(\phi), \forall\phi \in X. \tag{6.1}$$

Let X^h and X^H be subsets of $H_0^2(\Omega)$ with finite element mesh sizes h and H respectively, where $H > h$. Our goal is to take a solution of the *coarse-mesh* problem, solved over X^H, and use it to linearize the *fine-mesh* problem, solved over X^h. The procedure is as follows:

Algorithm 2 Two-level algorithm.

Step 1: Solve the nonlinear steady-state problem for $\psi^H \in X^H$

$$a(\psi^H, \phi^H) + b(\psi^H; \psi^H, \phi^H) + c(\psi^H, \phi^H) = l(\phi^H), \forall \phi^H \in X^H.$$

Step 2: Solve the following *linear* problem on a fine mesh for $\psi^h \in X^h$

$$a(\psi^h, \phi^h) + b(\psi^H; \psi^h, \phi^h) + c(\psi^h, \phi^h) = l(\phi^h), \forall \phi^h \in X^h.$$

The well-posedness of the nonlinear system is a known result [1]. Since the resulting system is linear, it may be shown by the Lax-Milgram lemma that the solution exists and is unique [8].

Proposition 6.1. *Given a solution ϕ^H of the coarse problem, the solution ϕ^h to the corresponding linear problem exists uniquely.*

Proof. Let $B : X^h \times X^h \to \mathbb{R}$ be the bilinear form given by

$$B(\psi^h, \phi^h) = a(\psi^h, \phi^h) + b(\psi^H; \psi^h, \phi^h) + c(\psi^h, \phi^h).$$

By [8] we have an inequality (for all $\phi^h \in X^h$)

$$B(\psi^h, \phi^h) \leq C|\psi^h|_2|\phi^h|_2.$$

Combining this with the stability estimate 5.1 implies that

$$B(\psi^h, \phi^h) \leq C\,||F||_{-2}\,|\phi^h|_2.$$

Hence B is continuous. As $b(\psi^H; \psi^h, \psi^h) = 0$ (due to the Jacobian term) and $c(\psi^h, \psi^h) = 0$, by Poincaré-Friedrichs we have that (for some other constant C)

$$B(\phi^h, \phi^h) \geq C\left|\left|\phi^h\right|\right|_2$$

for any $\phi^h \in X^h$. Therefore B is coercive. Therefore by the Lax-Milgram lemma ϕ^h exists and is unique. □

Finally, by Corollary 1 in [8] we have that the error satisfies the following error estimate

$$|\phi - \phi^h|_2 \leq C_1 h^4 + C_2\sqrt{|\ln(h)|}H^5. \tag{6.2}$$

This error result implies that we should observe *two* rates of convergence; for a very coarse mesh coupled with a very fine mesh, the error term will be dominated by the $O(H^5)$ convergence. For a fine mesh coupled with a similar coarse mesh (that is, $O(h) = O(H)$) the error term will be dominated by the $O(h^4)$ convergence.

6.2 Computing the Linearized Jacobian

6.2.1 Finding the Element on the Coarse Mesh

We determine if one element is inside another by checking if the *centroid* of the element on the fine mesh is inside some element on the coarse mesh. Given two meshes, we do not know *a priori* which element on the fine mesh corresponds to which element on the coarse mesh. Therefore we need an efficient look-up algorithm.

Algorithm 3 Algorithm for checking if a point is in a triangle.

Step 1: Given a point (x, y) and a triangle described by $(x_1, y_1), (x_2, y_2), (x_3, y_3)$, calculate the affine mapping from physical coordinates to reference coordinates $F : K \to \hat{K}$.

Step 2: Compute $(\hat{x}, \hat{y}) = F^{-1}(x, y)$; that is, the reference coordinates of the given point.

Step 3: If $\hat{x} < 0$ or $\hat{x} > 1$ then the point is not in the reference element.

Step 4: If $\hat{y} < 0$ or $\hat{y} > 1 - \hat{x}$ then the point is not in the reference element. Otherwise the point is in the reference element.

Code is available for Algorithm 3 in the appendix. The remaining problem is how to efficiently search through the coarse mesh to find the element corresponding to a given element on the fine mesh. The "obvious" algorithm is to loop through the elements on the coarse mesh until a match is found using Algorithm 3. This process can be improved with some preprocessing.

