CIS1 PA4 Report

Haoyu Shi, Jiahe Xu

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
    - tst.ipynb
    - eval.ipynb
- Data folders:
    - OUTPUT
    - EVAL
    - DATA
- Calculations:
    - ICPmatching.py
    - KD_tree.py
- main.py

Approach

First, the data has to be loaded and prerequisites need to be calculated.

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.

λ<0	*ProjectOnSegment(**c,r,p**)*
μ<0	*ProjectOnSegment(**c,p,q**)*
λ+μ>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 .

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

The above steps are the same as PA3.

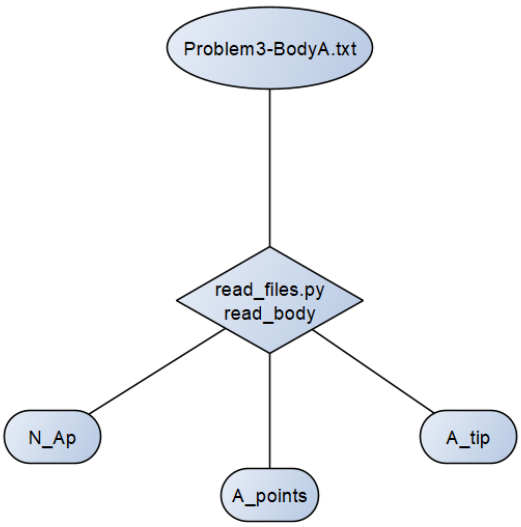
Then, apply the steps listed in the homeworks.

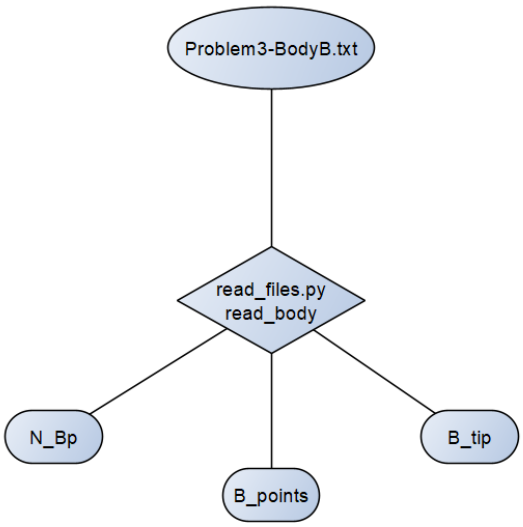
1. Make an initial guess $F_{reg} = I$
2. Apply this and obtain $s_k = F_{reg} \cdot d_k$
3. Use either KD tree searching by default or brute force to find the points c_k closest to s_k.
4. Use our registration_3d function to find ΔF_{reg}, the transformation between the original point s_k and the points c_k found by the closest point algorithm
5. $F_{reg} = \Delta F_{reg} \cdot F_{reg}$
6. Get the difference between F_{reg} and the old F_{old} from the beginning of the iteration.
7. If the difference is smaller than 1e-4, the calculations are considered complete. Otherwise, continue, the max iterations set is 60. During all tests on datasets A through K, none passed that max threshold.

Workflow:

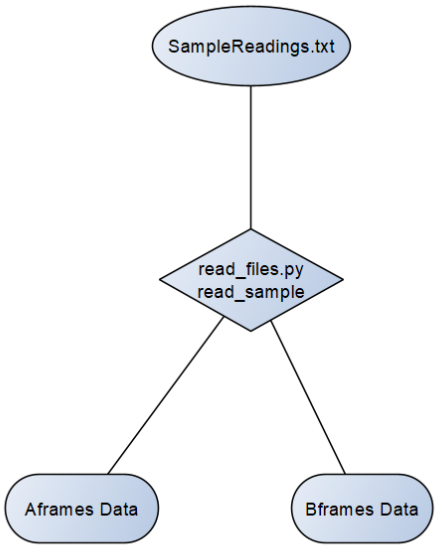
Preprocessing(reading all necessary data files):

