CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Homework 0 – Due Friday, January 17th, 4:30 P.M. P.S.T.

---

1 Overview

There are two parts to the initial homework, **setting up your work environment** and **introducing you to the graph class**.

1.1 Goals

The first part is to introduce you to working on a remote server and getting you familiar with GitHub. This part of the assignment should not take you more than an hour. If you encounter any problems contact the TA's.

The second part of the assignment is to familiarize yourself with the Graph class we will be working with for the rest of the quarter. For a basic overview of graphs use this link. You need to understand the provided code before you start writting your own. **read the comments in the Graph class**

The most important part of this initial assignment is to learn how to read other people's code and build off of it. You will also be introduced to the **proxy design pattern**.

1.2 Tasks

- Install environment (short)
- Setup GitHub (short)
- Create `Node` Class (time intensive)
- Create `Edge` Class (very time intensive)

1.3 Tips

- Start early!
- Make sure you understand the starter code provided before you write your own code.
- Draw out the structure of the `Graph` class and it's subclasses.
- 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.

2 Configuring Your Workstation

A virtual machine creates a stand alone instance of an operating system on your computer we use this to ensure that everyone has the same setp which works with the provided starter code. he standard computing environment for CME212 is Ubuntu version 16.04. All of your work must compile and run on Ubuntu 16.04. You have several options here. You may either

1. Install an Ubuntu virtualbox, which allows you to run a Linux OS locally on your machine.
2. (**Mac or Linux only**) configure your workstation to support the software we plan to use.

2.1 Install an Ubuntu 16.04 VirtualBox

To obtain a clean, virtual Linux system, follow the steps below.

1. Download the VirtualBox binary for your operating system here. (version $\geq$ 5.1.12)
2. Download the CME212 virtual machine "appliance" file from Stanford Box (3GB).
3. Import the `cme212-ubuntu-vm.ova` file into VirtualBox.[1] The appliance contains an install of Ubuntu 16.04 with the compilers and libraries you will need for homeworks.
4. The default username is: `cme212-user`, and the default password is: `cme212-pass`.[2]

Here are some tips for setting up your VM that should make your life easier.

From the "Oracle VM VirtualBox Manager" menu, select the cme212-ubuntu-vm in the RHS panel, and then open the "Settings" menu.

- Under "General" > "Advanced" > "Shared clipboard" select Bidirectional. This will allow you to copy/paste content between the VM and your local computer.
- Under "System" > "Motherboard" > "Base Memory" you might want to consider increasing the memory. This could improve some of the latency issues.
- Under "Display" >"Screen" > "Video Memory", again consider increasing the memory to improve latency.

If you want to share a folder between your local computer and the VM:

```
"Shared Folder" >> + Folder icon
    Folder path: Local path to folder, e.g. /Users/jesswetstone/
    Folder name: Name of local folder, e.g. cme212-jhwetstone
```

[1] (On MacOS select **Import Appliance** from the **File** menu.)
[2] You may have to install `tkinter` manually.

```
    Select Auto-mount
    Mount point: Name of folder on VM, e.g. cme212-jhwetstone
    Select Make Permanent
Save settings and log on to the VM. In a terminal window, type:

<code>sudo usermod -aG vboxsf $USER</code>

Log off and log back on.  Under "Files" on the VM, you should have access
to the shared folder /media/sf_[Mount point]
```

If you would like to use the Sublime text editor on the VM: http://tipsonubuntu.com/2017/05/30/install-sublime-text-3-ubuntu-16-04-official-way/

Note: If you are able to run code locally, you should probably be OK to develop locally for most of the classwork this quarter. We recommend having a working VM configured for testing purposes and if for some reason later on in the course your code won't compile locally.

2.2 Configure your own Macintosh Workstation

If you're on a Mac, we've had success with the following instructions in the past.

1. Install Homebrew by following instructions at: http://brew.sh/
2. Install `xquartz` and `libsdl` with

   ```
   brew update
   brew cask install xquartz
   brew install sdl --with-x11
   ```

   Reboot your computer or at least fully log out and log in.

2.3 Configure your own Ubuntu Workstation

If you're already running an Ubuntu operating system, it's easy:

```
sudo apt-get install libsdl2-2.0
sudo apt-get install libsdl2-dev
sudo apt-get install libsfml-dev
```

3 GitHub Repository Creation

Objectives

- Create GitHub account (if you don't have one)
- Create CME212 GitHub repo.
- Setup Git on your computer.
- Pull starter code.

3.1 Create GitHub homework repository

GitHub provides a remote server to manage your work. It allows you to backup your work, create different versions for experimentation, allows the teaching team to collect all the code and more. If you are inexperienced with Git here are a couple of tutorials. CME212 has a GitHub organization (`https://github.com/cme212`) which maintains all student code repositories. Each student will be granted a separate private code repository. The only people that can access the repository are the student and course staff.

