Programming Assignments 3 and 4 – 601.455/655 Fall 2024

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN 601.455 or 601.655

(one in each section is OK)

Name 1	Jayden Ma
Email	jma82@jh.edu
Other contact information (optional)	
Name 2	Edmund Sumpena
Email	esumpen1@jh.edu
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment Jayden Ma Edmund Sumpena

Grade Factor		
Program (40)		
Design and overall program structure	20	
Reusability and modularity	10	
Clarity of documentation and programming	10	
Results (20)		
Correctness and completeness	20	
Report (40)		
Description of formulation and algorithmic approach	15	
Overview of program	10	
Discussion of validation approach	5	
Discussion of results	10	
TOTAL	100	

Programming Assignment #3 Report

Overview

The goal of this project is to develop a 3D Cartesian library that can find the closest points from point cloud to a triangle surface mesh.

Description of Mathematical Approach

Before we can start finding the closest points, we have to first find the position of the pointer with respect to rigid body B for each sample frame k. To do this, we first transform from rigid body B coordinates to optical tracker coordinates, then we transform from optical tracker coordinates to rigid body A coordinates, then we transform from rigid body A to the pointer. This entire transformation would look like:

$$F_{B,k}^{-1} * F_{A,k} * \vec{A}_{tip}$$

Let the above transformation be represented by $\vec{d}_k$.
We can find $F_{B,k}$ and $F_{A,k}$ by registering the coordinates of body A and B to their respective sample coordinates for each frame.

The major mathematical objective is to determine how to find the closest distance from each point in the point cloud to the triangle mesh. We initially want to find the point of interest projected onto the plane of the triangle to find the closest point. To achieve this, we first re-write our triangle in Barycentric form, similar to the approach described in Dr. Taylor's slides on finding point pairs. We define the three vertices on the triangle as $\vec{v}_1, \vec{v}_2, \vec{v}_3$. We then choose one "reference point" (let's say v_1) and compute the edges of the triangle $\vec{e}_1 = \vec{v}_2 - \vec{v}_1$ and $\vec{e}_2 = \vec{v}_3 - \vec{v}_1$. We can compute the distances from all the points in the point cloud $\vec{p}$ from the reference point $\vec{v}_1$ to get $\vec{c}_j = \vec{p}_j - \vec{v}_1$. Solving the system $\vec{e}_1\lambda + \vec{e}_2\mu \approx \vec{c}_j$ as a least-squares problem allows us to find the Barycentric variables λ and μ. Since we know that the constraint $\nu + \mu + \lambda = 1$ must hold as a property of Barycentric form, we do not need to solve for the third variable ν in our system, since we know that $\nu = 1 - \mu - \lambda$. To find the projected position $\vec{p}_j^*$ of any data point $\vec{p}_j$, we simply evaluate $\vec{v}_1\nu + \vec{v}_2\lambda + \vec{v}_3\mu = \vec{p}_j^*$.

However, we only know that $\vec{p}_j^*$ is projected onto the *same plane* as the triangle, but we don't know whether $\vec{p}_j^*$ is on the triangle. We can look at the values of ν, λ, μ to determine whether $\vec{p}_j^*$ is within the triangle itself. If $\nu, \lambda, \mu \geq 0$, then $\vec{p}_j^*$ is within the triangle. Otherwise, $\vec{p}_j^*$ is either on the edge or outside the triangle. In each of these edge cases, we need to project $\vec{p}_j^*$ onto the closest edge to get $\vec{p}_j'$. We achieve this by linearly interpolating between two points with a scale factor determined by the ratio of dot products

$$\vec{p}_j' = a + \frac{(\vec{p}_j^* - a) \cdot (\vec{b} - \vec{a})}{(\vec{b} - \vec{a}) \cdot (\vec{b} - \vec{a})} \times (\vec{b} - \vec{a})$$

Intuitively, the scale factor determines how "far along" the line segment $\overrightarrow{ab}$ point $\vec{p}_j^*$ lies. We then use the scale factor to interpolate between the two vertices.

To search for the closest point, we can use simple mathematical concepts to improve the efficiency of our algorithm. If we know that the closest point on the mesh from a data point is distance d away, we can avoid the expensive computation of finding the closest point to the triangle if we know the point is not within d distance away from a *bounding box* around the triangle.

We also used 3D point cloud to point cloud registration as part of this homework assignment. We utilized the same iterative refinement algorithm from PA 1 and 2, which computes small rotation and translation "corrections" at each iteration until the error is small. The math behind the approach has already been detailed in past assignments.

