The nodal discontinuous Galerkin method

Steven Dorsher

December 11, 2014

1 Introduction

The disccontinous Galerkin method is a method for solving partial differential equations (PDEs) in time and space. It has two forms, the modal form and the nodal form. In both forms, space is modelled by numerous discrete elements, not necessarily of a fixed size or shape, that form a grid to fill the space of the simulation. Within each element, the simulated field is modelled at discrete set of points determined by the element's order. This modelling is done by a superposition of polynomials whose properties depend upon the form chosen (modal or nodal). At the boundaries of elements, the field is not necessarily continuous. Numerical fluxes across the surfaces of elements account for these discontinuities while minimizing residuals. The discontinuous Galerkin method takes advantage of properties of these polynomials to form a derivative matrix that can be used to solve the separated linear equations.

2 Elements and nodes

In the discontinuous Galerkin method, space is divided into M elements that each has an order N_k. Within the kth element, there are $N_p = N_k + 1$ nodes where the the solution will be computed. When optimally chosen, these are not uniformly placed throughout the element. Although the solution is only computed at a discrete number of points, the solution at these points can be used to model the solution at points in between. In the modal representation, the solution is built out of an orthonormal basis of polynomials with coefficients determined by the values at the nodes. In the nodal representation, the solution is built out of a non-orthonormal basis of polynomials that have the property that each polynomial has the value of one at exactly one node and is zero at all other nodes. Between them, the N_p polynomials have values of one at all N_p nodes. The solution to the PDE is given by the following formula

$$u_h^k(x,t) = \sum_{n=1}^{N_p} u_h^k(x_i,t)\ell_i^k(x) \tag{1}$$

where $\ell_i^k(x)$ is the interpolating Lagrange polynomial

$$\ell_i(x) = \prod_{j=1, j\neq i}^{N_p} \frac{x - x^j}{x^i - x^j} \tag{2}$$

and $\ell_i(x_j) = \delta_{ij}$ for node points x^i in the kth element.

3 Definition of the Galerkin method

In a generalized definition of the Galerkin method, the basis polynomials need not take on the form of Equation 2. Any polynomial, even a piecewise defined polynomial, that fits the requirements of Section 2 would be sufficient. With that in mind, consider a generalized form of the discretized PDE solution.

$$u_h(x) = \sum_{k=1}^{K} u(x^k) N^k(x) \tag{3}$$

where $N^i(x^j) = \delta_{ij}$ is the basis function.

Consider a general scalar first order PDE

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = g. \tag{4}$$

Let the discrete approximation to the solution be $u_h(x)$. The residuals represent the degree to which the numerical solution and the exact solution differ. In this example, the residuals are given by

$$R_h(x,t) = \frac{\partial u_h}{\partial t} + \frac{\partial f_h}{\partial x} - g(x,t). \tag{5}$$

This is obtained by taking u_h and inserting it into the PDE. To obtain a method to solve the PDE, we define a test function ϕ_h defined and require residual must be orthogonal to all such test functions.

$$\int R_h(x,t)\phi_h(x)dx = 0 \tag{6}$$

The discontinuous Galerkin method requires that the space spanned by the test functions $\phi_h(x)$ be the same as the space spanned by the basis functions $N^k(x)$.

4 Weak form and strong form

We would like to derive how to solve PDEs using the discontinuous Galerkin method. Consider the PDE shown in Equation 4. Beginning with the definition of the discontinuous Galerkin method,

$$\int R_h(x,t)\ell_j^k(x)dx = 0 \tag{7}$$

Gauss's theorem can be used to obtain

$$\int \frac{\partial u_h^k}{\partial t} - f_h^k \frac{d\ell_j^k}{dx} - g\ell_j^k dx = -[f_h^k \ell_j^k]_{x^k}^{x^{k+1}} \tag{8}$$

To account for the interactions between elements at the boundaries, the numerical flux f^* is introduced. The numerical flux takes into account information from both neighboring elements in a manner that depends upon the chosen definition for the problem at hand. Inserting f^* into the above equation in place of f_h^k, the weak form is obtained. Using Gauss's theorem again, the strong form follows:

$$\int R_h(x)\ell_j^k(x)dx = [(f_h^k - f^*)\ell_j^k]_{x^k}^{x^{k+1}} \tag{9}$$

5 The mass and stiffness matrices

Consider a second example, that of the advection equation.

