CIS PA3

Arijit Nukala and Ilana Chalom
anukala1, ichalom1

November 2022

1 Cartesian Math Package

For our program, we used native Python functions and various libraries—namely Numpy, Pandas, and time— to help perform vector/matrix mathematics, frame transformations, rotations, and ICP. For performing our ICP algorithms, we were able to use a combination of these libraries, the mathematical functions we wrote in PA 1, the mathematical functions from PA 3, and our newly written functions to compute the answers. For instance, we used our frame class and registration algorithm from the first programming assignment in writing our ICP algorithm. As always, we relied on the homework instructions and the ways we learned to approach this algorithm in class to complete the assignment.

2 Algorithmic and Mathematical Approach

In the following subsections, we will break down our approach to our complex computations first algorithmically, then it mathematically, and finally we will show how it is reflected in the code. The explanation for our ICP algorithm (iterative), Find Closest Point (on surface mesh), Linear ICP algorithm, Project On Segment computation, Efficient ICP (using Covariance trees) algorithm, Computing the covariance Frame, Registration Algorithm, Construct Subtrees and Split Sort Algorithm, Transformation computation, and Composition computation are all below. Our linear, or simple ICP, will cover the behavior for all of the functions in icp.py, except for project_on_segment(), which is covered in section 2.3 of this report.

2.1 Iterative Closest Point

Our ICP approach can be explained algorithmically by the following:

2.1.1 Algorithmic Approach

1. Find the closest points on the mesh using the previous closest points (using a tree search with the threshold as the previous match).
2. Compute the registration between d_ks, the tip in tracker coordinates, and c_ks, the closest points on A to B, below the threshold
3. Calculate the error statistics, σ_n, $\bar{\epsilon}_n$, and $(\epsilon_n)_{max}$ for the current frame.
4. Adjust the threshold. First set the threshold to $3\bar{\epsilon}_n$, and if the number of valid matched points drops significantly, increase the threshold.
5. Check termination conditions and terminate, or go back to the start. As we discussed in class, the conditions for termination are not necessarily concrete. We drew inspiration from the lecture slides, and stopped our iterations when σ_n, $\bar{\epsilon}_n$, and $(\epsilon_n)_{max}$ are less than the desired threshold (0.005, 0.001, and 0.001 respectively). We also check that $\gamma \leq \frac{\bar{\epsilon}_n}{\epsilon_{n-1}} \leq 1$ over the course the course of several iterations, where γ is set to 0.95. Of course, we also do not iterate past a maximum number of iterations.

2.1.2 Mathematical Approach

Mathematically speaking, the approach looked like this:

1. c_ks = FindClosestPoint
2. F_{reg} = registration(A, B), explained in section 2.9.

3. Calculate the error by:

$$s_{ks} = F_{reg} \cdot d_{ks}$$

$$\sigma_n = \frac{\Sigma_k \vec{e}_k \cdot \vec{e}_k}{numElements(E)}, \text{ where } E = \{..., \vec{e}_k, ...\}, \vec{e}_k = \vec{c}_{ks} - \vec{s}_{ks}$$

$$(\epsilon_n)_{max} = max_k(\sqrt{\vec{e}_k \cdot \vec{e}_k})$$

$$\bar{\epsilon}_n = \frac{\Sigma_k \sqrt{\vec{e}_k \cdot \vec{e}_k}}{numElements(E)}$$

4. The last two steps of the algorithm are conditionals discussed above in the algorithmic approach and will be displayed in the code in the programming approach, but cannot be displayed as easily mathematically.

2.1.3 Programming Approach

Reflected in our code, this looks like: Steps 1 and 2:

```
s_ks = F_reg.compose_transform(d_ks)
prev_A = len(A)
prev_B = len(B)
# find the closest points
new_c_ks, A, B = match_points(d_ks, c_ks, s_ks, cov_tree, thresh)
c_ks = new_c_ks
# compute registration between d_ks and c_ks below the threshold
F_reg = registration(A, B)
```

Step 3 is

```
eps_n_v, e_max_v, sig_n_v = compute_error_stats(F_reg, d_ks, c_ks)
```

where compute_error_stats is defined as:

```
s_ks = F_reg.compose_transform(d_ks)
# compute the error between the tip and the surface mesh
for i in range(len(s_ks)):
    E.append((c_ks[i] - s_ks[i]))
E = np.array(E)
dot_E = []
for i in range(len(E)):
    dot_E.append(np.dot(E[i], E[i]))
for i in range(len(E)):
    eps_n += dot_E[i] ** 0.5
    sig_n += dot_E[i]
# compute the error statistics
eps_n = eps_n / len(E)
e_max = np.max(dot_E) ** 0.5
sig_n = sig_n ** 0.5 / len(E)
return eps_n, e_max, sig_n
```

Steps 4 and 5 are:

```
thresh = 3 * eps_n[-1]
if (len(A) < 0.8 * prev_A) and (len(B) < 0.8 * prev_B):
    thresh = 15 * eps_n[-1]

if eps_n[-1] < 0.005 and e_max[-1] < 0.01 and sig_n[-1] < 0.001:
    break

if n_iter > 1:
    change = eps_n[-1]/eps_n[-2]
    if 0.95 < change < 1:
        term_stack.append(change)
    elif len(term_stack) > 1:
        term_stack.pop()
    if len(term_stack) > 10:
        break
```

2.2 Linear Iterative Closest Point

Our Linear ICP approach/finding the closest point can be explained algorithmically by the following:

2.2.1 Algorithmic Approach

1. Perform a registration to calculate the poses of $F_{A,k}$ and $F_{B,k}$ (the registration algorithm and mathematical approach are explained in the registration section). Using the newly computed poses, calculate $\vec{d_k}$ to find the position of the pointer tip with respect to rigid body B.
2. After finding a $\vec{d_k}$ for each frame, find sample points estimated to be on the surface mesh, $\vec{s_k}$, such that $\vec{s_k} = F_{reg} \cdot \vec{d_k}$
3. Find points $\vec{c_k}$ on the surface mesh that are closest to $\vec{s_k}$. We essentially find the closest point to the triangle by performing a least squares operation (and projecting our point onto the plane of the triangle if it is not).
4. Our last step is to use this computed point to find the distance from the point on the triangle to the mesh. If this distance is less than our bound, it is the closest distance.

