Math 597

Project #4 : Final Project

Damyn Chipman

Introduction

As the final project for this course, we will look at the 2D conservation form of a system of partial differential equations. We'll derive system matrices and update formulas for integrating the solution over time.

Primarilly, we'll use the advection equation with a given velocity field as the first test case for this implementation. We'll look at the current results of the implementation and the current issues and future work.

The code for this project is provided on GitHub at the HydroForest repository.

Problem Statement

Solve the following system of partial differential equations:

$$PDE: \ \frac{\partial q_k(x_d,t)}{\partial t} + \frac{\partial f_{kd}(q_k)}{\partial x_d} = s_k(x_d,t) \tag{1}$$

$$IC: \ q_k(x_d,0) = q_{k,0}(x_d) \tag{2}$$

with periodic boundary conditions, and $k = 0, \ldots, N_{equations} - 1$, and $d = 0, \ldots, N_{dimensions} - 1$.

This system of PDEs is expressed in conservation form. Several systems of PDEs can be expressed in this manner, including:

- The Advection-Diffusion Equation

$$q_k = [q] \tag{3}$$

$$f_{kd}(q_k) = [qu_d - \nu \frac{\partial q}{\partial x_d}] \tag{4}$$

where, q is some scalar quantity that can be advected, u_d is the velocity, ν is the diffusion coefficient.

- The Shallow Water Equations

$$q_k = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad f_{kd}(q_k) = \begin{bmatrix} hu & hv \\ hu^2 + \frac{1}{2}gh^2 & huv \\ huv & hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad s_k(x_d, t) = \begin{bmatrix} 0 \\ -gh\frac{\partial b}{\partial x} \\ -gh\frac{\partial b}{\partial y} \end{bmatrix}$$

where, h is water height, u, v are velocity components, g is the gravitational constant, and b is a bathymetry term.

- The Euler Equations

$$q_k = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix}, \quad f_{kd}(q_k) = \begin{bmatrix} \rho u & \rho v \\ \frac{u^2}{\rho} + P & \frac{uv}{\rho} \\ \frac{uv}{\rho} & \frac{v^2}{\rho} + P \\ \frac{u(E+P)}{\rho} & \frac{v(E+P)}{\rho} \end{bmatrix}, \quad s_k(x_d, t) = \begin{bmatrix} 0 \\ -\rho g_x \\ -\rho g_y \\ 0 \end{bmatrix} \tag{6}$$

where, ρ is density, u, v are velocity components, $E = \rho e$ is the total energy, P is the pressure, and g_x, g_y is the graviational constant in each direction it acts.

The goal is to derive a solution methodology to solve the general system of PDEs expressed above that can then be used to solve any system that can be cast into the conservation form. Indeed, the motivation for the structure of the following derivation of the DG method is to eventually study hydrodynamics using this code.

The Discontinuous Galerkin (DG) Method

Basis Function Expansion

To start, we begin by using the Galerkin methodology we have developed over the course of the semester. This starts by approximating q_k as an expansion of 2D basis functions:

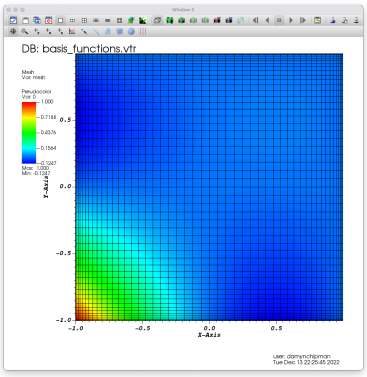
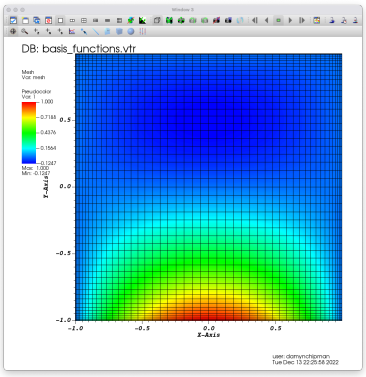
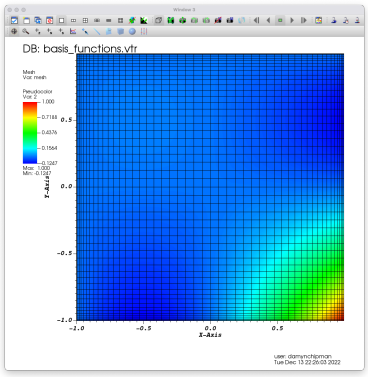
$$q_k(x_d, t) \approx q_{k,N}^{(e)}(x_d, t) = \sum_{i=0}^{M_N - 1} \psi_i(x_d) q_{ik}^{(e)}(t) \tag{7}$$

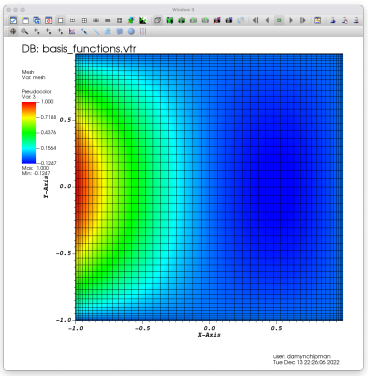
where we use a tensor product basis of 1D basis functions defined as

$$\psi_i(\xi_d) = \psi_i^{(2D)}(\xi_d) = \psi_j^{(1D)}(\xi) \otimes \psi_k^{(1D)}(\eta) \tag{8}$$

