Introduction

In this assignment you will be recursively estimating a vehicle trajectory using available measurements and motion model.

The vehicle is equipped with a LIDAR sensor, which returns range and bearing measurements corresponding to individual landmarks in the environment. The correspondences between landmarks and their global positions are assumed to be known beforehand. We also assume knowledge about which measurement belongs to which landmark.

Motion and measurement models

The motion model recieves odometry readings as inputs, and outputs the state (2d pose) of the vehicle $\mathbf{x}_k = [x \, y \, \theta]^T$:

\begin{align} \mathbf{x}*{k} &= \mathbf{x}*{k-1} + T

$$\begin{bmatrix} \cos\theta_{k-1} & 0 \\ \sin\theta_{k-1} & 0 \\ 0 & 1 \end{bmatrix}$$

\left(

$$\begin{bmatrix} v_k \\ \omega_k \end{bmatrix}$$

- \mathbf{w}_k \right) \, , \, \, \, \, \, \mathbf{w}_k = \mathcal{N}\left(\mathbf{0}, \mathbf{Q}\right) \end{align}

The measurement model relates the current pose of the vehicle to the range and bearing measurements $\mathbf{y}^l_k = [r \, \phi]^T$.

$$\mathbf{y}^l_k = \begin{bmatrix} \sqrt{(x_l - x_k - d\cos\theta_k)^2 + (y_l - y_k - d\sin\theta_k)^2} \\ atan2\left(y_l - y_k - d\sin\theta_k, x_l - x_k - d\cos\theta_k\right) - \theta_k \end{bmatrix} + \mathbf{n}^l_k \, , \quad \mathbf{n}^l_k = \mathcal{N}\left(\mathbf{0}, \mathbf{R}\right)$$

Getting started

Since the above models are nonlinear, we recommend using the Extended Kalman Filter (EKF) as the state estimator. Specifically, you will need to provide code implementing the following steps:

- the prediction step, which uses odometry measurements and above motion model to provide a state and covariance estimate at a given timestep
- the correction step, which uses the range and bearing measurements provided by the LIDAR to correct the estimates

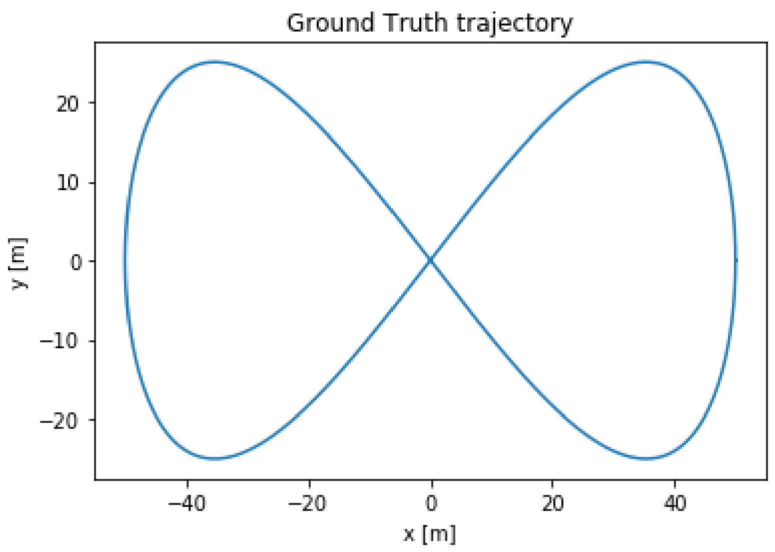
First, let's unpack the available data:

In [1]:
```python
import pickle
import numpy as np
import matplotlib.pyplot as plt

with open('data/data.pickle', 'rb') as f:
    data = pickle.load(f)

t = data['t']  # timestamps [s]

x_true = data['x_true']  # ground truth x position [m]
y_true = data['y_true']  # ground truth y position [m]
th_true = data['th_true']  # ground truth orientation [rad]

# input signal
v = data['v']  # translational velocity input [m/s]
om = data['om']  # rotational velocity input [rad/s]

# bearing and range measurements, LIDAR constants
b = data['b']  # bearing to each landmarks center in the frame attached
 to the laser [rad]
r = data['r']  # range measurements [m]
l = data['l']  # x,y positions of landmarks [m]
d = data['d']  # distance between robot center and laser rangefinder [m]
print(d)
```

```
[0]
```

Since the ground truth position and orientation are available, it is useful to plot them out before starting the assignment:

In [2]:
```python
gt_fig = plt.figure()
ax = gt_fig.add_subplot(111)
ax.plot(x_true, y_true)
ax.set_xlabel('x [m]')
ax.set_ylabel('y [m]')
ax.set_title('Ground Truth trajectory')
plt.show()
```



