CGGS Coursework 0: Geometry and Simulation with Eigen and PolyScope

Self-learning coursework

Errata

Nothing yet

Introduction

This is s self-learning coursework, which is neither submitted nor graded (however you can get support for it in the forum and the tutorials), whose purpose is to get acquainted with working on geometric and simulation tasks in `C++` with Eigen and PolyScope. Namely, how to use Eigen to work out computational linear algebra, basic geometry processing, basic physical simulation, and visualization using the 3D visualization package PolyScope. You will implement short functions that cover much of the functionality you need for the more specialized coursework practicals in the rest of the course. The learning objectives are:

- Get acquainted with the Eigen and PolyScope software packages and coding in `C++`, emphasizing geometric operations (no sophisticated object-oriented programming, just doing math).
- Experiment with geometry and time-based animation using PolyScope.
- Understand how geometry representation and visualization work.
- Experiment with solving linear systems.
- Implement basic geometric operations: computing normals and triangle areas.
- Implement a basic simulation of a hopping ball.
- Work with sparse linear systems.

This practical is long, as it explains things slowly, diversely, and comprehensively; feel free to solve it in parts throughout the course rather than in one sitting (if even possible). The point is introducing most computational techniques in this course but with simpler problems.

Installation basics

`C++` is a compiled programming language. This means you write a code file, and then need a compiler to create a runnable executable. There are popular IDEs for it. For instance, Visual Studio (mostly Windows), and XCode (macOS). You may use any operating system, IDE, or compiler as you wish. We are providing the project in CMake form to facilitate this. CMake is a multiplatform scripting language that can generate makefiles for any (reasonable) platform. Some even allow you to load a CMake file directly and generate the project files themselves (for instance CLion). Frankly, CMake is quite horrible to design, but you don't have to worry about it at all; follow this short procedure:

Obtain the project files inside a folder of your choice by writing the following in a terminal (you should have `git` installed):

```
git clone --recursive https://github.com/avaxman/CGGS-CW0.git
```

To generate the project files (for Linux or macOS) and build, enter in a terminal:

```
mkdir build
cd build
cmake ..
make
```

Otherwise, you can write:

```
cmake .. -G <IDE_name>
```

for selected installed IDEs, for instance, `Xcode`. In Windows, you need to use `cmake-gui`. Pressing twice configure and then generate will generate a Visual Studio solution in which you can work. The active solution should be `CourseWork0`. Note: it only seems to work in 64-bit mode. The 32-bit mode might give alignment errors.

The project solution will already link Eigen and PolyScope. Throughout the course, you are advised to consult the Eigen tutorial (`https://eigen.tuxfamily.org/dox/GettingStarted.html`) and the PolyScope tutorial (`https://polyscope.run/`) to understand their (rather intuitive) functionality, when needed.

The coursework

The main practical directory comprises two subfolders: `code`, where there are the main code files, including the ones you need to update, and `data`, containing the meshes and ground-truth results for grading. Your main project solution (or makefile) will have subprojects, where subprojects of the form `SectionX` correspond with "Section X" in this document.

You are also provided with automatic "graders" that will check your functions and provide feedback. There are other subprojects in the name `GradingX`, testing the corresponding Section X. The feedback for this practical will test your results

against a reference solution and say if it is correct; in later practicals, it will also provide a grade.

The practical text will indicate where you should be completing the portions of your code, and the code will document these places accordingly. In this practical (and likely in all of them), you will mostly need to complete functions in the `include` subfolder.

The practicals are typically done in the form of implementing functions, where you receive an envelope environment that executes your functions and helps you visualize the result. The envelope environment and the grader are distinct—the grader directly calls the functions you wrote with different input parameters and meshes, which are often more elaborate, whereas the envelope script helps you develop and debug your functions. As such, in future coursework, you only submit the implemented function files and are encouraged to alter the envelope script to be able to assess and debug your functions effectively.

1 Computational Linear Algebra

In this part of the practical, we will experiment with solving dense linear systems, rank-deficient systems, and consequent least squares solutions.