Algorithm 4 Algorithm for finding the element on the coarse mesh.

Before examining the fine mesh, sort the coarse mesh elements by the values of their centroids.

Step 1: Let (x_c, y_c) be the centroid of an element on the fine mesh.

Step 2: Perform a binary search across the x centroids of the coarse mesh elements. Stop when the distance between x_c and a coarse mesh centroid is less than H. Since the coarse centroids are sorted, any other elements that satisfy this condition must be adjacent to the coarse element we found. Save these elements as a list.

Step 3: Do the same binary search across the y centroids of the coarse mesh elements with the same stopping condition.

Step 4: Compute a new list of elements common to the two lists. These are the elements that may contain (x_c, y_c). Use algorithm 3 to determine if (x_c, y_c) is in any of the elements. Stop when we find the encompassing element.

Therefore given an element on the fine mesh, we may determine the element on the coarse mesh in logarithmic time with a small constant offset; a large improvement over the linear search.

6.3 Numerical Results for Two-Level SQGE

We use the test problem

$$\psi(x,y) = (\sin(\pi x)\sin(2\pi y))^2, \quad \Omega = [0,1]^2, \tag{6.3}$$

with $Re = Ro = 1$. To verify the $O(H^5)$ rate of convergence we will fix $h = 0.0063$ and vary H. To verify the rate of convergence in h we must be more cautious; if we fix H then the error from it will ultimately dominate and it will not be possible to observe the correct rate of convergence. We consider a scaling factor $C > 1$ such that $H = Ch$. This choice implies that the error should be now $O(h^4)$, as with this choice

$$\begin{aligned} C_2\sqrt{|\ln(h)|}H^5 &= C_2C\sqrt{|\ln(h)|}h^5 \\ &= C_2C\left(\sqrt{|\ln(h)|}h\right)h^4 \\ &\approx O(h^4) \end{aligned}$$

as by L'Hôpital's rule the limit as $h \to 0^+$ of $\sqrt{|\ln(h)|}h$ is zero but the limit of $\sqrt{|\ln(h)|}$ diverges. This scaling is not necessary for practical computations, but we use it here to monitor the rate of convergence with respect to h. For a choice of $C \approx 1/2$ we recover the correct order of convergence.

H	h	DoFs, H	DoFs, h	e_{L^2}	e_{H^2}	H^2 order
1.1	0.43	38	106	1.31×10^{-2}	5.832×10^{0}	–
0.43	0.21	106	350	2.25×10^{-3}	6.61×10^{-1}	2.56
0.21	0.10	350	1270	4.40×10^{-5}	3.65×10^{-2}	3.96
0.10	0.05	1270	4838	1.90×10^{-7}	2.00×10^{-3}	4.10
0.050	0.025	4838	18886	8.95×10^{-10}	1.20×10^{-4}	4.04
0.025	0.013	18886	74630	1.36×10^{-10}	7.40×10^{-6}	4.02
0.013	0.0063	74630	296710	2.22×10^{-9}	4.68×10^{-7}	3.99

Table 6.1: Two-level method: the L^2-norm of the error (e_{L^2}), the H^2-norm of the error (e_{H^2}), and the convergence rate with respect to h.

The $O(H^5)$ convergence is more difficult to obtain. Instead of balancing the errors as we did in the $O(h^4)$ case, we must pick a sufficiently fine mesh and then refine the coarse mesh. By Table 6.2, for a sufficiently refined coarse mesh the order of convergence drops as the error from the fine mesh is on the same order of magnitude as the error from the coarse mesh.

The final quantity that we are interested in measuring is the *performance benefit* of using the two level linearization. By Table 6.3, the two level method yields errors in the L^2 norm on the same order of accuracy as a purely nonlinear solve, while reducing solution time by about half.

H	h	DoFs, H	DoFs, h	e_{L^2}	e_{H^2}	H^2 order
0.43	0.0063	350	296710	2.26×10^{-3}	4.54×10^{-1}	–
0.21	0.0063	1270	296710	4.39×10^{-5}	1.755^{-2}	4.45
0.10	0.0063	4838	296710	1.86×10^{-7}	4.92×10^{-4}	5.04
0.05	0.0063	18886	296710	2.11×10^{-9}	1.32×10^{-5}	5.2
0.025	0.0063	74630	296710	$2..18 \times 10^{-9}$	6.02×10^{-7}	4.45

Table 6.2: Two-level method: the L^2-norm of the error (e_{L^2}), the H^2-norm of the error (e_{H^2}), and the convergence rate with respect to H.

