16-833: Robot Localization and Mapping, Spring 2021
Homework 4 - ICP

Aditya Ramakrishnan

April 17, 2022

Note: collaborated with Shravan Kumar Gulvadi - sgulvadi

1 Projective data association - *icp.py*

Question 2.1.1 (First filter section)

Filtering out indices that are outside the vertex map requires that the following criteria be met:

$$target\_u \in [0, w); \quad target\_v \in [0, h); \quad target\_ds \geq 0$$

Question 2.1.2 (Second filter section)

We implement a second filter to ensure that projection q is in the within the permitted neighbourhood of the source point p. This permitted neighbourhood is set by a threshold of 0.07 (provided in the source code)
It is important to understand why this thresholding is performed - in order to remove outliers and prevent a large drift in the fusing step. Ideally, if the transformed points are not close to the source points, they can be discarded.

```python
# TODO: first filter: valid projection
mask = np.zeros_like(target_us).astype(bool)
#(mask[target_us<w] & mask[target_us >= 0] & mask[target_vs >= 0] & mask[target_vs < h] & mask[target_ds >= 0]) = TRUE
mask = ((target_us < w) & (target_vs < h) & (target_us >= 0) & (target_vs >= 0) & (target_ds >= 0)).astype(bool)
# End of TODO

source_indices = source_indices[mask]
target_us = target_us[mask]
target_vs = target_vs[mask]
T_source_points = T_source_points[mask]

# TODO: second filter: apply distance threshold
#mask = np.zeros_like(target_us).astype(bool)

q_points = target_vertex_map[target_vs,target_us]
p_points = T_source_points

mask = ((np.linalg.norm((p_points - q_points), axis=1)) < dist_diff).astype(bool)
```

Figure 1: Fig. 1: First and Second filter implementation

2 Linearization - *icp.py*

Question 2.2 (Matrices A_i and b_i)

Here, we are required to linearize the error function. I have performed the linearization by hand as I am not able to resolve formatting issues in my latex (I have made the derivation clear and legible):

Q2.2 Linearization:

Error function can be written by

$$\sum_{i \in \Omega} r_i^2 (R, t) = \| n_{q_i}^T (R p_i + t - q_i) \|^2 \quad \ldots (i)$$

Parametrizing with a small angle assumption `$\delta R$' & `δt'

$$\therefore \quad \delta R = \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} \quad \ldots (ii)$$

$$\therefore \quad \delta t = [t_x, t_y, t_z] \quad \ldots (iii)$$

Additionally, we consider P-prime "p_i'":

$$p_i' = R^0 p_i + t^0$$

Reforming the equation $\Rightarrow$ $r_i(\alpha, \beta, \gamma, t_x, t_y, t_z) = A_i \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ t_x \\ t_y \\ t_z \end{bmatrix} + b_i$

Solving for A_i & b_i

Linearized error function: (from eqn.(ii))

$$\eta_{q_i}^{T} \left[(\delta R)\cdot(p_i') + (\delta t) - q_i \right]$$

Substituting for `δR' from eqn. (ii) and δt' from eqn

$$\eta_{q_i}^{T} \left(\begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} \begin{bmatrix} p_x' \\ p_y' \\ p_z' \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} + \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} \right)$$

