Mobile Robotics Assignment 4

Team ID 24

Roll No. 2019702002 Nivedita Rufus

Roll No. 2019702008 Sravya Vardhani S

Q) Localization using sensor fusion

The Kalman filter assumes a linear transition and observation model, and models all of its uncertainties in terms of Gaussian noise. In real- world situations we have non-linear functions lead to non Gaussian distributions, and therefore we cannot use Kalman filter, an Extended Kalman filter is used. EKF filter locally linearizes the data by taking the taylor series estimate.

Prediction:

$$g(u_t, x_{t-1}) \approx g(u_t, \mu_{t-1}) + \underbrace{\frac{\partial g(u_t, \mu_{t-1})}{\partial x_{t-1}}}_{=:\, G_t} (x_{t-1} - \mu_{t-1})$$

Correction:

$$h(x_t) \approx h(\bar{\mu}_t) + \underbrace{\frac{\partial h(\bar{\mu}_t)}{\partial x_t}}_{=:\, H_t} (x_t - \bar{\mu}_t)$$

In the above equations G_t and H_t are Jacobians.

1) Extract variables from the dataset

```
dataset = np.load('dataset.npz')
x = dataset['x_true'].ravel()
y = dataset['y_true'].ravel()
theta = dataset['th_true'].ravel()
ranges = dataset['r']
landmarks = dataset['l']
v = dataset['v'].ravel()
w = dataset['om'].ravel()
bearing = dataset['b']
v_var = dataset['v_var'][0,0]
w_var = dataset['om_var'][0,0]
b_var = dataset['b_var'][0,0]
r_var = dataset['r_var'][0,0]
d = dataset['d'][0,0]
```

2) Define functions

Prediction Function

The matrix for motion model

$$\begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} = \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ \theta_{k-1} \end{bmatrix} + dt \begin{bmatrix} \cos\theta_{k-1} & 0 \\ \sin\theta_{k-1} & 0 \\ 0 & 1 \end{bmatrix} \left(\begin{bmatrix} v_k \\ \omega_k \end{bmatrix} + \mathbf{w}_k \right)$$

Equations are :

$$x_k = x_{k-1} + dt\cos\theta_{k-1} v_k + dt\cos\theta_{k-1} W_1$$
$$y_k = y_{k-1} + dt\sin\theta_{k-1} v_k + dt\sin\theta_{k-1} W_1$$
$$\theta_k = \theta_{k-1} + dt\omega_k + dtW_2$$

The prediction part Jacobian by using definition of Jacobian

$$G_t = \frac{\partial g(u_t, \mu_{t-1})}{\partial \mu_{t-1}} = \begin{pmatrix} \frac{\partial x'}{\partial \mu_{t-1,x}} & \frac{\partial x'}{\partial \mu_{t-1,y}} & \frac{\partial x'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial y'}{\partial \mu_{t-1,x}} & \frac{\partial y'}{\partial \mu_{t-1,y}} & \frac{\partial y'}{\partial \mu_{t-1,\theta}} \\ \frac{\partial \theta'}{\partial \mu_{t-1,x}} & \frac{\partial \theta'}{\partial \mu_{t-1,y}} & \frac{\partial \theta'}{\partial \mu_{t-1,\theta}} \end{pmatrix}$$

Using the above equations we get the Jacobian as

$$\mathbf{G}_t = \begin{bmatrix} 1 & 0 & -\sin\theta_{k-1}V_k - \sin\theta_{k-1}W_1 \\ 0 & 1 & \cos\theta_{k-1}V_k + \cos\theta_{k-1}W_1 \\ 0 & 0 & 1 \end{bmatrix}$$

Code snippet:

Implementing it

```
def prediction(mu, sigma, v, w,v_var,w_var, Q, dt = 0.1):
      g = np.array([[cos(mu[2]),0],[sin(mu[2]),0],[0,1]])
      temp = np.array([[v+v_var],[w+w_var]])
      #print(Q.shape)
      g = dt*np.dot(g,temp)
      G = np.array([[1, 0, -v*dt*sin(mu[2])],
                  [0, 1, v*dt*cos(mu[2])],
                                  [0, 0, 1]])
      mu = mu + g
      L = dt * np.array([[cos(mu[2]),0],[sin(mu[2]),0],[0,1]])
      q = np.linalg.multi_dot([L,Q,L.T])
      sigma = np.dot(np.dot(G, sigma), G.T) + q
      mu[2] = get_to_pi(mu[2])
      return mu, sigma
```