1.1 Solving simple linear systems

The entire code should be written in the `main()` function of the corresponding subproject `Section11`.

Write code that inputs the following matrix (also left-hand side, or *lhs*) and right-hand side (abbreviated: *rhs*):

$$A = \begin{pmatrix} -12 & -10.8 & -13.4 \\ -18.6 & -12.1 & -19.6 \\ -15.8 & -10.4 & -11.5 \end{pmatrix}, \quad b = \begin{pmatrix} -0.4 \\ -0.6 \\ -1.4 \end{pmatrix}$$

The matrix should be a `Matrix3d` object, and the vector a `Vector3d` object. Next, solve the linear system

$$Ax = b$$

using Eigen's `Eigen::FullPivLU` LU decomposition. Verify that you get the following result approximately (print with `std::cout`):

```
x = [ 0.1822845  0.0476237 -0.171773 ]
```

The matrix you input is *full-rank*. Verify this by using the `rank()` function of the LU solver. That means that the solution is unique and exact. Verify that by writing code to compute the solution error as $max(|Ax - b|)$, which should be a very small number (To the magnitude of at most 10^{-14}). Note that $|Ax - b|$ is a vector in the size of b; the `.cwiseAbs().maxCoeff()` functions of the vector type produce its maximal absolute. You should be getting something like:

```
Rank of A:  3
Solution Error:  3.33067e-16
```

1.2 Exploring rank-deficient systems

We next explore linear systems in which the left-hand side is rank-deficient. As you learned in class, such systems can either have infinite or no solutions, depending on the right-hand side. First, write code to artificially construct a rank-deficient matrix C as follows:

$$C([1,2],:) = \begin{pmatrix} -2.8 & -1.5 & -3.6 & -1.7 \\ -1.85 & -1.2 & -1.61 & -4.24 \end{pmatrix}$$
$$C(3,:) = 0.4 \cdot C(1,:) + 0.23 \cdot C(2,:)$$
$$C(4,:) = -0.5 \cdot C(1,:) + 0.2 \cdot C(3,:)$$

The notation "$C(i,:)$" means "the entire row indexed i of matrix C" (resp. $C(:,j)$ for column j). You should use the `.row()` command to access entire rows as row vectors. Verify this matrix has rank 2 (again with LU decomposition), as only the first two rows are linearly independent. **Important:** when writing matrices in math mode, like $C(1,2)$, we use 1-indexing as is the way in math. When writing matrices in Eigen code, we code `C(0,1)`, using the `C++` 0-indexing.

We next construct a right-hand side that is compatible (in the augmented-matrix sense; verify!) with our linear system:

$$d([1,2]) = \begin{pmatrix} 5; -6 \end{pmatrix}$$
$$d(3) = 0.4 \cdot d(1) + 0.23 \cdot d(2)$$
$$d(4) = -0.5 \cdot d(1) + 0.2 \cdot d(3)$$

If you try to solve this system with LU decomposition, you will likely get an answer which will be correct, but not unique. Depending on your system, the decomposition might fire some warning. Check the norm of the resulting `x`. Nevertheless, this is not going to be, in general, the minimum 2-norm solution for the rank-deficient C (without any null space components). To get this, you should solve with `JacobiSVD<Matrix4d, ComputeThinU | ComputeThinV>`, rather than LU, which effectively computes the pseudoinverse C^+, and where eventually $x_0 = C^+ \cdot d$. You should be getting approximately:

```
x0: -0.841157 -0.245536 -1.83592 2.54873
norm of x0: 3.26105
Solution error: 4.44089e-15
```

To verify this is the minimum 2-norm solution, use the `null` function to obtain a basis `nullC` for the null space $ker(C)$ (in the columns of the result), and check that:

```
x0*nullC: 1.11022e-16 2.22045e-16
```

You got two vectors exactly in the null space, coded in a matrix sized 4×2. Think why! Note that the base vectors of the null space don't have to be the

same across different instances or computers; they just have to span the same space.