H	h	DoFs, H	DoFs, h	e_{L^2}	e_{H^2}	time, s
–	0.05146	–	4362	4.286×10^{-8}	1.648×10^{-3}	3.328
0.1083	0.05146	1158	4362	1.092×10^{-7}	1.709×10^{-3}	2.372
–	0.02561	–	16926	5.748×10^{-10}	1.009×10^{-4}	19.92
0.05146	0.02561	4362	16926	7.691×10^{-10}	1.016×10^{-4}	11.82
–	0.01597	–	43074	4.751×10^{-11}	1.793×10^{-5}	55.69
0.03384	0.01597	10983	43074	5.267×10^{-11}	1.797×10^{-5}	33.19
–	0.01277	–	66678	8.66×10^{-12}	6.207×10^{-6}	102.4
0.02561	0.01277	16926	66678	9.686×10^{-12}	6.217×10^{-6}	59.03
–	0.009659	–	116614	3.876×10^{-12}	2.382×10^{-6}	161.7
0.02035	0.009659	29501	116614	6.836×10^{-12}	2.385×10^{-6}	95.93
–	0.007959	–	170598	4.791×10^{-12}	1.111×10^{-6}	325.1
0.01597	0.007959	43074	170598	9.087×10^{-12}	1.112×10^{-6}	172.3
–	0.006854	–	230574	1.79×10^{-11}	6.16×10^{-7}	401.7
0.01436	0.006854	58131	230574	1.3×10^{-11}	6.163×10^{-7}	219.5
–	0.006374	–	264678	3.912×10^{-11}	3.846×10^{-7}	559.7
0.01277	0.006374	66678	264678	2.309×10^{-11}	3.848×10^{-7}	291.9
–	0.005264	–	389994	3.85×10^{-11}	2.086×10^{-7}	753.4
0.01101	0.005264	98133	389994	6.495×10^{-11}	2.087×10^{-7}	397.7

Table 6.3: Comparison of one-level and two-level methods: the L^2-norm of the error (e_{L^2}), the H^2-norm of the error (e_{H^2}) and simulation times.

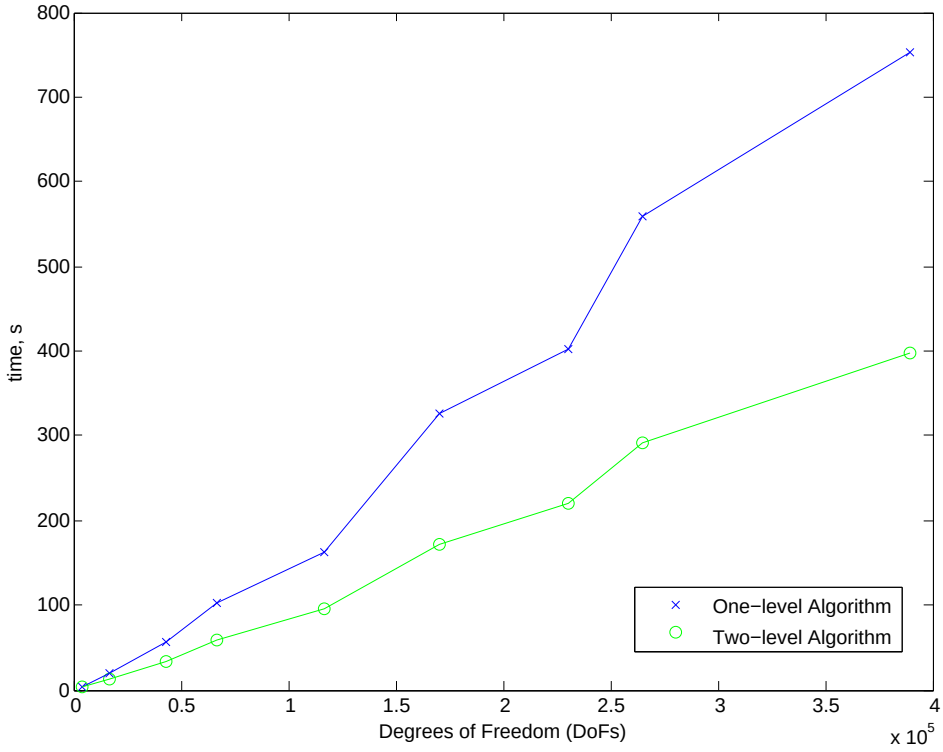

Figure 6.1: The simulation times of the one-level method (green) and of the two-level method (blue) are displayed for all the pairs (h, H) in Table 6.3.