2.2.2 Mathematical Approach

Mathematically, this approach looks like :

1. Using $F_{A,k}$ and $F_{B,k}$ the rigid body poses of A and B respectively, we compute $\vec{d_k}$, such that:

   $$\vec{d_k} = F_{B,k}^{-1} \cdot F_{A,k} \cdot \vec{A}_{tip}$$

2. Compute sample points $\vec{s_k}$ such that:

   $$\vec{s_k} = F_{reg} \cdot \vec{d_k}$$

   , where $F_{reg} = I$

3. Find the points $\vec{c_k}$ on the surface mesh that are closest to s_k, where $\vec{c_k} = F_{reg} \cdot \vec{d_k}$. This will look something like: Perform a least squares operation to solve for μ and λ in the equation

   $$\vec{a} - \vec{p} = \lambda(\vec{q} - \vec{p}) + \mu(\vec{r} - \vec{p})$$

   , where $\vec{p}$, $\vec{q}$, and $\vec{r}$ are the vertices of the triangle and a contains the sample points. Compute $\vec{c}$ such that:

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

   The results of λ and μ will determine where $\vec{c}$ lies in the triangle, so if $\mu \geq 0$, $\lambda \geq 0$, and $\mu + \lambda \leq 1$, $\vec{c}$ is in the triangle. Otherwise, find a point on the border (the approach to this is explained in the Project onto Segment section).

4. The last step to finding $\vec{c_k}$ is to check if $||\vec{c_k} - \vec{a}|| \leq$ bound . If it is, then bound $= ||\vec{c_k} - \vec{a}||$, and $\vec{c_k}$ is the closest point.

2.2.3 Programming Approach

These same steps are reflected in the code by:

1.
   ```
   d_ks = find_rigid_body_pose(a_read, b_read, a_tip, a_leds, b_leds)
   ```

   , where the function find_rigid_body_pose is :

   ```
   Nf = len(a_frames)

   # initialize the array to store the tip coordinates
   d_k_cloud = np.zeros((Nf, 3))
   # loop through the frames
   for i in range(Nf):
       # find the rigid body poses
       F_ak = registration(a_leds, a_frames[i])
       F_bk = registration(b_leds, b_frames[i])
   ```

```
        # calculate the pointer tip location
        d_k = Frame.compose_transform(F_bk.invert(),
                Frame.compose_transform(F_ak, a_tip))
        d_k_cloud[i] = d_k
    return d_k_cloud
```

2.
```
    for i in range(len(a_read)):
        F_reg = Frame(np.identity(3), np.zeros(3))
        s_k = find_sample_points(F_reg, d_ks[i])
```

where find_sample_points is defined as:

```
    sample_points = Frame.compose_transform(F_reg, d_k)
    return sample_points
```

3.
```
        c_ks[i] = (find_closest_point(vertices, indices, s_k).reshape(1, 3))
```

where find_closest_point does

```
    for i in range(len(indices)):
        cur_c_k = find_closest_point_triangle(mesh_vertices[indices[i]], s_k)
        cur_d_k = find_euclidian_distance(cur_c_k, s_k)
        if cur_d_k < d_min:
            d_min = cur_d_k
            c_min = cur_c_k
    return c_min
```

and where find_closest_point_triangle computes:

```
    p, q, r = vertices
    A_minus_p = s_k - vertices[0]
    B = np.vstack(((q - p), (r - p))).T
    lam, mu = np.linalg.lstsq(B, A_minus_p.T, rcond=None)[0]
    c = p + lam * (q - p) + mu * (r - p)

    if lam < 0:
        c = project_on_segment(c, r, p)
    elif mu < 0:
        c = project_on_segment(c, p, q)
    elif lam + mu > 1:
        c = project_on_segment(c, q, r)
    return c
```

and project_on_segment does:

```
    if np.linalg.norm(p - q) == 0:
        return p

    t = np.dot(c - p, q - p) / np.dot(q - p, q - p)
    t = np.clip(t, 0, 1)
    return p + t * (q - p)
```

2.3 Project Onto Segment

Our Project Onto Segment computation is sometimes performed after finding the closest point on the triangle. If the closest point is out of bounds (not in the plane of the triangle), we must find a point on the border of the triangle instead. This section of the report covers the behavior in the function project_on_segment from icp.py.

2.3.1 Algorithmic Approach

Algorithmically, the approach looks like:

1. Perform an orthogonal projection where we project $\vec{c}$ onto the plane of the triangle by projecting $\vec{c}$ onto the bounding edge of the triangle between two of the vertices.
2. After computing the projection, λ, confine to the bounds between [0,1] (essentially, if $\lambda \leq 0$, set it to 0, if $\lambda \geq 1$, set it to 1.
3. To finally get our correct closest point on the plane of the triangle, add λ, our corrected distance, to the vertex $\vec{p}$, and multiply by the bounding edge of the triangle, $\vec{q} - \vec{p}$.

2.3.2 Mathematical Approach

Mathematically speaking, the following steps are taken:

1. $$\lambda = \frac{(\vec{c} - \vec{p}) \cdot (\vec{q} - \vec{p})}{(\vec{q} - \vec{p}) \cdot (\vec{q} - \vec{p})})$$
2. $$\lambda^* = Max(0, Min(\lambda, 1))$$
3. Finally, we compute our $\vec{c}$, $$\vec{c} = \vec{p} + \lambda^* \times (\vec{q} - \vec{p})$$

2.3.3 Programming Approach

This is reflected in the code as:

```
def project_on_segment(c: np.ndarray, p: np.ndarray, q: np.ndarray):

if np.linalg.norm(p - q) == 0:
    return p

t = np.dot(c - p, q - p) / np.dot(q - p, q - p)
t = np.clip(t, 0, 1)
return p + t * (q - p)
```

2.4 Compute Covariance Frame

For our computation of the Covariance Frame in the Covariance Tree, we followed Dr. Taylor's "Finding Point-Pairs" slide to construct a Covariance Tree of Thing objects and used a modification of his algorithm to find the Frame.

2.4.1 Algorithmic Approach

The algorithmic approach is derived from the pseudocode on the "Finding Point-Pairs" slides. We compute the covariance frame using an approach virtually identical to the registration algorithm. We chose not to use the eigenvector/eigenvalue approach given on the slides because this approach is mathematically the same and allows for correction of the rotation if the determinant is negative. The approach is shown below:

1. Compute the mean of the corners of the triangles in the Covariance Tree
2. Calculate a matrix A by summing the outer products of the centered value of each corner
3. Find the SVD of A
4. Use the results of the SVD, specifically the V matrix to calulate the rotation matrix
5. Calculate the determinant of rotation: it should be 1, or correct the matrix if the determinant is negative one
6. Set the translation component of the frame to the mean of the corners (centroid)
7.

