Programming Assignments 3 and [✓ 4] – 600.445/645 Fall 2016

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN [✓ 600.445] or 600.645

Name 1	Michael Mudgett
Email	mmudget1@jhu.edu
Other contact information (optional)	
Name 2	Mariya Kazachkova
Email	mkazach1@jhu.edu
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment [illegible] [illegible]

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	

Michael Mudgett
Mariya Kazachkova

CIS Programming Assignment 4

Overview of program

The program that we have written for Programming Assignment 4 is very similar to the program submitted for Programming Assignment 3, with two major differences. First, instead of the brute force method of searching for the closest point in every single triangle, this program employs a hierarchical data structure (covariance trees) to quickly search the mesh and find the closest point. Furthermore, this program uses an iteration to improve the estimate of Freg and minimize the distance between the calculated s_k = F_reg * d_k and the closest points c_k on the surface mesh.

In this report, we will discuss these two added features in detail. Since the other aspects of the program (transforming input points and computing F_A, F_B, finding the closest point on a triangle, etc.) are unchanged from Programming Assignment 3, we refer the reader to the PA_3 report for information on those parts.

Description of formulation and algorithmic approach

Hierarchical Data Structure: Covariance Trees-

Construction:

To start, we will briefly discuss the structure of our covariance trees and then detail the subsequent construction and search functions. The initial, top-level node of the covariance tree holds three important structures. The first of these is an array of "things," here it is an array of triangles. Note that a triangle stores the coordinates of its three vertex points. The other two structures are the right and left subtrees. Unless a node is a terminating node, a covariance tree node will have a right and a left subtree, each with its own array of triangles and daughter nodes. Furthermore, each node has a boolean to keep track of whether the node has or does not have daughter nodes. Finally, each node has an "F_node" transformation, made up of a rotation matrix R_f and translation vector p_f -- more on this later.

When construction of the covariance tree begins, all the triangles in the surface mesh file are loaded into the "things" array of a single node. This is the top-level node. An array of triangles is formed by reading in the list of coordinates and indices from the surface mesh file. We iterate over the lists and pull out the three coordinates corresponding to the corners of a single triangle (denoted by the indices). These three vectors are input into a "Triangle" data structure that we have created. An array of these triangles is passed to the first node.

With our triangle array, we then use the points to calculate our node-specific frame. Keep in mind that the goal here is to find a transformation so that we have the axis of of the mesh structure with the highest variance parallel to the x-axis. That is, the "longest" portion of the surface should align with the x-axis in 3D space. To do this, we first find the centers of each triangle. This gives a good approximation of the triangles' locations in space. We then take the mean of all the center points (of each triangle). This vector becomes the origin of our node coordinate system. We extract all the corner points from our list of triangles, giving us an Nx3 array of coordinates. The mean/origin point that we just calculated is then subtracted from each of the extracted points, resulting in a shifted list of points that define the surface mesh but are centered on the point (0,0,0).

Here, it is best to think of the array of translated vectors, u, as a cloud of points centered on the origin. To find a rotation that aligns the longest axis of the cloud with the x-axis, we need to calculate the covariance matrix. This is done easily using the np.cov() function on u_transpose. With this covariance matrix, we can extract a rotation matrix that will rotate our point cloud as desired. The first step here is to perform the eigenvalue decomposition of the covariance matrix[1]. We used np.linalg.eig(), giving us lambda, a set of eigenvalues, and Q, a 3x3 matrix of eigenvectors. To put the axis of largest variance on the x-axis, we had to rearrange the Q matrix such that the eigenvector corresponding to the largest eigenvalue was in the first column of Q. For example, if the largest eigenvalue in lambda was in index 0, we didn't have to change anything, since the rotation would place the axis of largest variance on the x-axis. However, if the largest eigenvalue in lambda was in index 1, swapped column 0 and column 1 in matrix Q. If we didn't do this, the axis of largest variance would rotate to be parallel with the y-axis. Setting our (possibly altered) Q to R_f and our mean point to p_f, we have our node transformation.