Correction Function

The matrix for observation model

$$\begin{bmatrix} r_k^l \\ \phi_k^l \end{bmatrix} = \begin{bmatrix} \sqrt{(x_l - x_k - d\cos\theta_k)^2 + (y_l - y_k - d\sin\theta_k)^2} \\ \operatorname{atan2}(y_l - y_k - d\sin\theta_k, x_l - x_k - d\cos\theta_k) - \theta_k \end{bmatrix} + \mathbf{n}_k$$

The corrected part Jacobian by using definition of Jacobian

$$H_t = \frac{\partial h(\bar{\mu}_t, m)}{\partial x_t} = \begin{pmatrix} \frac{\partial r_t}{\partial \bar{\mu}_{t,x}} & \frac{\partial r_t}{\partial \bar{\mu}_{t,y}} & \frac{\partial r_t}{\partial \bar{\mu}_{t,\theta}} \\ \frac{\partial \phi_t}{\partial \bar{\mu}_{t,x}} & \frac{\partial \phi_t}{\partial \bar{\mu}_{t,y}} & \frac{\partial \phi_t}{\partial \bar{\mu}_{t,\theta}} \end{pmatrix}.$$

Using the above equations we get the Jacobian partial derivatives as

$$\frac{\partial r_t}{\partial x_k} = -\frac{x_l - x_k - d\cos\theta_k}{\sqrt{(x_l - x_k - d\cos\theta_k)^2 + (y_l - y_k - d\sin\theta_k)^2}}$$

$$\frac{\partial r_t}{\partial y_k} = -\frac{y_l - y_k - d\sin\theta_k}{\sqrt{(x_l - x_k - d\cos\theta_k)^2 + (y_l - y_k - d\sin\theta_k)^2}}$$

$$\frac{\partial r_t}{\partial \theta_k} = \frac{(x_l - x_k - d\cos\theta_k)d\sin\theta_k - (y_l - y_k - d\sin\theta_k)d\cos\theta_k}{\sqrt{(x_l - x_k - d\cos\theta_k)^2 + (y_l - y_k - d\sin\theta_k)^2}}$$

$$\frac{\partial \phi}{\partial x_k} = -\frac{1}{1+\frac{(y_l-y_k-d\sin\theta_k)}{(x_l-x_k-d\cos\theta_k)}^2}\frac{(y_l-y_k-d\sin\theta_k)}{(x_l-x_k-d\cos\theta_k)^2}$$

$$\frac{\partial \phi}{\partial y_k} = -\frac{1}{1+\frac{(y_l-y_k-d\sin\theta_k)}{(x_l-x_k-d\cos\theta_k)}^2}\frac{1}{(x_l-x_k-d\cos\theta_k)}$$

$$\frac{\partial \phi}{\partial \theta_k} = -\frac{1}{1+\frac{(y_l-y_k-d\sin\theta_k)}{(x_l-x_k-d\cos\theta_k)}^2}\frac{(x_l-x_k-d\cos\theta_k)d\cos\theta_k+(y_l-y_k-d\sin\theta_k)d\sin\theta_k}{(x_l-x_k-d\cos\theta_k)^2}$$

Code Snippet :

