CIS1 PA3 Report

Jiahe Xu, Haoyu Shi

Overview:

- Files for preprocessing:
  - read_file.py
- Files from PA1 and PA2:
  - cartesian.py
  - registration_3d.py
- Files used during development and evaluation:
  - readin.ipynb
  - find_closest_point_test.ipynb
- Data folders:
  - OUTPUT
  - EVAL
  - DATA
- Calculations:
  - ICPmatching.py
  - KD_tree.py
- main.py

Approach:

We followed the instructions in the homework and the slides on finding point pairs.

Steps:

1. Find d_k, this was done by following the equations given in the homework instructions: $\vec{\mathbf{d}}_k = \mathbf{F}_{B,k}{}^{-1} \bullet \mathbf{F}_{A,k} \bullet \vec{A}_{tip}$ This step is used in both kd_tree and the brute force matching methods and acts as an input value.

Steps for brute force matching:

1. We need to calculate the ProjectOnSegment in order to determine the region the closest point would be. The code follows the formula as following:

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

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

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

2. The find closest point algorithm can then be implemented using the formula on the slides:

$$\mathbf{a}-\mathbf{p} \approx \lambda(\mathbf{q}-\mathbf{p}) + \mu(\mathbf{r}-\mathbf{p})$$

$$\mathbf{c} = \mathbf{p} + \lambda(\mathbf{q}-\mathbf{p}) + \mu(\mathbf{r}-\mathbf{p})$$

   According to the different regions determined by the λ, μ, the closest point is returned. The following chart is the formula we followed.

$\lambda<0$	*ProjectOnSegment(***c,r,p***)*
$\mu<0$	*ProjectOnSegment(***c,p,q***)*
$\lambda+\mu>1$	*ProjectOnSegment(***c,q,r***)*

3. Bruteforce_matching: this is a simple method of doing matching. It takes every point and finds the closest point on every mesh triangle, and saves the optimal one .
4. approach for KD-tree:
   Before building the KD-tree, we need to find the smallest cube that contains all three points in a mesh triangle, and all the sides are parallel to at least one axis; this is done by finding (X_{min} Y_{min} Z_{min}) and (X_{max} Y_{max} Z_{max}) among the three vertices (we save it in "rectangle"). Also, we need to save the index for each mesh since we will need the index to find the right triangle to do projection (the last column in "rectangle").

kdtree.building():

For each node in KD-tree, we need to save the smallest cube that contains all the triangles in its subtrees, this is done by the “pushup()” function. Everytime we sort the smallest cube according to the current axis in KD-tree (depth % self.D) In this case, we only need to use (X_{min} Y_{min} Z_{min}) vertices to sort cubes.

“middle” is the current node’s index in the tree, “left” and “right” are the range of nodes on the tree this node’s corresponding block covers.

we need to use self.num to save the index of the mesh triangle it corresponds to.

Then, we split the node set of the left subtree and node set of the right subtree, and build subtrees recursively.

kdtree.pushup():

For the smallest cube(all the sides need to be parallel to at least one axis), we only need to save two vertices (X_{min} Y_{min} Z_{min}) and (X_{max} Y_{max} Z_{max}) of its subtrees, then we can have the cube.

kdtree.find():

This function is used to find the closest point in meshes to a given point. When we reach a node in KD-tree, we will see if the best possible solution is better than the current solution(this is done by kdtree.point_to_cube()), if not the function just returns, since it is impossible to find a better solution in this cube. Otherwise, we should check the node’s corresponding mesh triangle and then recursively check it’s son nodes on the tree.

kdtree.point_to_cube(point, cube):

This function is used to calculate the shortest distance between a point to a cube.