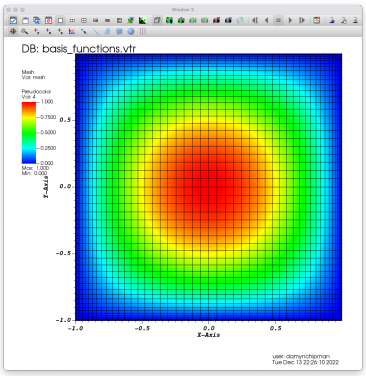
We can reuse the same functionality of our 1D basis functions to work in 2D. As verification and demonstration, below are figures of the 9 basis functions used for $p = 2$ (visualized in 2D and 3D):

Color Plots of 2D Basis Functions

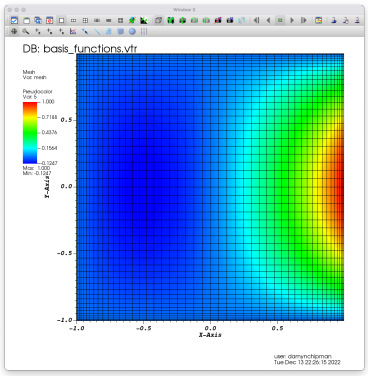
0	1	2
		
3	**4**	**5**



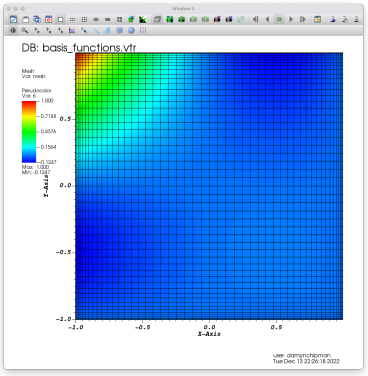
6

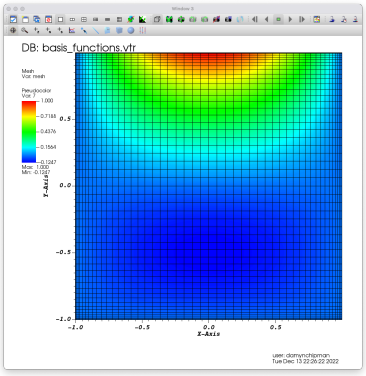


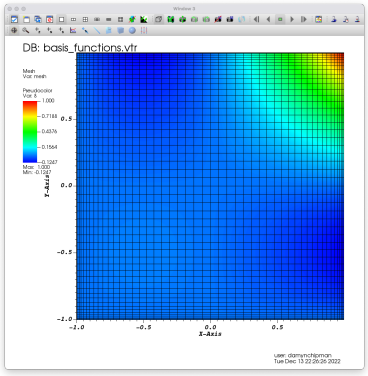
7



8

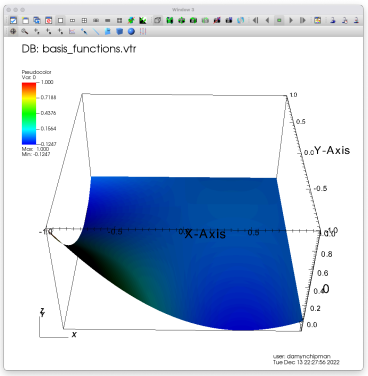




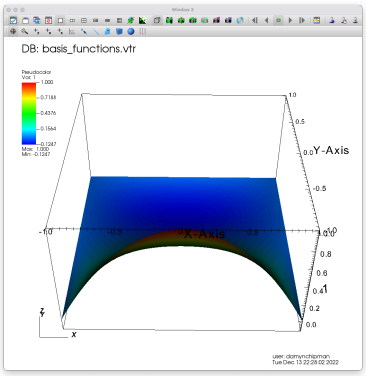


Surface Plots of 2D Basis Functions

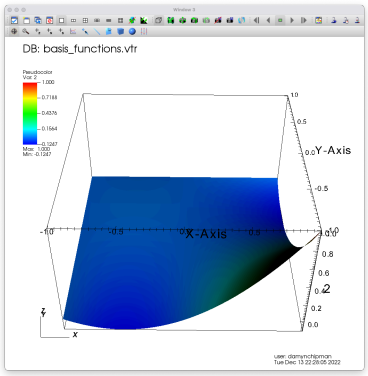
0



1



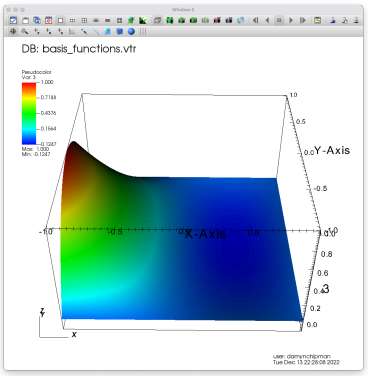
2

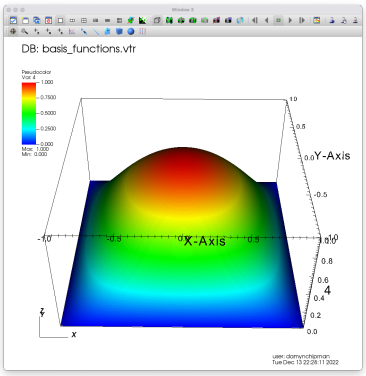


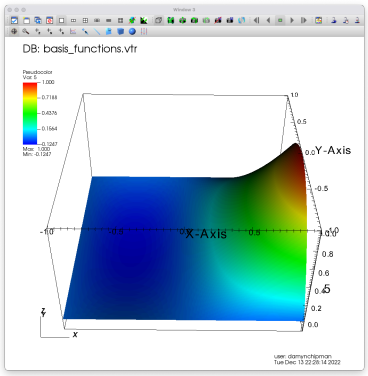
3

4

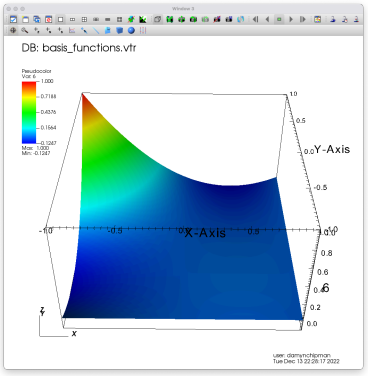
5

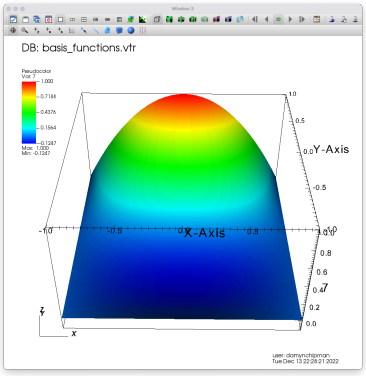

DB: basis_functions.vtr
X-Axis
Y-Axis

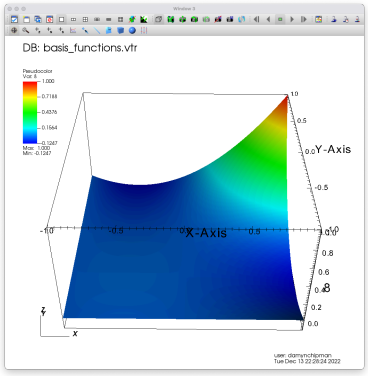

DB: basis_functions.vtr
X-Axis
Y-Axis


DB: basis_functions.vtr
X-Axis
Y-Axis

6 **7** **8**


DB: basis_functions.vtr
X-Axis
Y-Axis


DB: basis_functions.vtr
X-Axis
Y-Axis


DB: basis_functions.vtr
X-Axis
Y-Axis

Galerkin Approach

Now we will use the Galerkin principles to derive a linear system we can use to solve our proposed problem. We start by multiplying the PDE by a test function and integrating over the element:

$$\Rightarrow \int_{\Omega_e} \psi_i \frac{\partial q_{k,N}^{(e)}}{\partial t} d\Omega_e + \int_{\Omega_e} \psi_i \frac{\partial f_{kd,N}^{(e)}}{\partial x_d} d\Omega_e = \int_{\Omega_e} \psi_i s_{k,N}^{(e)} d\Omega_e. \tag{9}$$

