Previously, we’ve developed a quite robust `Graph` class to let us use `Node` and `Edge` objects and their relationships easily and efficiently. It’s not hard to see how the `Graph` class could be further improved and generalized to many other problems in computational science like network analysis, molecular dynamics, etc.

Another large class of computational methods include finite volume and finite element methods. These methods also use `Node` and `Edge` objects, but often rely on even higher order objects like triangles or tetrahedra. Below, we take a look at a classic finite volume method to see what kind of relationships between triangles, edges, and nodes we would want to represent in order to cleanly and robustly implement a problem of this type.

<table>
<thead>
<tr>
<th>Background – Shallow Water and Finite Volume Method</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Math!</td>
<td>2</td>
</tr>
<tr>
<td>Visualization</td>
<td>4</td>
</tr>
<tr>
<td>Problem 1 - Mesh Design</td>
<td>5</td>
</tr>
<tr>
<td>Implementation Details</td>
<td>6</td>
</tr>
<tr>
<td>Visualization</td>
<td>6</td>
</tr>
<tr>
<td>Working out the edge flux</td>
<td>6</td>
</tr>
<tr>
<td>Distributed Code and Test Meshes</td>
<td>7</td>
</tr>
<tr>
<td>Initial and Boundary Conditions</td>
<td>8</td>
</tr>
<tr>
<td>Problem 2 - Mesh and FVM Implementation</td>
<td>10</td>
</tr>
<tr>
<td>Submission Instructions</td>
<td>11</td>
</tr>
</tbody>
</table>
Background – Shallow Water and Finite Volume Method

Math!

First, the mathematical model. The 2D shallow water equations are a specific case of the Euler fluid dynamics equations and model fluid waves whose scale are proportional to or greater than the depth of the fluid. The shallow water equations are used in extrapolating tsunamis, modeling atmospheric waves and flows, and analyzing estuary and lake flows. We’ll use them to throw pebbles into a pond.

The shallow water equations in 2D can be written as a system of partial differential equations. These PDEs are derived from the fluid’s conservation of mass and conservation of momentum and act over the independent variables time, \( t \), and two space coordinates, \( x \) and \( y \). At each point, these equations describe how an infinitesimal column of water acts. The dynamics of this column are defined in terms of three variables: the depth (or height) of the column, and the \( x \) and \( y \) velocities of the water at that point in the column, which are averaged over the column’s depth. We start off by giving the full PDEs, but hold on to your hats: through a series of transformations we’ll end up with a quite simple computational model.

The shallow water PDEs are

\[
\frac{\partial}{\partial t} \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} hu^2 + gh^2/2 \\ hv \\ hv \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} hu \\ hv \\ hv^2 + gh^2/2 \end{bmatrix} = 0
\]

(1)

where \( h \) is the water depth, \( u \) and \( v \) are the depth-averaged velocity components in the \( x \) and \( y \)-directions respectively, and \( g = 9.81 m/s^2 \) is the acceleration due to gravity.

To simplify, we define \( Q \) as a vector holding all the important characteristics of a water column as function of space and time. Let

\[
Q(x,t) = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix} \quad \quad F_1(Q) = \begin{bmatrix} hu \\ hu^2 + gh^2/2 \\ hv \end{bmatrix} \quad \quad F_2(Q) = \begin{bmatrix} hv \\ hv^2 + gh^2/2 \\ hv \end{bmatrix}
\]

(2)

then Equation (1) can be written as

\[
\frac{\partial}{\partial t} Q + \frac{\partial}{\partial x} F_1(Q) + \frac{\partial}{\partial y} F_2(Q) = \frac{\partial Q}{\partial t} + \nabla \cdot F(Q) = 0
\]

(3)

where \( \nabla = [\frac{\partial}{\partial x}, \frac{\partial}{\partial y}] \) and \( F = [F_1, F_2]^T \). This is now written in the form of a 2D hyperbolic conservation law with conserved quantity \( Q \).

Suppose we now integrate these equations over some area in the domain called \( \Omega \). We get

\[
0 = \iint_\Omega \frac{\partial Q}{\partial t} + \nabla \cdot F \, dx = \frac{\partial}{\partial t} \iint_\Omega Q \, dA + \iint_\Omega \nabla \cdot F \, dx
\]

