


## Programming Assignments 3 and 4 – 601.455/655 Fall 2023

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	Mannah Puhov
Email	hpuhov1@jhu.edu
Other contact information (optional)	
Name 2	Nidhi Batra
Email	nbatra3@jhu.edu
Other contact information (optional)	
Signature (required)	<p>I (we) have followed the rules in completing this assignment</p> <div style="text-align: center;">   <u>Nidhi Batra</u> </div>

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	

## Description of formulation and algorithmic approach

The goal of this assignment was to implement and use a simplified version of iterative-closest point registration to find the locations of a pointer tip with respect to a specific rigid body in tracker coordinates.

We first inputted the marker locations of the bodies (A and B) and their tip locations into corresponding data structures, as well as the sample readings from the optical tracker. For each sample frame, we found registration frames from body coordinates into tracker coordinates. We then found the position  $d_k$  of the pointer tip with respect to body B with the following formula:

$$\vec{d}_k = \mathbf{F}_{B,k}^{-1} \bullet \mathbf{F}_{A,k} \bullet \vec{A}_{tip}$$

For each pointer tip  $d_k$ , we then found the closest point on the mesh, which created our final output. This was done as follows:

To begin, we input the given surface mesh into 2 arrays. One to hold the vertex coordinates for all the triangles and the other to hold the indices of the vertices for each triangle.

We then implemented a “find closest point on triangle” routine. For this approach we used a barycentric formulation and followed the approach on slides 14 and 15 of the Finding Point Pairs slides. We defined each barycentric system as having coordinates  $l$ ,  $u$ , and  $v$ . We enforced the constraint that  $l + u + v = 1$  since this is a requirement for barycentric coordinates. We found the barycentric coordinates ( $l$ ,  $u$ , and  $v$ ) for the closest point on the triangle to a given point by applying the constraint and minimizing using the given point. This was how we implied the least squares sense to solve for the following system. We used the mean point of the triangle vertices (the centroid of the triangle) as our initial guess.

$$\mathbf{a} \approx \lambda \mathbf{q} + \mu \mathbf{r} + \nu \mathbf{p} \text{ such that } \lambda + \mu + \nu = 1$$

We then used the barycentric coordinates found ( $l$ ,  $u$ , and  $v$ ) to calculate the closest point on the triangle to the given point. If the barycentric coordinates were all positive, then the closest point on the triangle lied within the triangle and was given by

$$\mathbf{c} = \lambda \mathbf{q} + \mu \mathbf{r} + \nu \mathbf{p}$$

Where  $q$ ,  $r$ , and  $p$  are the vertices of the triangle.

If one of the barycentric coordinates was negative, we used the vertices corresponding to the other 2 vertices of the triangle (if  $u$  was negative, we used the vertices  $q$  and  $p$ ) to project  $c$ , the point found onto the edge created by the other 2 vertices of the triangle.

We used the following formulas to project the point c onto the line segment giving us the point c\*.

$$\lambda = \frac{(\mathbf{c} - \mathbf{p}) \cdot (\mathbf{q} - \mathbf{p})}{(\mathbf{q} - \mathbf{p}) \cdot (\mathbf{q} - \mathbf{p})}$$

$$\lambda^{(seg)} = \text{Max}(0, \text{Min}(\lambda, 1))$$

$$\mathbf{c}^* = \mathbf{p} + \lambda^{(seg)} \times (\mathbf{q} - \mathbf{p})$$

c\* returned the point that is the closest point on the triangle.

Once we found the closest point on the triangle, we implemented a slow method to find the closest point on the mesh. This worked by linear searching through all the triangles. This involved using an initial guess of the first vertex of the first triangle and then iterating through each triangle to find the closest point in the triangle. We checked the current triangle with the previous one and if the distance from the desired point to the closest point on that triangle was closer then the distance from the desired point to the closest point on the previous triangle, the current triangle became the previous one and we moved to comparing it to the next triangle. If the previous one had the closest point closer to the desired point compared to the current triangle, the previous triangle remained as the previous triangle.

We then sorted the triangles into a kdTree based on their location in a grid in 3D space. The goal of this setup was to reduce the amount of time it takes to search, because the point 'a' will be placed into its respective grid location. Then, only triangles in and neighboring that grid location would be searched for the closest point, rather than the entire list of triangles. Since a smaller set of triangles would need to be searched each time, this would theoretically significantly decrease the amount of time the search takes.

### Structure of Program

In our program, we primarily have 2 types of files, files that process the data and store it in the desired structure, and files that perform the expected calculations. We also have a main file, called main3.py that called our functions and allowed us to debug with the given input files. This file also created all of the data objects to store them in one main location.

