COMPARISION OF CONTINUOUS AND DISCONTINUOUS GALERKIN FINITE ELEMENT METHODS FOR PARABOLIC PARTIAL DIFFERENTIAL EQUATIONSWITH IMPLICIT TIME STEPPING

by

Garret Dan Vo

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Mechanical Engineering

MONTANASTATEUNIVERSITY
Bozeman, Montana

April 2012

APPROVAL

of a thesis submitted by

Garret Dan Vo

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency and is ready for submission to The Graduate School.

Dr. Jeffrey Heys

Approved for the Department of Mechanical & Industrial Engineering

Dr. Christopher Jenkins

Approved for The Graduate School

Dr. Carl A. Fox

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for amaster's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" a prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Garret Dan Vo

April 2012

ACKNOWLEDGEMENTS

This work could not be completed without support from Idaho National Laboratory and the Flight Attendant Medical Research Institute. There are not many words to express my appreciation to many individuals in Montana, who have given me directions and advices for me to become a better person, such as Erwin, Adrielle, and Tanner. First, I would thank Dr. Heys, who is always available to answer my questions and discuss career choices with me. In addition, I appreciate him for giving me opportunities to participate in computational fluid dynamics. In addition to Dr. Heys, I want to thank Dr. Davis in the mathematics department for useful discussions about the topic. I want to thank my lab partner, Prathish, for giving me the data from his finite element code for me to complete this work. After living in Montana for six years, I would like to thank countless individuals who have given me chances to experience these amazing life experiences. I want to thank my fraternity, Sigma Phi Epsilon, for the support during my study here at Montana State University. I want also thank the taekwondo association of Montana with many great instructors, such as Master Williams, Notes, Rosbarsky and my friends Mark Austin and Clay Hunt. They have taught me to be humble and to have fighting spirit. In addition, I want to express my appreciation to my mentors, Dr. Combs and Dr. Funk, who have shown me gratitude and directions for career perspective. Lastly, I want to delicate this thesis to a good friend of mine, Lucy Ronning and my mother. I want to rephrase a quote from my taekwondo instructor “A master degree is not the end; it is a beginning of the journey.”

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

LIST OF FIGURES CONTINUED

Figure Page

ABSTRACT

A number of different discretization techniques and algorithms have been developed for approximating the solution of parabolic partial differential equations. A standard approach, especially for applications that involve complex geometries, is the classic continuous Galerkin finite element method. This approach has a strong theoretical foundation and has been widely and successfully applied to this category of differential equations. One challenging sub-category of problems, however, are equations that include an advection term that is large relative to the second-order, diffusive term. For these advection dominated problems, the continuous Galerkin finite element method can become unstable and yield highly inaccurate results. An alternative to the continuous Galerkin finite element method is the discontinuous Galerkin finite element method, and, through the use of a numerical flux term used in deriving the weak form, the discontinuous approach has the potential to be much more stable in highly advective problems. However, the discontinuous Galerkin finite element method also has significantly more degrees-of-freedom due to the replication of nodes along element edges and vertices. The work presented here compares the computational cost, stability, and accuracy (when possible) of continuous and discontinuous Galerkin finite element methods for four different test problems, including the advection-diffusion equation, viscous Burgers' equation, and the Turing pattern formation equation system. The comparison is performed using as much shared code as possible between the two algorithms and direct, iterative, and multilevel linear solvers. The results show that, for implicit time stepping, the continuous Galerkin finite element method is typically 5-20 times less computationally expensive than the discontinuous Galerkin finite element method using the same finite element mesh and element order. However, the discontinuous Galerkin finite element method is significantly more stable than the continuous Galerkin finite element method for advection dominated problems and is able to obtain accurate approximate solutions for cases where the classic, un-stabilized continuous Galerkin finite element method fails.

CHAPTER 1

BACKGROUNDS AND MOTIVATIONS

Partial differential equations arise in many areas in applied mathematics, physics, chemistry, and engineering. Partial differential equations are used to model different physical and engineering problems, such as quantum mechanics, continuum mechanics and electrodynamics. Partial differential equations are classified in three different categories, parabolic, hyperbolic, and elliptic. The classification provides a guide to set initial and boundary conditions, and the smoothness in the solution [1, 2].

Parabolic partial differential equations describe whole families of problems in science and engineering, such as heat transport, acoustic propagation, and mass transport. A partial differential equation of the form

$$Au_{xx} + Bu_{xy} + Cu_{xy} + Du_x + Eu_y + F = 0 \qquad (1)$$

is parabolic if

$$B^2 - 4AC = 0\,. \qquad (2)$$

A simple, characteristic example of a parabolic partial differential equation is the heat equation:

$$\frac{\partial u}{\partial t} = k\nabla^2 u \qquad (3)$$

where $u(\boldsymbol{x}, t)$ is the temperature, and k is the thermal conductivity [3].

Hyperbolic partial differential equations have been used to model phenomena in many different applications including mechanics, hydrodynamics, and traffic flow. The

solution of a hyperbolic partial differential equation has a wave-like behavior. A partial differential equation of the form

$$Au_{xx} + Bu_{xy} + Cu_{xy} + Du_x + Eu_y + F = 0 \qquad (4)$$

is hyperbolic if

$$B^2 - 4AC > 0\ . \qquad (5)$$

A simple, characteristic example of hyperbolic partial differential equation is the wave equation in gas dynamics:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u \qquad (6)$$

where $u(x,t)$ is the density or condensation of the gas, and c is the propagation speed of the wave [4].

Elliptic partial differential equations have a similar form to parabolic partial differential equations, but they are typically time independent. Elliptic partial differential equations often arise when modeling phenomena at steady-state or approaching an asymptotic limit. A partial differential equation of the form

$$Au_{xx} + Bu_{xy} + Cu_{xy} + Du_x + Eu_y + F = 0 \qquad (7)$$

is elliptic if

$$B^2 - 4AC < 0\ . \qquad (8)$$

A simple, characteristic example of elliptic partial differential equation is Poisson's equation:

$$\nabla^2 u = f \qquad (9)$$

where the temperature at steady-state is $u(x,t)$, and f is the flux [5].

Though there are several analytical methods to solve partial differential equations, such as separation of variables, method of characteristics, etc., these methods are normally unable to solve non-linear partial differential equations or linear partial differential equations with complex domain shapes or unusual boundary conditions. Thanks to advancements in digital computers, numerical methods can be used to solve these partial differential equations to approximate the solution for scientists and engineers. There are three main categories of numerical schemes to solve partial differential equations: finite difference, finite volume and finite element methods. Each method has advantages and disadvantages depending on the specific problem. For example, finite difference methods fail when there is a complex geometry, but finite volume methods can handle this issue. However, finite volume methods are extremely difficult to use if a high-order accuracy solution is desired. For parabolic and hyperbolic equations, the approximation method can discretize the temporal derivative either implicitly or explicitly with regards to the time stepping.

This research focuses on parabolic partial differential equations. The parabolic equation typically has second-order derivatives with respect to space and a first-order derivative with respect to time. There is also a restrictive condition on the size of the time step (i.e., the CFL stability condition) if explicit time stepping is used [6]. The spatial discretization can be the finite difference, finite volume or finite element method, but our research focuses on the discontinuous Galerkin finite element method.

The Finite Element Method

The finite element method is a numerical method like the finite difference or finite volume method. It is a more general framework because it can involve complex geometries, physics, and boundary conditions [7,8,9]. In the finite element method, a given domain is represented by a collection of sub-domains, and each sub-domain has an approximate solution obtained from a finite dimensional space, which is determined by a variational method. The main reason to seek an approximate solution on each sub-domain is that it is often possible to represent a complex function relatively accurately using a collection of simple polynomials [8, 11]

The Basic Features

The finite element method has three distinctive characteristics that can provide advantages over the finite difference and finite volume methods. First, a geometrically complex domain,Ω, of the problem, such as that shown figure 1

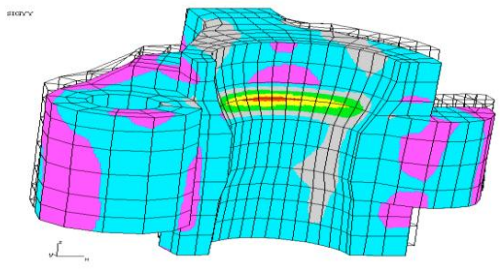

Figure 1: The complex domain is divided into different sub-domains and refined with elements (used without permission [10])

is represented as simple collection of sub-domains, called finite elements. Second, each domain is governed by relationships between the quantities of interest, such as stresses, strains, velocity, and pressure. Third, the elements are assembled (i.e. elements are put back into their original positions in the domain) using certain inter-elements relationship [7, 11, 12].

There are several ways to formulate equations for each element, such as creating the stiffness matrix based on Hooke's Law or a variation method (i.e. Galerkin method, Ritz, collocation, etc.). Since most physical phenomena are modeled using differential equations, the variation method may be a natural and rigorous way to approximate the solution to these differential equations and boundary conditions [7, 13]. The discussion below illustrates why the integral formulation is needed

The Need for the Integral Formation

In many approximation methods used to determine the solution of differential and/or integral equations, we seek the approximate solution in the form of

$$u(x) \approx U_N(x) = \sum_{j=1}^{N} c_j \phi_j(x) \tag{10}$$

where u is the solution of particular differential equation with associated boundary conditions.U_N is the approximation of u, and is expressed as a linear combination of unknown parameters c_j and known functions ϕ_j of position x in the domain Ω on which the problem is posed. The approximation of U_N is completely known, when the c_j are known. Therefore, we must find a means to determine c_j such that $U_N$satisfies the

equations governing u [7, 14]. An example, adapted from Reddy's finite element book [7], is used to illustrate the method.

Consider a differential equation

$$\frac{-d}{dx}\left[a(x)\frac{du}{dx}\right] + c(x)u = f(x)\ for\ 0\ < x < L \tag{11}$$

is subjected to the boundary conditions

$$u(0) = u_0 \text{ and } \left[a(x)\frac{du}{dx}\right]_{x=L} = Q_0 \tag{12}$$

where $a(x)$, $c(x)$, and $f(x)$ are known functions; u_0 and Q_0 are parameters.

We seek to approximate the solution $u(x)$ in the form

$$U_N = \sum_{j=1}^{N} c_j \phi_j(x) + \phi_0 \tag{13}$$

where c_j is going to be determined by chosen functions ϕ_j and ϕ_0 such that the boundary conditions of the problem are satisfied by the N-parameter approximate solution U_N.Theseϕ_j functions are typically referred to as shape or basis functions. The overall approximation of the solution can be improved by increasing the number,N, of basis functions used [11, 14, 15].

Assume $L\ =\ 1$, since ϕ_0 satisfies the boundary conditions, the sum $\sum c_j\ \phi_j$must satisfy, for arbitrary c_j, the homogenous form of the boundary conditions. For instance, $Bu\ =\ u$ is said to be a non-homogenous boundary condition when $u\ \neq\ 0$, and it is termed a homogenous boundary condition when $u\ =\ 0$; here B is an operator. Therefore, ϕ_0 and ϕ_j satisfy the condition

$$B\phi_0 = u\ and\ B\phi_j = 0 \text{ for all } j\ =\ 1, 2, \ldots, n\ . \tag{14}$$

For simplicity, substitute in $L = 1, u_0 = 1, Q_0 = 0, a(x) = x, c(x) = 1, f(x) = 0$, and $N = 2$. Then we choose the approximate solution

$$U_2 = c_1\phi_1 + c_2\phi_2 + \phi_0 \tag{15}$$

with $\phi_0 = 1, \phi_1 = x^2 - 2x,\ \phi_2 = x^3 - 3x$

that satisfies the boundary conditions of the problem for any values of c_1 and c_2 because

$$\phi_0(0) = 1, (x\frac{d\phi_0}{dx})_{x=1} = 0\,, \tag{16}$$

$$\phi_j(0) = 0, (\frac{d\phi_j}{dx})_{x=1} = 0\ for\ j = 1,2\,.$$

To make U_2 satisfy the differential equation, we must have

$$\begin{aligned}\frac{-dU_2}{dx} - x\frac{d^2U}{dx^2} + U_2 \\ = -2c_1(x-1) - 3c_2(x^2-1) - 2c_1x \\ - 6c_2x^2c_1(x^2-2x) + c_2(x^3-3x) + 1 = 0\,.\end{aligned} \tag{17}$$

Since this expression must be zero for any x, the coefficientsc_j's for various power of x must be zero. Therefore, a set of four equations is obtained

$$1+2c_1 + 3c_2 = 0 \tag{18}$$

$$-6c_1-3c_2 = 0$$

$$c_1 - 9c_2 = 0$$

$$c_2 = 0$$

The system of equations above has two unknowns and four equations; therefore, there are no solutions for $c_1$and c_2, which means that we cannot find the approximate solution for equation (11).An alternative method is needed to obtain the approximate solution,U_N, to satisfy the differential equation (11).

The integral formulation gives

$$\int_0^1 w(x)Rdx = 0 \tag{19}$$

where R is the left-hand side of equation (11), which is

$$R = -\frac{dU_N}{dx} - x\frac{d^2U_N}{dx^2} + U_N \tag{20}$$

and $w(x)$is called the weight function. Since $N = 2$, the number of weight functions must be restricted to be two in order to have exact the same number of equation as the number of unknown coefficients c_j's. According to Reddy, the chosen weight functions are 1 and x. Substituting the chosen weight functions into the integral equation (19) and following the same procedures as before, we obtain the coefficient $c_1 = \frac{222}{23}$ and $c_2 = \frac{-100}{23}$. With c_1 and c_2 known, we can approximate the solution U_N to satisfy the differential equation (11). The next section will introduce the most popular method to choose the weight functions, which ensures the approximate solution satisfies the differential equation.

The Galerkin Finite Element Method

There are a number of ways to choose the weight functions, such as the Ritz formulation, collocation, and least-square methods, but the most popular choice for the weight function is to just use the basis functions. This choice has a computational advantage over other methods because the matrix for the system of linear equations to solve for the c_j coefficients is symmetric [11, 16, 18].

This choice is called the Galerkin form or Galerkin finite element method. An example below from Mitchell and Wait [17] will illustrate the Galerkin method. Starting with a differential equation:

$$\frac{\partial}{\partial x}\left(p(u)\frac{\partial U}{\partial x}\right)+\frac{\partial}{\partial y}\left(q(u)\frac{\partial u}{\partial y}\right)+f(x,y)=0\,, \tag{21}$$

and multiplying both sides by the weight function (basis function) ϕ_j and integrating by parts gives

$$\iint\{p(U)(\tfrac{\partial U}{\partial x})(\tfrac{\partial \phi_j}{\partial x})+q(U)(\tfrac{\partial U}{\partial y})(\tfrac{\partial \phi_j}{\partial y})\}dxdy=(f,\phi_j). \tag{22}$$

These basis functions can be evaluated at the Gaussian quadrature points to facilitate the evaluation of the integrals later on. However, there are weaknesses in the Galerkin finite element method, including a mass matrix that is expensive to invert (unless it is lumped), and instabilities that arise in the approximate solution when information flows strongly in a specific direction (i.e. advection or convection dominated problems). These instabilities are common when the equation is a hyperbolic partial differential equation, but they can occur in parabolic partial differential equations when the first-order advection term is relatively large comparing to the second-order term in the equation. The instability can be addressed by Petrov-Galerkin method, which typically adds an artificial diffusion (or thermal conductivity) to control the instability [19].

The finite element method's approximate solution can suffer from instability in convection or advection dominated problem, but the finite volume method tends not to suffer from these limitations. However, the finite volume method is difficult to extend to a higher-order spatial approximation. An alternative approach is to try to combine the

strengths of both the finite element and finite volume methods, and the discontinuous Galerkin finite element method was developed as a result of efforts to achieve this best of both worlds combination.

Discontinuous Galerkin Method

The Discontinuous Galerkin (DG) method was first proposed and analyzed in the 1970's [20]. In 1973, Reed and Hill [21] used DG to solve a hyperbolic neutron transport equation. This approach retains the definition of elements, but the basis functions are only defined locally on each element [22, 23, 24]. The DG method uses these basis functions to formulate the Galerkin approximation. Given a mesh of a domain, DG approximates the solution within each element by a linear combination of these basis functions. For a pair of adjacent elements, the approximate solution in the interior of these elements does not need to agree on their common boundary. An example, adapted from Heshaven and Warburton [25], will briefly illustrate how the DG method works.

Consider an equation

$$\frac{\partial u}{\partial t} + \frac{\partial (au)}{\partial x} = 0 \; x \in [L, R] = \Omega \tag{23}$$

where the equation is subjected to the initial condition

$$u(x, 0) = u_0(x), \tag{24}$$

and the boundary conditions are given as:

$$u(L, t) = g(t) \; if \; a \geq 0 \tag{25}$$

$$u(R, t) = g(t) \; if \; a \leq 0 \, .$$

We approximateΩ by K overlapping elements. On each of these elements, the solution $u(x,t)$ of the equation (23) is expressed as a polynomial of order $N = N_P - 1$ as follows:

$$u(x,t) = \sum_{n=1}^{N_P} u_n^k(t)\psi_n(x) = \sum_{i=1}^{N_P} u_h^k\left(x_i^k, t\right)\phi_i^k(x) \tag{26}$$

where N_P is the number local grid points. In the expression for the approximation of $u(x,t)$, there are two formsfor the basis functions. Multiplying both sides of the equation (23) by ϕ and integrating by parts over the entire domain to obtain the weak-form, we get:

$$\int \left(\frac{\partial u_h^k}{\partial t}\phi_n - a u_h^k \frac{d\phi_n}{dx}\right) dx = -\int \hat{n} * a u_h^k \phi_n dx \tag{27}$$

$$1 \leq n \leq N_P$$

where $\hat{n}$ is the local outward normal unit vector and its value is 1 or -1 at the right and left interface, respectively.

Since there is a discontinuity in the solution, the approximated solution between the interfaces must be defined. We use the common term ‘numerical flux’ for the approximated solutions at the interfaces. The specific form of the numerical flux is most naturally related to the dynamics of the partial differential equation being solved. At the left of the domain, the numerical flux in equation (1) should be a function of $[au_h^{k-1}(x_r^{k-1}), au_h^k(x_l^k)]$, while the right end depends on $[au_h^k(x_r^k), au_h^{k+1}(x_l^{k+1})]$. At the boundary, a reasonable but not unique choice is $(auh)^* = ag(t)$ [21].

