Homework 0 - Eigen Tutorial

Handout date: January 7, 2021.
Submission deadline: January 14, 2021, 11:59pm.

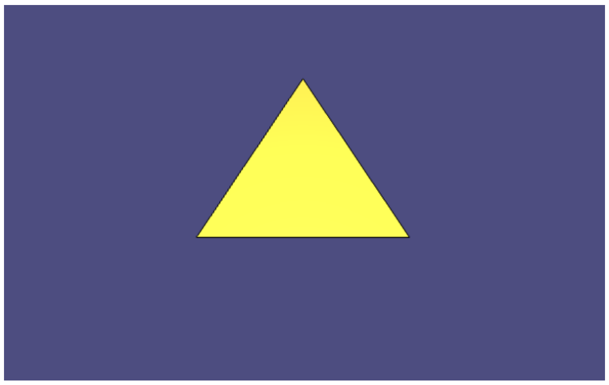

Figure 1: Expected view once you've completed the assignment.

This ungraded exercise will ensure you have a working C++ compiler along with all the libraries needed for the code we will write in this course. It will also introduce you to some of the features of `Eigen`, a powerful linear algebra library for C++.

1 Getting Started

1.1 Install the required tools

We will use the source code management tool `git` to download `libigl`, a geometry processing library that our assignment codes will use extensively. Then we will use `cmake` and a C++ compiler to build everything.

On Linux, all necessary tools can be installed straight from your package manager. E.g., on Ubuntu:

```
sudo apt install git cmake g++ ninja-build
```

On macOS, `git` should come pre-installed, and you can get a C++ compiler by running:

```
xcode-select --install
```

Then you can install `cmake` and `ninja` from their websites. However, I strongly recommend using a package manager like MacPorts or Homebrew. With MacPorts, you can do:

```
sudo port install cmake ninja
```

On Windows, you can install Visual Studio Community Edition and then install the “Desktop Development with C++” and “Linux Development with C++” tools within it by choosing `Tools->Get Tools and Features...` (these will give you a C++ compiler and `cmake`, respectively).

1.2 Download and build the code.

Make a directory to hold all of your coursework. I’ll refer to this as `DGP_HW` in the following. Clone a fresh copy of `libigl` here using git; on Linux/macOS this can be done by:

```
cd DGP_HW
git clone https://github.com/libigl/libigl.git
```

or you can use the code download links on GitHub.

Next, download the assignment zip file from Canvas and unzip it as a sub-directory of `DGP_HW`. Now, try building and running it. On Linux/macOS:

```
cd DGP_HW/hw0
mkdir build
cd build
cmake .. -GNinja # Generate the Ninja build file
ninja            # Compile the code
./hw0            # Launch the compiled binary
```

Using Visual Studio on Windows, choose `File->Open->CMake...` and select `DGP_HW/hw0/CMakeLists.txt`. This should run CMake and generate a VS project. Now you should be able to build and run the code by telling VS to run the `hw0` target.

If everything goes well, some output should be printed to the terminal, and a window should pop up displaying a yellow tetrahedron.

2 Meshes with libigl and Eigen

For most of the assignments in this class, we will not need a particularly fancy data structure for working with triangulated surfaces. We therefore will generally default to the simple “indexed face set” representation that `libigl` uses throughout. As we’ll soon see in class, this format consists of two arrays: a vertex array holding the 3D positions of all points, and a face array holding the indices of the vertices making up each triangle.

The convention of `libigl` is to represent these by two `Eigen::Matrix` objects called `V` and `F` respectively:

```
Eigen::MatrixX3d V; // |V| x 3 array of vertex coordinates
Eigen::MatrixX3i F; // |F| x 3 array of corner indices
```

Notice from the sizes that the coordinates of vertex i are stored in the i^{th} *row* of `V`. Likewise, the corner indices of face i are in the i^{th} *row* of `F`.

The `Eigen` types specified here are convenient aliases for two special instantiations of its general matrix class template. For instance, `Eigen::MatrixX3d` really means: `Eigen::Matrix<double, Eigen::Dynamic, 3>`, a matrix of double-precision floating point numbers with three columns and a resizeable number of rows. The `Eigen::MatrixX3i` type is the same, except its entries are of type `int`, not `double`. For more information on `Eigen` types, please consult the guide.

