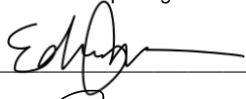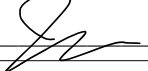# Programming Assignments 3 and 4 – 601.455/655 Fall 2024

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

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	

Overview

The goal of this project is to develop a 3D Cartesian library that can perform the full Iterative Closest Point algorithm; that is, we want to register the real-world point cloud of a bone to its digitized surface mesh version. To register the point cloud to the surface mesh, we will continuously find the transformation from point cloud to surface mesh until the two are sufficiently close to each other. The point cloud comes from sampled positions of the pointer tip when put on the surface of the bone, which we can get using the rigid bodies A and B as reference. Rigid body B is screwed into the bone, while rigid body A is the pointer device. The positions of both A and B are known because they are tracked by an optical tracker, and the surface mesh is already defined. To solve this problem, we will use the Iterative Closest Point algorithm with 4 different approaches to search for the closest point to every triangle: simple linear, vectorized linear, octree, and vectorized octree. We will compare the run times of these approaches to see which one is the fastest.

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 [1]. 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. Given that a point $\vec{p}_j$ is a distance d away from a triangle, we know that any potential candidate for closest point on the mesh must be closer that distance d, where distance is measured by the magnitude of the difference vector between two points. Thus, a candidate triangle must have a bounding box that overlaps with a sphere around the point with radius d. Conversely, we also conclude that a triangle can only be a candidate if the point is within the bounding box of the triangle plus a distance d away. 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 [1]. This allows us to rule out triangles that cannot be a candidate.

Another method we use to improve the efficiency of our algorithm by modifying the data structure in which we store our data. A simple approach stores each triangle in a linear list. However, a smarter approach is to build a spatial tree, an octree, to reduce the search space. More specifically, an octree is arranged hierarchically where each subtree is split into eight partitions or "octants." These octants are split based on the three axes through the centroid of all the elements in the tree [1]. Given an octree containing a set of triangles from the mesh, we can treat the center point of all the triangles as the origin, then form eight partitions – one for each octant of the coordinate system. Each of the octants of the original octree can then be further broken down into eight more partitions according to its new centroid. To reduce the search space, we can create a bounding box expanded by a distance d and check whether the data point $\vec{p}_j$ lies within the box (using the same mathematical backbone and concepts described above). However, the key conservation is that we do not need to do this for each triangle to eliminate its candidacy. We can construct expanded bounding boxes around each octant and eliminate *entire octants* instead of individual triangles. This reduces our search from being purely linear (as is the case with a list) to being closer to logarithmic. It is important to note that the search may not be purely logarithm if the points are not distributed evenly, i.e. in edge cases where most of the points end up within the same partition.

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.

After finding closest point, the next steps to implementing a successful ICP algorithm is to transform the point cloud and find the best "alignment" with the mesh [1]. To achieve this, we can constantly transform the point cloud in the direction of its closest points on the mesh, ultimately converging to a minima. Crucially, ICP may converge a local instead of the absolute minima, so factors like the algorithm's error metric for measuring the "similarity" of the point cloud and mesh, as well as its termination condition is important.

While there is not a hard rule for the error metric, most involve the *residual error*, or the difference between the transformed point cloud and their corresponding closest points on the mesh. This is represented as

$$\vec{e}_j = \vec{p}'_j - F_{best} \cdot \vec{p}_j$$

where $\vec{p}'_j$ is the closest point on the mesh corresponding to the point $\vec{p}_j$ in the point cloud. F_{best} represents the best rigid transformation predicted by the ICP algorithm. We opted to represent our overall point cloud to mesh similarity as the average magnitude of the residual error (or distance), which is expressed by

$$E = \frac{\sum_j \|\vec{e}_j\|}{n} = \frac{\sum_j \sqrt{\vec{e}_j \cdot \vec{e}_j}}{n}$$

where n is the number of points in the point cloud. We can also use the error term E to filter out outlier points. For example, if the closest distance from a point in the point cloud to its closest point on the mesh is significantly greater than the average error, then we can infer that the point is an outlier. Excluding outlier points will reduce noise in the registration and potentially improve the match between the point cloud and mesh.