$$\frac{\partial u}{\partial t} + \frac{\partial au}{\partial x} = 0 \tag{10}$$

where a is a constant. The residuals take the form

$$R_h(x) = \frac{\partial u_h}{\partial t} + \frac{\partial au_h}{\partial x} \tag{11}$$

By inserting that residual into the strong form from Equation 9, we obtain

$$\int R_h(x)\ell_j^k(x)dx \tag{12}$$

$$= \int \left(\tfrac{\partial u_h}{\partial t} + \tfrac{\partial au_h}{\partial x}\right)\ell_j^k(x)dx \tag{13}$$

$$= [((au)_h^k - f^*)\ell_j^k]_{x^k}^{x^{k+1}} \tag{14}$$

Inserting the definition of the nodal solution for u_h, Equation 1, we obtain

$$\int R_h(x)\ell_j^k(x)dx \tag{15}$$

$$= \int \left[\left(\frac{\partial}{\partial t} \sum_{i=1}^{N_p} u_h^k(x_i^k, \ell_i^k(x)) \right) + \frac{\partial}{\partial x} \left(\sum_{i=1}^{N_p} f_h^k(x_i, t)\ell_i^k(x) \right) \right] \ell_j^k dx \tag{16}$$

$$= [((au)_h^k - f^*)\ell_j^k]_{x^k}^{x^{k+1}} \tag{17}$$

This simplifies to a matrix solution to the PDE of the form

$$M^k \frac{d\boldsymbol{u}_h^k}{dt} + aS^k \boldsymbol{u}_h^k = (f_h^k(x^{k+1}) - f^*(x^{k+1}))\ell^k(x^{k+1}) - (f_h^k(x^k) - f^*(x^k))\ell^k(x^k) \tag{18}$$

where the vectors have one entry per interface per element, such that the vector is $2N_p$ long. The mass matrix is defined to be

$$M_{ij}^k = \int \ell_i^k(x)\ell_j^k(x)dx \tag{19}$$

and the stiffness matrix is defined to be

$$S_{ij}^k = \int \ell_i^k(x)\frac{d\ell_j^k}{dx}dx \tag{20}$$

For most PDEs, these are unrelated to the normal physics concepts of mass an stiffness.

6 Numerical fluxes

In general, numerical fluxes depend on both the value of the solution to the PDE at the boundary and the flux across the boundary, as naively defined. For that purpose, two operators are defined.

$$\{\{u\}\} = \frac{u^- + u^+}{2} \tag{21}$$

is the average of the solution to the PDE in the two elements adjacent to the interface. The jump between adjacent elements is given by

$$[[\boldsymbol{u}]] = \boldsymbol{n}^- u^- + \boldsymbol{n}^+ u^+ \tag{22}$$

$$\tag{23}$$

Consider the advection equation again, given by Equation 10. A consistent, numerically convergent definition for the numerical flux is given by

$$f^* = (au)* = \{\{au\}\} + |a|\frac{1-\alpha}{2}[[u]] \tag{24}$$

7 Reference elements

In one dimension, each element can be scaled to a reference element dependent only upon its width and vertex position. The absolute position of a node is related to the position of that node within the reference element by the mapping

$$x(r) = x_l^k = \frac{1+r}{2}h^k \tag{25}$$

where $h^k = x_r^k - x_l^k$ is the difference between the left and right cell boundaries of the kth cell. To get the physical derivitave it is necessary to scale by the Jacobian $\partial r/\partial x$.

8 Selection of the nodes

For a well-conditioned mass matrix, the optimal basis of the PDE solution is a basis of orthonormal polynomials, $\widetilde{P}_j(r)$. Put in interpolatory nodal form, the discretized solution to the PDE is

$$u(r) \approx u_h(r) = \sum_{n=1}^{N_p} \hat{u}_n \widetilde{P}_{n-1}(r) \tag{26}$$

Here, $\widetilde{P}_n(r) = \frac{P_n(r)}{\gamma_n}$ where $\gamma_n = \frac{2}{2n+1}$ and $P_n(r)$ is the classic Legendre polynomial. The Vandermonde matrix V_{ij} is defined as

$$V\hat{\boldsymbol{u}} = \boldsymbol{u}V_{ij} = \widetilde{P}_{j-1}(\zeta_i) \tag{27}$$

$$\hat{\boldsymbol{u}}_i = u_i \tag{28}$$

$$\boldsymbol{u}_i = u(\zeta_i) \tag{29}$$

where ζ_i is the ith node.