Dynamically-sized matrices like these start empty by default and can be resized using, e.g., `V.resize(1, 3)`. After resizing, the matrix will be filled with uninitialized data. If you wanted it filled with zeros you could have done `V.setZero(1, 3)`. Or if you want to fill it with custom values after resizing, you can use the comma initializer:

```
V << 1, 2, 3;
```

The provided code hard-codes an input mesh using this comma initializer syntax. Also, notice that it passed the appropriate sizes to `V` and `F`'s constructors to avoid the additional call to `resize`.

3 Basic linear algebra operations

This assignment asks you to perform some simple tasks on the hard-coded tetrahedron mesh to gain some familiarity with Eigen and discover its power. The tetrahedron is a regular tetrahedron as pictured here, but it has been rotated and translated by some unknown amount.

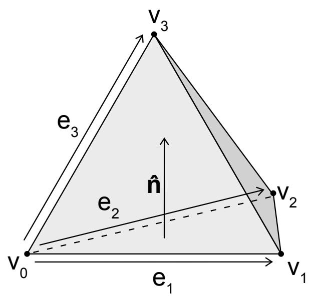

Figure 2: A regular tetrahedron annotated with some quantities needed below.

3.1 Compute the three outgoing edge vectors incident v_0.

Declare three objects of type `Eigen::Vector3d` called `e1`, `e2`, and `e3` and assign the appropriate vertex coordinate differences to them according to Figure 2. Hint: `V.row(i)` will access row `i` as a row vector. Also, Eigen overloads all arithmetic operators like `+`, `-`, and `*` to perform the appropriate matrix/vector operations.

Eigen generally complains if you try to assign one matrix to another of a different shape, but it will let you assign row vector expressions to a column vector of the same size. So, e.g., the line `Eigen::Vector3d a = V.row(1);` will compile and run as expected.

3.2 Compute the volume of the tetrahedron.

Next, you'll compute the tetrahedron's volume in two different ways. First, use the formula:

$$\mathrm{Vol} = \frac{1}{6}(\mathbf{e}_1 \times \mathbf{e}_2) \cdot \mathbf{e}_3.$$

This is easy to implement using `Eigen`'s `cross` and `dot` methods: given vectors `a` and `b`, you can call `a.cross(b)` and `a.dot(b)` to compute `a` × `b` and `a` · `b`, respectively.

Next, compute the volume using the equivalent expression:

$$\mathrm{Vol} = \frac{1}{6}\det(E) \quad \text{where } E = \left[\ \mathbf{e}_1 \ \middle|\ \mathbf{e}_2 \ \middle|\ \mathbf{e}_3\ \right].$$

Declare `E` as an `Eigen::Matrix3d`, and then you can actually use the comma initializer syntax to fill it with a sequence of column vectors: `E << e1, e2, e3;` (the comma operator intelligently fills out block matrices in order from left-to-right and top-to-bottom). Hint: `Eigen::Matrix3d` provides a `determinant` method.

If you do this correctly, you should get a volume of `1.0` with both approaches.

3.3 Compute the translation that centers the tetrahedron at the origin.

Construct a vector `t` that is the offset which must be added to each vertex to move the tet's barycenter to the origin. The barycenter is just the average of its vertex positions.

While this is easy to do with a `for` loop, you should get used to using Eigen's more advanced operations that let you avoid loops and generate efficient vectorized code. In this case, you can use a partial reduction `V.colwise().mean()` to average all vertices.

3.4 Compute a rotation to orient the tetrahedron.

Construct a rotation matrix `R` that, when applied to the points in `V`, orients edge vector $\mathbf{e}_1$ parallel to the +X axis and orients face 0 (consisting of vertices 0, 1, and 2) parallel to the XZ plane, making its unit normal vector $\hat{\mathbf{n}}$ (pictured in Figure 2) point in the +Y (not -Y) direction.