$$\because \eta_{q_i}^{T} = [n_1 \;\; n_2 \;\; n_3]$$

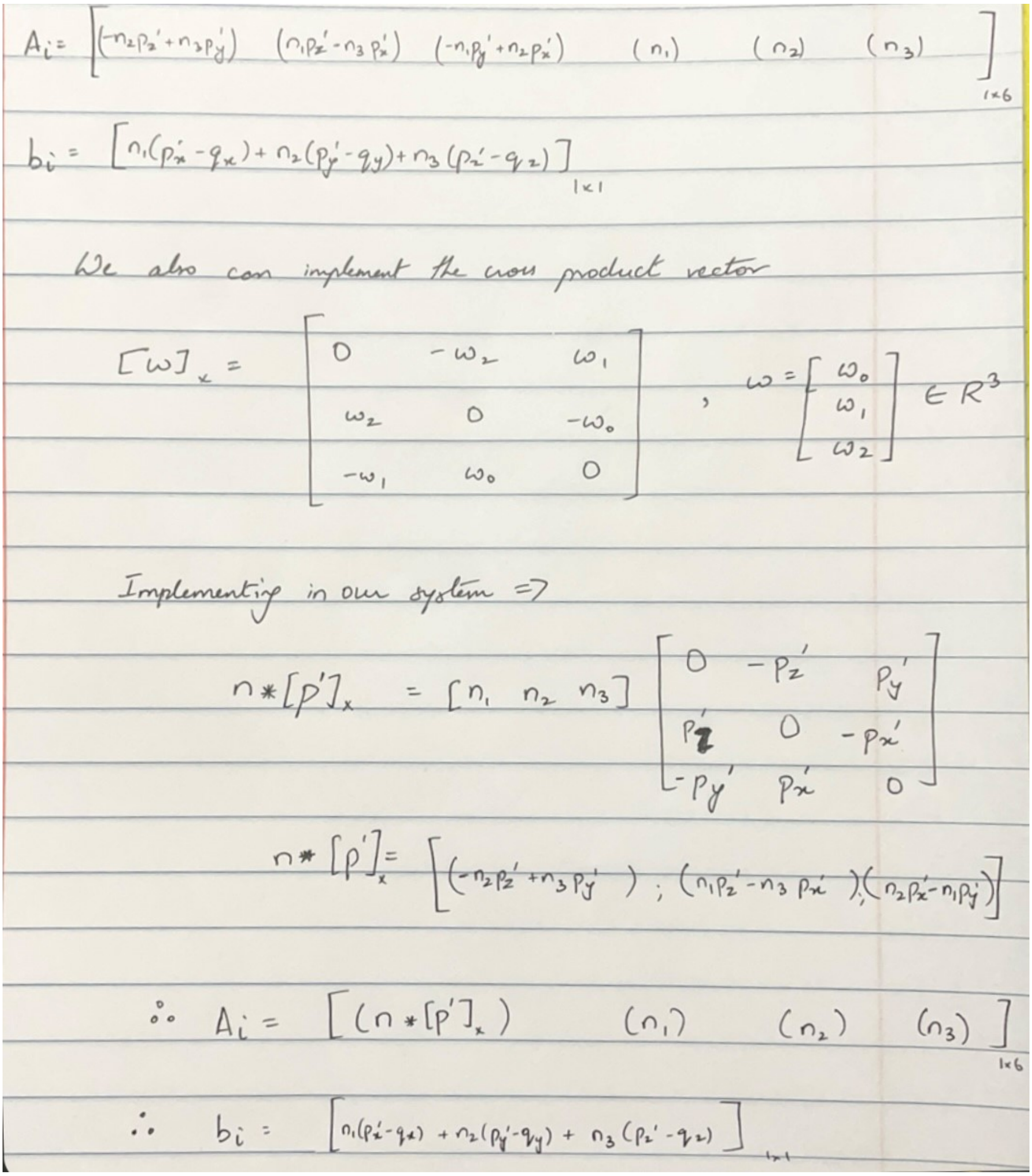
$\Rightarrow$ Linear Error function $= n_1(p_x' - \gamma p_y' + \beta p_z' + t_x - q_x)$
$+ n_2(\gamma p_x' + p_y' - \alpha p_z' + t_y - q_y)$
$+ n_3(\beta p_x' + \alpha p_y' + p_z' + t_z - q_z)$

Parsing this data into A_i & b_i

Thus we have our A_i and b_i matrices from parsing the resultant matrix from our previous step.