(4)
Recall the divergence theorem \( \int_{\Omega} \nabla \cdot \mathbf{F} \, dA = \oint_{\partial \Omega} \mathbf{F} \cdot \hat{n} \, ds \), which states that the total divergence of any differentiable vector field \( \mathbf{F} \) over a domain \( \Omega \) is equivalent to the total flux of \( \mathbf{F} \) over the boundary of the domain \( \partial \Omega \). Here, \( \hat{n} \) is the outward unit normal vector of the boundary. See Figure 1. This lets us convert area integrals into line integrals. Applying this to Equation (4), we get

\[
\frac{\partial}{\partial t} \int_{\Omega} Q \, dA + \int_{\partial \Omega} \mathbf{F} \cdot \hat{n} \, ds = \frac{\partial}{\partial t} \int_{\Omega} Q \, dA + \int_{\partial \Omega} \mathbf{F} \cdot \hat{n} \, ds
\]

where \( \partial \Omega \) is the boundary of \( \Omega \). Equation (5) is true for any compact set \( \Omega \) and simply states: A change in \( Q \) within some area is equal to the amount of \( Q \) that has passed through the border of that area.

Figure 1: Areas \( \Omega \) and \( T_k \), boundaries \( \partial \Omega \) and \( \partial T_k \), and outward unit normal vectors \( \hat{n} \).

Suppose we now take a triangle-centered approach. Let’s chop up \( \Omega \) into disjoint triangles \( \Omega = \bigcup_k T_k \) and let

\[
\overline{Q}_k(t) = \frac{1}{|T_k|} \int_{T_k} Q(x, t) \, dA
\]

be the average value of \( Q \) inside the \( k \)th triangle, \( T_k \). We can store these values at the center of the triangles and attempt to evolve them in time using the flow into and out of each neighboring triangle! To do so, we use Equation (6) to rewrite Equation (5) as

\[
|T_k| \frac{\partial \overline{Q}_k}{\partial t} = -\int_{\partial T_k} \mathbf{F} \cdot \hat{n} \, ds
\]

where \( \partial T_k \) is the boundary of the \( k \)th triangle. See Figure 1. First, we can expand the right-hand side. The boundary \( \partial T_k \) is simply the three edges of the triangle,

\[
= -\sum_{e \in T_k} \int_e \mathbf{F} \cdot \hat{n}_e \, ds
\]

We’ll work out the edge integral later. For now it is sufficient to know that it represents the flux of \( Q \) over the edge \( e \) and is a function of \( \overline{Q}_k \), the values in the triangle triangle across edge \( e \), \( \overline{Q}_m \), and the edge normal vector out of \( T_k \) into \( T_m \). See Figure 2. Let’s abbreviate the edge integral, \( \int_e \mathbf{F} \cdot \hat{n}_e \, ds \), as \( F^e_k = F^e_{km} = F(\overline{Q}_k, \overline{Q}_m, \mathbf{n}_{km}) \) for now. Then:

\[
|T_k| \frac{\partial \overline{Q}_k}{\partial t} = -\sum_{e \in T_k} F^e_k
\]
Figure 2: Triangle-centered flux transfer. A triangle exchanges flux with its neighbors. The flux for an edge $e$ is a function of the two adjacent triangles’ $Q$ value. For a conservative problem like this one, $F_{km} = -F_{mk}$. This means the flux going out of one triangle over a certain edge is equal to the flux coming into the neighbor triangle from that edge.

Finally, discretizing time, $Q^n_k = Q_k(t_n) = Q_k(t_0 + n\Delta t)$, a simple Euler time stepping makes the approximation

$$\frac{\partial Q_k}{\partial t} \approx \frac{Q^{n+1}_k - Q^n_k}{\Delta t}$$

so our stepping algorithm is

$$Q^{n+1}_k = Q^n_k - \frac{\Delta t}{|T_k|} \sum_{e \in T_k} F^e_k$$ (8)

**Visualization**

The **SDLViewer** is a node-centered viewer while the above model is a triangle-centered computation. At each visualization, we need to post-process the triangle-centered data to approximate node-centered data for visualization. We can approximate the value of $Q$ at the nodes of the mesh as the weighted average of the triangles around it:

$$Q_n = \frac{1}{|T(n)|} \sum_{T_k \in T(n)} |T_k| Q_k$$ (9)

where $T(n)$ is the set of triangles adjacent to node $n$ and $|T(n)|$ is the total area of all these triangles.
Problem 1 - Mesh Design

Clearly, this problem begs for a good representation of ensembles of triangles. Although we could add the notion of triangles to our Graph, what we’re actually interested in is a special type of graph that is made up of only nodes and edges that form triangles. We call this a Mesh.

