NAVAL POSTGRADUATE SCHOOL

THESIS

A STUDY INTO DISCONTINUOUS GALERKIN METHODS FOR THE SECOND ORDER WAVE EQUATION

by

Benjamin J. Davis

June 2015

Thesis Co-Advisors:	Jeremy E. Kozdon
	Lucas C. Wilcox

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 06-19-2015	3. REPORT TYPE AND DATES COVERED Master's Thesis 06-30-2013 to 06-19-2015

4. TITLE AND SUBTITLE

A STUDY INTO DISCONTINUOUS GALERKIN METHODS FOR THE SECOND ORDER WAVE EQUATION

5. FUNDING NUMBERS

6. AUTHOR(S)

Benjamin J. Davis

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

There are numerous numerical methods for solving different types of partial differential equations (PDEs) that describe the physical dynamics of the world. For instance, PDEs are used to understand fluid flow for aerodynamics, wave dynamics for seismic exploration, and orbital mechanics. The goal of these numerical methods is to approximate the solution to a continuous PDE with an accurate discrete representation. The focus of this thesis is to explore a new Discontinuous Galerkin (DG) method for approximating the second order wave equation in complex geometries with curved elements. We begin by briefly highlighting some of the numerical methods used to solve PDEs and discuss the necessary concepts to understand DG methods. These concepts are used to develop a one- and two-dimensional DG method with an upwind flux, boundary conditions, and curved elements. We demonstrate convergence numerically and prove discrete stability of the method through an energy analysis.

14. SUBJECT TERMS
Discontinuous Galerkin, Acoustic Wave Equation, Finite Element Method, Numerical Methods for PDEs

15. NUMBER OF PAGES   151

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

A STUDY INTO DISCONTINUOUS GALERKIN METHODS FOR THE SECOND ORDER WAVE EQUATION

Benjamin J. Davis
Captain, United States Army
B.S., United States Military Academy, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June 2015

Author: Benjamin J. Davis

Approved by: Jeremy E. Kozdon
                  Thesis Co-Advisor

                  Lucas C. Wilcox
                  Thesis Co-Advisor

                  Craig W. Rasmussen
                  Chair, Department of Applied Mathematics

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

There are numerous numerical methods for solving different types of partial differential equations (PDEs) that describe the physical dynamics of the world. For instance, PDEs are used to understand fluid flow for aerodynamics, wave dynamics for seismic exploration, and orbital mechanics. The goal of these numerical methods is to approximate the solution to a continuous PDE with an accurate discrete representation. The focus of this thesis is to explore a new Discontinuous Galerkin (DG) method for approximating the second order wave equation in complex geometries with curved elements. We begin by briefly highlighting some of the numerical methods used to solve PDEs and discuss the necessary concepts to understand DG methods. These concepts are used to develop a one- and two-dimensional DG method with an upwind flux, boundary conditions, and curved elements. We demonstrate convergence numerically and prove discrete stability of the method through an energy analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

CG   Continuous Galerkin

DG   Discontinuous Galerkin

FDM   Finite Difference Method

FEM   Finite Element Method

FVM   Finite Volume Method

LGL   Legendre–Gauss–Lobatto

NPS   Naval Postgraduate School

ODE   Ordinary Differential Equation

PDE   Partial Differential Equation

RK54   Runge–Kutta Five Stage Fourth Order

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I humbly express my gratitude to the members of the Department of Mathematics at the Naval Postgraduate School; you are an amazing group of individuals who have significantly helped me through the rigors of mathematics for the past two years. Professor F.X. Giraldo, I thank you for opening the world of Galerkin methods to me. Your course is one of the most challenging that I faced during my time with this department and the skills I learned greatly assisted me in completing this thesis. To my advisors: Assistant Professor Lucas Wilcox, I appreciate your patience, support, and advice throughout my thesis work; Assistant Professor Jeremy Kozdon, I can not thank you enough for your instruction, mentorship, and unwavering patience with me. I will always remember the numerous discussions and enormous amount of mathematical wisdom that you bestowed upon me. You are truly a master of your craft, a phenomenal instructor, and most importantly, a great person. Thank you.

To my wonderful wife: Allison, thank you for your love, understanding, and encouragement these past two years. Thank you for putting up with my numerous late nights and long hours. You are truly amazing, Little Red, and I love you with all my heart.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
An Introduction to Solving Partial Differential Equations

There are numerous numerical methods for solving different types of partial differential equations (PDEs) that describe the physical dynamics of the world. The goal of these numerical methods is to approximate the solution to the continuous PDE with a discrete representation [1]. Three notable methods are Finite Difference Methods (FDMs), Finite Volume Methods (FVMs), and Finite Element Methods (FEMs). Two common FEMs are Continuous Galerkin (CG) and Discontinuous Galerkin (DG), each of which comprises an element-based approach to solving a set of equations. The main focus of this thesis is to explore a new DG method, introduced by Appelö and Hagstrom [2], for approximating the second-order wave equation. However, my research differs from that of Appelö and Hagstrom by using a nodal form of DG with provable stability on curved elements with complex geometries.

1.1 Finite Difference Method

Finite difference methods are one of the simplest and oldest methods for solving partial differential equations [1]. Furthermore, it is arguably one of the most used methods for discretizing partial differential equations [3]. There is an enormous amount of published information about FDMs in various scholarly journals and books. This section describes only the basics of FDMs, and the interested reader is directed to, for instance, Gustafsson, Oliger, and Kreiss [4]. Finite difference methods focus on approximating the derivatives of the solution directly at a set of points in a domain. In terms of the calculus of finite differences, we are looking to approximate the derivatives by linear combinations of the function values along a grid of points [5].

One method of constructing the discretization is accomplished by a Taylor series expansion on a selected set of equally spaced points (i.e.,$\ldots < x_{i-1} < x_i < x_{i+1} < \ldots$).

For instance, suppose you wish to approximate the derivative of a function evaluated at x_i using grid points x_{i-1} and x_{i+1}. By conducting a Taylor series expansion of the neighboring values around x_i, we can construct a linear combination to get an accurate approximation of the derivative.

For example, let us consider the one-dimensional advection equation,

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \tag{1.1}$$

where $u = u(x,t)$ is the solution. We assume an appropriate set of initial conditions and boundary conditions for $u(x,t)$. Using the center stencil, we can Taylor expand in space around x_i, with grid spacing $\triangle x$, to find a derivative approximation for $\frac{\partial u}{\partial x}$, where $u_i = u(x_i,t)$ [3]. Starting with

$$u_{i+1} = u_i + \triangle x \frac{\partial u_i}{\partial x} + \frac{\triangle x^2}{2} \frac{\partial^2 u_i}{\partial x^2} + O(\triangle x^3), \tag{1.2}$$

$$u_{i-1} = u_i - \triangle x \frac{\partial u_i}{\partial x} + \frac{\triangle x^2}{2} \frac{\partial^2 u_i}{\partial x^2} + O(\triangle x^3), \tag{1.3}$$

(u_{i+1} and u_{i-1} are not boundary points) and taking the linear combination of (1.2) and (1.3) yields

$$\frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_{i-1}}{2\triangle x} + O(\triangle x^2), \tag{1.4}$$

which can be substituted into (1.1) to yield a system of ordinary differential equations. Lastly, we can solve these ODEs computationally through a numerical integration scheme, such as Runge–Kutta, where

$$\frac{\partial u_i}{\partial t} = -\frac{\partial u_i}{\partial x} \approx \frac{u_{i+1} - u_{i-1}}{2\triangle x}. \tag{1.5}$$

Another method for constructing difference formulas is to build a polynomial interpolant through the given grid of points, such as using Lagrange polynomials, and evaluating the derivative of this polynomial. For instance, using the grid of

points $[x_{i-1}, x_i, x_{i+1}]$, we have the interpolant

$$P_2(x) = \sum_{k=-1}^{1} u_{i+k} L_k(x), \tag{1.6}$$

where $L_k(x)$ is the Lagrange polynomial (see Section 2.1). The derivative approximation is then

$$\frac{\partial u_i}{\partial x} = \sum_{k=-1}^{1} L'_k(x_i) u_{i+k} \approx \frac{u_{i+1} - u_{i-1}}{2\triangle x}, \tag{1.7}$$

which is substituted back into (1.1). The Taylor series approach and the polynomial approach are equivalent when the maximum number of error terms are eliminated from the linear combination.

1.2   Finite Volume Method

Finite volume methods differ from FDMs in that instead of seeking pointwise solution values, we are seeking the average values over the elements using approximations. Depending on the dimension of the problem, examples of elements are cubes, triangles, or intervals, all organized in an unstructured fashion across the physical domain [1]. From the average values, the solution is reconstructed in order to evaluate a numerical flux to tie neighboring elements together and produce an approximation for the entire system. There are numerous FVMs and here we only discuss the very basics; interested readers are directed to LeVeque [6] for more information.

Let us again consider (1.1) on a domain that is represented by a collection of elements. For this example, let us use elements on a one-dimensional domain and each element is denoted by the index i. Let us further define a different set of notation with $\frac{\partial u}{\partial t} = u_t$ and $\frac{\partial u}{\partial x} = u_x$. Therefore, (1.1) becomes $u_t = -u_x$. Generally, the average value of u on an element is

$$Q_i(t) = \frac{1}{\triangle x} \int_{L_i}^{R_i} u(x,t) \, dx \tag{1.8}$$

where $\triangle x$, in this case, is the distance between the left and right boundaries of the one-dimensional interval and $Q_i(t)$ is the spatial average value of u on the ith element. If we were using cube-shaped elements, then $\triangle x$ would be the area of the cubic element. If we take the derivative of (1.8) with respect to time, we find

$$\frac{dQ_i}{dt} = \frac{1}{\triangle x} \int_{Li}^{R_i} u_t \, dx \tag{1.9}$$

and since $u_t = -u_x$, we can substitute $-u_x$ to obtain

$$\frac{dQ_i}{dt} = -\frac{1}{\triangle x} \int_{L_i}^{R_i} u_x \, dx \tag{1.10}$$

and integrate (1.10) to find an equation to build an approximation to the system:

$$\frac{dQ_i}{dt} = -\frac{1}{\triangle x} [u(R_i, t) - u(L_i, t)]. \tag{1.11}$$

Equation (1.11) is an update equation for the average value of the solution in the ith element, where the time derivative of the average value changes through the fluxes of the left and right boundaries of the element. There are a variety of numerical flux methods to accomplish this approximation. One method is the first order upwinding method, meaning that we select the average value of the interval always on the "upwind" side. This topic is discussed in further detail in Chapter 3. Ultimately, the FVM uses the average element values from across the domain to reconstruct an approximation for the system.

1.3   Finite Element Methods

Just like FDMs and FVMs, there are many different FEMs used to solve PDEs. Similar to the FVM, the physical domain is mapped into a reference domain of various sized geometrically flexible elements known as the grid. After the grid is built, one of the many different FEMs is applied to solve the system. As discussed at the beginning of this chapter, the FEM that is the focus of this paper is known as a Galerkin method. There are two main categories of Galerkin methods, Bubnov–Galerkin and Petrov–Galerkin [7]. With Petrov–Galerkin, the test functions and

basis functions are different, where with Bubnov–Galerkin, the test functions and basis functions are the same and reside in the same space [1, 7]. We discuss basis functions and test functions further in later chapters. For this paper, we be focusing on Bubnov-Galerkin methods, often called simply Galerkin methods [7]. As discussed before, the two main categories of Galerkin methods are Continuous Galerkin (CG) and Discontinuous Galerkin (DG). Both methods use an elemental approach to solving the system in integral form, but the difference between DG and CG is mainly whether the continuity between the elements is enforced strongly or weakly. In the following two paragraphs, let us lightly touch on both methods to give the reader a small insight into each method before focusing the rest of the paper entirely on element-based DG.

When using CG, we build a computational domain subdivided into various-sized elements, similar to FVM. However, within each element we use specially selected degrees of freedom. For example, in the one-dimensional sense, the degrees of freedom could be a grid of points across the cell. There are many different ways to build this grid of points, such as using Legendre–Gauss points, equally spaced points, as discussed in FDM, or some other combination or method. For the purpose of this research, we use Legendre–Gauss–Lobatto (LGL) points. These are discussed in further detail in later chapters, but they are a set of points that cluster toward the boundaries of each element and include points at the boundaries. Discretizing the integral form of this equation yields a semi-discrete scheme with elemental test functions [1]. These test functions are continuous across the domain in CG since we enforce the element boundary points to be equal between neighboring elements [3]. Each element has its own set of LGL points; however, using CG enforces the solution at the element boundaries to be continuous and this yields a global mass matrix. This global mass matrix is for the entire domain and must be inverted to solve time-dependent problems; this can be computationally expensive, depending on the type of problem [1].

Generally, DG is similar to CG in requiring numerical interpolation and integration as well as building the same computational domain; however, one of the main differences comes from the elemental boundaries. In CG, we require the boundaries

5

to be continuous, while in DG they are discontinuous and the continuity is enforced weakly. The domain in DG is still represented by a collection of elements; the union of these elements is accomplished through a numerical flux similar to FVMs. Discontinuous Galerkin still uses the same space of basis and test functions, similar to FEMs, and each element boundary has its own set of degrees of freedom [1]. Therefore, the solution is typically represented by a set of piecewise polynomials that are discontinuous at the element boundaries. The numerical flux, which is discussed in greater detail later, resolves this discontinuity to assist in finding the final solution. Furthermore, the mass matrix is constructed locally instead of globally and this allows it to be inverted at a reduced computational cost, yielding a semidiscrete scheme that is explicit [1]. Discontinuous Galkerin is the method used for the following study.

CHAPTER 2:
Discontinuous Galerkin Foundation

In this chapter, the information and terminology maybe confusing to the reader who has no knowledge of Galerkin methods. However, just like pulling out a road map in a foreign country, the intent is to show a DG road map and apply these concepts to a one-dimensional and a two-dimensional problem in subsequent chapters. Initially, this road map may speak a different language to the reader, as expected for anyone new to this numerical method; however, by the end of this chapter the intent is for the DG foundation to be clear and understandable.

2.1   Interpolation

In order to fully implement DG, we first need to construct the building blocks. This leads to polynomial interpolation [3]. Nodal interpolation is the construction of an Nth degree polynomial, $f_N(x)$, that is equal to a given function, $f(x)$, when evaluated at a set of x_i points, with $i = 0, \ldots, N$, that is $f_N(x_i) = f(x_i)$ [7]. There are many different interpolation methods and our focus for this section is Lagrange interpolation.

Let us explain Lagrange interpolation through an example. We are going to approximate the following Gaussian function in one dimension,

$$f(x) = e^{-4x^2}, \quad x \in [-1, +1]. \tag{2.1}$$

We begin by generating a grid of LGL points across the given domain in one dimension. As discussed in Section 1.3, LGL points are a specially selected set of points that includ the boundaries and cluster toward the ends of the domain. This clustering of points toward the ends of the domain assists in avoiding the Runge's phenomenon during the approximation, which is the oscillation of an interpolation near the boundaries. This phenomenon is evident when equally spaced grid points are used with high polynomial orders. Therefore, clustering points toward the ends of the domain helps improve the interpolation [8]. After the grid of LGL points is

generated, we construct the Lagrange basis functions, $L_i(x)$, by the equation

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{(x - x_j)}{(x_i - x_j)}, \quad i = 0, \ldots, N, \tag{2.2}$$

where x_i are the LGL points defined on the domain $[-1, +1]$. Using (2.2) for all the LGL points in the domain generates a set of Nth order polynomial basis functions associated with each point x_i in the domain. With these basis functions, the approximation $f_N(x)$ is

$$f_N(x) = \sum_{i=0}^{N} L_i(x) f_i, \tag{2.3}$$

where $f_i = f(x_i)$ for $i = 0, \ldots, N$. Moreover, since (2.2) has the cardinal property

$$L_i(x_j) = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j \end{cases}, \tag{2.4}$$

we have the interpolation property $f_i = f_N(x_i)$. In the end, $f_N(x)$ is an Nth order polynomial approximation for $f(x)$.

We now return to finding an approximation for representing (2.1). Figure 2.1 shows the polynomial interpolation of (2.1) using $N = 2$, 4, 8, and 16 with $N + 1$ LGL interpolation points. In Figure 2.2, we consider the error norms

$$\| \epsilon \|_{l1} = \frac{\sum_{k=1}^{50} |f_N(\hat{x}_k) - f(\hat{x}_k)|}{\sum_{k=1}^{50} |f(\hat{x}_k)|}, \tag{2.5}$$

$$\| \epsilon \|_{l2} = \sqrt{\frac{\sum_{k=1}^{50} (f_N(\hat{x}_k) - f(\hat{x}_k))^2}{\sum_{k=1}^{50} (f(\hat{x}_k))^2}}, \tag{2.6}$$

$$\| \epsilon \|_{\infty} = \frac{\max_{1 \leq k \leq 50} |f_N(\hat{x}_k) - f(\hat{x}_k)|}{\max_{1 \leq k \leq 50} |f(\hat{x}_k)|}, \tag{2.7}$$

where we have used an equally spaced grid of 50 points, $\hat{x}_k$, within the domain $[-1, +1]$. Figure 2.2 displays the error norm of $N = 1$ through $N = 40$ polynomial

interpolations.

As you can see in Figure 2.1, the 16th- order polynomial interpolation does a good job approximating the function as compared to the exact solution (denoted by $*$ to differentiate from the various interpolations). In Figure 2.2, we show the error in the interpolation vs. the Nth order polynomial interpolant based off of (2.5), (2.6), and (2.7). As the figures show, the approximation gets to machine precision around the 32nd order polynomial. Lastly, the reason that Figure 2.2 has various plateaus within the convergence rate is because $f(x)$ is an even function.
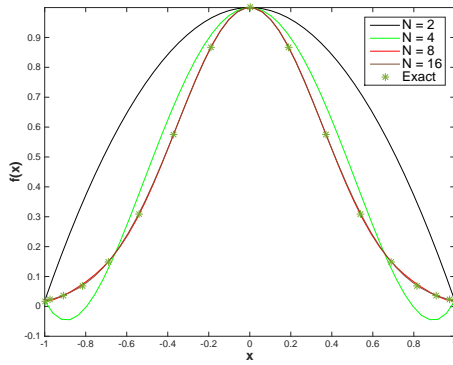
Figure 2.1: Lagrange Interpolation of e^{-4x^2} at different sets of LGL points for various polynomial orders of N.
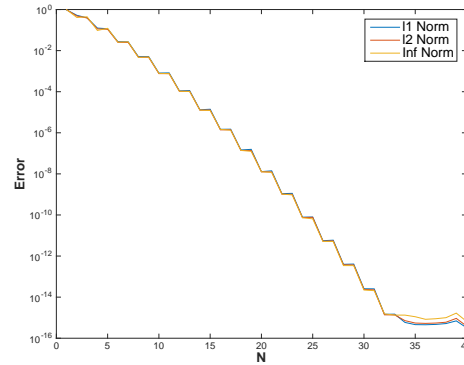
Figure 2.2: Approximation Error based off (2.5), (2.6), and (2.7) measured at polynomial orders $N = 1$ through $N = 40$.

2.2   Integration

The next building block for DG is the ability to conduct numerical integration. When performing numerical integration, also known as quadrature, the analysis is similar to that of interpolation. Integrating the Lagrange interpolation (2.3) gives the integral approximation

$$\int_{-1}^{+1} f_N(x)dx = \int_{-1}^{+1} \sum_{i=0}^{N} L_i(x)f_i dx = \sum_{i=0}^{N} f(x_i) \int_{-1}^{+1} L_i(x)dx. \tag{2.8}$$

9

Defining the quadrature weights

$$w_i = \int_{-1}^{+1} L_i(x)dx \qquad (2.9)$$

gives the numerical integration method [8]

$$\int_{-1}^{+1} f_N(x)dx = \sum_{i=0}^{N} w_i f_i. \qquad (2.10)$$

These quadrature weights only have to be precomputed once with the LGL points and not at each evaluation of a particular integral [8]. Figure 2.3 is the comparison
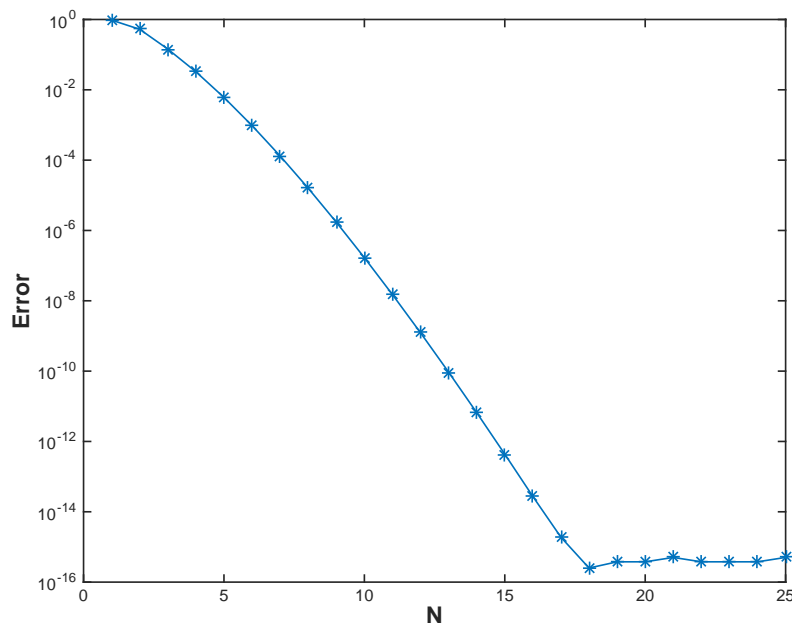
Figure 2.3: Numerical Integration Error evaluated with the Euclidean Norm vs. the Nth order quadrature.

of the numerical integration approximation to the actual value of $\int_{-1}^{+1} e^{(-4x^2)}dx \approx 0.882081$. The Euclidean Norm is used for the calculation of the error due to the integration of $f(x)$ producing a scalar. As you can see, the error decreases to machine precision around the 18th order quadrature.

10

In comparing Figure 2.2 to Figure 2.3, we see that when using the same order polynomial, integration is more accurate than interpolation. In general, using $N+1$ points, we can construct an Nth order polynomial. Since we fixed the LGL points at the boundaries, -1 and +1, there are only $N-1$ LGL points left to choose. We also have $N+1$ quadrature weights to choose. Therefore, we have $N-1$ points and $N+1$ weights, which means there are $2N$ degrees of freedom. We can thus set the degrees of freedom so that we integrate a polynomial of order $2N-1$ exactly (as this polynomial has $2N$ coefficients) [3]. For example, we can exactly integrate a fifth degree polynomial using $N+1=4$ LGL points; even though we can only construct a third-order interpolating polynomial using these points.

2.3   Concept of the Mass Matrix

After discussing interpolation and integration, let us now consider the concept of the mass matrix. The mass matrix, denoted by M, is used for integration. We discuss the integration of the product of two interpolants exactly and inexactly. Using exact integration results in a full mass matrix, while using inexact integration results in a diagonal mass matrix. For example, suppose we wanted to integrate $f(x)$ with a test function, $\psi(x)$,

$$\int_{-1}^{+1} f(x)\psi(x)dx, \tag{2.11}$$

over the domain of a single element from [-1,+1]. Test functions are defined in further detail in Section 3.3, but for this section all we need to consider is that $\psi(x)$ is a function that resides in the same space as $f(x)$.

Suppose both $\psi(x)$ and $f(x)$ can be numerically approximated with Lagrange polynomials. Then

$$\psi(x) \approx \psi_N(x) = \sum_{i=0}^{N} \psi(x_i)L_i(x) \text{ and} \tag{2.12}$$

$$f(x) \approx f_N(x) = \sum_{i=0}^{N} f(x_i)L_i(x). \tag{2.13}$$

This leads to the integral approximation

$$\int_{-1}^{+1} f(x)\psi(x)dx \approx \int_{-1}^{+1} f_N(x)\psi_N(x)dx = \int_{-1}^{+1} \sum_{i=0}^{N} \sum_{j=0}^{N} f(x_i)[L_i(x)L_j(x)]\psi(x_j)dx,$$

which is further simplified to

$$\sum_{i=0}^{N} \sum_{j=0}^{N} \psi(x_j) \left[\int_{-1}^{+1} L_i(x)L_j(x)dx \right] f(x_i) = \boldsymbol{\psi}^T \boldsymbol{M}^e \boldsymbol{f}, \qquad (2.14)$$

where

$$\int_{-1}^{+1} L_i(x)L_j(x)dx = \boldsymbol{M}^e_{ij} \qquad (2.15)$$

and $\boldsymbol{M}^e_{ij}$ is a full matrix per element that can integrate an order $2N$ function exactly. Furthermore, $\boldsymbol{\psi}$ and $\boldsymbol{f}$ are the evaluation of $\psi(x)$ and $f(x)$ at the LGL points stored in column vectors:

$$\boldsymbol{f} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_N) \end{bmatrix}, \quad \boldsymbol{\psi} = \begin{bmatrix} \psi(x_0) \\ \psi(x_1) \\ \vdots \\ \psi(x_N) \end{bmatrix}.$$

Inexact integration is similar to exact integration. Instead of numerically approximating the functions separately with Lagrange polynomials as seen above, we first multiply the function and test function together at the LGL points and produce a new interpolant for numerical integration:

$$\int_{-1}^{+1} f_N(x)\psi_N(x)dx \approx \int_{-1}^{+1} \sum_{i=0}^{N} f(x_i)\psi(x_i)L_i(x)dx = \sum_{i=0}^{N} f_i\psi_i\omega_i. \qquad (2.16)$$

Once again $L_i(x)$, under integration, generates the weights, w_i, as seen in (2.9), and w_i is only computed once at the LGL points. These weights can be stored

in a diagonal mass matrix, called M, and used in an one-dimensional problem. Therefore, (2.11) can now be written in the matrix form below:

$$\int_{-1}^{+1} f(x)\psi(x)dx \approx \psi^T M f. \tag{2.17}$$

As discussed in Section 2.2, using this method with LGL points allows us to integrate a polynomial order of $2N-1$ exactly. Throughout the remainder of this thesis, we only consider inexact integration. This is because the diagonal structure of the mass matrix greatly simplifies the implementation of the method. Additionally, the Jacobians and surface Jacobians are easier to handle when moving into multiple dimensions. We discuss more about Jacobians and surface Jacobians in subsequent sections and chapters.

2.4 Concept of the Differentiation Matrix

The differentiation matrix has a purpose similar to that of the mass matrix, but it is obviously used for differentiation. Like the mass matrix, the differentiation matrix is found using Lagrange polynomials by

$$D_{ij} = \frac{dL_j(x_i)}{dx}, \tag{2.18}$$

$$\frac{df_N(x_i)}{dx} = \sum_{i=0}^{N} \frac{dL_j(x_i)}{dx} f_j = \sum_{i=0}^{N} D_{ij} f_j, \tag{2.19}$$

where it should be evident from (2.19) that Df approximates $\frac{df}{dx}$ at all the nodes. As you can see, D is simply the derivative of the Lagrange polynomials evaluated at the LGL nodes. You should further notice that D is a full matrix.

Let us examine how the differentiation matrix is used for discretization. Consider the integral

$$\int_{-1}^{+1} \frac{df(x)}{dx} \psi(x)dx, \tag{2.20}$$

where $\psi(x)$ is a test function that resides in the same space as $f(x)$. Once again, the

13

functions are numerically approximated with a set of Lagrange polynomials

$$\psi_N(x) \approx \sum_{i=0}^{N} \psi(x_i) L_i(x), \tag{2.21}$$

$$\frac{df_N(x)}{dx} \approx \sum_{i=0}^{N} f(x_i) \frac{dL_i(x)}{dx}, \tag{2.22}$$

leading to

$$\int_{-1}^{+1} \frac{df(x)}{dx} \psi(x) dx \approx \int_{-1}^{+1} \sum_{i=0}^{N} \sum_{j=0}^{N} f(x_i) \left[\frac{dL_i(x)}{dx} L_j(x) \right] \psi(x_j) dx, \tag{2.23}$$

which further simplifies to

$$\sum_{i=0}^{N} \sum_{j=0}^{N} \psi(x_j) \left[\int_{-1}^{+1} \frac{dL_i(x)}{dx} L_j(x) dx \right] f(x_i) \approx \psi^T M D f. \tag{2.24}$$

As you can see, (2.18) is nestled right in the middle of (2.24). Once again, both ψ and f are the evaluation of $\psi(x)$ and $f(x)$ at the LGL points.

2.5   Change of Variables

Mapping from a physical space to a computational reference space requires a change of variables. In this case, we are going to map from $x \in [x_n, x_{n+1}]$ to $\xi \in [-1, +1]$, where $n = 0, \ldots, N$. Furthermore, a Jacobian arises when conducting this change of variables [9]. In one dimension, the Jacobian is simply $\frac{h}{2}$, where the element size is $h = \triangle x$ for an equally spaced grid of elements. This one-dimensional Jacobian is used extensively in Chapter 3 for discretization and is annotated by $J = \frac{h}{2}$. In two dimensions, it is a little more complicated and is discussed in greater detail in Chapter 4. However, the concept remains the same for one dimension and two dimensions.