Chapter 7

Conclusions & Future Work

In this work we describe new algorithms for computing solutions to the quasi-geostrophic equations. Our focus is on conforming solutions to the pure stream function form of the quasi-geostrophic equations; as this equation contains a fourth-order differential operator we use C^1 finite elements to achieve optimal error estimates. In particular, we describe the implementation of the Argyris finite element proposed by Dominguez et al [3].

The pure stream function formulation of QGE coupled with the Argyris finite element yields optimal error estimates for small data. We demonstrate this convergence with numerical experiments. In particular, we use a two-level method as a numerical technique to speed up solution time.

There are ample opportunities to expand this work in new directions. The implementation of the two-level algorithm should be expanded to work on time dependent problems. This brings up several new (and interesting) issues about how to march two grids forward in time. Another important practical issue is *stabilization*. For low Rossby numbers, the Coriolis force term dominates the flow and creates strong convective currents. To make things more complex, QGE has two convective terms (the nonlinearity as well as the Coriolis term). Perhaps both should be stabilized.

The theory for the finite element method applied to the stream function form of QGE is still incomplete; in particular, to the best of our knowledge there are no numerical results for the "full discretization" (that is, time marching schemes). While numerical experiments suggest that we can reach optimal orders of convergence for implicit methods, future work must include rigorously proving stability and convergence results.

Finally, the current formulation cannot handle multiply connected domains due to problems with uniqueness of stream function. This hinders our ability to accurately model the oceans, as it means there is no easy way to handle islands. There is some promising work [20] on how to guarantee stream function uniqueness. We plan to pursue this in a future study.

Bibliography

[1] M. E. Cayco and R. A. Nicolaides. Finite element technique for optimal pressure recovery from stream function formulation of viscous flows. *Math. Comp.*, 46(174), 1986.

[2] V. Dominguez and F. Sayas. A simple matlab implementation of Argyris element. Technical Report 25, Universidad de Zaragoza, 2006.

[3] V. Dominguez and F. Sayas. Algorithm 884: A simple matlab implementation of the Argyris element. *ACM Transaction on Mathematical Software*, 35(2), 2008.

[4] L. Evans. *Partial Differential Equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, 2010.

[5] E. Foster and D. Wells. *ArgyrisPack 2.0*. Virginia Polytechnic Institute and State University, 2013. `https://github.com/VT-ICAM/ArgyrisPack`.

[6] E. L Foster. *Finite Elements for the Quasi-Geostrophic Equations of the Ocean*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, 2013.

[7] E. L. Foster, T. Iliescu, and Z. Wang. A finite element discretization of the streamfunction formulation of the stationary quasi-geostrophic equations of the ocean. Comput. Meth. App. Mech. Eng., to appear, 2012.

[8] E. L. Foster, T. Iliescu, and David Wells. A two-level finite element discretization of the streamfunction formulation of the stationary quasi-geostrophic equations of the ocean. Comput. Math. App., under revision, 2013.

[9] C. Geuzaine and Jean-François Remacle. *Gmsh Reference Manual*. http://www.geuz.org/gmsh, 2.7.0 edition, 2013.

[10] V. Girault and P. A. Raviart. *Finite Element Approximation of the Navier-Stokes Equations: Theory and Algorithms*. Volume 749 of Lecture Notes in Mathematics. Springer-Verlag, 1979.

[11] M. D. Gunzburger. *Finite Element Methods for Viscous Incompressible Flows*. Computer Science and Scientific Computing. Academic Press Inc, 1989. A Guide to Theory, Practice, and Algorithms.

[12] M. D. Gunzburger and J.S. Peterson. Finite-element methods for the streamfunction-vorticity equations: Boundary-condition treatments and multiply connected domains. *SIAM J. Sci. Stat. Comp.*, 9:650, 1988.