Depending on your implementation, a (possibly incomplete) list of helpful operations that a Mesh designed for the above model could implement are:

- Adding triangles. [Derp]
- Storing and accessing information related to triangles, edges, and/or nodes. [Needed to store $Q_k$, $Q_n$, $F$, etc.]
- Accessing the area of a triangle. [Needed for Eq (8), (9)]
- Accessing outward normal vectors of an edge of a triangle. [Needed to compute the flux, Eq (7)] Be careful about this one.
- Accessing nodes of a triangle/edge. [Needed to compute normals/areas]
- Accessing edges of a triangle. [Needed to compute fluxes, Eq (7) and/or (8)]
- Accessing adjacent triangles of a triangle and/or edge. [Needed to compute fluxes, Eq (7)]
- Accessing adjacent triangles of a node. [Needed to compute node values, Eq (9)]

Your team’s task for the first part of this homework is to provide specifications and representations of a Mesh class that abstracts triangles, but also keeps many (or all) of the operations we have developed for Nodes and Edges. Answer the following questions in your team’s write-up.

1. What is the public interface of your Mesh class?
2. What is the (estimated) complexity of each public member function of your Mesh class? (Easiest to provide this info along with your public interface in Q1).
3. What are the (interesting) abstraction functions your Mesh class will implement?
4. What is the representation of your Mesh class?
5. Does your Mesh class obey any (interesting) representation invariants? What are they?
6. For the shallow water model above, what information should be associated with triangles? edges? nodes?
Implementation Details

Visualization

We have abstracted the notion of position in the SDLViewer. Like node color, you may now pass a functor to the SDLViewer which takes a Node and returns a Point representing the Node’s position. This defaults to node.position() of course. This way, you may easily plot \( h \), \( hu \), or \( hv \) at each node in place of the node’s \( z \)-coordinate to get a good visualization.

Working out the edge flux

We now want to simplify and get an explicit form for the edge flux, \( F^e \), so we can write our integrator. We defined the edge flux as

\[
F^e_k = \int_e \mathbf{F} \cdot \hat{n}^e_k \, ds
\]

First, on each edge the unit outward normal \( \hat{n}^e_k \) is constant,

\[
= \hat{n}^e_k \cdot \int_e \mathbf{F} \, ds
\]

Expanding this out in terms of \( \mathbf{F}_1 \) and \( \mathbf{F}_2 \), and abusing \( \mathbf{n} \)’s notation, we have

\[
= \hat{n}^e_x \int_e \mathbf{F}_1(Q) \, ds + \hat{n}^e_y \int_e \mathbf{F}_2(Q) \, ds
\]

but how do we evaluate the integrals? Since we have approximated the value of \( Q \) as a constant on each triangle and now wish the perform this integral on an edge between two triangles, the easiest thing we can do is take the average of both to compute the flux,

\[
= \frac{|e|}{2} \left[ (\mathbf{F}_1(\overline{Q}_k) + \mathbf{F}_1(\overline{Q}_m)) \hat{n}^e_x + (\mathbf{F}_2(\overline{Q}_k) + \mathbf{F}_2(\overline{Q}_m)) \hat{n}^e_y \right]
\]

where \( \overline{Q}_k \) is the value we have stored for triangle \( T_k \) (with outward normal \( \mathbf{n}^e_k \)) and \( \overline{Q}_m \), \( m \neq k \) is the value we have stored for triangle \( T_m \). See Figure 2. We can write this out explicitly as

\[
= \frac{|e|}{2} \left( \begin{array}{c}
h_k u_k + h_m u_m \\
h^2_k / 2 + h_m u^2_m + gh^2_m / 2
\end{array} \right) \hat{n}^e_x + \frac{|e|}{2} \left( \begin{array}{c}
h_k v_k + h_m v_m \\
h_k u_k v_k + h_m u_m v_m
\end{array} \right) \hat{n}^e_y
\]
This can be simplified by defining the edge normal velocity \( w = u\hat{n}_x + v\hat{n}_y = (u, v) \cdot \hat{n}_e \) and rewriting,

\[
\frac{|e|}{2} \begin{pmatrix}
    h_k w_k + h_m w_m \\
    h_k v_k w_k + h_m v_m w_m + \frac{g}{2} (h_k^2 + h_m^2) \hat{n}_x
\end{pmatrix}
\]

This numerical flux is correct, but leads to an unstable integration scheme. It can be fixed using a dissipative flux, where we add an extra term

\[
\frac{|e|}{2} \begin{pmatrix}
    h_k w_k + h_m w_m \\
    h_k v_k w_k + h_m v_m w_m + \frac{g}{2} (h_k^2 + h_m^2) \hat{n}_x
\end{pmatrix} - \alpha\Delta t \begin{pmatrix}
    h_m - h_k \\
    h v_m - h v_k
\end{pmatrix}
\]

In this case, we’ll use the local Lax-Friedricks scheme which sets \( \alpha = \max(V^2_k, V^2_m) \) where \( V = \sqrt{u^2 + v^2 + \sqrt{gh}} \).