Therefore, equation (23) will lead to the semi-discrete scheme:

$$\int \left(\frac{\partial u_h^k}{\partial t} \phi_n - a u_h^k \frac{d\phi_n}{dx} \right) dx = - \int \hat{n} * (a u_h)^* \, \phi_n dx \, . \qquad (28)$$

$$1 \leq n \leq N_P$$

Some DG approaches use the strong-form by integrating by part again for equation (28), because the right-hand side of the strong-form is responsible for recovering the global solution from the local solutions and imposing the boundary conditions. In addition, the strong-form does not require any smoothness of the basis functions. For example, the basis functions can include a delta-function. The strong-form for equation (23) is going to be

$$\int (\frac{\partial u_h}{\partial t} + \frac{\partial (a u_h)}{\partial x}) \phi_n(x) dx = \int \hat{n} * (a u_h^k - (a u_h)^*) \phi_n dx \qquad (29)$$

$$1 \leq n \leq N_P.$$

The DG method has desirable properties that have helped it to become popular, such as high order basis functions that are straight forward to build, capturing the discontinuity in the solution, the mass matrix consists of unconnected, local element matrices, and numerical fluxes that can lead to stable upwind schemes for problems with highly directional information flows.

Motivation

For parabolic partial differential equations, small values of the coefficient associated with the second-order differentiation term will potentially cause large gradients that are close to a discontinuity in the solution. For example, a smaller viscous term in the Burger's equation will lead to a large gradient in the solution, and a shock is

formed in the solution when the viscous term is zero. With the small value, the classical continuous Galerkin finite element method becomes unstable and fails to capture the large gradient or shock in the solution, but the discontinuous Galerkin finite element method will be stable.

For example, imagine a fuel rod inside a nuclear reactor that becomes hot. The cooling fluid around the rod may boil, and the coolant flow around the rod will become advection dominated. Existing continuous finite element algorithms do not always work well in this situation. The simulation of the coolant flow is often coupled to the simulation of other physical phenomena in a multi-physics simulation, and these multi-physics algorithms often require implicit time stepping. Most of the physical phenomena being modeled are based on parabolic partial differential equations.

The discontinuous Galerkin finite element method has historically been mostly used for hyperbolic partial differential equations. When the method is applied, the temporal discretization is typically based on an explicit time stepping scheme such as the Runge-Kutta methods. Sometimes, DG methods are used with semi-implicit time stepping [25], but implicit time stepping is extremely rare.

In nuclear reactor simulations, multi-physics modeling is critical in order to design a safe reactor. When there are large gradients within the solution, a multi-physics simulation using a continuous finite element method often fails to accurately simulate the result. The objective here is to implement the discontinuous Galerkin finite element method using implicit time stepping and compare the computational costs and accuracy with standard, continuous finite element methods.

For parabolic partial differential equations with implicit time stepping, the stability advantages of using a numerical flux are only realized for highly advection problems. In fact, the proper form for the numerical flux can often be challenging to determine. The local block mass matrix is less of an advantage because the solving of a large linear matrix is unavoidable. Finally the increased number of degrees of freedom when using DG method can be a liability because a linear matrix problem must be solved. In many ways, parabolic partial differential equations with implicit time stepping are ill-suited for approximation using discontinuous Galerkin finite element method.

However, the question we seek to answer is when the discontinuous Galerkin finite element method should be considered for solving parabolic differential equations with implicit time stepping. We seek to quantitatively answer this question by comparing discontinuous and continuous Galerkin finite element method in terms of accuracy and computational time for a number of common parabolic partial differential equations. There are, of course, a number of limitations when trying to perform a quantitative comparison of two different numerical algorithms. The biggest limitation is implementing both algorithms in a fair way. It is easy to make one approach look better than another approach by simply implementing one method more carefully. We have made every effort to avoid this situation by having the two algorithm share as much code as possible, but we believe it is impossible to achieve a perfectly fair comparison. Certainly, experts could examine the thousands of lines of code in each algorithm and identify small improvements. A second major limitation is that every problem, every parabolic partial differential equation is different, and it is practically impossible to

perform a comparison for all possible parabolic equations. We have selected three characteristic parabolic partial differential equation equations that hopefully form a foundation from which the performance for other parabolic partial differential equations can be estimated.

Other Approaches

Since the discontinuous Galerkin finite element method has been around since the 1970s, there are many mathematicians and engineers studying and applying the method to solve their problems. Gabard et al. [26] have done intensive studies on the method, and use it to solve hyperbolic partial differential equation, particularly the wave equation.

Cockburn [27, 28, 29, 30] has used the method to solve convection-dominated problems, but their problems have periodic boundary conditions, and they use the explicit Runge-Kutta numerical schemes to solve the time derivative part of the problem. Our algorithm has a flow with high advection velocity throughout the domain, and it uses implicit time stepping so that much larger time steps can be taken to minimize computational cost. In addition, the order of approximating polynomials is much less in the work presented here than the order of polynomials in Cockburn [27, 28, 29, 30].

For parabolic partial differential equation, Werder et al. [31] studied discontinuous Galerkin method for parabolic partial differential equation. The paper lists the mathematics, numerical schemes, and parallelization methods for discontinuous Galerkin method, but the paper does not compare discontinuous and continuous methods. For example, as is shown later, it is useless to use the discontinuous Galerkin method on

the advection-diffusion equation when the diffusivity's coefficient is relatively large, because the method will take a significantly larger amount of computational time to run. Instead, the continuous Galerkin method can handle the problem efficiently and yields the same result as the discontinuous Galerkin method. In the result section, the computational cost difference will be shown more explicitly.

Our choice of the basis function and numerical fluxes are similar to Wirasaset et al. [32]. His paper used an explicit-time stepping scheme to solve the problem, and the author needed to respect the CFL stability conditions. Our numerical scheme is fully implicit, and the temporal stability condition is not the concern for us. The algorithm presented here can take time steps of any size.

In our research, we use the interior penalty method for the numerical flux, which is well established by Arnold [33, 34] and Heshaven [25] and described in the Methods chapter. Previous research has focused on alternative forms for the numerical flux, but those results are too extensive to summarize here. Instead, we only note that the interior penalty method has become the standard form for the numerical flux, which is why it was chosen for the work presented in this thesis.

Olson [35] proposed a multi-grid method for the discontinuous Galerkin matrix problem, and they obtained optimal scalability by adjusting some of the methods parameters. In the present work, the multi-grid method is used at a basic level without parameter optimization in order to have a fair comparison between the continuous Galerkin and discontinuous Galerkin finite element methods.

Alternative methods to the discontinuous Galerkin finite element method for advection dominated problems include the Petrov-Galerkin method [51]. This approach was developed by the finite element community to handle problems with a small viscosity term using continuous finite elements. In this approach, artificial viscosity is added primarily in the advection dominated direction in order to make the viscosity parameter large enough for the method to be stable.

CHAPTER 2

THE DISCONTINUOUS GALERKIN METHOD

The discontinuous Galerkin finite element method has been used since the Reed and Hill's paper on neutron transport [21]. This chapter will give details about the discontinuous Galerkin finite element method and linear solvers. In addition, it will also provide details regarding the computational implementation that is used to compare the continuous and discontinuous Galerkin finite element methods.

The Discontinuous Galerkin Finite Element Method

The discontinuous Galerkin finite element method is a numerical technique for solving partial differential equations when there are discontinuities or jumps in the solution or highly advective flows. According to Heshaven [25, 36], the discontinuous Galerkin method has several important properties:

- The solutions are piecewise smooth, often polynomial, but discontinuous between elements.
- Boundary conditions and interface continuity are enforced only weakly.
- All operators are local.
- The schemes are well suited to variable order and element sizes, as all information exchange is across the interface only.

For simplicity, the discontinuous Galerkin finite element method is going to be described in one dimension.

The Discontinuous Galerkin Finite Element Method in One Dimension

From Heshaven [25], we assume that the global solution is a direct sum of the local piecewise polynomial solution as

$$u(x,t) \cong u_h(x,t) = \sum_{k=1}^{K} u_h^k(x^k,t). \qquad (30)$$

Equation (30) does not address what exactly happens at the element interfaces. Later on, we will use equation (30) to reflect that the global solution is obtained by combining the K local solutions, where K is the number of element.

The local solutions are assumed to be of the form

$$x \in D^k = \left[x_l^k, x_r^k\right]$$

$$u_h^k(x,t) = \sum_{n=1}^{N_P} \hat{u}_n^k(t)\psi_n(x) = \sum_{n=1}^{N_P} u_h^k(x_i^k,t)\phi_i^k(x) \qquad (31)$$

where N_P is the number of local grid points. In the expression for the approximation of $u(x,t)$, there are two formsfor the basis functions. The first one is known as the modal form, and it is expressed as the product of the approximated solution and the local polynomial basis, ψ. In the alternative form, the approximated solution is multiplied by the nodal polynomial, ϕ. Like the Galerkin finite element method, these basis functions are evaluated at the Gaussian quadrature points by performing the affine mapping from

the reference values [-1, 1] to the nodal values. The one dimensional affine transformation is

$$x \in D^k$$

$$x(r) = x_l^k + \frac{1+r}{2} h^k \, , \, h^k = x_r^k - x_l^k \tag{32}$$

with reference variable $r \in [-1, 1]$. After the transformation, the local polynomial solution becomes

$$x \in D^k$$

$$u_h^k(x(r), t) = \sum_{n=1}^{N_P} \hat{u}_n^k(t)\psi_n(r) = \sum_{n=1}^{N_P} u_h^k(x_i^k, t)\phi_i^k(r) \, . \tag{33}$$

First, let us discuss the modal approximation.

$$u_h^k(x(r), t) = \sum_{n=1}^{N_P} \hat{u}_n^k(t)\psi_n(r) \tag{34}$$

The natural choice for $\psi_n(r)$ is r^{n-1}. The next question is how to retrieve $\hat{u}_n^k(t)$, and the common method is to multiply both sides of equation (34) by $\psi_m(r)$ and integrate from -1 to 1 – a reference element, which is written

$$\big(u(r), \psi_m(r)\big) = \sum_{n=1}^{N_P} \hat{u}_n^k(t)(\psi_n(r) \, , \psi_m(r)) \tag{35}$$

Where $(u, v) = \int_{-1}^{1} uvdx$

Equation (34) will yield

$$M\hat{u} = u \tag{36}$$

Where $M_{ij} = \big(\psi_i, \psi_j\big)$, $\hat{u} = [\hat{u}_1, \dots\dots, \hat{u}_{N_P}]^T$, $u_i =$

$$(u, \psi_i)$$

which leads to N_P equations for the N_P unknown expansion coefficients, u_i. However,

$$M_{ij} = \frac{1}{i+j-1}[1 + (-1)^{i+j}] \tag{37}$$

is a Hilbert matrix, which is known for being poorly conditioned. Using this basis, it is impossible to obtain an accurate result for $\hat{u}$, and there is not a good approximate representation for u.

The alternative choice is to seek an orthonormal basis that is more suitable and computationally stable. For simplicity, we take $r^n$for ψ, and recover an orthonormal basis from an L^2-based Gram-Schimdt approach. From [36], the orthonormal basis is

$$\psi_n(r) = \tilde{P}_{n-1}(r) = \frac{P_{n-1}(r)}{\sqrt{\lambda_{n-1}}} \tag{38}$$

where $P_n(r)$ is the Legendre polynomial of order n and

$$\lambda_n = \frac{2}{2n+1}.$$

For $\hat{u}_n = (u, \psi_n)$, it can be computed using the Gaussian quadrature points in the form of

$$\hat{u}_n \cong \sum_{i=1}^{N_P} u(r_i)\tilde{P}_{n-1}(r_i)w_i \tag{39}$$

where r_i is the quadrature point, and w_i is the quadrature weight, but this method requires an interpolation scheme between the grid and quadrature points. For the interpolation process, define $\hat{u}_N$ such that the approximation is

$$u_n(\xi_i) \cong \sum_{i=1}^{N_P} \hat{u}_n\tilde{P}_{n-1}(\xi_i) \tag{40}$$

where ξ_i is a set of $N_P$distinct grid points, and they do not need to be associated with quadrature points.

To start, we introduce the Vandermonde matrix, which plays a critical role in deriving differential operators, because the Vandermonde matrix can evaluate a polynomial at a set of points, and it is widely used in polynomial interpolation [39]. The Vandermonde matrix establishes the connection between the modes, $\hat{u}_N$, and the nodal value, u_N. It transforms equation (39) into

$$V\hat{u} = u \tag{41}$$

here $V_{ij} = \tilde{P}_{n-1}(\xi_i)$, $\hat{u} = \hat{u}_i$, $u = u_i(\xi_i)$

It is best to choose quadrature points that will ensure the Vandermonde matrix is well-conditioned. According to Heshaven [25], optimal quadrature points are the Legendre-Gauss-Lobatto points. Since our algorithm focuses on two-dimensional problems, the calculation of Legendre-Gauss-Lobatto quadrature points will be discussed in more details in the two-dimensional section.

Beyond One Dimension in Discontinuous Galerkin Finite Element Method

In this section, we will use the equation below to illustrate the steps to formulate the discontinuous Galerkin method. The conservation law is written as

$$\frac{\partial u(x,t)}{\partial t} + \nabla * \mathrm{f(u(x,t),x,t)} = 0 x \in \Omega \in R^2 , \tag{42}$$

$$u(x,t) = g(x,t)\ x \in \partial\Omega ,$$

$$u(x,0) = f(x).$$

To secure geometrical flexibility, we assume that Ω is a domain divided into K triangular elements.

$$\Omega \cong \Omega_h = \bigcup_{k=1}^{K} D^k$$

where D^k a straight-sided or normal triangle, and the triangle is assumed to be geometrically conforming. $\partial\Omega$, the triangle boundary, is approximated by a liner polygon with each line segment being a face of a triangle.

With the discontinuous Galerkin finite element method, we multiply equation (42) by the test function and integrate it by parts once to get the weak form of the equation (42).

$$\int \left[\frac{\partial u_h^k}{\partial t} l_i^k(x) - f_h^k * \nabla l_i^k(x)\right] dx = -\int \hat{n} * f^* l_i^k(x) dx \tag{43}$$

and we integrate again to obtain the strong form of the equation (42)

$$\int \left[\frac{\partial u_h^k}{\partial t} l_i^k(x) + \nabla * f_h^k(x)\right] l_i^k(x) dx = -\int \hat{n} * [f_h^k - f^*] l_i^k(x) dx \tag{44}$$

where f^* is a numerical flux to be determined later. The later sections will discuss how to implement the discontinuous Galerkin method to solve equation (43) or (44).

Two-Dimensional Basis Functions

The local solution approximation for two-dimensional problems is similar to the one-dimensional problem, including the expression in equation (29). In contrast to the

one-dimensional case, N_P is the number of terms in the local expansion, and N is the order of the polynomial. They are related by

$$N_P = \frac{(N+1)(N+2)}{2} \tag{45}$$

Our algorithm uses a triangular mesh to approximate our domain. The mapping,Ψ, connecting the general straight-sided triangle or normal triangle with the standard triangle or reference triangle is defined as in figure below. The equation for the standard triangle is defined as

$$L = \{r = (r,s) | (r,s) \geq -1; r + s \leq 0\}\,. \tag{46}$$

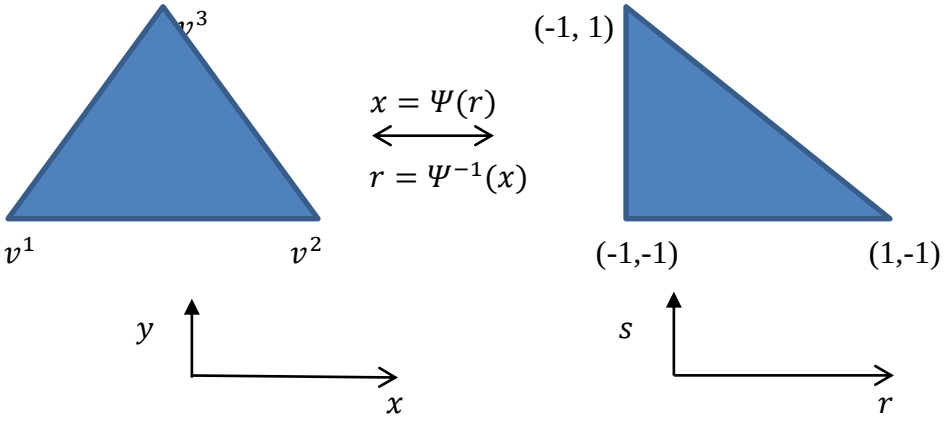

Figure 2: The transformation between normal and right triangles.

The notation v^n indicates the vertices' coordinate values of the triangle. Let x, y be coordinates of the standard triangle, and let r, s be coordinates of the straight-sided triangle. According to Heshaven [25], the direct mapping,Ψ, is

$$x = -\frac{r+s}{2}v^1 + \frac{r+1}{2}v^2 + \frac{s+1}{2}v^3 = \Psi(r) \tag{47}$$

The metric for the mapping can be found directly since

$$\frac{\partial x}{\partial r}\frac{\partial r}{\partial x} = \begin{bmatrix} x_r & x_s \\ y_r & y_s \end{bmatrix}\begin{bmatrix} r_x & r_y \\ s_x & s_y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \quad (48)$$

Therefore,

$$(x_r, y_r) = x_r = \frac{v^2 - v^1}{2},\ (x_s, y_s) = x_s = \frac{v^3 - v^1}{2} \quad (49)$$

yields

$$r_x = \frac{y_s}{J},\ r_y = -\frac{x_s}{J},\ s_x = -\frac{y_r}{J},\ s_y = \frac{x_r}{J} \quad (50)$$

where $J = x_r y_s - x_s y_r$ is the Jacobian.

Through this mapping, we are able to focus on the development of polynomials and operators.

We start with equation (29) and use the similar transformation as in one-dimensional case to obtain equation (39) with the Vandermonde matrix. In the two-dimensional case, the $u$vector

$$u = [u(r_1), \ldots\ldots, u(r_{N_P})] \quad (51)$$

represents the grid point values.