2.4.2 Mathematical Approach

Mathematically, the algorithmic approach can be represented as:

1. $centroid = corners - \bar{corners}$
2. $points = corners - centroid$

   $A = \sum_{i=1}^{N} \langle points, points \rangle$
3. $A = U \Sigma V^T$
4. Steps 4 and 5 of the algorithm can be combined into:

   $R = VV$

   where R is our Rotation matrix, assuming the determinant is 1
5. $p = centroid$

   where p is our translation component.

2.4.3 Programming Approach

Computing the covariance frame is reflected in our code in the function compute_cov_frame

1.
```
    def compute_cov_frame(self, ts: np.ndarray):

        points = np.array([ts[i].sort_point() for i in range(len(ts))])
        n_p = len(points)
        centroid = np.mean(points, axis=0)
```

2.
```
     for i in range(n_p):
            A += np.outer(points[i] - centroid, points[i] - centroid)
```

3.
```
        u, s, vt = np.linalg.svd(A)
        u = u.T
        vt = vt.T
```

4.
```
     comp_size = vt.shape[0]
        reflection_comp = np.eye(comp_size)
        reflection_comp[comp_size - 1][comp_size - 1] = np.linalg.det(np.dot(vt, u))

        R = np.dot(vt, np.dot(reflection_comp, vt))
        p = centroid

        return Frame(R, p)
```

2.5 Construct Subtrees and Split Sort

For our construction and sorting of the subtrees in the Covariance Tree, we followed Dr. Taylor's "Finding Point-Pairs" slide to construct a Covariance Tree of Thing objects.

2.5.1 Algorithmic Approach

The algorithmic approach to solving the subtree problem is summarized below:

1. If the number of triangles or the size of the bounds is less than our accepted limits, we do not construct further subtrees
2. We split the points in the Node into subtrees by putting each corner from the triangle into the frame of the node. If the x value is below zero, we add it to the left tree and otherwise, we add to the right tree.
3. We construct new subtrees from these subtrees in a recursive loop, ending the loop if the size of either the left or right tree is the same size as the number of points in the node (indicating that the subtrees are no longer splitting the node)

2.5.2 Mathematical Approach

Our mathematical approach for the construction and sorting of subtrees is as follows considering we have not reached the exit conditions described above:

1. Convert each corner of a triangle into the frame of the node.

$$v_x = (F_{node}^{-1} * [x, y, z]_{corner}).x$$

2. Add to the left or right subtree and return the subtrees

2.5.3 Programming Approach

Subtree construction and sorting is reflected in our code in the file cov_tree.py. Specifically, in the functions construct_subtrees and split_sort

1.
```
if n_t <= min_count or np.linalg.norm(self.UB - self.LB) <= min_diag:
    return None, None, False
```

2.
```
left_tree, right_tree = self.split_sort(n_t)
```

   where split_sort is defined by:

```
ts = self.things
for i in range(n_t):
    # transform triangle corners into the frame of the node
    if self.F.invert().compose_transform(ts[i].sort_point().reshape(1, 3))[0][0] <
        left_tree.append(ts[i])
    else:
        right_tree.append(ts[i])
return np.array(left_tree), np.array(right_tree)
```

3.
```
if len(left_tree) == n_t or len(right_tree) == n_t:
    return None, None, False
left, right = CovTreeNode(left_tree, len(left_tree)),
        CovTreeNode(right_tree, len(right_tree))
return left, right, True
```

2.6 Efficient ICP

For our efficient ICP implementation, we followed Dr. Taylor's "Finding Point-Pairs" slide to construct a Covariance Tree of Thing objects to find the closest point.

2.6.1 Algorithmic/Mathematical Approach

I am combining our Algorithmic and Mathematical approach given that much of the algorithm involves if/else statements. I will give the mathematical representation of the statements along with their usage.

1. First, we find the $d_k s$ as described in the Simple ICP Approach
2. We create an array of Thang (Thing) objects, representing the triangles of the surface mesh of the given object
3. We then create a CovTreeNode object using the computation of covariance frame and subtree construction algorithms described earlier. We initialize the previous closest value for this iterative approach to the first vertex in our surface mesh.
4. For the closest point computation, we iterate through each point in our d_k cloud, finding the norm between each point and our previous closest: $\|s - d_k\|$
5. We now find the closest point:

   (a) Find the local frame of the point:

$$v_{local} = F_{node}^{-}1 * v$$

(b) Ensure v_local is in the bounds defined by the covariance tree

$$LB < vlocal < UB$$

(c) If the node has subtrees and $v_{local}.xis < -bound$, search the left subtree recursive for a closest point. Else, search the right.

(d) If the node does not have subtrees, complete a linear search along the nodes points, updating the new closest point

2.6.2 Programming approach

Our code has the efficient ICP algorithm in pa_three.py in our efficient_icp, which calls the method find_closest_point in cov_tree.py. These two methods combined perform the efficient ICP algorithm. This looks like:

1.
   ```
       d_ks = icp.find_rigid_body_pose(a_read, b_read, a_tip, a_leds, b_leds)
   ```

2.
   ```
   ts = np.array([thang.Thang(vertices[indices[i]]) for i in
   range(len(indices))])
   ```

3.
   ```
   root = ct.CovTreeNode(ts, len(ts))
   previous_closest = ts[0].corners[0]
   ```

4.
   ```
       for _, s in enumerate(d_ks):
       bound = np.linalg.norm(s - previous_closest)
       closest.append(root.find_closest_point(s, bound, previous_closest))
   ```

5.
   ```
        previous_closest = closest[-1]
   c_ks = np.array(closest)

   mag_dif = icp.find_euclidian_distance(c_ks, d_ks)
   ```

where find_closest_point is defined as:

```
    v_local = self.F.invert().compose_transform(v.reshape(1, 3))
    for i in range(3):
        if v_local[0, i] > self.UB[0, i] + bound or v_local[0, i]
             < self.LB[0, i] - bound:
            return
    if self.have_subtrees:
        if v_local[0, 0] < -bound:
            return self.left.find_closest_point(v, bound, closest)
        elif v_local[0, 0] > bound:
            return self.right.find_closest_point(v, bound, closest)
        else:
            left = self.left.find_closest_point(v, bound, closest)
            right = self.right.find_closest_point(v, bound, closest)
            if left is not None and right is None:
                return left
            elif left is None and right is not None:
                return right
            elif left is None and right is None:
                return closest
            else:
                return min(left, right, key=lambda x: np.linalg.norm(x - v))
    else:
        for i in range(self.n_things):
            bound, closest = self.update_closest(self.things[i], v, bound, closest)
        return closest
```

2.7 Registration algorithm

For our registration algorithm, we used an approach developed by Arun et al., in which the authors explored a non-iterative algorithm which employs Singular Value Decomposition. Additionally, we compensated for rotation matrices with a determinant of negative one using an approach described by Sorkine-Hornung et al. This explanation will cover the behavior of the registration.py file.