We further implement the cross product operator as instructed in the Homework as shown below:

```
def cross_product_operator(w):
    return np.array([[0, -w[2], w[1]],
                     [w[2], 0, -w[0]],
                     [-w[1], w[0], 0]])
```



$$A_i = \left[(-n_2 p_z' + n_3 p_y') \quad (n_1 p_z' - n_3 p_x') \quad (-n_1 p_y' + n_2 p_x') \quad (n_1) \quad (n_2) \quad (n_3) \right]_{1\times 6}$$

$$b_i = \left[n_1(p_x' - q_x) + n_2(p_y' - q_y) + n_3(p_z' - q_z) \right]_{1\times 1}$$

We also can implement the cross product vector

$$[\omega]_\times = \begin{bmatrix} 0 & -\omega_2 & \omega_1 \\ \omega_2 & 0 & -\omega_0 \\ -\omega_1 & \omega_0 & 0 \end{bmatrix}, \quad \omega = \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \end{bmatrix} \in R^3$$

Implementing in our system =>

$$n * [p']_\times = [n_1 \ n_2 \ n_3] \begin{bmatrix} 0 & -p_z' & p_y' \\ p_z' & 0 & -p_x' \\ -p_y' & p_x' & 0 \end{bmatrix}$$

$$n * [p']_\times = \left[(-n_2 p_z' + n_3 p_y') ; (n_1 p_z' - n_3 p_x') ; (n_2 p_x' - n_1 p_y') \right]$$

$$\therefore A_i = \left[(n * [p']_\times) \quad (n_1) \quad (n_2) \quad (n_3) \right]_{1\times 6}$$

$$\therefore b_i = \left[n_1(p_x' - q_x) + n_2(p_y' - q_y) + n_3(p_z' - q_z) \right]_{1\times 1}$$

Figure 2: Fig. 2: Error function Linearization

3 Optimization - *icp.py*

Question 2.3.1 (Implement Linearized System)

Implement **build_linear_system** and the corresponding **solve**

```python
def build_linear_system(source_points, target_points, target_normals, T):
    M = len(source_points)
    assert len(target_points) == M and len(target_normals) == M

    R = T[:3, :3]
    t = T[:3, 3:]

    p_prime = (R @ source_points.T + t).T
    q = target_points
    n_q = target_normals
    # print(n_q[0])
    # print(" \n H- Stack:")
    # print(np.hstack(n_q[0]))
    # print(" \n Reshape:")
    # a=n_q[0]
    # b=a.reshape([1, 3])
    # print(b)
    # print(" \n New Axis: ")
    # print(n_q[0][np.newaxis, :])

    A = np.zeros((M, 6))
    b = np.zeros((M, ))

    # TODO: build the linear system
    for i in range(M):
        A[i, :] = np.hstack(([n_q[i][np.newaxis, :] @ cross_product_operator(p_prime[i]), -n_q[i][np.newaxis, :]]))
        b[i] = np.dot(n_q[i], (p_prime[i] - q[i]))
    # End of TODO
```

Figure 3: Build Linear System

```python
def solve(A, b):
    '''
    \param A (6, 6) matrix in the LU formulation, or (N, 6) in the QR formulation
    \param b (6, 1) vector in the LU formulation, or (N, 1) in the QR formulation
    \return delta (6, ) vector by solving the linear system. You may directly use dense solvers from numpy.
    '''
    # TODO: write your relevant solver
    Q, R = np.linalg.qr(A)
    d = np.dot(Q.T, b)
    return np.dot(np.linalg.inv(R), d)
```

Figure 4: Solve

Please note that the A_i matrix has the last 3 columns as the normal vector with a negative sign. This is attributed to the import of the QR solver from the previous assignment which solved for (AX - B), however here it is of the format (AX + B), hence I have corrected within the A matrix (it is also possible within the solver as well).

Question 2.3.2 (ICP Visualization)

Below are the results for visualization of frame 10-50 as well as frame 10-100.