Another interpolatory form is based on Equation 1 and uses the interpolating Lagrange polynomials.

$$u(r) \approx u_h(r) = \sum_{i=1}^{N_p} u(\zeta_i)\ell_i(r) \tag{30}$$

Comparing Equations 26 and 30 and using the definition of V from Equation 29,

$$V^T \boldsymbol{\ell}(r) = \widetilde{\boldsymbol{P}}(r) \tag{31}$$

$$\boldsymbol{\ell} = [\ell_1(r), ..., \ell_{N_p}(r)]^T \tag{32}$$

$$\widetilde{\boldsymbol{P}}(r) = [\widetilde{P}_0(r), ..., \widetilde{P}_{N_P}(r)]^T \tag{33}$$

The best conditioned matrix results when the N_p nodes occur at the Legendre-Gauss-Lobotto quadrature points given by the zeros of the following equation.

$$f(r) = (1 - r^2)\widetilde{P}'_{N_p}(r) \tag{34}$$

9 The derivative matrix

Operationally, the interpolating Lagrange polynomial is computed using the inverse of the transpose of the Vandermonde matrix on the Jacobi polynomial.

$$\ell_i(r) = \sum n = 1^{N_p} (V^T)_i^{-1} n \widetilde{P}_{n-1}(r) \tag{35}$$

The mass matrix is defined by Equation 19. In reference element form, it is given by $M_{ij} = \frac{h^k}{2}(\ell_i, \ell_j)$ where the factor in front is due to the deforming of the elements to a range of -1 to 1 rather than a dimension of h for each element. S_{ij} stays the same, since it is both contains a derivative and an integral along r. Plugging Equation 35 into these definitions for M and S, the following methods for calculating M and S are obtained.

$$M^k = \frac{h^k}{2} M = \frac{h^k}{2}(VV^T)^{-1} \tag{36}$$

$$S^k_{ij} = \int_{-1}^{1} \ell_i(r) \frac{d\ell_j(r)}{dr} dr = S_{ij} \tag{37}$$

The derivative matrix is defined as

$$D_{r,(i,j)} = \left. \frac{d\ell_j}{dr} \right|_{r_i} \tag{38}$$

It can be shown that $MD_r = S$. From this and Equation 29 we can see that the derivative matrix and the derivative of the Jacobi polynomials are linked by the Vandermonde matrix.

$$V^T \frac{d}{dr}\boldsymbol{\ell}(r) = \frac{d}{dr}\widetilde{\boldsymbol{P}}(r) \tag{39}$$

which can be rewritten in an operational definition of how to obtain the derivative matrix, when combined with some rules about the Jacobi polynomials in question.

$$D_r = (V_r)V^{-1} \tag{40}$$

$$V_{r,(i,j)} = \left.\frac{d\widetilde{P}_j}{dr}\right|_{r_i} \tag{41}$$

$$\frac{d\widetilde{P}_n}{dr} = \sqrt{n(n+1)}\widetilde{P}^{(1,1)}_{n-1}(r) \tag{42}$$

This derivative matrix is used to replace the spatial part of the derivative operator in the discretized form of the separated first order system of PDEs.

10 Rewriting the system of PDEs in first order separated form

Many PDEs of interest are second order in both time and space. To make them tractable numerically, it is necessary to separate them into a system of coupled equations that are first order in time. To do this, a new variables are assigned to the first derivative with respect to time and with respect to space. Those variable is inserted into the original equation. A system of coupled equations result.

Consider the scalar wave equation in one dimension.

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} \tag{43}$$

Substituation of $v = \partial u/\partial t$ and $w = \partial u/\partial x$ yields

$$\frac{\partial u}{\partial t} = v \tag{44}$$

$$\frac{\partial v}{\partial t} = c^2\frac{\partial w}{\partial x} \tag{45}$$

$$\frac{\partial w}{\partial t} = \frac{\partial v}{\partial x} \tag{46}$$

This can be rewritten in matrix form.