a. **Reading Body Data:**

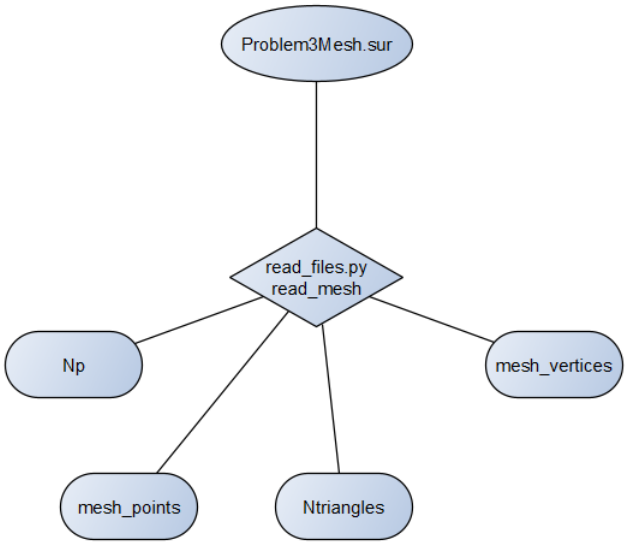




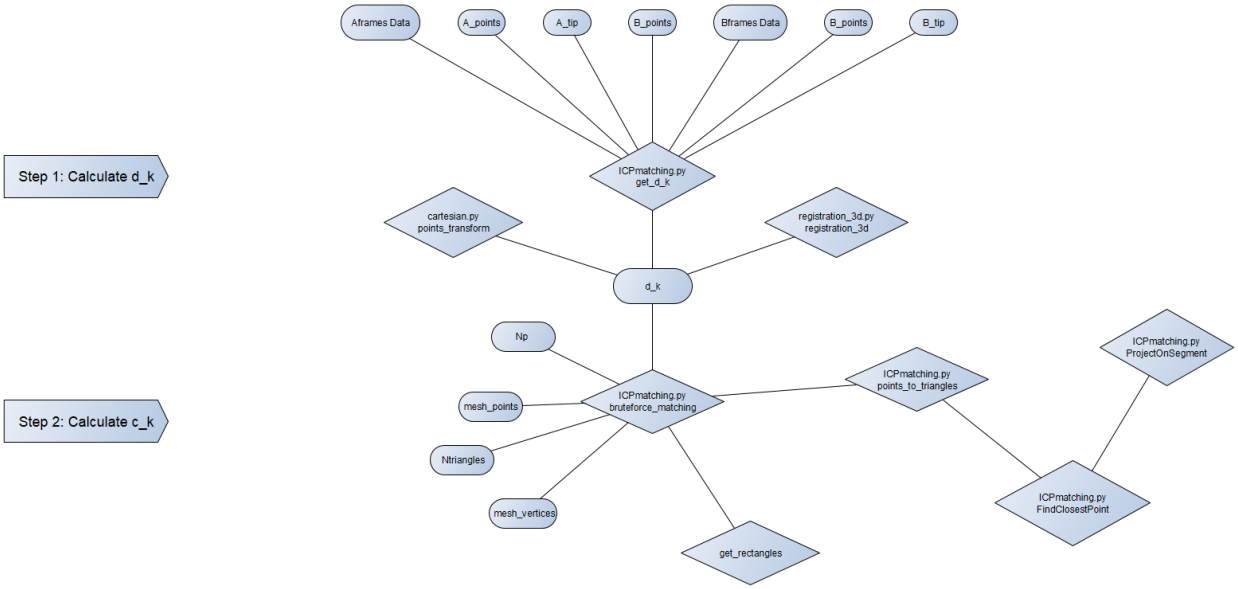