[13] M. D. Gunzburger and J.S. Peterson. On finite-element approximations of the streamfunction-vorticity and velocity-vorticity equations. *Internat. J. Numer. Meth. Fluids*, 8(10):1229–1240, 1988.

[14] K. Hoffman and R. Kunze. *Linear Algebra*. Prentice-Hall, Inc, 1971.

[15] W. Layton. A two-level discretization method for the Navier-Stokes equations. *Comp. Math. Applic.*, 26(2):33–38, 1993.

[16] W. Layton. *Introduction to the Numerical Analysis of Incompressible Viscous Flows*. Society for Industrial and Applied Mathematics, 2008.

[17] A. J. Majda and X. Wang. *Non-linear Dynamics and Statistical Theories for Basic Geophysical Flows*. Cambridge University Press, 2006.

[18] J. Pedlosky. *Geophysical Fluid Dynamics*. Springer, second edition, 1992.

[19] W. A. Stein et al. *Sage Mathematics Software (Version 5.8.0)*. The Sage Development Team, 2013. `http://www.sagemath.org`.

[20] M.B van Gijzen, C. B. Vreugdenhil, and H. Oksuzoglu. The finite element discretization for stream-function problems on multiply connected domains. *JCP*, 140:30–46, 1998.

[21] J. Xu. A novel two-grid method for semilinear elliptic equations. *SIAM J. on Sci. Comp.*, 15(1):231–237, 1994.

Appendix A

Example Code

A.1 Triangle Lookup Algorithm

```
1  bool in_triangle(double* point, double* x, double* y)
2  {
3          /*
4           * check if we are 'close' to the triangle, rather than exactly
5           * inside.
6           */
7          double tol = 10e-10;
8
9          double B00 = -x[0] + x[1];
10         double B01 = -x[0] + x[2];
11         double B10 = -y[0] + y[1];
12         double B11 = -y[0] + y[2];
13
14         double inv_det = 1/(B00*B11 - B01*B10);
15         double point0 = point[0] - x[0];
16         double point1 = point[1] - y[0];
17
18         double mapped_x = inv_det*(B11*point0 - B01*point1);
19         double mapped_y = inv_det*((-1)*B10*point0 + B00*point1);
20
21         if ((mapped_x < (0 - tol)) || (mapped_x > (1 + tol))) {
22                 return false;
23         }
24         else if ((mapped_y < (0 - tol)) ||
25                  (mapped_y > (1 - mapped_x + tol))) {
26                 return false;
27         }
28         else {
29                 return true;
30         }
31 }
```