To ensure stable numerical behavior of the Vandermonde matrix, V, we need to address the basis functions and node locations that lead to good behavior of the interpolating polynomial. The first issue is easy to resolve because equations (35) and (36), in the one-dimensional section, gave us a rough idea how to choose the basis functions. Based on equations (35) and (36), the resulting basis function is

$$\psi_m(r) = \sqrt{2}P_i(a)P_j^{(2i+1,0)}(b)(1-b)^i \quad (52)$$

for $\psi_m(r) = r^i s^i (i,j) \geq 0; i + j \leq N$

$$m = j + (N+1)i + 1 - \frac{i}{2}(i-1)$$

where $a = 2\frac{1+r}{1-s} - 1, b = s$

and $P_n^{(\alpha,\beta)}(x)$ is the n-th order Jacobi polynomial, and $\alpha = \beta = 0$ will give the Lendgre polynomial. Equations (48) and (49) are the polynomial basis functions used in our algorithm, but more general polynomials are discussed in Heshaven [25] and Kaneko et al [40].

It is challenging to find optimal locations for the$N_P$points for interpolation on the triangle, because the basis functions along the edge can be oscillatory inside the triangle if the locations are chosen naively. As a result, it could give a large error in the interpolation. However, at the end of the one-dimensional section, we mention the Legendre-Gauss-Lobatto points as integration points to evaluate these basis polynomials. In this section, we introduce a function $w(r)$, which is an N^{th}order polynomial approximation and is used to map the equidistant grid points to the Legendre-Gauss-Lobatto points. The warping function is written as

$$w(r) = \sum_{i=1}^{N_P} \left(r_i^{Legendge-Gauss-Lobatto} - r_i^e\right) l_i^e(r) \qquad (53)$$

where $l_i^e(r)$ is the Lagrange polynomials based on r_i^e, and $r_i^e$are the grid points. In other words, the Lagrange polynomial is used to approximate the solution on the nodal grid points and the Legendre polynomial is approximating the solution on the modal grid.

To utilize this, consider a symmetric equilateral triangle as shown in Figure 3.

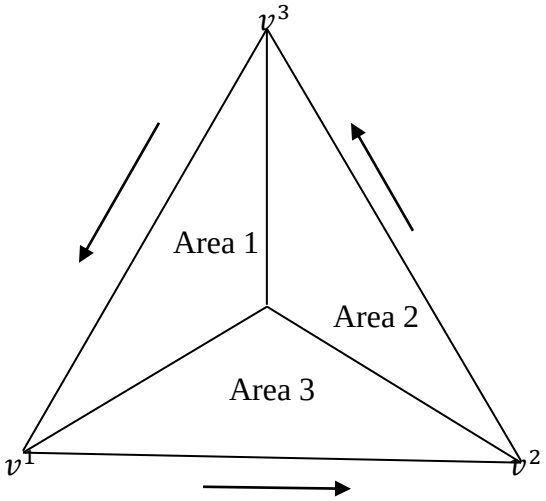

Figure 3: symmetric equilateral triangle with vertex coordinates (v^1, v^2, v^3) and baycentric coordinates $(\lambda^1, \lambda^2, \lambda^3)$.

The baycentric coordinates are in the form of

$$(i, j) \geq 0, i + j \leq N$$

$$(\lambda^1, \lambda^3) = \left(\frac{i}{N}, \frac{j}{N}\right), \lambda^2 = 1 - \lambda^1 - \lambda^3 \,. \tag{54}$$

Using these coordinates' values will give us an ill-conditioned Vandermonde matrix. However, we can utilize the warping function – equation (51) – to blend the edge mapping into the triangle [25]. For simplicity, the expression for warping and blending functions will be expressed in terms of the baycentric coordinates. For the first edge connecting vertices,$v^1$and v^2, the normal warping function is

$$w^1(\lambda^1, \lambda^2, \lambda^3) = w(\lambda^3 - \lambda^2)\begin{pmatrix}1\\0\end{pmatrix} \tag{55}$$

and the blending function is

$$b^1(\lambda^1, \lambda^2, \lambda^3) = \left(\frac{2\lambda^3}{2\lambda^3 + \lambda^1}\right)\left(\frac{2\lambda^2}{2\lambda^2 + \lambda^1}\right) \tag{56}$$

Equation (54) will be singular when$\lambda^1 = \lambda^2 = \lambda^3 = 0$. To account for this behavior, the warping equation is redefined as

$$\widetilde{w}(r) = \frac{w(r)}{1 - r^2} \tag{57}$$

Therefore, equation (54) is rewritten as

$$w^1(\lambda^1, \lambda^2, \lambda^3) = \widetilde{w}(\lambda^3 - \lambda^2)\begin{pmatrix}1\\0\end{pmatrix} \tag{58}$$

And equation (55) is also rewritten as

$$b^1(\lambda^1, \lambda^2, \lambda^3) = 4\lambda^3\lambda^2 \tag{59}$$

For the other two edges, in similar manner, the warping equations and blending equations are

$$w^2(\lambda^1, \lambda^2, \lambda^3) = \widetilde{w}(\lambda^1 - \lambda^3)\frac{1}{2}\begin{pmatrix}1\\\sqrt{3}\end{pmatrix} \tag{60}$$

$$w^3(\lambda^1, \lambda^2, \lambda^3) = \widetilde{w}(\lambda^2 - \lambda^1)\frac{1}{2}\begin{pmatrix}-1\\-\sqrt{3}\end{pmatrix}$$

$$b^2(\lambda^1, \lambda^2, \lambda^3) = 4\lambda^3\lambda^1$$

$$b^3(\lambda^1, \lambda^2, \lambda^3) = 4\lambda^2\lambda^1 .$$

Combining these equations for the triangular grid, the two-dimensional warping equation for the triangle is

$$w(\lambda^1, \lambda^2, \lambda^3) = b^1 w^1 + b^2 w^2 + b^3 w^3 . \tag{61}$$

The process for implementing equation (59) in a numerical algorithm is illustrated in appendix A in the data structure section and basis function construction codes. Using this method ensures that we obtain “good points” to construct the “well-behaved” Vandermonde matrix. With the Vandermonde matrix, we are able to transform directly

between the modal representation, using $\hat{u}_n$as the un-known, and the nodal form$u$. More details for the general case are discussed in Heshaven [25].

Creating Element Operators

With the local approximation finished, the development and assembly of the computational operators is the next step. From the one-dimensional section, the mass matrix M is

$$M_{ij}^k = \int l_i^k(x) l_j^k(x) dx = J^k \int l_i(r) l_j(r) dr \tag{62}$$

where l_i is the Lagrange polynomial, and J^k is the Jacobian. From the basis function and one-dimensional sections, equation (62) can be written in term of the Vandermonde matrix after it is built. Therefore, equation (62) becomes

$$M^k = J^k (VV^T)^{-1} \tag{63}$$

Like the Galerkin finite element method, we also need to build the stiffness or differentiation matrices. Unlike the Galerkin finite element method, the discontinuous Galerkin finite element method allows us to construct the operator matrices in term of the Vandermonde matrix. Using the chain rule,

$$\frac{\partial}{\partial x} = \frac{\partial r}{\partial x} D_r + \frac{\partial s}{\partial x} D_s \tag{64}$$

$$\frac{\partial}{\partial y} = \frac{\partial r}{\partial y} D_r + \frac{\partial s}{\partial y} D_s \, .$$

The differentiation of x and y can be written in terms of the standard-triangle coordinate system. From equation (59), we are able to obtain $\frac{\partial r}{\partial x}, \frac{\partial s}{\partial x}, \frac{\partial r}{\partial y}$, and $\frac{\partial s}{\partial y}$. To find D_r and D_s, we need

$$V_r = \frac{\partial \psi}{\partial r} \tag{65}$$

$$V_s = \frac{\partial \psi}{\partial s}$$

and ψ is obtained by equation (52). From equations (52), (53), (54), and (55), equation (65) becomes

$$\frac{\partial \psi}{\partial r} = \frac{\partial a}{\partial r}\frac{\partial \psi}{\partial a} \tag{66}$$

$$\frac{\partial \psi}{\partial s} = \frac{\partial a}{\partial s}\frac{\partial \psi}{\partial a} + \frac{\partial \psi}{\partial b}$$

Therefore, the matrices D_r and D_s follow directly

$$D_r V = V_r \tag{67}$$

$$D_s V = V_s .$$

For convenience, based on equation (67), we write the stiffness matrices as

$$S_r = M^{-1} D_r \tag{68}$$

$$S_s = M^{-1} D_s$$

Integration on an Edge of the Triangle

The previous section was dedicated to building operators, but the evaluation for the right-hand side of equations (43) or (44) is hardly mentioned due to its complexity in

higher dimensions. In equation (43) or (44), we can write the right-hand side in the generic form

$$\int \hat{n} \cdot g_h l_i^k(x) dx \tag{69}$$

where g_h is composed of either the numerical flux or the jump in the flux, depending whether we use the weak or the strong form. In the two-dimensional problem, equation (69) is divided into the three individual edge components, and each of them has

$$\int \hat{n} \cdot g_h l_i^k(x) dx = \sum_{j=1}^{N+1} \hat{n} \cdot g_j \int l_j^k(x) l_i^k(x) dx. \tag{70}$$

The integral on the right-hand side of equation (70) is integrated along an edge of the triangle; $N+1$ is the number of nodal points; and $\hat{n}$ is the outward normal vector, which takes a constant value.

Similar to the construction of operators, we use equation (62) and (63) to create the mass matrix,M. Though the matrix M has the size of $N_P \times (N_P + 1)$, $l_i(x)$ is a polynomial of order N, including along the edge. If x_i does not reside on the edge and $l_i(x)$ is an N^{th}-order polynomial, $l_i(x)$ is zero at $N+1$ grid points, which is zero along the edge. As a result, the mass matrix M has non-zero entries in row i, where x_i resides on the edge. If we define the Vandermonde matrix corresponding to the one-dimensional interpolation along an edge, we will have exactly the same equation as equation (63), but the Jacobian, J, is the transformation Jacobian long the face – the ratio between the length of the face in normal and reference triangles, respectively.

This is an advantage of the nodal basis, because it allows us to use information along the edges to compute the surface integrals. In the modal basis, all information is evaluated at a point; therefore, the surface integral is difficult to compute.

To implement the surface integral, we define a matrix called F_{mask}, which has the size of $N_{fP} \times 3$. $N_{fP}$is the number of nodes along each of the three faces on the triangle. We can use F_{mask} to extract the edge coordinates from the local Vandermonde and mass matrices along the edge. In addition, we also define the matrix ϵ, having the size of $N_P \times 3N_{fP}$ in order to compute the surface integral of equation (70) for the reference triangle.

All of the computational implementation for the construction of these operators is written in the C++ programming language and select code sections are given in appendix B. The implementation will return the matrix $Lift$ or a $Lift$ function in our C++ algorithm. $Lift$ has a size of $N_P \times 3N_{fP}$ as

$$Lift = M^{-1}\epsilon . \tag{71}$$

Our C++ algorithm also implements the gradient, divergence, and curl operators for use as needed.

Assembling the Grid

With everything in place, the actual computation of these matrix coefficients and the assembly of the global grid structure are carried out. We assume the following information in the assembly process

- Two row vectors (VX, VY) with the coordinates of the N_v vertices. These coordinate sets are numbered from 1 to N_v.
- An integer matrix EToV having the size $K \times 3$ with the three vertex numbers in each row forming one element for a total of K elements. The three vertices are ordered counter-clockwise such that the Jacobian matrices are positive.

All of the assembling process is included in our C++ algorithm in appendix C.

The Heat Equation Example

With all the formulation of the discontinuous Galerkin finite element method in placed to solve equation (43) or (44), a description of applying the approach for solving a parabolic differential equation is needed. Since our research compares the continuous and discontinuous Galerkin finite element method to solve parabolic partial differential equations, an example parabolic partial differential equation is helpful for illustrating the use of discontinuous Galerkin finite element method on this type of problem. For simplicity, we are going to use the time-dependent conductive heat equation as an example:

We assume that the thermal conductivity $k = 1$ for simplicity.

$$\frac{\partial u}{\partial t} = k\nabla^2 u \,. \tag{72}$$

Equation (72) will be rewritten as a first-order system

$$\frac{\partial u}{\partial t} = \nabla \cdot q \tag{73}$$

$$q = \nabla u\,.$$

Multiplying both equations in the system (73) by the test function and integrate by parts over the K elements

$$(\frac{\partial u}{\partial t}, \phi_h)_\Omega = -(q, \nabla\phi_h)_\Omega + \sum (n \cdot q_h^*, \phi_h)_\Gamma \tag{74}$$

$$(q_h, \psi_h)_\Omega = \sum (u_h^*, n \cdot \psi_h)_\Gamma - (u_h, \nabla \cdot \psi_h)_\Omega$$

For convenience, $(a, b)_\Omega$ implies that the product of functions a and b is integrated over the domain Ω. Γ represents an internal or external element edge.

In addition, q^* and $u^*$are the numerical flux terms on the boundary of every element. There are many different numerical flux choices for the discontinuous Galerkin finite element method, such as the unwinding scheme [28], Lax-Fredrichs [25], LDG [41], and hybridization method [42]. In our algorithm, since we often use an iterative method to solve our linear systems, our choice for the numerical flux is the internal penalty flux, which is appropriate for an iterative solver because the operator has greater sparsity. The internal penalty flux will be introduced later. In order to transform equations (96) and (97) to obtain the numerical flux and solve these two equations, we first define

$$\{u\} = \frac{u^- + u^+}{2} \tag{75}$$

$$[u] = n^- \cdot u^- + n^+ \cdot u^+.$$

The "+" and "–" indicate the interior and exterior information of the element. To deal with discontinuities in the solution, we extract information around the discontinuity between elements in order to average the information and approximate the solution based on the average. In the discontinuous Galerkin finite element method, we extract interior

and exterior element information, and use the system equation (75) to construct the numerical flux in order to approximate the solution.

It can easily be shown that

$$\sum (n \cdot u, \phi)_{\Gamma} = \oint \{u\} \cdot [\phi] dx + \int \{\phi\} \cdot [u] dx \,. \tag{76}$$

Equation (76) allows us to transform the system of equation (74) into

$$\begin{aligned} (\frac{\partial u}{\partial t}, \phi_h)_{\Omega} = & -(q, \nabla \phi_h)_{\Omega} \\ & + \oint \{q^*\} \cdot [\phi_h] dx + \int \{\phi_h\} \cdot [q^*] dx \\ (q_h, \psi_h)_{\Omega} = & \sum (u_h^*, n \cdot \psi_h)_{\Gamma} - (u_h, \nabla \cdot \psi_h)_{\Omega} \end{aligned} \tag{77}$$

Integrating the first equation in the system of equation (77) by part again gives

$$\begin{aligned} (q_h, \psi_h)_{\Omega} = & (\nabla u_h, \psi_h)_{\Omega} \\ & - \oint \{u_h^* - u_h\} \cdot [\psi_h] dx + \int \{\psi_h\} \cdot [u_h^* - u_h] dx \end{aligned} \tag{78}$$

Equation (78) is typically written in a differential form as

$$q_h = \nabla u_h + L(u_h) \tag{79}$$

$$\text{where } L(u_h) = [u_h^* - u_h] \,.$$

The numerical flux is assumed to be single valued, and utilizing the internal penalty method gives

$$[u_h^*] = 0 \tag{80}$$

$$[q_h^*] = 0$$

$$\{u_h^*\} = u_h^* = \{u_h^*\}$$

$$\{q_h^*\} = q_h^* = \{\nabla u_h\} - \tau [u_h]$$

where τ is the penalty parameter, which is set to:

$$\tau = 20(N+1)^2 h^{-1}\,. \tag{81}$$

Equation (81) will be seen in appendix D where our algorithm is going to be included for clarification. Substituting equation (79) and the numerical flux for q^* into equation (78) gives

$$\begin{aligned}(\frac{\partial u}{\partial t}, \phi_h)_\Omega = & -(\nabla u_h, \nabla \phi_h)_\Omega \\ & + \oint (\{\nabla u_h\} \cdot [\phi_h] + [u_h] \cdot \{\nabla \phi_h\}) dx - \oint \tau [u_h][\phi_h] dx\end{aligned} \tag{82}$$

Equation (82) is the equation used in our algorithm to solve for the heat equation. For advection-diffusion and Burger's equations, we apply the same procedure in order to use the discontinuous Galerkin finite element method to solve those parabolic differential equation problems. The next section will discuss how we compare the continuous and discontinuous Galerkin finite element methods to each other.

The Comparison to Continuous Galerkin Finite Element Method Scheme

The objective in developing the computational code for this comparison was to have as much shared code as practically possible between the continuous and discontinuous Galerkin finite element method algorithms. To simplify the comparison, the code was written for serial execution in a single thread, and all code developed in-house was written in C++. The importing and exporting of the mesh description file and the writing of the final approximate solution all utilized the ExodusII library, written in C [44]. The finite element meshes were all generated using the Cubit software (Sandia

National Laboratory) and the geometries are all 2-dimensional and meshed with 3 or 6 node triangles using the TriMesh algorithm.

Both the continuous and discontinuous Galerkin finite element method algorithms utilized some of the core packages from the Trilinos library [45, 46]. Specially, all global operators and vectors were built and stored using the Epetra package. The objective in selecting algorithms to solve the linear matrix problem was to utilize the most popular and robust methods available. Therefore, to compare performance using a direct matrix solver, the Amesos package from Trilinos is used [47], and to compare performance using an iterative matrix solver, the AztecOO GMRES algorithm with as many default settings as possible was used [48].Obviously, the computational performance could be improved through the use of more recent, specialized matrix solvers such as an algebraic multi-grid solver. The problem with using these types of solvers in a comparison is that they often have a number of adjustable parameters that can significantly impact the solver performance. It is difficult to determine the optimal solver parameters for a wide range of problems and discretization, and this makes it difficult to develop a fair comparison. If a uniform set of solver parameters is used in the comparison, it will almost always favor one discretization over another. The other issue with utilizing a wide range of matrix solvers is that, inevitably, some solvers will be left out and those might be very computationally efficient with one discretization. To minimize all these issues, every effort has been made to focus on just the most common linear solvers with very few adjustable parameters.

Since common libraries are used for mesh import and export, operator storage, linear operations, and linear solvers, the main sections of the algorithms that are not shared involve the building of basis functions and the construction of element level operators. For the continuous Galerkin finite element method code, standard quadratic (i.e., 6-node triangles) nodal basis functions were generated, and the local element operators were based on the well-known Galerkin weak form [49, 19, 7, and 50].In all cases, time stepping was accomplished using the second-order accurate trapezoidal rule (i.e., Crank-Nicolson method). Due to concerns about the accuracy and stability of the trapezoidal rule, time stepping using the second-order Backward Difference Formula (BDF2) was also tested, and it gave results nearly identical to the trapezoidal rule in every case tested.