b. Reading SampleReadingsData



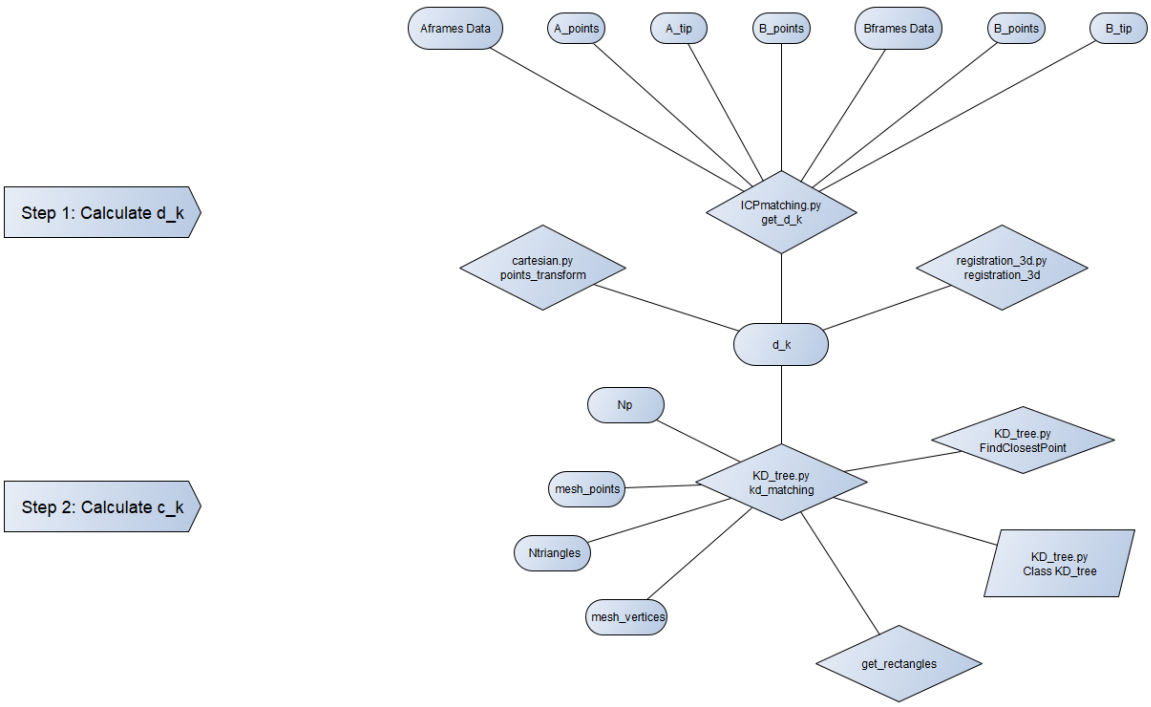
c. Reading Mesh Data



Brute Force Method:

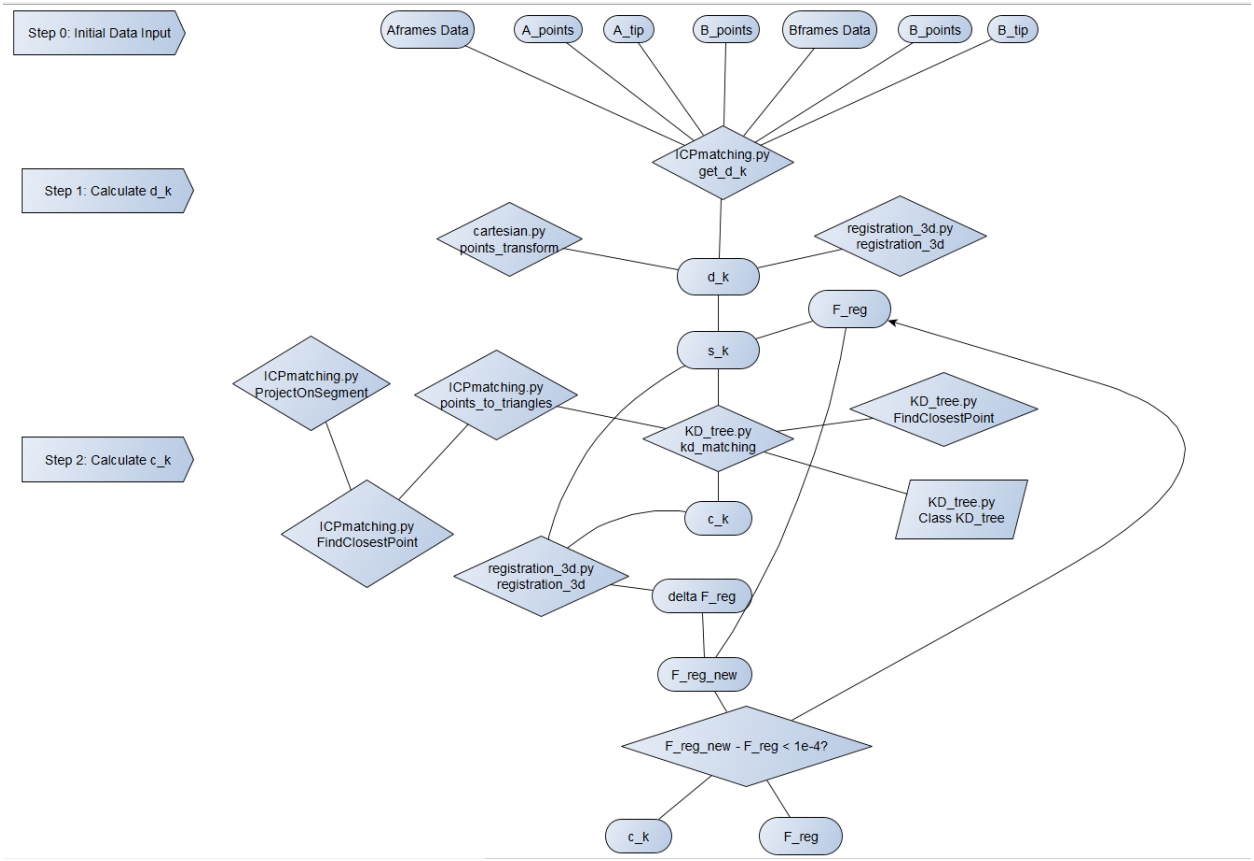


KD tree method



The above graphs are all copied from our reports in PA3 as those parts are the same.

Iterations: The following graph describes how the function ICP inside of KD_tree is implemented:



In-depth description of each file

Files from PA3:

- 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.

Files Adjusted and added functions:

- KDtree.py
    - Because the KD tree function and structure has already been implemented in PA3, we only made adjustments in the form of an addition of a function ICP in KDtree.py.
        - ICP: Calculates the Freg and ck like before. However, at the end of each iteration, the difference between the previous and current Freg is compared. If the difference is small enough(less than 1e-4), it is considered solved. In addition, the default option now is using the KD_tree method. We will further discuss the results and how huge the time difference is later.
          Note that this is placed in KD_tree.py in order to avoid causing mutual importing with ICPmatching.py.

Evaluation

We will be evaluating from the following aspects:

- Time differences for each set of data using Brute force and KD tree
- Confirming that the KD tree method works in the same way as KD tree.
- Comparing the average, max and min values of the differences. In addition, a percentage number will also be given by calculating the number of errors that are larger than 0.1.
- The last part of the discussion will focus on why two of the data sets were exceptionally erroneous.