Next, experiment with variating the minimum 2-norm solution by adding null space components, and seeing it doesn't change the solution error (and thus, span the solution space). Try to compute and print $x = x_0 + null(C) \cdot \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ for the following α and β: $(0.5, 0.5)$, $(-0.3, 10.2)$, and $(-3.981, 6.1)$, and demonstrate that the error $max(|C \cdot x - d|)$ is always negligible for any obtained solution.

Finally, we try an incompatible right-hand side to witness that we cannot get any solution. Code the following: $d(4) = 0.3945$, and try to solve for $C \cdot x = d$ in the least squares formulation. In this small and dense matrix case, you can use the SVD without change. You should however get a solution with a considerable $max(|C \cdot x - d|)$ error. Nevertheless, print the result of $|C^T Cx - C^T d|$ and see the effect. You should get for both something like (the numbers are not guaranteed exactly):

```
C*xe - d error:  2.7705
C.transpose()*C*xe - C.transpose()*d error: 3.90799e-14
```

So we know it is, indeed, a least-squares solution which is not a solution to the original system that became overdetermined.

2 Basic Mesh Processing

We next experiment with computing and visualizing basic quantities on a triangle mesh, namely normals and triangle areas. Here we begin using PolyScope (already preinstalled and linked). PolyScope (`https://polyscope.run/`) is a simple-to-use visualizer for geometry and simulation, which is highly customizable. Both envelope scripts already contain loading meshes and registering them to PolyScope visualization, so you can immediately play with it without writing any new code. Explore all options within the GUI. Especially note:

- You can select vertices and faces
- You can show or hide mesh edges
- You can alter the type of shading

You should get acquainted with PolyScope visualization model, which is essentially quite simple: you register geometric entities, like surface mesh, point cloud, or curve network (a graph of vertices and edges without faces), and can attribute scalar or vector quantities to them that are visualized using 3D vectors or color-coding. We next experiment with both.

2.1 Areas and normals

The script begins by evoking the `readOFF()` function that loads a mesh in the *indexed face list* simple data structure (see course notes for Lecture 2).

Essentially, we get a list of vertices `V` as an array of $|V| \times 3$ numbers, where each row represents a vertex coordinate in a xyz triplet, and a list of $|F| \times 3$ vertex indices in faces `F`, where each row is an *oriented* triplet of indices into `vertices`. `V` is a `MatrixXd` object (a matrix of doubles in arbitrary dimensions), and `F` is a `MatrixXi` object (the same for integers). To see how these two matrices interact, the script begins by printing all vertex coordinates for all vertices in face 100:

```
vertices of face 100:  3.53185  80.0961 -1.03406
-0.829358   80.5077    2.6428
3.75219 80.9468 3.42147
```

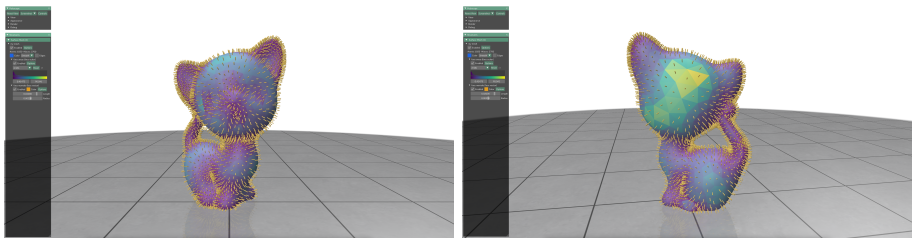

Figure 1: PolyScope viewing of face areas (color coded) and unit normals (Vectors from faces) from two different angles.

Your main task is to implement the function `compute_areas_normals()` that computes and outputs:

1. Triangle areas
2. *Unit* normals per face

For a face f with vertices v_1, v_2, v_3, the *unnormalized* normal can be computed as:

$$n_f = (v_2 - v_1) \times (v_3 - v_1)$$

The area of the face is then $A_f = \frac{1}{2}|n_f|$, and the normalized normal is $\hat{n}_f = \frac{n_f}{|n_f|}$. During the implementation, you may want to operator element-wise on a matrix (like for normalization, although you may use `.normalize()`). Eigen has functionality for that using the `.array()` function, and there are built-in `cwiseXYZ()` functions (like the `cwiseAbs()` we used before). You can also work single broadcasting operations row-wise with `.rowwise()` and respectively `.colwise()` for columns. You can use `.cross()` to do the cross product, but you'll have to cast the rows as `RowVector3d` objects.