2.7.1 Algorithmic Approach

The algorithmic approach using can be broken down into roughly five or so steps:

1. Center the point sets
2. Calculate a matrix H by multiplying one matrix by the transpose of the other
3. Find the SVD of H
4. Use the results of the SVD to calculate a matrix X
5. Calculate the determinant of X: it should be 1, or correct the matrix if the determinant is negative one

2.7.2 Mathematical Approach

Mathematically speaking, the function would look like

1. $a_i = a - \bar{a}$
   $b_i = b - \bar{b}$
2. $H = \sum_{i=1}^{N} a_i b_i^T$
3. $H = U \Sigma V^T$
4. Steps 4 and 5 of the algorithm can be combined into:
   $X = VU^T$
   where X is our Rotation matrix, assuming the determinant is 1

2.7.3 Programming Approach

Reflected in the code, this looked like:

1.
```
a_mean = np.mean(A.points, axis=1, keepdims=True)
b_mean = np.mean(B.points, axis=1, keepdims=True)
centered_a = A.points - a_mean
centered_b = B.points - b_mean
```
2.
```
H = np.dot(centered_a, centered_b.transpose())
```
3.
```
u, s, vt = np.linalg.svd(H)
```
4.
```
 u = u.transpose()
 vt = vt.transpose()
```
5.
```
comp_size = vt.shape[0]
reflection_comp = np.eye(comp_size)
reflection_comp[comp_size - 1][comp_size - 1] =
np.linalg.det(np.dot(vt, u))
```
6. And with the newly found rotation matrix, we returned a Frame:
```
R = np.dot(vt, np.dot(reflection_comp, u))
p = b_mean - np.dot(R, a_mean)
return Frame(R, p)
```

2.8 Transformation

Our frame transformation function, compose_transform, in frame.py looked something like:

2.8.1 Algorithmic Approach

1. Iterate through each row in the rotation matrix of the frame, R, and compute the dot product of that vector with the points to transform and add to it the translation vector, p.

2.8.2 Mathematical Approach

Mathematically, this looks like:

1. $v = F \cdot b$
   $v = [R, p] \cdot b$
   $v = R \cdot b + p$

2.8.3 Programming Approach

Reflected in the code, this likes like:

1.
```
        for i in range(frame_size):
            t_points[i] = np.dot(self.R, points[i]) + self.p
```

2.9 Composition

Our frame composition function, compose_frame, in frame.py looks something like:

2.9.1 Algorithmic Approach

1. Compute the dot product between Frame 1's rotation Matrix and Frame 2's rotation matrix
2. Compute the dot product between Frame 1's rotation Matrix and Frame 2's translation vector. Add to it Frame 1's translation vector

2.9.2 Mathematical Approach

Mathematically speaking, this looks like:

1. $F_1 \cdot F_2 = [R_1, p_1] \cdot [R_2, p_2]$
   $F_1 \cdot F_2 = [R_1 \cdot R_2, R_1 p_2 + p_1]$