Results and Validation

- Overview

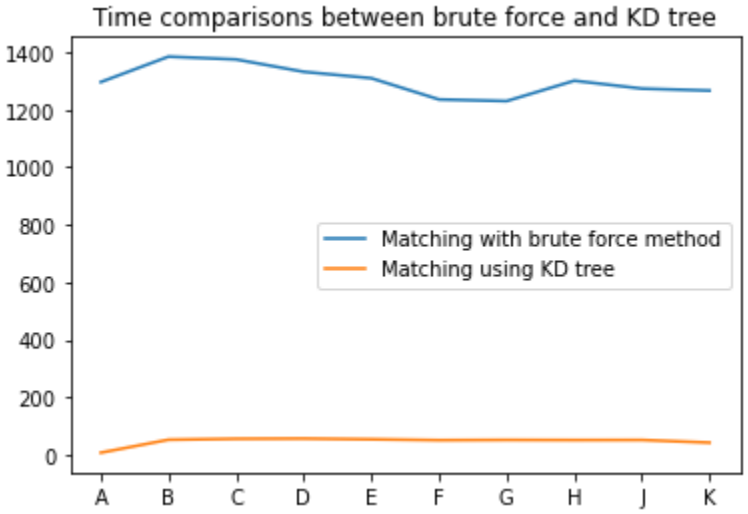
  The following image are the logs we inserted inside of the ICP program to have a clear view of the iterations it took to find the answers and ensure that at the end, the search converged.

```
/PA4-A-Debug
The results for has been found after 11 iterations!
Runtime is 5.142129898071289 seconds.
/PA4-B-Debug
The results for has been found after 42 iterations!
Runtime is 44.66003775596619 seconds.
/PA4-C-Debug
The results for has been found after 49 iterations!
Runtime is 54.368282079696655 seconds.
/PA4-D-Debug
The results for has been found after 43 iterations!
Runtime is 48.88608002662659 seconds.
/PA4-E-Debug
The results for has been found after 45 iterations!
Runtime is 48.728270053863525 seconds.
/PA4-F-Debug
The results for has been found after 38 iterations!
Runtime is 40.547048807144165 seconds.
/PA4-G-Unknown
The results for has been found after 46 iterations!
Runtime is 48.47833800315857 seconds.
/PA4-H-Unknown
The results for has been found after 50 iterations!
Runtime is 52.20839285850525 seconds.
/PA4-J-Unknown
The results for has been found after 46 iterations!
Runtime is 48.55504488945007 seconds.
/PA4-K-Unknown
The results for has been found after 30 iterations!
Runtime is 33.57232093811035 seconds.
```

As is shown above, the implemented algorithm has been able to find results for all datasets. Most of them converged in the area between 40-50 iterations. Further validations were done for the datasets A through F.

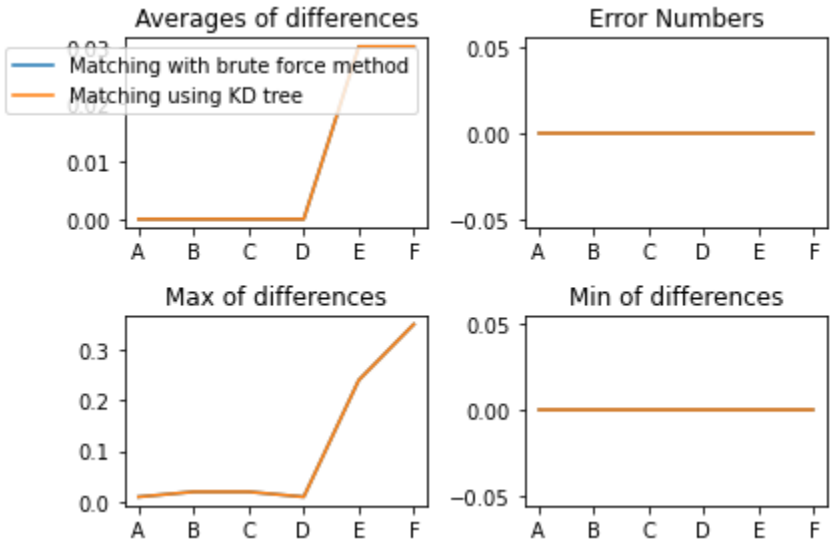
- Time differences:
  The graph below compares the run time for all datasets A through K. The significance of the difference between the two methods are evident. Matching

using KD_tree is maintained at a low run time for all datasets. The brute matching method takes up hundreds to thousands times more time.