Description of Algorithmic Approach

Our first goal was to find the closest points to every triangle on the surface mesh using a simple but slow linear search. The approach is simple – for each point, we search through all triangles to see whether any of them are closer than the current closest point. If there is, we replace the old closest point with the new one.

1. $Initialize\ \vec{m} \leftarrow \infty\ storing\ minimum\ distances\ to\ meshgrid$
2. $Initialize\ \vec{n} \leftarrow [0,0,0]^T\ storing\ closest\ points\ on\ the\ meshgrid$
3. $for\ each\ point\ \vec{p}_i\ in\ the\ point\ cloud$
4. $for\ each\ triangle\ \vec{t}_j\ in\ the\ meshgrid$
5. $Find\ bounding\ box\ b\ of\ \vec{t}_j\ and\ extend\ bottom\ left\ and\ top\ right\ corners\ by\ \vec{m}_i$
6. $If\ \vec{p}_i\ is\ within\ the\ expanded\ bounding\ box\ b, find\ closest\ distance\ from\ \vec{p}_i\ to\ \vec{t}_j$
7. $If\ new\ closest\ distance < \vec{m}_i, update\ \vec{m}_i, \vec{n}_i \leftarrow new\ closest\ distance\ and\ points$

We find the closest points from a triangle by following the mathematical approach described in the previous section. We first compute the edges of the triangle with respect to v_1 and solve a least squares problem using a numerical solver to get λ, μ. Next, we solve for $\nu = 1 - \mu - \lambda$ according to the Barycentric constraint. We create a matrix of vertices and a vector of ν, λ, μ. Matrix multiplication gives the projected point onto the plane of the triangle, which is denoted as $\vec{v}_1\nu + \vec{v}_2\lambda + \vec{v}_3\mu = \vec{p}_j^*$. To enforce the constraint that the point lies on the triangle, we check whether any of $\nu, \lambda, \mu < 0$ and project them onto the corresponding line segments. In the case that $\lambda < 0$ and that $\vec{v}_1$ is the reference point for $\vec{e}_1$ and $\vec{e}_2$, then we want to project $\vec{p}_j^*$ onto the line segment formed by $\vec{v}_2$ and $\vec{v}_3$. If $\mu < 0$, we project onto the line segment formed by $\vec{v}_1$ and $\vec{v}_3$. If $\nu < 0$, we project onto the line segment formed by $\vec{v}_1$ and $\vec{v}_2$. Finally, we compute the magnitude (L2 norm) of the difference between the closest point and original data point from the point cloud.

Since linear search is slow, we seek to implement a faster version of the algorithm using an octree. First, we construct an octree, which takes in a mesh of triangles. We then build the octree recursively by finding the center point and partitioning the data into octants to create eight subtrees. Each subtree is further broken down into octants and the process is repeated

until there are no more elements in the octree, or the size of the bounding box has fallen under some small threshold. After constructing the octree, we can search for closest points using the following algorithm (which uses the same idea as linear search):

1. $Initialize\ \vec{m} \leftarrow \infty\ storing\ minimum\ distances\ to\ meshgrid$
2. $Initialize\ \vec{n} \leftarrow [0,0,0]^T\ storing\ closest\ points\ on\ the\ meshgrid$
3. $Construct\ an\ Octree\ from\ the\ triangle\ mesh$
4. $for\ each\ point\ \vec{p}_i\ in\ the\ point\ cloud$
5. $If\ the\ tree\ contains\ no\ elements, then\ pass\ this\ iteration$
6. $Compute\ the\ bounding\ box\ b\ of\ the\ octree\ with\ minimum\ and\ maximum\ corner\ points$
7. $Extend\ bottom\ left\ and\ top\ right\ corners\ by\ \vec{m}_i$
8. $If\ \vec{p}_i\ does\ not\ lie\ within\ b, then\ pass\ this\ iteration$
9. $If\ octree\ is\ a\ child, find\ closest\ distance\ from\ \vec{p}_i\ to\ triangle\ \vec{t}_j\ in\ the\ tree$
10. $If\ new\ closest\ distance < \vec{m}_i, update\ \vec{m}_i, \vec{n}_i \leftarrow new\ closest\ distance\ and\ points$
11. $If\ octree\ is\ not\ a\ child, recursively\ search\ subtrees\ by\ repeating\ steps\ 5-11$

We also decided to write a more careful implementation of iterative linear and octree search of the closest point. In both of our initial approaches, we iterate through each point in the point cloud and process them individually. Instead, we rewrote each algorithm with *vectorized* operations, which allowed us to compute closest distance from a triangle for a batch of points, expand bounding boxes and test whether the box contains a point with a single vector operation, and exclusively update the points with newly found closest distances using array slicing/batched indexing. This completely removes the first loop over all points as the results for all points are computed at once. The runtime improvement is substantial and documented in the results section, since vectorized operations (like those in Numpy) can be parallelized while loops are strictly iterative. Vectorized versions of the baseline linear and octree search algorithm all perform faster than their iterative counterparts.