We have point represented as: (p_x,p_y,p_z)

Cube represented as：(x_{min},y_{min},z_{min}) and (x_{max},y_{max},z_{max})

The vector between the assigned point and the closest point on cube is (x,y,z)

$x = x_{min} - p_x$ if $p_x < x_{min}$, $x = 0$ if $x_{min}<p_x<x_{max}$, $x = p_x-x_{max}$ if $p_x>x_{max}$

$y = y_{min} - p_y$ if $p_y < y_{min}$, $y = 0$ if $y_{min}<p_y<y_{max}$, $y = p_y-y_{max}$ if $p_y>y_{max}$

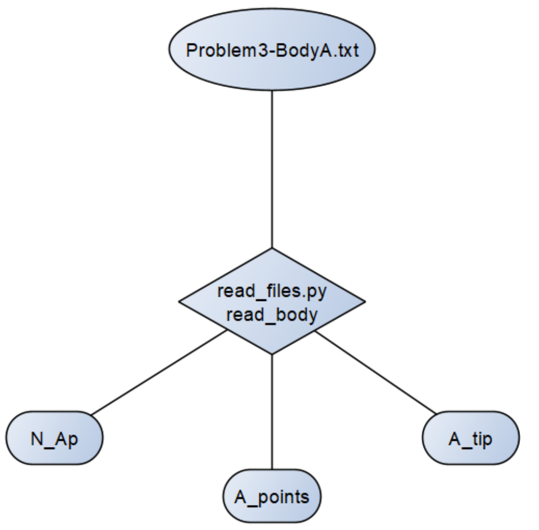
$z = z_{min} - p_z$ if $p_z < z_{min}$, $z = 0$ if $z_{min}<p_z<z_{max}$, $z = p_z-z_{max}$ if $p_z>z_{max}$

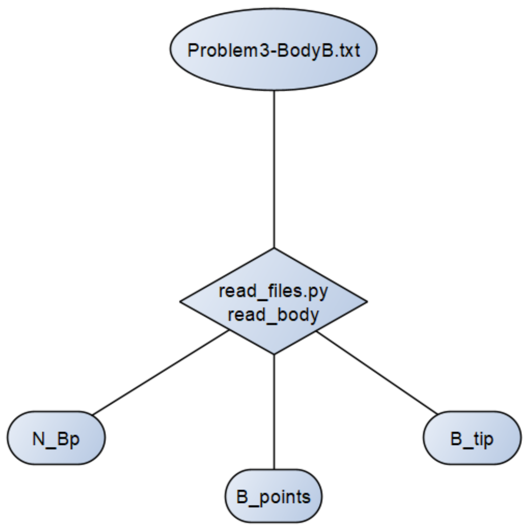
the L2-norm of (x,y,z) is the shortest distance between the point and the cube

Workflow:

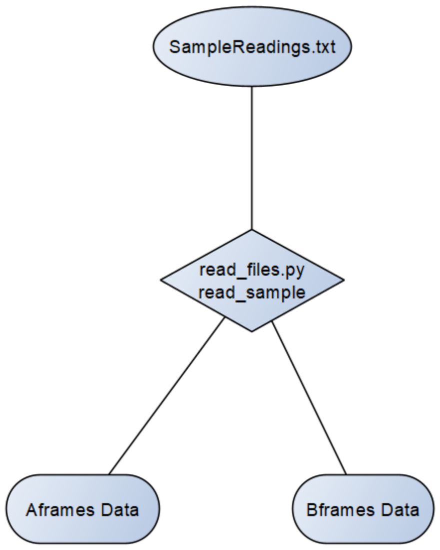
Preprocessing(reading all necessary data files):

