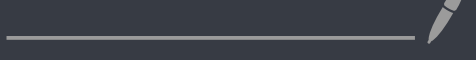


Problem 1

(a) $P_x(t+1) = P_x(t) + V_x(t)\Delta t$ where $\Delta t \in R$

It's same for y so we can get

$$\begin{bmatrix} P_x(t+1) \\ V_x(t+1) \\ P_y(t+1) \\ V_y(t+1) \end{bmatrix} = Ax_t + w_t \quad \text{where} \quad A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \& \quad w_t = \begin{bmatrix} 0 \\ w_x \\ 0 \\ w_y \end{bmatrix}$$

For noise in the model we consider the additional noise in velocity

$$P_x(t+1) = P_x(t) + V(t)$$
$$V_x(t+1) = V_x(t) + w_x(t) \quad = w_x + t_t$$

In this case, there exist no noise in the position model

(b)

$$d_i = \sqrt{(l_x^i - P_x^i)^2 + (l_y^i - P_y^i)^2} + V_i \qquad \text{for } i = 1, 2$$

(c) To make non-linear model linearized:

$$f(x+\delta_x, y) = f(x,y) + \frac{\partial f(x,y)}{\partial x}\delta_x$$

$$\hat{C} = \begin{bmatrix} \frac{\partial r_1}{\partial P_x} & \frac{\partial r_1}{\partial V_x} & \frac{\partial r_1}{\partial P_y} & \frac{\partial r_1}{\partial V_y} \\ \frac{\partial r_2}{\partial P_x} & \frac{\partial r_2}{\partial V_x} & \frac{\partial r_2}{\partial P_y} & \frac{\partial r_2}{\partial V_y} \end{bmatrix}$$

Apply the model and to find

$$\frac{\partial r_a}{\partial P_x} = \frac{(P_x - l_x^a)}{\sqrt{(l_x^a - P_x)^2 + (l_y^a - P_y)^2}} \qquad \frac{\partial r_a}{\partial P_y} = \frac{(P_y - l_y^a)}{\sqrt{(l_x^a - P_x)^2 + (l_y^a - P_y)^2}}$$

$$\& \quad \frac{\partial r_a}{\partial V_x} \Big/ \frac{\partial r_a}{\partial V_y} = 0 \qquad (\text{where } a = 1, 2)$$

With noise: $y_t = \hat{C}x_t + V$

(d) Given that $x_{t+1} = A_t x_t + w_t$

For propogation we can assert

$$x_t \sim N(M_t, \Sigma_t) \qquad w_t \sim N(0, R_t)$$

Plugin to get:

$$N(M_{t+1}, \bar{\Sigma}_{t+1}) = N(A_t M_t, A_t \Sigma_t A_t^T + R_t)$$

$$\Rightarrow A_t M_t = M_{t+1} \qquad \bar{\Sigma}_{t+1} = A_t \Sigma_t A_t^T + R_t$$

$$\bar{\Sigma}_{t+1} = A_t \cdot \begin{bmatrix} \sigma_{P_x}^2 & 0 & 0 & 0 \\ 0 & \sigma_{V_x}^2 & 0 & 0 \\ 0 & 0 & \sigma_{P_y}^2 & 0 \\ 0 & 0 & 0 & \sigma_{V_y}^2 \end{bmatrix} \cdot A_t^T \qquad \left(A_t = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)$$

$$= \begin{bmatrix} \sigma_{P_x}^2 + \sigma_{V_x}^2 & \sigma_{V_x}^2 & 0 & 0 \\ \sigma_{V_x}^2 & \sigma_{V_x}^2 + \sigma_{w_x}^2 & 0 & 0 \\ 0 & 0 & \sigma_{P_y}^2 + \sigma_{V_y}^2 & \sigma_{V_y}^2 \\ 0 & 0 & \sigma_{V_y}^2 & \sigma_{V_y}^2 + \sigma_{w_y}^2 \end{bmatrix}$$

The measurement model

$$y_t = \hat{C}_t x_t + v_t \qquad \Rightarrow \quad y_t \sim N(\hat{M}_t, \hat{\Sigma}_t)$$

$$\downarrow \qquad \searrow \; N(0, \hat{\varphi}_t)$$

$$N(\hat{M}_t, \hat{\Sigma}_t)$$

$\Rightarrow$ The kalman constant $K_t = \hat{\Sigma}_t \hat{C}_t^T (\hat{C}_t \hat{\Sigma}_t \hat{C}_t^T + \varphi_t)^{-1}$