To develop our code, we used several libraries, which we installed in a Python 3.9 Anaconda environment:

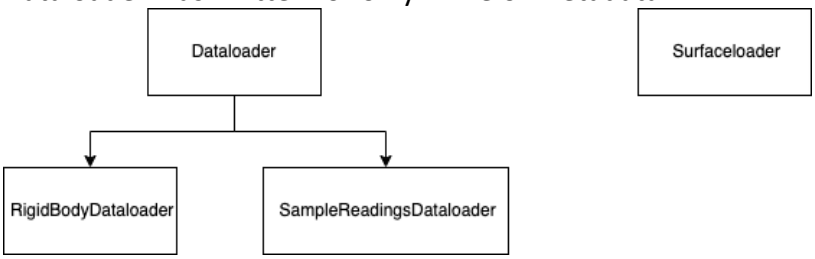
- **Numpy –** used to create vectors, create matrices, perform matrix multiplication operations, and solve least squares problems.
- **Pandas –** used to load and process the input data files as a .csv format.
- **Typing –** used to add Python type hinting for code clarity and to follow good practices.
- **Tqdm –** used to display progress bar and real-time error of the iterative registration algorithm.
- **Ipykernel** – used to run interactive Jupyter Notebook sessions for easy modular testing and data visualization.

Program Structure

Dataloader

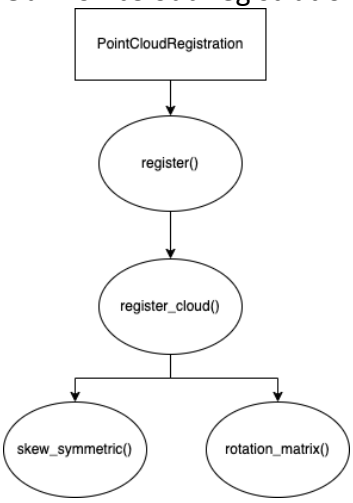
We split Dataloader into child classes for each file type. Since each file type is formatted differently, need to load the data from each file into arrays accordingly. Surfaceloader is a

separate class from Dataloader because it contains 2 separate lines of metadata, where Dataloader was written for only 1 line of metadata.



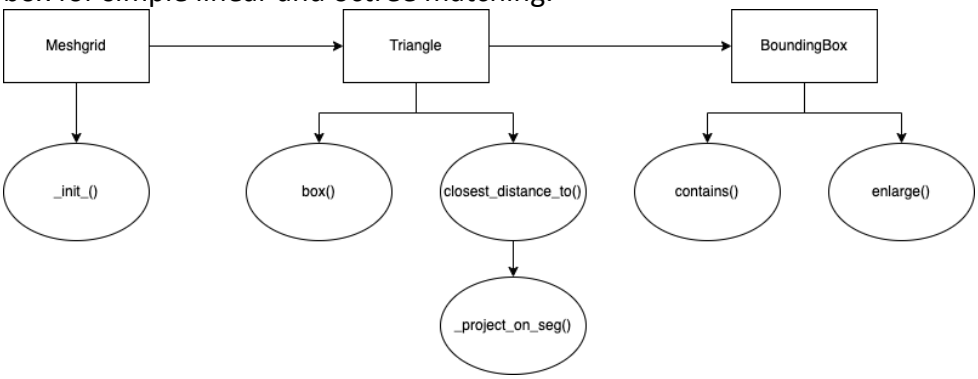
PointCloudRegistration

Our PointCloudRegistration class is the same from PA #1 and #2 (see report for PA #1 and #2)