In one dimension, we want the computational reference domain to be from $\xi \in [-1, +1]$ for two points from a linear element, x_n and x_{n+1}. This concept is depicted

in Figure 2.4.
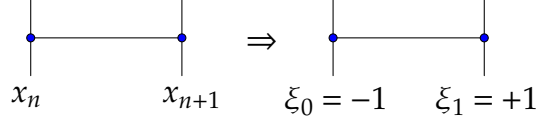
Figure 2.4: Physical Space to Computational Space

The change of variables for Figure 2.4 is obtained by a linear combination

$$x(\xi) = \left(\frac{(1-\xi)}{2}\right)x_n + \left(\frac{(1+\xi)}{2}\right)x_{n+1}, \tag{2.25}$$

that allows us to approximate the coordinates of the element. For example, if $\xi = -1$ in (2.25), then that maps to x_n and if $\xi = +1$, that maps to x_{n+1}. Taking the derivative of (2.25) yields

$$\frac{dx}{d\xi} = \frac{(x_{n+1} - x_n)}{2} = \frac{h}{2}, \tag{2.26}$$

and $dx = \frac{h}{2}d\xi$ [3]. Therefore, using (2.25) and (2.26) for the change of variables for integration gives

$$\int_{x_n}^{x_{n+1}} f(x)dx = \int_{-1}^{+1} \frac{h}{2} f(x(\xi))d\xi, \tag{2.27}$$

where x_n and x_{n+1} are the left and right boundaries for the physical element. Constructing this change of variables for each element is a key foundation for local element based Galerkin methods. Instead of building different matrices for each element and solving them individually, we can now construct the required matrices for a reference element and then use the metrics terms to scale the reference element to the physical space [3]. Metric terms are discussed in greater detail in Chapter 4. As for using the differentiation matrices, D, the change of variables requires the chain rule, which creates a $\frac{d\xi}{dx} = \frac{2}{h}$ multiplied by $\frac{dx}{d\xi} = \frac{h}{2}$, cancelling these terms out. The equation below is a visual example for the above sentence showing the change

15

of variables for (2.20):

$$\int_{x_n}^{x_{n+1}} \psi(x) \frac{df(x)}{dx} dx = \int_{-1}^{+1} \psi(\xi) \frac{d\xi}{dx} \frac{df(x(\xi))}{dx} \frac{dx}{d\xi} d\xi = \int_{-1}^{+1} \psi(\xi) \frac{df(x(\xi))}{dx} d\xi \approx \psi^T M D f.$$

This section assists in the construction of the one-dimensional grid further investigated in Section 3.1.

CHAPTER 3:
One-Dimensional Discontinuous Galerkin

Let us start working through the Discontinuous Galerkin method for the linear acoustic wave equation in one dimension. Consider the equation

$$\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p, \; x \in \Omega \subset \mathbb{R}^d, \; t > 0, \tag{3.1}$$

where p is the pressure, $c > 0$ is the wave speed, and d is the spatial dimension. On the boundary, represented by $\partial\Omega$, we impose periodic boundary conditions. We are now going to split this second-order wave equation into two equations with the auxiliary variable, ω, equalling the time derivative for pressure [2]:

$$\frac{\partial \omega}{\partial t} = c^2 \nabla^2 p, \tag{3.2}$$

$$\frac{\partial p}{\partial t} = \omega. \tag{3.3}$$

3.1   One-Dimensional Grid

Figure 3.1 shows an example of a one-dimensional grid of equally spaced elements, with polynomial order $N = 6$ and three elements, covering $\Omega = [-1, +1]$. The LGL points, depicted by red $\times$ symbols, are used for the interpolation points per element. Polynomial order, N, is defined as the maximum order of a polynomial that can be represented exactly on each element. Obviously, the grid varies depending on N and number of elements (Ne). As you can see, the LGL points are not evenly spaced across the individual elements and cluster towards the boundaries of each element. There are $N + 1$ LGL points per element, where each element is represented by Ω_j, with $j = 1, \ldots, Ne$. Remember, this is a one-dimensional grid with $\Omega = [-1, +1]$ and each element (Ω_j) is mapped to $\xi \in [-1, +1]$ through a change of variables, as discussed in Section 2.5. This one-dimensional grid is simple; however, in two dimensions, the grid is a little more complicated and is discussed in Chapter 4.
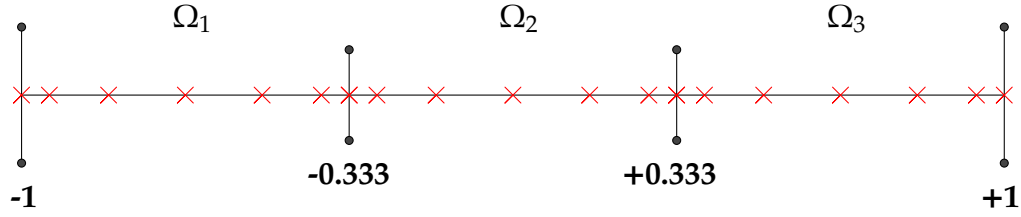
Figure 3.1: One-Dimensional Grid with N = 6 and Ne = 3.

3.2 Variational Form

Understanding (3.2) and (3.3) motivates the variational form that is needed for us to build the set of equations for DG. This section explains the process of finding the variational form of (3.2) and (3.3). Furthermore, the discretization of these equations yields the necessary equations for the DG numerical approximation. First, we need to focus on finding a variational form.

3.2.1 First Variational Equation

Equation (3.4) is the most trivial of the three variational form equations. If $\frac{\partial p}{\partial t} = \omega$, it follows that

$$\int_{\Omega_j} \left(\frac{\partial p}{\partial t} - \omega \right) = 0. \tag{3.4}$$

Using (3.4) in Section 3.4 assists in finding the needed equations for the DG approximation.

3.2.2 Second Variational Equation

Given that $\frac{\partial \omega}{\partial t} = c^2 \nabla^2 p$, by multiplying this equation by a test function, ψ, and integrating over Ω_j yields

$$\int_{\Omega_j} \psi \left(\frac{\partial \omega}{\partial t} - c^2 \nabla^2 p \right) = 0. \tag{3.5}$$

We define test functions in further detail in the following section. By conducting integration by parts on (3.5) and inserting a numerical flux at the boundary of the

18

element, we are able to find the second equation in the variational form to be

$$\int_{\Omega_j} \left(\psi \frac{\partial \omega}{\partial t} + c^2 \nabla \psi \cdot \nabla p \right) = c^2 \int_{\partial \Omega_j} \psi \, \boldsymbol{n} \cdot (\nabla p)^*. \tag{3.6}$$

For notation purposes, a * term in the equation identifies the numerical flux. Using a numerical flux is a numerical technique for coupling the elements that is commonly used in DG. For example, $(\nabla p)^*$ is the coupled numerical flux computation for the gradient of the pressure at the boundaries $(\partial \Omega_j)$ for neighboring elements within the domain. An explicit computation of the numerical flux is discussed in greater detail in Section 3.5.

3.2.3   Third Variational Equation

In order to find the last variational equation, we are going to multiply the gradient of $\left(\frac{\partial p}{\partial t} - \omega \right)$ with the gradient of a test function $(\nabla \psi)$:

$$\int_{\Omega_j} \nabla \psi \nabla \left(\frac{\partial p}{\partial t} - w \right) = \int_{\Omega_j} \nabla \psi \nabla \frac{\partial p}{\partial t} - \int_{\Omega_j} \nabla \psi \nabla \omega = 0. \tag{3.7}$$

Let us focus on the last portion of (3.7). By conducting integration by parts we find

$$\int_{\Omega_j} \nabla \psi \nabla \omega = \int_{\partial \Omega_j} (\nabla \psi \cdot \boldsymbol{n}) \omega - \int_{\Omega_j} \left(\nabla^2 \psi \right) \omega. \tag{3.8}$$

By introducing a numerical flux to the boundary term of (3.8) and substituting it into (3.7), we find that

$$\int_{\Omega_j} \nabla \psi \nabla \frac{\partial p}{\partial t} - \int_{\partial \Omega_j} (\nabla \psi \cdot \boldsymbol{n}) \omega^* + \int_{\Omega_j} \left(\nabla^2 \psi \right) \omega = 0. \tag{3.9}$$

Let us focus now on the last term from (3.9). Conducting integration by parts again on the last portion of (3.9) yields

$$\int_{\Omega_j} \left(\nabla^2 \psi \right) \omega = \int_{\partial \Omega_j} (\nabla \psi \cdot \boldsymbol{n}) \omega - \int_{\Omega_j} \nabla \psi \nabla \omega. \tag{3.10}$$

19

Next, we substitute (3.10) back into (3.9) and combine like terms to conclude the following:

$$\int_{\Omega_j} \nabla \psi \nabla \frac{\partial p}{\partial t} - \int_{\partial \Omega_j} (\nabla \psi \cdot \boldsymbol{n})(\omega^* - \omega) - \int_{\Omega_j} \nabla \psi \nabla \omega = 0. \tag{3.11}$$

Finally, from (3.11), we arrive at the final variational form

$$\int_{\Omega_j} \nabla \psi \nabla \left(\frac{\partial p}{\partial t} - \omega \right) = \int_{\partial \Omega_j} (\nabla \psi \cdot \boldsymbol{n})(\omega^* - \omega). \tag{3.12}$$

Overall, equations (3.4), (3.6), and (3.12) together are the variational form of (3.1) that we discretize using DG.

3.3   Test Functions

Using Discontinuous Galerkin, we are looking for p and ω from (3.2) and (3.3) that satisfies the variational form found in Section 3.2 for all piecewise polynomial test functions. Since we are working to build an approximation to a function with piecewise polynomials, our space is the space of Nth-degree piecewise polynomials, with the objective of solving for a piecewise polynomial that represents the solution. For example, suppose we wanted to find a constant approximation for $f(x)$, such that the error was orthogonal (in an integral sense) to all other constants. That said, we want to find $\alpha \in \mathbb{R}$ such that

$$\int_{-1}^{+1} \psi(x) f(x) dx \implies \int_{-1}^{+1} (C)(\alpha - f(x)) dx = 0,$$

for all $C \in \mathbb{R}$. If $f(x) = x^2$, then the problem is to find α such that

$$\int_{-1}^{+1} (C\alpha - Cx^2) dx = \left[C\alpha x - \frac{1}{3} Cx^3 \right]_{-1}^{+1} = 0,$$

which is simplified to

$$2C\alpha - \frac{2}{3} C = 2C \left(\alpha - \frac{1}{3} \right) = 0. \tag{3.13}$$

Therefore, $\alpha = \frac{1}{3}$ is an approximation for x^2 found using a 0th degree test function (C). Expanding this concept to DG is essentially the same, except we are using N^{th}-degree piecewise polynomials as the test functions to find a Nth degree polynomial approximation for the solution on each element. We use this concept for the remainder of this paper and Section 3.3.1 further highlights the use of test functions for the discretization in Section 3.4 and Section 4.2.

3.3.1 Test Functions in Discretization

As discussed in the previous section, we are using N^{th}-degree piecewise polynomials as test functions for DG. In the following section and chapter, we discretize the variational form to find a discrete approximation for the continuous equation in one dimension and two dimensions. These discrete approximations, with respect to ψ, all have the similar form of

$$\psi^T (Ap - B\omega) = 0, \tag{3.14}$$

where A and B are matrices and p and ω are the solution vectors. Since (3.14) holds for all ψ, then $Ap - B\omega$ must equal zero. This concept holds true for Section 3.4 and Section 4.2. Therefore, in what follows, ψ is not listed in the final discrete approximations.

3.4 One-Dimensional Discretization

In this section, we derive discrete versions of Equations (3.4), (3.6), and (3.12). The idea of discretization is to replace a continuous equation with a consistent discrete approximation. As you have probably noticed, with Galerkin methods we use the variational form for discretizing equations [3]. We focus on developing a spatial discretization of the equations and time is integrated separately with a Runge–Kutta method. We isolate the time derivatives and combine the discrete forms of (3.4) and (3.12) into one discrete equation [10]. As discussed in Section 2.5, the change of variables requires the Jacobian of $J = \frac{h}{2}$ for an equally spaced grid of elements. As a reminder, the concept from Sections 3.3.1 is applied to the discretization in Section 3.4.2 and Section 3.4.3. We discretize each equation individually.

3.4.1 First One-Dimensional Discretization Equation

Using the concepts established in Chapter 2, the discrete version of (3.4) is

$$\mathbf{1}^T MJ \frac{d\mathbf{p}}{dt} - \mathbf{1}^T MJ \boldsymbol{\omega} = 0. \tag{3.15}$$

Within (3.15), $\mathbf{1}$ is a vector of ones and is needed because in (3.4) we are integrating against the function $\psi(x) = 1$ [10]. Additionally, $\frac{d\mathbf{p}}{dt}$ and $\boldsymbol{\omega}$ are column-vector approximations of the solution at the grid points. This is similar to f from Section 2.3.

3.4.2 Second One-Dimensional Discretization Equation

The following is the discrete version of (3.6):

$$MJ \frac{d\boldsymbol{\omega}}{dt} + c^2 J^{-1} D^T MD\mathbf{p} = c^2 J^{-1} \left[e_N \left(\frac{\partial \mathbf{p}}{\partial \xi} \right)_N^* - e_0 \left(\frac{\partial \mathbf{p}}{\partial \xi} \right)_0^* \right]. \tag{3.16}$$

The column vectors e_N and e_0 consist entirely of zeros except for the last "row" in e_N and the first "row" of e_0 being ones. Using e_N and e_0 is a way of ensuring that the first and last portion of information from the numerical flux of $\left(\frac{\partial \mathbf{p}}{\partial \xi} \right)_N^*$ and $\left(\frac{\partial \mathbf{p}}{\partial \xi} \right)_0^*$ is used for the calculation. The numerical flux is discussed in greater detail in Section 3.5, but it is essentially a method for coupling the solution on either side of an element boundary. Additionally, discretization of

$$\int_{\Omega_j} \nabla \psi \cdot \nabla p$$

from (3.6) requires two differentiation matrices; D^T takes the derivative of the test function, M is the integral, and D is the derivative of p. This accounts for the $D^T MD$ portion of (3.16).

3.4.3 Third One-Dimensional Discretization Equation

The following is the discrete version of (3.12):

$$J^{-1} D^T MD \frac{d\mathbf{p}}{dt} - J^{-1} D^T MD \boldsymbol{\omega} = J^{-1} D^T e_N \left(\omega_N^* - e_N^T \boldsymbol{\omega} \right) - J^{-1} D^T e_0 \left(\omega_0^* - e_0^T \boldsymbol{\omega} \right). \tag{3.17}$$

Once again, the $\int_{\Omega_j} \nabla \psi \cdot \nabla \frac{\partial p}{\partial t}$ and $\int_{\Omega_j} \nabla \psi \cdot \nabla \omega$ from (3.12) are accounted for by the $D^T M D$ portions and ω^* is computed through the flux at the element boundaries.

3.4.4 Combination of One-Dimensional Discretization Equations

Now that we have three discrete equations, our goal is to get the problem into the form of two equations and two vector unknowns. Combining (3.15) and (3.17) yields

$$
\left(J^{-1}D^T M D + \mathbb{1}MJ\right)\frac{dp}{dt} = \left(J^{-1}D^T M D + \mathbb{1}MJ\right)w + J^{-1}D^T e_N\left(\omega_N^* - e_N^T\omega\right) \\
- J^{-1}D^T e_0\left(\omega_0^* - e_0^T\omega\right). \tag{3.18}
$$

For notational purposes, $\mathbb{1}$ is a necessary square matrix of ones of size $(N+1, N+1)$ because (3.15) is a scalar. Additionally, letting $\left(J^{-1}D^T M D + \mathbb{1}MJ\right)$ be equal to the variable $M_{\mathbb{1}}$ and simplifying (3.18) produces

$$
M_{\mathbb{1}}\frac{dp}{dt} = M_{\mathbb{1}}\omega + J^{-1}D^T\left(e_N\omega_N^* - e_0\omega_0^*\right) - J^{-1}D^T\left(e_N e_N^T\omega - e_0 e_0^T\omega\right). \tag{3.19}
$$

For the second equation, by rearranging (3.16) we can isolate $\frac{d\omega}{dt}$ as seen in the following:

$$
MJ\frac{d\omega}{dt} = -c^2 J^{-1}D^T M D p + c^2 J^{-1}\left(e_N\left(\frac{\partial p}{\partial \xi}\right)_N^* - e_0\left(\frac{\partial p}{\partial \xi}\right)_0^*\right). \tag{3.20}
$$

Equations (3.19) and (3.20) are the two systems of ordinary differential equations that we solve numerically using a Runge–Kutta method.

3.5 Numerical Flux

When using a DG method, we have to account for the discontinuity that exists between the elements. Meaning, when each element is represented by a polynomial for ω and p, sampling ω and p at the boundary of neighboring elements, as shown in Figure 3.2, yields different results [3]. The numerical flux is a method for enforcing (approximately) continuity between the elements. There are multiple flux methods, but in what follows we only discuss the central and upwinding fluxes.
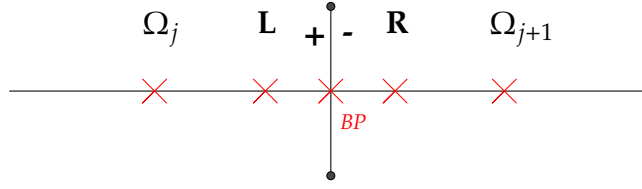
Figure 3.2: One-Dimensional Element Boundary

The central flux is the easiest method to understand, but it does not necessarily produce the best results. The central flux is the average of the values at each boundary element. In Figure 3.2, the neighboring elements, Ω_j and Ω_{j+1}, have different approximations at the boundary point (BP). For notation purposes, the superscript (+) represents the left element's gridpoint and the superscript (−) represents the right element's gridpoint. Furthermore, as in Section 3.4, the superscript (∗) denotes the numerical flux term. This notation comes into further use in the following section and chapters. The central flux, for both ω and $\frac{\partial p}{\partial x}$, is defined by the following:

$$\omega^* = \frac{1}{2}(\omega^- + \omega^+), \tag{3.21}$$

$$\frac{\partial p^*}{\partial x} = \frac{1}{2}\left[\left(\frac{\partial p}{\partial x}\right)^- + \left(\frac{\partial p}{\partial x}\right)^+\right]. \tag{3.22}$$

Using the central flux is a useful beginning step when coding the flux into any algorithm. The central flux is simple and still achieves convergence when coded correctly. However, central fluxes often have suboptimal convergence rates and can lead to oscillatory approximations due to a lack of dissipation. To improve this, we need a numerical flux that does not just average the information at the element boundaries, but can account for the physical propagation of information across the boundaries. This leads us to the upwinding flux.

An upwinding flux is a method for computing a flux across the element boundaries based on the physical propagation of information. In the acoustic wave equation, the waves that are propagating through the physical system can move in both directions across the element boundaries. We are looking at the information that

24

is "upwind" of the wave's direction of motion, hence called an upwind flux. For example, for the advection equation, if the wave is moving across the element boundary from the left to the right, we select the information from the left element and vice versa for the wave moving from right to left across the boundary. However, for the wave equation we want to decouple the wave propagation into two one-way advection equations to take advantage of this "upwind" concept.

Since we are working in one dimension, let us take (3.2) and derive the upwind flux. Defining the auxiliary variable $q = p_x$, we rewrite (3.2) as

$$\omega_t = c^2 q_x. \tag{3.23}$$

Additionally, understanding that $p_{xt} = p_{tx}$, we can further deduce that $q_t = \omega_x$ and write a first order system of equations as

$$\begin{bmatrix} \omega \\ q \end{bmatrix}_t = \begin{bmatrix} 0 & c^2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \omega \\ q \end{bmatrix}_x.$$

By defining $\boldsymbol{U} = \begin{bmatrix} \omega \\ q \end{bmatrix}$, we simplify the system of equations into a matrix vector form:

$$\boldsymbol{U}_t - \boldsymbol{A}\boldsymbol{U}_x = \boldsymbol{0}, \quad \boldsymbol{A} = \begin{bmatrix} 0 & c^2 \\ 1 & 0 \end{bmatrix}. \tag{3.24}$$

Solving for the eigenvalues and eigenvectors of $\boldsymbol{A}$, we find the eigenvalues, $\lambda_1 = c$ and $\lambda_2 = -c$, and the eigenvector matrix ($\boldsymbol{V}$) and its inverse ($\boldsymbol{V^{-1}}$) to be

$$\boldsymbol{V} = \begin{bmatrix} c & -c \\ 1 & 1 \end{bmatrix}, \text{ and } \boldsymbol{V^{-1}} = \begin{bmatrix} \frac{1}{2c} & \frac{1}{2} \\ -\frac{1}{2c} & \frac{1}{2} \end{bmatrix}.$$

We now want to find a linear combination of the original variables, ω and q, whose solutions are independent of each other. We can do this by diagonalizing (3.24):

$$\left[\boldsymbol{V^{-1}}\boldsymbol{U} \right]_t - \left[\boldsymbol{V^{-1}}\boldsymbol{A}\boldsymbol{V} \right] \left[\boldsymbol{V^{-1}}\boldsymbol{U} \right]_x = \boldsymbol{0}.$$

We define a new set of variables r_1 and r_2, known as the characteristic variables, by

$$r = V^{-1}U = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix},$$

and let Λ be the diagonal matrix of eigenvalues

$$\begin{bmatrix} V^{-1}AV \end{bmatrix} = \Lambda = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}.$$

We now have a system of first order one-way advection equations that allow us to upwind because the propagation of the waves are decoupled across the element boundaries by the characteristic variables, r_1 and r_2, as seen with

$$r_t - \Lambda r_x = 0 \Rightarrow \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}_t - \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}_x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{3.25}$$

By analysis of the function within the element, with f_1 and f_2 being the initial conditions for r_1 and r_2, we find that

$$r_1(x,t) = f_1(x + \lambda_1 t) = f_1(x + ct), \text{ and } r_2(x,t) = f_2(x + \lambda_2 t) = f_2(x - ct),$$

which further describes the propagation of the wave across the element boundaries with r_1, flowing right to left, and r_2, flowing left to right. Figure 3.3 depicts this propagation. We know the direction of propagation and can now find the flux values. By letting $r_1^* = r_1^-$ and $r_2^* = r_2^+$, we have chosen the numerical fluxes as the
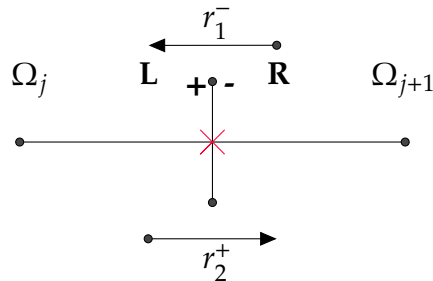
Figure 3.3: One-Dimensional Element Boundary Decoupled Wave Propagation

upwind values. Knowing that $r = V^{-1}U$, we can easily find a system of equations to solve. Note that

$$\begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{2c} & \frac{1}{2} \\ -\frac{1}{2c} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \omega \\ q \end{bmatrix},$$

so with $r_1^* = r_1^-$ and $r_2^* = r_2^+$ we have

$$r_1^- = \frac{1}{2c}\omega^- + \frac{1}{2}q^- = \frac{1}{2c}\omega^* + \frac{1}{2}q^* \text{ and} \tag{3.26}$$

$$r_2^+ = -\frac{1}{2c}\omega^+ + \frac{1}{2}q^+ = -\frac{1}{2c}\omega^* + \frac{1}{2}q^*. \tag{3.27}$$

We now have two equations and two unknowns where we can solve for ω^* and q^*:

$$q^* = \frac{1}{2c}(\omega^- - \omega^+) + \frac{1}{2}(q^+ + q^-); \tag{3.28}$$

$$\omega^* = \frac{1}{2}(\omega^- + \omega^+) + \frac{c}{2}(q^- - q^+). \tag{3.29}$$

Lastly, with $q = \frac{\partial p}{\partial x}$, (3.28) and (3.29) are the upwinding flux equations for ω and $\frac{\partial p}{\partial x}$ in one dimension. It should be further noted that both central flux equations, (3.21) and (3.22), are included in (3.28) and (3.29). The portion of (3.28) and (3.29) that include the variable c is considered the upwinding portion and is used separately for the upwind flux energy analysis in Chapter 5.

3.6   One-Dimensional Discontinuous Galerkin Results

After laying the groundwork, we now apply these concepts to an actual problem in one dimension using an upwind flux. We use the exact solution

$$p(x,t) = \sin(n\pi x)\sin(n\pi t), \tag{3.30}$$

$$\frac{\partial p}{\partial t} = \omega = n\pi\sin(n\pi x)\cos(n\pi t), \tag{3.31}$$

on the domain $\Omega = [-1, +1]$ with periodic boundary conditions where n is an integer. To build the initial condition, (3.30) and (3.31) are evaluated at $t = 0$ across a one-dimensional grid of LGL points where $x \in [-1, +1]$. These initial conditions are used

within (3.19) and (3.20) and evaluated through a Runge–Kutta five-stage fourth-order (RK54) accurate iterative method [11]. Runge–Kutta methods are numerical methods that approximate a system of ordinary differential equations. With the RK54 method, (3.19) and (3.20) are evaluated through five stages per time-step along the grid of LGL points. All of these evaluations are combined to produce a fourth order accurate or higher approximation [8]. The following section discusses the results of applying a DG method for the above one-dimensional problem. The DG implementation can be found in Appendix B.

The implementation is tested using various polynomial orders and numbers of equally spaced elements within the domain listed in Table 3.1. Once again, the

Table 3.1: Discontinuous Galerkin Tested Information

Polynomial Orders (N)	Number of Elements (Ne)
$N = 4, 6, 8$	$Ne = 2, 4, 8, 16$
$N = 16$	$Ne = 2, 4, 8$

DG algorithm uses inexact integration for computational speed to ensure an easily invertible diagonal mass matrix. Figure 3.4 shows the log of the error vs. log of the number of elements for $N = 4, 6, 8$ calculated using the global L^2 error norms. As expected for both ω and p, Figure 3.4 displays increasing convergence rates as the polynomial order increases. Increasing the order of the local approximation gives the fastest convergence rates, as apposed to increasing the number of elements, due to the fact that the global error is dependent on the polynomial order [1] as depicted by

$$\| \varepsilon \|_2 \leq Ch^q. \tag{3.32}$$

In (3.32), C is a constant that is not dependent on the element size h but does depend on the final time, t, of the solution. In this case, N is proportional to q in (3.32). As the polynomial order (N) increases, the convergence rates increase.

Table 3.2 displays the convergence rates for the tested information from Table 3.1 for ω and p. As you can see, as the polynomial order increases, the convergence rates increase. Furthermore, Table 3.2 depicts the convergence rates being near or
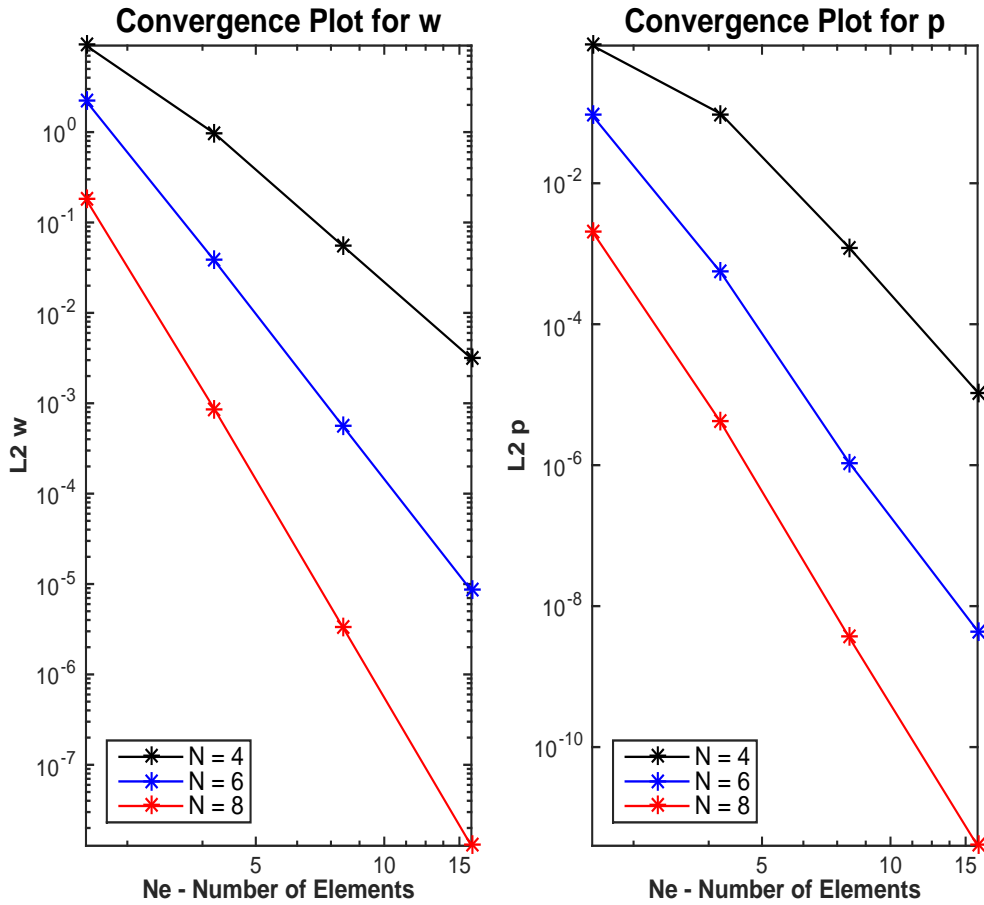
Figure 3.4: Convergence Rates For $N = 4, 6, 8$ at $Ne = 2, 4, 8, 16$

better than its associated polynomial order, where the convergence rate for ω is of order N and the convergence rate for p is of order $N + 2$.