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & c^2 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

At this point, the right hand sides of the equations can be evaluated using the derivative matrix defined in the previous section and the left hand sides can be solved with a numerical integrator. Additional terms will be present due to the boundary conditions at the ends of the elements, where the numerical fluxes must be taken into account.

For a general linear equation, the time evolution can be represented in matrix form.

$$\frac{\partial \boldsymbol{u}}{\partial t} = A \frac{\partial \boldsymbol{u}}{\partial x} + B\boldsymbol{u} \tag{47}$$

10.1 Numerical fluxes of linear hyperbolic systems

For systems linar in their spatial derivatives and hyperbolic, it is possible to construct a transformation T that diagonalizes A that has real eigenvalues.

$$A = T\Lambda T^{-1} \tag{48}$$

Separating the diagonal matrix into two halves by its positive and negative eigenvalues, it can be written

$$\Lambda = \Lambda^{+} + \Lambda^{-} \tag{49}$$

Because components with positive eigenvalues represent components propogating in the positive direction and components with negative eigenvalues represent propogation in the negative direction, the following numerical flux can be constructed for linear hyperbolic equations.

$$A\Lambda^{+}T^{-1}\boldsymbol{u}^{-} + \Lambda^{-}T^{-1}\boldsymbol{u}^{+} \tag{50}$$

10.2 Accounting for boundary conditions

To account for boundary conditions, the numerical flux must be taken into account in the right hand side of the system of differential equations. To obtain the form of the differential equation appropriate for use in numerical calculations that makes use of the derivative matrix, the strong form of the discontinuous Galerkin method must be modified by multiplying by M^{-1}. In the case of the advection equation, this results in the following form.

$$\frac{d\boldsymbol{u}_h^k}{dt} = -aD_r^k\boldsymbol{u}_h^k + (M^{-1})^k[(f_h^k(x^{k+1}) - f^*(x^{k+1}))\ell^k(x^{k+1}) - (f_h^k(x^k) - f^*(x^k))\ell^k(x^k)] \tag{51}$$

The value of the numerical flux $f*$ depends on the values of the solution to the PDE in adjacent elements and applies at the boundaries in Equation 9. However, multiplication by M^{-1} causes the numerical flux to effect the value the differential equation at every node within a given element, not just the end points.

11 Stepping forward in time

Once the right hand side of Equation 47 has been calculated, the time evolution is a matter of numerically integrating the system of first order (in time) PDEs. The classic approach to numerical integration would be the fourth order Runga Kutta (RK) method, where the integral occurs in four stages. At each stage, the right hand side is calculated at either the midpoint or one of the two ends of the time interval. At the fourth stage, the integral is calculated using the trapezoid rule taking a weighted average of the slopes obtained in each of the four previous steps for the line defining the top of the trapezoid. Explicitly, if

$$\frac{d\boldsymbol{u}_h}{dt} = F_h(\boldsymbol{u}_h, t) \tag{52}$$

then the classic fourth order Runga Kutta method is given by

$$\boldsymbol{k}^{(1)} = F_h(\boldsymbol{u}_h, t) \tag{53}$$

$$\boldsymbol{k}^{(2)} = F_h(\boldsymbol{u}_h + \frac{1}{2}\Delta t\boldsymbol{k}^{(1)}, t + \frac{1}{2}\Delta t) \tag{54}$$

$$\boldsymbol{k}^{(3)} = F_h(\boldsymbol{u}_h + \frac{1}{2}\Delta t\boldsymbol{k}^{(2)}, t + \frac{1}{2}\Delta t) \tag{55}$$

$$\boldsymbol{k}^{(4)} = F_h(\boldsymbol{u}_h + \Delta t\boldsymbol{k}^{(3)}, t + \Delta t) \tag{56}$$

$$\boldsymbol{u}_h^{n+1} = \boldsymbol{u}_h^n + \frac{1}{6}\Delta t(\boldsymbol{k}^{(1)} + 2\boldsymbol{k}^{(2)} + 2\boldsymbol{k}^{(3)} + \boldsymbol{k}^{(4)}) \tag{57}$$