We can use the definition of the product rule expressed as

$$\int_{\Omega_e} \frac{\partial}{\partial x_d}[\psi_i f_{kd}] d\Omega_e = \int_{\Omega_e} \frac{\partial \psi_i}{\partial x_d} f_{kd} d\Omega_e + \int_{\Omega_e} \psi_i \frac{\partial f_{kd}}{\partial x_d} d\Omega_e \tag{10}$$

in combination with the Divergence Theorem

$$\int_{\Omega_e} \frac{\partial}{\partial x_d}[\psi_i f_{kd}] d\Omega_e = \int_{\Gamma_e} \psi_i \hat{n}_d f_{kd} d\Gamma_e \tag{11}$$

to rewrite our PDE as the following:

$$\Rightarrow \int_{\Omega_e} \psi_i \frac{\partial q_{k,N}^{(e)}}{\partial t} d\Omega_e + \int_{\Gamma_e} \psi_i \hat{n}_d f_{kd,N}^{(e)} d\Gamma_e - \int_{\Omega_e} \frac{\partial \psi_i}{\partial x_d} f_{kd,N}^{(e)} d\Omega_e = \int_{\Omega_e} \psi_i s_{k,N}^{(e)} d\Omega_e \ ($$

Next we plug in our expansion of q_k to yield:

$$\Rightarrow \sum_{j=0}^{M_N-1} \int_{\Omega_e} \psi_i \psi_j \frac{\partial q_{jk}^{(e)}}{\partial t} d\Omega_e + \sum_{l=0}^{N_F-1} \int_{\Gamma_{e,l}} \psi_i \psi_j \hat{n}_d f_{jkd}^{(*,l)} d\Gamma_e - \sum_{j=0}^{M_N-1} \int_{\Omega_e} \frac{\partial \psi_i}{\partial x_d} \psi_j f_{jkd}^{(e)} d\Omega_$$

Now, we transform to the reference element to allow for flexibility in the element shape:

$$\Rightarrow \sum_{j=0}^{M_N-1} \int_{\hat{\Omega}} \psi_i \psi_j J_i \frac{\partial q_{jk}^{(e)}}{\partial t} d\hat{\Omega} + \sum_{l=0}^{N_F-1} \int_{\hat{\Gamma}_l} \psi_i \psi_j J_i \hat{n}_d f_{jkd}^{(*,l)} d\hat{\Gamma} - \sum_{j=0}^{M_N-1} \int_{\hat{\Omega}} \frac{\partial \psi_i}{\partial x_d} \psi_j J_i f_{jkd}^{(e)} d$$

where J_i is the Jacobian at the nodal point (more on how to compute this later).