One factor in comparing the continuous to discontinuous Galerkin finite element method is the complexity of implementation, and this factor is very difficult to quantify. We will simply note that the internal penalty weak form used here for the discontinuous Galerkin finite element method is more complex to derive and implement than the associated continuous Galerkin finite element method case.

The discontinuous Galerkin finite element method algorithm used here was based on the algorithms presented in [25]. Specially, the nodal basis functions and nodal element operators were based on algorithms presented in chapter 6 of Hesthaven and Warburton's book [25], and careful comparisons were made between the C++ code developed here and the MATLAB code provided by Hesthaven and Warburton to help support a proper and correct implementation.

In order to simplify and clarify the comparison between the continuous and discontinuous Galleria finite element methods for parabolic partial differential equations, the test problems used in the comparison are described in the results Chapter 3 so that the results can be closely connected to the relevant test problems.

CHAPTER 3

RESULTS

Four different test problems were selected for comparing the continuous and discontinuous Galerkin finite element method. The first problem is the steady-state heat equation, or Poisson’s equation, which is obviously not a parabolic equation, but has the advantages of sharing a number of properties with a simple parabolic equation, an analytical solution, and a focus on spatial error, not temporal error.

The second problem is the advection-diffusion equation with decreasing diffusivity values. The third problem is the viscous Burger's equation, which is similar tithe second problem, but is nonlinear. The final problem is the Turing pattern formation problem, which was included because it is a system of equations with multiple unknowns. We have deliberately avoided the Stokes and Navier-Stokes equations because they are the subject of so much ongoing research in the field of continuous and discontinuous Galerkin finite element methods and there is not general agreement on the best approach.

The contents of this chapter are based on a paper recently submitted for publication.

Poisson's Equation

The first test problem is the 2-dimensional Poisson equation:

$$\Delta u = -2\pi^2 \sin(\pi x) \sin(\pi y) \tag{83}$$

with the right-hand side chosen so that it has homogenous Dirichlet boundary conditions on the unit square and an analytical solution. Table 1 summarizes the computational time and error in the L^{∞}-norm for the continuous and discontinuous Galerkin finite element method. The analytical solution provides a method for validating our code to ensure that the implementation of the discontinuous Galerkin finite element method is correct. The figure 4 shows the solution of the heat equation

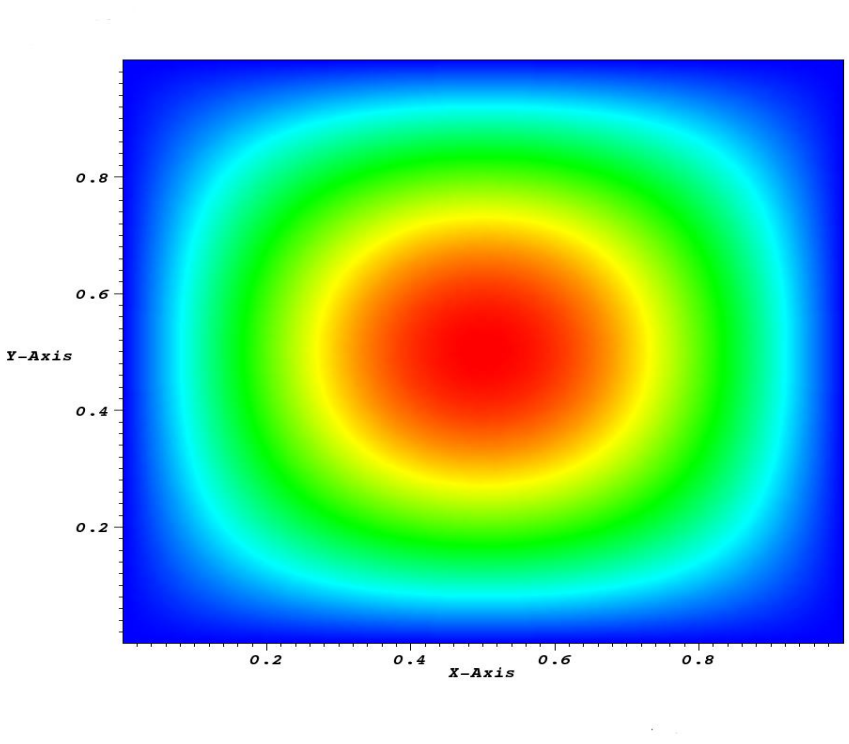

Figure 4: The solution of the steady time heat equation

Number of element	Continuous Galerkin finite element method				Discontinuous Galerkin finite element method			
	Error	Amesos	Aztec	ML	Error	Amesos	Aztec	ML
666	2.8e-5	0.043	0.056	0.08	3.1e-5	0.52	0.66	1.3
2664	2.7e-6	0.2	0.25	0.29	3e-6	4.5	4.4	5.5
4448	1.5e-6	0.42	0.39	0.57	1.5e-6	8.3	10.5	10.6
10656	3.7e-7	1.3	1.1	1.8	3.9e-7	23.2	39.6	28.6

Table 1: The error in the L^{∞}-norm and the total computational time (in seconds) for three different solvers for solving equation (83) using the continuous and discontinuous Galerkin finite element method with various mesh resolutions. Amesos is a direct solver, Aztec is an ILU-preconditioned GMRES solver, and ML is an algebraic multi-grid solver.

A number of important observations can be made from the results shown in table 1. First, this problem possesses a very smooth solution with relatively small gradients (i.e., gradients that are small relative to the domain size), and both methods have practically equal accuracy in the L^{∞}-norm. This is not surprisingsince both approaches used the same six-node, triangular elements and have the same order of accuracy.

The second observation that can be made is that the computational cost for the discontinuous Galerkin finite element method approach is consistently higher than for the continuous case. For Amesos, a sparse, direct solver, the computational cost is approximately a factor of ten higher for the discontinuous algorithm relative to the continuous one and the solver displays surprisingly good scalability in both cases. This trend continues with the Aztec solver, an ILU-preconditioned GMRES algorithm, but here the solver does not scale as well in the discontinuous case (order n^2) compared to the continuous case (order $n^{\frac{3}{2}}$).

The algebraic multi-grid solver, ML, gives similar scalability in either case, but this problem is really too small to take advantage of the benefits of a multilevel solver.

The performance of the ML solver could be improved by adjusting some of the solver parameters or using specialized aggregation schemes [35], but the default parameters were used for both cases here to keep the comparison as fair as possible.

The higher computational cost of the discontinuous algorithm is explained by the fact that the discontinuous approach has many more degrees-of-freedom relative to the continuous case. The six nodes of every triangular element lie either on the edge of the element or the vertex, so the unknown associated with each and every node is duplicated at least once in the discontinuous algorithm. As a result, the discontinuous approach has approximately three times as many unknowns as the continuous approach, and those additional unknowns significantly increase the computational costs. For this particular Poisson problem, there is not a compelling reason to use a discontinuous Galerkin finite element method approach. However, when we add advection in the next example, the case for using discontinuous Galerkin finite element method approaches becomes stronger.

Advection-Diffusion Equation

The second test problem is based on the advection-diffusion equation:

$$\frac{\partial u}{\partial t} + v \cdot \nabla u = D \Delta u \tag{84}$$

where u is an unknown field undergoing advection and diffusion, D is the diffusivity, and v is a given velocity field. The issue of primary interest here is the performance of the continuous and discontinuous Galerkin finite element method when D is small (i.e.,

D<<1). The domain and boundary conditions used in this test problem are shown in figure 5, and the function $g(x,t)$ used in the boundary condition is given as:

$$g(x,t) = 4\sin(\pi t)\,(x-1)(2-x) \text{ for } \sin(\pi t) > 0 \qquad (85)$$

otherwise $g(x,t) = 0$.

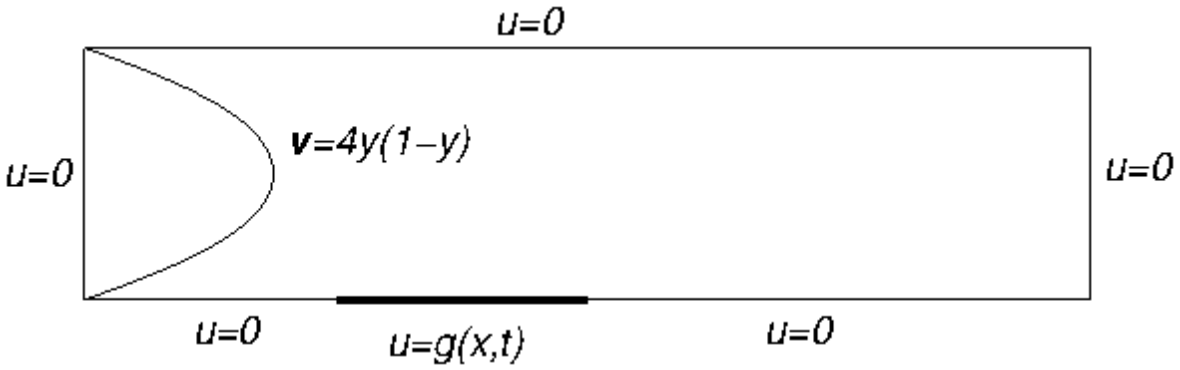

Figure 5: The domain and boundary conditions for both the advection-diffusion and viscous Burger's equation test problems. The height of the domain is 1.0, and the width is 4.0.

The test problem is basically a pulsing plate (e.g., a plate with an oscillating temperature) and the pulses are advected downstream to the right. For this channel flow problem, the analytical solution to the Navier-Stokes equations is available and used to specify the velocity in (84). The lower velocities near the walls reduce the advection rate when the diffusivity is reduced because the pulses stay closer to the walls. Figure 6a and 6b show a numerical solution for both the continuous and discontinuous Galerkin finite element method's simulations, respectively, when D = 0.01, and the solutions are quite similar at this diffusivity value.

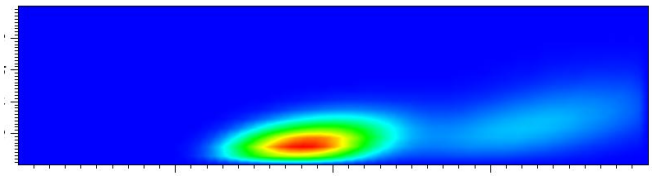

Figure 6a: Continuous Galerkin finite element method's solution for $D = 0.01$ at $t = 3.5\ s$

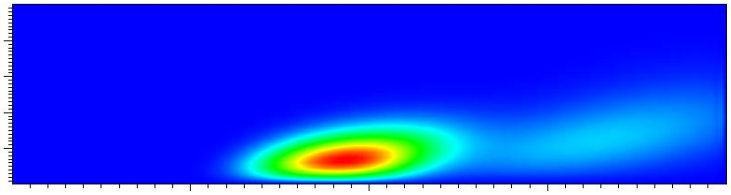

Figure 6b: Discontinuous Galerkin finite element method's solution for $D = 0.01$ at $t = 3.5\ s$

However, if the diffusivity is further reduced to $D = 0.0005$, the numerical solution using a continuous Galerkin finite element method discretization shows significant instability because the discretization does not include up winding. The discontinuous Galerkin finite element method solution is stable because up winding is incorporated into the discretization through the numerical flux. Figure 7a and 7b show the solution for the continuous and discontinuous Galerkin finite element methods respectively.

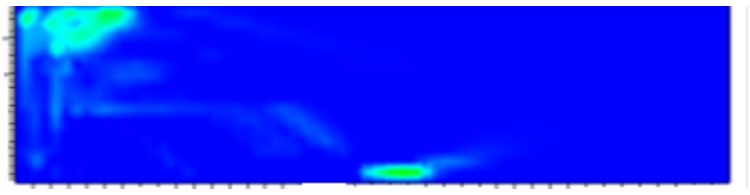

Figure 7a: Continuous Galerkin finite element method's solution for $D = 0.0005$ at $t = 0.6\ s$

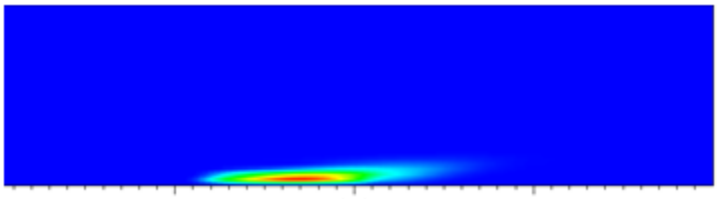

Figure 7b: Discontinuous Galerkin finite element method's solution for $D = 0.0005$ at $t = 0.6\ s$

The instability in the continuous GFEM can be addressed through a number of different mechanisms including the use of a Petrov-Galerkin finite element method (SUPG). The computational cost of the SUPG approximation was typically 5-10% higher than the continuous Galerkin finite element approximation, but it was still considerably lower than the discontinuous Galerkin finite element method case. The SUPG finite element method introduces some numerical diffusion in the approximate solution [51].Figures 8a and 8b will show the results for Petrov-Galerkin and discontinuous Galerkin finite element method.

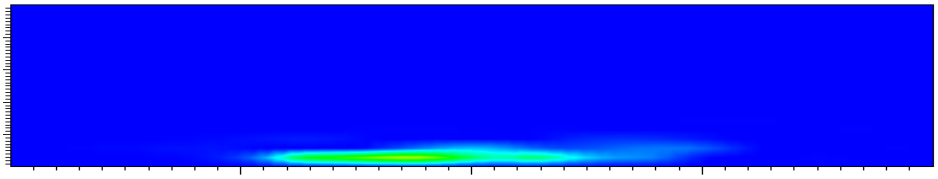

Figure 8a: Continuous Petrov-Galerkin (SUPG) finite element method's solution for D = 0.0005 at $t = 0.6\ s$

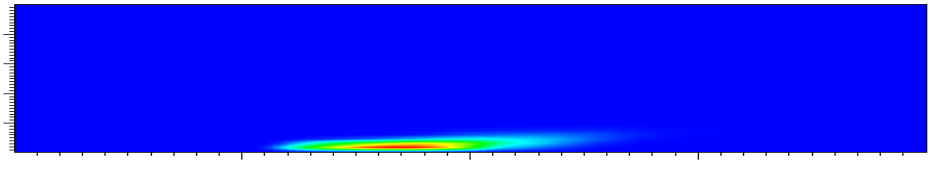

Figure 8b: Discontinuous Galerkin finite element method's solution for D = 0.0005 at $t = 0.6\ s$

It is important to emphasize that there are a number of different Petrov-Galerkin methods and many of them contain adjustable parameters. The future work, described in the next chapter, recommends a detailed comparison of the various Petrov-Galerkin approaches and discontinuous Galerkin finite element method in terms of accuracy, stability, and computational cost for this class of problems, but that comparison is large and beyond the scope of this work.

The computational costs of the continuous and discontinuous Galerkin finite element method for 800 times steps of the advection-diffusion test problem are summarized in table 2. The first observation that can be made is that for this small test problem, a sparse direct solver is faster than an iterative solver. The trend in the rates of solution time suggests that the iterative solvers scale better than the direct solver, which should be the case. The computational cost is also largely independent of the diffusivity, D. Finally, the computational cost of the continuous Galerkin finite element method is typical a factor of 5 lower than the discontinuous Galerkin finite element method due to the small number of degrees of freedom in the continuous case, but the continuous case is also much more susceptible to large instability errors at smaller values of diffusivity.

Diffusivity D	Number of elements	Continuous Galerkin finite element method		Discontinuous Galerkin finite element method	
		Amesos	Aztec	Amesos	Aztec
0.01	586	14.7	23.0	48.5	62.9
0.0005	586	14.4*	23.3*	48.4	58.5
0.01	2025	57.2	84.8	240.1	237.3
0.0005	2025	57.2*	84.9*	240.9	220.9

Table 2: The total computational time (in seconds) required for solving (84) with 800 time steps using the continuous and discontinuous Galerkin finite element methods with two different meshes, two different values for D, and two different solvers (Amesos = direct solver, Aztec = ILU-preconditioned GMRES solver). For the small values of D, the numerical solution using the continuous Galerkin finite element method can become unstable and contain large errors. Unstable solutions are indicated by a *.

Viscous Burger's Equation

The third test problem is the 2-dimensional viscous Burgers' equation.

$$\frac{\partial u}{\partial t} + \nabla(u^2) = \mu \Delta u \,, \tag{85}$$

which is similar to the advection-diffusion except the advection term is now nonlinear. The domain and boundary conditions for this problem are the same as those used for the advection diffusion equation, shown in figure five, but in this case there is no external advecting velocity field. The nonlinearity was handled in both the continuous and discontinuous algorithms using Newton's method, which requires the construction of a residual vector and a Jacobian matrix.

For the comparison, two different viscosities,μ, were tested with both continuous and discontinuous Galerkin finite element method. The results for $\mu = 0.01$ using both the continuous and discontinuous Galerkin finite element method are shown in figure 9.

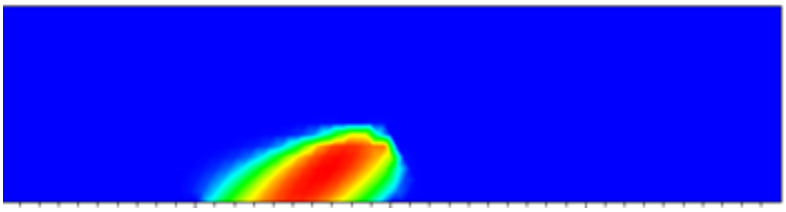

Figure 9a: Continuous Galerkin finite element method with $\mu = 0.01$at $t = 0.06s$

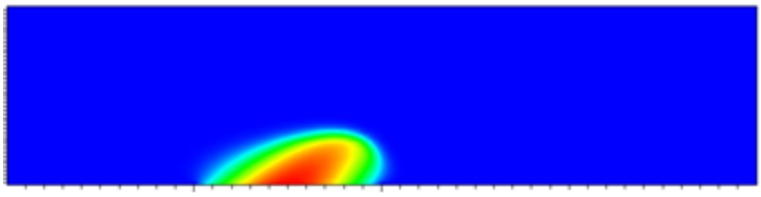

Figure 9b: Discontinuous Galerkin finite element method with $\mu = 0.01$ at $t = 0.06s$

The first observation is that the advection of the pulses from the plate along the bottom of the domain is at an45° angle away from the plate, and the velocity of the advection is(u,u). As a result, regions where u is larger move faster than regions where u issmaller. As the viscosity decreases, the faster moving regions tend to run intothe slower moving regions and a large gradient that approaches a shock (but does not become a shock for $\mu > 0$) forms along the front. The solution is resolved well by either approach as long as $\mu$is sufficiently large.