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, which we do in our _simple_linear_match() function. 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 $\overrightarrow{m}_i$*
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 $< \overrightarrow{m}_i$, update $\overrightarrow{m}_i, \vec{n}_i \leftarrow$ new closest distance and points*

Using our closest_distance_to() function, 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 $v = 1 - \mu - \lambda$ according to the Barycentric constraint. We create a

matrix of vertices and a vector of v, λ, μ. Matrix multiplication gives the projected point onto the plane of the triangle, which is denoted as $\vec{v}_1 v + \vec{v}_2 \lambda + \vec{v}_3 \mu = \vec{p}_j^*$. To enforce the constraint that the point lies on the triangle, we check whether any of $v, \lambda, \mu < 0$ 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 $v < 0$, 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. We return the magnitude and projected point when given a triangle and set of points.

Since linear search is slow, we seek to implement a faster version of the algorithm using an octree in our _simple_octree_match() function. 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 $\overline{m} \leftarrow \infty$ 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 $\overline{m}_i$*
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 $< \overline{m}_i$, update $\overline{m}_i, \vec{n}_i \leftarrow$ 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 in our _vectorized_linear_match() and _vectorized_octree_match() functions, 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. For all search algorithms, we take in a point cloud and a surface

mesh as input and output the closest points and distances to the surface mesh for each point in the point cloud.

Finally, we implement the full ICP algorithm. The overall approach is straight forward: we 1) find the closest points and distances, 2) remove any outlier closest points based on distance, 3) compute the transformation from the point cloud to the closest points, and 4) compute error of the current iteration and check termination condition. Below is the pseudocode our implementation of the practical ICP algorithm [1]:

1. $Initialize\ error\ E_0 \leftarrow \infty,\ outlier\ distance\ threshold\ T \leftarrow \infty,\ F_{best}, F_0 \leftarrow I^{4 \times 4}$
2. $Compute\ closest\ point\ \vec{p}_j'\ and\ distance\ d_j\ for\ each\ point\ \vec{p}_j\ in\ the\ point\ cloud$
3. $Remove\ point\ \vec{p}_j'\ and\ \vec{p}_j\ if\ d_j \geq T\ for\ all\ j \leq n\ (where\ n\ is\ number\ of\ points)$
4. $Compute\ rigid\ transformation\ F\ to\ register\ \vec{p}_j\ to\ \vec{p}_j', where\ j \in n';\ n' \leq n$
5. $Update\ F_k \leftarrow F_{k-1} \cdot F,\ apply\ transformation\ \vec{p}_j \leftarrow F \cdot \vec{p}_j\ for\ all\ j \leq n$
6. $Compute\ point\ cloud\ to\ closest\ point\ error\ E_k \leftarrow \frac{\Sigma_j \|\vec{p}_j' - \vec{p}_j\|}{n}$
7. $Update\ outlier\ distance\ threshold\ T \leftarrow 3 * E_k$
8. $if\ E_k/E_{k-1} < \gamma,\ then\ F_{best} \leftarrow F_k\ and\ reset\ counter\ c \leftarrow 0,\ otherwise\ c \leftarrow c + 1$
9. $if\ c \geq C,\ then\ early\ stop\ the\ ICP\ algorithm,\ otherwise\ repeat\ steps\ 2 - 9$

For our algorithm's termination condition, we set some upper limit on the number of iterations, i.e. 200 iterations. Additionally, we perform early stopping with a "patience" of 10 iterations. If the algorithm's improvement is slow or stops improving altogether (where the ratio $E_k/E_{k-1} \geq \gamma$, where γ is a constant less than 1, i.e. $\gamma = 0.95$) for 10 iterations, then we terminate the algorithm. Any further iterations would yield diminishing returns and is not worth the longer runtime.

