CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Homework 2 – Due Friday February 14th, 4:30 P.M. PST via GitHub

---

1 Overview

1.1 Goals

In this homework, you will begin to abstract a simple ordinary differential equation (ODE) integrator using your `Graph` from HW1. You will implement a cloth simulation with various forces and constraints using your graph class. First you do a simple implementation of a mass spring system to get familiar with the concepts. Then you will create generalized forces and constraints using using inheritance and polymorphism. Finally you will implement the removal of nodes and edges.

Concepts Introduced functors, iterators, templates, and inheritace.

1.2 Tasks

- Make the `Node` position modifiable. (short)
- Implement a simple mass spring. (short)
- Add a `value` attribute to `Edge`. (short)
- Improve the mass spring. (short)
- Generalize Forces. (time intensive)
- Generalize Constraints. (time intensive)
- Implement `Node` and `Edge` removal. (**Very time intensive**)
- Implement the tear constraint. (time intensive)

1.3 Tips

- Start early!
- Make sure you understand the starter code provided before you write your own code.
- The `Node` and `Edge` removal is very bug prone so make sure you have ample time for this section.
- We have provided you with `test_nodes.cpp` and `test_edges.cpp` that will test the removal. Use these to debug, you can modify the scripts if you want.
- You can always come to the teaching team with questions.
- Any error you encounter someone else will have had before. Stackoverflow is your best friend.
- Don't forget to pull the starter code.
- We advise committing and pushing frequently as you make progress on the assignment.
- Work together.

1.4 Starter code

- `test_nodes.cpp` and `test_edges.cpp` provide simple tests to debug your code. Feel free to modify these.

- `mass_spring.cpp` Where you implement the mass spring system. With `NodeData` which stores the mass and velocity of a node. `symp_euler_step`, which implements the symplectic Euler method for updating the Nodes' positions and velocities. And `Problem1Force` where you implement the simple mass spring system.
- `CME212/gridX.nodes` and `CME212/gridX.tets`. Contain the graphs used for `mass_spring`.

2 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 CME212

# Apply our changes to your repository
$ git merge CME212/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 CME212/master` instead.

3 Intro Mass-Spring

Objectives

- Make `Node::position()` modifiable.
- Setup intial conditions.
- Implement `Problem1Force`

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.

3.1 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:

```
class Graph
    class Node
        Point& position();
```

Then, consider the implementation of `symp_euler_step` in `mass_spring.cpp`. **Note** the data associated with each node: x_i (position), v_i (velocity), and m_i (mass).

3.2 Application: Mass-Spring!

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

- 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 sets the initial length uniformly across all edges.[1]

When solving the numerical system, the order of actions is

1. update position, i.e. calculate x^{n+1} while we still have v^n, then
2. calculate force using the updated position, $F(x^{n+1})$, and finally
3. update velocity, i.e. v^{n+1}.

To define `Problem1Force`, use the spring force f_i^{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`:

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

With these defined, you should be able to execute a rudimentary mass-spring simulation!

4 More Graph Operations

Objectives

- Add a `value` attribute to the `Edge` class.
- Give each edge its own spring constant and rest length.
- Update `Problem1Force`

[1] It might be useful to add a function "`double Edge::length() const`" to `Edge`.

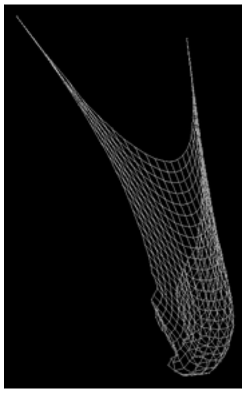


Figure 1: First version of mass-spring! (Rotated 90 degrees)

4.1 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 `Node`s could be associated with custom data. In this problem, we want a similar abstraction, but for edges!