Figure9 shows the solution for both continuous and discontinuous Galerkin finite element method when the viscosity, μ, is less than 0.001. When the viscosity is sufficiently small, the continuous Galerkin finite element method starts to become unstable because of the lack of up winding in the discretization, but the discontinuous

Galerkin finite element method is stable due to the inclusion of up winding in the numerical flux

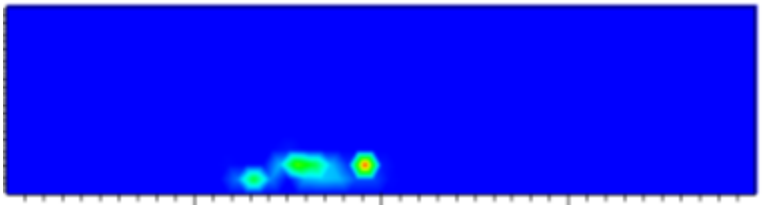

Figure 10a: Continuous Galerkin finite element Method $\mu = 0.000001$ at $t = 2.0\ s$

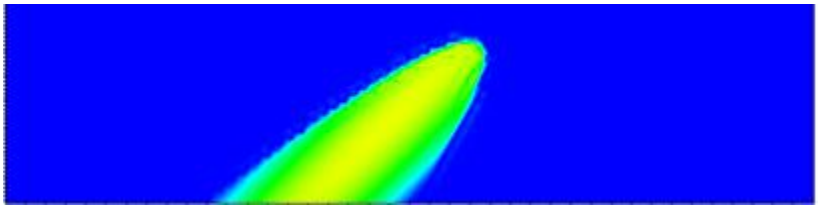

Figure 10b: Discontinuous Galerkin finite element method $\mu = 0.000001$ at $t = 2.0\ s$

The instabilities in the continuous finite element method solution can lead to divergence of Newton's method at approximately $t = 1.0\ s$in several of the problems considered. Very smalloscillations can also form using the discontinuous Galerkin finite element method, but do not appear to affect the results significantly over the range of μ values considered in thisstudy.

The computational time required for simulating 800 time steps for both the continuous and discontinuous Galerkin finite element method using different values for μ and different finite element meshes are summarized in table 3. Even though the same meshes are used, as in the advection-diffusion test problem, multiple iterationsare required by Newton's method each time step, and this increases the overall computational cost. For small values of μ, the continuous Galerkin finite element method diverged and

solution was not obtained. These simulations are indicated by `-' in table 3. When the continuous Galerkin finite element method did converge, it was significantly faster than the discontinuous Galerkin finite element method, often by a factor of 20 or more

Viscosity μ	Number of Elements	Continuous Galerkin finite element method		Discontinuous Galerkin finite element method	
		Amesos	Aztec	Amesos	Aztec
0.01	586	23.7	37.6	171.9	222.6
0.00005	586	-	-	185.9	221.7
0.01	2025	104.5	156.2	844.1	858.3
0.00005	2025	-	-	953.0	891.8

Table 3: The total computational time (in seconds) for solving the viscous Burgers' equation with 800 time steps and using the continuous and discontinuous Galerkin finite element methods with two different meshes, two different values for μ, and two different solvers (Amesos = direct solver, Aztec= ILU-preconditioned GMRES solver). For small values of μ, instabilities led to Newton's method failing to converge using the continuous Galerkin finite element method, and these cases are shown indicated with a `-'

Turing Pattern Formation

The final test problem is the Turing pattern formation problem (also known as the reaction-diffusion model), which is used to model the formation of biological patterns (e.g., zebra stripes). Simple Turing models include two biological morphogens; one morphogen is responsible for short-range positive feedback and the other is responsible for long-range negative feedback. In these systems, a random initial condition will lead to a stationary pattern given the proper choice of parameters [52]. This test problem was chosen because it has multiple equations and multiple unknowns. The following system of equations and model parameters were chosen because they are known to have a stationary pattern as a solution:

$$\frac{\partial u}{\partial t} = F(u,v) - 0.03u + 1.2 \cdot 10^{-6}\Delta u \tag{86}$$

$$\frac{\partial v}{\partial t} = G(u,v) - 0.08v + 3.1 \cdot 10^{-5}\Delta v$$

where $F(u,v) = 0.08u - 0.08v + 0.04$

$$G(u,v) = 0.1u - 0.15$$

and the reaction rates are limited to the range of $0 \leq F(u;\ v) \leq 0.2$ and $0 \leq G(u;\ v) \leq 0.5$. The limits on the reaction rate make the problem nonlinear, and the nonlinearity is addressed using Newton's method. The model domain was the unit square with Neumann boundary conditions on all sides. The final pattern was always based on a random initial guess so the final stationary solution was never the same. For all finite element meshes tested here, both the continuous and discontinuous Galerkin finite element method gave very accurate results and there was never a noticeable difference in accuracy between the methods. A typical solution is shown in figure 10.

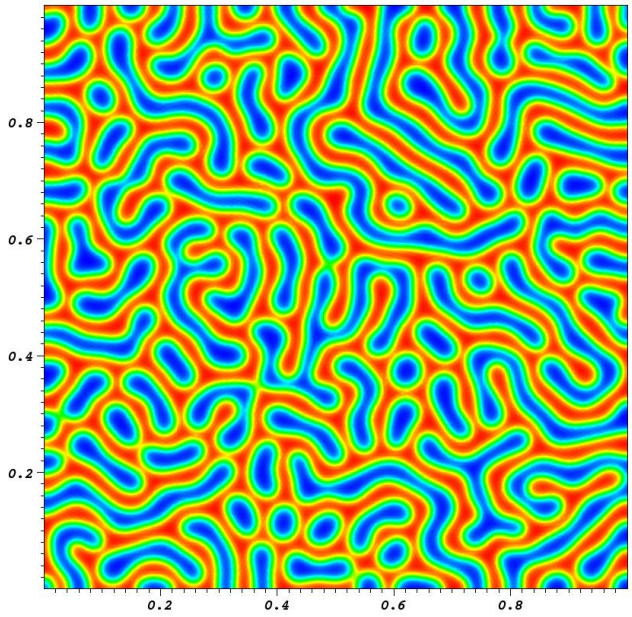

Figure 11: A typical solution for the concentration u in the Turing pattern formation problem using the given parameters. The solution is based on a random initial pattern and the final, stationary pattern does depend somewhat on the initial conditions.

The computational cost associated with solving equations (86) for a 1400 time steps is summarized in table 4. Again, considering only computational costs, the data leads to the conclusion that the continuous Galerkin finite element method with its reduced number of degrees of freedom is significantly less expensive. For both the continuous and discontinuous Galerkin finite element method, the problems are large enough that the ILU-precondition GMRES solver, Aztec, is significantly faster than the direct solver, Amesos, dueto the better scalability of the iterative solver. This problem can also be solved very efficiently using a multilevel solver such as ML, but the basic conclusion is still the same. The continuous Galerkin finite element method is significantly faster because it has fewer degrees of freedom. Without the presence of strong advection, it is difficult to justify the additional computational cost associated with the discontinuous Galerkin finite element method

Number of Elements	Continuous Galerkin finite element method		Discontinuous Galerkin finite element method	
	Amesos	Aztec	Amesos	Aztec
2264	985	988	13050	5568
4488	1998	1654	33358	102334
10656	6209	4043	140691	27539

Table 4: The total computational time in seconds for solving equations (86) for 1400 time steps using the continuous and discontinuous Galerkin finite element methods and various mesh resolutions.

CHAPTER 4

CONCLUSION AND FUTURE WORK

Conclusion

Both the continuous and discontinuous Galerkin finite element methods are useful tools for approximating the solution to parabolic differential equations. The best choice, of course, is highly problem dependent. For problems with relatively smooth solutions and a sufficiently large second-order term, the choice will usually be the continuous Galerkin finite element method due to the significantly reduced number of degrees-of-freedom. For the test problems explored here, the continuous Galerkin finite element method was typically 5-20 times less computationally expensive relative to the discontinuous Galerkin finite element method. For higher-order elements with more interior nodes, this difference may be less.

The choice between a continuous or discontinuous Galerkin finite element method is based on computational cost and the ability to handle a small viscosity or diffusivity term. From the examples shown in this thesis, the continuous finite element method is an obvious choice when the second-order derivative term is sufficient large. When the viscosity or diffusivity coefficient is less than O (10^{-3}) , the discontinuous Galerkin finite element method is an excellent choice, despite high computational cost, because it can provide a very accurate approximate solution.

For problems with large advection terms or, equivalently, small second-order terms in the equation, the instabilities that arise when using continuous Galerkin finite

element method can lead to these methods not being acceptable. In these cases, the discontinuous Galerkin finite element method can be an excellent choice because it displays much better stability without the addition of stabilization. It is important to note that the instabilities that plague the continuous Galerkin finite element method cannot be resolved with a moderate amount of mesh refinement for the examples shown here. Indeed, refining the mesh to the point that the continuous and discontinuous Galerkin finite element method had the same computational cost did not lead to stability of the continuous method.

Future Work

Comparing discontinuous to the continuous Galerkin methods, for the advection dominated problems examined here; the discontinuous Galerkin finite element method was the most effective option. An important question remains with regards to the choice between stabilized continuous and discontinuous Galerkin finite element method for advection dominated parabolic differential equations. It is well known that continuous finite elements can be stabilized using Petrov-Galerkin methods, but these methods typically introduce artificial viscosity. Similarly, in discontinuous Galerkin finite element method, there are many approaches to forming the inter-element flux terms that also can introduce artificial viscosity [53]. The choice between Petrov-Galerkin and discontinuous Galerkin finite element method is extremely complex due to the number of different Petrov-Galerkin method and the fact that the performance and accuracy of Petrov-Galerkin method can be highly problematic and geometry dependent. Ultimately, the

choice between Petrov-Galerkin and discontinuous Galerkin finite element methods will depend upon the unique problem and the accuracy requirements of the particular application. However, a comprehensive comparison between Petrov-Galerkin and discontinuous Galerkin finite element methods, similar to the comparison described here, could be very beneficial for helping practitioners choose a discretization for new mathematical models.

In addition, we would like to try higher-order elements in the future work. The computer code developed for the research described here is capable of using arbitrarily high-order elements. This capability was not used for the work presented here simply because the continuous finite element code was only able to support 3- and 6-node triangles. A comparison of these two approaches using higher-order elements (potentially up to 8^{th}- or 12^{th}-order elements) would be interesting because the disadvantages associated with the discontinuous Galerkin finite element method (specifically, the larger number of degrees-of-freedom) would be minimized with the higher-order elements.

Parallelization for DG is also a potential research venture. It is well known that discontinuous Galerkin finite element algorithms are relatively straightforward to parallelize because so much of the computation is restricted to the individual elements. It would be interesting to compare the parallel scalability of discontinuous and continuous algorithms so that we could better understand whether the straightforward implementation of parallel discontinuous schemes also leads to better computational performance and scalability.

The discontinuous Galerkin finite element method is relatively new and has only been applied to a few, specific applications. The number of potential applications is large, but a few possibilities are highlighted here. The discontinuous Galerkin finite element method could be used to study crack formulation in materials, such as composites, etc. There is also interest in combining the discontinuous Galerkin finite element method with the immersed boundary method to study the fluid-structure interaction for simulating a blast in a military vehicle [54]. It would also be interesting to further apply the discontinuous Galerkin finite element method to turbulent flow modeling, although there is one paper addressing the issue [55].

REFERENCES

1. Drabek, P. and G. Holubová, Elements of partial differential equations. De Gruyter textbook. 2007, Berlin ; New York: Walter de Gruyter. ix, 245 p.

2. Lee, H.J. and W.E. Schiesser, Ordinary and partial differential equation routines in C, C++, Fortran, Java, Maple, and MATLAB. 2004, Boca Raton: Chapman & Hall/CRC. 519 p.

3. Widder, D.V., The heat equation. Pure and applied mathematics (Academic Press). 1975, New York: Academic Press. xiv, 267 p.

4. Barton, G., Elements of Green's functions and propagation: potentials, diffusion, and waves, Oxford [Oxfordshire]New York: Clarendon Press ;Oxford University Press. xiii, 465 p.

5. Han, Q. and F. Lin (2011). Elliptic partial differential equations. New York, N.Y.Providence, R.I., Courant Institute of Mathematical sciences;American Mathematical Society.

6. Courant, R., Friedric.K, et al. (1967). "On Partial Difference Equations of Mathematical Physics." Ibm Journal of Research and Development**11**(2): 215-&.

7. Reddy, J. N. (1984). An introduction to the finite element method. New York, McGraw-Hill.

8. Reddy, J. N. (1984). Energy and variational methods in applied mechanics : with an introduction to the finite element method. New York, Wiley.

9. Huebner, K. H. and E. A. Thornton (1982). The finite element method for engineers. New York, Wiley.

10. Finite Element System Means V9, http://www.fem-infos.com/

11. Zienkiewicz, O. C. (1977). The finite element method. London; New York, McGraw-Hill.

12. Tong, P. and J. N. Rossettos (1977). Finite-element method : basic technique and implementation. Cambridge, Mass., MIT Press.

13. Strang, W. G. and G. J. Fix (1973). An analysis of the finite element method. Englewood Cliffs, N.J. Prentice-Hall.

14. Johnson, C. (1987). Numerical solution of partial differential equations by the finite element method. Cambridge [Cambridgeshire] ; New York, Cambridge University Press.

15. Dow, J. O. (1999). A unified approach to the finite element method and error analysis procedures. San Diego, Academic Press.

16. Lewis, P. E. and J. P. Ward (1991). The finite element method: principles and applications. Reading, Mass., Addison-Wesley.

17. Mitchell, A. and R. Wait (1977). The finite element method in partial differential equations. Chichester; New York, Wiley.

18. Mori, M. (1986). The finite element method and its applications. New York, Macmillan.

19. P. Gresho, R. Sani (1998). Incompressible Flow and the Finite Element Method. Advection-Diffusion and Isothermal Laminar Flow, John Wiley and Son,New York.

20. Di Pietro, D. A. and A. Ern (2012). Mathematical aspects of discontinuous galerkin methods. Mathématiques et Applications,. Berlin; New York, Springer.

21. W.H. Reed and T.R. Hill (1973):Triangular mesh methods for the neutron transport equation, Tech. Report LA-UR-73-479, Los Alamos Scientific Laboratory.

22. Rivière, B. and Society for Industrial and Applied Mathematics. (2008). Discontinuous Galerkin methods for solving elliptic and parabolic equations theory and implementation. Frontiers in applied mathematics 35. Philadelphia, Pa., Society for Industrial and Applied Mathematics.

23. Adjerid, S. and M. Baccouch (2007). "The discontinuous galerkin method for two-dimensional hyperbolic problems. Part I: Superconvergence error analysis." Journal of Scientific Computing**33**(1): 75-113.

24. Adjerid, S. and M. Baccouch (2009). "The Discontinuous Galerkin Method for Two-dimensional Hyperbolic Problems Part II: A Posteriori Error Estimation." Journal of Scientific Computing**38**(1): 15-49.

25. Hesthaven, J. S. and T. Warburton (2008). Nodal discontinuous Galerkin methods: algorithms, analysis, and applications. New York, Springer.

26. Gabard, G., P. Gamallo, et al. (2011). "A comparison of wave-based discontinuous Galerkin, ultra-weak and least-square methods for wave problems." International Journal for Numerical Methods in Engineering**85**(3): 380-402.

27. Cockburn, B., S. C. Hou, et al. (1990). "The Runge-Kutta Local Projection Discontinuous Galerkin Finite-Element Method for Conservation-Laws .4. The Multidimensional Case." Mathematics of Computation**54**(190): 545-581.

28. Cockburn, B., S. Y. Lin, et al. (1989). "Tvb Runge-Kutta Local Projection Discontinuous Galerkin Finite-Element Method for Conservation-Laws .3. One-Dimensional Systems." Journal of Computational Physics**84**(1): 90-113.

29. Cockburn, B. and C. W. Shu (1989). "Tvb Runge-Kutta Local Projection Discontinuous Galerkin Finite-Element Method for Conservation-Laws .2. General Framework." Mathematics of Computation**52**(186): 411-435.

30. Cockburn, B. and C. W. Shu (1991). "The Runge-Kutta Local Projection Rho-1-Discontinuous-Galerkin Finite-Element Method for Scalar Conservation-Laws." Rairo-Mathematical Modelling and Numerical Analysis-Modelisation Mathematique Et Analyse Numerique**25**(3): 337-361.

31. Werder, T., K. Gerdes, et al. (2001). "hp-discontinuous Galerkin time stepping for parabolic problems." Computer Methods in Applied Mechanics and Engineering**190**(49-50): 6685-6708.

32. Wirasaet, D., S. Tanaka, et al. (2010). "A performance comparison of nodal discontinuous Galerkin methods on triangles and quadrilaterals." International Journal for Numerical Methods in Fluids**64**(10-12): 1336-1362

33. Arnold, D. N., F. Brezzi, et al. (2002). "Unified analysis of discontinuous Galerkin methods for elliptic problems." Siam Journal on Numerical Analysis**39**(5): 1749-1779.

34. Arnold, D. N., F. Brezzi, et al. (2002). "Unified analysis of discontinuous Galerkin methods for elliptic problems." Siam Journal on Numerical Analysis**39**(5): 1749-1779.

35. Olson, L. N. and J. B. Schroder (2011). "Smoothed aggregation multigrid solvers for high-order discontinuous Galerkin methods for elliptic problems." Journal of Computational Physics**230**(18): 6959-6976.

36. Hesthaven, J. S. and T. Warburton (2004). "High-order nodal discontinuous Galerkin methods for the Maxwell eigenvalue problem." Philosophical Transactions of the Royal Society of London Series a-Mathematical Physical and Engineering Sciences**362**(1816): 493-524.

37. Szeg*o, G. (1959). Orthogonal polynomials. New York,, American Mathematical Society.

38. Johnson, C. and J. Pitkaranta (1986). "An Analysis of the Discontinuous Galerkin Method for a Scalar Hyperbolic Equation." Mathematics of Computation**46**(173): 1-26.

39. Lyuu, Y.-D. (2004). Information dispersal and parallel computation. Cambridge; New York, Cambridge University Press.

40. Kaneko, H., K. S. Bey, et al. (2006). "A discontinuous Galerkin Method for parabolic problems with modified hp-finite element approximation technique." Applied Mathematics and Computation**182**(2): 1405-1417

41. Cockburn, B., B. Dong, et al. (2008). "A superconvergent LDG-hybridizable Galerkin method for second-order elliptic problems." Mathematics of Computation**77**(264): 1887-1916.