The final step for constructing a node is partitioning the triangles into the left and right daughter nodes/subtrees. The first thing we do when starting this step is check the size of the list of triangles in the current node. Our code splits the triangles in such a way that triangles that overlap the partition are placed into both subtrees. In general, we found that there are never more than 25 triangles that cross the divide. Thus, the minimum number of triangles in a node is set to 25 (minCount). This check is required because if there are too many triangles that cross the partition, the size of the subtrees will never decrease. Having 25 as the minimum keeps this from causing an endless recursion. It is true that searching through 25 triangles might take longer than one would like, but this works much faster than the brute force method. After the program ensures that there are 25 or more triangles in the node (it is not a terminating node), the program iterates through the node's array of triangles and sorts them into subtrees.

To sort a triangle, the program takes the three vertices (p, q, and r) and transforms them into the node coordinate system. This is done by subtracting p_f and multiplying by the inverse of R_f (order does not matter). Next, we establish our partition; we decided to choose the y-z plane. Thus, the sign of the x-value of the vector determines whether the associated triangle goes into the right or left subtree. A triangle is put into the right subtree if any of its three vertices have an x-value (after being transformed) that is greater than 0. Likewise, if any of the three vertices have a negative x-value, it is placed in the left subtree. All triangles go in at least one subtree.

Unfortunately, the algorithm is not this simple. First, we must discuss the rotation matrix that was calculated through the eigenvalue decomposition. The matrix Q, which becomes R_f, is not always a rotation matrix. It has a determinant of either 1 (rotation) or -1 (reflection)[2]. If the determinant is 1, then we follow the protocol laid out in the paragraph above. However, if the determinant is -1 and our R_f is a reflection matrix we do the opposite, placing vertices with positive x-values in the left subtree and negatives in the right subtree. This keeps the values in the untransformed space consistent and lets us enter the correct subtree while searching.

Once the triangles are divided, the two lists become the "things" array for the left and right daughter nodes. These are created and the process of building a node starts all over again. This continues recursively until no more nodes can be created. Here, all bottom-level nodes have less than 25 triangles and are deemed terminating nodes.

Search:

To search the tree and find the closest point on the mesh, we take the point of interest and transform it into the coordinate system of the top-level node. Recall that this uses R_f, p_f and inverse transformation calculation. We then look at the x-value of the transformed point. If it is positive and R_f is a rotation we return the search method for the point in the right subtree; if it is negative, we do the same for the left subtree. In the case that R_f is a reflection, the search method is called with the opposite subtree. This recursive searching continues until the program hits a terminating node (one that does not have subtrees). Here, the program searches through all the trees in the node and calculates the distance between the point of interest and the closest point on each triangle. The mesh point that gives the smallest distance from the point of interest is returned. Here, it is important to note that with this method, the "brute force" search is performed on a maximum of 24 triangles. Compared to the 3000+ triangles in the purely brute force method, the covariance tree search is much more efficient.

Iteratively Calculating F_Reg-

As we stated in the Programming Assignment 3 report, the initial guess for F_reg was R_reg = I and p_reg = 0. We start by calculating our estimate for the closest point: s_k = F_reg * d_k where d_k is the known position of the pointer tip with respect to rigid body B. We search the tree structure with s_k to find the closest point on the mesh, c_closest. By the definition c_closest = F_reg * d_k, we do a point cloud to point cloud registration between our d_k points and our new c_closest values. This is assigned as the new F_reg. We iterate this process, computing s_k and then searching for the tree to find new c_closest values and a new F_reg. The loop terminates when the euclidean distance between the c_closest and s_k points changes by less than 0.5% from one iteration to the next.

<u>Function Descriptions</u>

Functions used in PA4 that were created/explained in earlier Programming Assignments:

- registration_3d_3d(A_1,B_1) [PA1]
- transform(R,p,c) [PA1]
- inv_transform(R,p,c) [PA1]
- read_coords(num,f) [PA1]
- find_closest_point(a,p,q,r) [PA3]
- project_on_segment(c,p,q) [PA3]
- distance(a,b) [PA3]
- print_part3() [PA3]

New Functions:

<u>*Class: Triangle*</u>

The Triangle class has no functions, and only acts as a data structure to store three 1x3 vectors which represent the three vertices of a triangle in 3D space.

