Homework 2 – Due October 10th, 11:59pm ET

In this homework, you will write an begin to abstract a simple ordinary differential equation (ODE) integrator using your Graph from HW1. To create an efficient, general model, we will be using templates and functors in our algorithm implementation and data abstractions to generalize and optimize our Graph.

<table>
<thead>
<tr>
<th>Setup</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1 - Intro Mass-Spring [15%]</td>
<td>3</td>
</tr>
<tr>
<td>Scary Math</td>
<td>3</td>
</tr>
<tr>
<td>Modifiable Node Position</td>
<td>3</td>
</tr>
<tr>
<td>Application: Mass-Spring!</td>
<td>4</td>
</tr>
<tr>
<td>Problem 2 - More Graph Operations [40%]</td>
<td>5</td>
</tr>
<tr>
<td>Generalized Graph Edge</td>
<td>5</td>
</tr>
<tr>
<td>Node and Edge Remove</td>
<td>5</td>
</tr>
<tr>
<td>Application: Better Mass-Spring!</td>
<td>6</td>
</tr>
<tr>
<td>Problem 3 - Generalizing Forces [20%]</td>
<td>8</td>
</tr>
<tr>
<td>Problem 4 - Generalizing Constraints [25%]</td>
<td>9</td>
</tr>
<tr>
<td>Submission Instructions</td>
<td>12</td>
</tr>
</tbody>
</table>
Setup

Our helper code for Homework 2 extends the code from Homework 1. To retrieve it, follow these steps:

# Check the status of your git repository
$ git status

# Should say "on branch master"
# Otherwise, save a HW1 branch and checkout the master branch
$ git branch hw1
$ git checkout master

# Should also show no changes (all commits should already be made for HW1)
# Otherwise, commit your changes.
$ git commit -am "Stray HW1 changes"

# Load our changes into your repository
$ git fetch cs207

# Apply our changes to your repository
$ git merge cs207/master

Then follow git’s instructions. For example, if there are conflicts, fix them and commit the result with git commit -a. If the merge produces too many conflicts, try git rebase cs207/master instead.
Problem 1 - Intro Mass-Spring [15%]

With your Graph class, we can make a simple mass-spring physics model that is often used in graphics for cloth simulations and some deformable models.

Scary Math

First, the math. Let \( x_i \) and \( m_i \) be the position and mass of the \( i \)th node respectively. Then, from Newton’s Second Law, \( F = ma \), we want to integrate the equation

\[
m_i \frac{\partial^2 x_i}{\partial t^2} = F_i(X, t)
\]

where \( X = [x_0, x_1, ..., x_{N-1}] \) represents all of the positions and \( F_i(X, t) \) is the force on the node \( i \) at time \( t \).

Let the set \( A_i(X) \) denote the set of all nodes in \( X \) adjacent to node \( i \). For a system of springs, the spring force can be written as

\[
f_{i}^{\text{spring}}(X, t) = \sum_{x_j \in A_i(X)} -K \frac{x_i - x_j}{|x_i - x_j|} (|x_i - x_j| - L)
\]

where \( K \) is a spring constant, \( L \) is the spring rest length, and \( |\cdot| \) is the euclidean distance between position \( x_i \) and \( x_j \) (length).

Then, the total force applied to node \( i \) is the sum of the spring force and any other forces (for example, gravity or drag).

\[
F_i(X, t) = f_{i}^{\text{spring}}(X, t) + f_{i}^{\text{grav}}(X, t) + f_{i}^{\text{drag}}(X, t) + \cdots
\]

To integrate this through time, we choose the symplectic Euler method. First, we split the second order differential equation into two first order ones:

\[
\frac{\partial x_i}{\partial t} = v_i \quad \text{and} \quad \frac{\partial v_i}{\partial t} = F_i(X, t)/m_i
\]