a. **Reading Body Data:**

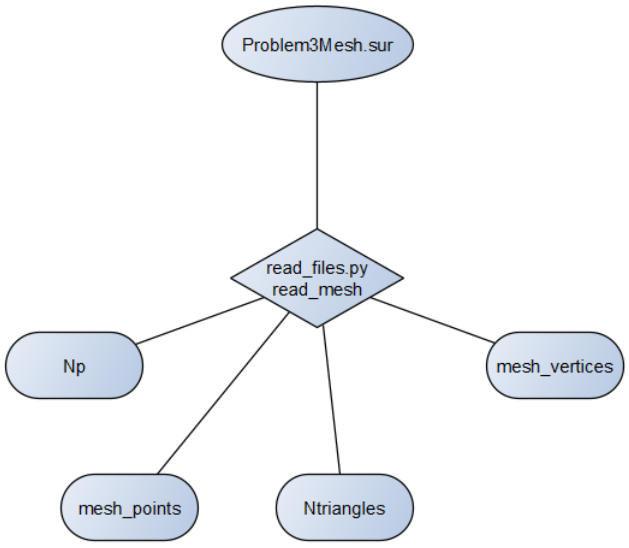




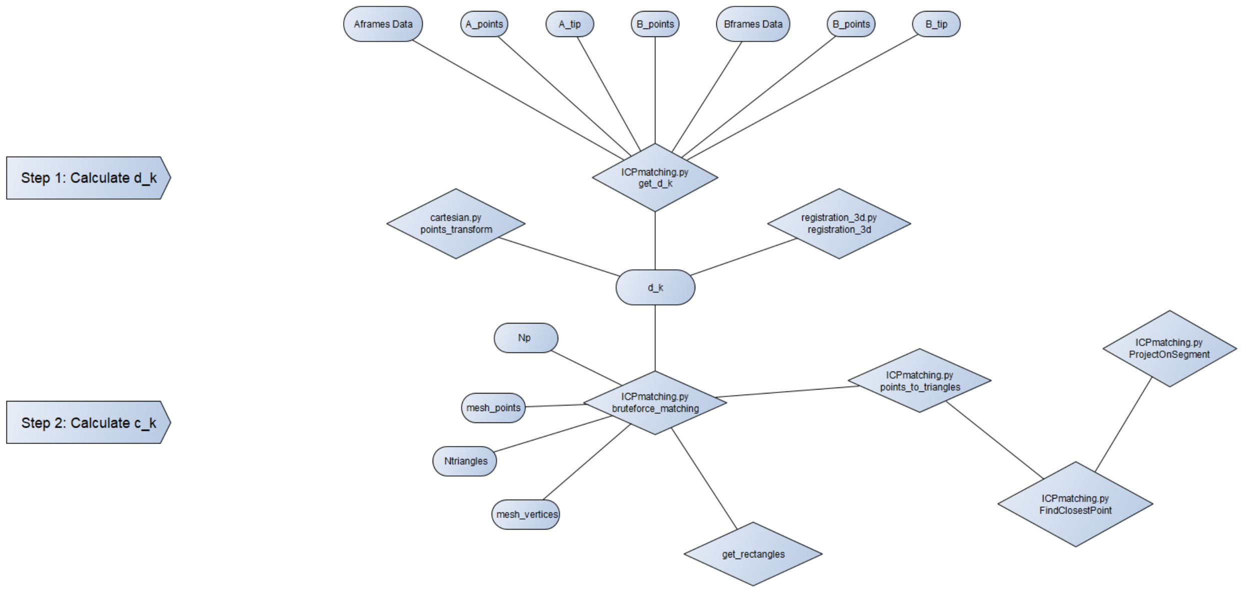
b. **Reading SampleReadingsData**



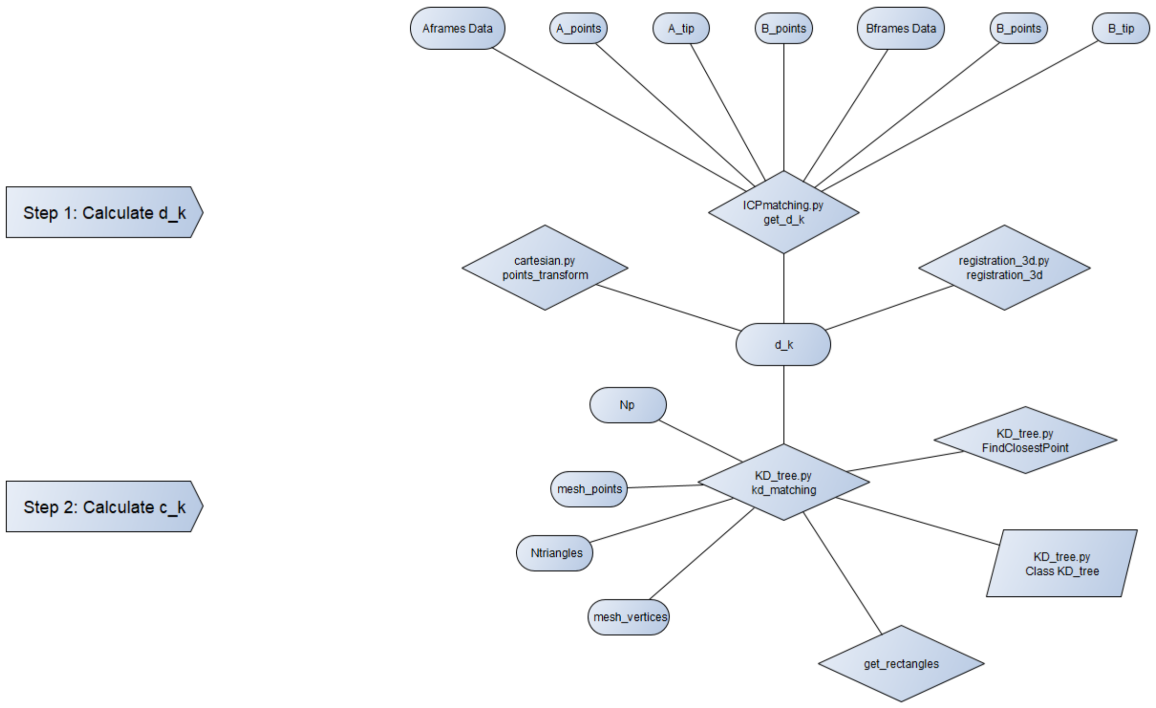
c. **Reading Mesh Data**



Brute Force Method:



KD tree method



In depth description of each file:

- Files for preprocessing:
  - read_file.py :

    During the last 2 assignments(PA1&PA2), we found that a lot of the functions required reading through files and this resulted in a large amount of redundant code. We therefore developed 4 reading functions. These can also be reused for PA4 since the data format stays the same.

    a. read_mesh: Reads the mesh file.
    b. read_body: Reads the Body A and Body B data.
    c. read_sample: Reads SampleTestReadings for each set of data.
    d. read_output: Used at the end to compare our results with. These data will be used as the ground truth to determine our error rate.
- Files from PA1 and PA2:

  - cartesian.py: No adjustments were made to this file as it only contains basic operations for matrices.
  - registration_3d.py: No adjustments were made as nothing could be further improved.
- Files used during development and evaluation:
  - readin.ipynb: Contains initial tests of the files. In addition, it was also used for graphing the results
  - main.py: Contains evaluations for each of the input dataset.
  - find_closest_point_test.ipynb: Contains tests for the function find_closest_point. This is used to ensure that this basic function is correct and would not add any more noise other than the noise contained in the data itself. The data used are all synthetic data generated by us.
- Files for actual calculation:
  - ICPmatching.py:
    - point_to_triangle and get_rectangles are both functions used for getting the bounding box.
    - brute_force_matching: Using a linear method for finding the closest points and completing the matching.
    - FindClosestPoint: Finds the closest point for brute_force_matching
    - get_d_k: Calculates d_k according to the given formula on the assignment instructions