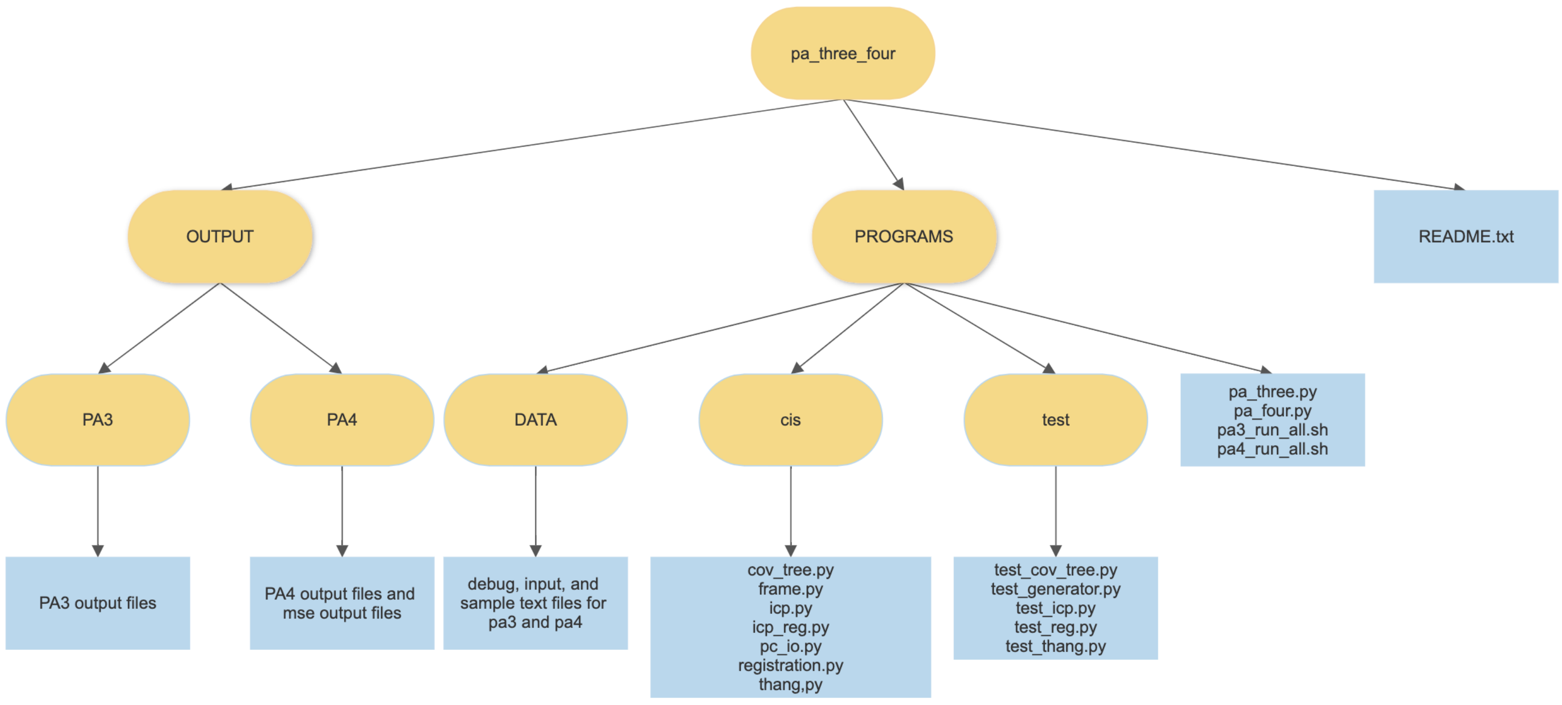
2.9.3 Programming Approach

This was represented in the code with the lines:

1.
```
        mat = np.dot(self.R, other_frame.R)
```
2.
```
        vec = np.dot(self.R, other_frame.p) + self.p
```

3 Program Structure

Our program structure is broken down into two main sections: the programs and input files, located in the PROGRAMS folder, and outputs, located in the OUTPUTS folder. Inside PROGRAMS, is our README.txt, main method for programming assignment 3, pa_three.py, and our main method for programming assignment 4, pa_four.py. We also have a folder containing all of the input files, called DATA, and we have a folder containing all of our test files, called test. Lastly, in PROGRAMS there is a folder titled cis; this is where the bulk of our program lies. cov_tree.py, frame.py, icp.py, icp_reg.py, pc_io.py, registration.py, and thang.py. The overall structure of our program can be seen in the diagram below, and is further elaborated following the diagram.


pa_three_four
OUTPUT
PROGRAMS
README.txt
PA3
PA4
DATA
cis
test
pa_three.py
pa_four.py
pa3_run_all.sh
pa4_run_all.sh
PA3 output files
PA4 output files and mse output files
debug, input, and sample text files for pa3 and pa4
cov_tree.py
frame.py
icp.py
icp_reg.py
pc_io.py
registration.py
thang,py
test_cov_tree.py
test_generator.py
test_icp.py
test_reg.py
test_thang.py

3.1 cov_tree.py

This file contains the CovTreeNode class which is the class we used to represent Covariance Tree Functions. This class has the following fields:

1. things: a 3x3 matrix used to represent the triangles on the mesh
2. n_things: an integer used to represent the number of things
3. F : the frame of the covariance tree node
4. UB: The upper bound of the covariance tree node
5. LB: The lower bound of the covariance tree node
6. left: The left subtree of the covariance tree node
7. right: The right subtree of the covariance tree node
8. have_subtrees: A boolean representing whether or not the covariance tree node has subtrees

In this file there is the constructor and methods compute_cov_bounds, compute_cov_frame, construct_subtrees, split_sort, find_closest_point, and update_closest.

1. The constructor takes in parameters ts, representing the triangles of the covariance tree node, and n_t, the number of triangles in the covariance tree node. It initializes the fields above, and does with help from the methods compute_cov_frame and construct_subtrees.
2. compute_cov_bounds is the method for computing the covariance bounds of a node. It takes in parameters ts, representing the triangles of the covariance tree node, and n_t, the number of triangles in the covariance tree node, and returns UB, the upper bound of the covariance tree node, and LB, the lower bound of the covariance tree node. To compute the minimum and maximum bounds of the triangle points, we iterate through the triangles and call the method enlarge_bounds.
3. compute_cov_frame is a method for computing the covariance frame of a node. It takes in the parameter ts, the triangles of the covariance tree node. To find the frame, we compute a covariance matrix A by centering the triangles. We then perform SVD resulting in vectors $\vec{u}$ and $\vec{v}\_t$, and validate the result by checking the determinant to account for any potential issues. Lastly, we return our newly computed frame.
4. construct_subtrees is a method for constructing the subtrees of a node. It takes in parameters, n_t, the number of triangles in the covariance tree node, min_count, the minimum number of triangles in a subtree, and min_diag, the minimum diagonal of a subtree. This function then returns the left and right subtrees of the covariance tree node.
5. split_sort is a method for dividing the triangles of a node into two subtrees. Then transforming the triangle corners into the frame of the node, we return the left and right subtrees.
6. find_closest_point is the method for finding the closest point to a given point where the parameters are v, the point to find the closest point to, bound, the current closest distance, and closest, the current closest point.
7. update_closest is the method for updating the closest point to a given point.

3.2 frame.py

this file was created in programming assignment 1 and contains the frame class. The functions of the class are compose_frame, compose_transform, and invert.

1. compose_frame is called on a frame and takes in another frame as the additional argument. It then computes a frame composition and returns the final frame.

2. compose_transform is also called on a frame and takes in a point set as the additional argument. The function computes a frame transform and returns the corresponding points.

3. invert is called on a frame and performs the frame inversion calculations. The final resulting frame is returned.

3.3 icp.py

this file contains the method (and corresponding helper methods) to our ICP algorithm. We have find_rigid_body_pose, find_sample_points, find_closest_point, find_closest_point_triangle, project_on_segment, find_euclidian_distance, and, of course, ICP_linear.

1. find_rigid_body_pose is, as the name suggests, the method for finding the rigid body pose given arguments a_frames, the xyz coordinates of A,,m,,,, body LED markers in tracker coordinates, b_frames, the xyz coordinates of B body LED markers in tracker coordinates, a_tip, the xyz coordinates of the tip in tracker coordinates, a_leds, the xyz coordinates of A body LED markers in body coordinates, and b_leds, the xyz coordinates of B body LED markers in body coordinates. After iterating through the frames and finding the pose, we return d_k_cloud, the xyz coordinates of the pointer tip with respect to rigid body B. This method calls the registration method and the compose_transform method.

2. find_sample_points is a method for finding sample points to match to the surface mesh given F_reg, the frame transformation of the surface mesh from the pointer tip, and d_k, the xyz coordinates of the tip with respect to rigid body B. This method returns sample_points, which are the xyz coordinates of sample points estimated to be on the surface mesh. This method calls the compose_transform method.