Frames 10 - 100 use 100 iterations to arrive fusion result. It can be inferred that for closer source and target frames, a fewer number of iterations are required for convergence.

Frames 10 - 100 use 100 iterations to arrive fusion result. It can be inferred that for far apart source and target frames, a larger number of iterations are required for convergence.

Figure 5: Frame10-50 (before ICP)

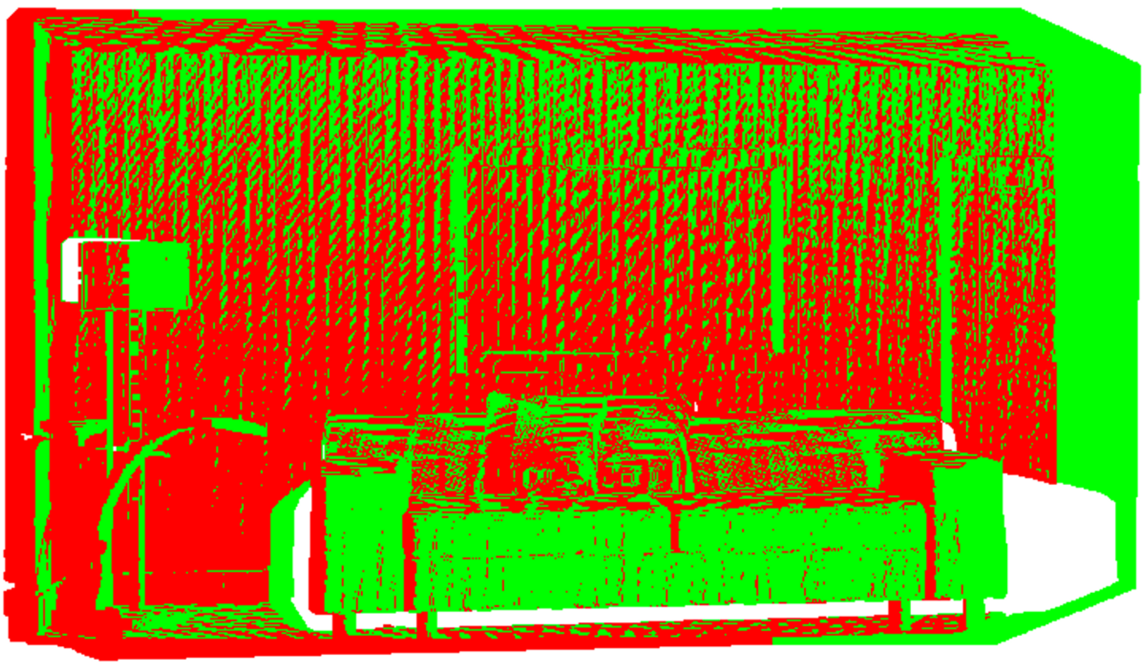


Figure 6: Frame 10-50 (after ICP)

Figure 7: Frame 10-100 (before ICP)

Figure 8: Frame 10-100 (after ICP)

Q3 Fusion.py

Question 3.1 (Filter Passes)

Filter pass1 masks all indices outside our vertex map limits. It also ensures depths are greater than 0.

Filter pass2 masks all indices beyond a threshold euclidean distance between source and input points. It also checks the angle in between source and input normals using cosine (dot product)

```python
def filter_pass1(self, us, vs, ds, h, w):
    '''
    TODO: implement the filter function
    \param self The current maintained map, unused
    \param us Putative corresponding u coordinates on an image, (N, 1)
    \param vs Putative corresponding v coordinates on an image, (N, 1)
    \param vs Putative corresponding d depth on an image, (N, 1)
    \param h Height of the image projected to
    \param w Width of the image projected to
    \return mask (N, 1) in bool indicating the valid coordinates
    '''
    mask = ((us >= 0) & (us < w) & (vs >= 0)  & (vs < h) & (ds >= 0)).astype(bool)
    return mask
```