The following files were created to help us organize the data:

- body.py organized the given body data (A and B) data into a list of marker LEDs and the pivot location.

- mesh.py organized the mesh file into an array of vertices and an array of triangles with their corresponding neighbors. It also had a method to get the vertices of a triangle from the index of that triangle.
- sampleReadings.py organized the readings of the body locations in tracker coordinates into a list of readings for body A and body B
- pa3Output.py created our output file with the results of the data and was called in main3.

The following files were used as “helper methods” or “helper classes:”

- We used our point.py and pointSet.py files from programming assignments 1 & 2. Our Point class was used to more clearly input the data and have access to our point set registration algorithms we wrote, which were in the pointSet.py file.
- In find Dk.py, using the marker LEDs we found the pointer tips with respect to the second LED body by calling our find\_registration and transform\_from methods from our programming 1 & 2 assignments and pytransform3d package<sup>1</sup>.
- Our method to perform the calculations for finding the closest point on a triangle was in a file called findClosestPointTriangle.py. In this we wrote a method to find the barycentric coordinates using SciPy’s optimization method<sup>2</sup> to minimize the squared distance function with the constraint given by a constrain function (coordinates must sum to 1) and then used the coordinates in a find\_closest\_point method to find the closest point on the triangle. We had a project\_on\_segment written that was called in this method when one of the barycentric coordinates was negative.
- The slow version of our find closest point method was written in findClosestPoint.py. We used a linear search method using a for loop and called the finding closest point on a triangle method for each triangle. If the point found on the current triangle was closer than for the previous triangle, the new one would become the closest point. We used the math library<sup>3</sup> to call the square root method for finding the distance between points to check what the smallest distance was to the point.
- In kdTree.py, the grid of point locations was created, and the candidate triangles (that is, the ones that are directly inside or next to the grid location of the test point) were searched for a closest point. This was made as a class because it prevents the need to continuously remake the grid, or copy all of the data. This also increases efficiency due to not having to keep track of an excessive amount of memory.

### Discussion of validation approach

We wrote a test suite that ran through our “find closest point on triangle” method with known values. This included comparisons where point ‘a’ was one of the vertices of the triangle, where point ‘a’ was on one of the edges, where point ‘a’ was the centroid of the triangle, and where

point 'a' was directly above the centroid in the z axis. The triangle was the same for all test cases, with coordinates (0, 0, 0), (1, 0, 0), and (0, 1, 0). This made the expected calculations easy. To avoid rounding errors with validation, we checked that the expected and actual values were within 0.01 units of each other. We did not include this file in our programs folder in order to save space.

We already had an existing test suite for our Point class and our point cloud registration, as those were both developed and thoroughly debugged for PA 1.

We ran a further validation by point checking each debug output to see if it matched the expected.

We were unable to finish validating the kdTree, so our final implementation includes the slow method.

### Discussion of results

For the most part, our debug outputs matched the expected ones. The main differences were likely due to rounding errors, as they showed up in the hundredths or thousandths place, and typically in the magnitude of the difference between our  $d_k$  and  $c_k$  values. Since we used a least squares minimizer that attempted to minimize the square difference, there was a large amount of computation involved that required rounding (finding the distance, squaring, minimizing it, multiplying the barycentric coordinates, etc.) It is likely that since we used the barycentric coordinate method, which has three unknowns rather than two, this contributed to the rounding error.

On occasion, there would be an issue with the registration calculation. Typically, re-running the program would fix this issue. For PA4, we would like to further increase the efficiency of our algorithm by finalizing the implementation of the kdTree. This can likely be completed simply through additional time running through each individual line of code with the debugger. Currently, there is likely an infinite loop in a certain location in the code, as it ran continuously without creating a clear output during trials.

### Workload Division Statement

The majority of the assignment was done together, with both of us brainstorming solutions to the problems. During in-person meetings, we came up with approaches and wrote the corresponding code. We also debugged primarily together. The report was added to by both of us, with Nidhi

writing most of the formulation description and program structure, and Hannah writing most of the validation and results discussions. We collaborated on all parts to discuss and edit each other's work.

## References:

1. Fabisch, A. (2019). pytransform3d: 3D Transformations for Python. *Journal of Open Source Software*, 4(33), 1159. <https://doi.org/10.21105/joss.01159>
2. Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) **SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python**. *Nature Methods*, 17(3), 261-272.
3. Van Rossum, G. (2020). *The Python Library Reference, release 3.8.2*. Python Software Foundation.