3. find_closest_point is a method for finding the closest point on the surface mesh. It takes in parameters mesh_vertices, the xyz coordinates of the vertices of the surface mesh, indices, the indices of the vertices of the surface mesh, and s_k, the xyz coordinates of the sample points. After calculating the closest point, this method returns c_min, the xyz coordinates of the closest point on the surface mesh. This method also calls the methods find_closest_point_triangle and find_euclidian_distance.

4. find_closest_point_triangle is the method for finding the closest point on a triangle given arguments vertices, the xyz coordinates of the vertices of the triangle, and s_k, the xyz coordinates of the sample points. It returns c the xyz coordinates of the closest point on the triangle. This method also calls the helper method project_on_segment.

5. project_on_segment is the method for projecting a point on a segment given arguments c, the xyz coordinates of the point to be projected, p, the xyz coordinates of the first point on the segment, and q, the xyz coordinates of the second point on the segment. This then returns the xyz coordinates of the projected point.

6. find_euclidian_distance is the method for finding the euclidian distance between the sample points and the surface mesh given c_k, the xyz coordinates on the surface mesh found from F_reg * d_k, and d_k, the xyz coordinates of the tip with respect to rigid body B. This then returns mag_dif, the euclidian distance between the sample points and the surface mesh.

7. ICP_linear is our method for finding the rigid body pose using ICP. It takes in mesh_vertices, the xyz coordinates of the vertices of the surface mesh, indices, the indices of the vertices of the surface mesh, a_frames, the xyz coordinates of A body LED markers in tracker coordinates, b_frames, the xyz coordinates of B body LED markers in tracker coordinates, a_tip, the xyz coordinates of the tip in tracker coordinates, a_leds, the xyz coordinates of A body LED markers in body coordinates, b_leds, the xyz coordinates of B body LED markers in body coordinates. This then returns the xyz coordinates of the pointer tip with respect to rigid body B. This method makes calls to helper methods find_rigid_body_pose, find_sample_points, find_closest_point, and find_euclidian_distance.

3.4 icp_reg.py

This file contains all of the methods to perform an iterative ICP algorithm for PA4: icp, match_points, and compute_error_stats.

1. icp is the method for finding the rigid body pose using ICP (iteratively). It takes in the parameters a_read, the xyz coordinates of A body LED markers in tracker coordinates, b_read, the xyz coordinates of B body LED markers in tracker coordinates, a_tip, the xyz coordinates of the tip in tracker coordinates, a_leds the xyz coordinates of A body LED markers in body coordinates, b_leds, the xyz coordinates of B body LED markers in body coordinates, vertices, the xyz coordinates of the vertices of the surface mesh, indices, the indices of the vertices of the surface mesh, and max_iter, the maximum number of iterations. It eventually returns the xyz coordinates of the pointer tip with respect to rigid body B. This function calls helper functions find_rigid_body_pose (from icp.py), compose_transform, match_points, registration, and compute_error_stats.

2. match_points is the method for finding the closest points on the mesh using the previous closest points. It takes in parameters d_ks, he xyz coordinates of the tip in tracker coordinates, c_ks, the estimated xyz coordinates of the vertices of the surface mesh, s_ks, the xyz coordinates of the tip in body coordinates, root, the root node of the covariance tree, and threshold, the threshold for the distance between the closest points. It then returns the updated closest points. This method calls the function find_closest_points (from cov_tree.py).

3. compute_error_stats is the method for computing the error statistics. It takes in the arguments F_reg, the rigid body transformation between the tip and the surface mesh, d_ks, the xyz coordinates of the tip in tracker coordinates, and c_ks, the estimated xyz coordinates of the vertices of the surface mesh. It returns the error statistics σ_n, $\bar{\epsilon}_n$, and $(\epsilon_n)_{max}$. This method calls the compose_transform helper function.

3.5 pc_io.py

This file contains all of the methods to read from the input files and to write to the output files. We have the methods import_rigid_body, import_surface_mesh, import_sample_readings, output_pa34, read_answer_pa3, and read_answer_pa4.

1. import_rigid_body is the method for importing the rigid body design data. It takes in the parameter fName, which is the name of the data file, and, after reading the file, returns the point clouds representing the xyz coordinates of the marker LEDs in body coordinates, and the xyz coordinate of the tip in body coordinates.

2. import_surface_mesh is our method for importing body surface definition data. It also takes in the argument fName, which is the name of the data file, and after reading the file, returns the point clouds representing the xyz coordinates of vertices in CT coordinates and the xyz coordinates of the triangle indices.

3. import_sample_readings is our method for importing sample readings. It takes in the arguments fName, the name of the data file, Na, the number of A markers, and Nb, the number of B markers. After reading the file, it returns the point clouds representing frames of xyz coordinates of A body LED markers, B body LED markers, and D (unneeded) body LED markers.

4. output_pa34 is our method for outputting PA34 data, and takes in parameters: output_dir, the directory to output the data, name, the name of the data output file, cs, the xyz coordinates on the surface mesh found from F_reg * d_k, ds, the xyz coordinates of the tip with respect to rigid body B, and mag_dif, the magnitude of the difference between the tip in CT coordinates and the tip in DCS coordinates.

5. read_answer_pa3 is our method for reading the answer file for PA3. It takes in fname, the path of the answer file, and returns point clouds representing the xyz coordinates of the tip in CT coordinates.

6. read_answer_pa4 is our method for reading the answer file for PA4. It takes in fname, the path of the answer file, and returns point clouds representing the xyz coordinates of the tip in CT coordinates.

3.6 registration.py

This file contains the registration function for our registration algorithm from assignment 1, registration().

1. registration is our method performing a non-Iterative registration, employing Arun, Huang, and Blostein's algorithm. It takes in arguments A and B, the point cloud and the point cloud to be mapped to, respectively, and returns Frame, the point cloud transformation for the two point cloud inputs.

3.7 thang.py

This file contains the thang class, which is the class used to represent triangles in 3D space. This class has the field corners, a 3x3 matrix representation of the corners of the triangle. In addition to the constructor, it has the methods, sort_point, closest_point_to, enlarge_bounds, bounding_box, and may_be_in_bounds.

1. sort_point is the method for sorting the points of the triangle. Taking no parameters, this function returns one corner of the triangle.

2. closest_point_to is the method for finding the closest point on the triangle to a given point. It takes in the parameter point, which is the point to find the closest point on the triangle to. It calls the function find_closest_point_triangle from icp.py, and returns the resulting closest point on the triangle to the given point.