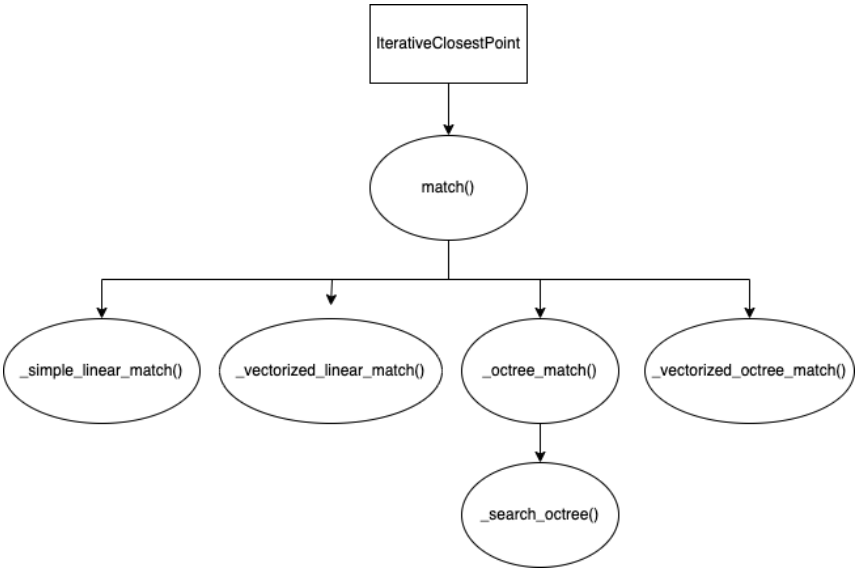
Meshgrid

The Meshgrid class stores the surface mesh as an Nx3 matrix of vertices, a Tx3 matrix of vertex indices, and List of Triangle objects. The Triangle class contains the 3 vertices that make up the triangle and the corresponding bounding box. We calculate the closest distances from a set of points to the triangle using the closest_distance_to() function. We represent the Triangle's bounding box with the BoundingBox class, which has the contains() function to determine whether a set of points are within the bounding box, and enlarge() which enlarges the bounding box for simple linear and octree matching.



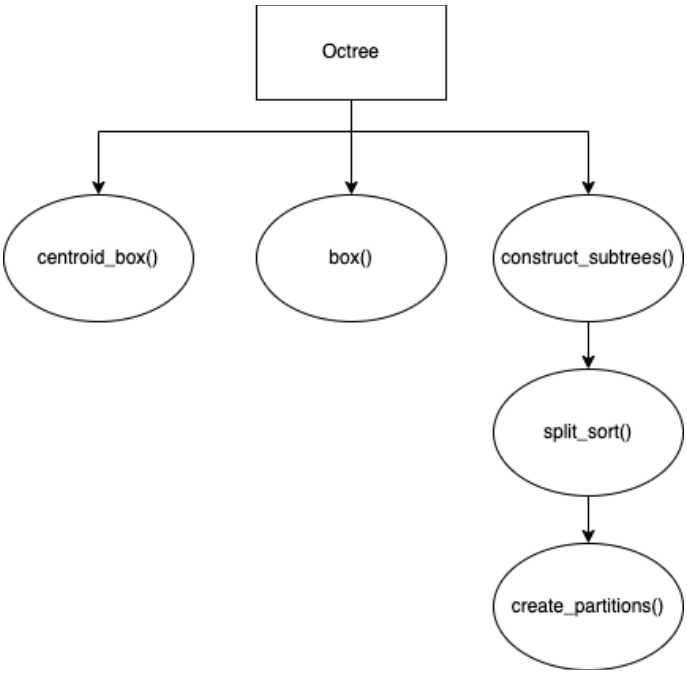
IterativeClosestPoint

Our IterativeClosestPoint class contains the base function match(), which calls a specific matching algorithm depending on the specified match mode in the class parameter. We implemented 4 matching algorithms: simple linear, vectorized linear, octree, and vectorized octree.



Octree

Our octree algorithms use the Octree helper class to build an octree. The Octree class contains the centroid_box() function that computes the bounding box of the triangle center point, the box() function that computes the bounding box of the triangle corners, and the construct_subtrees() function that splits the input elements into 8 regions, which uses helper functions split_sort() and create_partitions() that partition the region and assigns the input elements to their respective partition.