This can be expressed in matrix-index form as follows:

$$\Rightarrow M_{ij}^{(e)} \frac{dq_{jk}^{(e)}}{dt} + \sum_{l=0}^{N_F-1} \left(F_{ijk}^{e,l}\right)^T f_{jk}^{(*,l)} - \left(\tilde{D}_{ij}^{(e)}\right)^T f_{jk}^{(e)} = M_{ij}^{(e)} s_{jk} \tag{15}$$

At this point, we have not specified how we will evaluate the integrals to compute the entries in $M_{ij}^{(e)}$, $F_{ijk}^{(e,l)}$, or $\tilde{D}_{ij}^{(e)}$. We will look at the exact vs. inexact integration question when we consider the implementation of this method.

Rearranging, we get:

$$\Rightarrow M_{ij}^{(e)} \frac{dq_{jk}^{(e)}}{dt} = M_{ij}^{(e)} s_{jk} + \left(\tilde{D}_{ij}^{(e)}\right)^T f_{jk}^{(e)} - \sum_{l=0}^{N_F-1} \left(F_{ijk}^{e,l}\right)^T f_{jk}^{(*,l)} \tag{16}$$

For reasons we will observe later, let's define the following:

$$R_{VOLUME,ik}^{(e)} = \left(\tilde{D}_{ij}^{(e)}\right)^T f_{jk}^{(e)} \tag{17}$$

$$R_{FLUX,ik}^{(e,l)} = \left(F_{ijk}^{e,l}\right)^T f_{jk}^{(*,l)} \tag{18}$$

which allows us to write the system as:

$$\Rightarrow M_{ij}^{(e)} \frac{dq_{jk}^{(e)}}{dt} = M_{ij}^{(e)} s_{jk} + R_{VOLUME,ik}^{(e)} - \sum_{l=0}^{N_F-1} R_{FLUX,ik}^{(e,l)} \tag{19}$$

Finally, we multiply by the inverse of the mass matrix to give us the final update formula for the system:

$$\Rightarrow \frac{dq_{ik}^{(e)}}{dt} = s_{ik} + \left(M_{ij}^{(e)}\right)^{-1} \left(R_{VOLUME,jk}^{(e)} - \sum_{l=0}^{N_F-1} R_{FLUX,jk}^{(e,l)}\right) = \hat{R}_{ik}(q_{ik}^{(e)}) \tag{20}$$

This is the form of the system that we will use in our time integration.

Implementation Details

2D Grids, Elements, and Meshes

As this is a 2D code, we need some tools to work on 2D grids.

`Grid2D.hpp`

The 2D tensor product grid is implemented with the `LobattoTensorProductGrid2D` class found in `Grid2D.hpp`. To construct a `LobattoTensorProductGrid2D`, we provide the order basis functions we want in the x- and y-directions. We can access the 1D versions which act as the foundation for the 2D version. In addition, there are utility functions for getting index sets for the points along a boundary or indexing into the grid with a single index or a pair of indices corresponding to an x or a y index.

`Element2D.hpp`

The 2D elements are implemented with the `QuadElement2D` class found in `Element2D.hpp`. Construction requires a set of points with the physical vertices of the element, and the desired basis function order in the x- and y-direction. Upon construction, the element computes and stores the element normals, the area, and Jacobian and metric terms. The Jacobian and metric terms are computed using Algorithm 12.1. The `QuadElement2D` class has utility functions for mapping between the reference and physical grids. As currently implemented, we *should* be able to work on an unstructured grid, though only rectangular elements have been tested. In addition, each element has a `std::map` of `Vector`s and `Matrix`s that can be used to store data on the element such as variable values at nodal points.

`Mesh2D.hpp`

The 2D mesh is implemented in the `ElementMesh2D` class found in `Mesh2D.hpp`. The `ElementMesh2D` class wraps some of the `p4est` functionality and stores the `p4est_t` object that is used with `p4est`. Currently, upon construction, `ElementMesh2D` creates a `p4est_t` object for a periodic, rectangular domain. Each of the `QuadElement2D` instances are stored in each of the `p4est_quadrant_t` objects. This way, we can use the functionality of `p4est` for iterating over volumes and faces, as well as future implementations with adaptivity and unstructured meshes. The `ElementMesh2D` class provides utility functions to iterate over elements, as well as set variables on each element, set solutions on each element, and output the mesh to a VTK PRectilinearGrid format. These files can be visualized with ParaView or VisIt.

Exact vs. Inexact Integration

At the end of chapter 16, Giraldo shows Figure 16.3, which is a work-precision diagram for the exact and inexact integration approaches. This is also something we have experimented with in 1D in the other projects for this course. The time savings for the

minimal difference in error demonstrates that the clear choice for implementation for a quick code is to use inexact integration. Plus, this simplifies many of the algorithms we need to implement as the nodal and quadrature points are the same.

Volume and Flux Integral Contributions

Now let's examine the matrices we need to form (or may not need to form...)