A.2 Class Method for Lookup Data Structure Initialization

```
function initializePointLookup(self)
  % Assume that the domain is roughly rectanglar to calculate approximate
  % splits for the x ranges.
  xMin = min(self.nodes(:,1));
  xMax = max(self.nodes(:,1));
  yMin = min(self.nodes(:,2));
  yMax = max(self.nodes(:,2));

  % fudge the max values slightly for safety.
  xMin = xMin - abs(xMin)*0.01;
  xMax = xMax + abs(xMax)*0.01;
  yMin = yMin - abs(yMin)*0.01;
  yMax = yMax + abs(yMax)*0.01;

  stepSizes = sqrt( ...
      (self.nodes(self.elements(:,1), 1) - self.nodes(self.elements(:,2), 1)).^2 + ...
      (self.nodes(self.elements(:,1), 2) - self.nodes(self.elements(:,2), 2)).^2);

  % also fudge the step size for safety.
  stepSize = mean(stepSizes) + 0.1*std(stepSizes);

  self.elementXRanges = nan(ceil(abs(xMax - xMin)/(stepSize)), 2);
  self.elementYRanges = nan(ceil(abs(yMax - yMin)/(stepSize)), 2);
  self.elementsByXLocation = cell(length(self.elementXRanges));
  self.elementsByYLocation = cell(length(self.elementYRanges));

  for i=1:length(self.elementXRanges)
    self.elementXRanges(i,:) = [xMin + (i-1)*stepSize - stepSize/10, ...
        xMin + i*stepSize + stepSize/10];
  end

  for i=1:length(self.elementYRanges)
    self.elementYRanges(i,:) = [yMin + (i-1)*stepSize - stepSize/10, ...
        yMin + i*stepSize + stepSize/10];
  end

  for i=1:length(self.elements)
    xMin = min(self.nodes(self.elements(i,1:3), 1));
    xMax = max(self.nodes(self.elements(i,1:3), 1));
    yMin = min(self.nodes(self.elements(i,1:3), 2));
    yMax = max(self.nodes(self.elements(i,1:3), 2));

    for j=1:length(self.elementXRanges)
      % if we are past the element then stop.
      currentRange = self.elementXRanges(j,:);
      if xMax < currentRange(1)
        break
      end

      % Check if xMin falls in the correct range, a < xMin < b.
      if currentRange(1) < xMin && currentRange(2) > xMin
        self.elementsByXLocation{j} = [self.elementsByXLocation{j} i];
        continue
      end

      % Check if xMax falls in the correct range, a < xMax < b.
      if currentRange(1) < xMax && currentRange(2) > xMax
        self.elementsByXLocation{j} = [self.elementsByXLocation{j} i];
        continue
      end

      % Check if xMin < a and xMax > b (so element falls partly inside)
      if xMin < currentRange(1) && xMax > currentRange(2)
```

```
64          self.elementsByXLocation{j} = [self.elementsByXLocation{j} i];
65        end
66      end
67
68      for j=1:length(self.elementYRanges)
69        currentRange = self.elementYRanges(j,:);
70        if yMax < self.elementYRanges(j,1)
71          break
72        end
73
74        if currentRange(1) < yMin && currentRange(2) > yMin
75          self.elementsByYLocation{j} = [self.elementsByYLocation{j} i];
76          continue
77        end
78
79        if currentRange(1) < yMax && currentRange(2) > yMax
80          self.elementsByYLocation{j} = [self.elementsByYLocation{j} i];
81          continue
82        end
83
84        if yMin < currentRange(1) && yMax > currentRange(2)
85          self.elementsByYLocation{j} = [self.elementsByYLocation{j} i];
86        end
87      end
88    end
89  end
```

A.3 Class Method for Element Lookup

```
function elementNumber = getElementNumber(self, point)
% self.getElementNumber(POINT): Return the number of the element containing
% point POINT. For example,
%
% >>  disp(mesh.getElementNumber([0.1, 0.1]))
%
%        5
% >>
  if ~self.hasPointLookup
    self.initializePointLookup();
    self.hasPointLookup = true;
  end
  foundElement = false;

  leftXRangeGuess = 1;
  rightXRangeGuess = length(self.elementXRanges);
  xRangeGuess = floor((leftXRangeGuess + rightXRangeGuess)/2);
  while (0 < xRangeGuess) && ...
        (xRangeGuess < length(self.elementXRanges) + 1)
    xRangeGuess = floor((leftXRangeGuess + rightXRangeGuess)/2);
    testRange = self.elementXRanges(xRangeGuess,:);

    if point(1) < testRange(1)
      rightXRangeGuess = xRangeGuess - 1;
    elseif point(1) > testRange(2)
      leftXRangeGuess = xRangeGuess + 1;
    else
      break;
    end
  end

  leftYRangeGuess = 1;
  rightYRangeGuess = length(self.elementYRanges);
  yRangeGuess = floor((leftYRangeGuess + rightYRangeGuess)/2);
  while (0 < yRangeGuess) && ...
        (yRangeGuess < length(self.elementYRanges) + 1)
    yRangeGuess = floor((leftYRangeGuess + rightYRangeGuess)/2);
    testRange = self.elementYRanges(yRangeGuess,:);

    if point(2) < testRange(1)
      rightYRangeGuess = yRangeGuess - 1;
    elseif point(2) > testRange(2)
      leftYRangeGuess = yRangeGuess + 1;
    else
      break;
    end
  end

  possibleElements = vectorIntersectUniqueMex( ...
      self.elementsByXLocation{xRangeGuess}, ...
      self.elementsByYLocation{yRangeGuess});
  for i=possibleElements
    if inTriangleMex(point, self.nodes(self.elements(i,1:3),1), ...
                             self.nodes(self.elements(i,1:3),2));
      foundElement = true;
      elementNumber = i;
    end
  end

  if ~foundElement
    error('Failed to find an element correlated to the given point')
  end
end
```