Most DG methods are expected to be order N, $N + \frac{1}{2}$, or $N + 1$ [1]. However, the results yielded N and $N + 2$ when using the same space of functions for ω and p. In Appelö and Hagstrom's paper [2] they proved that when p is an $N + 1$ order polynomial and ω is an N order polynomial you get optimal convergence of $N + 1$ for this method [2]. In Table 3.2 and 3.3, these convergence rates are an observation; a proof would require further analysis.

Table 3.2: Convergence Rates for N = 4, 6, 8

Convergence Rates			
N / Ne	Ne = 2 to 4	Ne = 4 to 8	Ne = 8 to 16
N = 4 (ω)	3.2286	4.1370	4.1317
N = 4 (p)	3.2103	6.3134	6.8160
N = 6 (ω)	5.8699	6.0861	6.0456
N = 6 (p)	7.3975	8.9715	7.9343
N = 8 (ω)	7.7170	8.0165	8.0291
N = 8 (p)	8.9915	10.1413	9.8260

Table 3.3: Convergence Rate for N = 16

Convergence Rates		
N / Ne	Ne = 2 to 4	Ne = 4 to 8
N = 16 (ω)	14.0437	15.6555
N = 16 (p)	16.5784	17.5193

When testing higher order polynomials, we can increase the spatial error for the given function and decrease the time-step in order to ensure that we can neglect any time-stepping errors during testing [1]. Doing this allowed accurate convergence rate measurements when testing 16th order polynomials. Figure 3.5 and Table 3.3 display how even higher order polynomials can be used to approximate the solution with a much higher convergence rate. However, using higher order polynomials comes with a computational cost with respect to time. Higher orders are much more accurate, as seen in Figure 3.5, but the computational time increases.
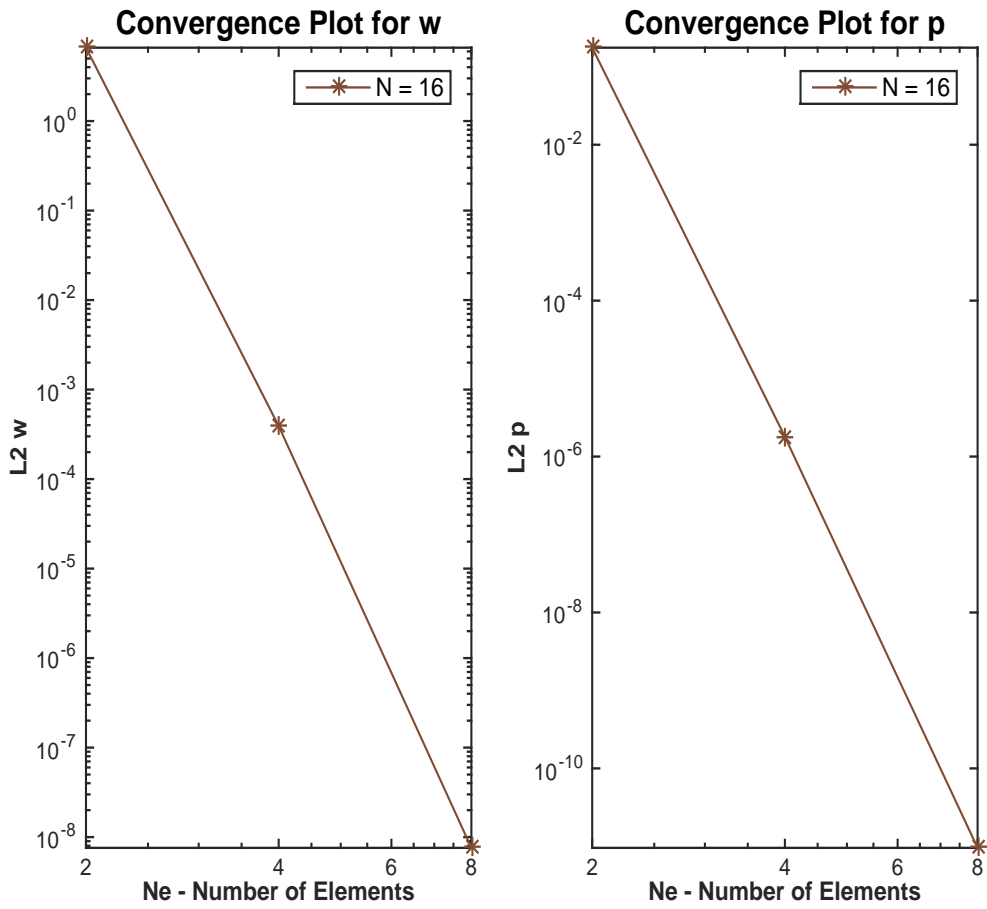
Figure 3.5: Convergence Rates For N = 16 at Ne = 2,4,8

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Two-Dimensional Discontinuous Galerkin

Moving into two dimensions is obviously more challenging than one dimension. The first portion of Chapter 4 focuses on an example two-dimensional grid with quadrilaterals and explains the formulation of the metric terms needed for discretization. Section 4.2 is the discretization of (3.4), (3.6), and (3.12) in two dimensions. As discussed in Chapter 2, working with integrals and mapping the physical domain into a reference domain requires a change of variables. In two dimensions, this change of variables produces a set of Jacobian determinants as well as surface Jacobians on each element. Lastly, we apply these concepts to a two-dimensional problem on a washer domain with curved elements.

4.1 Two-Dimensional Grid

Instead of a one-dimensional grid, which is just a line of elements in Chapter 3, moving into two dimensions requires a two-dimensional grid of elements. Building the grid requires a change of variables, from the x and y spatial coordinates to the ξ and η reference coordinates [9]. Figure 4.1 is an example of a mapping for
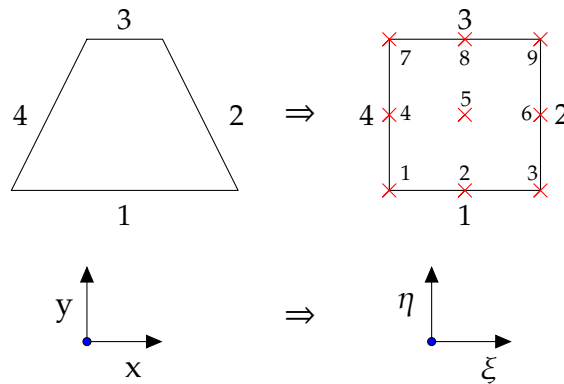
Figure 4.1: Example Mapping of a Quadrilateral Element in Two Dimensions with Degrees of Freedom for $N = 2$.

a quadrilateral element in two dimensions. When using quadrilaterals for two dimensions, we have $(N+1)^2$ LGL points in both the η and ξ directions, as depicted

by the red × symbols in Figure 4.1. The numbered order of the LGL points in Figure 4.1 is important for the storage of data. For example, in Section 4.2 the solution is stored in column vectors

$$
\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_M \end{bmatrix}, \quad p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_M \end{bmatrix},
$$

where there are $M = (N+1)^2$ LGL points or degrees of freedom.

For notational purposes, the matrix of Jacobian determinates is annotated by J and the matrix of surface Jacobians is annotated by one of the four sides of the element, such as S_{j1} for side one of the element, as labeled in Figure 4.1. Though Figure 4.1 is an example of elements with straight boundaries, we develop the scheme for general curvilinear quadrilateral elements.

4.1.1 Metric Terms

As discussed before, we need to conduct a change of variables from (x, y) to (ξ, η), where

$$
x = x(\xi, \eta), \quad y = y(\xi, \eta), \tag{4.1}
$$

and we assume the inverse mapping exists with

$$
\xi = \xi(x, y), \quad \eta = \eta(x, y). \tag{4.2}
$$

Let us consider a single quadrilateral element, such as the one in Figure 4.1. The following explanation for the metric terms is based on Professor F.X. Giraldo's lecture notes [3]. Consider the differentials defined from (4.1) and (4.2). They are

$$
dx = \frac{\partial x}{\partial \xi} d\xi + \frac{\partial x}{\partial \eta} d\eta, \quad d\xi = \frac{\partial \xi}{\partial x} dx + \frac{\partial \xi}{\partial y} dy,
$$

$$
dy = \frac{\partial y}{\partial \xi} d\xi + \frac{\partial y}{\partial \eta} d\eta, \quad d\eta = \frac{\partial \eta}{\partial x} dx + \frac{\partial \eta}{\partial y} dy,
$$

which can be written in the matrix form

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix}, \tag{4.3}$$

$$\begin{bmatrix} d\xi \\ d\eta \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix}. \tag{4.4}$$

The Jacobian and the inverse Jacobian, along with their determinants, from (4.3) and (4.4) are

$$\mathcal{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}, \ J = det(\mathcal{J}) = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial y}{\partial \xi}\frac{\partial x}{\partial \eta}, \tag{4.5}$$

$$\mathcal{J}^{-1} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix}, \ J^{-1} = det(\mathcal{J}^{-1}) = \frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial y} - \frac{\partial \eta}{\partial x}\frac{\partial \xi}{\partial y}. \tag{4.6}$$

Focusing on (4.5), taking the inverse of the Jacobian, $\mathcal{J}$, yields

$$(\mathcal{J})^{-1} = \frac{1}{J} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \xi} & \frac{\partial x}{\partial \xi} \end{bmatrix}, \tag{4.7}$$

which must equal $\mathcal{J}^{-1}$ in (4.6) and thus

$$\begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} = \frac{1}{J} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial x}{\partial \eta} \\ -\frac{\partial y}{\partial \xi} & \frac{\partial x}{\partial \xi} \end{bmatrix}. \tag{4.8}$$

Equation (4.8) gives the metric relations

$$J\frac{\partial \xi}{\partial x} = \frac{\partial y}{\partial \eta}, \quad J\frac{\partial \eta}{\partial x} = -\frac{\partial y}{\partial \xi}, \quad J\frac{\partial \xi}{\partial y} = -\frac{\partial x}{\partial \eta}, \quad J\frac{\partial \eta}{\partial y} = \frac{\partial x}{\partial \xi}, \tag{4.9}$$

which arise after the chain rule is used to map from the physical domain to the reference domain. Lastly, on each element, we store the Jacobian determinants and metric terms (4.9) in diagonal matrices, where the storage order is as the LGL gridpoint order in Figure 4.1.

4.2 Two-Dimensional Discretization

The beauty of moving into two dimensions with quadrilateral elements is that the constructions of the mass and differentiation matrices are relatively simple. The evaluation of the surface integral over each element requires the Kronecker product of the one-dimensional mass matrix with itself. This allows us to integrate in both the ξ and η directions for each element and is annotated by $M \otimes M$, where M is just the one-dimensional mass matrix. Additionally, the differentiation matrices are

$$D_\xi = I \otimes D,$$
$$D_\eta = D \otimes I,$$

where I is the identity matrix of size $(N+1) \times (N+1)$ and D is the one-dimensional differentiation matrix.

4.2.1 First Two-Dimensional Discretization Equation

The discretization of (3.4) is very similar to the one-dimensional discretization except for the addition of the Kronecker product

$$\mathbf{1}^T J (M \otimes M) \frac{d\boldsymbol{p}}{dt} - \mathbf{1}^T J (M \otimes M) \boldsymbol{\omega} = 0. \tag{4.10}$$

Since we are in two dimensions, $\mathbf{1}$ is a vector of ones of size $(N+1)^2$ and is needed because in (3.4) we are integrating against the function $\psi(x) = 1$ [10].

4.2.2 Second Two-Dimensional Discretization Equation

The discretization for (3.6) requires much more analysis than (3.4) because of the gradients and surface terms. Due to this, we focus on the left-hand and right-hand sides separately. Here we explicitly introduce the differential into the integral for two dimensions, whereas in Section 3.4, this differential was implicit. Focusing on the left-hand side, we see that

$$\int_{\Omega_j} \left(\psi \frac{\partial \omega}{\partial t} + c^2 \nabla \psi \cdot \nabla p \right) dA = \boldsymbol{\psi}^T J (M \otimes M) \frac{d\boldsymbol{\omega}}{dt} + c^2 \int_{-1}^{+1} \int_{-1}^{+1} [\nabla \psi \cdot \nabla p] J d\xi d\eta,$$

36

where $\nabla \psi$ and ∇p are equal to the following after the change of variables:

$$\nabla p = \left[\frac{\partial \xi}{\partial x}\hat{i} + \frac{\partial \xi}{\partial y}\hat{j}\right]\frac{\partial p}{\partial \xi} + \left[\frac{\partial \eta}{\partial x}\hat{i} + \frac{\partial \eta}{\partial y}\hat{j}\right]\frac{\partial p}{\partial \eta},$$

$$\nabla \psi = \left[\frac{\partial \xi}{\partial x}\hat{i} + \frac{\partial \xi}{\partial y}\hat{j}\right]\frac{\partial \psi}{\partial \xi} + \left[\frac{\partial \eta}{\partial x}\hat{i} + \frac{\partial \eta}{\partial y}\hat{j}\right]\frac{\partial \psi}{\partial \eta}.$$

Therefore, $\nabla \psi \cdot \nabla p$ yields eight separate terms:

$$
\begin{aligned}
\nabla \psi \cdot \nabla p &= \left[\frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \xi} + \frac{\partial \eta}{\partial x}\frac{\partial p}{\partial \eta}\right]\left[\frac{\partial \xi}{\partial x}\frac{\partial \psi}{\partial \xi} + \frac{\partial \eta}{\partial x}\frac{\partial \psi}{\partial \eta}\right] + \left[\frac{\partial \xi}{\partial y}\frac{\partial p}{\partial \xi} + \frac{\partial \eta}{\partial y}\frac{\partial p}{\partial \eta}\right]\left[\frac{\partial \xi}{\partial y}\frac{\partial \psi}{\partial \xi} + \frac{\partial \eta}{\partial y}\frac{\partial \psi}{\partial \eta}\right] \\
&= \frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \xi}\frac{\partial \xi}{\partial x}\frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \xi}\frac{\partial \eta}{\partial x}\frac{\partial \psi}{\partial \eta} + \frac{\partial \eta}{\partial x}\frac{\partial p}{\partial \eta}\frac{\partial \xi}{\partial x}\frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \eta}\frac{\partial \eta}{\partial x}\frac{\partial \psi}{\partial \eta} \\
&\quad + \frac{\partial \xi}{\partial y}\frac{\partial p}{\partial \xi}\frac{\partial \xi}{\partial y}\frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial y}\frac{\partial p}{\partial \xi}\frac{\partial \eta}{\partial y}\frac{\partial \psi}{\partial \eta} + \frac{\partial \eta}{\partial y}\frac{\partial p}{\partial \eta}\frac{\partial \xi}{\partial y}\frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial y}\frac{\partial p}{\partial \eta}\frac{\partial \eta}{\partial y}\frac{\partial \psi}{\partial \eta}.
\end{aligned}
\tag{4.11}
$$

Focusing on the first term of the second equation in (4.11), we find

$$\int_{-1}^{+1}\int_{-1}^{+1}\left(\frac{\partial \xi}{\partial x}\right)\frac{\partial p}{\partial \xi}\left(\frac{\partial \xi}{\partial x}\right)\frac{\partial \psi}{\partial \xi}J d\xi d\eta = \int_{-1}^{+1}\int_{-1}^{+1}\left(J\frac{\partial \xi}{\partial x}\right)\frac{\partial p}{\partial \xi}\left(J\frac{\partial \xi}{\partial x}\right)\frac{\partial \psi}{\partial \xi}J^{-1} d\xi d\eta, \quad (4.12)$$

where we have multiplied in JJ^{-1} to place a Jacobian determinate with both of the metric terms. By substituting in the metric terms from (4.9) and discretizing, we produce

$$
\begin{aligned}
\int_{-1}^{+1}\int_{-1}^{+1}\frac{\partial y}{\partial \eta}\frac{\partial p}{\partial \xi}\frac{\partial y}{\partial \eta}\frac{\partial \psi}{\partial \xi}J^{-1} d\xi d\eta &= \left(\frac{\partial y}{\partial \eta}D_\xi \psi\right)^T J^{-1}(M\otimes M)\left(\frac{\partial y}{\partial \eta}D_\xi p\right) \\
&= \psi^T D_\xi^T\left[\frac{\partial y}{\partial \eta}J^{-1}(M\otimes M)\frac{\partial y}{\partial \eta}\right]D_\xi p.
\end{aligned}
\tag{4.13}
$$

The same process is completed for each of the eight terms in (4.11) to find the final discrete form for the left-hand side of (3.4) to be

$$\psi^T J(M\otimes M)\frac{d\omega}{dt} + c^2\psi^T Tp, \tag{4.14}$$

where

$$
\begin{aligned}
T = \bigg(&\; D_\xi^T \left[\frac{\partial y}{\partial \eta} J^{-1} (M \otimes M) \frac{\partial y}{\partial \eta} \right] D_\xi - D_\xi^T \left[\frac{\partial y}{\partial \eta} J^{-1} (M \otimes M) \frac{\partial y}{\partial \xi} \right] D_\eta \\
&- D_\eta^T \left[\frac{\partial y}{\partial \xi} J^{-1} (M \otimes M) \frac{\partial y}{\partial \eta} \right] D_\xi + D_\eta^T \left[\frac{\partial y}{\partial \xi} J^{-1} (M \otimes M) \frac{\partial y}{\partial \xi} \right] D_\eta \\
&+ D_\xi^T \left[\frac{\partial x}{\partial \eta} J^{-1} (M \otimes M) \frac{\partial x}{\partial \eta} \right] D_\xi - D_\xi^T \left[\frac{\partial x}{\partial \eta} J^{-1} (M \otimes M) \frac{\partial x}{\partial \xi} \right] D_\eta \\
&- D_\eta^T \left[\frac{\partial x}{\partial \xi} J^{-1} (M \otimes M) \frac{\partial x}{\partial \eta} \right] D_\xi + D_\eta^T \left[\frac{\partial x}{\partial \xi} J^{-1} (M \otimes M) \frac{\partial x}{\partial \xi} \right] D_\eta \bigg).
\end{aligned}
\tag{4.15}
$$

As you can see, the discretization for two dimensions is more tedious than one dimension.

The right-hand side of (3.6) is much simpler than the left-hand side. As a reminder, the right-hand side of (3.6) is

$$
c^2 \int_{\partial \Omega_j} \psi \, \boldsymbol{n} \cdot (\nabla p)^* ds.
$$

Since we are working with quadrilaterals, we integrate the four boundaries of each element, labeled by $\partial \Omega_j$. Figure 4.1 displays how the four boundaries of each element are numerically labeled. We pull the information from each side individually by using the following operators:

$$
L_1 = e_0^T \otimes I, \; L_2 = I \otimes e_N^T, \; L_3 = e_N^T \otimes I, \; L_4 = I \otimes e_0^T.
\tag{4.16}
$$

The four L_ℓ operators (4.16) isolate the correct information from each element boundary. Additionally, the change of variables for the right-hand side produces a set of surface Jacobians found in either the ξ or η directions by

$$
S_{j\ell} = J \sqrt{\left(\frac{\partial \eta}{\partial x} \right)^2 + \left(\frac{\partial \eta}{\partial y} \right)^2} = \sqrt{\left(\frac{\partial x}{\partial \xi} \right)^2 + \left(\frac{\partial y}{\partial \xi} \right)^2}, \quad \ell = 1,3,
\tag{4.17}
$$

$$
S_{j\ell} = J \sqrt{\left(\frac{\partial \xi}{\partial x} \right)^2 + \left(\frac{\partial \xi}{\partial y} \right)^2} = \sqrt{\left(\frac{\partial x}{\partial \eta} \right)^2 + \left(\frac{\partial y}{\partial \eta} \right)^2}, \quad \ell = 2,4.
\tag{4.18}
$$

By calculating the outward unit normal, n, for the sides of the element and incorporating it in with the calculation of the numerical flux, the discretization of the right-hand side is

$$c^2 \psi^T \left[L_1^T M S_{j1} f_{p1} + L_2^T M S_{j2} f_{p2} + L_3^T M S_{j3} f_{p3} + L_4^T M S_{j4} f_{p4} \right], \qquad (4.19)$$

where $f_p = n \cdot (\nabla p)^*$. As a reminder from Section 4.1, S_{j1}, S_{j2}, S_{j3}, and S_{j4} are the matrix of surface Jacobians for each side of an element, as labeled in Figure 4.1. This same notation is used for all four sides of the flux term, f_p, listed in (4.19). The numerical flux for two dimensions is discussed in greater detail in Section 4.3. Finally, we combine the two sides of (3.6) to form the final discretization for the second variational equation to be

$$J(M \otimes M)\frac{d\omega}{dt} + c^2 T p = c^2 \left(L_1^T M S_{j1} f_{p1} + L_2^T M S_{j2} f_{p2} + L_3^T M S_{j3} f_{p3} + L_4^T M S_{j4} f_{p4} \right). (4.20)$$

As a reminder, ψ does not appear in (4.20) because it must hold for all ψ, as discussed in Section 3.3.1.

4.2.3 Third Two-Dimensional Discretization Equation

The discretization process of (3.12) is similar to the discretization of (3.6) in the previous section. Starting with the left-hand side,

$$\int_{\Omega_j} \nabla \psi \nabla \left(\frac{\partial p}{\partial t} - \omega \right) dA,$$

we have eight terms from the product of the gradients

$$\begin{aligned}
\nabla \psi \nabla \left(\frac{\partial p}{\partial t} - \omega \right) = &\frac{\partial \xi}{\partial x} \frac{\partial}{\partial \xi} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \xi}{\partial x} \frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial x} \frac{\partial}{\partial \xi} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \eta}{\partial x} \frac{\partial \psi}{\partial \eta} \\
&+ \frac{\partial \eta}{\partial x} \frac{\partial}{\partial \eta} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \xi}{\partial x} \frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial x} \frac{\partial}{\partial \eta} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \eta}{\partial x} \frac{\partial \psi}{\partial \eta} \\
&+ \frac{\partial \xi}{\partial y} \frac{\partial}{\partial \xi} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \xi}{\partial y} \frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial y} \frac{\partial}{\partial \xi} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \eta}{\partial y} \frac{\partial \psi}{\partial \eta} \\
&+ \frac{\partial \eta}{\partial y} \frac{\partial}{\partial \eta} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \xi}{\partial y} \frac{\partial \psi}{\partial \xi} + \frac{\partial \xi}{\partial y} \frac{\partial}{\partial \eta} \left(\frac{\partial p}{\partial t} - \omega \right) \frac{\partial \eta}{\partial y} \frac{\partial \psi}{\partial \eta}.
\end{aligned} \qquad (4.21)$$

Focusing on the first term of (4.21), we find the discrete form to be

$$\int_{-1}^{+1} \int_{-1}^{+1} \left(\frac{\partial \xi}{\partial x} \right) \frac{\partial}{\partial \xi} \left(\frac{\partial p}{\partial t} - \omega \right) \left(\frac{\partial \xi}{\partial x} \right) \frac{\partial \psi}{\partial \xi} J d\xi d\eta$$

$$= \psi^T D_\xi^T \left[\frac{\partial y}{\partial \eta} J^{-1} (M \otimes M) \frac{\partial y}{\partial \eta} \right] D_\xi \left(\frac{dp}{dt} - \omega \right). \quad (4.22)$$

Expanding the discretization for all eight terms in (4.21) yields the full discrete form for the left-hand side of (3.12),

$$\psi^T T \left(\frac{dp}{dt} - \omega \right) = \psi^T T \frac{dp}{dt} - \psi^T T \omega, \quad (4.23)$$

where T is equal to Equation (4.15).

Let us focus on the right-hand side of (3.12),

$$\int_{\partial \Omega_j} (\nabla \psi \cdot n)(\omega^* - \omega) ds, \quad (4.24)$$

and discretize side one of a single element (see Figure 4.1). Conducting a change of variables yields

$$\int_{-1}^{+1} \nabla \psi \cdot n \, f_\omega S_j d\xi, \quad (4.25)$$

where the $f_\omega = (\omega^* - \omega)$. For notational purposes, allow

$$D_x = \left[\frac{\partial \xi}{\partial x} D_\xi + \frac{\partial \eta}{\partial x} D_\eta \right],$$

$$D_y = \left[\frac{\partial \xi}{\partial y} D_\xi + \frac{\partial \eta}{\partial y} D_\eta \right].$$

The discretization of (4.24) for side one becomes

$$\psi^T \left[(L_1 D_x)^T n_x + (L_1 D_y)^T n_y \right] M S_{j1} f_{\omega 1},$$

40

where the flux, $f_{\omega 1}$, is from side one of an element as annotated in Figure 4.1. A similar calculation for the remaining sides gives

$$\boldsymbol{\psi}^T \left(D_x^T L_1^T \boldsymbol{n}_x + D_y^T L_1^T \boldsymbol{n}_y\right) MS_{j1} f_{\omega 1} + \boldsymbol{\psi}^T \left(D_x^T L_2^T \boldsymbol{n}_x + D_y^T L_2^T \boldsymbol{n}_y\right) MS_{j2} f_{\omega 2}$$
$$+\boldsymbol{\psi}^T \left(D_x^T L_3^T \boldsymbol{n}_x + D_y^T L_3^T \boldsymbol{n}_y\right) MS_{j3} f_{\omega 3} + \boldsymbol{\psi}^T \left(D_x^T L_4^T \boldsymbol{n}_x + D_y^T L_4^T \boldsymbol{n}_y\right) MS_{j4} f_{\omega 4}. \tag{4.26}$$

Equation (4.26) is the discrete form for the right side of (3.12). Combining (4.26) and (4.23) produces the final discrete approximation for the third variational equation to be

$$T\frac{d\boldsymbol{p}}{dt} - T\boldsymbol{\omega} = \left(D_x^T L_1^T \boldsymbol{n}_x + D_y^T L_1^T \boldsymbol{n}_y\right) MS_{j1} f_{\omega 1} + \left(D_x^T L_2^T \boldsymbol{n}_x + D_y^T L_2^T \boldsymbol{n}_y\right) MS_{j2} f_{\omega 2}$$
$$+ \left(D_x^T L_3^T \boldsymbol{n}_x + D_y^T L_3^T \boldsymbol{n}_y\right) MS_{j3} f_{\omega 3} + \left(D_x^T L_4^T \boldsymbol{n}_x + D_y^T L_4^T \boldsymbol{n}_y\right) MS_{j4} f_{\omega 4}. \tag{4.27}$$

4.2.4 Combination of Two-Dimensional Discretization Equations

As in one dimension, we have three discrete equations and our goal is to get the problem into the form of two equations and two vector unknowns. By combining (4.10) and (4.27), we find

$$\left(T + \mathbb{1} J (M \otimes M)\right)\frac{d\boldsymbol{p}}{dt} = \left(T + \mathbb{1} J (M \otimes M)\right)\boldsymbol{\omega} + \left(D_x^T L_1^T \boldsymbol{n}_x + D_y^T L_1^T \boldsymbol{n}_y\right) MS_{j1} f_{\omega 1}$$
$$+ \left(D_x^T L_2^T \boldsymbol{n}_x + D_y^T L_2^T \boldsymbol{n}_y\right) MS_{j2} f_{\omega 2} + \left(D_x^T L_3^T \boldsymbol{n}_x + D_y^T L_3^T \boldsymbol{n}_y\right) MS_{j3} f_{\omega 3} \tag{4.28}$$
$$+ \left(D_x^T L_4^T \boldsymbol{n}_x + D_y^T L_4^T \boldsymbol{n}_y\right) MS_{j4} f_{\omega 4},$$

where $\mathbb{1}$ is a matrix of ones since (4.10) produces a scalar. Additionally, for the second equation, (4.20) can be re-arranged to isolate $\frac{d\boldsymbol{\omega}}{dt}$ for the final discrete equation of

$$J (M \otimes M)\frac{d\boldsymbol{\omega}}{dt} = c^2 \left(\left(L_1^T MS_{j1} f_{p1} + L_2^T MS_{j2} f_{p2} + L_3^T MS_{j3} f_{p3} + L_4^T MS_{j4} f_{p4}\right) - T\boldsymbol{p}\right). \tag{4.29}$$

Equations (4.28) and (4.29) are the two-dimensional ordinary differential equations that we solve numerically using the RK54 scheme [11].