Now that our data is loaded, we can start getting things ready for our solver. One of the most important aspects of a filter is setting the input andmeasurement noise covariance matrices, as well as the initial state and covariance values. We set the values here:

In [8]:
```
v_var = 0.01  # translation velocity variance
om_var = 0.01  # rotational velocity variance
r_var = 0.1  # range measurements variance
b_var = 0.1  # bearing measurement variance

Q = np.diag([v_var, om_var]) # input noise covariance
R = np.diag([r_var, b_var])  # measurement noise covariance

x_est = np.zeros([len(v), 3])  # estimated states, x, y, and theta
P_est = np.zeros([len(v), 3, 3])  # state covariance matrices

x_est[0] = np.array([x_true[0], y_true[0], th_true[0]]) # initial state
P_est[0] = np.diag([1, 1, 0.1]) # initial state covariance
```

Measuement update

First, let's implement the measurement update function, which takes an available landmark measurement l and updates the current state estimate $\check{\mathbf{x}}_k$. For each landmark measurement in a given timestep k you should implement the following steps:

$$
\begin{aligned}
\mathbf{K}_k &= \check{\mathbf{P}}_k \mathbf{H}_k^T \left(\mathbf{H}_k \check{\mathbf{P}}_k \mathbf{H}_k^T + \mathbf{M}_k \mathbf{R}_k \mathbf{M}_k^T\right)^{-1} \\
\check{\mathbf{y}}_k^l &= \mathbf{h}_k \left(\check{\mathbf{x}}_k, \mathbf{0}\right) \\
\hat{\mathbf{x}}_k &= \check{\mathbf{x}} + \mathbf{K}_k \left(\mathbf{y}_k^l - \check{\mathbf{y}}_k^l\right) \\
\hat{\mathbf{P}}_k &= \left(\mathbf{1} - \mathbf{K}_k \mathbf{H}_k\right) \check{\mathbf{P}}_k
\end{aligned}
$$

In [15]:
```
def wraptopi(x):
    if x > np.pi:
        x = x - (np.floor(x / (2 * np.pi)) + 1) * 2 * np.pi
    elif x < -np.pi:
        x = x + (np.floor(x / (-2 * np.pi)) + 1) * 2 * np.pi
    return x

# TODO Implement this function, which should compute and return the meas
urement Jacobian
# H_k with respect to the state, x.
# params: xl - the x position of the input landmark.
#         yl - the y position of the input landmark.
#         x  - the current estimated state.
def measurement_jacobian(xl, yl, x):
    cos_term = xl - x[0,0] - d[0]*np.cos(x[2,0])
    sin_term = yl - x[1,0] - d[0]*np.sin(x[2,0])

    ind1 = -cos_term*(np.power(cos_term, 2) + np.power(sin_term, 2))**-
0.5
    ind2 = -sin_term*(np.power(cos_term, 2) + np.power(sin_term, 2))**-
0.5
    ind3 = (d*np.sin(x[2,0])*cos_term - d*np.cos(x[2,0])*sin_term)*(np.p
ower(cos_term,2)+np.power(sin_term,2))**-0.5
    ind4 = sin_term/(np.power(cos_term,2)+np.power(sin_term,2))
    ind5 = -cos_term/(np.power(cos_term,2)+np.power(sin_term,2))
    ind6 = -ind4*(-d*np.cos(x[2,0])) - ind5*d*np.sin(x[2,0])-1


    h_jac = np.array([[ind1, ind2, ind3], [ind4, ind5, ind6]])

    return h_jac

# TODO Implement this function, which performs the Kalman update using t
he equations
# above.
# params: lk      - the array corresponding to the (x, y) coordinates of
the landmark.
#         rk      - the current range measurement.
#         bk      - the current bearing measurement.
#         P_check - the current covariance estimate.
#         x_check - the current estimated state.
def measurement_update(lk, rk, bk, P_check, x_check):
    # 3.1 Compute measurement Jacobian using the landmarks and the curre
nt estimated state.
    H = measurement_jacobian(lk[0], lk[1], x_check)

    # 3.2 Compute the Kalman gain.
    K = np.dot(np.dot(P_check, np.transpose(H)), np.linalg.inv(np.dot(H,
np.dot(P_check, np.transpose(H))) + R))

    # 3.3 Correct the predicted state.
    # NB : Make sure to use wraptopi() when computing the bearing estima
te!
    cos_term = lk[0] - x_check[0] - d*np.cos(x_check[2])
    sin_term = lk[1] - x_check[1] - d*np.sin(x_check[2])
    y_check = np.array([np.power(np.power(cos_term, 2) + np.power(sin_te
rm,2), 0.5), wraptopi(np.arctan2(sin_term, cos_term) - x_check[2])])
    y = np.array([[rk], [bk]])
    x_check += np.dot(K, (y-y_check))

    # 3.4 Correct the covariance.
    P_check = np.dot((np.identity(3) - np.dot(K,H)), P_check)

    return x_check, P_check
```

