CSCI 8980 Assignment 3b:
Eulerian Fluid Simulation

Due date: Tuesday, November 22, 2016

This assignment is fairly short and in 2D to avoid taking too much time away from your final projects.

We will implement a basic grid-based incompressible fluid simulator with smoke tracking, buoyancy, and static obstacles. The two key steps are advection (of the velocity field, as well as of smoke concentration) and pressure projection.

1 Basic setup

In a grid-based fluid simulator, the main state variable is the velocity field $\mathbf{u}$, stored on a staggered grid. We will also need some other grids to store the pressure, smoke concentration, obstacle flags, and other auxiliary data. These are generally scalar data and will therefore be stored on regular grids. Eventually we will create a Fluid class to hold everything together, but for now our program will keep them as separate variables in the main function.

The class StaggeredGrid stores vector fields with one component on each cell face. In practice, this is implemented as two regular grids (three grids in 3D), one for each component. We will use the convention that a Grid of size $m \times n$ with origin $(x_0, y_0)$ and grid spacing $h$ (x0 and dx in the code) stores $m \times n$ scalar values at grid points

$$(x_0, y_0), \quad (x_0 + h, y_0), \quad \ldots \quad (x_0 + mh, y_0)$$
$$(x_0, y_0 + h), \quad (x_0 + h, y_0 + h), \quad \ldots \quad (x_0 + mh, y_0 + h)$$
$$\vdots \quad \vdots \quad \vdots$$
$$(x_0, y_0 + nh), \quad (x_0 + h, y_0 + nh), \quad \ldots \quad (x_0 + mh, y_0 + nh).$$

An $m \times n$ StaggeredGrid with origin $(x_0, y_0)$ and spacing $h$ stores vector field components on the face centers (really, the edge centers in 2D) of a grid with $m \times n$ square cells of size $h$. That is, the $x$-component values are stored at the edges normal to the $x$-axis, starting from $(x_0, y_0 + h/2)$ onwards and forming a $(m + 1) \times n$ grid, and similarly for the $y$-component values (see below).
Be careful that if we want to store scalar values on the centers of the corresponding cells, you would have to use an \( m \times n \) Grid with origin \((x_0 + h/2, y_0 + h/2)\).

Implement the StaggeredGrid constructor to create the underlying Grids for the vector field components \( x \) and \( y \). Also implement the method interpolate to find the interpolated vector field value at a point \( x \) by calling interpolate on the component Grids.

2 Advection

As a test vector field, we will use

\[
\mathbf{u}(x, y) = \begin{bmatrix}
-\sin(\pi x) \cos(\pi y) \\
\cos(\pi x) \sin(\pi y)
\end{bmatrix}
\]

which is a simple divergence-free velocity field in the \([0, 1] \times [0, 1]\) square. Create an \( n \times n \) StaggeredGrid on \([0, 1] \times [0, 1]\) in the main program and fill in its values using the provided function.

Implement the function advectPoint which moves a point \( \mathbf{x}^0 \) through the velocity field \( \mathbf{u} \) for time \( \Delta t \) and returns its final location. That is, it integrates the ODE \( \dot{\mathbf{x}} = \mathbf{u}(\mathbf{x}) \) for a single time step \( \Delta t \) starting from \( \mathbf{x}(0) = \mathbf{x}^0 \). Use StaggeredGrid::interpolate to evaluate \( \mathbf{u}(\mathbf{x}) \), and RK2 to perform the integration. (You don’t have to use our existing TimeIntegrator to do it, as it’s designed for second-order systems of the form \( M\ddot{\mathbf{x}} = \mathbf{f} \). Just apply the RK2 formulas directly.)

In the main loop, use advectPoint to move each tracer particle through the velocity field. Now, the particles should circulate smoothly around the domain.

Next, implement semi-Lagrangian advection for in advectScalarField and advectVectorField. The former should take in two Grids, one which is the initial state and one into which you will write the advected field. For each cell in the output grid, call advectPoint with a time step \(-\Delta t\) to find the location to interpolate from in the input grid. To advect a vector field, simply perform scalar advection on each component.

Since we don’t want to overwrite the values in the grid we are advecting from, we have to use an auxiliary grid to write to every time we perform advection. The standard approach is to then swap their pointers so that the output grid is treated as the current grid, while the original grid serves as the auxiliary grid next time. The starter code already takes care of creating two grids of the same size and shape. In the main loop, call advectScalarField and then swap the grid pointers. Now, the values in the background grid should also begin to circulate. There will be a lot of numerical diffusion; it will decrease with larger
n, but will never completely go away. You should verify that the motion of the blob and the tracer particles are consistent.

### 3 Pressure projection

A Fluid class has been added that represents a bare-bones fluid simulator — with the fluid completely filling a rectangular domain, and no extra internal state apart from the velocity field \( \mathbf{u} \). Time stepping the fluid only involves performing self-advection on \( \mathbf{u} \), then a pressure projection to make it divergence-free. The pressure projection part is what we must implement next.