**Aside:** This extra term can be derived from adding the dissipation term \( \alpha\Delta t \nabla^2 Q \) to the right-hand side of Equation (3). Note that the numerical scheme remains consistent in the limit \( \Delta t \to 0 \), but now waves will dissipate or “damp out” over the course of the simulation.

**Aside:** A possibly helpful identity:

\[ F_{km} = -F_{mk} \]

We have included in the distributed code a reference flux functor that implements these operations. See the documentation for this function. You may use or edit this functor as you wish.

**Distributed Code and Test Meshes**

We have included a number of triangular meshes denoted with _tris.txt_.

**Obtaining the Code**

To get our starter code for HW4B:

// Check the status of your git repository
// Should show no changes (all mods and commits should be included in your HW3)
// Should say on branch master
$ git status

// If it doesn’t say on branch master, save a HW3 branch
// and checkout the master branch
$ git branch hw3
$ \text{git checkout master}

// Load our changes into your repository (but don’t apply them yet)
$ \text{git fetch cs207}

// Apply our changes to your repository
$ \text{git merge cs207/master}

Then follow git’s instructions. For example, if there are conflicts, fix them and commit
the result with \text{git commit -a}. If the merge produces too many conflicts, try \text{git rebase}
cs207/master instead.

\section*{Initial and Boundary Conditions}

\subsection*{Boundary Conditions}

Since we are using a two-dimensional triangular mesh, we impose boundary conditions when
we compute the flux on an edge that is incident to only one triangle. In this case, let $Q_I$
be the values for the triangle that exists and $Q_G$ be the value for the triangle that does not
exist, or the “ghost” triangle. Set the ghost triangle’s value to

$$Q_G = \begin{pmatrix} h_I \\ 0 \\ 0 \end{pmatrix}$$

where $h_I$ is the water height on the interior triangle. This will impose boundary conditions
that act like walls so waves will be reflected.

\subsection*{Initial Conditions}

It is very important to note that if $h$ goes to zero, the original shallow water equations
degenerate and become ill-defined. For the meshes we gave you, implement the following
initial conditions to be sampled at the positions of each node (initial triangle values, $Q_k(t = 0)$, can be computed by averaging the initial node values):

$$Q(x, 0) = \begin{pmatrix} 1 - 0.75e^{-80((x-0.75)^2+y^2)} \\ 0 \\ 0 \end{pmatrix}$$

This models a pebble being thrown in and causing a depression. Another interesting case is

$$Q(x, 0) = \begin{pmatrix} 1 + 0.75H((x - 0.75)^2 + y^2 - 0.15^2) \\ 0 \\ 0 \end{pmatrix} \quad H(x) = \begin{cases} 1 & \text{if } x < 0 \\ 0 & \text{otherwise} \end{cases}$$
which releases a large column of water and causes a very sharp, fast wave. Finally, for the dam meshes, the initial condition

\[ Q(x, 0) = \begin{pmatrix} 1 + 0.75H(x) \\ 0 \\ 0 \end{pmatrix} \]

models a dam break and a temporary waterfall as the two sides attempt to equalize.
Problem 2 - Mesh and FVM Implementation

Implement your design of the Mesh class and use it to integrate the shallow water equations with some initial condition and the given boundary conditions. Your Mesh implementation need not be complete – more complicated functions like remove do not have to be implemented – but should include all the functions and relations you need to cleanly implement all three shallow water simulations. It should be obvious from your code how to enable all three initial conditions above. You may also include other initial conditions and/or meshes of your own design; describe them in the README.txt file.

See Piazza for a few gifs of the expected results and debugging information.
Submission Instructions

Since you made staff-cs207 “collaborators” on your git repository, we will examine your work in that repository.

Use a git tag to mark the version of code you want to submit. Here’s how:

```bash
$ git add <files to track>                  # Tells git which files to track
$ git status                                # View files git is tracking
$ git commit -am "Describe last few edits"  # Commit files
$ git tag -am "My HW4" hw4                  # Tag this commit with tag ‘‘hw4’’
$ git push --tags                           # Push all commits to git repo
```

It is possible to change your tag later, too, if you discover a bug after submitting:

```bash
$ git tag -d hw4                           # Overwrite tag
$ git tag -am "My HW4" hw4                # Tag new commit
$ git push --tags                         # Push new tag
```

We will use Git timestamps on tags and the associated commits to check deadlines.

Be careful with overwriting tags – you don’t want to lose the submission tag. We can handle versioned tags, such as hw4_1, hw4_2, etc.

To verify that all of the files were pushed correctly, you can click on your repository to the right of your code.seas account and view the “Source Tree”. There may be a delay between your push and the files showing up in the browser.