Figure 9:

```python
def filter_pass2(self, points, normals, input_points, input_normals,
                 dist_diff, angle_diff):
    '''
    TODO: implement the filter function
    \param self The current maintained map, unused
    \param points Maintained associated points, (M, 3)
    \param normals Maintained associated normals, (M, 3)
    \param input_points Input associated points, (M, 3)
    \param input_normals Input associated normals, (M, 3)
    \param dist_diff Distance difference threshold to filter correspondences by positions
    \param angle_diff Angle difference threshold to filter correspondences by normals
    \return mask (N, 1) in bool indicating the valid correspondences
    '''
    mask_dist = (np.linalg.norm((points - input_points), axis=1)) < dist_diff
    mask_ang = np.abs(np.arccos((normals * input_normals).sum(axis=1))) < angle_diff
    mask = np.logical_and(mask_dist, mask_ang)

    return mask
```

Figure 10:

Question 3.2 (Merge function)

The Merge function takes the incoming transformed points, normals and colors and merges it with the existing class attributes to produce a "merged" output map represnetation.

$$\left[P = \frac{(\omega * p) + q}{\omega + 1} \right] ; \left[n = \frac{(\omega * n_p) + n_q}{\omega + 1} \right] ; \left[c = \frac{(\omega * c) + \text{new-colors}}{\omega + 1} \right]$$

$$\left[\omega = \omega + 1 \right]$$

Figure 11:

```python
def merge(self, indices, points, normals, colors, R, t):
    '''
    TODO: implement the merge function
    \param self The current maintained map
    \param indices Indices of selected points. Used for IN PLACE modification.
    \param points Input associated points, (N, 3)
    \param normals Input associated normals, (N, 3)
    \param colors Input associated colors, (N, 3)
    \param R rotation from camera (input) to world (map), (3, 3)
    \param t translation from camera (input) to world (map), (3, )
    \return None, update map properties IN PLACE
    '''
    den = (self.weights[indices] + 1)
    num_p = (self.weights[indices] * self.points[indices] + (R @ points.T + t).T)
    num_n = (self.weights[indices] * self.normals[indices] + (R @ normals.T).T)

    self.points[indices] = num_p/den
    self.normals[indices] = num_n/den

    # Note: a normalization of normals is required after weight average.
    self.normals[indices] /= np.linalg.norm(self.normals[indices], axis=1, keepdims=True)

    # Colours
    num_c = (self.weights[indices] * self.colors[indices] + colors)
    self.colors[indices] = num_c/den

    den += 1

    pass
```

Figure 12:

Question 3.3 (Add function)

The Add function takes the incoming transformed points, and concatenates them to the existing mapping. As stated in the paper, it provides this added data to the next merge function call.

```
def add(self, points, normals, colors, R, t):
    '''
    TODO: implement the add function
    \param self The current maintained map
    \param points Input associated points, (N, 3)
    \param normals Input associated normals, (N, 3)
    \param colors Input associated colors, (N, 3)
    \param R rotation from camera (input) to world (map), (3, 3)
    \param t translation from camera (input) to world (map), (3, )
    \return None, update map properties by concatenation
    '''
    p_unassociated = (R @ points.T + t)
    n_unassociated = (R @ normals.T)

    self.points = np.concatenate((self.points, p_unassociated.T))
    self.normals = np.concatenate((self.normals, n_unassociated.T))

    # Colours
    self.colors = np.concatenate((self.colors, colors))

    weight_incr = np.ones((points.shape[0], 1))

    self.weights = np.concatenate((self.weights,weight_incr))

    pass
```

Figure 13:

Question 3.4 (Fusion)

Once the fusion.py file is run, we get 2 point-based fusion output maps - the actual fused map and the map of normals.

The compression ratio can be given by the total number of points (1155977), divided by, the total size of the map times the number of frames (w*h*200).