However, this algorithm requires four intermediate stages to be stored in memory for each position at which $\boldsymbol{u}_h$ is evaluated. To make better use of memory there is a low storage explicit Runga Kutta (LSERK) technique that does not require storage of its intermediate steps as it evaluates its intermediate steps. Since it takes five stages rather than four, it takes more processor cycles to compute. It is given by

i	a_i	b_i	c_i
1	0	$\frac{1432997174477}{9575080441755}$	0
2	$-\frac{567301805773}{1357537059087}$	$\frac{5161836677717}{13612068292357}$	$\frac{1432997174477}{9575080441755}$
3	$-\frac{2404267990393}{2016746695238}$	$\frac{1720146321549}{2090206949498}$	$\frac{2526269341429}{6820363962896}$
4	$-\frac{3550918686646}{2091501179385}$	$\frac{3134564353537}{4481467310338}$	$\frac{2006345519317}{3224310063776}$
5	$-\frac{1275806237668}{842570457699}$	$\frac{2277821191437}{14882151754819}$	$\frac{2802321613138}{2924317926251}$

Table 1: Coefficients for the LSERK numerical integration technique.

$$\boldsymbol{k}^{(i)} = a_i \boldsymbol{k}^{(i-1)} + \Delta t F_h(\boldsymbol{p}^{(i-1)}, t^n + c_i \Delta t) \tag{58}$$

$$\boldsymbol{p}^{(i)} = \boldsymbol{p}^{(i-1)} + b_i \boldsymbol{k}^{(i)} \tag{59}$$

where $\boldsymbol{p}^{(0)}$ is initialized to $\boldsymbol{p}^{(0)} = \boldsymbol{u}^n$ and the updated value of $\boldsymbol{u}$ is obtained on the fifth step by setting $\boldsymbol{u}^{n+1} = \boldsymbol{p}^{(5)}$. The coefficients a_i, b_i, and c_i are given in Table 11. In this manner, only two values need to be stored at any given sub-step.

12 Stability

A numerical algorithm is numerically stable if it produces output that is bounded in its growth. Consider a hyperbolic system in one dimension,

$$\frac{\partial u}{\partial t} = A \frac{\partial u}{\partial x} \tag{60}$$

An algorithm will be numerically stable if

$$\lim_{dof \to \infty} || \exp(-A D_x(t)) ||_{\Omega,h} \leq C_h \exp(\alpha_h t) \tag{61}$$

for constants C_h and α_h and for the derivative matrix D_x as defined in Section **??**, with the degrees of freedom (dof) taken to infinity by increasing the order N or decreasing the cell size h. The Sobelev norm is defined as

$$||u||^2_{\Omega,q} = \sum_{|a|=0}^{q} ||u^(a)||^2_{\Omega} \tag{62}$$

where the L^2 norm is defined to be

$$||u||^2_{\Omega} = \int_{\Omega} u^2 dx \tag{63}$$

Now consider a hyperbolic system of equations.

$$\frac{\partial \boldsymbol{u}}{\partial t} = A\frac{\partial \boldsymbol{u}}{\partial x} \tag{64}$$

By definition, the A matrix will be diagonalizable, $A = T\Lambda T^{-1}$. The resulting stability condition is

$$||\boldsymbol{u}_h|| = ||T||^2_{\Omega,h}||T^{-1}||^2_{\Omega,h}\exp(\alpha_h t)||\boldsymbol{u}_h||^2_{\Omega,h} \tag{65}$$

which results in stability as long as the product of the two terms involving T is less than a constant.

13 Error

The error for the discontinuous Galerkin method can be shown to be bounded by

$$||\epsilon_h||_{\Omega,h} \leq C(N)h^{N+1}(1 + C_1(N)T) \tag{66}$$

This demonstrates that decreasing h results in a power law scaling of the error while increasing N results in an exponential scaling of the error (assuming h is less than one). The error also increases over time in a manner dependent upon the order but not the cell size.

14 Software design

We have designed an object oriented framework in C++ for solving PDEs using the nodal discontinuous Galerking method. In this design, there are Grid, ReferenceElement, DifferentialEquation, TimeEvolution, GridFunction, and VectorGridFunction objects.