Validation Methodology

We created unit tests for:

- Closest point to a triangle when the point is directly above the triangle
- Closest point to a triangle when the point is outside of the triangle
- A bounding box containing a certain point

The TestClosestPoint class tests the closest distance from a triangle to a point. There are two cases:

Closest point to a triangle when the point is directly above the triangle

When the point is directly above the triangle, the closest distance should be a line from the point perpendicular to the surface of the triangle. The test proceeds as following:

1. $generate\ 3\ random\ points\ that\ make\ up\ a\ triangle$
2. $generate\ random\ point\ p\ that\ lies\ on\ the\ plane\ of\ the\ triangle$
3. $project\ a\ vector\ p'\ from\ p\ of\ random\ distance\ perpendicular\ to\ the\ plane\ of\ the\ triangle$
4. $find\ experimental\ closest\ point\ on\ and\ distance\ to\ triangle\ by\ calling\ closest\_distance\_to(p')$
5. $Becuase\ closest\_distance\_to(p')\ is\ only\ doing\ a\ simple\ geometric\ calculation,\ the\ error\ between\ experimental\ and\ expected\ point\ and\ distance\ should\ both\ be\ 0.$

Closest point to a triangle when the point is outside of the triangle

When the point is directly above the triangle, the closest distance should be a line from the point perpendicular to the edge of the triangle but not necessarily the surface.

1. $generate\ 3\ random\ points\ that\ make\ up\ a\ triangle$
2. $randomly\ pick\ an\ edge\ e\ of\ the\ triangle$
3. $generate\ random\ point\ p\ that\ lies\ on\ e$
4. $project\ a\ vector\ p'from\ p\ of\ random\ distance\ perpendicular\ to\ e$
5. $project\ another\ vector\ p''from\ p'of\ random\ distance\ perpendicular\ to\ the\ plane$

$of\ the\ triangle$

6. $find\ experimental\ closest\ point\ on\ and\ distance\ to\ triangle\ by\ calling\ closest\_distance\_to(p')$
7. $Becuase\ closest\_distance\_to(p')\ is\ only\ doing\ a\ simple\ geometric\ calculation,\ the\ error\ between\ experimental\ and\ expected\ point\ and\ distance\ should\ both\ be\ 0.$

A bounding box containing a certain point

The TestBoundingBoxContains class tests that BoundingBox's contains() function returns true when given a point that is inside the bounding box, and false when a point is outside the bounding box. The test proceeds as following:

1. $generate\ two\ (x, y, z)\ coordinates\ that\ define\ the\ maximum\ and\ minimum\ corners\ of\ the\ bounding\ box$
2. $call\ a\ BoundingBox\ object\ with\ max\ and\ min\ corners\ to\ create\ the\ bounding\ box$
3. $generate\ a\ point\ t\ with\ coordinates\ in\ the\ range\ of\ max\ and\ min$
4. $generate\ a\ point\ f\ with\ coordinates\ outside\ the\ range\ of\ max\ and\ min$
5. $assert\ that\ contains(t)\ returns\ true$
6. $assert\ that\ contains(f)\ returns\ false$

Results

To compare the predicted output with the ground-truth debug files, we compute the (x,y,z) component-wise mean absolute error of pointer tip $\vec{d}_k$, closest point $\vec{c}_k$, and $||\vec{d}_k - \vec{c}_k||$.

Dataset	Pointer tip MAE	Closest point MAE	Error norm
Debug A	0.004	0.002	0.003
Debug B	0.002	0.001	0.003
Debug C	0.002	0.002	0.003
Debug D	0.003	0.001	0.006
Debug E	0.004	0.003	0.003
Debug F	0.003	0.002	0.004

The following is our output for unknown files G, H, and J:

Dataset	Pointer tip average	Closest point average	Error norm average
G	3.205	4.449	2.061
H	3.876	3.393	1.342
J	8.256	7.35	2.177

To compare the run time of the different algorithms, we ran each algorithm 5 times to get an average run time with a standard deviation. All run times are in seconds.

Algorithm	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Runtime
Simple linear	3.03	3.23	2.97	3.33	3.79	3.27 ± 0.29
Vectorized linear	0.49	0.53	0.45	0.44	0.47	0.48 ± 0.03
Octree	2.06	1.84	2.43	1.83	1.88	2.01 ± 0.23
Vectorized octree	1.31	1.13	1.16	2.05	1.21	1.37 ± 0.34

Vectorized linear performs the best, while simple linear has the slowest run time. Vectorized linear is fast because computing distance and checking which points are within each triangle's bounding box are done with parallelizable matrix operations. Simple linear is slow because those same computations are done in sequence for each combination of point-to-triangle. Octree and vectorized octree are slower than vectorized linear because they require more pre-processing, such as creating the tree and partitions. However, they are still faster than simple linear because traversing a tree is faster than traversing an array. Both vectorized versions performed significantly better (> 1 standard deviation) than their iterative counterparts.

We worked on everything together through pair programming, then split up the report. Edmund worked on the "Overview and Algorithmic Approaches" and "Results" sections. Jayden worked on the "Program Structure" and "Validation Methodology" sections.