<u>*Class: Cov_Tree_Node*</u>

The Cov_Tree_Node is the structure for a single node in our covariance tree. Details on the construction and searching are given above.

Constructor: __init__(self,ts,nT)

The constructor initializes the instance variables and then calls compute_cov_frame() to define a node coordinate system followed by construct_subtress() to create the subnode and start the recursive tree construction. "Ts" is the list of triangles and "nT" is the number of triangles.

compute_cov_frame(self, ts, nT)

This function extracts the individual vertex points from the list of triangles and finds the mean. It subtracts the mean from the center points of all the triangles and calculates the covariance matrix of the shifted centers. A rotation/reflection matrix is then calculated using the eigenvalue decomposition of the covariance matrix and the columns are permuted so that the axis of largest variance will lie on the x-axis. It sets the rotation matrix and mean vector to instance variables for use in later transformations.

extract_points(self, ts, nT)

This function iterates through the array of triangles and pulls out each of the vertex points, creating a long list of points. It compresses the list to eliminate repeats and returns the resulting array.

centroid(self, points)

This function takes an input of a list of points (Nx3 array) and calculates the mean. It returns the mean, which is the centroid of the list of points.

construct_subtress(self)

This function iterates through the list of triangles and transforms the vertices into the node coordinate system. The triangles are placed into at least one list, depending on the signs of the x-values of the vertices. For more information on decision making see the previous section. A right and left list are created. The constructor is called twice, with each of the lists as the “ts” input. This enters a recursive loop until all terminating nodes have been created.

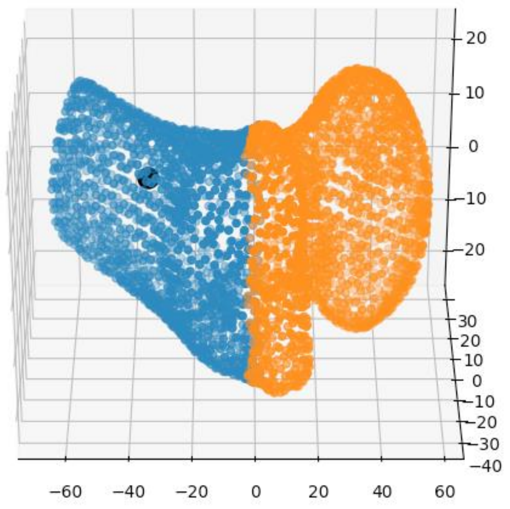
three_d_plot(self,l,r,v)
This is a function used for testing and visualizing the surface mesh.

search_tree(self, a, c,dist)
This function searches the tree to find the closest point on the mesh. The inputs are a - the point of interest, c - the previous closest point (we set this to a very far away value), and dist - the distance between a and c. The function searches the tree as described above and returns the point on the surface mesh that is closest to a.

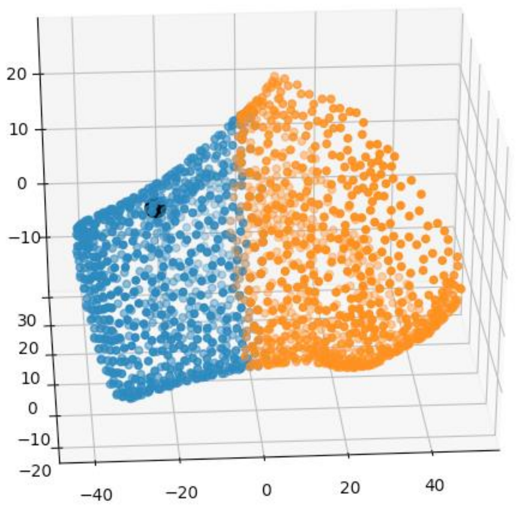
Discussion of validation approach

To ensure that our functions were working correctly, we created some unit tests. Specifically, we wanted to test the functions in our Covariance Tree class. We tested the creation of the coordinate transformation into the node's coordinate system by creating a small list of Triangle objects, creating a covariance tree node using that list, and running our function for computing the transformation of the node. We then compared our results to the expected results. We tested our function for extracting points in a similar way: creating a test node with test data and making sure that our extract points function returned the extracted points from the list of Triangles (we made sure that there was one duplicate point between 2 triangles and that only one copy of that point was returned when points were extracted). Additionally, we tested our centroid function by creating the same test node and making sure that the correct vector was returned for the center of all the points.