The script proceeds by attributing both the scalar face areas and the vector normals to the PolyScope windows, where after choosing some GUI options to enable their views, it should look like Figure 1. Note that you can:

- Play with the ranges of the scalar function

- Change color schemes
- Play with the vectors' radius and length

Having done with the script, run the `Grading21` project to give you feedback on your implementation.

2.2 Computing edges

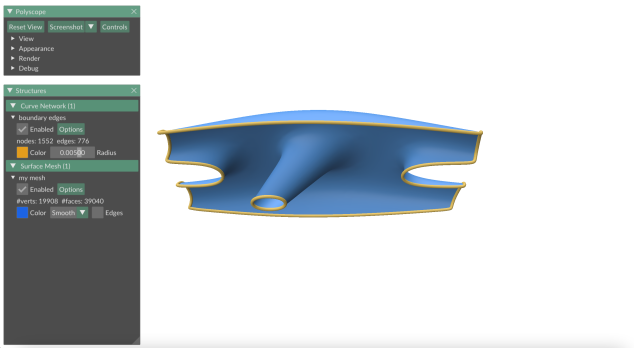

Figure 2: The boundary edges of the mesh are highlighted (as a PolyScope curve network).

Your next task is to compute a representation for the halfedges and the edges of a mesh, from which you isolate the boundary edges. These are computed within the function `create_edge_list()`. The halfedges output `H` is an array of size $3|F| \times 2$ of all halfedges of the mesh. The edges output `E` is a matrix of edge indices (which would be $|E| \times 2$ of indices into V), where the list you are giving back should have a *unique* row for every edge, in the sense that every existing edge ij is represented once as a row, without ji being an extra row. The `edgeBoundMask` output is an array of size $|E|$ which is 1 for a boundary edge, and 0 otherwise. You may safely assume the input is an orientable 2-manifold.

In general, the order of the elements in the edge lists should not matter as long as they correspond with each other; however, for compatibility with PolyScope, the list should correspond to a specific ordering detailed here: `https://polyscope.run/structures/surface_mesh/indexing_convention/#default-ordering`. In essence:

1. The halfedges appear in the same order of vertices on faces. For instance, if face 0 has vertices ijk, then the first three halfedges are ij, jk, and ki, and then on to face 1, etc.

2. Every halfedge belongs to an edge, where two halfedges (on each side) belong to an internal edge, and one halfedge belongs to a boundary edge. This list of edges should follow the same order as the list of halfedges,

where if an ij appears and then ji later in the halfedge list, we only use the first ij in the edges list.

This is a more complicated task than it seems; however it's more a thought-boggling task than a coding one, and it could be done in the following manner with just a few lines of code:

- Compute the matrix `H` of all *halfedges*, which amounts to stacking all the three edges within each triangle.
- Use the function `sort_rows()` to sort a copy of `H` so that every row of ji where $j > i$ gets sorted into ij, exposing the double-sidedness.
- use the `unique()` function that provides three outputs: `uniqueIndices`, the list of row indices into the input matrix that are the firsts out of all equals, `counts`, how many times they appear in the input, and `inverse`, mapping from `uniqueIndices` back to the input. Think how you need these to create `E` and `boundEMask`.

Rather than follow the above as exact steps, use them as guidelines; it is better if you understand the logic of the indexing used by PolyScope, and the way `unique()` works, and figure out your own solution. As a confidence check, the input file `TrainStation.off` has exactly boundary 776 edges. Running the rest of the script has PolyScope highlight the boundary edges you computed, which should look like Figure 2. Run `Grading22` to verify your code.

3 Sparse Linear Systems

Figure 3: Harmonic interpolation with a "harmonic function" enabled scalar function from fixed values (marked with a point cloud of two red points).