    - ProjectOnSegment: Using interpolation to find c* in the given formula.
  - KD_tree.py:
    - Class KD_tree:
      - pushup, point_to_cube, find and build are all basic functions of the KD_tree
      - FindClosestPoint is as the name suggests, used for calculating the closest point on a kd tree structure.
    - kd_matching: First get rectangles then use the results to build the kd tree. Then apply the find_closest_point function to get the results.

- Folders with Data:
    - DATA: Contains all provided files.
    - OUTPUT: The output files generated in main.py by our calculations. They have the same format as the provided output files.
    - EVAL: Contains evaluation results of the input file. The generation of these files are inside of the main.py file. The headers explain the data types, the first part lists the averages, maxs, mins and error_nums(error rate) of the resulting Output variables compared to the provided Output.txt file; the second part shows the runtime of both brute_force and kd_tree methods accordingly in seconds.

Evaluation:

The produced results will be evaluated from the following aspects:

1. Tests for find_closest_point algorithm:
   Random datasets were manually generated and expected values were also manually calculated for comparison against our algorithm.
2. Time:
   We compared the results of running the algorithm for c_k using both brute_force and kd_tree methods. The runtime of both methods are recorded and compared for each individual file. The results are shown and discussed below.
3. Comparing results against Output.txt
    a. c_k
    b. d_k
    c. Magnitude of differences between c_k and d_k

   The difference is found for each of the results above(calculated_value - ground_truth_value). The **average**, **min** and **max** of the differences are recorded and plotted. In addition, we also filtered out the errors that are larger than 1. The graphs' error numbers are the error rates for each individual variable.