Main loop

Now, implement the main filter loop, implementing the prediction step of the EKF using the provided motion model:

$$\begin{aligned}\check{\mathbf{x}}_k &= \mathbf{f}\left(\hat{\mathbf{x}}_{k-1}, \mathbf{u}_{k-1}, \mathbf{0}\right) \\ \check{\mathbf{P}}_k &= \mathbf{F}_{k-1}\hat{\mathbf{P}}_{k-1}\mathbf{F}^T_{k-1} + \mathbf{L}_{k-1}\mathbf{Q}_{k-1}\mathbf{L}^T_{k-1} \ .\end{aligned}$$

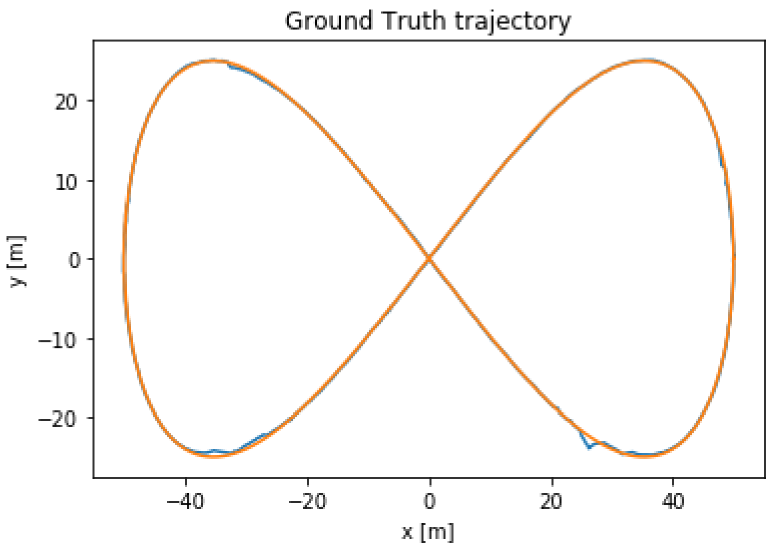
Where

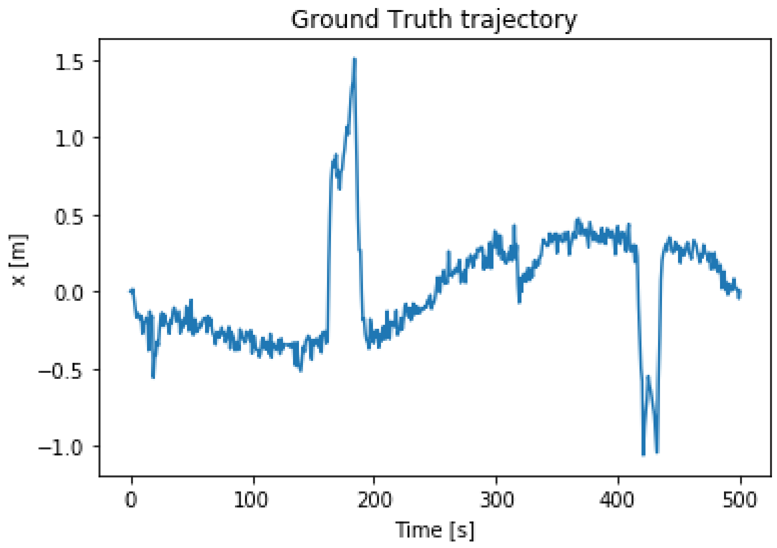
$$\mathbf{F}_{k-1} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}_{k-1}}\bigg|_{\hat{\mathbf{x}}_{k-1},\mathbf{u}_k,0} \ , \quad \mathbf{L}_{k-1} = \frac{\partial \mathbf{f}}{\partial \mathbf{w}_{k}}\bigg|_{\hat{\mathbf{x}}_{k-1},\mathbf{u}_k,0} \ .$$