4.3 Two-Dimensional Numerical Flux

The numerical flux for two dimensions is conceptually similar to the numerical flux for one dimension, except in two dimensions we have to account for more degrees of freedom, boundaries, and the element normals. In Section 3.5, we accounted for the discontinuity that existed between the two different approximations at the boundary point. In this section, we account for the discontinuity that exists between two different approximations at multiple boundary points along the surface of two neighboring elements. Thus, the numerical flux in two dimensions is an extension of the numerical flux from Section 3.5. Figure 4.2 is an example depiction of a two-dimensional flux boundary for a quadrilateral element.
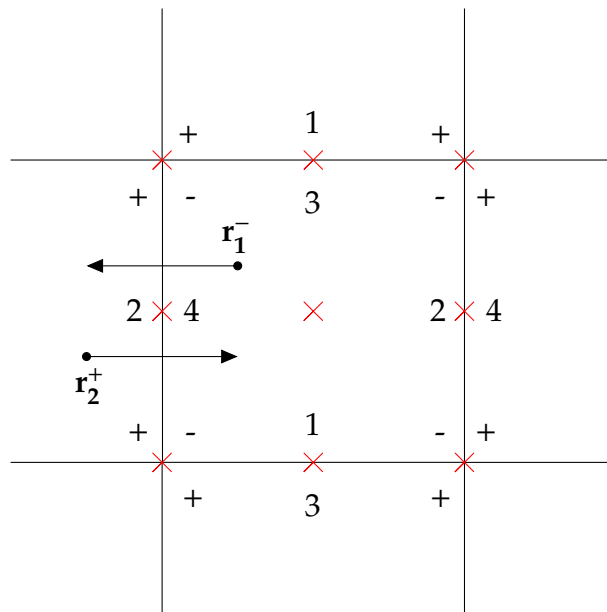

Figure 4.2: Example Two-Dimensional Flux Boundary on a Quadrilateral with $N = 2$.

In two dimensions, the element to be considered is the $(-)$ side and the neighboring element is the $(+)$ side. This is the same for the $(-)$ representing the element of focus's gridpoints and the $(+)$ representing the neighboring elements gridpoints.

42

In referencing Figure 4.2, the center element requires a flux to be computed with its neighbors at the boundary LGL points. Once again, Figure 4.2 is an example for discussion purposes; the actual elements are curvilinear quadrilaterals and the element normals at the LGL points are required in the numerical flux computation of $(n \cdot \nabla p)^*$. The formulation of the form of the central flux and upwind flux can be found in Section 3.5. The central flux equations for two dimensions are

$$\omega^* = \frac{1}{2}(\omega^- + \omega^+), \tag{4.30}$$

$$n \cdot (\nabla p)^* = \frac{1}{2}(n^- \cdot (\nabla p^-) - n^+ \cdot (\nabla p^+)), \tag{4.31}$$

where (4.30) element normals are represented in (4.27). The element normals in two dimensions are outward of each element. For example, n^- is the normal directed away from the $(-)$ boundary toward the neighboring $(+)$ element and n^+ is directed toward the element of focus (i.e., the center element in Figure 4.2). Typically, the central flux is an average of both sides of an element boundary, but the reader may notice that (4.31) has a minus instead of a plus sign. The reason for this is that $n^+ = -n^-$, and thus the minus sign actually produces an average. As a reminder, the numerical flux is incorporated in the discretization equations by $f_p = n \cdot (\nabla p)^*$ from Section 4.2.2 and $f_\omega = (\omega^* - \omega)$ from Section 4.2.3. Similar to one dimension, the central flux is easy to understand and produces a stable algorithm, but it does not take into account the physical propagation of information like the upwind flux.

The upwind flux equations for two dimensions are

$$\omega^* = \frac{1}{2}(\omega^- + \omega^+) - \frac{c}{2}(n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-)), \tag{4.32}$$

$$n \cdot (\nabla p)^* = \frac{1}{2}(n^- \cdot (\nabla p^-) - n^+ \cdot (\nabla p^+)) + \frac{1}{2c}(\omega^+ - \omega^-). \tag{4.33}$$

Equations (4.32) and (4.33) are formulated by the decoupling of the wave equation into two one-way advection equations at the boundary points through the characteristic variables. This is the same concept from Section 3.5, but in two dimensions, there are three characteristic variables. However, the third characteristic variable is associated with a zero eigenvalue and does not propagate information across

element boundaries. As depicted in Figure 4.2, r_1^- and r_2^+ are example characteristic variables propagating information across one elemental boundary.

4.4   Two-Dimensional Discontinuous Galerkin Results

As a test problem we consider a washer domain with curved elements as shown in Figure 4.3. Along the inner and outer radius of the washer, we impose the free surface boundary conditions of $p = 0$. We use the exact solution of

$$p(x, y, t) = \sin\left(nt - \beta\theta\right) J_\beta\left(r(x, y)\right), \tag{4.34}$$

$$\frac{\partial p}{\partial t} = \omega(x, y, t) = n\cos\left(nt - \beta\theta\right) J_\beta\left(r(x, y)\right), \tag{4.35}$$

with the conversion equations

$$r(x, y) = \sqrt{x^2 + y^2},$$

$$\theta(x, y) = \cos^{-1}\left(\frac{x}{r(x, y)}\right)$$

for mapping between polar and Cartesian coordinates. Here, J_β is a Bessel function of the first kind with parameter β and n is an integer; we use $\beta = 4$ and $n = 1$. To ensure (4.34) is equal to zero at the inner and out radius for all time, we enforced the inner and outer radius of the washer grid to be the second and fourth roots of the Bessel function. For interested readers, the DG implementation for two dimensions can be found in Appendix C.

We initially tested the implementation on a square grid of quadrilateral elements with periodic boundary conditions. After the successful implementation on a square grid, we built and tested the algorithm on a washer grid with curved elements using the exact solution of (4.34) and (4.35) evaluated at $t = 0$ as the initial condition. To enforce the free surface boundary condition of $p = 0$, we used

$$\omega^+ = -\omega^-, \quad (\boldsymbol{n}^+ \cdot (\nabla p^+)) = -(\boldsymbol{n}^- \cdot (\nabla p^-))$$

in the numerical flux routine for the points along the inner and outer radius of the
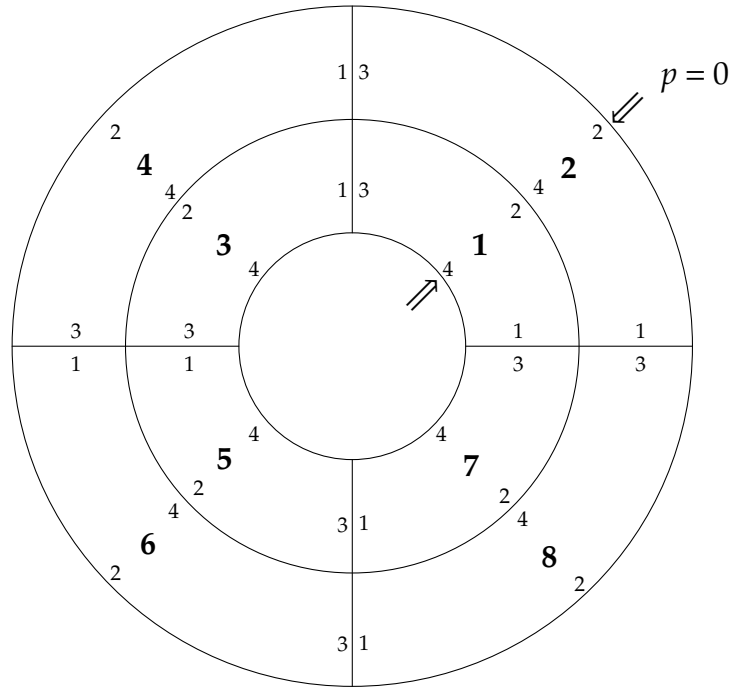
Figure 4.3: Example Two-Dimensional Washer Grid: Ne = 8

washer. Once again, a visual depiction of an example washer grid with $Ne = 8$ semi-curved elements is seen in Figure 4.3. Initially, testing the entire washer grid became too computationally expensive with the large number of elements. To counter this problem and still maintain integrity of the results, we evaluated one quarter of the washer by enforcing $\frac{\pi}{2}$ periodic boundary conditions for the left and right boundaries through the bessel function. By letting β be any multiple of four, we enforced the $\frac{\pi}{2}$ periodic conditions. For example, in referencing Figure 4.3, this means boundaries 1 and 3 from elements 1 and 2 are equal.

Table 4.1 lists the different polynomial orders and elements tested using the initial condition with (4.28) and (4.29) and evaluated through a RK54 iterative method. Figure 4.4 shows the log of the error vs. log of the number of elements for $N = 2, 4, 6, 8$ calculated using the global L^2 error norms. As expected for both ω and p, Figure 4.4 displays increasing convergence rates as the polynomial order increases. Once again, the higher the order of the local approximation, the faster the convergence rates are due to the the global error being dependent on the

45

Table 4.1: Discontinuous Galerkin Tested Information

Polynomial Orders (N)	Number of Elements (Ne)
$N = 2, 4, 6, 8$	$Ne = 4, 16, 64, 256$
$N = 10, 12$	$Ne = 4, 16, 64$
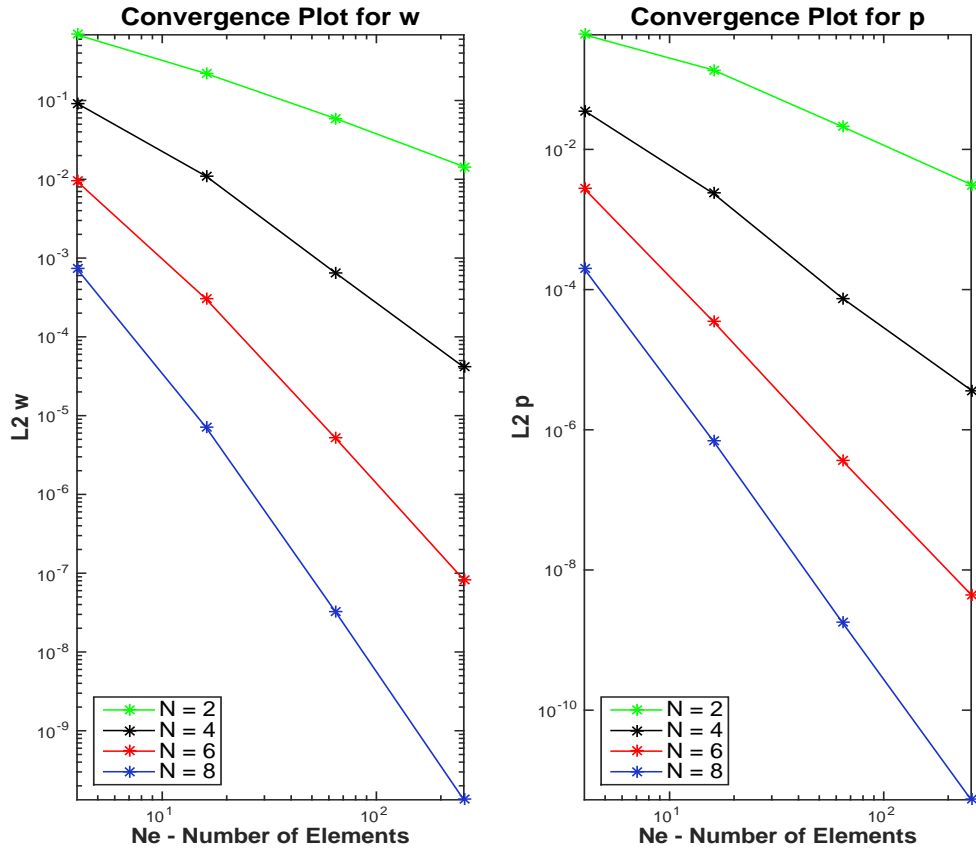
Figure 4.4: Convergence Rates For N = 2, 4, 6, 8

polynomial order [1] (as discussed in Chapter 3 with (3.32)).

Table 4.2 displays the convergence rates corresponding to Figure 4.4. As seen, the convergence rates increase with increasing polynomial order. Using the same space of functions for ω and p, the two-dimensional results yielded convergence rates being near its associated polynomial order, with ω being near N and p being $N + \frac{1}{2}$ or $N + 1$. Once again, most DG methods are expected to be of the order N,

Table 4.2: Convergence Rates for N = 2, 4, 6, 8

Convergence Rates			
N/Ne	Ne = 4 to 16	Ne = 16 to 64	Ne = 64 to 256
N = 2 (ω)	1.9240	2.0570	2.1274
N = 2 (p)	1.9663	2.9243	2.8680
N = 4 (ω)	3.3438	4.2448	4.0822
N = 4 (p)	4.2739	5.1808	4.4615
N = 6 (ω)	5.2521	6.0193	6.0944
N = 6 (p)	6.6670	6.7889	6.4519
N = 8 (ω)	6.9754	7.9190	8.0543
N = 8 (p)	8.5157	8.7873	8.5069

$N + \frac{1}{2}$, or $N + 1$ [1]. These results are different from the one-dimensional convergence rates, where ω is N and p is $N + 2$. Understanding this difference requires further analysis and is an area for future research.

Table 4.3: Convergence Rate for N = 10, 12

Convergence Rates		
N/Ne	Ne = 4 to 16	Ne = 16 to 64
N = 10 (ω)	8.7628	10.0287
N = 10 (p)	10.2080	10.6994
N = 12 (ω)	10.6378	12.0772
N = 12 (p)	12.0461	11.8315

To use higher-order polynomials, we decreased the number of elements in order to avoid reaching machine precision with the first datapoint. Figure 4.5 shows the error and Table 4.3 gives the convergence rates for $N = 10$ and $N = 12$. The convergence rates are still on the order of the polynomial, but the $N = 12$ case seems to be reaching machine precision. Both Figure 4.5 and Table 4.3 display how higher order polynomials can be used to approximate the solution with a much higher convergence rate.

Using higher order polynomials, especially on complex geometries with curved elements, comes with a computational cost with respect to time for approximating the solution. Higher orders are more accurate, as seen in Figure 4.5, but the
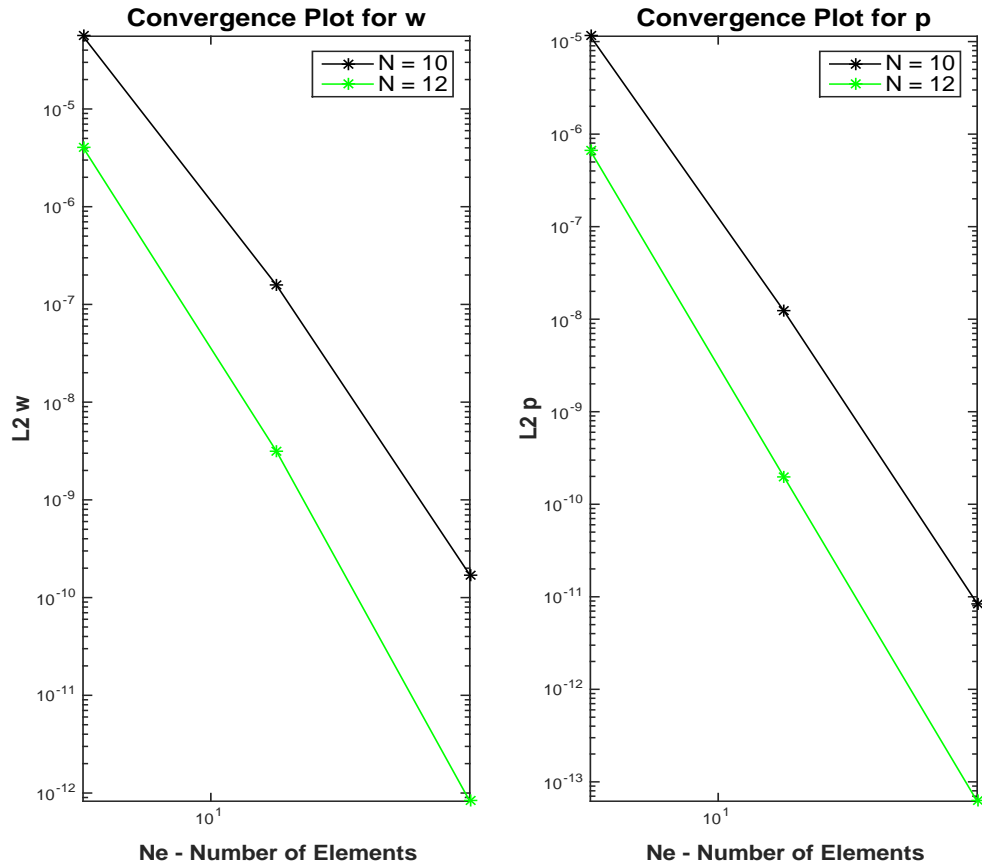
Figure 4.5: Convergence Rates For N = 10, 12

computational time increases since the matrices to be inverted are larger with higher polynomial orders and require a smaller time-step. In the code we use a time-step of $\triangle t \approx \frac{h}{N^2}$.

CHAPTER 5:
Energy Conservation

Proving stability of a method can be accomplished through an energy analysis. The following energy analysis is similar to the method employed by Appelö and Hagstrom [2]. By conducting the energy analysis on an element with neighboring elements, we are able to find out if energy is dissipated or conserved throughout the system. If energy is dissipated, then we know that all of the eigenvalues of the system have a negative real part and thus the ODE is stable. If energy (E) is conserved, then $\frac{dE}{dt} = 0$ and all the eigenvalues are purely imaginary. If E_k is the energy on an element k, then the total energy is

$$E = \sum_{k=1}^{Ne} E_k, \tag{5.1}$$

and we want to show

$$\frac{dE}{dt} = \sum_{k=1}^{Ne} \frac{dE_k}{dt} \leq 0 \tag{5.2}$$

in order to prove that energy is dissipated throughout the system as time progresses.

5.1 Basic Theory of Energy Conservation

Let us look at the continuous energy equation on the domain $x \in [a,b]$ for one dimension:

$$E = \int_a^b \frac{1}{2\lambda}\omega^2 + \frac{1}{2\rho}\left(\frac{\partial p}{\partial x}\right)^2 dx, \tag{5.3}$$

with $p_t = \omega$, $\omega_t = c^2 p_{xx}$, and $c^2 = \frac{\lambda}{\rho}$. We assume c, λ, and ρ are constants on the domain. As a reminder, for clarification purposes, $\frac{\partial p}{\partial t} = p_t$ for a short notation form and this follows through for the remaining terms in Section 5.1. The first portion of (5.3) is the kinetic energy and the second portion is the potential energy for the

system. By taking the time derivative of (5.3)

$$E_t = \int_a^b \frac{1}{\lambda}\omega_t\omega + \frac{1}{\rho}p_{xt}p_x dx, \qquad (5.4)$$

we can now begin to manipulate (5.4) to show (5.2). Specifically, we want to manipulate (5.4) in order to select our boundary conditions to ensure $E_t \leq 0$. Let us substitute $\omega_t = c^2 p_{xx}$ and $\omega = p_t$ into (5.4)

$$E_t = \int_a^b \frac{1}{\lambda}c^2 p_{xx}p_t + \frac{1}{\rho}p_{xt}p_x dx. \qquad (5.5)$$

With $c^2 = \frac{\lambda}{\rho}$, it follows that (5.5) becomes

$$E_t = \frac{1}{\rho}\int_a^b p_{xx}p_t + p_{xt}p_x dx, \qquad (5.6)$$

and by conducting integration by parts on the last portion of (5.6) we find the following:

$$E_t = \frac{1}{\rho}\left[\int_a^b p_{xx}p_t dx + p_x p_t \big|_a^b - \int_a^b p_{xx}p_t dx\right]. \qquad (5.7)$$

The two remaining integrals cancel and we have that

$$E_t = \frac{1}{\rho}p_x p_t \big|_a^b = \frac{1}{\rho}\left[p_x\omega \big|_b - p_x\omega \big|_a\right], \qquad (5.8)$$

where we can now choose the boundary conditions to ensure $E_t \leq 0$. For example, with periodic boundary conditions, it should be obvious that (5.8) equals zero when $a = b$ and energy is conserved.

5.2   Two-Dimensional Energy Analysis

Section 5.1 is a relatively simple example of the analysis of energy for the continuous one-dimensional system. Moving into two dimensions requires more tedious analysis and also depends on the conditions set within the system. For example,

50

when using an upwind flux, we want to show that energy is dissipated and when using a central flux, we want to show that energy is conserved. Let us initially investigate a general two-dimensional case before moving into the flux analysis. The energy equation for two dimensions is

$$E_k = \int_{\Omega_j} \frac{1}{2\lambda}\omega^2 + \frac{1}{2\rho}\left(p_x^2 + p_y^2\right)dA, \tag{5.9}$$

where

$$\frac{\partial p}{\partial x} = \frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \xi} + \frac{\partial \eta}{\partial x}\frac{\partial p}{\partial \eta},$$

and

$$\left(\frac{\partial p}{\partial x}\right)^2 = \frac{\partial p}{\partial \xi}\left(\frac{\partial \xi}{\partial x}\right)^2\frac{\partial p}{\partial \xi} + \frac{\partial p}{\partial \xi}\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x}\frac{\partial p}{\partial \eta} + \frac{\partial p}{\partial \eta}\frac{\partial \eta}{\partial x}\frac{\partial \xi}{\partial x}\frac{\partial p}{\partial \xi} + \frac{\partial p}{\partial \eta}\left(\frac{\partial \eta}{\partial x}\right)^2\frac{\partial p}{\partial \eta}.$$

Moreover, $\left(\frac{\partial p}{\partial y}\right)^2$ is formulated through the same method as $\left(\frac{\partial p}{\partial x}\right)^2$, except with respect to y. The change of variables of (5.9) yields

$$\begin{aligned}
E_k = \int_{-1}^{+1}\int_{-1}^{+1} &\frac{1}{2\lambda}\omega J\omega + \frac{1}{2\rho}\left[\frac{\partial p}{\partial \xi}J\left[\left(\frac{\partial \xi}{\partial x}\right)^2 + \left(\frac{\partial \xi}{\partial y}\right)^2\right]\frac{\partial p}{\partial \xi}\right.\\
&+ \frac{\partial p}{\partial \xi}J\left[\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y}\right]\frac{\partial p}{\partial \eta} + \frac{\partial p}{\partial \eta}J\left[\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y}\right]\frac{\partial p}{\partial \xi}\\
&\left.+ \frac{\partial p}{\partial \eta}J\left[\left(\frac{\partial \xi}{\partial x}\right)^2 + \left(\frac{\partial \xi}{\partial y}\right)^2\right]\frac{\partial p}{\partial \eta}\right]d\xi d\eta
\end{aligned} \tag{5.10}$$

and the discretization of (5.10) is

$$E_k = \frac{1}{2\lambda}\omega^T J(M\otimes M)\omega + \frac{1}{2\rho}p^T Tp, \tag{5.11}$$

51

where T is the same simplifying variable used in Equation (4.14) from Section 4.2.2. Taking the time derivative of (5.11) we have

$$\frac{dE_k}{dt} = \frac{1}{2\lambda}\left(\frac{d\boldsymbol{\omega}}{dt}\right)^T J(\boldsymbol{M}\otimes\boldsymbol{M})\,\boldsymbol{\omega} + \frac{1}{2\lambda}\boldsymbol{\omega}^T J(\boldsymbol{M}\otimes\boldsymbol{M})\left(\frac{d\boldsymbol{\omega}}{dt}\right)$$
$$+ \frac{1}{2\rho}\left(\frac{d\boldsymbol{p}}{dt}\right)^T \boldsymbol{T}\boldsymbol{p} + \frac{1}{2\rho}\boldsymbol{p}^T \boldsymbol{T}\left(\frac{d\boldsymbol{p}}{dt}\right),$$

which can be simplified to

$$\frac{dE_k}{dt} = \frac{1}{\lambda}\boldsymbol{\omega}^T J(\boldsymbol{M}\otimes\boldsymbol{M})\left(\frac{d\boldsymbol{\omega}}{dt}\right) + \frac{1}{\rho}\boldsymbol{p}^T \boldsymbol{T}\left(\frac{d\boldsymbol{p}}{dt}\right). \tag{5.12}$$

Upon inspection of (5.12), we can substitute (4.27) and (4.29) in for $J(\boldsymbol{M}\otimes\boldsymbol{M})\left(\frac{d\boldsymbol{\omega}}{dt}\right)$ and $\boldsymbol{T}\left(\frac{d\boldsymbol{p}}{dt}\right)$ from Chapter 4 to produce

$$\frac{dE_k}{dt} = \frac{c^2}{\lambda}\boldsymbol{\omega}^T\left[\left(\boldsymbol{L}_1^T \boldsymbol{M} \boldsymbol{S}_{j1} f_{p1} + \boldsymbol{L}_2^T \boldsymbol{M} \boldsymbol{S}_{j2} f_{p2} + \boldsymbol{L}_3^T \boldsymbol{M} \boldsymbol{S}_{j3} f_{p3} + \boldsymbol{L}_4^T \boldsymbol{M} \boldsymbol{S}_{j4} f_{p4}\right) - \boldsymbol{T}\boldsymbol{p}\right]$$
$$+ \frac{1}{\rho}\boldsymbol{p}^T\left[\boldsymbol{T}\boldsymbol{\omega} + \left(\boldsymbol{D}_x^T \boldsymbol{L}_1^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_1^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j1} f_{\omega 1} + \left(\boldsymbol{D}_x^T \boldsymbol{L}_2^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_2^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j2} f_{\omega 2} \tag{5.13}$$
$$+ \left(\boldsymbol{D}_x^T \boldsymbol{L}_3^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_3^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j3} f_{\omega 3} + \left(\boldsymbol{D}_x^T \boldsymbol{L}_4^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_4^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j4} f_{\omega 4}\right].$$

For notational purposes, let

$$\boldsymbol{F}_p = \left(\boldsymbol{L}_1^T \boldsymbol{M} \boldsymbol{S}_{j1} f_{p1} + \boldsymbol{L}_2^T \boldsymbol{M} \boldsymbol{S}_{j2} f_{p2} + \boldsymbol{L}_3^T \boldsymbol{M} \boldsymbol{S}_{j3} f_{p3} + \boldsymbol{L}_4^T \boldsymbol{M} \boldsymbol{S}_{j4} f_{p4}\right),$$
$$\boldsymbol{F}_\omega = \left(\boldsymbol{D}_x^T \boldsymbol{L}_1^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_1^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j1} f_{\omega 1} + \left(\boldsymbol{D}_x^T \boldsymbol{L}_2^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_2^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j2} f_{\omega 2}$$
$$+ \left(\boldsymbol{D}_x^T \boldsymbol{L}_3^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_3^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j3} f_{\omega 3} + \left(\boldsymbol{D}_x^T \boldsymbol{L}_4^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_4^T \boldsymbol{n}_y\right)\boldsymbol{M} \boldsymbol{S}_{j4} f_{\omega 4}.$$

Therefore, Equation (5.13), with the substitution of $\frac{c^2}{\lambda} = \frac{1}{\rho}$, becomes

$$\frac{dE_k}{dt} = \frac{1}{\rho}\boldsymbol{\omega}^T\left(\boldsymbol{F}_p - \boldsymbol{T}\boldsymbol{p}\right) + \frac{1}{\rho}\boldsymbol{p}^T\left(\boldsymbol{T}\boldsymbol{\omega} + \boldsymbol{F}_\omega\right), \tag{5.14}$$

and since $\boldsymbol{\omega}^T \boldsymbol{T} \boldsymbol{p} = \boldsymbol{p}^T \boldsymbol{T} \boldsymbol{\omega}$, (5.14) is further simplified to

$$\frac{dE_k}{dt} = \frac{1}{\rho}\left(\boldsymbol{\omega}^T \boldsymbol{F}_p + \boldsymbol{p}^T \boldsymbol{F}_\omega\right). \tag{5.15}$$

As you can see, (5.15) is dependent on the flux and boundary information from each element, as included in $\boldsymbol{F}_p$ and $\boldsymbol{F}_\omega$. Knowing that

$$\boldsymbol{\omega}_\ell^T = \boldsymbol{\omega}^T \boldsymbol{L}_\ell^T,$$

$$\left(\boldsymbol{n}_\ell \cdot \left(\nabla p_\ell\right)\right)^T = \boldsymbol{p}^T \left(\boldsymbol{D}_x^T \boldsymbol{L}_\ell^T \boldsymbol{n}_x + \boldsymbol{D}_y^T \boldsymbol{L}_\ell^T \boldsymbol{n}_y\right),$$

for $\ell = 1,2,3,4$, (5.15) becomes

$$\begin{aligned}\frac{dE_k}{dt} = \frac{1}{\rho}\Big[&\boldsymbol{\omega}_1^T \boldsymbol{M} \boldsymbol{S}_{j1} \boldsymbol{f}_{p1} + \boldsymbol{\omega}_2^T \boldsymbol{M} \boldsymbol{S}_{j2} \boldsymbol{f}_{p2} + \boldsymbol{\omega}_3^T \boldsymbol{M} \boldsymbol{S}_{j3} \boldsymbol{f}_{p3} + \boldsymbol{\omega}_4^T \boldsymbol{M} \boldsymbol{S}_{j4} \boldsymbol{f}_{p4} \\ &+ \left(\boldsymbol{n}_1 \cdot \left(\nabla p_1\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j1} \boldsymbol{f}_{\omega 1} + \left(\boldsymbol{n}_2 \cdot \left(\nabla p_2\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j2} \boldsymbol{f}_{\omega 2} \\ &+ \left(\boldsymbol{n}_3 \cdot \left(\nabla p_3\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j3} \boldsymbol{f}_{\omega 3} + \left(\boldsymbol{n}_4 \cdot \left(\nabla p_4\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j4} \boldsymbol{f}_{\omega 4}\Big]. \end{aligned} \tag{5.16}$$

Recall that $f_p = \boldsymbol{n} \cdot (\nabla p)^*$ and $f_\omega = (\omega^* - \omega)$ from Section 4.2.2 and Section 4.2.3, thus we can consolidate the boundary information for each side of an element

$$\begin{aligned}\frac{dE_k}{dt} = \frac{1}{\rho}\Big[&\left(\boldsymbol{\omega}_1^T \boldsymbol{M} \boldsymbol{S}_{j1}\left(\boldsymbol{n}_1 \cdot \left(\nabla p_1\right)\right)^* + \left(\boldsymbol{n}_1 \cdot \left(\nabla p_1\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j1}\left(\omega^* - \omega_1\right)\right) \\ &+ \left(\boldsymbol{\omega}_2^T \boldsymbol{M} \boldsymbol{S}_{j2}\left(\boldsymbol{n}_2 \cdot \left(\nabla p_2\right)\right)^* + \left(\boldsymbol{n}_2 \cdot \left(\nabla p_2\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j2}\left(\omega^* - \omega_2\right)\right) \\ &+ \left(\boldsymbol{\omega}_3^T \boldsymbol{M} \boldsymbol{S}_{j3}\left(\boldsymbol{n}_3 \cdot \left(\nabla p_3\right)\right)^* + \left(\boldsymbol{n}_3 \cdot \left(\nabla p_3\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j3}\left(\omega^* - \omega_3\right)\right) \\ &+ \left(\boldsymbol{\omega}_4^T \boldsymbol{M} \boldsymbol{S}_{j4}\left(\boldsymbol{n}_4 \cdot \left(\nabla p_4\right)\right)^* + \left(\boldsymbol{n}_4 \cdot \left(\nabla p_4\right)\right)^T \boldsymbol{M} \boldsymbol{S}_{j4}\left(\omega^* - \omega_4\right)\right)\Big]. \end{aligned} \tag{5.17}$$

Equation (5.17) is in the final form that we use to conduct the stability analysis. Summing (5.17) for all elements, we can analyze the energy for the whole domain. In what follows, we prove stability with the boundary conditions imposed from Chapter 4 for the central flux and the upwind flux.