- Comparing accuracies between the two methods

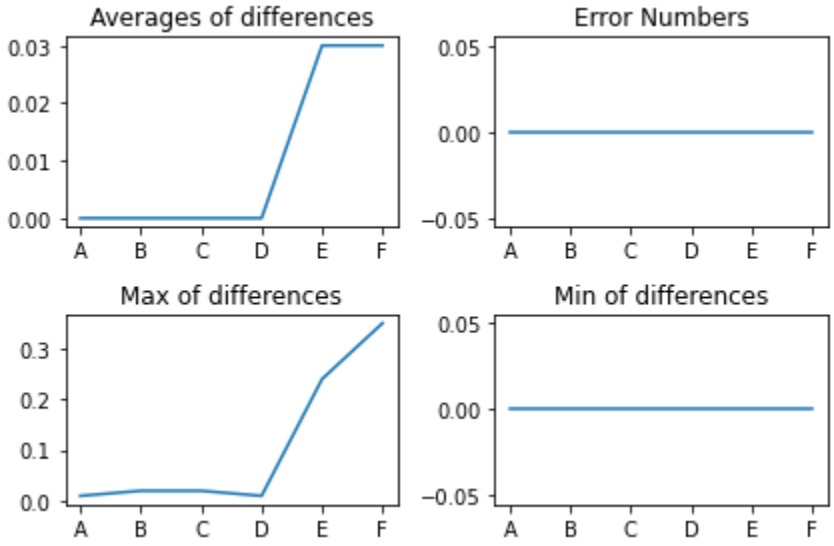
  The following graph displays the values of c_k for both methods. The lines are completely matched which proves that the accuracy was not compromised. **So the KD-tree version is correct.**



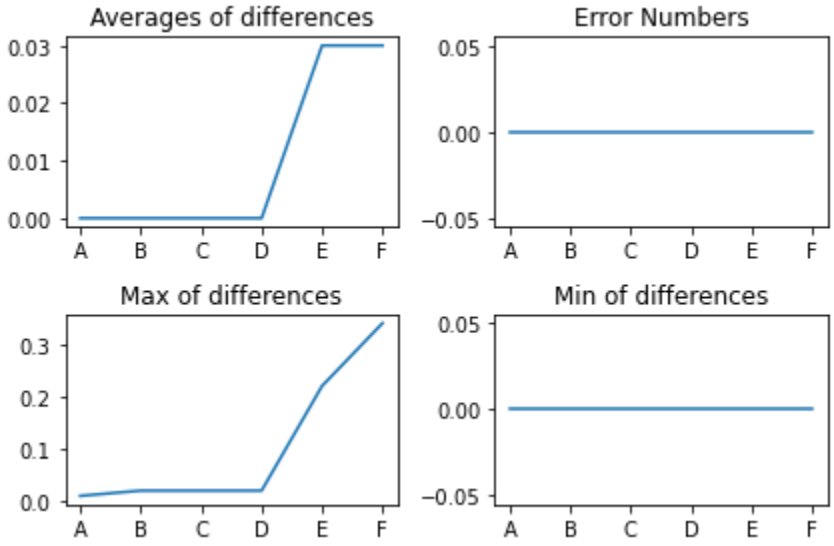
- Comparing and evaluating the results.

  The three graphs display the results of evaluating(in the following order) c_k, s_k and the magnitude of difference between s_k and c_k.

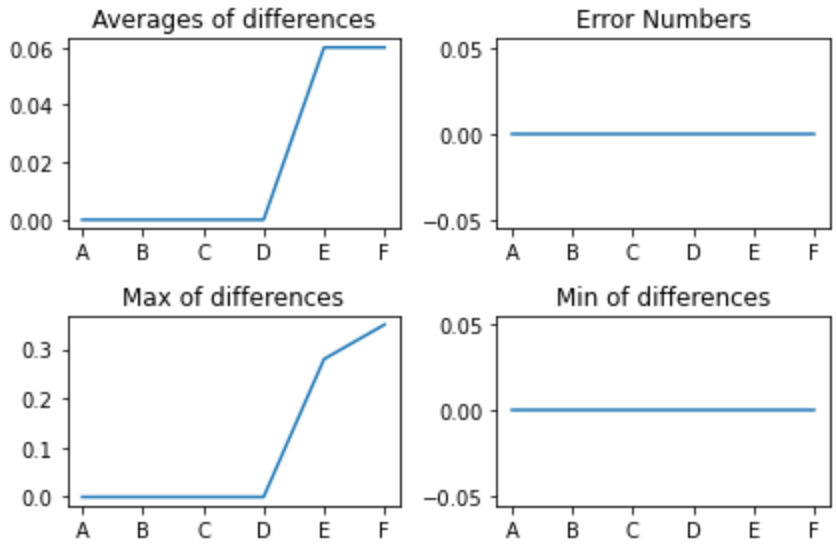
c_k:



s_k:

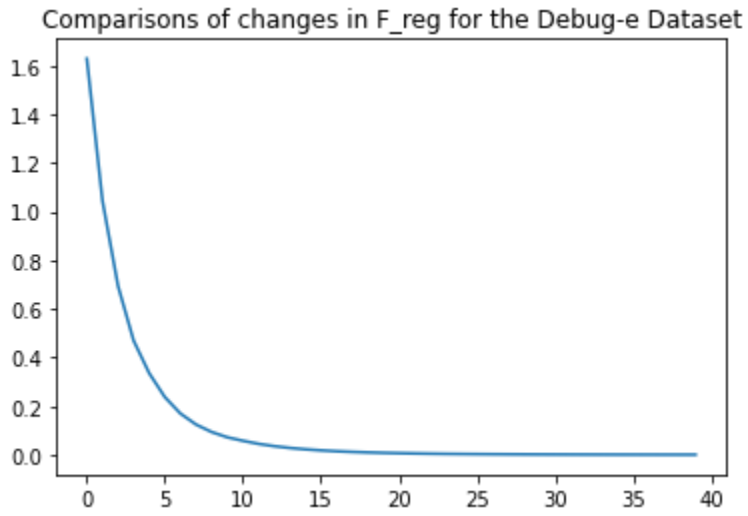


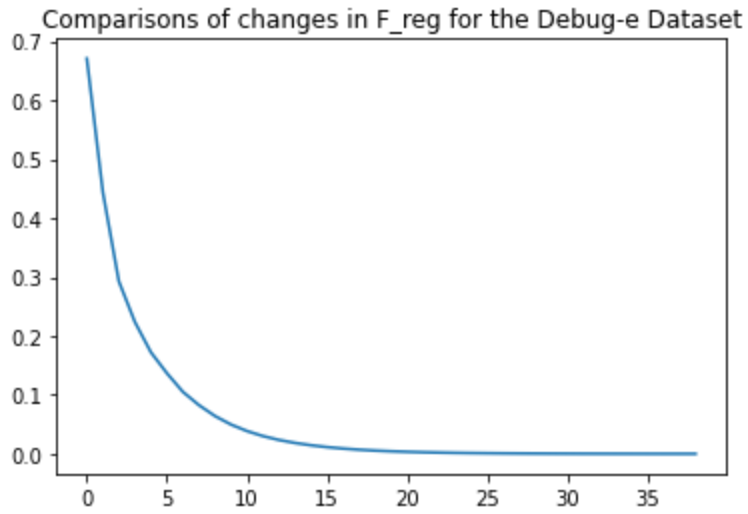
Magnitude of difference between s_k and c_k:



All graphs show that the datasets A to D performed exceptionally well and resulted in no error larger than 0.1. **This proves that the algorithm can perform correctly with minimal to low noise**. However, the datasets E and F both show a very rapid increase from no error to almost 20% in errors. In order to analyze what the issue could be, the F_reg data during intervals are examined below.

- Evaluating the spike in errors in datasets E and F
  The following graphs show the changes in F_reg throughout the calculation process.





Both of the graphs show a similar trend: the changes on F_reg through the iterations start to fall gradually. At the end of the iterations, before converging, both of them have minimal to none changes. This proves that our algorithm is stable.
Therefore, without other evidence and adding into consideration the high success rate in the previous datasets, we can conclude that our method is stable and meets the requirements. The most likely cause in the drop in accuracy for E and F is the noise in the data when generated or collected.

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