The easiest way to do this is to build a right-handed orthonormal frame consisting of unit vectors $\hat{\mathbf{e}}_1 := \frac{\mathbf{e}_1}{\|\mathbf{e}_1\|}$, $\hat{\mathbf{n}}$, and $\hat{\mathbf{e}}_1 \times \hat{\mathbf{n}}$ in this order, where the unit normal $\hat{\mathbf{n}}$ can be found using the the cross product of $\mathbf{e}_1$ and $\mathbf{e}_2$. You will probably find Eigen's `normalized` method convenient here. If you place these vectors in the columns of a 3×3 matrix, then the rotation you want is the inverse of this matrix. You can use the `inverse` method to compute it or, since this matrix is orthonormal, just the `transpose` method.

3.5 Apply the transformation $\mathbf{x} \mapsto R(\mathbf{x} + \mathbf{t})$.

Using `R` and `t` computed above, transform the tetrahedron so that it looks like in Figure 1. You will need to apply the transformation $\mathbf{x} \mapsto R(\mathbf{x} + \mathbf{t})$ to every vertex in `V`. The most straightforward way to do this with a `for` loop over row indices from `0` to `V.rows() - 1`, so try this first. You'll want to be careful about row and column vectors; you'll need to transpose a row of `V` before you can add `t` to it or apply `R`.

But you can actually do this operation in a single line using Eigen expressions (which again is likely to produce faster vectorized code than your for loop). The trick here is to use another partial reduction; it turns out that `V.rowwise() + t.transpose()` creates a matrix expression the same size as `V` that has `t` added to each of its rows. Then you can apply `R` to this expression. The other catch is that `R * V` does not apply `R` to each point since points are stored along `V`'s rows instead of its columns. You can, however, multiply the *transpose* of R on the *right* of `V`. Please convince yourself that this is the correct operation.

4 Gotchas

There are a few surprising pitfalls that you should be aware of about Eigen—some of which will produce errors at runtime, but others will just give undefined behavior! Here are the two I think you're most likely to encounter.

4.1 Inferring weird types

The first is for those of you who like using modern C++'s `auto` keyword. The `auto` keyword lets you save some typing by making the compiler deduce a variable's type for you. This is especially great when writing generic code using templates. However, using `auto` will quickly unveil the secret that expressions in Eigen are not exactly what they seem.

Eigen uses an advanced C++ technique called expression templates to generate efficient code (and avoid creating temporary copies to hold the result of each sub-operation in an expression). But this means that the following snippet:

```
Eigen::Vector3d a(0, 1, 2), b(1, 1, 1);
auto c = a + b; // Uh-oh!
```

declares a variable `c` that is actually *not* of type `Eigen::Vector3d` but rather a magical expression template type that describes how to compute entries of the result `a + b` on demand. The catch is, an expression template is only guaranteed to be evaluable within the C++ statement (line) that generated them. This particular example is fairly innocuous and might actually "work" depending how you use `c`. But if `a` and `b` get destroyed or changed before you read entries of `c`, then you won't get the result you intended. And forget trying to assign new values to `c`! (Thankfully, assignment triggers compilation error instead of doing something even stranger).

The work-around is to run the `eval` method on the expression template:

```
auto c = (a + b).eval(); // Now `c` is deduced as an Eigen::Vector3d
```

The `eval` method returns the true result of the expression template (in this case a `Eigen::Vector3d`), so `c` will be deduced properly.

4.2 Aliasing

The second is another consequence of Eigen's expression templates: aliasing. Fortunately, Eigen is *usually* smart enough to detect this for us at runtime, so the program will crash with an error instead of giving more subtle bugs. You may actually have encountered this depending on how you implemented Section 3.4. Suppose you first built the orthonormal frame in the columns of `R` and then want to transpose it in place:

```
R = R.transpose(); // Oops!
```

This will actually crash with an assertion at runtime:

```
$ ./hw0
Assertion failed: ((!check_transpose_aliasing_run_time_selector...
```

What's happening here is that `R.transpose()` is again just an expression template that references the original entries of `R` (swapping the accessed row and column indices). This poses a problem when updating `R` in scanline order (left-right top-down) to implement the assignment: once Eigen reaches `R`'s lower triangle, the expression template would read the already-overwritten entries in the upper triangle, so the lower triangle effectively doesn't change! To avoid this bug, you need to use `R.transpose().eval()` to evaluate the transpose in a temporary object or use the special `transposeInPlace()` method.

5 What to turn in.

Please submit a `.zip` or `.tgz` archive containing a text file with the console output of `hw0`, a screenshot of the program window, and your modified `main.cc` file.