5.2.1 Central Flux Analysis

In order to simplify the analysis, we focus on a single side of an element with its adjacent neighboring element. This concept is depicted in Figure 3.2 in Chapter 3. For notation purposes, the $(-)$ side is the element of focus and the $(+)$ side is the neighboring element. Additionally, ω^- represents the solution at the degrees of freedom on the associated side of the quadrilateral element as depicted in Figure 4.2. This same notation applies for (5.19) and (5.20). The equations for the central flux for a generic side without boundary conditions are

$$\omega^* = \frac{1}{2}(\omega^- + \omega^+), \tag{5.18}$$

$$n^- \cdot (\nabla p^*) = \frac{1}{2}[n^- \cdot (\nabla p^-) - n^+ \cdot (\nabla p^+)], \tag{5.19}$$

$$n^+ \cdot (\nabla p^*) = \frac{1}{2}[n^+ \cdot (\nabla p^+) - n^- \cdot (\nabla p^-)]. \tag{5.20}$$

Since we are working with one side of a quadrilateral element, we add together the terms from one edge of the element with the terms from the neighbor's corresponding edge

$$\frac{dE_k^{\pm}}{dt} = \left[(\omega^+)^T MS_j(n^+ \cdot (\nabla p)^*) + (n^+ \cdot (\nabla p^+))^T MS_j(\omega^* - \omega^+)\right] \\ + \left[(\omega^-)^T MS_j(n^- \cdot (\nabla p)^*) + (n^- \cdot (\nabla p^-))^T MS_j(\omega^* - \omega^-)\right]. \tag{5.21}$$

Substituting in (5.18), (5.19), and (5.20) for the flux portions in (5.21) yields

$$\frac{dE_k^{\pm}}{dt} = \frac{1}{2}\left[(\omega^+)^T MS_j(n^+ \cdot (\nabla p^+)) - (n^+ \cdot (\nabla p^+))^T MS_j\omega^+ + (n^+ \cdot (\nabla p^+))^T MS_j\omega^- \right. \\ - (\omega^-)^T MS_j(n^+ \cdot (\nabla p^+)) + (\omega^-)^T MS_j(n^- \cdot (\nabla p^-)) - (n^- \cdot (\nabla p^-))^T MS_j\omega^- \\ \left. + (n^- \cdot (\nabla p^-))^T MS_j\omega^+ - (\omega^+)^T MS_j(n^- \cdot (\nabla p^-))\right]. \tag{5.22}$$

Inspecting (5.22), all of the terms cancel out to show $\frac{dE_k^{\pm}}{dt} = 0$ for one element boundary. This can be expanded to all four boundaries of the quadrilateral. Thus, energy is conserved and the ODE is stable when using the central flux.

5.2.2 Upwind Flux Analysis

As discussed at the end of Section 3.5, the upwind flux consists of the central flux plus an upwinding portion. This is evident in (3.28) and (3.29). Therefore, since we know that energy is conserved (i.e., $\frac{dE_k^{\pm}}{dt} = 0$) for the central flux portion, we only investigate the upwinding portion. The equations for the upwinding section are

$$\omega_u^* = -\frac{c}{2}[n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-)], \tag{5.23}$$

$$(n^+ \cdot (\nabla p)^*)_u = \frac{1}{2c}(\omega^- - \omega^+), \tag{5.24}$$

$$(n^- \cdot (\nabla p)^*)_u = \frac{1}{2c}(\omega^+ - \omega^-), \tag{5.25}$$

which is substituted into the elemental boundary equation (5.26) for the upwinding portion,

$$\begin{aligned}
\left(\frac{dE_k^{\pm}}{dt}\right)_u &= \left[(\omega^+)^T MS_j(n^+ \cdot (\nabla p)^*)_u + (n^+ \cdot (\nabla p^+))^T MS_j \omega_u^*\right] \\
&\quad + \left[(\omega^-)^T MS_j(n^- \cdot (\nabla p)^*)_u + (n^- \cdot (\nabla p^-))^T MS_j \omega_u^*\right].
\end{aligned} \tag{5.26}$$

The substitution of (5.23), (5.24), and (5.25) into (5.26) yields

$$\begin{aligned}
\left(\frac{dE_k^{\pm}}{dt}\right)_u &= \frac{1}{2c}\left[(\omega^+)^T MS_j(\omega^- - \omega^+) + (\omega^-)^T MS_j(\omega^+ - \omega^-)\right] \\
&\quad - \frac{c}{2}\left[(n^+ \cdot (\nabla p^+))^T MS_j[n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-)]\right. \\
&\quad \left. + (n^- \cdot (\nabla p^-))^T MS_j[n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-)]\right].
\end{aligned} \tag{5.27}$$

Factoring out a negative from the first portion of (5.27) makes $(\omega^- - \omega^+)$ become $(\omega^+ - \omega^-)$. Therefore, (5.27) can be re-written to

$$\begin{aligned}
\left(\frac{dE_k^{\pm}}{dt}\right)_u &= -\frac{1}{2c}\left[(\omega^+ - \omega^-)^T MS_j(\omega^+ - \omega^-)\right] \\
&\quad - \frac{c}{2}\left[(n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-))^T MS_j(n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-))\right].
\end{aligned} \tag{5.28}$$

Inspection of (5.28) displays two terms:

$$\left[(\omega^+ - \omega^-)^T MS_j (\omega^+ - \omega^-)\right],$$

$$\left[(n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-))^T MS_j (n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-))\right]$$

that are both positive definite. MS_j in both sections is always positive; therefore, regardless if $(\omega^+ - \omega^-)$ or $(n^+ \cdot (\nabla p^+) + n^- \cdot (\nabla p^-))$ produce negative or positive values, the square of each is always be positive. Furthermore, with the negative in front of both sections of (5.28), $\left(\frac{dE_k^{\pm}}{dt}\right)_u < 0$. The combination of $\left(\frac{dE_k^{\pm}}{dt}\right)_u$ with $\frac{dE_k^{\pm}}{dt}$ from the central flux section (Section 5.2.1) yields a negative value per face. The summation of all the elements in the domain shows Equation (5.2) to be true. Therefore, this method is stable for both the central flux and upwind flux on complex geometries with curved elements.

5.2.3   Boundary Condition Analysis

In Chapter 4, we tested the method on a washer mesh with curved elements and the boundary conditions of $p = 0$ on the inner and outer radius of the washer. In order to implement the boundary conditions, the following constraints were set at the inner and outer boundaries of the washer

$$\omega^+ = -\omega^-, \quad (n^+ \cdot (\nabla p^+)) = -(n^- \cdot (\nabla p^-)). \tag{5.29}$$

Additionally, we are only concerned with the $(-)$ portion of (5.21) for the central flux and the $(-)$ portion of (5.26) for the upwind flux since there are no neighboring elements $((+)$ portions) at the inner and outer radius of the washer.

Boundary Conditions with Central Flux

Once again, substituting in the central flux Equations (5.18) and (5.19) for the $(-)$ portion of (5.21) yields

$$\frac{dE_k^-}{dt} = (\omega^-)^T MS_j \left(\frac{1}{2} [n^- \cdot (\nabla p^-) - n^+ \cdot (\nabla p^+)]\right) + (n^- \cdot (\nabla p^-))^T MS_j \left(\frac{1}{2}(\omega^+ - \omega^-)\right),$$

which becomes

$$\frac{dE_k^-}{dt} = (\boldsymbol{\omega}^-)^T MS_j\left(\frac{1}{2}[2(\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-))]\right) + (\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-))^T MS_j\left(\frac{1}{2}(-2\boldsymbol{\omega}^-)\right) \qquad (5.30)$$

after the substitution of the boundary conditions listed in (5.29). Simplifying (5.30) produces

$$\frac{dE_k^-}{dt} = (\boldsymbol{\omega}^-)^T MS_j(\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-)) - (\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-))^T MS_j(\boldsymbol{\omega}^-) = 0, \qquad (5.31)$$

and energy is conserved with the boundary conditions of $p = 0$ using a central flux.

Boundary Conditions with Upwind Flux

For the upwind flux boundary condition analysis, we use the same process as above, except we use (5.23) and (5.25) for substitution into the $(-)$ portion of (5.26). This yields

$$\left(\frac{dE_k^-}{dt}\right)_u = (\boldsymbol{\omega}^-)^T MS_j\left(\frac{1}{2c}(\boldsymbol{\omega}^+ - \boldsymbol{\omega}^-)\right) + (\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-))^T MS_j\left(-\frac{c}{2}[\boldsymbol{n}^+ \cdot (\nabla \boldsymbol{p}^+) + \boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-)]\right)$$

which becomes

$$\left(\frac{dE_k^-}{dt}\right)_u = (\boldsymbol{\omega}^-)^T MS_j\left(\frac{1}{2c}(-2\boldsymbol{\omega}^-)\right) + (\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-))^T MS_j\left(-\frac{c}{2}[-\boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-) + \boldsymbol{n}^- \cdot (\nabla \boldsymbol{p}^-)]\right)$$

and is simplified to

$$\left(\frac{dE_k^-}{dt}\right)_u = -\frac{1}{c}\left[(\boldsymbol{\omega}^-)^T MS_j\boldsymbol{\omega}^-\right] < 0. \qquad (5.32)$$

Once again, the combination of (5.31) and (5.32) produces a negative result for the upwind flux boundary condition analysis.

The global energy dissipation rate is the sum of the energy dissipation rates at all of the faces. Therefore, for both the central and upwind fluxes, the method discussed in Chapter 4 is stable.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6:
Conclusion

The complexities of solving PDEs through Finite Difference Methods, Finite Volume Methods, and Finite Element Methods is and will continue to be an area of ongoing research. In this paper, we explored a new Discontinuous Galerkin method for approximating a second order wave equation with curved elements in complex geometries. The beginning two chapters discuss the tools needed for constructing the DG method, while Chapter 3 and Chapter 4 tested the method in one and two dimensions. Chapter 5 proved energy stability of the method through an energy analysis.

In Chapter 3, convergence rates on a one-dimensional grid for both low and high polynomial orders yielded rates near the associated polynomial order. The results yielded convergence rates with ω being on the order of N and p being on the order of $N+2$, which is higher than most DG methods that yield results on the order of N, $N+\frac{1}{2}$, and $N+1$. Additionally, using higher polynomial orders required fewer elements for a given error level.

In two dimensions, we tested the problem on a washer grid with curved elements. In this case, the inner and outer radius had the free surface boundary condition of $p = 0$ implemented through the numerical flux routine. The two-dimensional results yielded convergence rates different than the one-dimensional method with ω and p being near its associated polynomial order N or $N+\frac{1}{2}$. Similar to the one-dimensional results, testing high polynomial orders achieved machine precision at a faster rate with less elements.

Chapter 5 explored the energy dissipation of the method, with and without the imposed boundary conditions of $p = 0$, for the central flux and upwind flux. In all cases, the method either conserved energy ($\frac{dE}{dt} = 0$) or dissipated energy ($\frac{dE}{dt} < 0$). Therefore, the eigenvalues of the system have a negative real part proving that the ODE is stable and the method is viable.

6.1 Future Work

Future research into this Discontinuous Galerkin method can be conducted on a variety of topics. These topics include, but are not limited to, the following:

- Accuracy: As discussed in Chapter 3 and Chapter 4, the convergence rates listed are observations; proving these results could lead to further insight and improvement.
- Efficiency: Is it possible to template the curvilinear elements so that the mass matrix is the same for all elements by modifying the approximation space for each element? The benefit of doing this is the ability to store one mass matrix for all elements [12, 13].
- Coupling: Development of the numerical coupling procedures for elastic–acoustic interfaces within this second order form [14].
- Form: Relationship between this second order formulation and the standard first order formulation of the acoustic wave equation.
- Extension: Consider this method for other equations, such as the Einstein equations governing black hole dynamics where there are 10 equations in the second order form versus 50 equations in the first order form [15].

APPENDIX A:
Interpolation and Integration

The following are the main MATLAB codes for Interpolation and Integration from Chapter 2.

A.1 Interpolation

```matlab
%-----------------------------------------------------------------------%
%This is the main driver for Interpolation.
%Written by Benjamin Davis
%             Department of Applied Mathematics
%             Naval Postgraduate School
%             Monterey, CA 93943-5216
%
%Synopsis: Conducting Interpolation using LGL points.
%-----------------------------------------------------------------------%
%Interpolate a known function f(x) using legendre-gauss-labatto points.
clear
%Input Nth order interpolation (N) and number of evaluation points (k)
N = 40;
k = 50;
%Initialization
errL1num = zeros(N,1);
errL1den = zeros(N,1);
errL1 = zeros(N,1);
errL2num = zeros(N,1);
errL2den = zeros(N,1);
errL2 = zeros(N,1);
errinfabsn = zeros(k,1);
errinfabsd = zeros(k,1);
errinf = zeros(N,1);

for n = 1:N
    %Setting up Grids
    x = legendre_gauss_lobatto(n+1);
```

```matlab
29      z = linspace(-1,1,50);
       %Set up data for Lagrange Li(Xk)
31      f = exp(-4*x.^2);
        %Constuct Lagrange
33      [L,dL] = lagrange_basis(x,z);
       %Evaluation
35      Pn = f * L;
       %Error Analysis
37      fex = exp(-4*z.^2);
       for i = 1:k
39          errL1num(n) = abs(Pn(i)-fex(i)) + errL1num(n);
            errL1den(n) = abs(fex(i)) + errL1den(n);
41          errL2num(n) = (Pn(i)-fex(i))^2 + errL2num(n);
            errL2den(n) = (fex(i))^2 + errL2den(n);
43          errinfabsn(i) = abs(Pn(i)-fex(i));
            errinfabsd(i) = abs(fex(i));
45      end

47      errL1(n) = errL1num(n)/errL1den(n);
       errL2(n) = sqrt(errL2num(n)/errL2den(n));
49      errinf(n) = max(errinfabsn)/max(errinfabsd);
end
51 % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   %       1       2       3       4       5       6       7               8
53 c = [1 0 0;0 0 0;0 0 0; 0 1 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 0 0; 0.5 0.3 0.2];
55 %       9      10      11      12      13      14      15      16
   % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
57      %plot Pn vs. Actual Function
        hold on
59      plot(x,f,'*')
        title('Lagendre Gauss Lobatto Approximation')
61      xlabel('x')
        ylabel('f(x)')
63      axis([-1 1 -0.1 1])
        legend('N = 2', 'N = 4','N = 8', 'N = 16', 'Exact')

65
       %Plot Error Norms
67      figure
```

```matlab
      N = 1:40;
      semilogy(N,errL1 ,N,errL2 ,N, errinf )
      title('Legendre Gauss Lobatto Interpolation Error')
      xlabel('N')
      ylabel('Error Norm')
      legend('L1 Norm' ,'L2 Norm' ,'Inf Norm')
```

A.2 Integration

```matlab
%------------------------------------------------------------------------%
%This is the main driver for Integration .
%Written by Benjamin Davis
%             Department of Applied Mathematics
%             Naval Postgraduate School
%             Monterey, CA 93943-5216
%
%Synopsis: Conducting Integration using LGL points .
%------------------------------------------------------------------------%
%Integration : Using the interpolation functions , sampling the
%basis functions at LGL and LG integration points to perform the Gauss
%quadrature of the given equation from the project set .
clear

%Initialization
N = 19;

for n = 1:N
    %Quadratrue points and weights
    [x,w] = legendre_gauss_lobatto(n+1);
    %Given Equation for evaluation
    f = exp(-4*x.^2);
    %Constuct Lagrange matrix and differentiation matrix
    [L,dL] = lagrange_basis(x,x);
    %Evaluation
    Pn = (f*L)* w';
    %Evaluation of Error
```

```matlab
      exact = (sqrt(pi)/2)*erf(2);
29    errL1(n) = abs(Pn - exact)/abs(exact);
      errL2(n) = sqrt((Pn - exact)^2/(exact)^2);
31 end
      %Plot Error
33    N = 1:19;
      semilogy(N, errL2)
35    title('Legendre Gauss Lobatto Integration Error')
      xlabel('N')
37    ylabel('Error')
      legend('L2 Norm')
```

APPENDIX B:
One-Dimensional Discontinuous Galerkin

The following are the main MATLAB codes for the one-dimensional problem from Chapter 3.

```matlab
%------------------------------------------------------------------%
%This is the Driver function using a Discontinuous Galerkin method for
%approximating a 2nd Order Acoustic Wave equation in one dimension
%with an Upwind Flux.
%
%Written by Benjamin Davis Created: October 2014
%              Department of Applied Mathematics
%              Naval Postgraduate School
%              Monterey, CA 93943-5216
%
%Synopsis: Discontinuous Galerkin Method for wave equation in second
%order form using inexact integration an Upwind flux.  The outputs
%are currently four plots: Convergence rates for w and P and plots
%of the numerical and exact solutions.
%------------------------------------------------------------------%
clear
%Initial Inputs
N = 4;   %Polynomial Order
n = 3;
dtscale = 1/4;
c = 1;
z = 1;
t_final = 0.58;
ngl = N+1;

for Ne =   2.^(1:4)
    fprintf('Number of Elements %4d with polynomial order %2d\n',Ne,N)
    Np = Ne*(ngl);
    %Interpolation and Integration Points
    [psx,w] = legendre_gauss_lobatto(ngl);
```

```matlab
        %Constuct Lagrange matrix and differentiation matrix
32      [L,dL] = lagrange_basis(psx,psx);
        %Construct Mass and Differentiation Matrices
34      [M,D] = mass_diff1D(L,dL,w);
        %Construct Global Mass and Differentiation Matrices
36      [coord,intma] = create_grid(ngl,Ne,psx);
        dx = coord(N+1)-coord(1);
38      M = M.*dx/2;
        D = M\D;
40      Dh = D';

42      grad2 = Dh*M*D;
        one = ones(ngl,ngl);
44      onesM = one*M;

46      M1 = grad2 + onesM;

48      e0 = sparse(1,1,1,ngl,1);
        en = sparse(ngl,1,1,ngl,1);
50
        %Building the Initial Condition
52      for e = 1:Ne
            for i = 1:N+1
54              I = intma(e,i);
                x = coord(I);
56              q0(e,i,1) = n*pi*sin(n*pi*x);  %dp/dt=w at t=0
                q0(e,i,2) = 0; %P at t=0
58          end
        end
60      q1 = q0(:,:,:);
        %Time Step Calculation
62      dt = dtscale*(1/Ne)/(N^2);
        Nt = round(t_final/dt);
64      dt = t_final/Nt;
        %Periodic Boundary Conditions
66      [sidep] = sideper(Ne);
        %Facemap
68      [fmp] = facemap(Ne);
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Compute RK Time-Integration Coefficients
kstages = 4; %=1 is RK1; =2 is RK2; = 3 is RK3; and = 4 is RK4
if (kstages < 4)
  [a0,a1,beta] = compute_ti_coefficients(kstages);

  for k = 1:Nt
      for a = 1:kstages
              [wf,dpf] = upwindflux(Ne,q1,D,ngl,sidep);
              [R] = RHSDG(en,e0,Dh,D,M,q1,Ne,M1,wf,fmp,dpf,c);


              qm=a0(a)*q0 + a1(a)*q1 + dt*beta(a)*R;
              q1 = qm;
      end
      q0 = qm;
  end
elseif (kstages == 4)

  a = [0,1/2,1/2,1];
  b = [1/6,1/3,1/3,1/6];
  R = 0;
  for k = 1:Nt
    qm = q0;
    for s = 1:4
        qs = q0+dt*a(s)*R;

        [wf,dpf] = upwindflux(Ne,qs,D,ngl,sidep);
        [R] = RHSDG(en,e0,Dh,D,M,qs,Ne,M1,wf,fmp,dpf,c);

        qm = qm + dt*b(s)*R;
    end
    q0 = qm;
  end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Plotting Purposes
for e = 1:Ne
        for i = 1:N+1
```

```matlab
                I = intma(e,i);
110             x = coord(I);
                qex(e,i,1) = n*pi*sin(n*pi*x)*cos(n*pi*t_final);
112             qex(e,i,2) = sin(n*pi*x)*sin(n*pi*t_final);
            end
114     end
    %Plotting purposes
116     m=1;
    for e = 1:Ne
118         for i = 1:N+1
                w(m) = qm(e,i,1);
120             wexact(m) = qex(e,i,1);

122             p(m) = qm(e,i,2);
                pexact(m) = qex(e,i,2);
124
                I = intma(e,i);
126             xp(m) = coord(I);
                m=m+1;
128         end
    end
130     %Building the Error Plot Information
    L2errw = 0;
132     L2errp = 0;

134     for elm = 1:Ne
      pnts = (elm-1)*(N+1) + (1:(N+1));
136     dw    = w(pnts)-wexact(pnts);
      dp    = p(pnts)-pexact(pnts);
138     L2errw = dw*M*dw'+L2errw;
      L2errp = dp*M*dp'+L2errp;
140     end

142     L2err(1,z) = Np;
    L2err(2,z) = sqrt(L2errw);
144     L2err(3,z) = sqrt(L2errp);
    L2err(4,z) = Ne;
146     z = z+1;
end
```

```matlab
148 %      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %       1       2       3       4       5       6       7          8
150 c = [1 0 0;0 1 0;0 0 0; 0 0 0; 0 0 0; 0 0 1; 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 0 0; 0.5 0.3 0.2];
152 %       9       10      11      12      13      14      15         16
    % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
154 %%%%%%%%%%%%%%%W Convergence Plot%%%%%%%%%%%%%%%%%%%%%%%%%%%
    figure(1)
156 subplot(1,2,1); %hold on
       loglog(L2err(4,:),L2err(2,:),'*-','color',c(N,:))
158    title('Convergence Plot for w')
       xlabel('Ne - Number of Elements')
160    ylabel('L2 w')
       axis tight
162    legend('N = 4', 'N = 6','N = 8','N = 16')
    %%%%%%%%%%%%%%%P Convergence Plot%%%%%%%%%%%%%%%%%%%%%%%%%%%
164 subplot(1,2,2); %hold on
       loglog(L2err(4,:),L2err(3,:),'*-','color',c(N,:))
166    title('Convergence Plot for p')
       xlabel('Ne - Number of Elements')
168    ylabel('L2 p')
       axis tight
170    legend('N = 4', 'N = 6', 'N = 8','N = 16')
    % %%%%%%%%%%%%%%%Plot Numerical Solution P(x,t)%%%%%%%%%%%%
172  figure(2)
     plot(xp,p,xp,pexact,'*')
174  title('Numerical Solution for p(x,t)')
     xlabel('x')
176  ylabel('t')
     legend('Numerical','Exact')
178 % %%%%%%%%%%%%%%%Plot Numerical Solution w(x,t)%%%%%%%%%%%%
     figure(3)
180  plot(xp,w,xp,wexact,'*')
     title('Numerical Solution for w(x,t)')
182  xlabel('x')
     ylabel('t')
184 legend('Numerical','Exact')

186 rate_w = (log(L2err(2,1:end-1))-log(L2err(2,2:end))) ./ ...
```

```matlab
               (log(L2err(1,2:end))-log(L2err(1,1:end-1)));
rate_p = (log(L2err(3,1:end-1))-log(L2err(3,2:end))) ./ ...
               (log(L2err(1,2:end))-log(L2err(1,1:end-1)));

fprintf('Convergence rate for w: ')
disp(rate_w)

fprintf('Convergence rate for p: ')
disp(rate_p)
```

```matlab
%-----------------------------------------------------------------------%
%Function for the Right Hand Side of Discretized equations for
%one dimension DG from Chapter 3.
%Written by Benjamin Davis Created: November 2014
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%-----------------------------------------------------------------------%
function [R] = RHSDG(en,e0,Dh,D,M,qn,Ne,M1,wf,fmp,dpf,c)
    for e = 1:Ne
        R(e,:,2) = M1*qn(e,:,1)';
        R(e,:,2) = R(e,:,2)' + Dh*(en*wf(fmp(2,e))-e0*wf(fmp(1,e)));
        R(e,:,2) = R(e,:,2)' - Dh*(en*en'*qn(e,:,1)'-e0*e0'*qn(e,:,1)')
    ;
        R(e,:,1) = -c^2*Dh*M*D*qn(e,:,2)';
        R(e,:,1) = R(e,:,1)' + c^2*(en*dpf(fmp(2,e))-e0*dpf(fmp(1,e)));
    end
    R1 = M1\R(:,:,2)';
    R1 = R1';
    R(:,:,2) = R1;
    R2 = M\R(:,:,1)';
    R2 = R2';
    R(:,:,1) = R2;
end
```

```matlab
%-------------------------------------------------------------------------%
%Function for the Center Flux for the one-dimensional DG problem from
%Chapter 3
%Written by Benjamin Davis Created: October 2014
%              Department of Applied Mathematics
%              Naval Postgraduate School
%              Monterey, CA 93943-5216
%-------------------------------------------------------------------------%
function [wf,dpf] = centerflux(Ne,qn,D,ngl,sidep)
    for s = 1:Ne
        Ls = sidep(1,s);
        Rs = sidep(2,s);

        qLw = qn(Ls,ngl,1);
        qRw = qn(Rs,1,1);

        dpL = D*qn(Ls,:,2)';
        dpL = dpL(ngl);
        dpR = D*qn(Rs,:,2)';
        dpR = dpR(1);

        wf(s,:)  = (1/2)*(qLw + qRw);
        dpf(s,:) = (1/2)*(dpL + dpR);
    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Upwind Flux function for one dimension DG. Used in one dimension driver
%for 2nd Order Acoustic wave equation.
%Written by Benjamin Davis Created: October 2014
%              Department of Applied Mathematics
%              Naval Postgraduate School
%              Monterey, CA 93943-5216
%-------------------------------------------------------------------------%
function [wf,dpf] = upwindflux(Ne,qn,D,ngl,sidep)
```

```matlab
c = 1;
    for s = 1:Ne
         Ls = sidep(1,s);
         Rs = sidep(2,s);

         qLw = qn(Ls,ngl,1);
         qRw = qn(Rs,1,1);

         dpL = D*qn(Ls,:,2)';
         dpL = dpL(ngl);
         dpR = D*qn(Rs,:,2)';
         dpR = dpR(1);

         wf(s,:) = (1/2)*(qLw + qRw) + c/2*(dpR - dpL);
         dpf(s,:) = (1/2)*(dpL + dpR) + 1/(2*c)*(qRw - qLw);
    end
end
```

```matlab
%------------------------------------------------------------------------%
%Code given to Professor's F.X. Giraldo MA4245 Class July 2014
%Used by Ben Davis
%This code computes the Legendre-Gauss-Lobatto points and weights
%which are the roots of the Lobatto Polynomials.
%Written by F.X. Giraldo on 4/2000
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%------------------------------------------------------------------------%
function [xgl,wgl] = legendre_gauss_lobatto(P)
p=P-1;
ph=floor( (p+1)/2 );

for i=1:ph
   x=cos( (2*i-1)*pi/(2*p+1) );
   for k=1:20
       [L0,L0_1,L0_2]=legendre_poly(p,x);
```

```matlab
        %Get new Newton Iteration
        dx=-(1-x^2)*L0_1/(-2*x*L0_1 + (1-x^2)*L0_2);
        x=x+dx;
        if (abs(dx) < 1.0e-20)
            break
        end
    end
    xgl(p+2-i)=x;
    wgl(p+2-i)=2/(p*(p+1)*L0^2);
end
%Check for Zero Root
if (p+1 ~= 2*ph)
    x=0;
    [L0,L0_1,L0_2]=legendre_poly(p,x);
    xgl(ph+1)=x;
    wgl(ph+1)=2/(p*(p+1)*L0^2);
end
%Find remainder of roots via symmetry
for i=1:ph
    xgl(i)=-xgl(p+2-i);
    wgl(i)=+wgl(p+2-i);
end
```

```matlab
%-----------------------------------------------------------------------%
%Function for building the Lagrange Polynomials.
%Written by Benjamin Davis in MA4245 Created: July 2014 in MA4245
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%-----------------------------------------------------------------------%
function [L,dL] = lagrange_basis(x,z)
%Nth order interpolation
n = length(x);
%Length of the equally spaced grid for k = 1:50
h = length(z);
```

```matlab
%Initialize the Lagrange Matrix
L = ones(n,h);
dL = zeros(n,h);
%Computation for Lagrange Matrix
    for k = 1:h
        for i = 1:n
            for j = 1:n
                dl = 1;
                if j ~= i % If j does not equal i
                    %Equation for the Lagrange Polynomial
                    L(i,k) = (z(k)-x(j))./(x(i)-x(j)) * L(i,k);
                    for l = 1:n
                        if (l ~= i) && (l ~= j)
                            dl = dl*(z(k)-x(l))./(x(i)-x(l));
                        end
                    end
                    dL(i,k) = dL(i,k) + dl/(x(i)-x(j));
                end
            end
        end
    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Code given by Professor F.X. Giraldo to MA4245 class.
%Used by Ben Davis
%This code computes the Legendre Polynomials and its 1st and 2nd
%derivatives
%Written by F.X. Giraldo on 4/2000
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%
%This code was written by Professor Giraldo and given to his MA4245
%Galerkin Methods class in July 2014.
%-------------------------------------------------------------------------%
```

```matlab
14  function [L0,L0_1,L0_2] = legendre_poly(p,x)

16  L1=0;L1_1=0;L1_2=0;
    L0=1;L0_1=0;L0_2=0;

18
    for i=1:p
20      L2=L1;L2_1=L1_1;L2_2=L1_2;
        L1=L0;L1_1=L0_1;L1_2=L0_2;
22      a=(2*i-1)/i;
        b=(i-1)/i;
24      L0=a*x*L1 - b*L2;
        L0_1=a*(L1 + x*L1_1) - b*L2_1;
26      L0_2=a*(2*L1_1 + x*L1_2) - b*L2_2;
    end
```

```matlab
1  %------------------------------------------------------------------%
   %Function for building the Mass and Differentiation matrices for
3  %One-Dimension.  Used in Thesis 1D Upwind code.
   %Written by Benjamin Davis Created: July 2014
5  %           Department of Applied Mathematics
   %           Naval Postgraduate School
7  %           Monterey, CA 93943-5216
   %------------------------------------------------------------------%
9  function [M,D] = mass_diff1D(L,dL,w)
   n = length(L(:,1));
11 M = zeros(n,n);
   D = zeros(n,n);

13
       for i = 1:n
15         for j = 1:n
               M(i,j) = ((L(i,:).*L(j,:))*w(1,:)');
17             D(i,j) = ((L(i,:).*dL(j,:))*w(1,:)');
           end
19     end
   end
```

```matlab
%------------------------------------------------------------------------%
%Code given to Professor's F.X. Giraldo's MA4245 class
%Used by Ben Davis
%This function computes the LGL grid and elements.
%Written by F.X. Giraldo on 10/2003
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%------------------------------------------------------------------------%
function [coord, intma] = create_grid(ngl, nelem, xgl)
%Set some constants
xmin=-1;
xmax=+1;
dx=(xmax-xmin)/nelem;
%Generate Grid Points
ip=1;
coord(1)=xmin;
for i=1:nelem
    x0=xmin + (i-1)*dx;
    intma(i,1)=ip;
    for j=2:ngl
        ip=ip + 1;
        coord(ip)=( xgl(j)+1 )*dx/2 + x0;
        intma(i,j)=ip;
    end
end
```

```matlab
%------------------------------------------------------------------------%
% Function for implementing periodic boundary conditions for Thesis 1D
% Upwind code.
%Written by Benjamin Davis Created: November 2014
```

```matlab
%                Department of Applied Mathematics
%                Naval Postgraduate School
%                Monterey, CA 93943-5216
%-----------------------------------------------------------------------%
function [sidep] = sideper(Ne)
    sidep = zeros(2,Ne);
    for e = 1:Ne
        sidep(1,e) = e-1;
        sidep(2,e) = e;
        if e==1
            sidep(1,e) = Ne;
        end
    end
end
```

```matlab
%-----------------------------------------------------------------------%
%Facemaping periodic function for 1-Dimension and used in Thesis 1D
%Upwind code.
%Written by Benjamin Davis Created: November 2014
%                Department of Applied Mathematics
%                Naval Postgraduate School
%                Monterey, CA 93943-5216
%-----------------------------------------------------------------------%
function [fmp] = facemap(Ne)
    fmp = zeros(2,Ne);
    for e = 1:Ne
        fmp(1,e) = e;
        fmp(2,e) = e+1;
        if e==Ne
            fmp(2,e) = fmp(1,1);
        end
    end
end
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:
Two-Dimensional Discontinuous Galerkin

The following are the main MATLAB codes for the two-dimensional problem from Chapter 4.