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
Our program consists of a main Python notebook that performs the matching from a point cloud to a mesh grid and a set of utility files that contain helper functions. We have the following utility files:

- Dataloader.py – loads data
- Coordinate_calibration.py – performs point cloud registration

- Meshgrid.py – represents a mesh grid as a set of vertices and triangles
- Icp.py – performs the ICP algorithm
- Octree.py – represents an octree data 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. Dataloader and its child classes take in a file containing raw data as input, and when initialized, outputs an object of itself containing the data organized into arrays of each type of data (i.e. an array for marker coordinates, an array for tip coordinates, etc). 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. The Surfaceloader class takes in a file containing raw data about triangle and vertex coordinates and when initialized, outputs an object of itself containing the data organized into separate arrays for triangles and vertices.
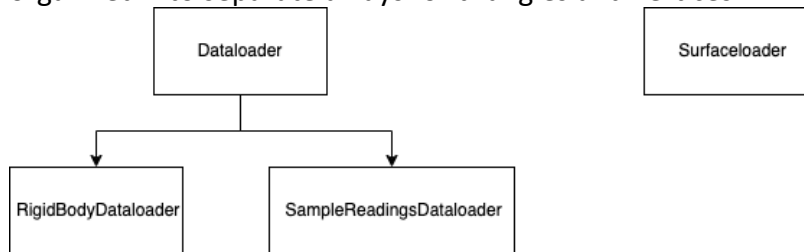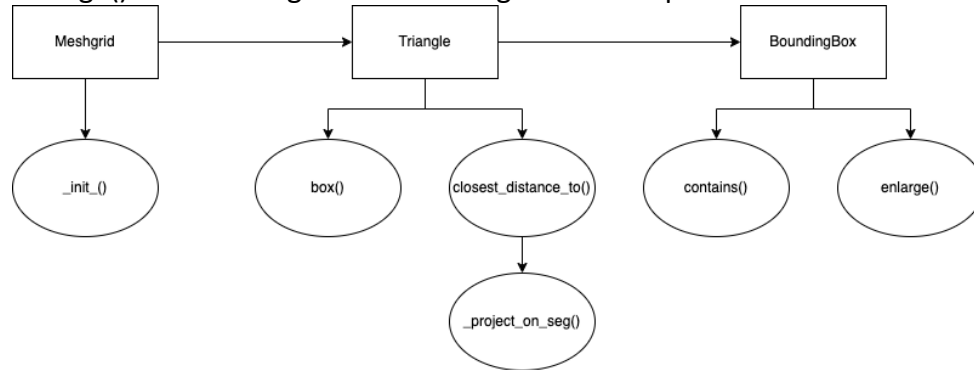