3. enlarge_bounds is a method for finding the bounding box of the triangle. It takes the parameters frame, the frame to be composed with, LB, the lower bound of the bounding box, and UB, the upper bound of the bounding box. This function then returns LB, the lower bound of the bounding box, and UB, the upper bound of the bounding box. It also calls the functions invert and compose_transform.

4. bounding_box is the method for finding the bounding box of the triangle. It takes in frame, the frame to be composed with, as an argument. It calls the function enlarge_bounds and then returns LB, the lower bound of the bounding box, and UB, the upper bound of the bounding box.

5. may_be_in_bounds is the method for checking if the triangle is in the bounding box. It takes in parameters frame, the frame to be composed with, LB, the lower bound of the bounding box, and UB, the upper bound of the bounding box. It calls the functions invert and compose_transform and eventually returns a boolean representing whether the triangle is in the bounding box.

3.8 pa_three.py

This file is our file that essentially runs the program. It contains our main method, the method simple_ICP, and the method efficient_ICP.

1. main is our main method for PA3 that runs our simple and efficient ICP algorithms. It takes in the parameters data_dir, the directory of the data files, sample_readings_type, the name of the sample readings file, output_dir, the directory to output the data, and name, the name of the data output file. This main method calls our other methods from pc_io.py, import_surface_mesh, import_sample_readings, and output_pa34, and also calls methods simple_ICP, and efficient_ICP.

2. simple_ICP is our method for performing a simple ICP. It takes in paramters a_read, the readings from the first rigid body, b_read, the readings from the second rigid body, a_tip, the tip of the first rigid body, a_leds, the LEDs of the first rigid body, b_leds, the LEDs of the second rigid body, vertices, the vertices of the surface mesh, and indices, the indices of the surface mesh. This function calls the ICP_linear function from icp.py, and returns the resulting d_ks, the points on the surface mesh, c_ks, the points on the rigid body, and mag_dif, the magnitude of the difference between the points.

3. efficient_ICP is our method for performing an efficient ICP. It takes in paramters a_read, the readings from the first rigid body, b_read, the readings from the second rigid body, a_tip, the tip of the first rigid body, a_leds, the LEDs of the first rigid body, b_leds, the LEDs of the second rigid body, vertices, the vertices of the surface mesh, and indices, the indices of the surface mesh. This function calls find_rigid_body_pose from icp.py, the thang constructor, the CovTreeNode constructor, find_closest_point from cov_tree.py, and find_euclidian_distance from icp.py. It returns the resulting d_ks, the points on the surface mesh, c_ks, the points on the rigid body, and mag_dif, the magnitude of the difference between the points.

3.9 pa_four.py

This file contains our main method for PA4, main, and the method to compute the mean squared error (mse), mse.

1. main is our main method for PA4. It takes in arguments data_dir, the directory of the DATA files, sample_readings_type, the name of the sample readings file, output_dir, the directory to output the data, and name, the name of the data output file. It calls methods import_rigid_body, import_surface_mesh, import_sample_readings, and output_pa34 from pc_io.py. It also calls compose_transform from frame.py, icp, from icp_reg.py, find_euclidean_distance from icp.py, and mse.

2. mse is our method to compute the mean squared error of our output compared to the given output. It takes the parameters name, the name of the data output file, sample_readings_type, the type of sample readings, output_dir, the directory to output the data, c_ks, the computed c_ks from ICP, and icp_time, the time it took to run ICP. It calls on the helper method read_answer_pa4 from pc_io.py.

4 Verification

We compared our output to the debug output to verify our program. Our results were nearly the same as the debugging output, with occasional micrometer differences that are negligible. In our main method, we added a print statement that computed and logged the MSE between the simple and efficient ICP output coordinates, and the MSE was always 0. Moreover in our output folder we have an MSE output text file for most of the results which show the MSE, which was 0, as well as the length of time it took for our efficient ICP algorithm to compute. In creating our algorithm, we originally had a perfect output every time, however the program took hours to run, and this was because there was a logical bug that caused it to build new subtrees on every iteration. So, even though our output was perfect, the program was inefficient and therefore not usable. We had to compromise a bit on accuracy in order to have a much quicker algorithm. By logging the time it took for our ICP to run, we ensured that our algorithm always finished in less than 2 minutes (closer to a minute and a half). We were able to verify not only accuracy in our computations, but speed as well, which is just as important. We also created unit tests to verify our algorithms.

5 Results

5.1 PA4-A-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-4.77 20.39 13.36	-4.76 20.39 13.36	0.002
-6.13 17.37 12.30	-6.13 17.37 12.29	0.002
-0.25 4.92 -21.88	-0.25 4.91 -21.88	0.003
-30.48 -21.33 -44.17	-30.48 -21.34 -44.18	0.007
⋮	⋮	⋮

5.2 PA4-B-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
6.68 -4.80 63.42	6.68 -4.80 63.42	0.001
-37.19 -17.17 -41.07	-37.19 -17.17 -41.07	0.001
28.12 19.27 12.82	28.12 19.27 12.82	0.001
-32.92 -28.12 -19.42	-32.92 -28.12 -19.43	0.001
⋮	⋮	⋮

5.3 PA4-C-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
3.14 -7.01 52.60	3.14 -7.02 52.60	0.005
30.05 12.94 16.46	30.05 12.94 16.46	0.001
-33.87 -23.65 -13.48	-33.87 -23.65 -13.48	0.001
2.23 -12.95 1.39	2.23 -12.96 1.39	0.004
⋮	⋮	⋮

5.4 PA4-D-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
4.64 10.65 -11.34	4.64 10.65 -11.33	0.002
-33.66 -26.58 -17.16	-33.66 -26.58 -17.16	0.002
3.95 19.38 0.63	3.96 19.38 0.63	0.002
-14.56 6.26 -40.59	-14.56 6.26 -40.59	0.002
$\vdots$	$\vdots$	$\vdots$

5.5 PA4-E-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-1.54 23.96 21.66	-1.53 24.11 21.65	0.154
-16.94 6.21 -13.17	-16.93 6.17 -13.22	0.065
6.56 16.61 -5.96	6.52 16.64 -5.99	0.055
11.10 -6.58 56.98	11.11 -6.58 56.97	0.006
$\vdots$	$\vdots$	$\vdots$

5.6 PA4-F-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-3.42 4.74 -23.90	-3.43 4.72 -23.90	0.021
-16.34 10.68 -34.00	-16.35 10.64 -33.99	0.034
5.51 20.48 43.46	5.51 20.46 43.46	0.020
11.75 25.78 11.49	11.75 25.80 11.49	0.018
$\vdots$	$\vdots$	$\vdots$