Furthermore, we decided to try and visualize what was happening within our tree as we were searching to ensure that the correct subtree was being searched. This was done by plotting the centers of the triangles from each subtree as well as the sample point for each iteration with respect to the coordinate system of the current node. We were able to then visually follow through and ensure that we were always searching in the correct subtree. This really helped in debugging as well as getting visual assurance that our program was behaving as intended. Below are a few screenshots of the procedure:



Here you can see the centers of the two subtree plotted in different colors, as well as the current sample point plotted in black. From this, we expect to move into the blue subtree.



This figure shows the next node in the search. Although at a different angle, one can clearly tell that we have moved into the blue subtree. Now we can see the left and right subtrees for the current node. From this we would expect to once again move into the blue subtree.

By using this process, we were able to visually see that we were moving into the correct subtrees.

<u>Discussion of results</u>

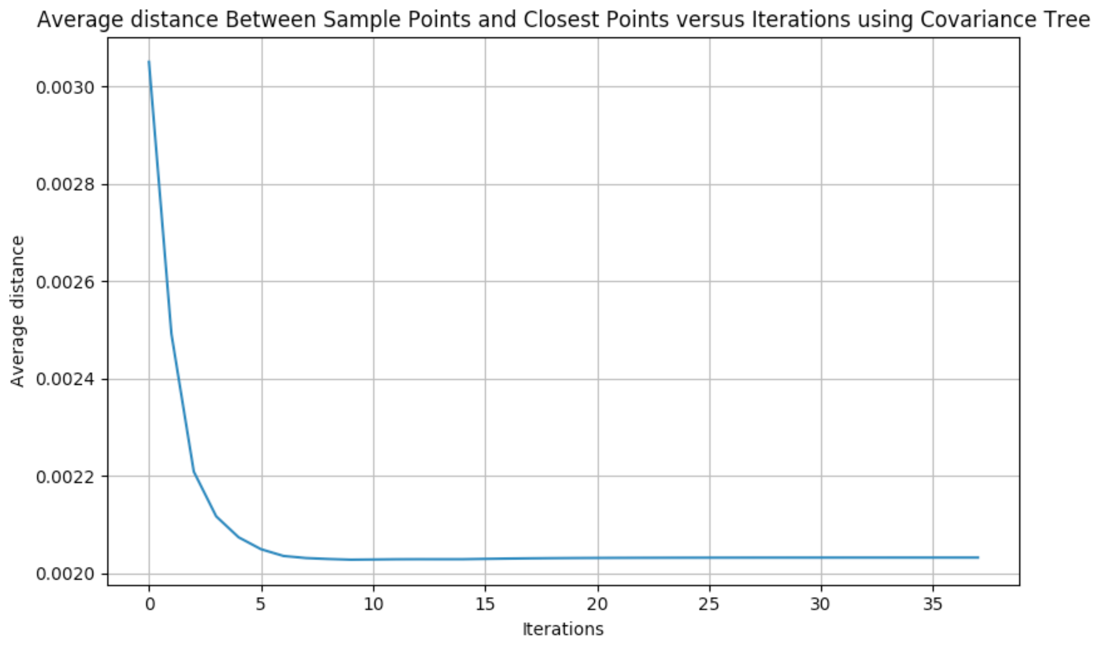
Comparison of brute force approach and covariance tree approach based on running time:

Name of file	Number of sample frames	Brute force method (single iteration)	Covariance Tree method (single iteration)	Covariance Tree method (until convergence)
PA4-A-Debug	75	33.49 seconds	0.37 seconds	10.50 seconds
PA4-F-Debug	200	90.14 seconds	0.69 seconds	33.46 seconds