```matlab
%---------------------------------------------------------------------%
%This is the main driver for approximating the 2D Acoustic Wave
%Equation with an upwind flux on a washer grid. The washer grid
%can be scaled to various sizes based off of user input.
%Written by Benjamin Davis and Asst. Professor Jeremy Kozdon
%               Department of Applied Mathematics
%               Naval Postgraduate School
%               Monterey, CA 93943-5216
%Synopsis: Discontinuous Galerkin Method for wave equation in second
%order form using inexact integration and an upwind flux.  The outputs
%are currently five plots: Convergence rates for w and P and plots of
%the numerical and exact solutions.
%---------------------------------------------------------------------%
clear

N = 2;
n = 1;
c = 1;
t_final = pi/2;
%Define beta to be zero and will run with no theta dependence.
beta = 4;
% skew_mesh = 0  :: rectangular mesh
% skew_mesh = 1  :: skew element (straight sided)
% skew_mesh = 2  :: skew element (curved elements)
skew_mesh = 2;
%Define anonymous functions for the solution
r_ex = @(x,y) sqrt(x^2 + y^2);
theta = @(x,y) acos(x/r_ex(x,y));
g_ex = @(r_ex) besselj(beta,r_ex);
f_ex = @(x,y,t) sin(c*t-beta*theta(x,y));
```

```matlab
   P_ex = @(x,y,t)  f_ex(x,y,t)*g_ex(r_ex(x,y)); %p(x,y,t)
32 w_ex = @(x,y,t)  c*cos(c*t-beta*theta(x,y))*g_ex(r_ex(x,y)); %dp/dt
   %Select inner and outer radius of washer.
34 %Found these values using fzero command on Bessel function.
   r1 = fzero(g_ex,6);
36 r2 = fzero(g_ex,14);
   disp([r1,r2]);

   %Full Washer.
40 %rn is what 2*pi will be divided by to change the size of the mesh.
   %For example, if you use 4, you will get pi/2 or 1/4 of the washer.
42 rn = 4;

44 %Scaling the washer.
   %rn = ceil((2*pi*r2)/(r2-r1));

   %For storing information, Don't Change
48 z=1;
   for nel = 2.^(1:3)

       nq = N+1; %Number of integration/quadrature points
52     ngl = N+1; %Number of interpolation points in one direction
       nelx = nel;
54     nely = nel;

56     %Interpolation and Integration Points
       [psx,w] = legendre_gauss_lobatto(N+1);
58     %Create Grid
       [coord,intma,bsido,iperiodic,Np,Ne,nboun,nface] = create_grid_2d(
       nelx,nely,N,psx,plot_grid,skew_mesh);

       fprintf('Number of Elements %4d with polynomial order %2d\n',Ne,N)

       %Create Sides/Edge Information for DG
64     [iside,jeside] = create_side(intma,bsido,Np,Ne,nboun,nface,ngl);
       [face,imapl,imapr] = create_face(iside,intma,nface,ngl);
66     s_face = face;
       %Changing face code to enforce new BC.
68     [face] = faceBC(nface,face);
```

```matlab
        %Will not be periodic with face from faceBC.  If you want periodic,
        %comment out faceBC function.
        face = create_face_periodicity(iside,face,coord,nface,nboun);
        %Building the washer.
        [Rad] = radius(coord,r1,r2);
        [theta] = thetapolar(coord,rn);
        [coord] = newcoord(Rad,theta,coord);
        %Construct Lagrange Basis and Jacobian Matrix
        [L,dL] = lagrange_basis(psx,psx);
        [ksi_x,ksi_y,eta_x,eta_y,x_ksi,x_eta,y_ksi,y_eta,jac] = metrics2(
        coord,intma,L,dL,Ne,ngl,nq);
        %Building and Element to Element function.
        %EtoEfunctBC incorporates boundary conditions.
        EtoEBC = EtoEfunctBC(nface,face,Ne);
        %Constuct M, D, D_ksi, D_eta Matrices
        [M,D] = mass_diff2D(L,dL,w);
        M_1D = M;
        M = kron(M,M);

        D_eta = kron(D,eye(ngl));
        D_ksi = kron(eye(ngl),D);

        %Construct L1,L2,L3,L4
        e0 = sparse(1,1,1,ngl,1);
        en = sparse(ngl,1,1,ngl,1);

        L1 = kron(e0',eye(ngl));
        L2 = kron(eye(ngl),en');
        L3 = kron(en',eye(ngl));
        L4 = kron(eye(ngl),e0');
        %Building Matrix Terms and new Jacobian for solving the RHS for
        equation 5, 6 and 7.
        [MAT,MAT_ksi,MAT_eta,Je,A,B,C,D,E,F,G,H] = MatrixTerms2D(Ne,D_ksi,
        D_eta,y_eta,y_ksi,x_eta,x_ksi,jac,M);
        %Building Dx and Dy for RHS
        [Dx,Dy] = DxDy(Ne,ksi_x,ksi_y,eta_x,eta_y,D_ksi,D_eta);
        %Building the surface jacobians for the faces
        [Sj1,Sj2,Sj3,Sj4] = SurfaceJac2D(Ne,L1,L2,L3,L4,x_ksi,x_eta,y_ksi,
```

```matlab
        y_eta ) ;
        %Compute the Normals per element
        [nx_1,nx_2,nx_3,nx_4,ny_1,ny_2,ny_3,ny_4] = ElementNormals2D(Ne,L1,
        L2,L3,L4,x_ksi,x_eta,y_ksi,y_eta,Sj1,Sj2,Sj3,Sj4);
        %Big Matrix of Ones
        M1 = ones(ngl*ngl);
        %Building Initial Condition
        for e = 1:Ne
            for i = 1:ngl
                for j = 1:ngl
                    I = intma(e,i,j);
                    x = coord(I,1);
                    y = coord(I,2);
                    q0{1}(e,i,j) = w_ex(x,y,0); %dp/dt = w at t=0
                    q0{2}(e,i,j) = P_ex(x,y,0);  %P(x,y,t) at t=0;

                end
            end
        end
        q1 = q0;
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Time Step Calculation
        dt = (1/2)*((1/Ne)/(N^2));
        Nt = round(t_final/dt);
        dt = t_final/Nt;
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Beginning of the RK54 Time integrator
        a = [ 0.0,
              -567301805773.0/ 1357537059087.0,
              -2404267990393.0/ 2016746695238.0,
              -3550918686646.0/ 2091501179385.0,
              -1275806237668.0/ 842570457699.0];

        b = [1432997174477.0/ 9575080441755.0,
              5161836677717.0/ 13612068292357.0,
              1720146321549.0/ 2090206949498.0,
              3134564353537.0/ 4481467310338.0,
```

```matlab
                 2277821191437.0/  14882151754819.0];


        R{1} = 0;
        R{2} = 0;


        for  k = 1:Nt
            qm = q0;
            for  s = 1:5
                R{1} = a(s)*R{1};
                R{2} = a(s)*R{2};


                %Building  all  the  flux  information  with  Boundary  Conditions
                [fw1,fw2,fw3,fw4,fp1,fp2,fp3,fp4] = UpwindFlux2DFwFpBC(Ne,qm
    , L1,L2,L3,L4,EtoEBC,Dx,Dy,nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4);
                %Solving  Right  Hand  Side
                [R1] = RHSDG2Dn(Ne,MAT,Je ,M,M_1D,M1,qm,Dx,Dy,L1,L2,L3,L4,
    nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4,Sj1,Sj2,Sj3,Sj4,fw1,fw2,fw3,
    fw4,fp1,fp2,fp3,fp4,c,MAT_ksi,MAT_eta);


                R{1} = R1{1} + R{1};
                R{2} = R1{2} + R{2};


                qm{1} = qm{1} + dt*b(s)*R{1};
                qm{2} = qm{2} + dt*b(s)*R{2};
            end
            q0 = qm;
        end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%For  plotting  purposes
        %Solving  the  exact  solution  at  final  time .
        for  e = 1:Ne
            for  i = 1:ngl
                for  j = 1:ngl
                    I = intma(e,i,j);
                    x = coord(I,1);
                    y = coord(I,2);
                    qex{1}(e,i,j) = w_ex(x,y,t_final);
                    qex{2}(e,i,j) = P_ex(x,y,t_final);
```

```matlab
                  end
            end
        end
        %Plotting the exact solution vs. numerical solution.
        m=1;
        for e = 1:Ne
            for i = 1:ngl
                for j = 1:ngl

                    w(m) = qm{1}(e,i,j);
                    wexact(m) = qex{1}(e,i,j);

                    p(m) = qm{2}(e,i,j);
                    pexact(m) = qex{2}(e,i,j);

                    I = intma(e,i,j);
                    xp(m) = coord(I);
                    m=m+1;
                end
            end
        end
        %Building the Error Plot Information
        L2errw = 0;
        L2errp = 0;

        for elm = 1:Ne
            pnts = (elm-1)*(N+1)^2 + (1:(N+1)^2);
            dw   = w(pnts)-wexact(pnts);
            dp   = p(pnts)-pexact(pnts);
            L2errw = dw*M*Je{elm}*dw'+L2errw;
            L2errp = dp*M*Je{elm}*dp'+L2errp;
        end

        L2err(1,z) = sqrt(Np);
        L2err(2,z) = sqrt(L2errw);
        L2err(3,z) = sqrt(L2errp);
        z = z+1;
        %Convergence Rates
        rate_w = (log(L2err(2,1:end-1))-log(L2err(2,2:end))) ./ (log(L2err
```

```matlab
            (1,2:end))-log(L2err(1,1:end-1)));
        rate_p = (log(L2err(3,1:end-1))-log(L2err(3,2:end))) ./ (log(L2err
            (1,2:end))-log(L2err(1,1:end-1)));

        fprintf('Convergence rate for w: ')
        disp(rate_w)

        fprintf('Convergence rate for p: ')
        disp(rate_p)
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        1         2        3       4       5       6        7          8
c = [1 0 0;0 1 0;0 0 0; 0 0 0; 0 0 0; 0 0 0; 0 0 0; 0.1 0.2 0.8;...
       0 0 0;0 0 0;0 0 0; 0 0 0; 0 0 0; 0 0 0; 0 0 0; 0.5 0.3 0.2];
%        9        10        11      12      13      14       15         16
figure(1);
%%%%%%%%%%%%%%%%W Convergence Plot%%%%%%%%%%%%%%%%%%
subplot(1,2,1); %hold on
    loglog(L2err(1,:),L2err(2,:),'*-','color',c(N,:))
    title('Convergence Plot for w')
    xlabel('Np')
    ylabel('L2 w')
    axis tight
%%%%%%%%%%%%%%%%P Convergence Plot%%%%%%%%%%%%%%%%%%
subplot(1,2,2);% hold on
    loglog(L2err(1,:),L2err(3,:),'*-','color',c(N,:))
    title('Convergence Plot for p')
    xlabel('Np')
    ylabel('L2 p')
    axis tight
%Plot Numerical Solution P(x,y,t)
figure(2);
title('Numerical Solution for P')
xlabel('x')
ylabel('y')
zlabel('t')
plot3(coord(intma,1), coord(intma,2), qm{2}(:),'*')
```

```matlab
254 %Plot Exact Solution of P(x,y,t)
    figure(3);
256 title('Exact Solution for P')
    plot3(coord(intma,1), coord(intma,2), qex{2}(:),'*')
258 %Plot Numerical Solution w
    figure(4);
260 title('Numerical Solution for w')
    % xlabel('x')
262 % ylabel('y')
    % zlabel('t')
264 plot3(coord(intma,1), coord(intma,2), qm{1}(:),'*')
    %Plot Exact Solution of w
266 figure(5);
    title('Exact Solution for w')
268 plot3(coord(intma,1), coord(intma,2), qex{1}(:),'*')
```

```matlab
    %----------------------------------------------------------------------%
 2  %This is the main driver for approximating the 2D Acoustic Wave
    %Equation with Upwind Flux on a Square Grid. The grid can be rotated
 4  %various degrees and skewed based on user input.
    %Written by Benjamin Davis and Asst. Professor Jeremy Kozdon
 6  %              Department of Applied Mathematics
    %              Naval Postgraduate School
 8  %              Monterey, CA 93943-5216
    %%Synopsis: Discontinuous Galerkin Method for wave equation in
10  %second order form using inexact integration and upwind flux.  The
    %outputs are currently four plots: Convergence rates for w and P and
12  %plots of the numerical and exact solutions.
    %----------------------------------------------------------------------%
14  clear

16  N = 6;
    n = 1;
18  c = 1;
    t_final = .25;
20
```

```
   % skew_mesh = 0  ::  rectangular mesh
22 % skew_mesh = 1  ::  skew element ( straight sided )
   % skew_mesh = 2  ::  skew element ( curved elements )
24 skew_mesh = 2;

26 %Rotation Angle
   grid_rotation_angle =0; %CCW rotation in degrees
28 %For information storage . Don' t change .
   z=1;
30 for nel = 2.^(1:3)
       nq = N+1;  %Number of integration / quadrature points
32     ngl = N+1; %Number of interpolation points in one direction of an
       element
       nelx = nel ;
34     nely = nel ;

36     plot_grid = 0; %1 will display the grid , 0 will not display the grid

38     %Interpolation and Integration Points
       [psx ,w] = legendre_gauss_lobatto (N+1);
40     %Create Grid
       [coord , intma , bsido , iperiodic ,Np,Ne, nboun , nface ] = create_grid_2d (
       nelx , nely ,N,psx , plot_grid , skew_mesh );

42
       fprintf ( 'Number of Elements %4d with polynomial order %2d\n ' ,Ne,N)
44     %Rotate Grid
       [coord_rotated ] = rotate_grid_v2 (coord , intma ,Np,Ne, ngl , plot_grid ,
       grid_rotation_angle );
46     %Store Rotated COORDS
       coord=coord_rotated ;
48     %Construct Lagrange Basis and Jacobian Matrix
       [L,dL] = lagrange_basis (psx , psx );
50     [ksi_x , ksi_y , eta_x , eta_y , x_ksi , x_eta , y_ksi , y_eta , jac ] = metrics2 (
       coord , intma ,L,dL,Ne, ngl ,nq);
       %Create Sides /Edge Information for DG
52     [iside , jeside ] = create_side (intma , bsido ,Np,Ne, nboun , nface , ngl );
       [face , imapl , imapr ] = create_face (iside , intma , nface , ngl );
54     face = create_face_periodicity (iside , face , coord , nface , nboun );
       %Building and Element to Element function
```

```matlab
56          EtoE = EtoEfunct(nface,face,Ne);
         %Constuct M, D, D_ksi, D_eta Matrices
58          [M,D] = mass_diff2D(L,dL,w);
         M_1D = M;
60          M = kron(M,M);

62          D_eta = kron(D,eye(ngl));
         D_ksi = kron(eye(ngl),D);
64          %Construct L1,L2,L3,L4
         e0 = sparse(1,1,1,ngl,1);
66          en = sparse(ngl,1,1,ngl,1);

68          L1 = kron(e0',eye(ngl));
         L2 = kron(eye(ngl),en');
70          L3 = kron(en',eye(ngl));
         L4 = kron(eye(ngl),e0');
72          %Building Matrix Terms and new Jacobian for solving the RHS for
         equation 5, 6 and 7.
         [MAT,MAT_ksi,MAT_eta,Je,A,B,C,D,E,F,G,H] = MatrixTerms2D(Ne,D_ksi,
         D_eta,y_eta,y_ksi,x_eta,x_ksi,jac,M);
74          %Building Dx and Dy for RHS
         [Dx,Dy] = DxDy(Ne,ksi_x,ksi_y,eta_x,eta_y,D_ksi,D_eta);
76          %Building the surface jacobians for the faces
         [Sj1,Sj2,Sj3,Sj4] = SurfaceJac2D(Ne,L1,L2,L3,L4,x_ksi,x_eta,y_ksi,
         y_eta);
78          %Compute the Normals per element
         [nx_1,nx_2,nx_3,nx_4,ny_1,ny_2,ny_3,ny_4] = ElementNormals2D(Ne,L1,
         L2,L3,L4,x_ksi,x_eta,y_ksi,y_eta,Sj1,Sj2,Sj3,Sj4);
80          %Big Matrix of Ones
         M1 = ones(ngl*ngl);
82          %Building Initial Condition
         %Equation for initial condition is the following:
84          %dp/dt = n*pi*sin(n*pi*x)*sin(n*pi*y)*cos(n*pi*t)
         %P(x,y,t) = sin(n*pi*x)*sin(n*pi*y)*sin(n*pi*t)
86          for e = 1:Ne
             for i = 1:ngl
88                  for j = 1:ngl
                     I = intma(e,i,j);
90                      x = coord(I,1);
```

```matlab
                        y = coord(I,2);
                        q0{1}(e,i,j) = sqrt(2)*n*pi*sin(n*pi*x)*sin(n*pi*y);
                        q0{2}(e,i,j) = 0;
                    end
                end
            end
        q1 = q0;
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Time Step Calculation
        dt = ((1/Ne)/(N^2));
        Nt = round(t_final/dt);
        dt = t_final/Nt;
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        % Beginning of the RK54 Time integrator
         a = [ 0.0,
              -567301805773.0/ 1357537059087.0,
              -2404267990393.0/ 2016746695238.0,
              -3550918686646.0/ 2091501179385.0,
              -1275806237668.0/ 842570457699.0];

         b = [1432997174477.0/ 9575080441755.0,
              5161836677717.0/ 13612068292357.0,
              1720146321549.0/ 2090206949498.0,
              3134564353537.0/ 4481467310338.0,
              2277821191437.0/ 14882151754819.0];

        R{1} = 0;
        R{2} = 0;


        for k = 1:Nt
            qm = q0;
            for s = 1:5
                R{1} = a(s)*R{1};
                R{2} = a(s)*R{2};
                %Building all the flux information
                [fw1,fw2,fw3,fw4,fp1,fp2,fp3,fp4] = UpwindFlux2DFwFp(Ne,qm,
        L1,L2,L3,L4,EtoE,Dx,Dy,nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4);
```

```matlab
                %Solving Right hand side
                [R1] = RHSDG2Dn(Ne,MAT, Je ,M,M_1D,M1,qm,Dx,Dy,L1,L2,L3,L4,
        nx_1,ny_1, nx_2,ny_2,nx_3,ny_3,nx_4,ny_4,Sj1,Sj2,Sj3,Sj4,fw1,fw2,fw3
        ,fw4,fp1,fp2,fp3,fp4,c,MAT_ksi,MAT_eta);

                R{1} = R1{1} + R{1};
                R{2} = R1{2} + R{2};

                qm{1} = qm{1} + dt*b(s)*R{1};
                qm{2} = qm{2} + dt*b(s)*R{2};
            end
            q0 = qm;
        end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%For plotting purposes
    %Solving the exact solution at final time.
    for e = 1:Ne
        for i = 1:ngl
            for j = 1:ngl
                I = intma(e,i,j);
                x = coord(I,1);
                y = coord(I,2);
                qex{1}(e,i,j) = sqrt(2)*n*pi*sin(n*pi*x)*sin(n*pi*y)*cos(
        sqrt(2)*n*pi*t_final);
                qex{2}(e,i,j) = sin(n*pi*x)*sin(n*pi*y)*sin(sqrt(2)*n*pi*
        t_final);
            end
        end
    end
    %Plotting the exact solution vs. numerical solution.
    m=1;
    for e = 1:Ne
        for i = 1:ngl
            for j = 1:ngl

                w(m) = qm{1}(e,i,j);
                wexact(m) = qex{1}(e,i,j);
```

90

```
164                 p(m) = qm{2}(e,i,j);
                    pexact(m) = qex{2}(e,i,j);
166
                    I = intma(e,i,j);
168                 xp(m) = coord(I);
                    m=m+1;
170             end
            end
172     end
    %Building the Error Plot Information
174     L2errw = 0;
    L2errp = 0;
176
    for elm = 1:Ne
178         pnts = (elm-1)*(N+1)^2 + (1:(N+1)^2);
        dw   = w(pnts)-wexact(pnts);
180         dp   = p(pnts)-pexact(pnts);
        L2errw = dw*M*Je{elm}*dw'+L2errw;
182         L2errp = dp*M*Je{elm}*dp'+L2errp;
    end
184     L2err(1,z) = sqrt(Np);
    L2err(2,z) = sqrt(L2errw);
186     L2err(3,z) = sqrt(L2errp);
    z = z+1;
188     %Convergence Rates
    rate_w = (log(L2err(2,1:end-1))-log(L2err(2,2:end))) ./ (log(L2err
    (1,2:end))-log(L2err(1,1:end-1)));
190     rate_p = (log(L2err(3,1:end-1))-log(L2err(3,2:end))) ./ (log(L2err
    (1,2:end))-log(L2err(1,1:end-1)));

192     fprintf('Convergence rate for w: ')
    disp(rate_w)
194
    fprintf('Convergence rate for p: ')
196     disp(rate_p)
  end
198 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
200 %       1       2       3       4       5       6       7           8
```

```matlab
      c = [1 0 0;0 1 0;0 0 0; 0 0 0; 0 0 0; 0 0 0; 0 0 0; 0.1 0.2 0.8;...
202        0 0 0;0 0 0;0 0 0; 0 0 0; 0 0 0; 0 0 0; 0 0 0; 0.5 0.3 0.2];
      %     9      10      11      12      13      14      15      16
204 figure (1);
      %%%%%%%%%%%%%%%W Convergence Plot%%%%%%%%%%%%%%%%%%%%
206 subplot (1,2,1); %hold on
          loglog (L2err (1 ,:) ,L2err (2 ,:) ,'color ',c(N,:) )
208        title ('Convergence Plot for w')
          xlabel ('Np')
210        ylabel ('L2 w')
          axis tight
212 %%%%%%%%%%%%%%%P Convergence Plot%%%%%%%%%%%%%%%%%%%%
      subplot (1,2,2) ;% hold on
214        loglog (L2err (1 ,:) ,L2err (3 ,:) ,'color ',c(N,:) )
          title ('Convergence Plot for p')
216        xlabel ('Np')
          ylabel ('L2 p')
218        axis tight
      %Plot Numerical Solution P(x,y,t)
220 figure (2);
      title ('Numerical Solution for P')
222 xlabel ('x')
      ylabel ('y')
224 zlabel ('t')
      plot3 (coord (intma ,1) , coord (intma ,2) , qm{2}(:) ,'*')
226 %Plot Exact Solution of P(x,y,t)
      figure (3);
228 title ('Exact Solution for P')
      plot3 (coord (intma ,1) , coord (intma ,2) , qex {2}(:) ,'*')
230 %Plot Numerical Solution w
      figure (4);
232 title ('Numerical Solution for w')
      plot3 (coord (intma ,1) , coord (intma ,2) , qm{1}(:) ,'*')
234 %Plot Exact Solution of w
      figure (5);
236 title ('Exact Solution for w')
      plot3 (coord (intma ,1) , coord (intma ,2) , qex {1}(:) ,'*')
```