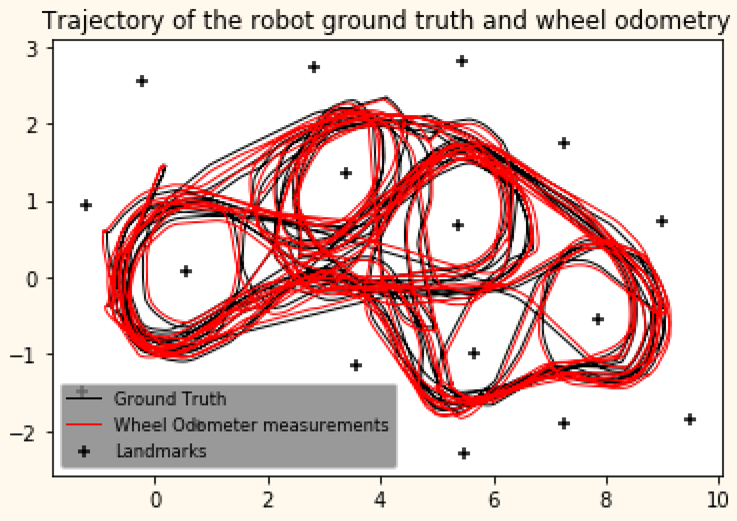
```
def correction(mu, sigma, ranges, bearing, landmarks, d, r_var, b_var,R):
      idx = np.where (ranges != 0.0)
      H = np.empty([2,3])
      r = np.empty([2,1])
      z = np.empty(r.shape)
      j = 0
      for i in idx[0]:
            z[j] = ranges[i]
            bearing[i] = get_to_pi(bearing[i])
            z[j+1] = bearing[i]

            #range
            p = sqrt((landmarks[i,0] - mu[0] - d*cos(mu[2]))**2 +
(landmarks[i,1] - mu[1] - d*sin(mu[2]))**2)
            r[j] = p
            dx = -(landmarks[i,0] - mu[0] - d*cos(mu[2]))/p
            dy = -(landmarks[i,1] - mu[1] - d*sin(mu[2]))/p
            dtheta = (-dx*d*sin(mu[2])) + (dy*d*cos(mu[2]))
            H[j] = dx,dy,dtheta
            #bearing
            p = atan2((landmarks[i,1] - mu[1] - d*sin(mu[2])),
(landmarks[i,0] - mu[0] - d*cos(mu[2]))) - mu[2]
```

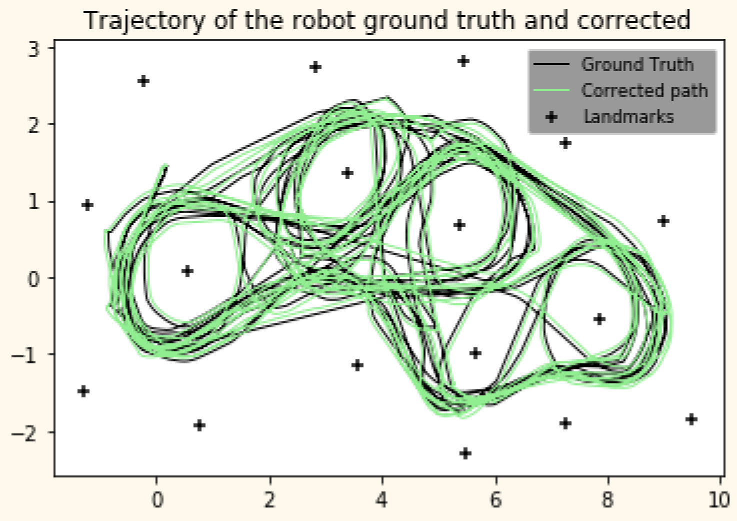
```
            r[j+1] = get_to_pi(p)
            s = ((landmarks[i,1] - mu[1] - d*sin(mu[2]))/(landmarks[i,0] -
mu[0] - d*cos(mu[2])))**2
            s = 1/(1+s)
            t = 1/(landmarks[i,0] - mu[0] - d*cos(mu[2]))
            dx = s * (landmarks[i,1] - mu[1] - d*sin(mu[2])) * (t**2)
            dy = -s * t
            dtheta = s * (t**2) * ((-d*cos(mu[2])*(landmarks[i,0] - mu[0] -
d*cos(mu[2]))) - (d*sin(mu[2])*(landmarks[i,1] - mu[1] - d*sin(mu[2])))) -1
            H[j+1] = dx,dy,dtheta
            L = np.eye(2)
            c = np.linalg.multi_dot([L,R,L.T])
            a = np.dot(np.dot(H,sigma),np.transpose(H))
            c = a + R
            c = np.matrix(c,dtype=float)
            inverse = inv(c)
            K = np.dot(np.dot(sigma,np.transpose(H)), inverse)
            c = np.dot(K,H)
      #update equations
            mu = mu + np.dot(K,(z.reshape(r.shape) - r))
            mu[2] = get_to_pi(mu[2])
            sigma = np.dot((np.eye(len(c)) - c),sigma)
      return mu, sigma
```