Giraldo details how to form the element matrices $M_{ij}^{(e)}$, $F_{ijk}^{(e,l)}$, or $\tilde{D}_{ij}^{(e)}$. This would allow us to perform the time update by forming the element matrices, multiplying by the associated variable vectors, then inverting the mass matrix to update q_k. This is precisely Algorithm 16.6. However, as Giraldo notes, this approach would only work for linear problems and static grids. Plus, it is terribly inefficient as each element must have not only the data vectors associated with the system of PDEs, but also each of the system matrices. Because of the static grid concern and for better efficiency, we'll try a different approach.

Above, we define R_{VOLUME} and R_{FLUX} that represents the volume and flux integral contributions to the right-hand side. We can develop an algorithm to perform the action of $\tilde{D}_{ij}^{(e)}$ or $F_{ijk}^{(e,l)}$ on the associated vector. This allows us to not store a system matrix and allows us to update each element, even if the mesh is adapted. As our intention is to implement this on an adaptive mesh, this is more optimal.

Algorithms 16.7 and 16.8 detail how to form the volume and flux contributions with exact integration. Algorithms 16.9 and 16.10 detail how to form the volume and flux contributions with inexact integration. We implement Algorithms 16.9 and 16.10 below:

The following is part of the `QuadElement2D` class found in `src/Element2D.hpp`. In each of the implementations, we forego the element loop from Algorithms 16.9 and 16.10 and include them as part of the update later on. These functions compute R_{VOLUME} and R_{FLUX} on an element.

```
Vector<FloatingDataType> computeVolumeIntegralInexact() {

    // Algorithm 16.9
    int N = referenceGrid_.size();
    Vector<FloatingDataType> R(N, 0);
    Vector<FloatingDataType>& flux_x = vecs_["fx"];
    Vector<FloatingDataType>& flux_y = vecs_["fy"];
    Matrix<FloatingDataType> W = referenceGrid_.basisWeights();

    // Compute grad(psi)
```

```
    Matrix<FloatingDataType> dpsi_x, dpsi_y;
    {
        Matrix<FloatingDataType> dh_dxi =
referenceGrid_.xPoly().derivative(referenceGrid_.xPoints());
        Matrix<FloatingDataType> h_eta = referenceGrid_.yPoly()
(referenceGrid_.yPoints());
        Matrix<FloatingDataType> h_xi = referenceGrid_.xPoly()
(referenceGrid_.xPoints());
        Matrix<FloatingDataType> dh_deta =
referenceGrid_.yPoly().derivative(referenceGrid_.yPoints());
        dpsi_y = kroneckerProduct(dh_dxi, h_eta);
        dpsi_x = kroneckerProduct(h_xi, dh_deta);
    }

    for (auto j = 0; j < N; j++) {
        for (auto i = 0; i < N; i++) {
            std::pair<int, int> ID = referenceGrid_.ID(j);
            FloatingDataType w_j = W(ID.first, ID.second);
            FloatingDataType J_j = vecs_["jacobian"][j];

            R[i] += w_j * J_j * (dpsi_x(i,j)*flux_x[j] +
dpsi_y(i,j)*flux_y[j]);
        }
    }

    return R;

}

Vector<FloatingDataType> computeFluxIntegralInexact(int
faceIndexInternal) {

    // Algorithm 16.10
    int N = referenceGrid_.size();
    Vector<FloatingDataType> R(N, 0);

    Vector<int> IS_internal = referenceGrid_.faceIndexSets()
[faceIndexInternal];
    Vector<FloatingDataType>& q_internal = vecs_["q"];
    Vector<FloatingDataType>& flux_x_internal = vecs_["fx"];
    Vector<FloatingDataType>& flux_y_internal = vecs_["fy"];
    Vector<FloatingDataType>& u_x_internal = vecs_["ux"];
    Vector<FloatingDataType>& u_y_internal = vecs_["uy"];
    SpaceVector2D<FloatingDataType>& nhat =
normals_[faceIndexInternal];
    Vector<FloatingDataType> w = (faceIndexInternal % 2) ?
referenceGrid_.yWeights() : referenceGrid_.xWeights();
```

```
    Vector<FloatingDataType>& J = vecs_["jacobian"];

    QuadElement2D<FloatingDataType>& neighbor =
*neighbors_[faceIndexInternal];
    int faceIndexExternal = neighborFaceIndex(faceIndexInternal);
    Vector<int> IS_external =
neighbor.referenceGrid().faceIndexSets()[faceIndexExternal];
    Vector<FloatingDataType>& q_external = neighbor.vector("q");
    Vector<FloatingDataType>& flux_x_external =
neighbor.vector("fx");
    Vector<FloatingDataType>& flux_y_external =
neighbor.vector("fy");
    Vector<FloatingDataType>& u_x_external =
neighbor.vector("ux");
    Vector<FloatingDataType>& u_y_external =
neighbor.vector("uy");

    for (auto i = 0; i < IS_internal.size(); i++) {
        int ii = IS_internal[i];

        FloatingDataType u_internal =
fabs(u_x_internal[i]*nhat.x() + u_y_internal[i]*nhat.y());
        FloatingDataType u_external =
fabs(u_x_external[i]*nhat.x() + u_y_external[i]*nhat.y());
        FloatingDataType lambda = fmax(u_internal, u_external);
        FloatingDataType f_star_x = 0.5*(flux_x_internal[ii] +
flux_x_external[ii] - lambda*(q_external[ii] -
q_internal[ii])*nhat.x());
        FloatingDataType f_star_y = 0.5*(flux_y_internal[ii] +
flux_y_external[ii] - lambda*(q_external[ii] -
q_internal[ii])*nhat.y());

        R[ii] += w[i] * J[ii] * (f_star_x*nhat.x() +
f_star_y*nhat.y());
    }

    return R;

}
```