Therefore, **Compression Ratio = 1155977/(480*600*200) = 0.02**

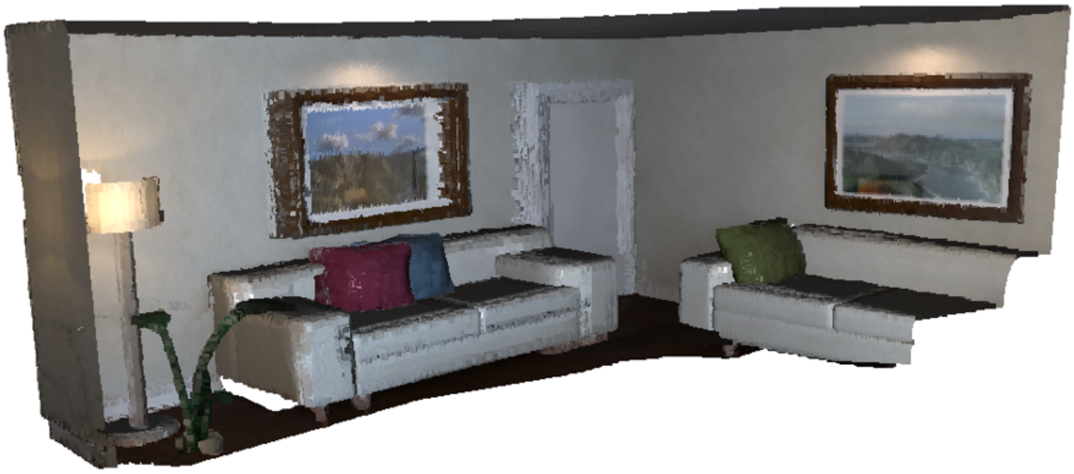


Figure 14: Fusion-point based map

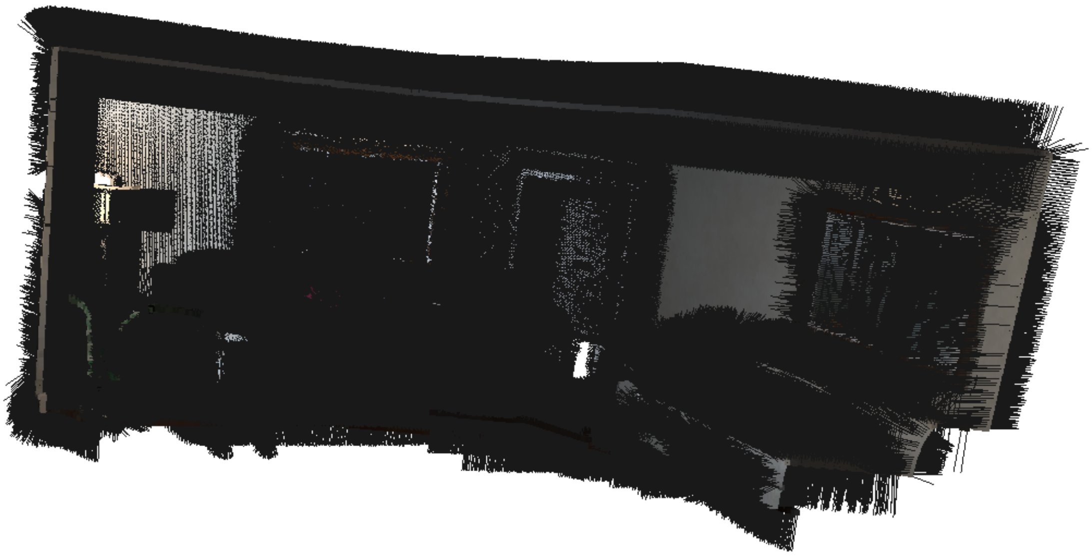


Figure 15: Normal map

Q4 Main.py

The source frame is the RGBD frame and the target frame is the map and their roles should not be switched as the projection to the vertex map happens first which produces weights accordingly, and would otherwise be incorrect. The target map consequently is then projected back from the vertex map. The map after running main.py which combines both ICP and fusion is shown below.