We then discretize time by letting \( t^n = t_0 + n\Delta t \) and \( x_i^n = x_i(t^n) \) and make the approximation

\[
v_i^n = \frac{\partial x_i}{\partial t} \approx \frac{x_i^{n+1} - x_i^n}{\Delta t} \quad \text{and} \quad F_i(X^{n+1}, t^n)/m_i = \frac{\partial v_i}{\partial t} \approx \frac{v_i^{n+1} - v_i^n}{\Delta t}
\]

Thus, our update step looks like

\[
x_i^{n+1} = x_i^n + v_i^n \Delta t
\]

\[
v_i^{n+1} = v_i^n + F_i(X^{n+1}, t^n) \Delta t/m_i
\]

where we assume we have \( x_i^n \) and \( v_i^n \) and wish to compute \( x_i^{n+1} \) and \( v_i^{n+1} \).

Potential Gotcha: We have to be careful about \( n \) vs \( n + 1 \) in the update equations above. If we get this wrong, our “blanket” will go crazy. Specifically, all positions should be updated before forces are computed.
Modifiable Node Position

In order to compile and run mass.spring.cpp, the node position must be made modifiable. Provide your Node with the public function position that returns a reference:

```cpp
1 class Graph
2     class Node
3         Point& position();
```

Then, consider the implementation of symp_euler_step in mass.spring.cpp. You should not need to modify Graph for this problem any further. Note the data associated with each node: \( x_i \) (position), \( v_i \) (velocity), and \( m_i \) (mass).

Application: Mass-Spring!

You will need to set up initial conditions and define the force in Problem1Force to be applied. Use the initial conditions:

- Initial positions: \( x_i^0 = x_i(0) \) set from the initial coordinates of the nodes files.
- Zero initial velocity: \( v_i^0 = v_i(0) = 0 \).
- Mass: \( m_i = 1/N \) where \( N \) is the number of nodes in the graph (constant density).
- Spring constant: \( K = 100 \).
- Spring rest-length: \( L \) set to the initial length of the edges. In this problem all edges should have the same initial length. (You may find it useful to add a function “double Edge::length() const” to Edge.)

To define Problem1Force, use the spring force \( f_i^{\text{spring}} \) and add the force due to gravity \( f_i^{\text{grav}}(x, t) = m_i(0, 0, -g) \). Finally, to prevent the cloth from simply falling to infinity, we can constrain two corners of the cloth by returning a zero force from Problem1Force:

```cpp
1 if (n.position() == Point(0,0,0) || n.position() == Point(1,0,0))
2     return Point(0,0,0);
```

With these defined, you should be able to execute a rudimentary mass-spring simulation that looks something like Figure 1!
Problem 2 - More Graph Operations [40%]

Generalized Graph Edge

In Problem 1, all of the edges are considered to have the same rest length and spring constant, but it is easy to imagine a situation where we would like more freedom. In HW1, we templated the Graph so that Nodes could be associated with custom data. In this problem, we want a similar abstraction for edges.

Template your Graph on an edge_value_type with the following interface:

```cpp
template <typename V, typename E>
class Graph
{
    typedef V node_value_type;
    typedef E edge_value_type;

    class Edge
    {
        edge_value_type & value();
        const edge_value_type & value() const;
    }
};
```

Although it may be possible to accomplish this without adding additional data structures to your Graph implementation, this is not required (but impressive). Accessing an edge’s value should be as efficient as possible, no more than $O(d)$ time where $d$ is the largest degree of a node but hopefully less.

Node and Edge Remove

Finally, we would like to be able to remove Nodes and Edges.
class Graph
public:
  size_type remove_node(const Node&);
  node_iterator remove_node(node_iterator n_it);
  size_type remove_edge(const Node&, const Node&);
  size_type remove_edge(const Edge&);
  edge_iterator remove_edge(edge_iterator e_it);

where remove_edge should take at most $O(\text{num\ nodes}() + \text{num\ edges}())$ and remove_node can take at most $O(\text{num\ nodes}()^2)$, but hopefully a lot less. Removing a Node should remove all Edges that are incident to it.