14.1 ReferenceElement

The ReferenceElement handles the calculations that are independent of a particular physical scaling. It requires that the order of the element be set at initialization. Most interestingly, it calculates the spatial derivative matrix with respect to the reference element length variable (D_r). It also obtains the positions of the nodes within the element, again scaled to r. Currently we plan to handle the linear algebra with the TNT linear algebra package.

```
#include "tnt_array1d.h"

#include "tnt_array2d."
```

```
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"

class ReferenceElement
{
 private:
  int order; //order of element
  Array1D refNodeLocations; // node locations scaled to r
  Array2D vandermondeMatrix;
  Array2D derivativeMatrix;
  Array2D dVdr;
 public:
  void initialize(int N); //static initializer outside of constructor
 private:
  void jacobiPolynomails();
  void jglQuadraturePoints();
  void computeVandermonde();
  void computedVdr();
  void computeDr();
 public:
  Array2D getDr(); //get derivative matrix
};
```

14.2 Grid

The Grid holds the location of the nodes along the spatial axis of the simulation in a member variable of type GridFunction. This is created at construction. Element boundaries are read from a file. Those are used to scale the reference element nodes to their physical coordinates. The result is stored in a GridFunction.

```
#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
```

```
#include "tnt_array2d_utils.h"
#include "ReferenceElement.h"
#include "GridFunction.h"

class Grid
{

 private:
  string fileElemBoundaries;
  int NumElem;
  Array1D elementBoundaries; //2N
  GridFunction nodesLocations; //NxNp

 public:
  Grid(string fileElemBoundaries, ReferenceElement refelem);
  //initializes elementBoundaries from file,
  //obtains node locations from reference element and puts them in array

  int getN();//return number of elements, calculated from input file
  GridFunction getNodeLocations();
};
```

14.3 GridFunction and VectorGridFunction

A GridFunction consists of a vector of Array1D's. Each Array1D stores information about a specific element of the grid. The Array1D has a length equal to the number of nodes and the vector has a lenght equal to the number of elements. VectorGridFunctions are a vector of GridFunctions, used to store the values of solutions to PDEs ($\boldsymbol{u}_h$) and the right hand side of the partial differential equation. These may have vector form in a system of PDEs.

GridFunctions and VectorGridFunctions require getters and setters to access a specific value along a specific dimension. They also require getters to obtain information about the dimensions. There must be a way to initialize them, either to zero or from values specified in a file.

It is necessary to define a copy assignment operator for the purposes of doing arithmetic GridFunctions and VectorGridFunctions. Move constructors and operators are important to optimizing performance. For example, when adding the right hand side of the PDE and the boundary conditions together in the rk4 routine, we would like to move the result to the intermediate storage rather than copying it, replacing what was there. The addition operator also has been defined to make addition of the differential equation right hand side and the boundary conditions possible, as well as to enable addition between different intermediate time steps in the LSERK rk4 algorithm.

There is a lot of apparant repetition between the VectorGridFunction class and the GridFunction class, yet VectorGridFunction cannot inherit from GridFunction because it would make a type conversion from VectorGridFunction to GridFunction possible, and they have data representations that are of fundamentally different types. While it is true that a scalar is a special case of a vector, we find it easier to define a scalar (GridFunction) and build our vector (VectorGridFunction) out of that than to define a GridFunction as a VectorGridFunction with an external vector dimension of one.

```
#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"

class GridFunction
{
 private:
  vector<Array1D<double>> data;
  int GFvectorDim;
  int GFarrayDim;
 public:
  GridFunction(int vecSize, int arraySize,bool initZeros);
  void initFromFile(string filename);
  double get(int vcoord, int acoord);
  double set(int vcoord, int acoord);
  int getGFvecDim();
  int getGFarrDim();
  GridFunction(GridFunction&&);
  GridFunction& operator=(GridFunction&&);
  //need move constructors for addition of RHS and boundary conditions
  GridFunction(const GridFunction&);
  GridFunction& operator=(const GridFunction&);
  //need copy constructors for intermediate steps of time evolution
  void operator+(GridFunction, GridFunction);
  //need addition operator for addition of RHS and boundary conditions
  void print(string filename)
};

#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"

class VectorGridFunction
{
 private:
  vector<GridFunction> data;
  int VGFvectorDim;
  int GFarrayDim;
  int GFvectorDim;

 public:
  GridFunction(int VGFvecSize, int GFvecSize, int GFarraySize, bool initZero);
  void initFromFile(string filename);
  double get(int VGFvcoord, int GFvcoord, int GFacoord);
  void set(int VGFvcoord, int GFvcoord, int GFacoord);
  int getVGFvecDim();//the dimension of the external vector
  int getGFvecDim();//the dimension of the vector within the GridFunction
  int getGFarrayDim();//the dimension of the array within the GridFunction
  VectorGridFunction(VectorGridFunction&&);
  VectorGridFunction& operator=(VectorGridFunction&&);
  //need move constructors for addition of RHS and boundary conditions
  VectorGridFunction(const VectorGridFunction&);
  VectorGridFunction& operator=(const VectorGridFunction&);
  //need copy constructors for intermediate steps of time evolution
  void operator+(VectorGridFunction, VectorGridFunction);
  //need addition operator for addition of RHS and boundary conditions
  void print(vector<string> filenames); //vector of filenames to print to
};
```