To create your repository please visit the following link and login with your GitHub account credentials: CME212 homework repository. The location of the created repository is: `https://github.com/cme212/cme212-[github_user]`. Here, `[github_user]` is your GitHub username. For example, Andreas' GitHub username is `asantucci` so his repository address is `https://github.com/cme212/cme212-asantucci`.

At this point it is a good idea to visit the webpage for your repository and look around. You will see a line "We recommend every repository include a README, LICENSE, and .gitignore." in your repository. Please create the README and .gitignore. The 'README.md' file contains a short description of the repository. The '.gitignore' file tells 'git' which files to ignore. For now .gitignore can be empty.

3.2 Configure `git` in your (virtual) machine

(Launch your CME212 virtual machine.) Open a terminal and instruct Git who you are:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Replace the name and email with your information. It is also a good idea to tell `git` which editor you want to use for commit messages. For new users, I recommend using `nano`:

```
$ git config --global core.editor nano
```

These commands store information in the user's `git` configuration file, located at `~/.gitconfig`. You can inspect the contents of the file with `cat`. Here is Andreas's:

```
$ cat ~/.gitconfig
[user]
    name = Andreas Santucci
    email = santucci@stanford.edu
[core]
    editor = vim
```

See: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup

3.3 Clone your repository

To clone your repo, open a terminal and enter the command:

```
$ git clone https://github.com/cme212/cme212-[github_user].git
```

Replace `[github_user]` with your GitHub username. The GitHub web interface provides a link for convenience.

It is useful, but not required, to generate SSH keys for use in communicating with GitHub. This will allow you to push and pull from GitHub without entering your username and password each time. Instructions to generate SSH keys are provided by GitHub.

3.4 Create `STUDENT` file

The CME212 grading tools will look at the `STUDENT` file in your homework directory to determine who you are. Create and open a text file named `STUDENT` at the top level of your homework directory.

For example, Andreas's `STUDENT` file has the contents:

```
[cme212-student]
name = Andreas Santucci
stanford_email = santucci@stanford.edu
stanford_id = 01234567
github_user = asantucci
```

Your `STUDENT` file must maintain the same header (`[cme212-student]`) and variable names (`name`, `stanford_email`, `stanford_id`, `github_user`). Note that these are all case-sensitive. You must replace Andreas' information with your own. The `stanford_id` must be 8 digits, so include the leading `0` if you have one. **It is very important that this information is correct. Without a valid file, it is likely your homeworks will not get graded.**

Now, commit your `STUDENT` file and push to GitHub.

3.5 Add the remote `CME212-2020` repository

Now, we will tell `git` about a remote repository where you can access the starter code. Use `cd` to navigate to your submission directory. Execute the following command to add `CME212-2020` as a remote repository:

```
$ git remote add CME212-2020 https://github.com/cme212/CME212-2020.git
```

Check that the remote was correctly added:

```
$ git remote -v
CME212-2020  https://github.com/cme212/CME212-2020.git (fetch)
CME212-2020  https://github.com/cme212/CME212-2020.git (push)
origin  https://github.com/cme212/cme212-nwh.git (fetch)
origin  https://github.com/cme212/cme212-nwh.git (push)
```

Note that `CME212-2020` and `origin` are simply labels for the remote repositories. You see `origin` in many commands, because it is the conventional name for the primary remote repository.

3.6 Pull the starter code

Pull the starter code with the following command:

```
$ git pull CME212-2020 master
```

This may cause one of two errors, here are the workarounds If this causes a `fatal: refusing to merge unrelated histories` error, then run the above with the flag `--allow-unrelated-histories`. Also, `git` may ask you for a commit message to merge the starter code into your repository. If you get a merge conflict error for `README.md`, simply call `git add README.md` followed by `git commit`, then retry the pull. Save the commit message and push the starter code to your private remote repo:

```
$ git push origin master
```

You can check on GitHub to see if the files have been properly pushed to your private remote repository. You are now ready to work on fun part of the assignment!

4 Introduction to Code Design

Objectives

- Familiarize yourself with the `Graph` class.
- Implement the `Node` class.
- Implement the `Edge` class.

4.1 Starter Code

- `Point.hpp` Contains the `Point` class and all it's method. A point is defined by 3 coordinates (x,y,z). All the standard operations for points (e.g. addition, scaling, norms, etc) have already been implemented. Make sure you understand the `Point` class before moving on.
- `SFML_Viewer.hpp` Provides the code to visualize your Graph.
- `viewer.cpp` This is the script you will run to actual create the image.

  - `data/XX.nodes` - The x, y, z coordinates of nodes
  - `data/XX.tets` - The a, b, c, d node numbers of tetrahedra. 0-indexed.

  Interacting with Viewer Once an SFML_Viewer window has been launched and `event_loop()` has been called, you may interact with it:
  - Left Drag (One Finger): Rotate
  - Right Drag (Two Finger): Pan
  - Scroll: Zoom
  - Keyboard 'c': Center View
  - Keyboard 'esc' or 'q': Close
- `Makefile` automates the compilation. automate the compilation procedure. To compile, type `$ make viewer`. The `Makefile` compiles the source code into the executable `viewer`. To delete all associated files, type `$ make clean`.
  For Ubuntu systems: Go ahead and edit the Makefile.inc, such that instead of attempting to compile with clang++, you instead choose g++. I.e. replace the macro-expansion with the simple triplet of characters "g++" (don't even use quotes).

4.2 Graph Design and Requirements

The only thing that defines a Graph is that it has nodes and edges. To get started with our Graph you will need to implement these two classes within the Graph class. You have already been provided with the *interfaces* for these classes, i.e. all the methods that you need to implement. **Read all the comments before you start!**

Here are some specifications on our `Graph` class:

- A user shouldn't be able to construct valid `Nodes`/`Edges` except through `Graph` methods.
- `Nodes` and `Edges` should be lightweight – we expect to have many of them and want the construction and copy to be fast – but still be extendable and potentially have lots of data associated with them. We will place a hard limit on the size of these objects:
  - `sizeof(Graph::Node)` $\leq$ 16 bytes
  - `sizeof(Graph::Edge)` $\leq$ 32 bytes

4.3 Nodes

First, we'll implement the `Nodes` in the `Graph`. We will provide the following public interface

Explaining our Class Nodes are defined by their position, so a node is created with `Point` objects that specify their positions. Nodes in a `Graph` are `indexed` by non-negative integers. The `add_node` function returns a `Graph::Node` object. We want this `Node`'s accessor function(s) to return the most up-to-date information about that `Node` anywhere and anytime we ask about it. Notice that if we provide a `Node` object to a user, and since this `Node` object has a method which returns its `index`, that we must have a way for the `Node` to "stay current" with the `Graph` its associated with; e.g. inserting a new `Node` in the graph *could* result in re-`indexing`.

Implementation Details and use of STL You should delegate all memory management to standard containers from the **C++ STL**. This has several advantages, the big one being that you don't need to worry about dangling pointers or double frees. It also introduces a challenge: how can a `Node` object that was exported to the user keep up to date with the contents of some STL container hidden within the `Graph`?

Proxy Design Patterns One solution is related to the proxy design pattern. The `Node` class doesn't need to contain the `Node`'s actual `Point` object. Instead, we store all the information in the `Graph` class and use the `Node` objects only as an access to the underlying data. The advantage of this implementation is that your code does not have to allocate a `Point` in memory every time you access a `Node`. This makes the script much more efficient. We've provided an example of the proxy pattern in `proxy_example.cpp`. The key is the private data members of the proxy class and how they are used.

Since each `Node` provides an index and we provide an accessor based on index, we'll impose one important condition on `Nodes`:

- For all `i` with $0 \leq$ `i` $<$ `graph.size()`, `graph.node(i).index() == i`

I.e. if we query the `i`th node and then request its index, it had certainly better be `i`.

Hint: Select appropriate STL containers from this list to hold the node's data and use a proxy pattern to access it.

4.4 Testing `Node`

If you have implemented the `Node` class properly you will be able to compile and run `viewer.cpp` with the following commands.

```
$ make viewer
$ ./viewer data/XX.nodes data/XX.tets
```

This should display all the nodes of the graph but not the edges.

4.5 Edges

`Edge`s define connections between two `Node`s. There are several ways to implement this relation. Look at how graphs are represented in general and make sure you understand the representation before implementing it. The Edge class has the following interface.

4.6 Testing `Edge`

When your `Edge`s are functional, you go to `viewer.cpp`, comment out `draw_graph_nodes(graph)` and uncomment `draw_graph(graph)`. Then recompile and rerun `viewer`. The execution and output should be

```
$ ./viewer data/XX.nodes data/XX.tets
A B
```

where `A =` the number of nodes in `XX.nodes` and `B =` the number of unique undirected edges in the graph, and an SFML_Viewer window displaying the object.

5 Submission Instructions

You will submit your work using GitHub. Common work flows involve the following commands:

```
# ... hack away at your code
$ git status        # see what files have been added/changed
$ git add <files>   # tell git to track new files
$ git rm <files>    # tell git to remove old files
$ git commit -am "description of changes"   # commit changes locally
# ... more work, more commits
$ git push origin master   # push changes to remote repository
```

5.1 Commit, tag, and push your work for grading

We will use a Git *tag* to mark the commit that you want graded. A Git *tag* is essentially a bookmark to a particular commit. After you have committed the version of the code you would like graded, create a `hw0` tag with the following command:

```
$ git tag -am "HW0 Submission" hw0
```

The quotes contain a tag message. In the above command `hw0` is the actual tag. You can push commits along with tags to the remote repo with:

```
$ git push --tags origin master
```

It is a good idea to check if the tags show up on GitHub. These appear in the "release" tab on the web interface.

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

```
$ git tag -d hw0                # delete the hw0 tag pointing to old commit
$ git tag -am "My HW0" hw0      # create hw0 tag pointing to current commit
$ git push --tags origin master # push changes to code and tags
```

We will use Git time-stamps on tags and the associated commits to check deadlines. Be careful with overwriting tags – you don't want to lose the submission tag.