Advancing in Time

Provided in the `examples/math597-P4/main.cpp` file, we have the logic for advancing the solution in time:

```
void advance(HydroForest::ElementMesh2D<double>& mesh, double
```

```
tFinal, bool plotFlag) {

    // Initialize all the things
    HydroForest::HydroForestApp& app =
HydroForest::HydroForestApp::getInstance();
    HydroForest::Options& options = app.getOptions();

    // Store mass matrix in all elements
    mesh.iterate([&](HydroForest::QuadElement2D<double>& element)
{
        HydroForest::DGMassMatrixInexact2D<double> M(element);
        element.vectorMap()["mass"] = M;
        element.vectorMap()["mass-inverse"] = M.inverse();
    });

    // Get max speed and length
    double maxSpeed = 0;
    double maxLength = 0;
    mesh.iterate([&](HydroForest::QuadElement2D<double>& element)
{
        // Get max speed
        HydroForest::Vector<double>& ux = element.vector("ux");
        HydroForest::Vector<double>& uy = element.vector("uy");

        for (auto& u_i : ux.data()) maxSpeed = fmax(maxSpeed,
fabs(u_i));
        for (auto& u_i : uy.data()) maxSpeed = fmax(maxSpeed,
fabs(u_i));

        // Get max length
        double len = sqrt(element.area());
        maxLength = fmax(maxLength, len);
    });

    // Iterate over time
    double tStart = 0;
    int nTime = 100000;
    double time = tStart;
    HydroForest::RungeKutta3<double> timeIntegrator;
    HydroForest::Vector<double> plotTimes =
HydroForest::vectorLinspace(tStart, tFinal, 20);
    int nPlot = 0;
    bool doPlot = false;
    double CFLAdjust = 5e-2;
    for (auto n = 0; n <= nTime; n++) {

        // Time advance logic
```

```
        double dt =
CFLAdjust*timeIntegrator.getMaxTimeStep(maxSpeed, maxLength);
        if (time + dt > tFinal) {
            dt = tFinal - time;
        }
        if (time + dt >= plotTimes[nPlot] && plotFlag) {
            dt = plotTimes[nPlot] - time;
            doPlot = true;
        }
        time += dt;
        app.log("Timestep = %i, Time = %f, dt = %f", n, time, dt);

        // Iterate over elements
        mesh.iterate([&](HydroForest::QuadElement2D<double>&
element) {

            // Get element values
            int N = element.size();
            HydroForest::Vector<double>& q_n =
element.vector("q");
            HydroForest::Vector<double>& ux =
element.vector("ux");
            HydroForest::Vector<double>& uy =
element.vector("uy");
            HydroForest::Vector<double>& fx =
element.vector("fx");
            HydroForest::Vector<double>& fy =
element.vector("fy");
            HydroForest::Vector<double>& s = element.vector("s");
            HydroForest::Vector<double>& M_inv =
element.vector("mass-inverse");

            // Update via time integration
            HydroForest::Vector<double> q_update =
timeIntegrator.update(time, dt, q_n, [&]
(HydroForest::Vector<double> q_n) {

                element.vector("q") = q_n;

                // Construct RHS vector
                HydroForest::Vector<double> R(N, 0);
                {
                    HydroForest::Vector<double> R_volume =
element.computeVolumeIntegralInexact();
                    HydroForest::Vector<double> R_flux(N, 0);
                    for (auto l = 0; l < 4; l++) {
                        R_flux +=
```