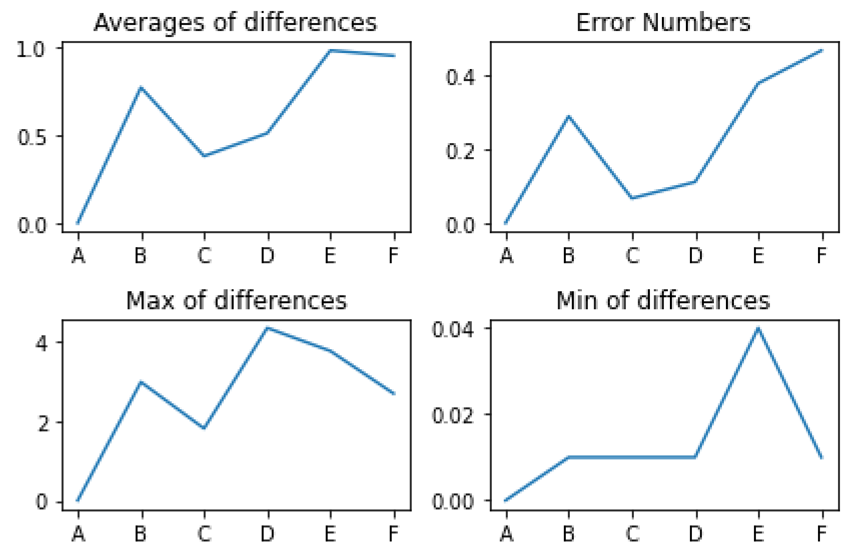
Results and Discussion:

Tests for find_closest_points algorithm:

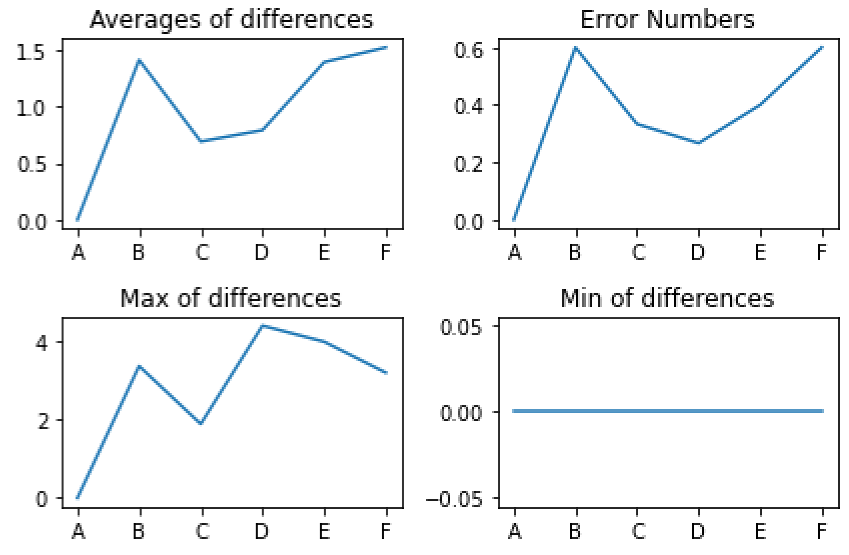
The outputs are in the notebook outputs shown in the file while the corresponding expected values are all shown in the inline comments. The expected results shown are calculated manually and seen as the ground truth. The comparison shows that all results are exactly as expected.

Final Results:

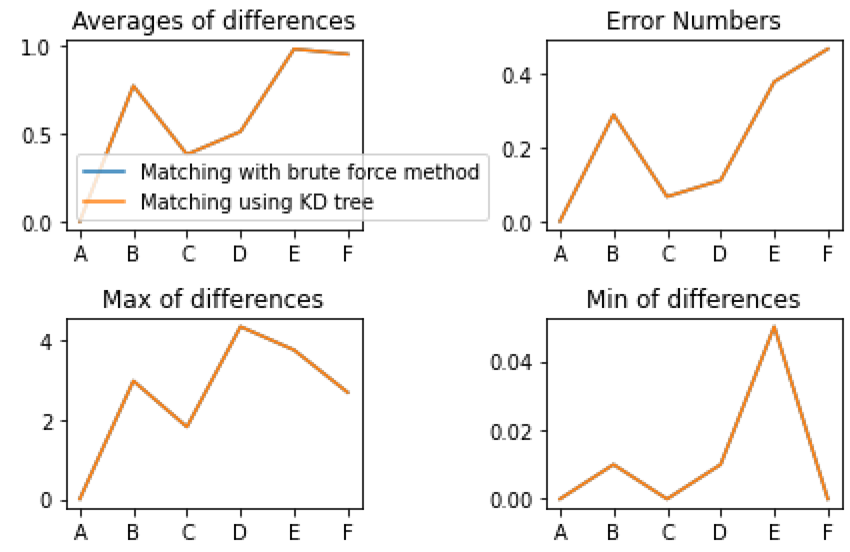
The three graphs below show accordingly our evaluation of the variables: "c_k", "d_k", "magnitude of difference between c_k and d_k". We will be mostly looking at the graphs reflecting average values and error rates. Max values will be evaluated if an exceptionally large value is met.
From all three figures below, the overall conclusion is that the dataset "Debug-a" is the most accurate and close to ground truth calculation with little to none error reflected in each of the graphs. Aside from A, the other two that perform exceptionally better than the others in all three aspects in the datasets C and D. Although we have not been given the chart describing noise as PA1&2, we can estimate that these follow the previous patterns as well. The different amount and type of noise in the datasets vary which causes the rising and falling error rates. In addition, given that in the process, the steps are only attempting to find a solution with minimal error, noise cannot be eliminated and that it could propagate and increase through the steps.



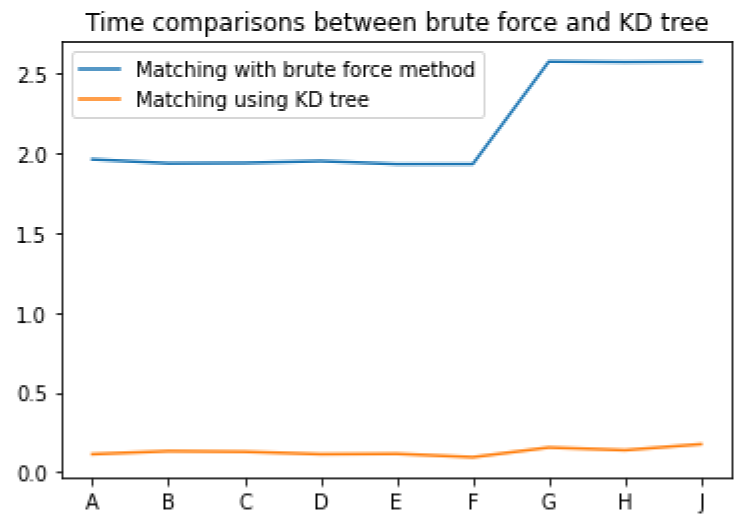
The last graph for the magnitude of difference between dk and ck shows slightly different results compared to the previous ones. The overall patterns are still similar. Min can be ignored since it provides less information. The change is in error numbers, it seems that B and F had a rapid increase in its error rate.



In addition, we have also finished the KD tree method. Therefore, we would like to examine its accuracy along with the brute force method. As we can see from the graph below, the legends of the graph show that the blue line shows the brute force method while the green line shows the KD tree method. In the graph, both lines are perfectly aligned, showing that the results are the same and the time efficient KD tree method does not affect its accuracy.



Finally, the graph below is the comparison between the runtime for each of the methods used in calculating each file. It shows that while most of the other files have been a smooth and flat line, there has been a noticeable rise of runtime for the three unknown test datasets. This is possibly due to an increase in the number of sample readings provided in the data files.(example: H has 20 samples while F only has 15)
Despite the increase, the increase of runtime in the KDtree method is minimal. This has also been obvious for each individual file. The KD tree method leads by at least 10 times less time. The lead is even more noticeable when it comes to the three unknown datasets with a larger dataset size. Along with the validated output data, we are able to conclude that the KD tree we have implemented is relatively successful.



In conclusion, the evaluation results were relatively satisfactory given that noise exists. The evaluations show that the Debug files A, C and D show a low error rate of mostly under 10% while the files B, E and F had a higher error rate possibly due to noise. In addition, the implemented KD tree method proves to be successful as it demonstrates the same error rate compared to the brute force method but shortened the runtime by at least 10 times.

Participation:
Jiahe Xu: KD-tree, bruteforce, report.
Haoyu Shi: ICPmatching.py, eval.py,report