14.4 DifferentialEquation

The DifferentialEquation class encapsulates the information about the right hand side of the differential equation and the boundary conditions, including the numerical flux, that go into it. This class stores no data, but rather exists to separate out some code that is rather specific to the specific physics of the problem.

```
#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"

class DifferentialEquation
{
 public:
  void rightHandSide(GridFunction& nodes, VectorGridFunction& uh,
     VectorGridFunction& RHS);
  // depends on position, solution to PDE, and returns RHS
  void boundaryConditions(GridFunction& nodes, VectorGridFunction& uh,
  VectorGridFunction& RHS)
  //depends on position and solution to PDE,
    //returns sum of RHS and lift and external boundary conditions
};
```

14.5 TimeEvolution.h

The TimeEvolution class stores coefficients for the LSERK rk4 algorithm, or any other algorithm that one should wish to use. It also provides intermediate storage. In the case of the LSERK algorithm, that is a single VectorGridFunction. Although this part of the code has not been implemented yet, there is a standard implementation of the rk4 routine. It will need to call the TimeEvolution::RHS() and TimeEvolution::boundaryConditions() routines at each intermediate step. It will need to update the solution to the PDE at each full step. Calling rk4lowStorage once advances the time evolution by one full time step.

```
#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"
#include "GridFunction.h"
#include "VectorGridFunction.h"
#include "DifferentialEquation.h"

class TimeEvolution
{
 private:
   Array2D coeffs;
   VectorGridFunction intermediateStep;
   DifferentialEquation de;
 public:
  void rk4lowStorage(GridFunction& nodes, VectorGridFunction& uh,
                     VectorGridFunction& RHS);
  //calls de.RHS and de.boundaryConditions for each intermediate time step
};
```

14.6 The main program

The following is a short example of how the main program would look. The ReferenceElement must be initialized first, then the Grid. The PDE solution (uh) and right hand side (RHS) variables can be declared and initialized at any time. Finally, there is a loop over time advancing the rk4 routine one full time step at a time. Output occurs when some output condition is met, to files specified by the user.

```
#include "tnt_array1d.h"
#include "tnt_array2d."
#include "tnt_array1d_utils.h"
#include "tnt_array2d_utils.h"
#include "DifferentialEquation.h"
#include "TimeEvolution.h"
#include "GridFunction.h"
#include "VectorGridFunction.h"

int main()
{
  int timesteps=10000;
  int PDEnum = 1; //number of independent PDEs.
  int Np=21; //order of element

  //initialization of reference element
  ReferenceElement refelem();
  refelem.initialize(Np);

  //initialization of grid
  Grid thegrid(fileElemBoundaries, refelem);
  int NumElem=theGrid.getN();

  //declaration of calculation variables and
  //initialization to either zero or value read from file
  VectorGridFunction(PDEnum,N,Np,false) uh;
  //solution to PDE, possibly a vector
  VectorGridFunction(PDEnum,N,Np,true) RHS; //right hand side of PDE
  uh.initFromFile(scalarfilename);

  //evolve in time and print out at select time steps
  TimeEvolution te();

  for(i=0;i<timesteps;i++){
    TimeEvolution::rk4lowstorage(thegrid.getNodeLocations(),uh, RHS);
    if(outputcondition) uh.print(filenames);
  }

}
```

15 Aknowledgments

I'd like to thank Dr. Peter Diener for his much needed guidance on the software design and for the time and effort he has put into explaining the discontinuous Galerkin method to me.