PointCloudRegistration

Our PointCloudRegistration class is the same from PA #1 and #2 (see report for PA #1 and #2). The register() function takes in a batch of two point clouds as input and outputs a 4x4 matrix representing the transformation that will register the first point cloud to the second point cloud.
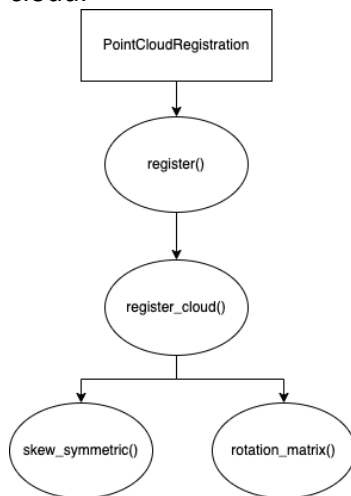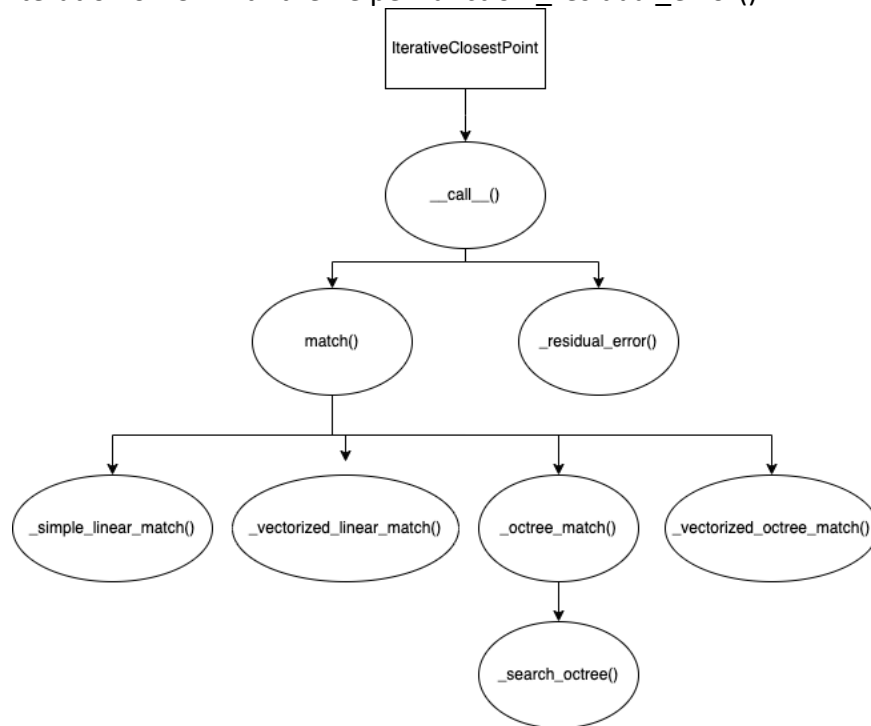

Meshgrid

The Meshgrid class stores the surface mesh as an Nx3 matrix of vertices, a Tx3 matrix of vertex indices, and a 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, which takes in a set of points as input and outputs a Tuple containing the closest distance to and closest point on the triangle. 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.

```
┌──────────┐        ┌──────────┐        ┌──────────────┐
│ Meshgrid │───────▶│ Triangle │───────▶│ BoundingBox  │
└────┬─────┘        └────┬─────┘        └──────┬───────┘
     │              ┌────┴────┐           ┌────┴────┐
     ▼              ▼         ▼           ▼         ▼
  ┌──────┐      ┌──────┐ ┌──────────────┐ ┌─────────┐ ┌─────────┐
  │_init_()│    │ box() │ │closest_distance_to()│ │contains()│ │enlarge()│
  └──────┘      └──────┘ └──────┬───────┘ └─────────┘ └─────────┘
                                ▼
                         ┌──────────────┐
                         │_project_on_seg()│
                         └──────────────┘
```