Outputs :

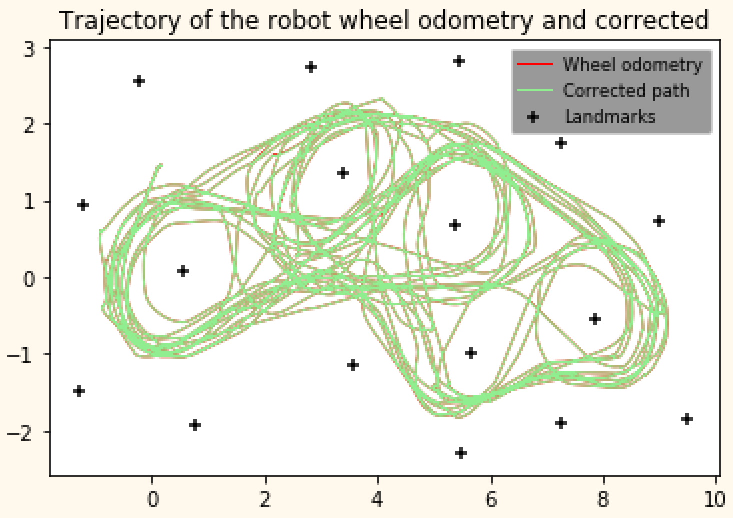
1) Plot of the trajectory estimated only using the wheel odometer measurements and ground truth.



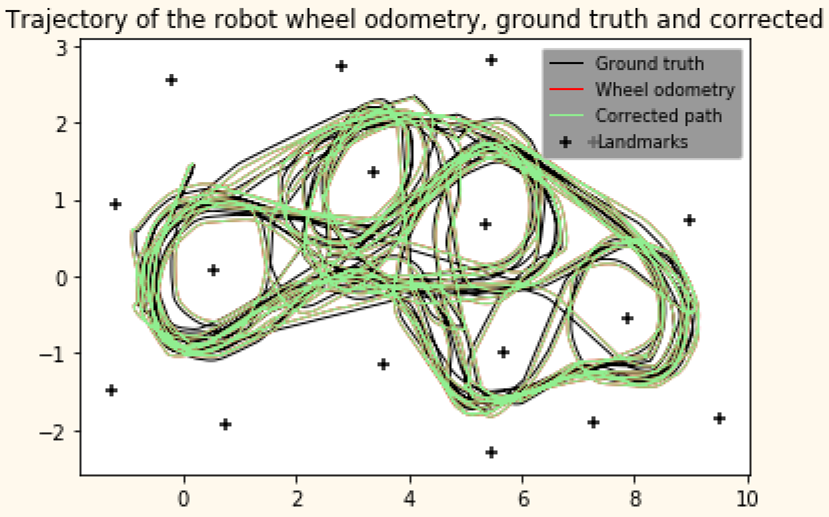
2) Plot of the trajectory estimated using the Extended Kalman filter and ground truth.