```matlab
%------------------------------------------------------------------------%
% Function for computing fw and fw with an Upwind Flux with p=0
% Boundary Conditions enforced on a washer mesh.
% Written by Benjamin Davis and Asst. Professor Jeremy Kozdon
%               Department of Applied Mathematics
%               Naval Postgraduate School
%               Monterey, CA 93943-5216
%------------------------------------------------------------------------%
function [fw1,fw2,fw3,fw4,fp1,fp2,fp3,fp4] = UpwindFlux2DFwFpBC(Ne,q1,L1
    ,L2,L3,L4,EtoEBC,Dx,...
    Dy,nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4)
    c = 1;
    for k = 1:Ne
        nE1 = EtoEBC(k,1);
        nE2 = EtoEBC(k,2);
        nE3 = EtoEBC(k,3);
        nE4 = EtoEBC(k,4);

        %Side one of element face
        wm1 = L1*q1{1}(k,:)';
        wp1 = L3*q1{1}(nE1,:)';
        if k == nE1
            wp1 = -wm1;
        end
        %Side two of element face
        wm2 = L2*q1{1}(k,:)';
        wp2 = L4*q1{1}(nE2,:)';
        if k == nE2
            wp2 = -wm2;
        end
        %Side three of element face
        wm3 = L3*q1{1}(k,:)';
        wp3 = L1*q1{1}(nE3,:)';
        if k == nE3
            wp3 = -wm3;
        end
        %Side four of element face
        wm4 = L4*q1{1}(k,:)';
```

```matlab
            wp4 = L2*q1{1}(nE4,:)';
        if k == nE4
            wp4 = -wm4;
        end
        %Side one of element face
        pm1 = nx_1{k}*L1*Dx{k}*q1{2}(k,:)'+ny_1{k}*L1*Dy{k}*q1{2}(k,:)';
        pp1 = nx_3{nE1}*L3*Dx{nE1}*q1{2}(nE1,:)'+ny_3{nE1}*L3*Dy{nE1}*q1{2}(nE1,:)';
        if k == nE1
            pp1 = -pm1;
        end
        %Side two of element face
        pm2 = nx_2{k}*L2*Dx{k}*q1{2}(k,:)'+ny_2{k}*L2*Dy{k}*q1{2}(k,:)';
        pp2 = nx_4{nE2}*L4*Dx{nE2}*q1{2}(nE2,:)'+ny_4{nE2}*L4*Dy{nE2}*q1{2}(nE2,:)';
        if k == nE2
             pp2 = -pm2;
        end
        %Side Three of element face
        pm3 = nx_3{k}*L3*Dx{k}*q1{2}(k,:)'+ny_3{k}*L3*Dy{k}*q1{2}(k,:)';
        pp3 = nx_1{nE3}*L1*Dx{nE3}*q1{2}(nE3,:)'+ny_1{nE3}*L1*Dy{nE3}*q1{2}(nE3,:)';
        if k == nE3
             pp3 = -pm3;
        end
        %Side Four of element face
        pm4 = nx_4{k}*L4*Dx{k}*q1{2}(k,:)'+ny_4{k}*L4*Dy{k}*q1{2}(k,:)';
        pp4 = nx_2{nE4}*L2*Dx{nE4}*q1{2}(nE4,:)'+ny_2{nE4}*L2*Dy{nE4}*q1{2}(nE4,:)';
        if k == nE4;
            pp4 = -pm4;
        end
        %Calculation of fp1,fp2,fp3,fp4,fw1,fw2,fw3,fw4
        fp1{k} = (1/2)*(pm1 - pp1) + 1/(2*c)*(wp1-wm1);
        fp2{k} = (1/2)*(pm2 - pp2) + 1/(2*c)*(wp2-wm2);
        fp3{k} = (1/2)*(pm3 - pp3) + 1/(2*c)*(wp3-wm3);
        fp4{k} = (1/2)*(pm4 - pp4) + 1/(2*c)*(wp4-wm4);


        fw1{k} = (1/2)*(wp1 - wm1) - c/2*(pp1+pm1);
```

```
73        fw2{k} = (1/2)*(wp2 - wm2) - c/2*(pp2+pm2);
          fw3{k} = (1/2)*(wp3 - wm3) - c/2*(pp3+pm3);
75        fw4{k} = (1/2)*(wp4 - wm4) - c/2*(pp4+pm4);
      end
77 end
```

```
1 %------------------------------------------------------------------------%
  %Function for computing fw and fp with and Upwind flux with no boundary
3 %conditions being enforced. Used with periodic boundary conditions.
  %Written by Benjamin Davis and Asst. Professor Jeremy Kozdon
5 %           Department of Applied Mathematics
  %           Naval Postgraduate School
7 %           Monterey, CA 93943-5216
  %------------------------------------------------------------------------%
9 function [fw1,fw2,fw3,fw4,fp1,fp2,fp3,fp4] = UpwindFlux2DFwFp(Ne,q1,L1,
      L2,L3,L4,EtoE,Dx,...
      Dy,nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4)
11    c = 1;
      for k = 1:Ne
13        nE1 = EtoE(k,1);
          nE2 = EtoE(k,2);
15        nE3 = EtoE(k,3);
          nE4 = EtoE(k,4);
17        %Side one of element face
          wm1 = L1*q1{1}(k,:)';
19        wp1 = L3*q1{1}(nE1,:)';
          %Side two of element face
21        wm2 = L2*q1{1}(k,:)';
          wp2 = L4*q1{1}(nE2,:)';
23        %Side three of element face
          wm3 = L3*q1{1}(k,:)';
25        wp3 = L1*q1{1}(nE3,:)';
          %Side four of element face
27        wm4 = L4*q1{1}(k,:)';
          wp4 = L2*q1{1}(nE4,:)';
29        %Side one of element face
```

```matlab
        pm1 = nx_1{k}*L1*Dx{k}*q1{2}(k,:)'+ny_1{k}*L1*Dy{k}*q1{2}(k,:)';
        pp1 = nx_3{nE1}*L3*Dx{nE1}*q1{2}(nE1,:)'+ny_3{nE1}*L3*Dy{nE1}*q1{2}(nE1,:)';
        %Side two of element face
        pm2 = nx_2{k}*L2*Dx{k}*q1{2}(k,:)'+ny_2{k}*L2*Dy{k}*q1{2}(k,:)';
        pp2 = nx_4{nE2}*L4*Dx{nE2}*q1{2}(nE2,:)'+ny_4{nE2}*L4*Dy{nE2}*q1{2}(nE2,:)';
        %Side Three of element face
        pm3 = nx_3{k}*L3*Dx{k}*q1{2}(k,:)'+ny_3{k}*L3*Dy{k}*q1{2}(k,:)';
        pp3 = nx_1{nE3}*L1*Dx{nE3}*q1{2}(nE3,:)'+ny_1{nE3}*L1*Dy{nE3}*q1{2}(nE3,:)';
        %Side Four of element face
        pm4 = nx_4{k}*L4*Dx{k}*q1{2}(k,:)'+ny_4{k}*L4*Dy{k}*q1{2}(k,:)';
        pp4 = nx_2{nE4}*L2*Dx{nE4}*q1{2}(nE4,:)'+ny_2{nE4}*L2*Dy{nE4}*q1{2}(nE4,:)';
        %Calculation of fp1,fp2,fp3,fp4,fw1,fw2,fw3,fw4
        fp1{k} = (1/2)*(pm1 - pp1) + 1/(2*c)*(wp1-wm1);
        fp2{k} = (1/2)*(pm2 - pp2) + 1/(2*c)*(wp2-wm2);
        fp3{k} = (1/2)*(pm3 - pp3) + 1/(2*c)*(wp3-wm3);
        fp4{k} = (1/2)*(pm4 - pp4) + 1/(2*c)*(wp4-wm4);

        fw1{k} = (1/2)*(wp1 - wm1) - c/2*(pp1+pm1);
        fw2{k} = (1/2)*(wp2 - wm2) - c/2*(pp2+pm2);
        fw3{k} = (1/2)*(wp3 - wm3) - c/2*(pp3+pm3);
        fw4{k} = (1/2)*(wp4 - wm4) - c/2*(pp4+pm4);
    end
end
```

```matlab
%------------------------------------------------------------------------%
%Central Flux Routine.  Provides the center flux values for fw and fp
%for the two-dimensional problem.
%Written by Benjamin Davis
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%------------------------------------------------------------------------%%
```

```
function [fw1,fw2,fw3,fw4,fp1,fp2,fp3,fp4] = CenterFlux2DFwFp(Ne,q1,L1,
    L2,L3,L4,EtoE,Dx,...
    Dy,nx_1,ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4)
    for k = 1:Ne
        nE1 = EtoE(k,1);
        nE2 = EtoE(k,2);
        nE3 = EtoE(k,3);
        nE4 = EtoE(k,4);
        %Side one of element face
        wm1 = L1*q1{1}(k,:)';
        wp1 = L3*q1{1}(nE1,:)';
        %Side two of element face
        wm2 = L2*q1{1}(k,:)';
        wp2 = L4*q1{1}(nE2,:)';
        %Side three of element face
        wm3 = L3*q1{1}(k,:)';
        wp3 = L1*q1{1}(nE3,:)';
        %Side four of element face
        wm4 = L4*q1{1}(k,:)';
        wp4 = L2*q1{1}(nE4,:)';

        fw1{k} = (1/2)*(wp1 - wm1);
        fw2{k} = (1/2)*(wp2 - wm2);
        fw3{k} = (1/2)*(wp3 - wm3);
        fw4{k} = (1/2)*(wp4 - wm4);


        %Calculation of fp1,fp2,fp3,fp4
        %Side one of element face
        pm1 = nx_1{k}*L1*Dx{k}*q1{2}(k,:)'+ny_1{k}*L1*Dy{k}*q1{2}(k,:)';
        pp1 = nx_3{nE1}*L3*Dx{nE1}*q1{2}(nE1,:)'+ny_3{nE1}*L3*Dy{nE1}*q1
    {2}(nE1,:)';
        %Side two of element face
        pm2 = nx_2{k}*L2*Dx{k}*q1{2}(k,:)'+ny_2{k}*L2*Dy{k}*q1{2}(k,:)';
        pp2 = nx_4{nE2}*L4*Dx{nE2}*q1{2}(nE2,:)'+ny_4{nE2}*L4*Dy{nE2}*q1
    {2}(nE2,:)';
        %Side Three of element face
        pm3 = nx_3{k}*L3*Dx{k}*q1{2}(k,:)'+ny_3{k}*L3*Dy{k}*q1{2}(k,:)';
        pp3 = nx_1{nE3}*L1*Dx{nE3}*q1{2}(nE3,:)'+ny_1{nE3}*L1*Dy{nE3}*q1
    {2}(nE3,:)';
```

```matlab
            %Side Four of element face
            pm4 = nx_4{k}*L4*Dx{k}*q1{2}(k,:) '+ny_4{k}*L4*Dy{k}*q1{2}(k,:) ';
            pp4 = nx_2{nE4}*L2*Dx{nE4}*q1{2}(nE4,:) '+ny_2{nE4}*L2*Dy{nE4}*q1
    {2}(nE4,:) ';

            fp1{k} = (1/2)*(pm1 - pp1);
            fp2{k} = (1/2)*(pm2 - pp2);
            fp3{k} = (1/2)*(pm3 - pp3);
            fp4{k} = (1/2)*(pm4 - pp4);
        end
    end
```

```matlab
%-----------------------------------------------------------------------%
% Function to solve the combined discretized equations from Chapter 4
% for the 2D Acoustic Wave equation.  Computes RHS and inverts the
% required matrices to compute w and p for second order wave equation.
% Written by Benjamin Davis and Asst. Professor Jeremy Kozdon
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%-----------------------------------------------------------------------%
function [R] = RHSDG2Dn(Ne,MAT, Je ,M,M_1D,M1, qs ,Dx,Dy,L1,L2,L3,L4,nx_1,
    ny_1,nx_2,ny_2,nx_3,ny_3,nx_4,ny_4,Sj1,Sj2,Sj3,Sj4,fw1,fw2,fw3,fw4,
    fp1,fp2,fp3,fp4,c,MAT_ksi,MAT_eta)

    R{1} = zeros(size(qs{1}));
    R{2} = zeros(size(qs{2}));

    for e = 1:Ne
      R{2}(e,:) = (MAT{e}+M1'*Je{e}*M)*qs{1}(e,:) ';

      R{2}(e,:) = R{2}(e,:) ' + ((Dx{e}'*L1'*nx_1{e}+ Dy{e}'*L1'*ny_1{e})
    *M_1D*Sj1{e}*fw1{e});
      R{2}(e,:) = R{2}(e,:) ' + ((Dx{e}'*L2'*nx_2{e}+ Dy{e}'*L2'*ny_2{e})
    *M_1D*Sj2{e}*fw2{e});
```

```matlab
        R{2}(e,:) = R{2}(e,:)' + ((Dx{e}'*L3'*nx_3{e}+ Dy{e}'*L3'*ny_3{e})
    *M_1D*Sj3{e}*fw3{e});
        R{2}(e,:) = R{2}(e,:)' + ((Dx{e}'*L4'*nx_4{e}+ Dy{e}'*L4'*ny_4{e})
    *M_1D*Sj4{e}*fw4{e});

        R{1}(e,:) = c^2*(L1'*M_1D*Sj1{e}*fp1{e}+ L2'*M_1D*Sj2{e}*fp2{e}+L3
    '*M_1D*Sj3{e}*fp3{e}+L4'*M_1D*Sj4{e}*fp4{e});

        R{1}(e,:) = R{1}(e,:)' - c^2*(MAT_ksi{e}+MAT_eta{e})*qs{2}(e,:)';

        M11{e} = (MAT{e}+M1'*Je{e}*M);

        [L_1,U_1] = lu(M11{e}, 'vector');

        R1 = U_1 \ (L_1 \ R{2}(e,:)');
        R1 = R1';
        R{2}(e,:) = R1;

        M1_2{e} = Je{e}*M;
        [L_2,U_2] = lu(M1_2{e}, 'vector');

        R2 = U_2 \ (L_2 \ R{1}(e,:)');
        R2 = R2';
        R{1}(e,:) = R2;

    end
end
```

```matlab
9  %               Department of Applied Mathematics
   %               Naval Postgraduate School
11 %               Monterey , CA 93943-5216
   % INPUT LIST: nelx and nely are the number of elements in x and y
13 %               nop is the polynomial order
   %               xgl are the interpolation points on the element.
15 % OUTPUT LIST:
   %               coord are the coordinates: x=coord(:,1) and y=coord(:,2)
17 %               intma is the connectivity list that points to the global
   %               gridpoint number
19 %               bsido is the boundary data (used by ISIDE and FACE)
   %               iperiodic points to another point if periodicity is
21 %               applicable
   %               npoin = number of global points
23 %               nelem = number of elements
   %               nboun = number of boundary edges
25 %               nface = number of faces/edges in the grid
   %------------------------------------------------------------------------%
27 function [coord ,intma ,bsido ,iperiodic ,npoin ,nelem ,nboun ,nface] = ...
       create_grid_2d(nelx ,nely ,nop ,xgl ,plot_grid , skew_mesh)
   %Define Grid Dimensions
29 ngl=nop+1;
   npoin=(nop*nelx + 1)*(nop*nely + 1);
31 nelem=nelx*nely;
   nboun=2*nelx + 2*nely;
33 nface=2*nelem + nelx + nely;
   %Initialize Global Arrays
35 coord=zeros(npoin ,2);
   intma=zeros(nelem ,ngl ,ngl);
37 bsido=zeros(nboun ,4);
   iperiodic=zeros(npoin ,1);
39 %Initialize Local Arrays
   node=zeros(npoin ,npoin);
41 %Set some constants
   xmin=-1;
43 xmax=+1;
   ymin=-1;
45 ymax=+1;
   dx=(xmax-xmin)/nelx;
```

```matlab
   dy=(ymax-ymin)/nely;
   nop=ngl-1;
   nx=nelx*nop + 1;
   ny=nely*nop + 1;
   %GENERATE COORD
   ip=0;
   jj=0;
   for k=1:nely
      y0=ymin + real(k-1)*dy;
      if (k == 1)
          l1=1;
      else
          l1=2;
      end

      for l=l1:ngl
          jj=jj+1;
          ii=0;
          for i=1:nelx
              x0=xmin + real(i-1)*dx;
              xc = x0 + [0,1;0,1]*dx;
              yc = y0 + [0,0;1,1]*dy;
              if(skew_mesh==1)
                 xc=xc+(1/8)*sin(pi*yc).*(1-xc).*(1+xc);%x;
                 yc=yc+(1/8)*sin(pi*xc).*(1-yc).*(1+yc);%y;
              end
              if (i == 1)
                  j1=1;
              else
                  j1=2;
              end
              for j=j1:ngl
                  ii=ii + 1;
                  ip=ip + 1;
                  ax=(xgl(j)+1)/2;
                  ay=(xgl(l)+1)/2;
                  x = xc(1,1)*(1-ax)*(1-ay) + xc(1,2)*ax*(1-ay)...
                      + xc(2,1)*(1-ax)*(  ay) + xc(2,2)*ax*(  ay);
                  y = yc(1,1)*(1-ax)*(1-ay) + yc(1,2)*ax*(1-ay)...
```

```matlab
                    + yc(2,1)*(1-ax)*(   ay) + yc(2,2)*ax*(   ay);
                coord(ip,1)=x;
                coord(ip,2)=y;
                if (skew_mesh==2)
                  coord(ip,1)=x  +(1/8)*sin(pi*y)*(1-x)*(1+x);%x;
                  coord(ip,2)=y  +(1/8)*sin(pi*x)*(1-y)*(1+y);%y;
                end
                node(ii,jj)=ip;
            end %j

        end %i
    end %l
end %k
%GENERATE INTMA
ie=0;
for k=1:nely
    for i=1:nelx
        ie=ie+1;
        for l=1:ngl
            jj=(ngl-1)*(k-1) + l;
            for j=1:ngl
                ii=(ngl-1)*(i-1) + j;
                ip=node(ii,jj);
                intma(ie,j,l)=ip;
            end %j
        end %l
    end %i
end %k
%Generate BSIDO
ib=0;
for i=1:nelx
    ie=i;
    ib=ib+1;
    i1=(i-1)*(ngl-1) + 1;
    i2=(i-1)*(ngl-1) + ngl;
    ip1=node(i1,1);
    ip2=node(i2,1);
    bsido(ib,1)=ip1;
    bsido(ib,2)=ip2;
```

```matlab
125        bsido(ib,3)=ie;
           bsido(ib,4)=6;
127 end
    %Right Boundary
129 for i=1:nely
        ie=(nelx)*(i);
131     ib=ib+1;
        i1=(i-1)*(ngl-1) + 1;
133     i2=(i-1)*(ngl-1) + ngl;
        ip1=node(nx,i1);
135     ip2=node(nx,i2);
        bsido(ib,1)=ip1;
137     bsido(ib,2)=ip2;
        bsido(ib,3)=ie;
139     bsido(ib,4)=6;
    end
141 %Top Boundary
    for i=nelx:-1:1
143     ie=nelem - (nelx - i);
        ib=ib+1;
145     i1=(i-1)*(ngl-1) + ngl;
        i2=(i-1)*(ngl-1) + 1;
147     ip1=node(i1,ny);
        ip2=node(i2,ny);
149     bsido(ib,1)=ip1;
        bsido(ib,2)=ip2;
151     bsido(ib,3)=ie;
        bsido(ib,4)=6;
153 end
    %Left Boundary
155 for i=nely:-1:1
        ie=(nelx)*(i-1) + 1;
157     ib=ib+1;
        i1=(i-1)*(ngl-1) + ngl;
159     i2=(i-1)*(ngl-1) + 1;
        ip1=node(1,i1);
161     ip2=node(1,i2);
        bsido(ib,1)=ip1;
163     bsido(ib,2)=ip2;
```

```matlab
        bsido(ib,3)=ie;
        bsido(ib,4)=6;
    end
%Periodicity
for i=1:npoin
    iperiodic(i)=i;
end
%X-Periodicity
for i=1:ny
    i1=node(1,i);
    i2=node(nx,i);
    iperiodic(i2)=i1;
end
%Y-Periodicity
for i=1:nx
    i1=node(i,1);
    i2=node(i,ny);
    iperiodic(i2)=iperiodic(i1);
end
%Plot Grid
if (plot_grid == 1)
    x=zeros(5,1);
    y=zeros(5,1);
    figure;
    hold on;
    for e=1:nelem
        for j=1:ngl-1
            for i=1:ngl-1
                i1=intma(e,i,j);
                i2=intma(e,i+1,j);
                i3=intma(e,i+1,j+1);
                i4=intma(e,i,j+1);
                x(1)=coord(i1,1); y(1)=coord(i1,2);
                x(2)=coord(i2,1); y(2)=coord(i2,2);
                x(3)=coord(i3,1); y(3)=coord(i3,2);
                x(4)=coord(i4,1); y(4)=coord(i4,2);
                x(5)=coord(i1,1); y(5)=coord(i1,2);
                plot_handle=plot(x,y,'-r');
                set(plot_handle,'LineWidth',1.5);
```

104

```matlab
203                    end
              end
205           i1=intma(e,1,1);
              i2=intma(e,ngl,1);
207           i3=intma(e,ngl,ngl);
              i4=intma(e,1,ngl);
209           x(1)=coord(i1,1); y(1)=coord(i1,2);
              x(2)=coord(i2,1); y(2)=coord(i2,2);
211           x(3)=coord(i3,1); y(3)=coord(i3,2);
              x(4)=coord(i4,1); y(4)=coord(i4,2);
213           x(5)=coord(i1,1); y(5)=coord(i1,2);
              plot_handle=plot(x,y,'-b');
215           set(plot_handle,'LineWidth',2);
       end
217    title_text=['Grid Plot For: Ne = ' num2str(nelem) ', N = ' num2str(
      nop)];
       title([title_text],'FontSize',18);
219    xlabel('X','FontSize',18);
       ylabel('Y','FontSize',18);
221    axis image
end
```

```matlab
%------------------------------------------------------------------------%
2  %Function provided to Professor F.X. Giraldo's MA4245 class.
   %Used by Benjamin Davis for his thesis project.
4  %This subroutine creates the array ISIDE which stores all of
   %the information concerning the sides of all the elements.
6  %Written by Francis X. Giraldo on 1/01
   %           Naval Postgraduate School
8  %           Department of Applied Mathematics
   %           Monterey, CA 93943-5502
10 % INPUT LIST: intma = element connectivity
   %             bsido = boundary info (which points are on a boundary,
12 %                     which element it belongs to and the boundary
   %                     condition).
14 %             npoin = number of global points
```

```matlab
%            nelem = number of elements
%            nboun = number of boundary faces/edges
%            nside=nface are the number of sides/face/edges in the grid
%            ngl = number of points in one direction in an element
% OUTPUT LIST: iside = face information such as which points are on a
%                     face which elements they belong to and, if a
%                     boundary, what is the boundary condition.
%               jeside = for each element and each edge gives the FACE
%               number
%--------------------------------------------------------------------%
function [iside, jeside] = create_side(intma, bsido, npoin, nelem, nboun,
    nface, ngl)
%global arrays
iside = zeros(nface,4);
jeside= zeros(nelem,4);
%local arrays
lwher = zeros(npoin,1);
lhowm = zeros(npoin,1);
icone = zeros(5*npoin,1);
inode = zeros(4,1);
jnode = zeros(4,1);
%Fix lnode
inode(1)=1;
inode(2)=ngl;
inode(3)=ngl;
inode(4)=1;
jnode(1)=1;
jnode(2)=1;
jnode(3)=ngl;
jnode(4)=ngl;
%count how many elements own each node
for in=1:4
    for ie=1:nelem
        ip=intma(ie,inode(in),jnode(in));
        lhowm(ip)=lhowm(ip) + 1;
    end %ie
end %in
%track elements owning each node
lwher(1)=0;
```

106

```matlab
     for ip=2:npoin
54       lwher(ip)=lwher(ip-1) + lhowm(ip-1);
     end %ip
56 %another tracker array
   lhowm = zeros(npoin,1);
58 for in=1:4
       for ie=1:nelem
60           ip=intma(ie,inode(in),jnode(in));
           lhowm(ip)=lhowm(ip) + 1;
62           jloca=lwher(ip) + lhowm(ip);
           icone(jloca)=ie;
64       end %ie
   end %in
66 %LOOP OVER THE NODES
   iloca=0;
68 for ip=1:npoin
       iloc1=iloca;
70       iele=lhowm(ip);
       if (iele ~= 0 )
72           iwher=lwher(ip);
           %LOOP OVER THOSE ELEMENTS SURROUNDING NODE IP
74           ip1=ip;
           for iel=1:iele
76               ie=icone(iwher+iel);
               %find out position of ip in intma
78               for in=1:4
                   in1=in;
80                   ipt=intma(ie,inode(in),jnode(in));
                   if (ipt == ip)
82                       break
                   end
84               end %in
               %Check Edge of Element IE which claims IP
86               j=0;
               for jnod=1:2:3
88                   iold=0;
                   j=j+1;
90                   in2=in + jnod;
                   if (in2 > 4)
```

```matlab
                    in2=in2 -4;
                end
                ip2=intma ( ie , inode ( in2 ) , jnode ( in2 ) ) ;
                if ( ip2 >= ip1 )
                   %check whether side is old or new
                   if ( iloca ~= iloc1 )
                      for is=iloc1 +1: iloca
                           iside ( is ,2) ;
                           jloca=is ;
                           if ( iside ( is ,2) == ip2 )
                               iold =1;
                               break ;
                           end
                      end %is
                   end %iloca
                   if ( iold == 0)
                      %NEW SIDE
                      iloca=iloca + 1;
                      iside ( iloca ,1)=ip1 ;
                      iside ( iloca ,2)=ip2 ;
                      iside ( iloca ,2+ j )=ie ;
                   elseif ( iold == 1)
                      %OLD SIDE
                      iside ( jloca ,2+ j )=ie ;
                   end %iold
                end %ip2
            end %jnod
        end %iel
        %Perform some Shifting to order the nodes of a side in CCW
    direction
        for is=iloc1 +1: iloca
            if ( iside ( is ,3) == 0)
                iside ( is ,3)=iside ( is ,4) ;
                iside ( is ,4) =0;
                iside ( is ,1)=iside ( is ,2) ;
                iside ( is ,2)=ip1 ;
            end %iside
        end %is
    end %if iele
```

```matlab
130  end %ip
     if (iloca ~= nface)
132      disp( 'Error in SIDE. iloca nface = ');
         iloca
134      nface
         pause
136  end
     %RESET THE BOUNDARY MARKERS
138  for is=1:nface
         if (iside(is,4) == 0)
140          il=iside(is,1);
             ir=iside(is,2);
142          ie=iside(is,3);
             for ib=1:nboun
144              ibe=bsido(ib,3);
                 ibc=bsido(ib,4);
146              if (ibe == ie)
                     ilb=bsido(ib,1);
148                  irb=bsido(ib,2);
                     if (ilb == il && irb == ir)
150                      iside(is,4)=-ibc;
                         break
152                  end %ilb
                 end %ibe
154          end %ib
         end %iside
156  end %is
     %FORM ELEMENT/SIDE CONNECTIVITY ARRAY
158  for is=1:nface
         iel=iside(is,3);
160      ier=iside(is,4);
         is1=iside(is,1);
162      is2=iside(is,2);
         %LEFT SIDE
164      for in=1:4
             i1=intma(iel,inode(in),jnode(in));
166          in1=in + 1;
             if (in1 > 4)
168              in1=1;
```

```matlab
            end %in1
            i2=intma(iel,inode(in1),jnode(in1));
            if ((is1 == i1) && (is2 == i2))
                jeside(iel,in)=is;
            end %is1
        end %in
        %RIGHT SIDE
        if (ier > 0)
            for in=1:4
                i1=intma(ier,inode(in),jnode(in));
                in1=in + 1;
                if (in1 > 4)
                    in1=1;
                end %in1
                i2=intma(ier,inode(in1),jnode(in1));
                if ((is1 == i2) && (is2 == i1))
                    jeside(ier,in)=is;
                end %is1
            end %in
        end %ier
end %is
```

```matlab
%-------------------------------------------------------------------------%
%Function provided to Professor F.X. Giraldo's MA4245 class. Used by
%Benjamin Davis. This subroutine constructs the Side Information for
%a High Order Spectal Element Quads
%Written by Francis X. Giraldo
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
% INPUT LIST: iside = face information to know which points are on a
%                     face and which elements they belong to
%             intma = element connectivity
%             nface = number of faces/edges in the grid
%             ngl = number of interpolation points in one direction
%                   in an element
```

110