IterativeClosestPoint

Our IterativeClosestPoint class contains __call__(), which executes ICP for a specified matching algorithm when an IterativeClosestPoint object is created. The matching portion of ICP is done in the helper function match(), which calls a specific matching algorithm depending on the specified match mode in the class parameter. Each matching algorithm takes in a point cloud and a surface mesh as input and outputs the closest points and distances to the surface mesh for each point in the point cloud. We implemented 4 matching algorithms: simple linear, vectorized linear, octree, and vectorized octree. We also calculate the residual error for each iteration of ICP with the helper function _residual_error().

```
          ┌──────────────────────┐
          │ IterativeClosestPoint │
          └───────────┬──────────┘
                      ▼
               ┌───────────┐
               │ __call__() │
               └─────┬─────┘
              ┌──────┴───────┐
              ▼              ▼
         ┌─────────┐  ┌─────────────────┐
         │ match() │  │ _residual_error() │
         └────┬────┘  └─────────────────┘
    ┌─────────┼─────────┬──────────────┐
    ▼         ▼         ▼              ▼
┌─────────────────┐ ┌─────────────────────┐ ┌──────────────┐ ┌──────────────────────┐
│_simple_linear_match()│ │_vectorized_linear_match()│ │_octree_match()│ │_vectorized_octree_match()│
└─────────────────┘ └─────────────────────┘ └──────┬───────┘ └──────────────────────┘
                                                   ▼
                                            ┌──────────────┐
                                            │_search_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.
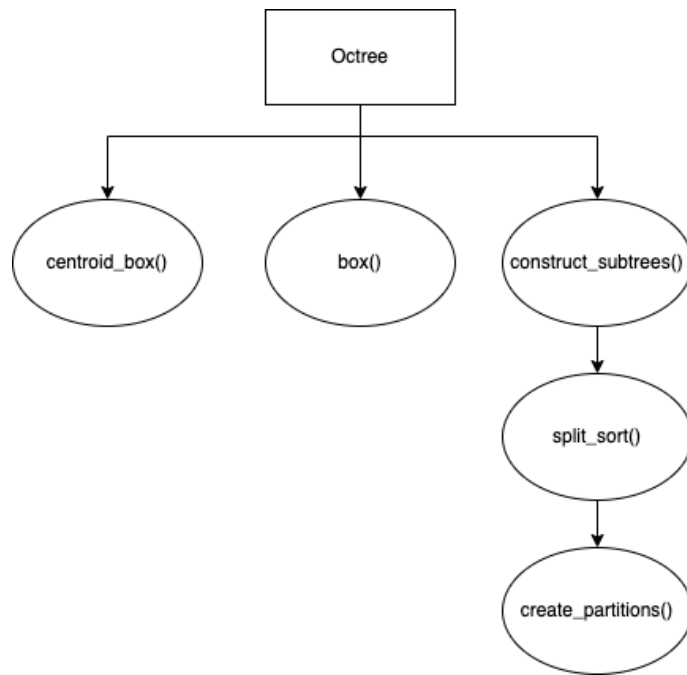
Validation Methodology

When performing our matching algorithms, we set our termination condition to be when the match ratio is greater than or equal to 0.95 because any improvements in matching past this ratio are small enough to be negligible.

To validate our search algorithms, we compared simple octree, vectorized linear, and vectorized octree with simple linear, which we know is correct from the previous assignment. The errors between our faster algorithms and simple linear is 0, meaning our faster algorithms are as accurate as simple linear.

Matching Algorithm	Error d_x	Error d_y	Error d_z	Error c_x	Error c_y	Error c_z	Error $\|\|d\_k - c\_k\|\|$
Simple Octree	0	0	0	0	0	0	0
Vectorized Linear	0	0	0	0	0	0	0
Vectorized Octree	0	0	0	0	0	0	0

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

For the closest point tests, the error between our ground truth and experimental points was very small, up to 1×10^{-16}, which is effectively 0.

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 $\|\vec{d}_k - \vec{c}_k\|$. We generated our predicted output using vectorized linear matching algorithm.

Dataset	Pointer tip MAE	Closest point MAE	Error norm
Debug A	0.002	0.001	0.001
Debug B	0.003	0.003	0.002
Debug C	0.006	0.005	0.007
Debug D	0.017	0.013	0.018
Debug E	0.039	0.035	0.026
Debug F	0.065	0.059	0.038

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

Dataset	Pointer tip average	Closest point average	Error norm average
G	-1.763	-1.76	0.013
H	0.404	0.402	0.008
J	-0.487	-0.484	0.063

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. We only ran simple linear and simple octree once because they took a long time to run.

Algorithm	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Runtime
Simple linear	708.06					708.06
Vectorized linear	13.67	14.35	13.46	13.28	13.05	13.56 ± 0.5
Simple octree	341.27					341.27
Vectorized octree	31.64	32.02	31.99	33.47	31.92	32.21 ± 0.72

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 (> 3 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.

References

[1] https://ciis.lcsr.jhu.edu/lib/exe/fetch.php?media=courses:455-655:lectures:finding_point-pairs.pdf