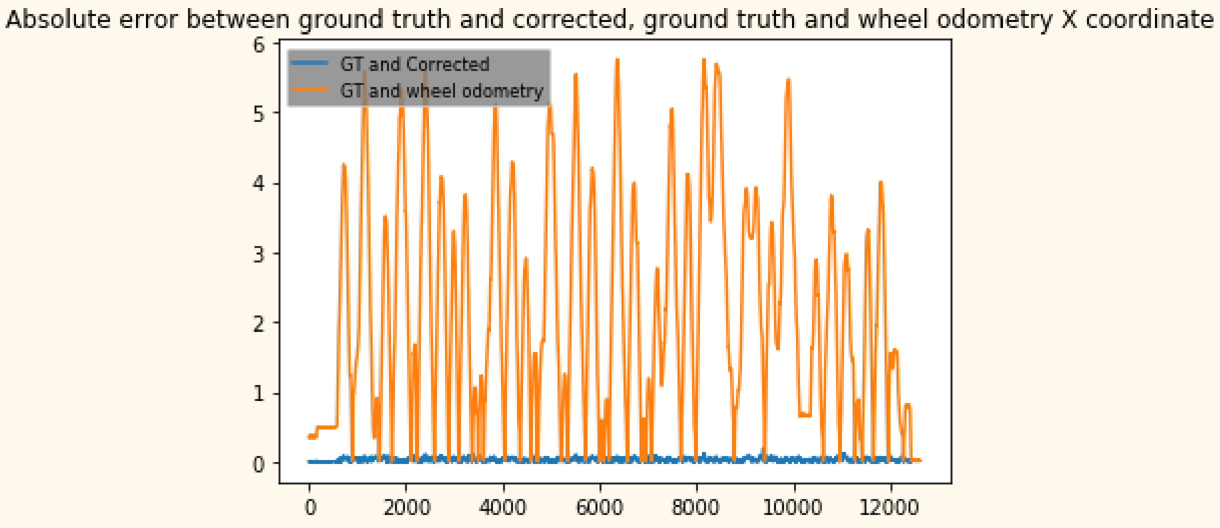
3) Plot of the trajectory estimated using the Extended Kalman filter and wheel odometry.

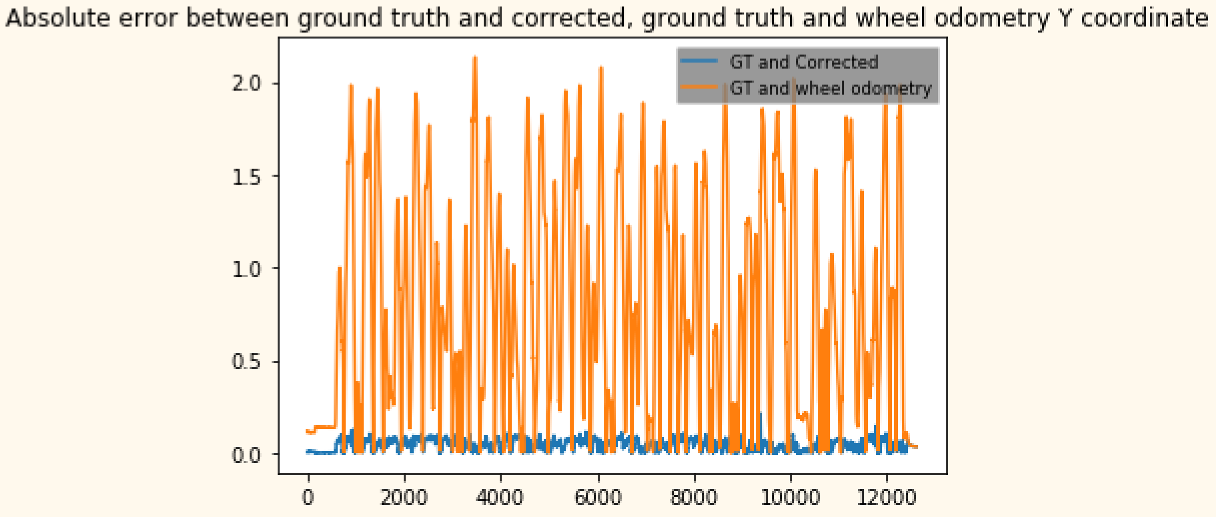


4) Plot of the trajectory estimated using the wheel odometry, ground truth and Extended Kalman filter.



We cannot appreciate the effect of the extended Kalman filter directly from these plots, but the Mean square error between the Kalman filter estimates and ground truth, mean square error between the wheel odometry measurements and the ground truth gives us a better idea.





As we can see above the error is significantly minimized by the use of EKF.

[BONUS]

Video included in the zip file

References

[1] Robot Mapping Extended Kalman Filter Cyrill Stachniss-
http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam03-ekf.pdf

Roles:

We have discussed together and implemented the functions there is no distinct role division.