```matlab
% OUTPUT LIST: face = face information stating which local edge number
%                    the face is on and which elements they belong to
%                      (left and right neighbors).are the metric terms
%                  needed to imapl and imapr give the tensor-product (i,j)
%                  points of the edge points.
%-----------------------------------------------------------------%
function [face,imapl,imapr]=create_face(iside,intma,nface,ngl)
%global arrays
face=zeros(nface,4);
imapl=zeros(4,2,ngl);
imapr=zeros(4,2,ngl);
%local arrays
inode=zeros(4,1);
jnode=zeros(4,1);
%Construct Boundary Pointer
inode(1)=1;
inode(2)=ngl;
inode(3)=ngl;
inode(4)=1;
jnode(1)=1;
jnode(2)=1;
jnode(3)=ngl;
jnode(4)=ngl;
%Construct IMAP arrays
for l=1:ngl
    %eta=-1
    imapl(1,1,l)=l;
    imapl(1,2,l)=1;
    imapr(1,1,l)=ngl+1-l;
    imapr(1,2,l)=1;
    %ksi=+1
    imapl(2,1,l)=ngl;
    imapl(2,2,l)=l;
    imapr(2,1,l)=ngl;
    imapr(2,2,l)=ngl+1-l;
    %eta=+1
    imapl(3,1,l)=ngl+1-l;
    imapl(3,2,l)=ngl;
    imapr(3,1,l)=l;
```

111

```matlab
      imapr(3,2,l)=ngl;
      %ksi=-1
      imapl(4,1,l)=1;
      imapl(4,2,l)=ngl+1-l;
      imapr(4,1,l)=1;
      imapr(4,2,l)=l;
end %l
%loop thru the sides
for i=1:nface
    ip1=iside(i,1);
    ip2=iside(i,2);
    iel=iside(i,3);
    ier=iside(i,4);
    %check for position on Left Element
    for j=1:4
        j1=j;
        j2=j+1;
        if (j2 > 4)
            j2=1;
        end %j2
        jp1=intma(iel,inode(j1),jnode(j1));
        jp2=intma(iel,inode(j2),jnode(j2));
        if (ip1 == jp1 && ip2 == jp2)
            face(i,1)=j;
            break; %leave J loop
        end %ip1
    end %j
    %check for position on Right Element
    if (ier > 0)
        for j=1:4
            j1=j;
            j2=j+1;
            if (j2 > 4)
                j2=1;
            end %j2
            jp1=intma(ier,inode(j1),jnode(j1));
            jp2=intma(ier,inode(j2),jnode(j2));

            if (ip1 == jp2 && ip2 == jp1)
```

```
93            face(i,2)=j;
             break;              %leave J loop
95          end %ip1
         end %j
97     end %ier
     %Store Elements into face
99     face(i,3)=iel;
     face(i,4)=ier;
101 end %i
```

```
1  %------------------------------------------------------------------%
   %Function for changing the face code to enforce p=0 boundary conditions
3  %for 2nd order acoustic wave equation on a washer.
   %Written by Benjamin Davis Created: 09 May 2015
5  %            Department of Applied Mathematics
   %            Naval Postgraduate School
7  %            Monterey, CA 93943-5216
   %------------------------------------------------------------------%
9  function [face] = faceBC(nface,face)
       for k = 1:nface
11         if face(k,1) == 2
               if face(k,4) == -6
13                 face(k,4) = -1;
               end
15         end
       end
17     for k = 1:nface
           if face(k,1) == 4
19             if face(k,4) == -6
                   face(k,4) = -1;
21             end
           end
23     end
   end
```

```matlab
%-------------------------------------------------------------------------%
%Function provided by Professor F.X. Giraldo to his MA4245 class.
%Used by Benjamin Davis
%This subroutine builds Periodic BCs along the 4 edges of a rectangular
%domain.
%Written by Francis X. Giraldo on 2/2007
%           Naval Postgraduate School
%           Department of Applied Mathematics
%           Monterey, CA 93943-5216
% INPUT LIST: iside = face information
%             face = more face information
%             coord = gridpoint coordinates
%             nface = number of faces
%             nboun = number of boundaries
% OUTPUT LIST: face = augments the FACE data structure to include
%                     periodicity
%-------------------------------------------------------------------------%
function face = create_face_periodicity(iside,face,coord,nface,nboun)
%Constant
tol=1e-6;
%Local arrays
ileft=zeros(nboun/4,1);
iright=zeros(nboun/4,1);
itop=zeros(nboun/4,1);
ibot=zeros(nboun/4,1);
%initialize
nleft=0; nright=0; ntop=0; nbot=0;
%Find Extrema of Domain
xmax=max(coord(:,1)); xmin=min(coord(:,1));
ymax=max(coord(:,2)); ymin=min(coord(:,2));
%loop thru sides and extract Left, Right, Bot, and Top
for is=1:nface
    %Check for Periodicity Edges
    ier=face(is,4);
    if (ier == -6)
        i1=iside(is,1); i2=iside(is,2);
        xm=0.5*( coord(i1,1) + coord(i2,1) );
        ym=0.5*( coord(i1,2) + coord(i2,2) );
```

```matlab
           %check Grid Point
40         if ( abs(xm - xmin) < tol ) %left boundary
               nleft=nleft + 1;
42             ileft(nleft)=is;
           elseif ( abs(xm - xmax) < tol ) %right boundary
44             nright=nright + 1;
               iright(nright)=is;
46         elseif ( abs(ym - ymin) < tol ) %bottom boundary
               nbot=nbot + 1;
48             ibot(nbot)=is;
           elseif ( abs(ym - ymax) < tol ) %top boundary
50             ntop=ntop + 1;
               itop(ntop)=is;
52         else
               disp('No match in PERIODIC_BCS for is ier = ');
54             is
               ier
56             pause
           end %if
58     end %ier
   end %is
60 %Loop through Periodic BCs
   %First: Do Left and Right
62 for i=1:nleft
       isl=ileft(i);
64     i1=iside(isl,1);
       yl1=coord(i1,2);
66     %Search for Corresponding Right Edge
       for j=1:nright
68         isr=iright(j);
           i2=iside(isr,2);
70         yr2=coord(i2,2);
           if ( abs(yl1-yr2) < tol ) %they match
72             face(isl,2)=face(isr,1);
               face(isl,4)=face(isr,3);
74             face(isr,3)=-6; %means skip it due to Periodicity
               iside(isl,4)=iside(isr,3);
76             iside(isr,3)=-6;
               break;
```

115

```
78          end %if
       end %j
80 end %i
   %Second: Do Top and Bottom
82 for i=1:nbot
       isl=ibot(i);
84     i1=iside(isl,1);
       xl1=coord(i1,1);
86     %Search for Corresponding Top Edge
       for j=1:ntop
88         isr=itop(j);
           i2=iside(isr,2);
90         xr2=coord(i2,1);
           if ( abs(xl1-xr2) < tol ) %they match
92             face(isl,2)=face(isr,1);
               face(isl,4)=face(isr,3);
94             face(isr,3)=-6; %means skip it due to Periodicity
               iside(isl,4)=iside(isr,3);
96             iside(isr,3)=-6;
               break;
98         end %if
        end %j
100 end %i
```

```
%--------------------------------------------------------------------%
2 %Function given to Professor F.X. Giraldo's MA4245 class.  Modified by
   %Ben Davis and Jeremy Kozdon.
4 %This function computes the Metric Terms for General 2D Quad Grids.
   %Written by F.X. Giraldo on 4/2008
6 %          Department of Applied Mathematics
   %          Naval Postgraduate School
8 %          Monterey, CA 93943-5216
   % INPUT LIST: coord = gridpoint coordinates
10 %             intma = element connectivity
   %             psi = basis functions defined as psi(NGL,NQ)
12 %             dpsi = derivative of basis functions defined as
```

```matlab
%                              dpsi (NGL,NQ)
%                   nelem = number of elements
%                   ngl = number of interpolation points in one direction in
%                         an element
%                   nq = number of integration/quadrature points in one
%                        direction in an element.
% OUTPUT LIST: ksi_x , ksi_y , eta_x , eta_y = are the metric terms needed to
%                                              compute derivatives in
%                                              physical space
%                   x_ksi , x_eta , y_ksi , y_eta = are the metric terms needed to
%                                              compute derivatives in
%                                              physical space
%                   jac = weights*Jacobian defined as jac (nelem , nq , nq )
%------------------------------------------------------------------------%
function [ ksi_x , ksi_y , eta_x , eta_y , x_ksi , x_eta , y_ksi , y_eta , jac ] = ...
    metrics2 ( coord , intma , psi , dpsi , nelem , ngl , nq )
%Initialize Global Arrays
ksi_x=zeros (nelem , nq , nq ) ;
ksi_y=zeros (nelem , nq , nq ) ;
eta_x=zeros (nelem , nq , nq ) ;
eta_y=zeros (nelem , nq , nq ) ;
jac=zeros (nelem , nq , nq ) ;
%Initialize Local Arrays
x_ksi=zeros (nelem , nq , nq ) ;
x_eta=zeros (nelem , nq , nq ) ;
y_ksi=zeros (nelem , nq , nq ) ;
y_eta=zeros (nelem , nq , nq ) ;
x=zeros (ngl , ngl ) ;
y=zeros (ngl , ngl ) ;
%loop thru the elements
for ie =1:nelem
    %Store Element Variables
    for j =1:ngl
    for i =1:ngl
        ip=intma (ie , i , j ) ;
        x( i , j )=coord (ip ,1) ;
        y( i , j )=coord (ip ,2) ;
    end %i
    end %j
```

```matlab
      %Construct Mapping Derivatives: dx/dksi, dx/deta, dy/dksi, dy/deta
52    [x_ksi(ie,:,:),x_eta(ie,:,:)]=map_deriv(psi,dpsi,x,ngl,nq);
      [y_ksi(ie,:,:),y_eta(ie,:,:)]=map_deriv(psi,dpsi,y,ngl,nq);
54    %Construct Inverse Mapping: dksi/dx, dksi/dy, deta/dx, deta/dy
      for j=1:nq
56    for i=1:nq
          xjac=x_ksi(ie,i,j)*y_eta(ie,i,j) - x_eta(ie,i,j)*y_ksi(ie,i,j);
58        ksi_x(ie,i,j)=+1.0/xjac*y_eta(ie,i,j);
          ksi_y(ie,i,j)=-1.0/xjac*x_eta(ie,i,j);
60        eta_x(ie,i,j)=-1.0/xjac*y_ksi(ie,i,j);
          eta_y(ie,i,j)=+1.0/xjac*x_ksi(ie,i,j);
62        jac(ie,i,j)=abs(xjac);
      end %i
64    end %j
  end %ie
```

```matlab
1  ----------------------------------------------------------------------%
   % Function created for taking the x coordinates from coord and
3  % turning them into radius polar coordinates based on user inputs.
   %Written by Benjamin Davis %Created: 14 April 2015
5  %             Department of Applied Mathematics
   %             Naval Postgraduate School
7  %             Monterey, CA 93943-5216
   %----------------------------------------------------------------------%
9  function[R] = radius(coord,r1,r2)
   n = length(coord(:,1));
11     for i = 1:n
           R(i,1) = ((r2-r1)/2)*coord(i,1) + (r1 + r2)/2;
13     end
   end
```

118

```matlab
%-------------------------------------------------------------------------%
%Function to turn the y coord into theta. Conversion from Cartesian to
%polar coordinates. Also, rn is a scaling portion of the washer.
%Written by Benjamin Davis Created: 14 April 2015
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%-------------------------------------------------------------------------%
function [theta] = thetapolar(coord,rn)
n = length(coord(:,2));

r1 = 0;
r2 = (2*pi)/rn;
    for i = 1:n
        theta(i,1) = ((r2-r1)/2)*coord(i,2) + (r1 + r2)/2;
    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Function to overwrite original coord for the conversion to polar mesh.
%Function is required for the washer mesh.
%Written by Benjamin Davis Created: 14 April 2015
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%-------------------------------------------------------------------------%
function[coord] = newcoord(R,theta,coord)
n = length(coord(:,1));
    for i = 1:n
        coord(i,1) = R(i)*cos(theta(i));
        coord(i,2) = R(i)*sin(theta(i));
    end
end
```

```matlab
%-----------------------------------------------------------------------%
%Element to Element function with Boundary Conditions for Washer Grid
%Synopsis: Element to element function for identifying and storing
%information for looking across the faces of an element to it's
%corresponding neighbor.
%Written by Benjamin Davis
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%
%           pL = Face of Left Element
%           pR = Face of Right Element
%           Ls = Number of Left Element
%           Rs = Number of Right Element
%Output:    EtoE is a Matrix of Number of Elements (Row) vs. the Number
%of Sides per element (Four for a square grid). For each element, it
%displays its corresponding neighbor with respect to boundary
%conditions.
%-----------------------------------------------------------------------%
function [EtoEBC] = EtoEfunctBC(nface,face,Ne)
EtoEBC = zeros(Ne,4);
    for l=1:nface
        pL = face(l,1); %Face of left element
        pR = face(l,2); %Face of Right Element
        Ls = face(l,3); %Number of Left Element
        Rs = face(l,4); %Number of Right Element

            if Rs == -1
                EtoEBC(Ls,pL) = Ls;
            elseif (Ls ~= -6 && Rs ~= -6)
                EtoEBC(Ls,pL) = Rs;
                EtoEBC(Rs,pR) = Ls;
            end
    end
end
```

120

```matlab
%-----------------------------------------------------------------------%
%Element to Element function for a square grid with periodic boundary
%conditions.
%%Synopsis: Element to element function for identifying and storing
%information for looking across the faces of an element to it's
%corresponding neighbor.
%Written by Benjamin Davis
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%
%            pL = Face of Left Element
%            pR = Face of Right Element
%            Ls = Number of Left Element
%            Rs = Number of Right Element
%
%Output:     EtoE is a Matrix of of Number of Elements (Row) vs. the
%    Number
%of Sides per element (Four for a square grid).  For each element, it
%displays its corresponding neighbor.
%-----------------------------------------------------------------------%
function [EtoE] = EtoEfunct(nface,face,Ne)
EtoE = zeros(Ne,4);
    for l=1:nface
        pL = face(l,1);
        pR = face(l,2);
        Ls = face(l,3);
        Rs = face(l,4);

        if (Ls ~= -6 && Rs ~= -6)
            EtoE(Ls,pL) = Rs;
            EtoE(Rs,pR) = Ls;
        end
    end
end
```

```matlab
%------------------------------------------------------------------------%
%Function for building the Mass and Differentiation matrices for
%2-Dimension.
%Written by Benjamin Davis Created: July 2014 Modified Jan 2015
%              Department of Applied Mathematics
%              Naval Postgraduate School
%              Monterey, CA 93943-5216
%------------------------------------------------------------------------%
function [M,D] = mass_diff2D(L,dL,w)
n = length(L(:,1));
M = zeros(n,n);
D = zeros(n,n);
    for i = 1:n
        for j = 1:n
            M(i,j) = ((L(i,:).*L(j,:))*w(1,:)');
            D(i,j) = L(i,:)*dL(j,:)';
        end
    end
end
```

```matlab
%------------------------------------------------------------------------%
%Written by Benjamin Davis
%              Department of Applied Mathematics
%              Naval Postgraduate School
%              Monterey, CA 93943-5216
%              23 Jan 2015
%Synopsis: Building Matrix Terms in order to solve the equations for
%the 2D Acoustic Wave Equation.
%Output:  Matrix elements used to solve the RHS in the RK54 scheme.
%------------------------------------------------------------------------%
function [MAT,MAT_ksi,MAT_eta,Je,A,B,C,D,E,F,G,H] = MatrixTerms2D(Ne,
    D_ksi,D_eta,y_eta,y_ksi,x_eta,x_ksi,jac,M)
    for k = 1:Ne
        Je{k} = diag(jac(k,:));
        Jeinv{k} = diag(1./jac(k,:));
```

```matlab
        A = D_ksi'*(diag(y_eta(k,:))*Jeinv{k}*M*diag(y_eta(k,:)))*D_ksi;
        B = D_eta'*(diag(y_ksi(k,:))*Jeinv{k}*M*diag(y_eta(k,:)))*D_ksi;
        C = D_ksi'*(diag(y_eta(k,:))*Jeinv{k}*M*diag(y_ksi(k,:)))*D_eta;
        D = D_eta'*(diag(y_ksi(k,:))*Jeinv{k}*M*diag(y_ksi(k,:)))*D_eta;
        E = D_ksi'*(diag(x_eta(k,:))*Jeinv{k}*M*diag(x_eta(k,:)))*D_ksi;
        F = D_eta'*(diag(x_ksi(k,:))*Jeinv{k}*M*diag(x_eta(k,:)))*D_ksi;
        G = D_ksi'*(diag(x_eta(k,:))*Jeinv{k}*M*diag(x_ksi(k,:)))*D_eta;
        H = D_eta'*(diag(x_ksi(k,:))*Jeinv{k}*M*diag(x_ksi(k,:)))*D_eta;

        MAT{k} = A-B-C+D+E-F-G+H;
        MAT_ksi{k} = A-B+E-F;
        MAT_eta{k} = -C+D-G+H;%%%-C+D-G+H


    end
end
```

```matlab
%-------------------------------------------------------------------%
%Function to build Dx and Dy used to solve the RHS in the RK54 scheme.
%Written by Benjamin Davis
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%            23 Jan 2015
%-------------------------------------------------------------------%
function [Dx,Dy] = DxDy(Ne,ksi_x,ksi_y,eta_x,eta_y,D_ksi,D_eta)
    for k = 1:Ne

        Dx{k} = diag(ksi_x(k,:))*D_ksi + diag(eta_x(k,:))*D_eta;
        Dy{k} = diag(ksi_y(k,:))*D_ksi + diag(eta_y(k,:))*D_eta;

    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Function for building the Surface Jacobians for the faces on a 2D grid
% Written by Benjamin Davis
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%           23 Jan 2015
%-------------------------------------------------------------------------%
function [Sj1,Sj2,Sj3,Sj4] = SurfaceJac2D(Ne,L1,L2,L3,L4,x_ksi,x_eta,
    y_ksi,y_eta)
    for k = 1:Ne
        %Building the surface jacobians for the faces
        Sj1_3{k} = sqrt(x_ksi(k,:).^2 + y_ksi(k,:).^2);
        Sj2_4{k} = sqrt(x_eta(k,:).^2 + y_eta(k,:).^2);
        %Isolate Surface Jacobians individually for four faces
        Sj1{k} = diag(L1*Sj1_3{k}');
        Sj3{k} = diag(L3*Sj1_3{k}');

        Sj2{k} = diag(L2*Sj2_4{k}');
        Sj4{k} = diag(L4*Sj2_4{k}');
    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Function to build the element normals for the fours faces on a 2D grid
%Written by Benjamin Davis
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%           23 Jan 2015
%-------------------------------------------------------------------------%
function [nx_1,nx_2,nx_3,nx_4,ny_1,ny_2,ny_3,ny_4] = ElementNormals2D(Ne
    ,L1,L2,L3,L4,x_ksi,x_eta,y_ksi,y_eta,Sj1,Sj2,Sj3,Sj4)
    for k=1:Ne
        %Compute the Normals per element
```

```matlab
        nx_1{k} = diag(L1*y_ksi(k,:) './diag(Sj1{k}));
        ny_1{k} = -diag(L1*x_ksi(k,:) './diag(Sj1{k}));

        nx_2{k} = diag(L2*y_eta(k,:) './diag(Sj2{k}));
        ny_2{k} = -diag(L2*x_eta(k,:) './diag(Sj2{k}));

        nx_3{k} = -diag(L3*y_ksi(k,:) './diag(Sj3{k}));
        ny_3{k} = diag(L3*x_ksi(k,:) './diag(Sj3{k}));

        nx_4{k} = -diag(L4*y_eta(k,:) './diag(Sj4{k}));
        ny_4{k} = diag(L4*x_eta(k,:) './diag(Sj4{k}));
    end
end
```

```matlab
%-------------------------------------------------------------------------%
%Function provided to Professor F.X. Giraldo's MA4245 class.
%Modified by Asst. Professor Jeremy Kozdon and Benjamin Davis
%This function rotates the grid and plots it.
%Written by F.X. Giraldo on 4/2008
%           Department of Applied Mathematics
%           Naval Postgraduate School
%           Monterey, CA 93943-5216
%
% INPUT LIST: coord are the coordinates
%             intma is the connectivity list
%             npoin are the number of global points
%             nelem are the number of elements
%             ngl is the number of interpolation points in an element
%             (polynomial order + 1)\
%             plot_grid is a switch to either plot or not plot
%             grid_rotation_angle is the grid rotation in degrees
% OUTPUT LIST:
%             coord are the new rotated coordinates: x=coord(:,1)
%              and y=coord(:,2)
%-------------------------------------------------------------------------%
```

```matlab
function [coord_rotated] = rotate_grid_v2(coord,intma,npoin,nelem,ngl,...
    plot_grid,grid_rotation_angle)
nop=ngl-1;
coord_rotated=zeros(npoin,2);
%Rotate Grid
alpha=grid_rotation_angle*pi/180;
for i=1:npoin
 xn=cos(alpha)*coord(i,1) - sin(alpha)*coord(i,2);
 yn=sin(alpha)*coord(i,1) + cos(alpha)*coord(i,2);
 coord_rotated(i,1)=xn;
 coord_rotated(i,2)=yn;
end
%Plot Grid
if (plot_grid == 1)
    x=zeros(5,1);
    y=zeros(5,1);
    figure;
    hold on;
    for e=1:nelem
        for j=1:ngl-1
            for i=1:ngl-1
                i1=intma(e,i,j);
                i2=intma(e,i+1,j);
                i3=intma(e,i+1,j+1);
                i4=intma(e,i,j+1);
                x(1)=coord_rotated(i1,1); y(1)=coord_rotated(i1,2);
                x(2)=coord_rotated(i2,1); y(2)=coord_rotated(i2,2);
                x(3)=coord_rotated(i3,1); y(3)=coord_rotated(i3,2);
                x(4)=coord_rotated(i4,1); y(4)=coord_rotated(i4,2);
                x(5)=coord_rotated(i1,1); y(5)=coord_rotated(i1,2);
                plot_handle=plot(x,y,'-r');
                set(plot_handle,'LineWidth',1.5);
            end
        end
        i1=intma(e,1,1);
        i2=intma(e,ngl,1);
        i3=intma(e,ngl,ngl);
        i4=intma(e,1,ngl);
        x(1)=coord(i1,1); y(1)=coord(i1,2);
```

```matlab
            x(2)=coord(i2,1); y(2)=coord(i2,2);
            x(3)=coord(i3,1); y(3)=coord(i3,2);
            x(4)=coord(i4,1); y(4)=coord(i4,2);
            x(5)=coord(i1,1); y(5)=coord(i1,2);
            plot_handle=plot(x,y,'-b');
            set(plot_handle,'LineWidth',2);
        end
        title_text=['Rotated Grid Plot For: Ne = ' num2str(nelem) ', N = '
        num2str(nop)];
        title([title_text],'FontSize',18);

        xlabel('X','FontSize',18);
        ylabel('Y','FontSize',18);
        axis image
end
```

```matlab
%------------------------------------------------------------------------%
%Function for building the Lagrange Polynomials.
%Written by Benjamin Davis in MA4245 %Created: July 2014
%            Department of Applied Mathematics
%            Naval Postgraduate School
%            Monterey, CA 93943-5216
%------------------------------------------------------------------------%%
function [L,dL] = lagrange_basis(x,z)
%Nth order interpolation
n = length(x);
%Length of the equally spaced grid for k = 1:50
h = length(z);
%Initialize the Lagrange Matrix
L = ones(n,h);
dL = zeros(n,h);
%Computation for Lagrange Matrix
    for k = 1:h
        for i = 1:n
            for j = 1:n
                dl = 1;
```

```matlab
21                    if j ~= i % If j does not equal i
                          %Equation for the Lagrange Polynomial
23                        L(i,k) = (z(k)-x(j))./(x(i)-x(j)) * L(i,k);
                          for l = 1:n
25                            if (l ~= i) && (l ~= j)
                                  dl = dl*(z(k)-x(l))./(x(i)-x(l));
27                            end
                          end
29                        dL(i,k) = dL(i,k) + dl/(x(i)-x(j));
                      end
31                end
            end
33      end
    end
```

```matlab
%------------------------------------------------------------------------%
2 %Code given to F.X. Giraldo MA4245 Class July 2014
  %Used by Ben Davis.
4 %This code computes the Legendre-Gauss-Lobatto points and weights
  %which are the roots of the Lobatto Polynomials.
6 %Written by F.X. Giraldo on 4/2000
  %           Department of Applied Mathematics
8 %           Naval Postgraduate School
  %           Monterey, CA 93943-5216
10 %------------------------------------------------------------------------%
  function [xgl,wgl] = legendre_gauss_lobatto(P)
12 p=P-1;
  ph=floor( (p+1)/2 );
14 for i=1:ph
     x=cos( (2*i-1)*pi/(2*p+1) );
16   for k=1:20
         [L0,L0_1,L0_2]=legendre_poly(p,x);
18       %Get new Newton Iteration
         dx=-(1-x^2)*L0_1/( -2*x*L0_1 + (1-x^2)*L0_2);
20       x=x+dx;
         if (abs(dx) < 1.0e-20)
```

```
22          break
         end
24     end
    xgl(p+2-i)=x;
26     wgl(p+2-i)=2/(p*(p+1)*L0^2);
end
28 %Check for Zero Root
if (p+1 ~= 2*ph)
30     x=0;
    [L0,L0_1,L0_2]=legendre_poly(p,x);
32     xgl(ph+1)=x;
    wgl(ph+1)=2/(p*(p+1)*L0^2);
34 end
%Find remainder of roots via symmetry
36 for i=1:ph
    xgl(i)=-xgl(p+2-i);
38     wgl(i)=+wgl(p+2-i);
end
```

```
1 %-----------------------------------------------------------------------%
%Code given to F.X. Giraldo MA4245 Class July 2014
3 %Used by Ben Davis
%This code computes the Legendre Polynomials and its 1st and 2nd
5 %derivatives
%
7 %Written by F.X. Giraldo on 4/2000
%           Department of Applied Mathematics
9 %           Naval Postgraduate School
%           Monterey, CA 93943-5216
11 %-----------------------------------------------------------------------%
function [L0,L0_1,L0_2] = legendre_poly(p,x)
13 L1=0;L1_1=0;L1_2=0;
L0=1;L0_1=0;L0_2=0;
15 for i=1:p
    L2=L1;L2_1=L1_1;L2_2=L1_2;
17     L1=L0;L1_1=L0_1;L1_2=L0_2;
```

```
    a=(2*i-1)/i;
19    b=(i-1)/i;
    L0=a*x*L1 - b*L2;
21    L0_1=a*(L1 + x*L1_1) - b*L2_1;
    L0_2=a*(2*L1_1 + x*L1_2) - b*L2_2;
23 end
```

List of References

[1] J. S. Hesthaven and T. Warburton, *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. New York, NY: Springer Science & Business Media, 2007, vol. 54.

[2] D. Appelö and T. Hagstrom, "A new discontinuous Galerkin formulation for wave equations in second order form," to appear in *SIAM Journal of Numerical Analysis*.

[3] F. X. Giraldo, "Element-based Galerkin methods," Class Notes for MA4245: Mathematical Principles of Galerkin Methods, Department of Applied Mathematics, Naval Postgraduate School, August 2014.

[4] B. Gustafsson, H. Kreiss, and J. Oliger, *Time Dependent Problems and Difference Methods*. Hoboken, New Jersey: Wiley-Interscience, 1996.

[5] A. Iserles, *A First Course in the Numerical Analysis of Differential Equations*. New York, NY: Cambridge University Press, 1996, vol. 44.

[6] R. J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, ser. Cambridge Texts in Applied Mathematics. New York, NY: Cambridge University Press, 2002.

[7] R. Boucher, "Galerkin optimal control," Ph.D. dissertation, Naval Postgraduate School, Monterey, California, 2014. [Online]. Available: http://hdl.handle.net/10945/44526.

[8] U. M. Ascher and C. Greif, *A First Course on Numerical Methods*. Philadelphia, PA: Siam, 2011, vol. 7.

[9] J. Stewart, *Calculus*. Belmont, CA: Cengage Learning, 2011.

[10] J. E. Kozdon and L. C. Wilcox, "Skew-symmetric splitting for Sommerfield DG," Thesis Advisor Notes, Department of Applied Mathematics, Naval Postgraduate School, September 2014.

[11] M. H. Carpenter and C. A. Kennedy, "Fourth-order 2N-storage Runge-Kutta schemes," National Aeronautics and Space Administration Langley Research Center, Hampton, VA, NASA Tech. Rep. 109112, June 1994.

[12] T. Warburton, "A low storage curvilinear discontinuous Galerkin time-domain method for electromagnetics," in *Electromagnetic Theory (EMTS), 2010 URSI International Symposium on*. IEEE, 2010, pp. 996–999.

[13] T. Warburton, "A low-storage curvilinear discontinuous Galerkin method for wave problems," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. A1987–A2012, 2013.

[14] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media," *Journal of Computational Physics*, vol. 229, no. 24, pp. 9373–9396, 2010.

[15] A. E. Fischer and J. E. Marsden, "The Einstein evolution equations as a first-order quasi-linear symmetric hyperbolic system, I," *Communications in Mathematical Physics*, vol. 28, no. 1, pp. 1–38, 1972.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California