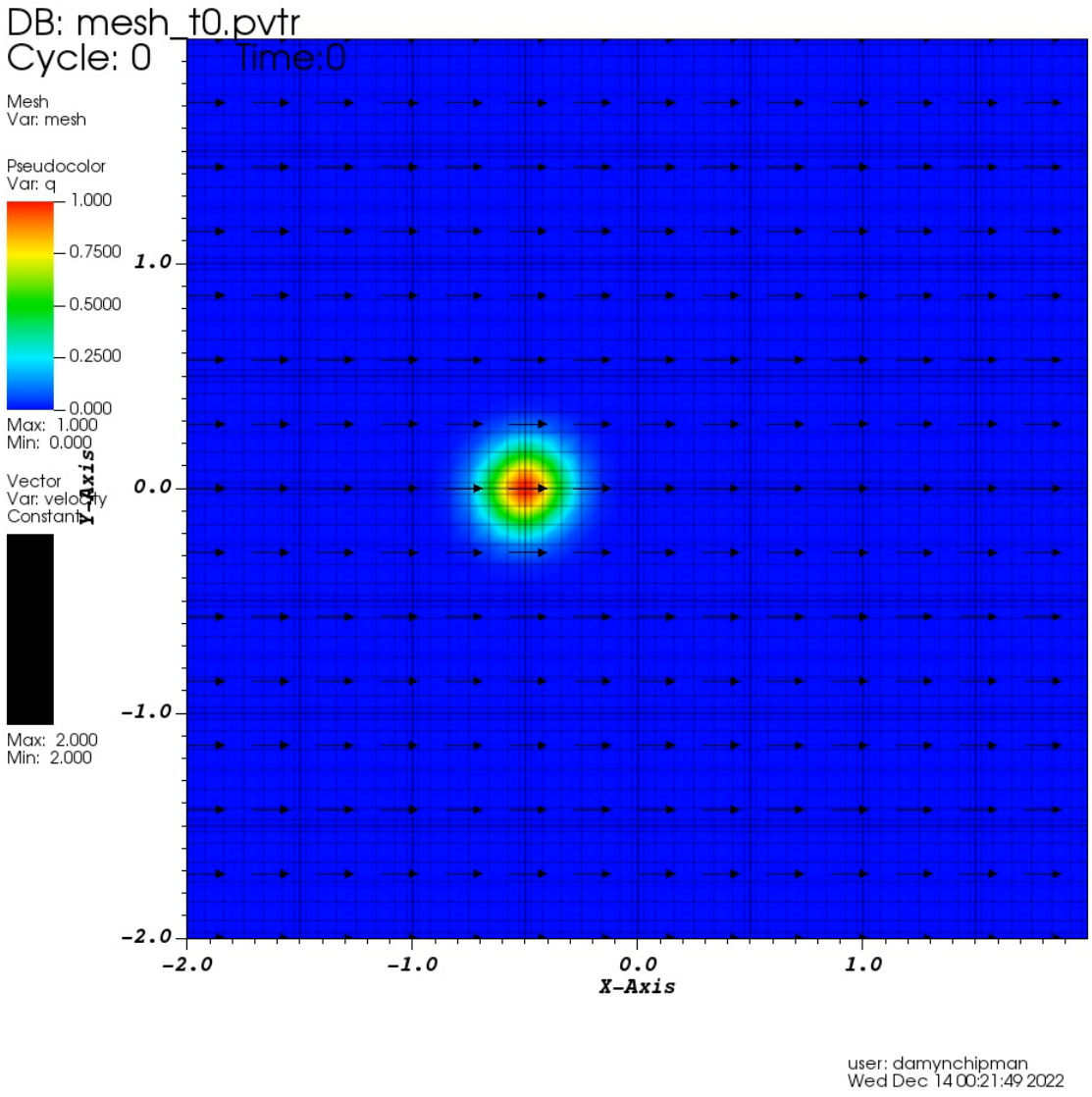
```
element.computeFluxIntegralInexact(l);
                    }
                    R = R_volume - R_flux;
                }

                // Apply inverse of mass matrix
                HydroForest::Vector<double> Rhat(N, 0);
                for (auto i = 0; i < N; i++) {
                    Rhat[i] = M_inv[i] * R[i];
                }

                // Add source term
                Rhat += s;

                return Rhat;
            });

            // Update variables
            for (auto i = 0; i < N; i++) {
                element.vector("q")[i] = q_update[i];
                element.vector("fx")[i] = q_update[i] * ux[i];
                element.vector("fy")[i] = q_update[i] * uy[i];
            }

            return;
        });

        if (doPlot && plotFlag) {
            mesh.toVTK("mesh_t" + std::to_string(nPlot), {"q",
"ux", "uy", "fx", "fy", "s", "q_final"});
            nPlot++;
            doPlot = false;
        }

        if (time >= tFinal) {
            break;
        }

    } // Time loop

}
```

With the pieces detailed above, we should have most of the pieces needed to solve our system of PDEs.