Create a class template for your `Graph` on an `edge_value_type` with the following interface:

```
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.

4.2 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

$$f_i^{\text{spring}}(X,t) = \sum_{x_j \in A_i(X)} -K_{ij} \frac{x_i - x_j}{|x_i - x_j|} \left(|x_i - x_j| - L_{ij}\right)$$

You probably noticed the following code in `mass_spring.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. 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.*

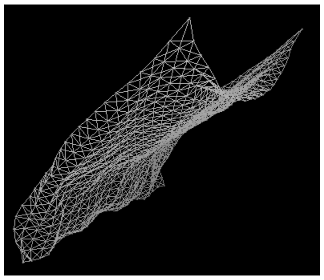


Figure 2: Diagonal edges included.

Hint: You should initialize all the edge values when you create he graph, not during the simulation itself.

5 Generalizing Forces

Objectives

- Construct the parent Force class `ZeroForce`.
- Implement the 3 derived Force classes.
- implement a function to combine an arbitrary amount of forces.

Shortcomings of Previous Implementation Our `Problem1Force` class implements a lot of different forces and constraints. The downside of this is that you cannot use a subset of forces and will have to add to an ever growing function every time you want a new force. It is better to create a class for each force. Luckily forces add so all we have to do is add the output of different force objects. To ensure that we can easily add a variable number of forces we first create a Parent force that all the other forces will derive from.

5.1 Force inheritance

Note that `Problem1Force` only has one attribute, the `operator()` method that takes in a `Node` and a `double`. To ensure that all force objects impelement this method we will use inheritance and polymorphism (lecture 7). Create a class called `ZeroForce` that implements this method as a `virtual` function that returns zero force.
Create the following classes that all inherit from `ZeroForce`.

- `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.

5.2 Combining the forces

If we pass all these forces to `symp_euler_step` individually we would have to change the signature every time we added/removed a force. instance, the combination of `GravityForce` and `MassSpringForce` should equal the result of `Problem1Force` (but see below).

Design and implement a functor called `CombinedForces` that is constructed with a vector of different forces. And applies all these forces with `operator()(Node n, double t)`.
Hint look at the polymorphism in lecture 7.

Testing Your Code 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.

6 Generalizing Constraints

Objectives

- Construct a Parent Constraint class.
- Implement the 3 derived Constraint classes.
- Implement a function to combine an arbitrary amount of forces.

Where Problem 5 generalized forces, in Problem 6 we generalize *constraints*. A constraint is a restriction on the degrees of the freedom in the equation. We will consider constraints of the form
$$C_i(x_i, t) = 0 \quad \text{and} \quad C_i(x_i, t) \le 0$$
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. E.g. keeping nodes fixed in Problem 1.

- Some constraints may apply to any node. E.g. defining an obstacle that no node of the graph may penetrate, or a moving obstacle that destroys parts of the graph.

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 doesnt fit well into the force concept, so we create a new **constraint** concept.

6.1 Constraint inheritance

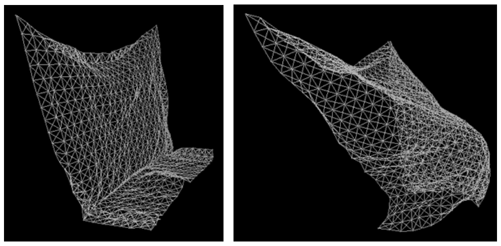
Similar to the force objects you will implement constraints through inheritance and polymorphism. Design constraints as functors (a.k.a. function objects) that take a `Graph` and a `double` for the time.
Create a parent constraint class `ZeroConstraint` with a virtual `operator()`.
Create the following classes that inherit from `ZeroConstraint`

- `PinConstraint` which keeps the nodes at (0,0,0) and (1,0,0) fixed.
- `PlaneConstraint` 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.
- `SphereConstraint`
  - 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 $\cdot$ is the inner (dot) product.

6.2 Combining Constraints

Similarly to the `CombinedForces` make a `CombinedConstraints` functor that takes in an arbitrary number of constraints and applies them to the graph. It is okay if the constraints are enforced sequentially.[2]

[2]The constraints should be applied after position update, before the forces are calculated. So the constraints are imposed between updating position (x^{n+1}) and force ($F(x^{n+1})$). That means, the constraints fixes x^{n+1} and v^n, but not v^{n+1}.



7 Node and Edge Remove

Objectives

7.1 Implementing the removal methods

- Implement `Node` and `Edge` removal.
- Test removal with `test_nodes.cpp` and `test_edges.cpp`.
- implement `TearConstraint`.

Being able to remove `Nodes` and `Edges` could let us model tears or holes in the fabric! Add the following methods to the `Graph clas` interface:

```
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 $\mathcal{O}(\texttt{num\_nodes}() + \texttt{num\_edges}())$ and `remove_node` can take at most $\mathcal{O}(\texttt{num\_nodes}())$, but hopefully a lot less. Removing a `Node` should remove all `Edges` that are incident to it. You may assume that the Graph is sparse, so that the maximum degree of a node is much less than the number of nodes.

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`, `iterators`) that are invalidated by these functions.

Hints

- Implement `remove_edge` first.
- Depending on your implementation the information about nodes and edges may be stored in several places. Go through your code first to list what should be updated when you remove a node/edge.
- When removing nodes first remove all the connected edges.
- The `test_nodes.cpp` and `test_edges.cpp` are there to help you debug, feel free to

change them as you see fit.

Before continuing, let's clarify some 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.
- `g.node(i).index() == i` for all `i` with $0 \le$ `i` $<$ `g.num_nodes()`.
- `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 $\mathcal{O}(1)$ operations unless otherwise specified.
- `remove_node` and `clear` may require $\mathcal{O}($`num_nodes`$)$ operations. All edge methods within the `Graph` must use *at most* $\mathcal{O}($`num_nodes() + num_edges()`$)$ operations, but hopefully less. All other methods require $\mathcal{O}(1)$ operations.

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.

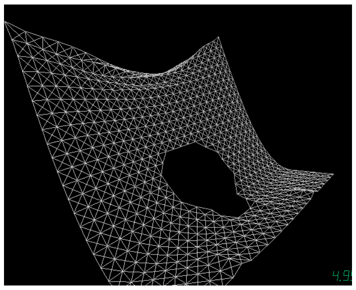
7.2 Using the removal methods

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 other `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:

```
    // 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);
```



8 Submission Instructions

Use a Git *tag* to mark the version of code you want to submit. Here's how:

```
$ git status                                 View files git is tracking
$ git add <files to track>                   Tells git which files to track
$ git commit -am "Describe last few edits"   Commit files
$ git tag -am "My HW2" hw2                   Tag this commit with tag ''hw2''
$ git push --tags origin master              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 origin master
```

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.

View your repository on GitHub to verify that all of the files and tags were pushed correctly.