You are required to write complete specifications for the remove functions. These specifications should include post-conditions on the Graph, complexity of the operation, and notes on all objects (Nodes, Edges, iterator) that are invalidated by these functions.

Before we continue, let’s clarify some of the specifications we have imposed on our Graph so far:

- A user should not be able to construct valid Node or Edge objects except by calling Graph methods.
- For any Node n of a Graph g, Node::index() returns an integer in $[0, \text{g.num\ nodes}())$.
- g.node(i).index() == i.
- g.node(n.index()) == n.
- num_edges() returns the number of unique undirected edges.
- Edge and Node should be lightweight – we expect to have many of them and want their construction and copy/assignment to be fast – and require no more than 16 bytes and 32 bytes respectively.
- All methods require $O(1)$ operations unless otherwise specified.
- No self-edges (edges from a Node to itself) or edges to Nodes that this Graph does not own should be allowed. Specify the preconditions and failure behavior.

Once you’re confident with everything Node related, compile test_nodes.cpp and see if you pass our tests.

Once you’re confident with everything Edge related, compile test_edges.cpp and see if you pass our tests.

These test some, but not all, of these specifications and provide a good sanity check before continuing.

**Application: Better Mass-Spring!**

When this is complete, each edge may now have its own spring constant $K_{ij}$ and rest-length $L_{ij}$. That is, the force may be generalized to

$$f_{\text{spring}}^i(X, t) = \sum_{x_j \in A_i(X)} -K_{ij} \frac{x_i - x_j}{|x_i - x_j|} (|x_i - x_j| - L_{ij})$$
You probably noticed the following code in `mass_spring.cpp`:

```cpp
#if 0
    // Diagonal edges,
    // Only include if you can handle edges of varying length
    graph.add_edge(nodes[t[0]], nodes[t[3]]);
    graph.add_edge(nodes[t[1]], nodes[t[2]]);
#endif
```

which excludes some edges from being added. These edges are the diagonal edges that have a rest length longer than the grid edges. By removing the `#if 0` and `#endif` lines, the diagonal edges will be added to the Graph. Initialize all edges to have a rest length equal to their initial length.
Problem 3 - Generalizing Forces [20%]

In Problem 1, your `Problem1Force` class represented a lot of different things. It was responsible for the force due to the edge-springs, the force due to gravity, and for a simple method of constraining nodes. These were combined into a single function object, making it hard to change its behavior or reuse partial behaviors. In practice, we’re likely to have a lot of different forces that we wish to write, upkeep, and apply separately.

Luckily, the function object passed to `symp_euler_step` can easily represent a combination of forces!

Replace your `Problem1Force` class with the following classes.

- `GravityForce` implements the force of gravity.
- `MassSpringForce` implements the spring forces.
- `DampingForce` a new force that implements damping (a form of friction). Use the damping force

\[ f_i^{\text{damping}} = -c v_i \]

where \( v_i \) is the velocity of node \( i \) and \( c \) is a damping constant.

Instead of passing all these forces to `symp_euler_step` individually – we would have to change the signature every time we added/removed a force! – design and implement a way to combine forces by adding their effects. For instance, the combination of `GravityForce` and `MassSpringForce` should equal the result of `Problem1Force` (but see below).

Design, specify, and implement these functions:

- `make_combined_force(f1, f2)` returns the combination force \( f_1 + f_2 \).
- `make_combined_force(f1, f2, f3)` returns the combination force \( f_1 + f_2 + f_3 \).

A combination force function object finds its component forces’ values and returns their sum. You can implement these functions in several ways. For instance, you can use an object-oriented programming style, or a style based on generic programming. We leave the details up to you. (Hint: How does `std::make_pair()` work?)

You may test your code by calling `symp_euler_step` (which has the same signature as in Problem 1) with the result of `make_combined_force(GravityForce(), MassSpringForce())`. You should see your blanket falling directly downward. This is because `Problem1Force` also applied constraints to node positions. Constraints don’t fit well into a combined-force framework; for now, implement the constraints on nodes \( (0, 0, 0) \) and \( (1, 0, 0) \) within `symp_euler_step` itself by simply skipping the update steps for these nodes. Once you do this, the results should be the same as in Problem 2.