42. Cockburn, B. and J. Gopalakrishnan (2009). "The Derivation of Hybridizable Discontinuous Galerkin Methods for Stokes Flow." Siam Journal on Numerical Analysis**47**(2): 1092-1125.

43. Burman, E. (2009). "A Posteriori Error Estimation for Interior Penalty Finite Element Approximations of the Advection-Reaction Equation." Siam Journal on Numerical Analysis**47**(5): 3584-3607.

44. L. Schoof, V. Yarberry, Exodus ii (1994): A finite element data model, Tech. Rep.SAND92-2137, Sandia National Laboratories.

45. M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tumi-naro, J.Willenbring, A. Williams(2003):An Overview of Trilinos, Tech. Rep.SAND2003-2927, SandiaNational Laboratories.

46. M. A. Heroux, J. M. Willenbring, Trilinos Users Guide, Tech. Rep. SAND2003 2952, Sandia National Laboratories (2003).

47. M. Sala, K. Stanley, M. Heroux, Amesos (2006): A set of general interfaces to sparse direct solver libraries, in: Proceedings of PARA'06 Conference, Umea, Sweden.

48. M. A. Heroux, AztecOO User Guide, Tech. Rep. SAND2004-3796, SandiaNational Laboratories (2007).

49. Chung, T. J. (1977). Finite element analysis in fluid dynamics. New York, McGraw-Hill International Book Co.

50. Oden, J. T. and J. N. Reddy (1976). An introduction to the mathematical theory of finite elements. New York, Wiley.

51. Donéa, J. and A. Huerta (2003). Finite element methods for flow problems. Chichester; Hoboken, NJ, Wiley.

52. Kondo, S. and T. Miura (2010). "Reaction-Diffusion Model as a Framework for Understanding Biological Pattern Formation." Science**329**(5999): 1616-1620.

53. Zhu, J., J. X. Qiu, et al. (2011). "RKDG methods with WENO type limiters and conservative interfacial procedure for one-dimensional compressible multi-medium flow simulations." Applied Numerical Mathematics**61**(4): 554-580.

54. Lew, A. J. and G. C. Buscaglia (2008). "A discontinuous-Galerkin-based immersed boundary method." International Journal for Numerical Methods in Engineering**76**(4): 427-454.

55. Crivellini, A. and F. Bassi (2011). "An implicit matrix-free Discontinuous Galerkin solver for viscous and turbulent aerodynamic simulations." Computers & Fluids**50**(1): 81-93.

APPENDICES

APPENDIX A

OPERATOR FORMULATION

The following appendices are C++ codes used to implement the discontinuous Galerkin Finite Element Method. Only selected code sections are included here, and the full source code is available upon request. The first appendix includes the formulation of the Jacobian and Vandermonde matrices, the calculation of two-dimensional orthonormal polynomial basis functions, and the computation of the warp functions which is used to transform the reference triangle to the regular triangle.

```
void DGLocalOps::JacobiP(double *pts, double *P, int alpha, int beta,
int numpts, int lN)
{ /* Purpose: Evaluate Jacobi Polynomial of type alpha, beta, order = N
at points x and returns P of length numpts.This is an orthonormal basis
in 1D based on a Gram-Schmidt orthogonalization approach.  For more
details see Hesthavenand Warburon page 45 and appendix A. The return
vector P must beallocated by the calling routine, but it is initiallized
here. */
double gamma0, gamma1, aold, anew, bnew, h1;
double **PL;
  PL = new double *[lN+1];
For(int i=0; I < (N+1); i++){
   PL[i] = new double[numpts];
for(int j=0; j<numpts;j++)
      PL[i][j] = 0.0;
}
double ga1 = exp(lgamma((double) alpha+1));
  double gb1 = exp(lgamma((double) beta+1));
  double gab1 = exp(lgamma((double) alpha+beta+1));
  gamma0 = pow(2,alpha + beta + 1) * ga1 * gb1 / ((alpha + beta + 1) *
gab1);
for(int i=0; i<numpts; i++)
PL[0][i] = 1.0 / sqrt(gamma0);
if (lN == 0){
for(int i=0; i< numpts; i++)
      P[i] = PL[0][i];
  } else {
```

```
gamma1 = (alpha + 1.0) * (beta + 1.0) * gamma0 / (alpha + beta + 3.0);
for(int i=0; i< numpts; i++){
PL[1][i] = (alpha+beta+2.0) * pts[i] / 2.0 + (alpha - beta) / 2.0;
PL[1][i] = PL[1][i] / sqrt(gamma1);
    }
if (lN == 1){
for(int i=0; i< numpts; i++)
        P[i] = PL[1][i];
    } else {
aold = 2.0 * sqrt((alpha+1.0)*(beta+1.0)/(alpha+beta+3.0));
aold = aold /(2.0 + alpha + beta);
for(int i=1; i<lN; i++){
        h1 = 2.0 * i + alpha + beta;
anew = 2.0*sqrt((i+1.0)*(i+1.0+alpha+beta)*(i+1.0+alpha)*(i+1.0+beta)/
            ((h1+1.0)*(h1+3.0)))/(h1+2.0);
bnew = -(alpha*alpha - beta*beta)/(h1 * (h1+2.0));
for(int j=0; j<numpts; j++)
PL[i+1][j] = ((-aold * PL[i-1][j])+((pts[j]-bnew) * PL[i][j]))/anew;
aold = anew;
      }
for(int i=0; i< numpts; i++)
        P[i] = PL[lN][i];
    }
  }
for(int i=0; i<(lN+1); i++)
delete[] PL[i];
delete[] PL;
}
void DGLocalOps::Vandermonde2D()
{
  /* Build the 2D Vandermonde matrix, V2D, which is a public member of
the
class, for transforming a modal to a nodal representation. */
double *a, *b, *Vtmp;
int sk;
  a = new double[Np];
b = new double[Np];
  Vtmp = new double[Np];
RStoAB(a, b);
```

```
for (int i=0; i < Np; i++){
for (int j=0; j < Np; j++){
      V2D[i][j] = 0.0;
    }
  }
sk = 0;
for (int i=0; i < N+1; i++){
for (int j=0; j < N-i+1; j++){
Simplex2DP(a, b, i, j, Vtmp);
for(int k=0; k < Np; k++)
        V2D[k][sk] = Vtmp[k];
sk = sk+1;
    }
  }
delete[] a;  delete[] b;
delete[] Vtmp;
}
void DGLocalOps::Simplex2DP(double *a, double *b, int i, int j, double
*P)
{
  /* Evaluates the 2D orthonormal polynomial basis at points (a,b) and
returns the value of the polynomial *P of order (i,j).  The vector
     P must be allocated by the calling function. They are pretty
pictures
of the basis functions in Hesthaven and Warburton on page 174. */

double *h1, *h2;

  h1 = new double[Np];
  h2 = new double[Np];
JacobiP(a, h1, 0, 0, Np, i);
JacobiP(b, h2, 2*i+1, 0, Np, j);
for(int k=0; k<Np; k++){
    P[k] = sqrt(2.0)*h1[k]*h2[k]*pow((1-b[k]),i);
  }
delete[] h1;
delete[] h2;
}
void DGLocalOps::Nodes2D(){
```

```
  /* Purpose: Compute x and y node locations in equilateral triangle for
polynomial of order N.  The nodes are compressed at the vertices and
some wonderful pictures can be seen on page 180 of Hesthaven and
Warburton.  These nodes are later mapped to the standard or reference
triangle by the XYtoRS function. */
double alpopt[] = {0.0, 0.0, 1.4152, 0.1001, 0.2751, 0.9800, 1.0999,
      1.2832, 1.3648, 1.4773, 1.4959, 1.5743, 1.5770, 1.6223, 1.6258};
double alpha;
double *L1, *L2, *L3;
double *L32, *L13, *L21;
double *blend1, *blend2, *blend3;
double *warp1, *warp2, *warp3;
  // Set Optimized parameter, alpha, depending on order N
alpha = alpopt[N-1];
  // Create equidistributed nodes on equilateral triangle
  L1 = new double[Np]; L2 = new double[Np]; L3 = new double[Np];
  warp1 = new double[Np]; warp2 = new double[Np]; warp3 = new
double[Np];
  L32 = new double[Np]; L13 = new double[Np]; L21 = new double[Np];
  blend1 = new double[Np]; blend2 = new double[Np];
  blend3 = new double[Np];
int sk=0;
for (int n=0; n<(N+1); n++){
for (int m=0; m<(N+1-n); m++){
L1[sk] = (double) n/ (double) N;
L3[sk] = (double) m/ (double) N;
sk++;
    }
  }
for (int n=0; n<Np; n++)
L2[n] = 1.0 - L1[n] - L3[n];
for (int n=0; n<Np; n++){
x[n] = -L2[n] + L3[n];
    y[n] = (-L2[n] - L3[n] + 2*L1[n])/sqrt(3.0);
}
  // Compute blending function at each node for each edge
for (int n=0; n<Np; n++){
blend1[n] = 4.0 * L2[n] * L3[n];
blend2[n] = 4.0 * L1[n] * L3[n];
```

```
blend3[n] = 4.0 * L1[n] * L2[n];
L32[n] = L3[n] - L2[n];
L13[n] = L1[n] - L3[n];
L21[n] = L2[n] - L1[n];
  }
Warpfactor(L32, Np, warp1);
Warpfactor(L13, Np, warp2);
Warpfactor(L21, Np, warp3);
  // Combine blend and warp
for (int n=0; n<Np; n++){
warp1[n] = blend1[n] * warp1[n] * (1.0 + pow((alpha * L1[n]),2.0));
warp2[n] = blend2[n] * warp2[n] * (1.0 + pow((alpha * L2[n]),2.0));
warp3[n] = blend3[n] * warp3[n] * (1.0 + pow((alpha * L3[n]),2.0));
  }
  // Accumulate deformations associated with each edge
for (int n=0; n<Np; n++){
x[n] = x[n] + 1.0 * warp1[n] + cos(2.0 * PI / 3.0) * warp2[n] +
cos(4.0 * PI / 3.0) * warp3[n];
y[n] = y[n] + 0.0 * warp1[n] + sin(2.0 * PI / 3.0) * warp2[n] +
sin(4.0 * PI / 3.0) * warp3[n];
  }
delete[] blend1; delete[] blend2; delete[] blend3;
delete[] L1; delete[] L2; delete[] L3;
delete[] L13; delete[] L21; delete[] L32;
delete[] warp1; delete[] warp2; delete[] warp3;
}
```

APPENDIX B

EVALUATION AT CUBATURE POINTS

After building the necessary operators and transformations, the evaluation of these operators at cubatures points for the reference elements nodes along each face is included in this appendix. In addition, this section also has the C++ code for the calculation of the $Lift$ and $Fmask$ functions to evaluate surface integrals.

```
Cubature::Cubature(TriMesh *mesh, DGLocalOps *localOps, DGGlobalOps
*globalOps) {
  /* Constructor */
  N = localOps->N;
  Np = localOps->Np;
  NumElem = mesh->NumElem;
  Nint = (3.0 * N / 2.0 + 0.5);
  CubatureOrder = 2*(Nint+1);
Cubature2D();
AllocVars();
InterpMatrix2D(localOps);
Dmatrices2D(localOps);
GeometricFactors2D(localOps, globalOps);

double *diag;
diag = new double[Ncub];
for(int e=0; e<NumElem; e++){
for(int i=0; i<Ncub; i++)
diag[i] = Jac[i][e] * cubW[i];
for(int i=0; i<Np; i++){
for(int j=0; j<Np; j++){
for(int k=0; k<Ncub; k++){
mm[i][j][e] += cubV[k][i] * diag[k] * cubV[k][j]; // mass matrix
        }
      }
    }
for(int i=0; i<Ncub; i++){
      W[i][e] = cubW[i] * Jac[i][e];
for(int j=0; j<Np; j++){
x[i][e] += cubV[i][j] * globalOps->x[j][e];
```

```
y[i][e] += cubV[i][j] * globalOps->y[j][e];
      }
    }
  }
}
void DGLocalOps::BuildFmask()
{
  /* Builds Fmask, a public data member of the class, which has size
     Nfaces x Nfp, and contains a list of reference element nodes along
each face in (r,s) space (i.e., the reference element space). */

int num1 = 0, num2 = 0, num3 = 0;

for(int i=0; i<Np; i++){
if(abs(s[i] + 1.0) < NODETOL)
Fmask[num1++][0] = i;
if(abs(r[i] + s[i]) < NODETOL)
Fmask[num2++][1] = i;
if(abs(r[i] + 1.0) < NODETOL)
Fmask[num3++][2] = i;
if((num1 > Nfp) or (num2 > Nfp) or (num3 > Nfp)){
cout<< "ERROR in BuildFmask -- Change NODETOL?" <<endl;
break;
    }
  }
}
void DGLocalOps::Lift2D()
{
  /* Compute the surface to volume lift term, Lift, which is a public
member
of the class.  The size of Lift is Np x Nfaces * Nfp. */
double **Emat, **V1D, **V1Dsq;
double *faceR, *faceS;
faceR = new double [Nfp];
faceS = new double [Nfp];
Emat = new double *[Np];
  V1D = new double *[Nfp];
  V1Dsq = new double *[Nfp];
for(int i=0; i<Np; i++){
```

```
    Emat[i] = new double[Nfaces*Nfp];
for(int j=0; j<(Nfaces*Nfp); j++)
      Lift[i][j] = 0.0;
  }
for(int i=0; i<Nfp; i++){
    V1D[i] = new double[Nfp];
    V1Dsq[i] = new double[Nfp];
  }
  // Face 1
for(int i=0; i<Nfp; i++)
faceR[i] = r[Fmask[i][0]];
Vandermonde1D(faceR, V1D, Nfp);
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
      V1Dsq[i][j] = 0.0;
for(int k=0; k<Nfp; k++){
        V1Dsq[i][j] += V1D[i][k] * V1D[j][k];
      }
    }
  }
DirInv(V1Dsq, Nfp);
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
Lift[Fmask[i][0]][j] = V1Dsq[i][j];
    }
  }
  // Face 2
for(int i=0; i<Nfp; i++)
faceR[i] = r[Fmask[i][1]];
Vandermonde1D(faceR, V1D, Nfp);
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
      V1Dsq[i][j] = 0.0;
for(int k=0; k<Nfp; k++){
        V1Dsq[i][j] += V1D[i][k] * V1D[j][k];
      }
    }
  }
DirInv(V1Dsq, Nfp);
```

```
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
Lift[Fmask[i][1]][Nfp+j] = V1Dsq[i][j];
    }
  }
 // Face 3
for(int i=0; i<Nfp; i++)
faceS[i] = s[Fmask[i][2]];
Vandermonde1D(faceS, V1D, Nfp);
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
      V1Dsq[i][j] = 0.0;
for(int k=0; k<Nfp; k++){
        V1Dsq[i][j] += V1D[i][k] * V1D[j][k];
      }
    }
  }
DirInv(V1Dsq, Nfp);
for(int i=0; i<Nfp; i++){
for(int j=0; j<Nfp; j++){
Lift[Fmask[i][2]][2*Nfp+j] = V1Dsq[i][j];
    }
  }
for(int i=0; i<Np; i++){
for(int j=0; j<(Nfp*Nfaces); j++){
      Emat[i][j] = 0.0;
for(int k=0; k<Np; k++){
        Emat[i][j] += V2D[k][i] * Lift[k][j];
      }
    }
  }
for(int i=0; i<Np; i++){
for(int j=0; j<(Nfp*Nfaces); j++){
      Lift[i][j] = 0.0;
for(int k=0; k<Np; k++){
        Lift[i][j] += V2D[i][k] * Emat[k][j];
      }
}
  }
```

```
for(int i=0; i<Np; i++)
delete[] Emat[i];
for(int i=0; i<Nfp; i++){
delete[] V1D[i];
delete[] V1Dsq[i];
  }
delete[] V1D; delete[] V1Dsq; delete[] Emat;
delete[] faceR; delete[] faceS;
}
```