So the new Σ_t and M_t is as

$$\Sigma_t = (I - K_t \hat{C}_t) \hat{\Sigma}_t \qquad M_t = \hat{M}_t + K_t (y_t - C_t \hat{M}_t)$$

We imported the following:

```
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.metrics import pairwise_distances_argmin_min
```

(a) The code of estimate correspondence is as follows:

```
def estimate_correspondences(X, Y, t, R, d_max):
    correspondences = []
    x_transformed = ((R @ X.T) + t).T
    distances, indices = pairwise_distances_argmin_min(x_transformed, Y)

    for i, index in enumerate(indices):
        if distances[i] < d_max:
            correspondences.append((i, index))

    return correspondences
```

Where I transformed the X points and used the argmin_min function to generate a list of shortest distance points from list 1 to list 2.

From there I checked if that distance is shorter than d_max and append if true.

(b) The code of computing optimal rigid registration is as follows (next page):

In the code, I followed the pseudo code to calculate centroid, and covariance, constructed the optimal rotation, and recovered the optimal translation.

```python
def optimal_rigid_registration(X, Y, C):

    n_correspondences = len(C)

    X_associated = []
    Y_associated = []
    for i, j in C:
        x = X[i]
        y = Y[j]
        X_associated.append(x)
        Y_associated.append(y)

    x_centroid = np.sum(X_associated, axis=0) / n_correspondences
    y_centroid = np.sum(Y_associated, axis=0) / n_correspondences

    X_centered = X - x_centroid
    Y_centered = Y - y_centroid

    W = np.zeros((3, 3))

    for i, j in C:
        x = X_centered[i]
        y = Y_centered[j]
        W += y.reshape(3, 1) @ x.reshape(1, 3)

    U, S, V_T = np.linalg.svd(W / n_correspondences)

    R = U @ V_T

    t = y_centroid.reshape(3, 1) - R @ x_centroid.reshape(3, 1)

    return R, t
```

(c) The ICP algorithm is as follows:

```python
# ICP function
def ICP(X,Y,t0,R0,dmax,num_ICP_iters):
    #initialization
    t=t0
    R=R0
    for i in range (num_ICP_iters):
        C = estimate_correspondences(X,Y,t,R,dmax)
        R,t = optimal_rigid_registration(X,Y,C)
    return t,R,C
```

It takes in input and uses the above two algorithms to perform fixed number of times as requested.

(d) RMSE:

```
#Finding RMSE
distance_list = []
XX=((Rf@X.T)+tf).T
for i,j in Cf:
    distance=np.linalg.norm(Y[j]-XX[i])
    distance_list.append(distance**2)
RMSE=math.sqrt(sum(distance_list)/len(distance_list))
RMSE
```

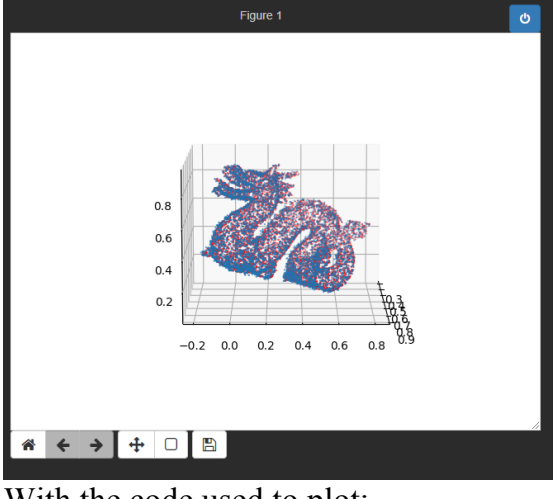
```
0.008950576587683131
```

T & R

```
tf,Rf
```

```
(array([[ 0.49661487],
        [-0.29392971],
        [ 0.29645004]]),
 array([[ 0.95126601, -0.15043058, -0.26919069],
        [ 0.22323628,  0.9381636 ,  0.26460276],
        [ 0.21274056, -0.31180074,  0.92602471]]))
```

And the output is



With the code used to plot:

```
%matplotlib notebook
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Y data
x1 = Y[:,0]
y1 = Y[:,1]
z1 = Y[:,2]

X_tran = ((Rf@X.T)+tf).T
x2 = X_tran[:,0]
y2 = X_tran[:,1]
z2 = X_tran[:,2]

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x1, y1, z1, s=0.5)
ax.scatter(x2, y2, z2, s=0.5, color='r')
ax.view_init(10, 0)
plt.show()
```

It’s seen that it’s a beautiful dragon.