As you may have noticed, the simulation never calms down or blows up. One property of symplectic methods is that they conserve some quantity exactly – in this case, a slightly modified notion of energy. To add a simple version of friction and remove energy from the system slowly over time, add your `DampingForce` with a damping coefficient of \( c = 1/N \), where \( N \) is the number of nodes in the the `Graph`.
Problem 4 - Generalizing Constraints [25%]

Where Problem 3 generalized forces, in Problem 4 we generalize constraints. A constraint is a restriction on the degrees of the freedom in the equation. We will consider constraints of the form

\[ C_i(x_i, t) = 0 \quad \text{and} \quad C_i(x_i, t) \leq 0 \]

We’ve seen an example in the previous problems already when we imposed equality constraints like

\[ x_0(t) = (0, 0, 0) \]

In Problem 1, we implemented these by returning zero force for nodes at particular positions. In Problem 3, we implemented these by skipping the update step on nodes at particular positions.

Some constraint properties that we want to capture are:

- Some constraints apply only to specific nodes. For example, keeping specific nodes fixed in Problem 1.
- Some constraints may apply to any node. For example, defining an obstacle that no node of the graph may penetrate, or an obstacle that destroys parts of the graph as it passes through.

More general constraints are much more difficult to implement accurately, but a good approximation is to simply find violated constraints after the position update and reset nodes’ positions and velocities before the forces are calculated. This doesn’t fit well into the force concept, so we create a new constraint concept.

Design constraints as functors that take a Graph and the time, search for nodes in the Graph that violate the constraint, and reset their positions and velocities. Implement your constant node constraints in this form (though an \( O(N) \) search should not be necessary) and a way to combine constraints together as we did for forces in Problem 3.

Add a constraint for a plane with the following properties:

- Plane \( z = -0.75 \).
- A node violates this constraint if \( x_i \cdot (0, 0, 1) < -0.75 \).
- To fix a Node that violates this constraint:
  - Set the position to the nearest point on the plane.
  - Set the z-component of the Node velocity to zero.

Add a constraint for a sphere with the following properties:

- Sphere center \( c = (0.5, 0.5, -0.5) \), sphere radius \( r = 0.15 \).
- A node violates this constraint if \( |x_i - c| < r \).
- To fix a Node that violates this constraint:
  - Set the position to the nearest point on the surface of the sphere.
  - Set the component of the velocity that is normal to the sphere’s surface to zero:

\[
    v_i = v_i - (v_i \cdot R_i)R_i
\]
where $R_i = (x_i - c)/|x_i - c|$ and the · is the inner (dot) product.

Add a constraint for a sphere with the following properties:

- Sphere center $c = (0.5, 0.5, -0.5)$, sphere radius $r = 0.15$.
- A node violates this constraint if $|x_i - c| < r$.
- To fix a node that violates this constraint:
  - Remove the node and all of its edges using your `remove_node` function.

**HINTS:**
- This is very similar to the `SphereConstraint`, except that rather than staying on the surface of the sphere, the parts of the grid that touch the sphere should disappear! Be sure that your `remove_node` function removes all edges that contain the removed node.
- Since you’re updating the nodes and edges, you’ll need to clear the current `viewer` and `node_map` and re-draw the graph after each `symp_euler_step`. You can do this with:

```cpp
// Clear the viewer's nodes and edges
viewer.clear()
node_map.clear()

// Update viewer with nodes' new positions and new edges
viewer.add_nodes(graph.node_begin(), graph.node_end(), node_map);
viewer.add_edges(graph.edge_begin(), graph.edge_end(), node_map);
```
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:

$ 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 HW2" hw2  
Tag this commit with tag ‘‘hw2’’

$ 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:

$ git tag -d hw2  
$ git tag -am "My HW2" hw2  
$ git push --tags

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 hw2_1, hw2_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.