The task here is to solve linear least-squares systems using sparse matrices, and some fixed (constant) variables. This is a task that would repeat itself in variations in the rest of the course. The system we are going to solve is like the one learned in Lecture 4 (with its accompanying lecture notes):

$$x = argmin \left| d_0 \cdot x \right|^2,$$

subject to certain indices B within x having fixed prescribed values x_B, where the rest of the values x_I for the *free* indices I will be computed. The problem then amounts to solving the following least-squares linear system:

$$d_{0|I}^T \cdot d_{0|I} \cdot x_I = -d_{0|I}^T \cdot d_{0|B} \cdot x_B,$$

where $d_{0|B}$ is the matrix comprising the columns of d_0 that are in the index set B, and $d_{0|I}$ is the complement for only the free vertices in I. Note the rows are not sliced. d_0 is the $|E| \times |V|$ differential matrix defined in class as follows:

$$d_0(i,j) = \begin{cases} -1 & j = source(e_i) \\ 1 & j = target(e_i) \\ 0 & otherwise \end{cases}$$

This system solves a *harmonic interpolation* problem, where the free function values are chosen to minimize the difference between neighboring vertex values as much as possible. The results should appear very smooth. You will need to implement this entire logic in `harmonic_interpolation()` that receives the vertices, edges, fixed indices (B here) and fixed values (x_B here), and returns the *full* x, of size $|V|$. To represent d_0, you **must** only use a sparse matrix (never dense! It can be very large and almost entirely made of zeros). Use `Eigen::SparseMatrix<double>` that can be initialized in COO format (see Lecture 4), of triplets of respective (row, column, value) that you need to build with `Eigen::Triplet`. Use `SimplicialLDLT<SparseMatrix<double>>` to solve the system. You will likely also need to form I as the complement set of indices to B; for this, you can use `set_diff()`. *Important tip:* always minimize (and try to entirely avoid) random-access into a sparse matrix (i.e., querying some $d_0(i,j)$). The CSC format is compressed and this is very expensive. In general, after the construction of sparse matrices, you should manipulate them as little as possible, and here it only amounts to slicing into B and I parts, using the provided `slice_columns_sparse()`. The solution should look as in Figure 3, and use `Grading3` to verify your code.

4 Physical Simulation

here, we will explore the visualization of a simple simulation by an animation using PolyScope's callback mechanism. The script already set up the callback code, also adding an `isAnimating` mechanism to stop or continue the animation. Your job is to implement a bouncing ball of radius 1 (as a surface mesh loaded

from a file `spherers.off`), whose kinematics are controlled by the `displacement` of its center of mass, and its `velocity`. You will determine what happens in every iteration, advancing time forward by Δt which is `timeStep`, and always only update `currV`, the current location of the vertices relative to the original mesh defined by `origV` (which you should never update). In every time step, you simulate what happens during a short period Δt, where you should do the following according to order:

1. Increase velocity by $(0, -9.8, 0)$ times Δt, which means a free fall with earth's gravity acceleration (see Lecture 0). Note the "up" direction is the positive y axis (also corresponding to PolyScope's ground plane that we put at $y = 0$).

2. add velocity times Δt to the displacement. Have `currV` be the sum of `origV` and the displacement.

3. If the ball touches the ground, which means its position is in $y = 0.5$, bounce it back by inverting the sign of velocity; be careful to check that it's not already positive before you flip.

4. Update the visualization mesh by calling `.updateVertexPositions()`.

This is a very simple and inaccurate simulation. The effect is of a ball bouncing on the floor, where after a bit you will notice two main artifacts: 1) the ball will lose momentum and stop bouncing eventually, and consequently 2) it will slowly get "buried" in the ground. Think why that happens, and remember later in the course what we learn about these artifacts.

To witness different effects try the following:

- Replace steps (1) and (2). The ball will instead climb higher and higher.

- Don't check for velocity already being positive before flipping the sign. You will see considerable jittering.

- check if the minimum point of `currV` passed the plane (negative values of y), and correct for it by negating the offset.

Think why you are getting these artifacts! With this simple scheme, no answer will be entirely correct physically.