Current Results

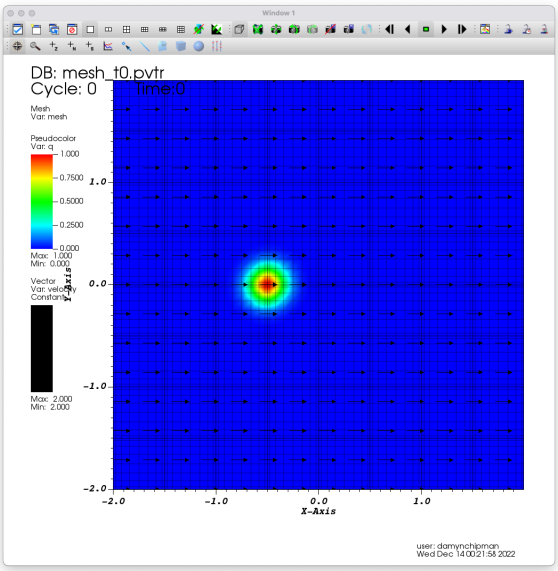
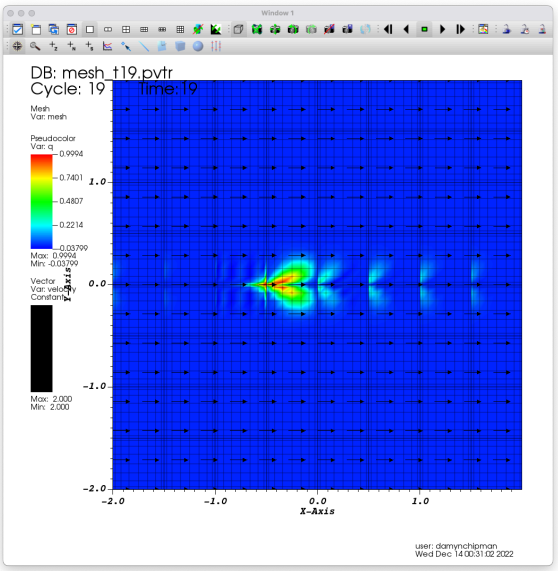
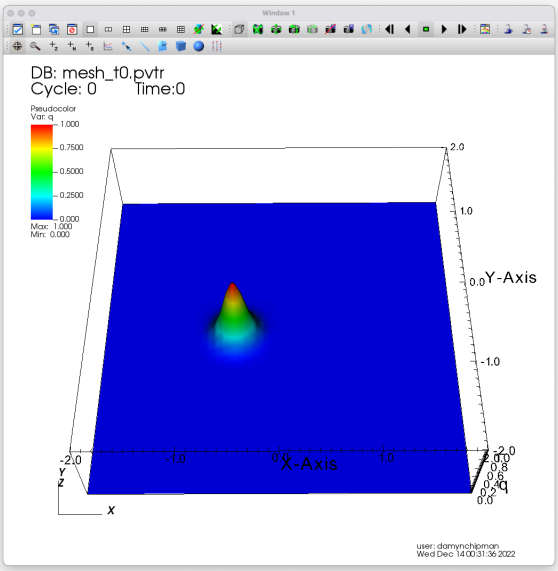
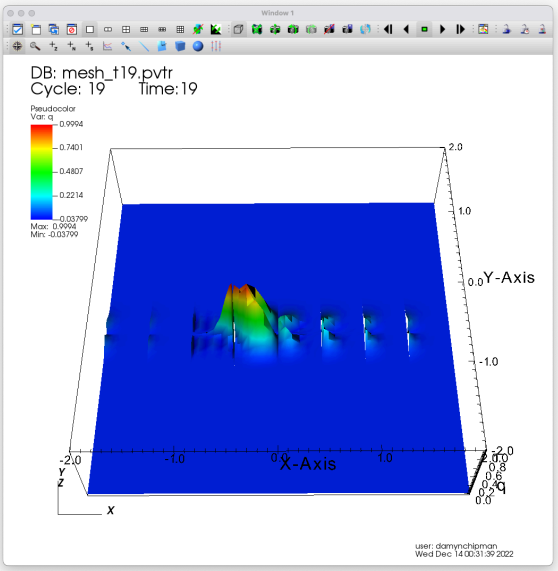
Unfortunately, the current results are buggy and incorrect... Thus, I am unable to provide convergence or timing results.

Below is an animation of the advection equation as currently implemented. This is done with 8th order elements on a level 3 refined grid (8x8 elements) for a total of 5184 degrees of freedom. This ran in about 3 minutes.



The solution *should* just advect with the velocity field to the right. However, there is some smearing and diffusing. Plus, the solution does not move as fast as it should; by the end of the simulation, the solution should have already gone around the periodic domain.

Below is another look at the initial and final snapshots of the current implementation:

`tStart`	`tFinal`
	
	

As this test case is not working, I have not implemented other more exciting cases such as a more complex velocity field or a test case with a system of PDEs.

Discussion

There could be a number of issues with the current implementation, though I have verified many pieces that do work. For example, I have verified that the 2D utilities work as intended (the 2D basis function plots and other test cases show this). I also have verified that the mesh and the elements are being constructed properly, with the proper neighbor pattern and periodicity. The `p4est` utilities are helpful for proving that. In addition, though we don't use the system matrices, I have constructed the mass, derivative, and flux matrices for linear elements and checked them with the results in the textbook and they match. This proves to me that I am constructing `psi` and it's various flavors properly.

Currently, I believe the issue to be in the flux and the right-hand side vectors R_{VOLUME} and R_{FLUX}. I have exhausted my time debugging up to this point, but with more time, that is my next guess.

Future Endeavors

My hope for this code is to serve as a spring board into finite elements, AMR, and hydrodynamics. While this project is currently due, I will continue working on this code base secondary to my dissertation work. My dissertation work is focused on solving elliptic PDEs on adaptive meshes. The equations in hydrodynamics would greatly benefit from a fast solver for the elliptic components such as the pressure field or mass diffusion. So, I foresee the chance to combine this code `HydroForest` with my dissertation code `EllipticForest`.

Future features to this code include:

- Getting this case to actually work...
- The ability to solve system of PDEs
- Adaptive mesh capability
  - `HydroForest` already wraps pieces of `p4est`, I just need to work on the 2-1 interfaces and fluxes
- Unstructured mesh
  - `p4est` has a Gmsh reader, and the `QuadElement2D` instances are stored in the `p4est` quadrants, so with little work, I should be able to create a Gmsh mesh file and create a mesh from that
- Integrate with dissertation code for elliptic solvers: `EllipticForest`
- Parallel implementation through `Petsc`

Most of these features would be part of a potential post-doc project. Who knows what the future holds?!

Conclusion

In conclusion, we looked at the 2D conservation form of a system of PDEs and derived a solution methodology using a discontinuous Galerkin approach. We derived an update formula for integrating the solution over time. We detailed how to compute the matrices and vectors needed for the update. In addition, we looked at some of the implementation details and the motivation behind those details, such as exact vs. inexact integration and the form of the volume and flux contributions. The 2D utilities, algorithms, and code are also provided. We looked at the current buggy results and potential issues and fixes. Finally, we discussed the future potential of this code base.

In [ ]: