CME212, Winter 2020
Advanced Software Development for Scientists and Engineers

Homework 3 – Due February 28th, 4:30 P.M. P.S.T.

---

1 Overview

1.1 Goals

In this HW we will use our `Graph` to represent a sparse matrix and solve a PDE. We will make a `GraphSymmetricMatrix` class using our `Graph`. This matrix class implements all standard matrix operations and will allow us to interface with two libraries: Boost and the Matrix Template Library (MTL4) to solve Poisson's equation using a conjugate gradient solver. Tying together graphs with linear algebra inches us closer to spectral graph theory.

1.2 Tasks

- Install Boost and MTL4. (short)
- Implement a couple of simple operations with MTL. (medium)
- Setup the Poisson system with the `GraphSymmetricMatrix`. (time intensive)
- Solve the system. (medium)
- Visualize the convergence of the system. (medium)

1.3 Tips

- MTL and Boost both have great documentation and tutorials, use these to understand how to use their functions.
- You should not have to change anything in `Graph.hpp`
- MTL requires a couple of type definitions. The code has been provided for you in this document.
- If you can't get your system to converge try a smaller system where you know the answer. Use the print method to check your matrix is correct.
- In this assignment you will build on top of existing libraries so make sure you understand them first.
- Start early!
- Make sure you understand the starter code provided before you write your own code.
- 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

- `mtl_test.cpp` Where you will implement the `IdentityMatrix` to get familiar with MTL.
- `poisson.cpp` Where you implement the `GraphSymmetricMatrix` and solve the system $Au = b$ using the MTL library.

2 Setup

Our helper code for Homework 3 extends the code from Homework 2. To retrieve it, follow these steps:

```
# Check the status of your git repository
$ git status
```

```
# Should say "on branch master"
# Otherwise, save a HW2 branch and checkout the master branch
$ git branch hw2
$ git checkout master

# Should also show no changes (all commits should already be made for HW2)
# Otherwise, commit your changes.
$ git commit -am "Stray HW2 changes"

# Load our changes into your repository
$ git fetch CME212-2020

# Apply our changes to your repository
$ git merge CME212-2020/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 CME212-2020/master` instead.

3 Install Boost and MTL4

Objectives

- Install Boost
- Setup MTL4
- Modify Makefile

3.1 Installing Boost

Boost is a collection of libraries that provides tools for linear algebra, image processing and much more. It is extremely powerful and will be useful for almost any task in C++. You can install it using the command,

```
# Install the boost library
$ sudo apt-get install libboost-dev
```

In general this command is preferable because it compiles certain components (that we will not be using) and also copies all of the header files to `/usr/include/`. This directory is, by default, included by the `g++` compiler to look for header files that you are importing with `#include`.

3.2 Setting Up MTL4

The Matrix Template Library (MTL4) is a linear algebra library. As long as your matrix class has a couple of methods defined it can use all the tools in the MTL4 library making it extremely useful for doing linear algebra. The library is header-only so no components need to be compiled. We can download these files from

- http://old.simunova.com/en/node/145.

and unpack them to access the source code.

3.3 Configuring `Makefile` to Specify Dependencies

Finally, we need to instruct the `g++` compiler to find any header files from MTL4 that we `#include` in our code. To do this, we add `-I/path/to/MTL4` (where `/path/to/MTL4` is where you have stored the source code) to the `INCLUDES` flag in the `Makefile`.

Once Boost and MTL are installed and included, you may add `mtl_test.cpp` to the list of executable in the `Makefile`. Use this file for the introductory problem below. We emphasize you must `#include` the relevant files, i.e.

```
1 #include <boost/numeric/mtl/mtl.hpp>
2 #include <boost/numeric/itl/itl.hpp>
```

THESE NEED TO BE AT THE TOP OF THE INCLUDE STATEMENTS IN YOUR CODE!!!!!
Else, you'll run into cryptic and difficult to interpret errors.

4 Intro MTL4

Objectives

- Implement the `IdentityMatrix` class.
- Add the interface with MTL4.
- Implement an outstream operator.
- Solve a conjugate gradient sytem with the `IdentityMatrix`.

In this first section, we will guide you through MTL's design and how to create "Matrix free" linear operator. In the following problems, we will modify your solutions to use your Graph class as a matrix to solve an interesting problem.

We will be following the "Matrix-free operations" guide linked here:

http://old.simunova.com/docs/mtl4/html/matrix__free.html

What are matrix-free operations and why are they attractive? Consider the the matrix-vector product

$$\mathbf{y} = \mathbf{I}\mathbf{x}$$

where $\mathbf{I}$ is the identity matrix. This is a trivial example, but how might we implement this operator and its action on a vector? Well, in `mtl_test.cpp`, write

```
struct IdentityMatrix {
  /** Compute the product of a vector with this identity matrix
   */
  template <typename Vector>
  Vector operator*(const Vector& x) const {
    return x;
  }
 private:
  // Empty!
};
```

In this (dumb) example, we don't need to store the values of the matrix and the implementation of its multiplication with a general vector is trivial. Slightly interestingly, we're *representing* an identity matrix with zero memory and its action on a vector with no mathematical operations!

Let's implement the identity matrix above so that it interfaces with MTL4. Then, we can construct identity matrices and pass them to Conjugate Gradient solvers to solve equations of the form

$$\mathbf{I}\mathbf{u} = \mathbf{b}$$

4.1 Interfacing with MTL

In order to interface with MTL4, any matrix (or more general linear operator) must define the following methods:

```
/** The number of elements in the matrix. */
inline std::size_t size(const IdentityMatrix& A)

/** The number of rows in the matrix. */
inline std::size_t num_rows(const IdentityMatrix& A)

/** The number of columns in the matrix. */
inline std::size_t num_cols(const IdentityMatrix& A)
```

Implement these methods for the `IdentityMatrix`.

In addition we need to add the following `typedef`s so MTL4 can get information about our linear operator.

```
/** Traits that MTL uses to determine properties of our IdentityMatrix. */
namespace mtl {
namespace ashape {

/** Define IdentityMatrix to be a non-scalar type. */
template<>
struct ashape_aux<IdentityMatrix> {
  typedef nonscal  type;
};
}  // end namespace ashape

/** IdentityMatrix implements the Collection concept
 * with value_type and size_type */
template<>
struct Collection<IdentityMatrix> {
  typedef double    value_type;
  typedef unsigned  size_type;
};
}  // end namespace mtl
```

Explanation: By default MTL assumes that any user-defined type is a scalar type. The first typedef tells MTL that this is not the case for the `IdentityMatrix`. The other `typedef`s should look more familiar to you and provide the `value_type` and `size_type` associated with an `IdentityMatrix`. Now we can use our `IdentityMatrix` with the MTL!

4.2 A faster `operator*`

In writing the `operator*` method above, you performed an copy into a new vector when computing the result of $\mathbf{y} = \mathbf{I}\mathbf{x}$. That is, $\mathbf{y}$ is simply a copy of $\mathbf{x}$. This is fine, but consider what happens when we actually want to compute

$$\mathbf{y} \mathrel{+}= \mathbf{I}\mathbf{x}$$

With the `operator*` definition above, writing this code would translate to:

```
auto temp = I * x;
y += temp;
```

which requires extra memory and two extra runs through the data.

Ideally, we could just add each x_i to each y_i and call it a day. The MTL is clever and provides a way to avoid these extra copies. Rather than just take in the input $\mathbf{x}$ vector, we will take in the input vector $\mathbf{x}$, the output vector $\mathbf{y}$ and an assignment operator. The assignment

operator will be one of various operations we would like to perform such as $+=$, $-=$, or $=$. This is very useful for iterative methods such as conjugate gradient descent.

First, we need to write a `mult` method that is very similar to our `operator*`. The function signature is provided below,

```
class IdentityMatrix
  /** Helper function to perform multiplication. Allows for delayed
   *   evaluation of results.
   * Assign::apply(a, b) resolves to an assignment operation such as
   *   a += b, a -= b, or a = b.
   * @pre @a size(v) == size(w) */
  template <typename VectorIn, typename VectorOut, typename Assign>
  void mult(const VectorIn& v, VectorOut& w, Assign) const
```

It would be messy to have to write `I.mult(x,y,mtl::assign::plus_sum())` to compute $\mathbf{y} += \mathbf{I}\mathbf{x}$. Instead, we can rewrite our `operator*` to construct a lazy-evaluator which will eventually call our `mult` above with the appropriate assignment operator. This is done via,

```
class IdentityMatrix
  /** Matvec forwards to MTL's lazy mat_cvec_multiplier operator */
  template <typename Vector>
  mtl::vec::mat_cvec_multiplier<IdentityMatrix, Vector>
  operator*(const Vector& v) const {
    return {*this, v};
  }
```

Now we can write code such as `y += I*x` and the correct computation will be performed without any additional overhead! A more detailed explanation of how this works can be found at http://old.simunova.com/docs/mtl4/html/matrix__free.html.

4.3 Conjugate Gradient

A common problem that arises in linear algebra is to solve the equation,

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

where $\mathbf{A}$ is a known matrix, $\mathbf{b}$ is a known vector, and $\mathbf{x}$ is an unknown vector.

When the matrix is symmetric positive definite, this system of equations can be solved using the conjugate gradient method. A matrix is considered positive definite if for any vector $\mathbf{x} \neq 0$,

$$\mathbf{x}^T\mathbf{A}\mathbf{x} > 0.$$

If you have taken CME200 or CME302 you will already be familiar with this method. A good explanation of this method is provided on https://en.wikipedia.org/wiki/Conjugate_gradient.

The `IdentityMatrix` defined above is clearly symmetric and is positive definite because for any $x \neq 0$

$$\mathbf{x}^T\mathbf{I}\mathbf{x} = \mathbf{x}^T\mathbf{x} = ||\mathbf{x}|| = \sqrt{\sum_{i=1}^{n} x_i^2} > 0.$$

This allows us to use the `IdentityMatrix` we just constructed in the conjugate gradient solver in MTL.

The conjugate gradient solver uses only matrix-vector multiplications in order to solve the system of equations, so we're ready to go. Complete `mtl_test.cpp` to solve the system of equations,

$$\mathbf{I}\mathbf{x} = \mathbf{b}$$

using MTL. A tutorial for using predefined linear solvers in the MTL can be found at http://old.simunova.com/docs/mtl4/html/using__solvers.html. Obviously, we can "solve" huge equations with our easy little identity matrix.

5 Poisson with an Implicit Matrix

Objectives

- Implement `remove_box`
- Create the `GraphSymmetricMatrix` class.
- Implement the boundary conditions.
- Create the vector b of our system $Ax = b$.
- Solve the system with MTL's conjugate gradient solver.
- Check the convergence using `it::cyclic_iteration`

One equation that arises frequently in many applications is Poisson's equation given by

$$\begin{aligned} \Delta u(\mathbf{x}) &= f(\mathbf{x}) \qquad & \mathbf{x} \in \Omega \\ u(\mathbf{x}) &= g(\mathbf{x}) \qquad & \mathbf{x} \in \partial\Omega \end{aligned} \tag{1}$$

where Ω is the two-dimensional domain we wish to solve on, $\partial\Omega$ is the boundary of the domain, and Δ is the Laplacian operator given by

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

Here $f(\mathbf{x})$ and $g(\mathbf{x})$ can be any arbitrary function.

In the last homework we saw how to approximate derivatives in time using the finite difference method. Now we want to approximate derivatives in space. The Laplacian operator Δ can be approximated for a structured graph by the discrete Laplacian matrix

$$L_{ij} = \begin{cases} -\deg(n_i) & \text{if } i = j, \\ 1 & \text{if node } i \text{ and node } j \text{ share an edge,} \\ 0 & \text{otherwise} \end{cases}$$

where $\deg(n_i)$ is degree of node i in the graph.

To solve (1) numerically, we discretize the system and arrive at the linear system of equations

$$\mathbf{Au} = \mathbf{b}$$

where

$$A_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and node } i \text{ is on a boundary,} \\ 0 & \text{if } i \neq j \text{ and node } i \text{ or node } j \text{ is on a boundary,} \\ L_{ij} & \text{otherwise} \end{cases}$$

and

$$b_i = \begin{cases} g(x_i) & \text{if node } i \text{ is on a boundary} \\ h^2 f(x_i) - \sum_j g(x_j) & \text{for all adjacent nodes } j \text{ that are on a boundary} \end{cases}$$

where x_i is the position of node i and h is the length of the edges in the graph (Note that for the graph defined in `poisson.cpp` and used with `gridX.nodes/tets`, the edges all have the same length).

Graph as a Sparse Matrix

In the first problem of this homework, we saw that the representation of matrices clearly doesn't have to match the abstract value of the matrices. In this section you'll be implementing a general sparse matrix that interfaces with MTL4, but the representation will be your Graph class!

Sparse matrices are nice because they allow us to avoid storing a huge amount of zeros. In general, a matrix of dimension n requires $\mathcal{O}(n^2)$ storage and time for matrix-vector multiplication. However, the storage and operation count is really only proportional to the number of non-zeros in the matrix!

Thus, we will be implementing our matrix-vector multiplications so that

$$y_i = \sum_{j:A_{ij}\neq 0} A_{ij}\, u_j$$

is efficient.

Above, we have defined a numerical formulation of the Poisson problem expressed naturally in terms of a graph, its nodes, and the nodes' adjacent nodes. Although we could manually define and construct a sparse matrix $\mathbf{A}$ and fill it with appropriate values computed from the Graph, we already have all the information we need in Graph to compute this product efficiently!

Implicit Matrix – GraphSymmetricMatrix

In the file `poisson.cpp`:

First, complete the `remove_box` function to be able to create holes in our Graph.

With the Graph constructed, define a `GraphSymmetricMatrix` class that wraps a Graph and, without copying the graph or explicitly storing elements of the matrix, implements an MTL-compatible matrix that abstractly represents the matrix $\mathbf{A}$ above and its action on a vector $\mathbf{x}$.

Hint: You may want to tag boundary nodes in some way so your matrix can efficiently detect them.

Now, using the MTL's conjugate gradient solver, solve the Poisson equation with forcing function

$$f(\mathbf{x}) = 5\cos(\|\mathbf{x}\|_1)$$

where $\|\cdot\|_1$ is the L1-norm (see `Point.hpp`) and boundary conditions

$$g(\mathbf{x}) = \begin{cases} 0 & \text{if } \|\mathbf{x}\|_\infty == 1 \\ -0.2 & \text{if } \|\mathbf{x} - (\pm 0.6, \pm 0.6, 0)\|_\infty < 0.2 \\ 1 & \text{if } \mathbf{x} \text{ is in the bounding box defined by } (-0.6, -0.2, -1) \text{ and } (0.6, 0.2, 1) \end{cases}$$

where $\|\cdot\|_\infty$ is the infinity-norm (see `Point.hpp`). Any node in the domain of $g(\mathbf{x})$ is considered a boundary node.

5.1 Verification

We want to make sure the solution is converging. See

http://old.simunova.com/docs/mtl4/html/using__solvers.html

for documentation on using `*_iteration` objects to provide output during the solve.

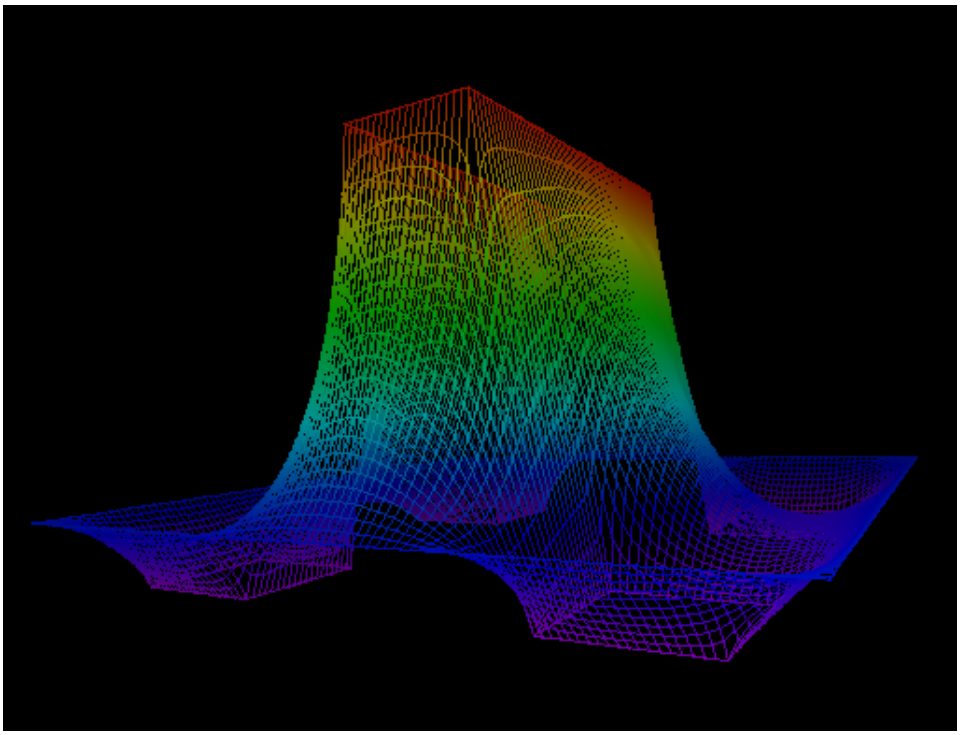
Construct and pass a `itl::cyclic_iteration` to check the convergence of the Poisson system. This can be used to output the residual every c iterations. Do this with $c = 50$ and verify that the relative residual decreases below a tolerance of 10^{-10}. Make sure you have a high enough max iteration number so that it reaches a valid solution.

6 Viewing the solution

Objectives

- Define the color functor.
- Define the NodePosition functor.
- Implement the `visual_iteration` class.
- Verify the convergence with the visualization.

Solving the equation above using our graph class is good, but we would like to see what the solution looks like using the SFML_Viewer. In order to create the plot of the solution shown above we have to override the position and color functors for SFML_Viewer.



In HW1 we created a custom color functor that takes a `Node` and returns a `CME212::Color`. Similarly, in this problem we can color Nodes based on the solution value $\mathbf{u}$. Design a `NodeColor` functor that displays your solution in an interesting way (Note that it doesn't have to match the image provided).

More interestingly, we also want to change how we view the positions of each node. Specifically, we would like to plot each node with the position

$$n_i : (x_i, y_i, u_i)$$

where x_i and y_i are the x and y-components of the node's position, and u_i is the scalar solution to Poisson's problem at node i.

In `SFML_Viewer`, there is another version of `add_nodes` which accepts a functor that returns the position associated with each node. Naturally, this defaults to a functor which simply returns `node.position()`. Write a NodePosition functor that changes how nodes are plotted so the solution can be viewed on the z-axis. See `SFML_Viewer.hpp` for documentation and the default NodePosition functor.

6.1 Visualization at each Iteration

Using a `itl::cyclic_iteration` to periodically output the residual is a good way to keep track of convergence. On the other hand, it would be even nicer to plot the current solution at every iteration of cg so we can *watch* it converge!

This can be done by extending the iteration callback `cyclic_iteration` to plot data in addition to outputting the residual periodically. Define your own iteration callback called `visual_iteration` that inherits from `cyclic_iteration`. This class will use the SFML_Viewer, the Graph, and the current solution to plot the solution periodically and defers to the `cyclic_iteration` for printing residuals.

For reference, see how the `cyclic_iteration` class

http://old.simunova.com/docs/mtl4/html/cyclic__iteration_8hpp_source.html

extends and overrides the `basic_iteration` class

http://old.simunova.com/docs/mtl4/html/basic__iteration_8hpp_source.html

Note that `itl::cyclic_iteration` inherits member functions from `itl::basic_iteration` and overloads methods to print data out periodically. Still, it calls member functions of the `super` class when appropriate. Similarly, your `visual_iteration` may inherit from `itl::cyclic_iteration` and will likely need to overload the member functions

```
1 bool finished()
2
3 template <typename T>
4 bool finished(const T& r)
```

Your `visual_iteration` class should update the SFML_Viewer with a visualization of the solution at every iteration in the cg solver as well as output the residual.

7 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 HW3" hw3                    Tag this commit with tag ‘‘hw3’’
$ 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 hw3
$ git tag -am "My HW3" hw3
$ 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.