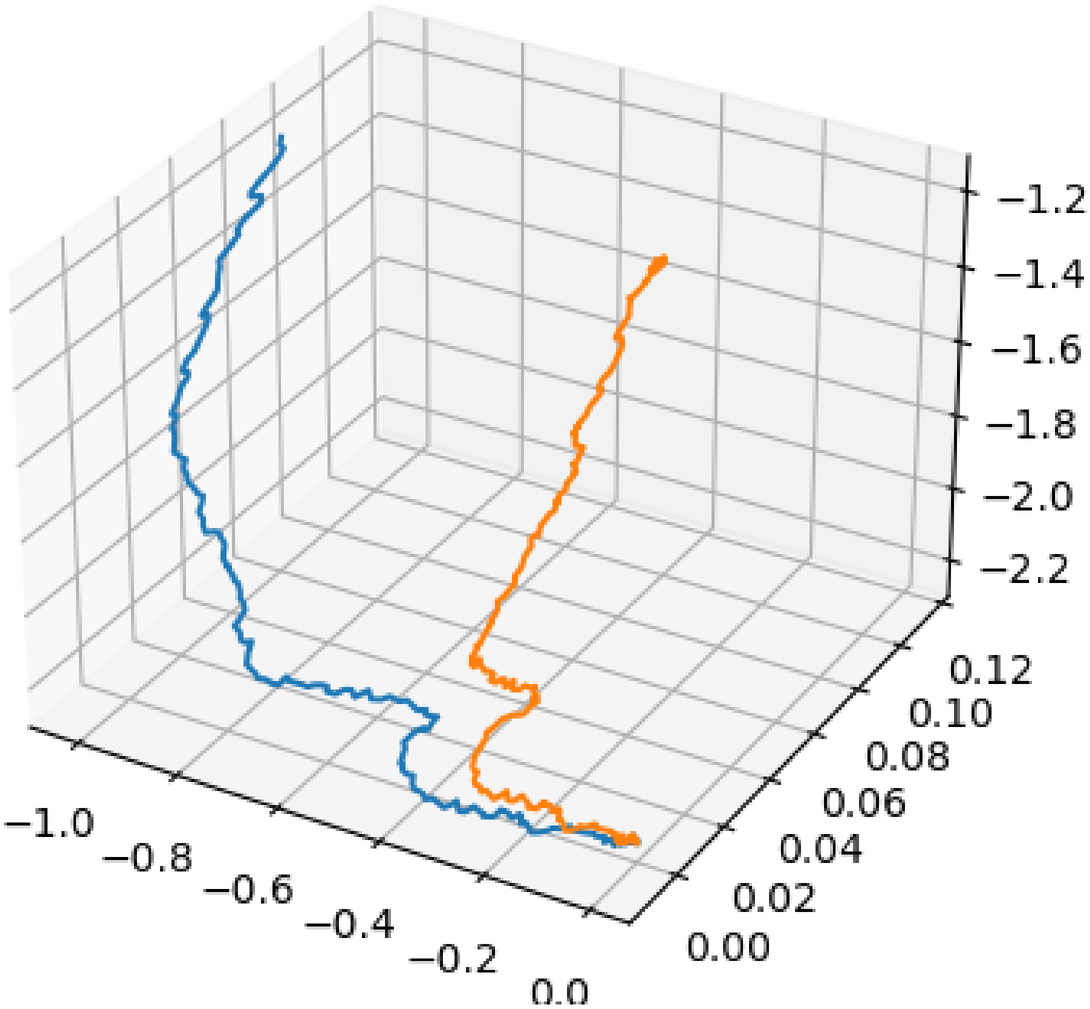


Figure 16: Final output (drift)