5.7 PA4-G-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-31.40 2.95 -13.76	-31.40 2.95 -13.76	0.001
-41.16 -3.38 -17.37	-41.15 -3.38 -17.37	0.002
-8.87 -20.46 -43.13	-8.87 -20.46 -43.13	0.003
-7.55 8.85 -0.39	-7.55 8.86 -0.39	0.005
$\vdots$	$\vdots$	$\vdots$

5.8 PA4-H-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-43.06 -6.12 -30.09	-43.06 -6.12 -30.09	0.001
5.44 23.97 20.28	5.44 23.98 20.28	0.006
-37.60 -14.73 -11.31	-37.60 -14.73 -11.31	0.000
35.47 10.44 -17.03	35.47 10.44 -17.04	0.003
$\vdots$	$\vdots$	$\vdots$

5.9 PA4-J-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
3.09 23.84 11.19	3.15 23.77 11.21	0.095
8.91 20.81 52.54	8.85 20.99 52.56	0.185
-1.36 -11.02 11.50	-1.35 -10.92 11.49	0.098
-2.70 -22.81 -19.86	-2.62 -22.90 -19.83	0.129
⋮	⋮	⋮

5.10 PA4-K-output.txt

s_x, s_y, s_z	c_x, c_y, c_z	$\lvert\vec{s}_k - \vec{c}_k\rvert$
-30.55 9.02 -20.14	-30.56 9.06 -20.13	0.045
-9.10 3.57 25.65	-9.01 3.59 25.62	0.100
34.06 -6.45 -12.94	34.08 -6.46 -12.93	0.025
-7.67 5.64 -30.39	-7.68 5.63 -30.39	0.018
⋮	⋮	⋮

6 Discussion

As described in the Verification section, we added a method to write the MSE of our ICP output coordinates into another output file, as well as the length of time taken to compute, and the MSE was always 0 or close to 0. In addition to this, we added print statements in our main method to check how long our ICP algorithm took. Overall, using a cov_tree proved to be essential in the speed and accuracy of our ICP algorithm. While the first iteration took some time to run since it was searching for an initial set of points that were far from the closest set, it ended up allowing for an overall faster algorithm once the closest points were found and our bounding boxes were constrained. At the beginning of our assignment, we faced an issue where our covariance tree was accurate but took far took long to complete each iteration, mimicking a linear time ICP operation. This issue was not noticed in PA3, as the first iteration for both PA3 and PA4 ran slowly. However, subsequent iterations in PA4 should have been rapid. We found that our find_closest_point method was looking for points outside of the search bounds first, determining whether to search the right tree or the left. We fixed the issue by changing our code to check both the left and right trees and return the closest point found from each subtree. This drastically increased our speed by preventing a search outside of the bounds. As discussed in the verification, compromise between speed and accuracy was required. While we were easily able to create a highly accurate and extremely slow algorithm, or an inaccurate but speedy algorithm, finding a good medium, however, was more challenging.

7 Contributions

Ilana: created the initial files and program structure, wrote the frame.py and registration.py files, helped debug/conceptually work through the ICP algorithm, and worked on README.txt. Wrote the report and program structure diagram.
Arijit: created cov_tree.py, pc_io.py, icp.py, icp_reg.py, thang.py, pa_three.py, pa_four.py, and worked on the README.txt. Also worked on debugging and did the program testing.

8 Sources

1. Arun, K. S., et al. Least Squares Fitting of Two 3-D Point Sets. IEEE, Sept. 1997, www.ece.queensu.ca/people/S-D-Blostein/papers/PAMI-3DLS-1987.pdf.

2. Chintalapani, G., & Taylor, R. H. (2007). C-arm distortion correction using patient CT as a fiducial. In 2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro - Proceedings (pp. 1180-1183). [4193502] (2007 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro - Proceedings). https://doi.org/10.1109/ISBI.2007.357068

3. Florian, et al. "Pytest-Dev/Pytest." GitHub, 13 Oct. 2022, github.com/pytest-dev/pytest/blob/main/CITATION. Accessed 13 Oct. 2022.

4. Gennery, Donald B. "Least-Squares Camera Calibration Including Lens Distortion and Automatic Editing of Calibration Points." California Institute of Technology, 1998.

5. Guillemot, Thierry, et al. "Covariance Trees for 2D and 3D Processing." IEEE Xplore.

6. Harris, Charles R., et al. "Array Programming with NumPy." Nature, vol. 585, no. 7825, 16 Sept. 2020, pp. 357–362, 10.1038/s41586-020-2649-2.

7. "Logging — Logging Facility for Python — Python 3.10.0 Documentation." Docs.python.org, docs.python.org/3/library/logging.html.

8. McKinney, Wes. "Data Structures for Statistical Computing in Python." PROC. OF the 9th PYTHON in SCIENCE CONF, 2010.

9. "Point-Set Registration." Wikipedia, 2 July 2022, en.wikipedia.org/wiki/Point-set_registration. Accessed 13 Oct. 2022.

10. Sorkine-Hornung, Olga, and Michael Rabinovich. Least-Squares Rigid Motion Using SVD. 16 Jan. 2017, igl.ethz.ch/projects/ARAP/svd_rot.pdf. Accessed 13 Oct. 2022.

11. Taylor, Russell. Finding Point-Pairs Lecture Slides. Johns Hopkins University, 2022.

12. "Time — Time Access and Conversions — Python 3.7.2 Documentation." Python.org, 2000, docs.python.org/3/library/time.html.

13. Virtanen, Pauli, et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." Nature Methods, vol. 17, no. 3, 3 Feb. 2020, pp. 261–272, 10.1038/s41592-019-0686-2.

14. "VPI - Vision Programming Interface: Lens Distortion Correction." Docs.nvidia.com, docs.nvidia.com/vpi/algo_ldc.html. Accessed 27 Oct. 2022.

15. "Welcome to Click — Click Documentation (8.1.x)." Click.palletsprojects.com, click.palletsprojects.com/en/8.1.x/.

16. Yaniv, Ziv. "Which Pivot Calibration?" SPIE Proceedings, 18 Mar. 2015, www.yanivresearch.info/writtenMaterial/pivotCalib-SPIE2015.pdf, 10.1117/12.2081348. Accessed 13 Oct. 2022.