APPENDIX C

THE ASSEMBLE PROCESS

This section is the assemble process after everything is computed. In this section, the assemble process for the grid and the global matrix is carried out. The appendix will give you the global object of the code.

```
#include "dg-main.h"
DGGlobalOps::DGGlobalOps(TriMesh *mesh, DGLocalOps *localOps)
{
  /* CONSTRUCTOR */
  Np = localOps->Np;
  NumElem = mesh->NumElem;
  NumNodes = mesh->NumNodes;
AllocVars(localOps);
double loc1, loc2, loc3;
for(int e=0; e<NumElem; e++){
    loc1 = mesh->getNodeLocX(mesh->EToV[0][e]);
    loc2 = mesh->getNodeLocX(mesh->EToV[1][e]);
    loc3 = mesh->getNodeLocX(mesh->EToV[2][e]);
for(int i=0; i< Np; i++){
x[i][e] = -0.5 *(localOps->r[i] + localOps->s[i]) * loc1;
x[i][e] += 0.5 *(1.0 + localOps->r[i]) * loc2;
x[i][e] += 0.5 *(1.0 + localOps->s[i]) * loc3;
    }
    loc1 = mesh->getNodeLocY(mesh->EToV[0][e]);
    loc2 = mesh->getNodeLocY(mesh->EToV[1][e]);
    loc3 = mesh->getNodeLocY(mesh->EToV[2][e]);
for(int i=0; i< Np; i++){
y[i][e] = -0.5 *(localOps->r[i] + localOps->s[i]) * loc1;
y[i][e] += 0.5 *(1.0 + localOps->r[i]) * loc2;
y[i][e] += 0.5 *(1.0 + localOps->s[i]) * loc3;
    }
  }
BuildF(mesh, localOps);
GeometricFactors2D(localOps);
Normals2D(localOps);
```

```
triConnect2D(mesh);
BuildMaps2D(mesh, localOps);
BuildBCMaps2D(mesh, localOps->Nfp);
}
void DGGlobalOps::BuildF(TriMesh *mesh, DGLocalOps *localOps)
{
  int col;
  for(int e=0; e<NumElem; e++){
    for(int j=0; j<(localOps->Nfaces); j++){
      for(int i=0; i<(localOps->Nfp); i++){
        col = j * localOps->Nfp + i;
        Fx[col][e] = x[localOps->Fmask[i][j]][e];
        Fy[col][e] = y[localOps->Fmask[i][j]][e];
      }
    }
  }
}
void DGGlobalOps::GeometricFactors2D(DGLocalOps *localOps)
{
  /* Computes the physical to reference element mappings and derivatives
     on the reference element. Specifically, this routine calculates
     drdx, dsdx, drdy, dsdy, and Jac (the jacobian of the mapping).  See
     page 172 in Hesthaven and Warburton for the details. */
  double **dxdr, **dxds, **dydr, **dyds;
  dxdr = new double *[Np];
dxds = new double *[Np];
dydr = new double *[Np];
  dyds = new double *[Np];
  for(int i=0; i<Np; i++){
    dxdr[i] = new double [NumElem];
    dxds[i] = new double [NumElem];
    dydr[i] = new double [NumElem];
    dyds[i] = new double [NumElem];
    for(int j=0; j<NumElem; j++){
      dxdr[i][j] = dxds[i][j] = dydr[i][j] = dyds[i][j] = 0.0;
      drdx[i][j] = dsdx[i][j] = drdy[i][j] = dsdy[i][j] = 0.0;
      Jac[i][j] = 0.0;
    }
  }
```

```
  for(int i=0; i<Np; i++){
    for(int j=0; j<NumElem; j++){
      for(int k=0; k<Np; k++){
        dxdr[i][j] += localOps->Dr[i][k] * x[k][j];
        dxds[i][j] += localOps->Ds[i][k] * x[k][j];
        dydr[i][j] += localOps->Dr[i][k] * y[k][j];
        dyds[i][j] += localOps->Ds[i][k] * y[k][j];
      }
    }
  }
  for(int i=0; i<Np; i++){
    for(int j=0; j<NumElem; j++){
      Jac[i][j] = -dxds[i][j] * dydr[i][j] + dxdr[i][j] * dyds[i][j];
      drdx[i][j] = dyds[i][j] / Jac[i][j];
      dsdx[i][j] = -dydr[i][j] / Jac[i][j];
      drdy[i][j] = -dxds[i][j] / Jac[i][j];
      dsdy[i][j] = dxdr[i][j] / Jac[i][j];
    }
  }
  for(int i=0; i<Np; i++){
    delete[] dxdr[i]; delete[] dxds[i];
    delete[] dydr[i]; delete[] dyds[i];
  }
  delete[] dxdr; delete[] dxds;
  delete[] dydr; delete[] dyds;

}
void DGGlobalOps::Normals2D(DGLocalOps *localOps)
{
  /* Compute the outward pointing normals at each point on each face of
each
     element (stored in public arrays nx and ny [Nfaces * Nfp][NumElem])
and
     surface jacobians (stored in sJ). See page 191 in Hesthaven and
     Warburton. */
  double **dxdr, **dxds, **dydr, **dyds;
  double **fxr, **fxs, **fyr, **fys;
  int col;
  dxdr = new double *[Np];
```

```
  dxds = new double *[Np];
  dydr = new double *[Np];
  dyds = new double *[Np];
fxr = new double *[facePts];
  fxs = new double *[facePts];
  fyr = new double *[facePts];
  fys = new double *[facePts];
  for(int i=0; i<Np; i++){
    dxdr[i] = new double [NumElem];
    dxds[i] = new double [NumElem];
    dydr[i] = new double [NumElem];
    dyds[i] = new double [NumElem];
    for(int j=0; j<NumElem; j++){
      dxdr[i][j] = dxds[i][j] = dydr[i][j] = dyds[i][j] = 0.0;
    }
  }
  for(int i=0; i<facePts; i++){
    fxr[i] = new double [NumElem];
    fxs[i] = new double [NumElem];
    fyr[i] = new double [NumElem];
    fys[i] = new double [NumElem];
    for(int j=0; j<NumElem; j++){
      fxr[i][j] = fxs[i][j] = fyr[i][j] = fys[i][j] = 0.0;
      nx[i][j] = ny[i][j] = sJ[i][j] = Fscale[i][j] = 0.0;
    }
  }
  for(int i=0; i<Np; i++){
    for(int j=0; j<NumElem; j++){
      for(int k=0; k<Np; k++){
        dxdr[i][j] += localOps->Dr[i][k] * x[k][j];
        dxds[i][j] += localOps->Ds[i][k] * x[k][j];
        dydr[i][j] += localOps->Dr[i][k] * y[k][j];
        dyds[i][j] += localOps->Ds[i][k] * y[k][j];
      }
    }
  }
  for(int k=0; k<NumElem; k++){
    for(int j=0; j<(localOps->Nfaces); j++){
      for(int i=0; i<(localOps->Nfp); i++){
```

```
        col = j*(localOps->Nfp) + i;
        fxr[col][k] = dxdr[localOps->Fmask[i][j]][k];
        fxs[col][k] = dxds[localOps->Fmask[i][j]][k];
        fyr[col][k] = dydr[localOps->Fmask[i][j]][k];
        fys[col][k] = dyds[localOps->Fmask[i][j]][k];
      }
    }
  }
  for(int k=0; k<NumElem; k++){
    for(int i=0; i<(localOps->Nfp); i++){
      nx[i][k] = fyr[i][k];
      ny[i][k] = -fxr[i][k];
    }
for(int i=(localOps->Nfp); i<(2*(localOps->Nfp)); i++){
      nx[i][k] = fys[i][k] - fyr[i][k];
      ny[i][k] = -fxs[i][k] + fxr[i][k];
    }
    for(int i=(2*(localOps->Nfp)); i<(3*(localOps->Nfp)); i++){
      nx[i][k] = -fys[i][k];
      ny[i][k] = fxs[i][k];
    }
  }
  // Normalize
  for(int k=0; k<NumElem; k++){
    for(int i=0; i<facePts; i++){
      sJ[i][k] = sqrt(nx[i][k] * nx[i][k] + ny[i][k] * ny[i][k]);
      nx[i][k] = nx[i][k] / sJ[i][k];
      ny[i][k] = ny[i][k] / sJ[i][k];
    }
  }
  for(int k=0; k<NumElem; k++){
    for(int j=0; j<(localOps->Nfaces); j++){
      for(int i=0; i<(localOps->Nfp); i++){
        col = j*(localOps->Nfp) + i;
        Fscale[col][k] = sJ[col][k] / Jac[localOps->Fmask[i][j]][k];
      }
    }
  }
  for(int i=0; i<Np; i++){
```

```
    delete[] dxdr[i]; delete[] dxds[i];
    delete[] dydr[i]; delete[] dyds[i];
  }
  for(int i=0; i<facePts; i++){
    delete[] fxr[i]; delete[] fxs[i];
    delete[] fyr[i]; delete[] fys[i];
  }
  delete[] dxdr; delete[] dxds;
  delete[] dydr; delete[] dyds;
  delete[] fxr; delete[] fxs;
  delete[] fyr; delete[] fys;

}
void DGGlobalOps::triConnect2D(TriMesh *mesh)
{
  // Purpose: triangle face connect algorithm (original by Toby Isaac).
  // This function builds the EToE (Element to Element) and EToF
(Element
  // to Face) connection information.  In other words, EToE[e][f]
returns
  // the element number connected to e on face f, and EToF[e][f] returns
the
  // face number for that neighboring element.

  int **matched, *id, node0, node1;
  id = new int [3*NumElem];
  matched = new int *[3];
  for(int i=0; i<3; i++){
    matched[i] = new int[NumElem];
    for(int j=0; j<NumElem; j++){
      EToE[i][j] = j;
      EToF[i][j] = i;
      matched[i][j] = 0;
    }
  }
  for(int e=0; e<NumElem; e++){
    if(mesh->EToV[0][e] < mesh->EToV[1][e]){
      node0 = mesh->EToV[0][e];
      node1 = mesh->EToV[1][e];
```

```
  } else {
    node0 = mesh->EToV[1][e];
    node1 = mesh->EToV[0][e];
  }
  id[e] = node0 * NumNodes + node1 + 1;
  if(mesh->EToV[1][e] < mesh->EToV[2][e]){
    node0 = mesh->EToV[1][e];
    node1 = mesh->EToV[2][e];
  } else {
    node0 = mesh->EToV[2][e];
    node1 = mesh->EToV[1][e];
  }
  id[e+NumElem]=node0 * NumNodes + node1 + 1;
  if(mesh->EToV[2][e] < mesh->EToV[0][e]){
    node0 = mesh->EToV[2][e];
    node1 = mesh->EToV[0][e];
  } else {
    node0 = mesh->EToV[0][e];
    node1 = mesh->EToV[2][e];
  }
  id[e+2*NumElem]=node0 * NumNodes + node1 + 1;
}
// Scan id list for matching edge
int myid;
for(int e=0; e<NumElem; e++){
  for(int f=0; f<3; f++){
    myid = id[f*NumElem + e];
    if(matched[f][e] == 0){
      for(int e2=0; e2<NumElem; e2++){
        for(int f2=0; f2<3; f2++){
          if((myid == id[f2*NumElem + e2]) and (e != e2)){
            EToE[f][e] = e2;
            EToF[f][e] = f2;
            EToE[f2][e2] = e;
            EToF[f2][e2] = f;
            matched[f2][e2] = 1;
            f2 = 3;
            e2 = NumElem;
          }
```

```
          }
        }
      }
    }
  }
/*
  for(int e=0; e<NumElem; e++){
    for(int f=0; f<3; f++){
      cout << EToE[f][e] << " ";
    }
    cout << "   ";
    for(int f=0; f<3; f++){
cout << EToF[f][e] << " ";
    }
    cout << endl;
}
*/
  for(int i=0; i<3;i++)
    delete[] matched[i];
  delete[] matched;
  delete[] id;

}
void DGGlobalOps::BuildMaps2D(TriMesh *mesh, DGLocalOps *localOps)
{
  // Purpose: Connectivity and boundary tables in the K # of Np elements
  //
  // Summary of data arrays:
  // nodeids -- a unique number to every node in the domain (length:
E*Np)
  // mapM -- a unique number to every element edge node
  //         (length: E * Nfaces * Nfp), what's the point of keeping
this?
  // vmapM -- map from mapM to nodeids
  // vmapP -- map to neighbor nodeid for adjacent (E, face, node)
  // vmapB -- lists of nodeids for boundary edges
  // mapP -- map to neighbor mapM id
  // mapB -- list of mapM ids for boundary edges
  int **nodeids, k2, f2, v1, v2, row, *vidM, *vidP;
```

```
int *idP, nd, totaln;
  double refd, *x1, *x2, *y1, *y2, D;
nodeids = new int *[Np];
  for(int i=0; i<Np; i++){
    nodeids[i] = new int[NumElem];
    for(int j=0; j<NumElem; j++){
      nodeids[i][j] = j*Np+i;
    }
  }
  totaln = localOps->Nfp * localOps->Nfaces * NumElem;
  x1 = new double[localOps->Nfp];
  x2 = new double[localOps->Nfp];
  y1 = new double[localOps->Nfp];
  y2 = new double[localOps->Nfp];
  vidM = new int[localOps->Nfp];
  vidP = new int[localOps->Nfp];
  idP = new int[localOps->Nfp];
  for(int e=0; e<NumElem; e++){
    for(int f=0; f<(localOps->Nfaces); f++){
      for(int i=0; i<(localOps->Nfp); i++){
        row = e * (localOps->Nfaces) * (localOps->Nfp);
        row += f * (localOps->Nfp) + i;
        mapM[row] = row;
        mapP[row] = row;
        vmapM[row] = nodeids[localOps->Fmask[i][f]][e];
        vmapP[row] = 0;
      }
    }
  }
  for(int k1=0; k1<NumElem; k1++){
    for(int f1=0; f1<(localOps->Nfaces); f1++){
      // get neighbor
      k2 = EToE[f1][k1];
      f2 = EToF[f1][k1];
      // get reference length of edge
      v1 = mesh->EToV[f1][k1];
      v2 = mesh->EToV[(f1+1)%(localOps->Nfaces)][k1];
      refd = pow((mesh->getNodeLocX(v1) - mesh->getNodeLocX(v2)),2.0);
      refd += pow((mesh->getNodeLocY(v1) - mesh->getNodeLocY(v2)),2.0);
```

```
      refd = sqrt(refd);
      // Find volume node numbers of left and right nodes
      for(int i=0; i<(localOps->Nfp); i++){
        row = f1 * (localOps->Nfp) + i;
        row += k1 * (localOps->Nfaces) * (localOps->Nfp);
        vidM[i] = vmapM[row];
        x1[i] = x[localOps->Fmask[i][f1]][k1];
        y1[i] = y[localOps->Fmask[i][f1]][k1];
        for(int j=0; j<(localOps->Nfp); j++){
          row = f2 * (localOps->Nfp) + j;
          row += k2 * (localOps->Nfaces) * (localOps->Nfp);
          vidP[j] = vmapM[row];
          x2[j] = x[localOps->Fmask[j][f2]][k2];
y2[j] = y[localOps->Fmask[j][f2]][k2];
D = pow((x1[i] - x2[j]),2) + pow((y1[i] - y2[j]),2);
          if(sqrt(abs(D)) < (refd*localOps->NODETOL)){
            idP[i] = j;
          }
        }
        row = f1 * (localOps->Nfp) + i;
        row += k1 * (localOps->Nfaces) * (localOps->Nfp);
        vmapP[row] = vidP[idP[i]];
        mapP[row] = f2 * (localOps->Nfp) + idP[i];
        mapP[row] += k2 * (localOps->Nfaces) * (localOps->Nfp);
      }
    }
  }
  int Bcount = 0;
  for(int i=0; i<totaln; i++){
    if(vmapP[i] == vmapM[i]){
      Bcount++;
    }
  }
  mapB = new int [Bcount];
  vmapB = new int [Bcount];
  Bcount = 0;
  for(int i=0; i<totaln; i++){
    if(vmapP[i] == vmapM[i]){
      mapB[Bcount] = i;
```

```
      vmapB[Bcount] = vmapM[mapB[Bcount]];
      Bcount++;
    }
  }
  for(int i=0; i<Np; i++){
    delete[] nodeids[i];
  }
  delete[] nodeids;
  delete[] vidM; delete[] vidP; delete[] idP;
  delete[] x1; delete[] x2; delete[] y1; delete[] y2
}
void DGGlobalOps::BuildBCMaps2D(TriMesh *mesh, int Nfp)
{
   /* Build similar maps to BuildMaps2D, but this time they are lists of
nodes
      for specific boundary conditions.  The vmap* data arrays hold data
      the list of boundary conditions nodes in terms of face nodes. */
   int numIn, numOut, numWall, numCyl;
   int Nfaces, row;
   numIn=0; numOut=0; numWall=0; numCyl=0;
   Nfaces = 3;
   for(int e=0; e<NumElem; e++){
     for(int i=0; i<Nfaces; i++){
       if(mesh->BCType[i][e] == INBC)
         numIn++;
       if(mesh->BCType[i][e] == OUTBC)
         numOut++;
       if(mesh->BCType[i][e] == WALLBC)
         numWall++;
       if(mesh->BCType[i][e] == CYLBC)
         numCyl++;
     }
   }

   mapIn = new int[(numIn*Nfp)];
   mapOut = new int[(numOut*Nfp)];
   mapWall = new int[(numWall*Nfp)];
   mapCyl = new int[(numCyl*Nfp)];
   vmapIn = new int[(numIn*Nfp)];
```

```
    vmapOut = new int[(numOut*Nfp)];
    vmapWall = new int[(numWall*Nfp)];
    vmapCyl = new int[(numCyl*Nfp)];

    numIn=0; numOut=0; numWall=0; numCyl=0;
    for(int e=0; e<NumElem; e++){
      for(int i=0; i<Nfaces; i++){
        row = e * Nfp * Nfaces + i * Nfp;
        if(mesh->BCType[i][e] == INBC){
          for(int j=0; j<Nfp; j++){
            mapIn[numIn] = row + j;
            vmapIn[numIn] = vmapM[row+j];
            numIn++;
          }
        }
        if(mesh->BCType[i][e] == OUTBC){
          for(int j=0; j<Nfp; j++){
            mapOut[numOut] = row + j;
            vmapOut[numOut] = vmapM[row+j];
            numOut++;
          }
        }
        if(mesh->BCType[i][e] == WALLBC){
          for(int j=0; j<Nfp; j++){
            mapWall[numWall] = row + j;
            vmapWall[numWall] = vmapM[row + j];
            numWall++;
          }
        }
        if(mesh->BCType[i][e] == CYLBC){
          for(int j=0; j<Nfp; j++){
            mapCyl[numCyl] = row + j;
            vmapCyl[numCyl] = vmapM[row + j];
            numCyl++;
          }
        }
      }
    }
}
```