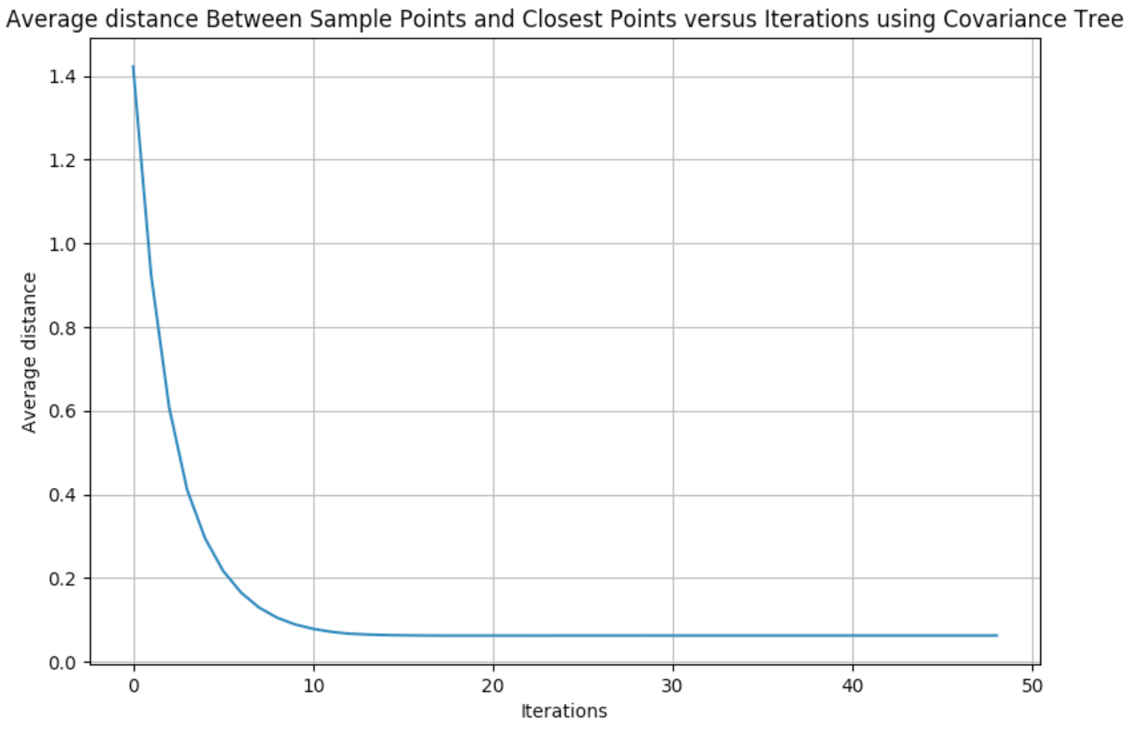
It is obvious that the brute force (Brute force) approach is significantly more time-intensive than the covariance tree method.

Plot of average distance between c_closest and s_k points after each iteration-

PA4-A-Debug:



PA4-F-Debug:



In the two above figures, we see that the distance between c_closest and s_k rapidly drops off. This is because at the start, the transformation F_reg is just a guess. As we iterate, however, the approximation gets better and better, eventually reaching a stable value. The remaining error is on the order of microns for Debug A. For Debug F, where more noise was introduced, we see that the error levels out at a little less than 0.1 mm, which would still be acceptable in most settings.

Finally, in comparing our output to the supplied output files using our compare_output.py script, we found that the difference in c_closest and s_k was negligible.

Who did what

All code was written together. The write up was separated as follows:
Michael - Overview of program and mathematical/algorithmic approach
Mariya - Function descriptions and validation/results

Libraries Used

The python libraries Numpy and Scipy were used in this program. We made use of the Numpy functions like cov, dot, transpose, reshape, and add to easily manipulate our data arrays. From the Scipy library, we used functions from the linalg package, including eig (to compute the eigenvectors and eigenvalues of an array), inv (to find the inverse of an array) and pinv (to find the pseudoinverse of an array). These libraries were used to simplify the tedious math steps without solving the actual problem.

References

1. Srpuyt, "A geometric interpretation of the covariance matrix", 2014. http://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/
2. https://en.wikipedia.org/wiki/Rotation_matrix

How to run

See README.