Problem 3

(a) From the normal distribution

$$\varepsilon_L = N(0, \sigma_L^2) \qquad \varepsilon_R = N(0, \sigma_R^2)$$

$$P(X_{t_2} | X_{t_1}, \varphi_l, \varphi_r, r, w, \sigma_l, \sigma_r)$$

where if we plug in φ_l and φ_r

$$\sim P(X_{t_2} | X_{t_1}, \varepsilon_l, \varepsilon_R) \quad \text{by removing constants}$$

```
vl = pdL + np.random.normal(0,stdL)
vr = pdR + np.random.normal(0,stdR)
```

Now that we have $\tilde{\varphi}_l$ and $\tilde{\varphi}_r$, the velocity in body centric frame:

$$V_B = \begin{bmatrix} V_{x_B} \\ V_{y_B} \\ \dot{\theta}_R \end{bmatrix} = \begin{bmatrix} \frac{r \cdot (\tilde{\varphi}_l + \varphi_R)}{2} \\ 0 \\ \frac{r \cdot (\tilde{\varphi}_R - \varphi_l)}{w} \end{bmatrix}$$

From here the velocity at $I \in SE(2)$ Ω can be derived as given so the $y(t_2) = X_{t_1} \exp\left((t_2 - t_1)\Omega(\varphi_l, \varphi_r)\right)$

```
def find_pose(t1,X_t1,t2):
    vl = pdL + np.random.normal(0,stdL)
    vr = pdR + np.random.normal(0,stdR)

    V=np.array([[(r/2)*(vr+vl)],[0],[(r/w)*(vr-vl)]])

    omega = np.array([[0,-V[2][0],V[0][0]],[V[2][0],0,0],[0,0,0]])

    X_t2 = X_t1 @ expm((t2-t1)*omega)

    return X_t2
```

(b) Because $\varepsilon_p \sim N(0, \sigma_p^2 I_2)$

and z_t, l_t are fixed for a known i

$p(z_t | l_t)$ is also $\sim N(0, \sigma_p^2 I_2)$

and $w_i \triangleq p(z_t | x_t^{[i]}) \triangleq p(\varepsilon_p)$

$$w_i = \frac{1}{\sigma_p \sqrt{2\pi}} \cdot \exp\left(-\frac{1}{2}\left(\frac{\varepsilon_p^2}{\sigma_p}\right)\right)$$

$$= \frac{1}{\sigma_p \sqrt{2\pi}} \cdot \exp\left(-\frac{1}{2}\left(\frac{(z_t - l_t)^2}{\sigma_p}\right)\right)$$

```
def calculate_importance_weights(X, z):
    weights = []
    distances = []

    particle_array = np.array(X)
    translations = particle_array.T[2]

    for i in range(N):
        x, y = translations[0:2, i]
        distance = np.linalg.norm(z - np.array([[x], [y]]))
        weight = norm.pdf(distance, loc=0, scale=stdP)
        distances.append(distance)
        weights.append(weight)

    weight_array = np.array(weights)
    weight_nparr = np.reshape(weight_array, (1, N))

    return weight_nparr
```

(c)

```
def particle_filter_propagate(t1,X_T1,t2,N):
    X_T2=[]
    for i in range(N):
        if len(X_T1)>10:
            X_2pose = find_pose(t1,X_T1[i],t2)
        else:
            X_2pose = find_pose(t1,X_T1,t2)
        X_T2.append(X_2pose)
    return X_T2
```

Using the find_pose function from (a) we can easily propagates the particle by append new pose

(d)

```
def particle_filter_update(X_list,z):
    X_array = np.array(X_list)

    weights = calculate_importance_weights(X_list,z)

    weights /= np.sum(weights)
    pos = (np.arange(N) + np.random.random())/ N

    indices = np.searchsorted(np.cumsum(weights),pos)

    X_array[:] = X_array[indices]

    return X_array
```

Using the calculate_importance_weight function from (b) we can follow the pseudo code to implement this