APPENDIX D

HEAT EQUATION EXAMPLE

After everything is assembled, this appendix will provide a brief example of the heat equation, which is built upon the previous three appendices. This section includes the core of algorithm of the heat equation. A complete code is available upon request.

```
#include "dg-ins.h"
void INSSolver::INSPressureSetup(Epetra_Map map, TriMesh *mesh,
    DGLocalOps *localOps, DGGlobalOps *globalOps, Cubature *cub,
    FaceMesh *face)
{
  int type, k2, f2, col, row;
  int *indices, *bcindices;
  int *rowindices, NumEntries, Length;
  double *zeros, *bczeros, *rowvalues;
  double frhs, xloc, yloc;
  Epetra_Vector PRrhs(map);
  Epetra_Vector sol(map);
  indices = new int [Np];
  bcindices = new int [NGauss];
  rowindices = new int [Nfaces * NGauss];
  zeros = new double [Np];
  bczeros = new double [NGauss];
  rowvalues = new double [Nfaces * NGauss];
  // I may need to zero out PRsysBC and PRsys here!
  for(int k1=0; k1<NumElem; k1++){
for(int i=0; i<Np; i++){
      for(int j=0; j<Np; j++){
indices[j] = k1*Np+j;
zeros[j] = 0.0;
      }
PRsys->InsertMyValues(k1*Np+i, Np, zeros, indices);
   MMsys->InsertMyValues(k1*Np+i, Np, zeros, indices);
    }
    for(int f1=0; f1<Nfaces; f1++){
      type = mesh->BCType[f1][k1];
      k2 = globalOps->EToE[f1][k1];
      if(type == 0){
```

```
        for(int i=0; i<Np; i++){
          for(int j=0; j<Np; j++){
indices[j] = k2*Np+j;
zeros[j] = 0.0;
          }
          PRsys->InsertMyValues(k1*Np+i, Np, zeros, indices);
        }
      }
      if(type){
        for(int i=0; i<Np; i++){
          for(int j=0; j<NGauss; j++){
bcindices[j] = k1*Nfaces*NGauss + f1*NGauss + j;
            bczeros[j] = 0.0;
          }
          PRsysBC->InsertMyValues(k1*Np+i, NGauss, bczeros, bcindices);
}
      }
    }
  }
  PRsys->FillComplete();
  PRsysBC->FillComplete();
  MMsys->FillComplete();
  PressureIPDGbc2D(mesh, localOps, globalOps, face);
  for(int i=0; i<NumElem*Np; i++){
    refrhsbcPR[i] = 0.0;
    for(int j=0; j<Nfaces*NGauss; j++){
      rowvalues[j] = 0.0;
      rowindices[j] = 0;
    }
    PRsysBC->ExtractMyRowCopy(i, Nfaces*NGauss, NumEntries, rowvalues,
        rowindices);
    for(int c=0; c<NumEntries; c++){
      int j, k;
col = rowindices[c];
j = (int) col/(Nfaces*NGauss);
k = col % (Nfaces*NGauss);
      refrhsbcPR[i] += rowvalues[c] * refbcPR[k][j];
    }
    PRrhs[i] = refrhsbcPR[i];
```

```
  }
  PressureIPDG2D(mesh, localOps, globalOps, cub, face);

  for(int k1=0; k1<NumElem; k1++){
    for(int i=0; i<Np; i++){
      if(Length=MMsys->NumMyEntries(k1*Np+i) != Np)
        cout << "ERROR: missing entries in MMsys" << endl;
      MMsys->ExtractMyRowCopy(k1*Np+i, Np, NumEntries, zeros);
      frhs = 0.0;
      for(int j=0; j<Np; j++){
indices[j] = k1*Np+j;
xloc = globalOps->x[j][k1];
        yloc = globalOps->y[j][k1];

        frhs += 2.0 * PI * PI * sin(xloc * PI) * sin(yloc * PI) *
zeros[j];
      }
      PRrhs[k1*Np+i] += frhs;
    }
  }
  Epetra_LinearProblem *PR_LinProb = new Epetra_LinearProblem;
  PR_LinProb->SetOperator(PRsys);
  PR_LinProb->SetRHS(&PRrhs);
  PR_LinProb->SetLHS(&sol);
  if(!trilinos_solver.compare("KLU")){
    Teuchos::ParameterList AmesosParams;
    Amesos_BaseSolver *PRSolver;
    Amesos PR_Factory;
    string Choice = "Amesos_Klu";
    PRSolver = PR_Factory.Create(Choice, *PR_LinProb);
    assert (PRSolver);
    PRSolver->NumericFactorization();
    PRSolver->Solve();
    delete PR_LinProb;
    delete PRSolver;
  } else if (!trilinos_solver.compare("Aztec")){
    AztecOO Solver(*PR_LinProb);
    Solver.SetAztecOption(AZ_solver, AZ_gmres);
    Solver.Iterate(10000, 1e-8);
```

```
  } else{
    Teuchos::ParameterList MLList;
    ML_Epetra::MultiLevelPreconditioner* MLPrec = new
ML_Epetra::MultiLevelPreconditioner(*PRsys,MLList,true);
    AztecOO Solver(*PR_LinProb);
    Solver.SetPrecOperator(MLPrec);
    Solver.SetAztecOption(AZ_solver, AZ_gmres);
    Solver.Iterate(10000, 1e-8);
  }
  for(int k=0; k<NumElem; k++){
    for(int i=0; i<Np; i++){
      row = k*Np + i;
      PR[i][k] = sol[row];
    }
  }
  DG_ex_pv_out(globalOps, mesh);
  delete[] indices; delete[] zeros;
  delete[] bcindices; delete[] bczeros;
  delete[] rowindices; delete[] rowvalues;
}
void INSSolver::INSPressure(Epetra_Map map, TriMesh *mesh, DGLocalOps
*localOps,
    DGGlobalOps *globalOps, Epetra_LinearProblem *PR_LinProb,
    Amesos_BaseSolver *PRSolver)
{
  /* Solve the pressure Poisson problem */
  int e1, e2, num1, num2, row;
double **DivUT, **CurlU, **dCurlUdx, **dCurlUdy;
  double **res1, **res2, tmp;
  double **dPRdx, **dPRdy;
  Epetra_Vector PRrhs(map);
  Epetra_Vector sol(map);

  DivUT = new double *[Np];
  CurlU = new double *[Np];
dCurlUdx = new double *[Np];
dCurlUdy = new double *[Np];
  res1 = new double *[Np];
  res2 = new double *[Np];
```

```
  dPRdx = new double *[Np];
  dPRdy = new double *[Np];
  for(int i=0; i<Np; i++){
    DivUT[i] = new double [NumElem];
    CurlU[i] = new double [NumElem];
    dCurlUdx[i] = new double [NumElem];
    dCurlUdy[i] = new double [NumElem];
    res1[i] = new double [NumElem];
    res2[i] = new double [NumElem];
    dPRdx[i] = new double [NumElem];
    dPRdy[i] = new double [NumElem];
  }

  Div2D(DivUT, UxT, UyT, localOps, globalOps);
  Curl2D(CurlU, Ux, Uy, localOps, globalOps);
  Grad2D(dCurlUdx, dCurlUdy, CurlU, localOps, globalOps);

  for(int k=0; k<NumElem; k++){
    for(int i=0; i<Np; i++){
      res1[i][k] = -NUx[i][k] - nu * dCurlUdy[i][k];
      res2[i][k] = -NUy[i][k] + nu * dCurlUdx[i][k];
    }
    for(int i=0; i<(Nfp * Nfaces); i++){
      dpdnold[i][k] = dpdn[i][k];
      dpdn[i][k] = 0.0;
    }
  }
// FOR Neumann boundaryies, apply pressure bc
  for(int i=0; i<(mesh->NumInflowEdges*Nfp); i++){
    e1 = (int) globalOps->mapIn[i] / (Nfp * Nfaces);
    num1 = globalOps->mapIn[i] % (Nfp * Nfaces);
    e2 = (int) globalOps->vmapIn[i] / (Np);
    num2 = globalOps->vmapIn[i] % (Np);
    dpdn[num1][e1] = globalOps->nx[num1][e1] * res1[num2][e2];
    dpdn[num1][e1] += globalOps->ny[num1][e1] * res2[num2][e2];
    dpdn[num1][e1] -= bcdUndt[num1][e1];
  }
  for(int i=0; i<(mesh->NumWallEdges*Nfp); i++){
    e1 = (int) globalOps->mapWall[i] / (Nfp * Nfaces);
```

```
    num1 = globalOps->mapWall[i] % (Nfp * Nfaces);
    e2 = (int) globalOps->vmapWall[i] / (Np);
    num2 = globalOps->vmapWall[i] % (Np);
    dpdn[num1][e1] = globalOps->nx[num1][e1] * res1[num2][e2];
    dpdn[num1][e1] += globalOps->ny[num1][e1] * res2[num2][e2];
    dpdn[num1][e1] -= bcdUndt[num1][e1];
  }
  for(int i=0; i<(mesh->NumCylEdges*Nfp); i++){
    e1 = (int) globalOps->mapCyl[i] / (Nfp * Nfaces);
    num1 = globalOps->mapCyl[i] % (Nfp * Nfaces);
    e2 = (int) globalOps->vmapCyl[i] / (Np);
    num2 = globalOps->vmapCyl[i] % (Np);
    dpdn[num1][e1] = globalOps->nx[num1][e1] * res1[num2][e2];
    dpdn[num1][e1] += globalOps->ny[num1][e1] * res2[num2][e2];
    dpdn[num1][e1] -= bcdUndt[num1][e1];
  }
  for(int k=0; k<NumElem; k++){
    for(int i=0; i<Np; i++){
      row = k * Np + i;
      PRrhs[row] = 0.0;
      sol[row] = 0.0;
      for(int j=0; j<Np; j++){
        tmp = 0.0;
        for(int j2=0; j2<(Nfaces * Nfp); j2++){
          tmp += localOps->Lift[j][j2] * globalOps->sJ[j2][k] * (b0 *
              dpdn[j2][k] + b1 * dpdnold[j2][k]);
        }
        PRrhs[row] += localOps->MassMatrix[i][j] * (globalOps->Jac[j][k]
*
            -DivUT[j][k] * g0 / dt + tmp);
      }
      PRrhs[row] += rhsbcPR[row];
    }
  }
  PR_LinProb->SetRHS(&PRrhs);
  PR_LinProb->SetLHS(&sol);
  PRSolver->Solve();
//  CholSolve(PRsys, PRrhs, sol, NumElem*Np);
  for(int k=0; k<NumElem; k++){
```

```
    for(int i=0; i<Np; i++){
      row = k*Np + i;
      PR[i][k] = sol[row];
    }
  }
  Grad2D(dPRdx, dPRdy,PR, localOps, globalOps);
  for(int k=0; k<NumElem; k++){
    for(int i=0; i<Np; i++){
      UxTT[i][k] = UxT[i][k] - dt * dPRdx[i][k] / g0;
      UyTT[i][k] = UyT[i][k] - dt * dPRdy[i][k] / g0;
    }
  }
  for(int i=0; i<Np; i++){
    delete[] DivUT[i]; delete[] CurlU[i];
    delete[] dCurlUdx[i]; delete[] dCurlUdy[i];
    delete[] dPRdx[i]; delete[] dPRdy[i];
    delete[] res1[i]; delete[] res2[i];
  }
  delete[] DivUT; delete[] CurlU;
  delete[] dCurlUdx; delete[] dCurlUdy;
  delete[] dPRdx; delete[] dPRdy;
  delete[] res1; delete[] res2;
}
void INSSolver::PressureIPDGbc2D(TriMesh *mesh, DGLocalOps *localOps,
    DGGlobalOps *globalOps, FaceMesh *face)
{
  /* Builds the operator for building the pressure RHS */
int type, *indices;
  double **BOP11, hinv;
  double gtau;
  double *Xloc, *Yloc;
  double **DnM;
  double *values;

  indices = new int [NGauss];
  values = new double [NGauss];
  BOP11 = new double *[Np];
  Xloc = new double [Np];
Yloc = new double [Np];
```

```
  DnM = new double *[NGauss];
  for(int i=0; i<Np; i++)
    BOP11[i] = new double [NGauss];
  for(int i=0; i<NGauss; i++)
    DnM[i] = new double [Np];
  for(int k1=0; k1<NumElem; k1++){
    for(int f1=0; f1<Nfaces; f1++){
      type = mesh->BCType[f1][k1];
      if(type){
        for(int i=0; i<Np; i++){
          Xloc[i] = globalOps->x[i][k1];
          Yloc[i] = globalOps->y[i][k1];
        }
        PhysDmatrices(Xloc, Yloc, localOps, face, f1);

        hinv = globalOps->Fscale[f1*Nfp][k1];
        gtau = 20 * (N+1) * (N+1) * hinv;
    for(int i=0; i<NGauss; i++){
          for(int j=0; j<Np; j++){
            DnM[i][j] = face->nx[f1*NGauss + i][k1] * dVdxM[i][j];
            DnM[i][j] += face->ny[f1*NGauss + i][k1] * dVdyM[i][j];
          }
        }
      }
/* Dirichlet Boundaries */
      if(type){
        for(int i=0; i<Np; i++){
          for(int j=0; j<NGauss; j++){
            BOP11[i][j] = face->interp[f1*NGauss+j][i] *
                 face->W[f1*NGauss+j][k1] * gtau;
            BOP11[i][j] -= DnM[j][i] * face->W[f1*NGauss+j][k1];
          }
        }
      }
/* Neuman Boundaries
      if((type == INBC) or (type == WALLBC) or (type == CYLBC)){
        for(int i=0; i<Np; i++){
          for(int j=0; j<NGauss; j++){
            BOP11[i][j] = face->interp[f1*NGauss+j][i] *
```

```
                face->W[f1*NGauss+j][k1];
          }
        }
      }
*/
      if(type){
        for(int i=0; i<Np; i++){
          for(int j=0; j<NGauss; j++){
indices[j] = k1*Nfaces*NGauss + f1*NGauss + j;
values[j] = BOP11[i][j];
          }
          PRsysBC->SumIntoMyValues(k1*Np+i, NGauss, values, indices);
        }
      }
    }
  }
  for(int i=0; i<NGauss; i++)
    delete[] DnM[i];
  for(int i=0; i<Np; i++)
    delete[] BOP11[i];
  delete[] BOP11; delete[] DnM;
  delete[] Xloc; delete[] Yloc;
  delete[] indices; delete[] values;
}
void INSSolver::PressureIPDG2D(TriMesh *mesh, DGLocalOps *localOps,
    DGGlobalOps *globalOps, Cubature *cub, FaceMesh *face)
{
  int k2, f2, type, revk;
int *indices, NumEntries;
  double *values, *MMvalues;
  double **OP11, **OP12, hinv;
  double gtau;
  double *Xloc, *Yloc;
  double *Xloc2, *Yloc2;
double **gDnM, **gDnP;
  indices = new int [Np];
  values = new double [Np];
  MMvalues = new double [Np];
  OP11 = new double *[Np];
```

```
  OP12 = new double *[Np];
  Xloc = new double [Np];
  Yloc = new double [Np];
  Xloc2 = new double [Np];
  Yloc2 = new double [Np];
  gDnM = new double *[Ncub];
  gDnP = new double *[Ncub];
  for(int i=0; i<Np; i++){
    OP11[i] = new double [Ncub];
    OP12[i] = new double [Ncub];
  }
  for(int i=0; i<Ncub; i++){
    gDnM[i] = new double [Np];
    gDnP[i] = new double [Np];
  }
  for(int k1=0; k1<NumElem; k1++){
    for(int i=0; i<Np; i++){
Xloc[i] = globalOps->x[i][k1];
      Yloc[i] = globalOps->y[i][k1];
    }
    PhysDmatrices(Xloc, Yloc, localOps, cub);
    for(int i=0; i<Np; i++){
      for(int j=0; j<Np; j++){
        OP11[i][j] = 0.0;;
        for(int k=0; k<Ncub; k++){
          OP11[i][j] += cDx[k][i] * cub->W[k][k1] * cDx[k][j];
          OP11[i][j] += cDy[k][i] * cub->W[k][k1] * cDy[k][j];
        }
      }
    }
    for(int f1=0; f1<Nfaces; f1++){
type = mesh->BCType[f1][k1];
      k2 = globalOps->EToE[f1][k1];
      f2 = globalOps->EToF[f1][k1];
      for(int i=0; i<Np; i++){
Xloc2[i] = globalOps->x[i][k2];
        Yloc2[i] = globalOps->y[i][k2];
      }
```

```
      PhysDmatrices(Xloc, Yloc, localOps, face, f1);
      PhysDmatrices(Xloc2, Yloc2, localOps, face, f2, 2);

      for(int i=0; i<NGauss; i++){
        for(int j=0; j<Np; j++){
          gDnM[i][j] = face->nx[f1*NGauss + i][k1] * dVdxM[i][j];
          gDnM[i][j] += face->ny[f1*NGauss + i][k1] * dVdyM[i][j];
          gDnP[i][j] = face->nx[f1*NGauss + i][k1] * dVdxP[i][j];
          gDnP[i][j] += face->ny[f1*NGauss + i][k1] * dVdyP[i][j];
        }
      }
      hinv = globalOps->Fscale[f1*Nfp][k1];
      if(globalOps->Fscale[f2*Nfp][k2] > hinv)
        hinv = globalOps->Fscale[f2*Nfp][k2];
      gtau = 20 * (N+1) * (N+1) * hinv;
      switch(type){
        case OUTBC:
        case INBC:
        case WALLBC:
        case CYLBC:
// Dirichlet BC's
          for(int i=0; i<Np; i++){
            for(int j=0; j<Np; j++){
              for(int k=0; k<NGauss; k++){
                OP11[i][j] += face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gtau * face-
>interp[f1*NGauss+k][j];
                OP11[i][j] -= face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gDnM[k][j];
                OP11[i][j] -= gDnM[k][i] * face->W[f1*NGauss+k][k1] *
                 face->interp[f1*NGauss+k][j];
              }
            }
          }
        break;
// Neuman BC's
/*
        case INBC:
        case WALLBC:
```

```
        case CYLBC:
// pass
        break;
*/
        default:
          for(int i=0; i<Np; i++){
            for(int j=0; j<Np; j++){
              OP12[i][j] = 0.0;
              for(int k=0; k<NGauss; k++){
                revk = NGauss - 1 - k;
                OP11[i][j] += 0.5 * face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gtau * face-
>interp[f1*NGauss+k][j];
                OP11[i][j] -= 0.5 * face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gDnM[k][j];
                OP11[i][j] -= 0.5 * gDnM[k][i] * face-
>W[f1*NGauss+k][k1] *
                 face->interp[f1*NGauss+k][j];

                OP12[i][j] -= 0.5 * face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gtau *
                 face->interp[f2*NGauss+revk][j];
                OP12[i][j] -= 0.5 * face->interp[f1*NGauss+k][i] *
                 face->W[f1*NGauss+k][k1] * gDnP[k][j];
                OP12[i][j] += 0.5 * gDnM[k][i] * face-
>W[f1*NGauss+k][k1] *
                 face->interp[f2*NGauss+revk][j];
              }
              indices[j] = k2*Np+j;
              values[j] = OP12[i][j];
//              PRsys[k1*Np+i][k2*Np+j] += OP12[i][j];
            }
            PRsys->SumIntoMyValues(k1*Np+i, Np, values, indices);
          }
      }
    }
    for(int i=0; i<Np; i++){
      for(int j=0; j<Np; j++){
indices[j] = k1*Np+j;
```

```
values[j] = OP11[i][j];
        MMvalues[j] = cub->mm[i][j][k1];
//        PRsys[k1*Np+i][k1*Np+j] += OP11[i][j];
//        MM[k1*Np+i][k1*Np+j] += cub->mm[i][j][k1];
      }
      PRsys->SumIntoMyValues(k1*Np+i, Np, values, indices);
      MMsys->SumIntoMyValues(k1*Np+i, Np, MMvalues, indices);
    }
  }
  for(int i=0; i<Np; i++){
    delete[] OP11[i]; delete[] OP12[i];
  }
  for(int i=0; i<Ncub; i++){
    delete[] gDnM[i]; delete[] gDnP[i];
  }
  delete[] indices; delete[] values; delete[] MMvalues;
  delete[] gDnM; delete[] gDnP;
  delete[] OP11; delete[] OP12;
  delete[] Xloc; delete[] Yloc;
  delete[] Xloc2; delete[] Yloc2;
}
```