As a warm-up, implement two functions divergence and gradient in grid.cpp. The divergence function should take a StaggeredGrid \( \mathbf{v} \) and a Grid<double> \( c \) as input and write \( \nabla \cdot \mathbf{v} \) into \( c \), and vice versa for gradient. You will need to be careful about which entries in the staggered grid are adjacent to the chosen cell and vice versa; see the figure in Sec. 1 for reference. At the grid boundaries, the gradient is not defined, so set it to 0 there. Verify that if you create a \( 3 \times 3 \) scalar field

\[
c = \begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

its gradient \( \nabla c \) has components

\[
\partial_x c = \frac{1}{\Delta x} \begin{bmatrix}
0 & 0 & 0 \\
0 & 1 & -1 \\
0 & 0 & 0
\end{bmatrix}, \quad \partial_y c = \frac{1}{\Delta y} \begin{bmatrix}
0 & 0 & 0 \\
0 & -1 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

and that the divergence of \( \nabla c \), in turn, is

\[
\nabla^2 c = \nabla \cdot (\nabla c) = \frac{1}{\Delta x^2} \begin{bmatrix}
0 & 1 & 0 \\
1 & -4 & 1 \\
0 & 1 & 0
\end{bmatrix}.
\]

We will perform pressure projection by solving the partial differential equation

\[
\nabla^2 p = \nabla \cdot \mathbf{u},
\]

and then updating the velocity via \( \mathbf{u} = \mathbf{u} - \nabla p \). (Note that we have omitted the factor \( \rho^{-1} \Delta t \) from both equations; this changes the scaling of \( p \) but not the resulting \( \mathbf{u} \).) Unlike the approach in the Bridson and Müller-Fischer notes which uses a custom linear solver specialized to work directly on grids, we will form a matrix-vector system \( \mathbf{A} \mathbf{p} = \mathbf{d} \) and use Eigen to solve it. So we have to translate between the values of \( p \) and \( \nabla \cdot \mathbf{u} \) on grid cells and the entries in the vectors \( \mathbf{p} \).
and \( \mathbf{d} \), similar to how we transferred state data from particles to the vectors \( \mathbf{x} \) and \( \mathbf{v} \) in a mass-spring system. The `enumerateCells` function is provided to give each grid cell an index into the corresponding vector (much like the index of each particle in a mass-spring system), and return the total number of grid cells \( N \).

Implement this in the `pressureProjection` function. Here is a rough sketch of how to go about it.

1. Using divergence or otherwise, compute \( \nabla \cdot \mathbf{u} \). Store it into a `VectorXd` of size \( N \) using the cell indices.
2. Build an \( N \times N \) sparse matrix \( \mathbf{A} \) using the `SparseMatrixBuilder` helper class. Its entries should correspond to the finite difference coefficients
   \[
   (\nabla^2 p)_{i,j} \approx \frac{1}{\Delta x^2} (p_{i-1,j} + p_{i+1,j} + p_{i,j-1} + p_{i,j+1} - 4p_{i,j}).
   \]
   To do this, iterate over each cell \((i,j)\) in the grid, and use \( \mathbf{A}.\text{add}() \) to populate the corresponding row using the indices of the cell and of its neighbors. At boundaries, assume solid boundary conditions \((p_{\text{neighbor}} = p_{i,j})\); therefore the corresponding coefficient should be added to the diagonal entry instead.
3. Use \( \mathbf{A}.\text{solve}(\mathbf{d}) \) to obtain the pressure field and, again, use the cell indices to store it back on a grid.
4. Using gradient or otherwise, update the velocity field via \( \mathbf{u} - = \nabla p \). Conveniently, our gradient convention \((\nabla p |_{\text{normal}} = 0 \) at boundaries\) is compatible with the solid boundary conditions, so we don’t have to do anything special there.

Some suggestions for testing and debugging as you go:

- As a sanity check, verify that \( \mathbf{A} \) applied to the \( 3 \times 3 \) grid \( \mathbf{c} \) defined earlier gives the same \( \nabla \cdot (\nabla \mathbf{c}) \). You could consider defining functions `gridToVector` and `vectorToGrid` to make this easier; then you would be looking at `vectorToGrid(A.getMatrix() * gridToVector(c))`.
- When first implementing the pressure projection, don’t call `step()` in the main program, just call `pressureProjection()` and visualize the computed scalar field. For example, once you’ve computed \( \nabla \cdot \mathbf{u} \), you could store it into the \( \mathbf{p} \) variable and check that the divergence looks correct. Similarly, if you solve for \( p \) but don’t apply the velocity update, you’ll be able to see the pressure distribution as you manipulate the velocity field.