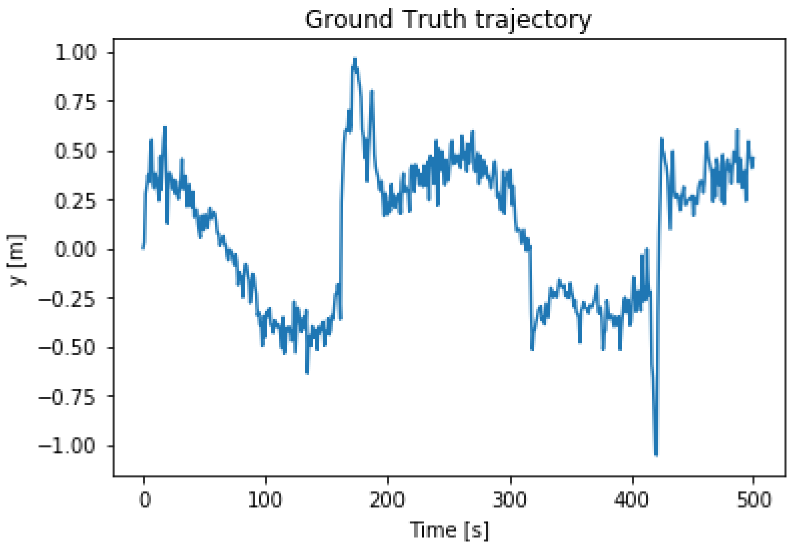
In [20]:
```
# TODO Implement the main kalman filter loop, using the above equations.
# Make use of the measurement_update() function defined above.
#### 5. Main Filter Loop #################################################
#######################
for k in range(1, len(t)):  # Start at 1 because we have initial predict
ion from ground truth.

    delta_t = t[k] - t[k - 1]  # time step (difference between timestamp
s)

    # 1. Update the state with IMU input.
    # NB : Make sure to use wraptopi() when computing the state heading!
    F_mat = np.array([[np.cos(x_est[k-1][2]), 0], [np.sin(x_est[k-1][2
]), 0], [0,1]])
    u_k = np.array([[v[k]], [om[k]]])
    w_k = np.array([[v_var], [om_var]])

    x_check = x_est[k-1].reshape(-1,1) + delta_t*np.dot(F_mat, (u_k + w_
k))
    x_check[2] = wraptopi(x_check[2])

    # 1.1 Linearize Motion Model
    # Compute the Jacobian of f w.r.t. the last state.
    F = np.array([[1,0,-delta_t*np.sin(x_est[k-1][2])*(v[k]+v_var)],
                 [0,1, delta_t*np.cos(x_est[k-1][2])*(v[k]+v_var)],
                 [0,0,1]])
    # Compute the Jacobian w.r.t. the noise variables.
    L = np.array([[delta_t*np.cos(x_est[k-1][2]),0],
                  [delta_t*np.sin(x_est[k-1][2]),0],
                  [0, delta_t]])

    # 2. Propagate uncertainty by updating the covariance.
    P_check = F*P_est[k-1]*np.transpose(F)+np.dot(L,np.dot(Q, np.transpo
se(L)))


    # 3. Update state estimate using available landmark measurements r
[k], b[k].
    for i in range(len(r[k])):
        pass
        x_check, P_check = measurement_update(l[i], r[k,i], b[k,i], P_ch
eck, x_check)

    # Set final state predictions for this kth timestep.
    x_est[k] = x_check.flatten()
    P_est[k] = P_check
```

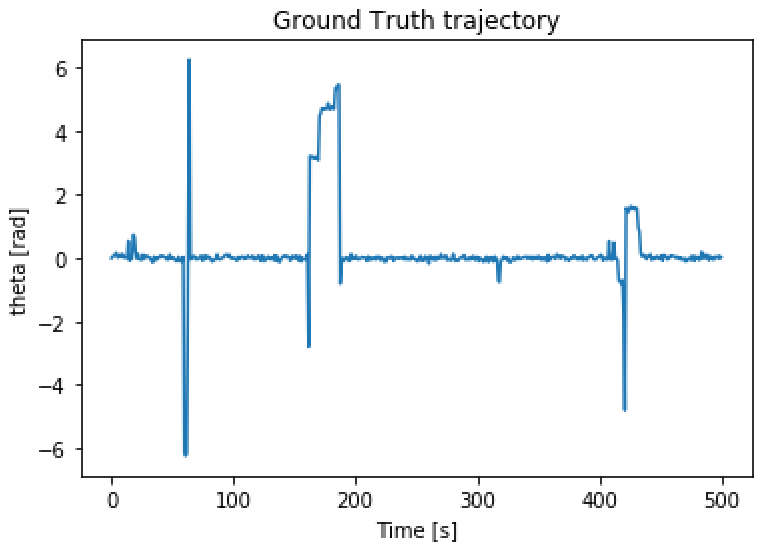
Let's plot the resulting state estimates:

In [21]:
```
gt_fig = plt.figure()
ax = gt_fig.add_subplot(111)
ax.plot(x_est[:, 0], x_est[:, 1])
ax.plot(x_true[:], y_true[:])
ax.set_xlabel('x [m]')
ax.set_ylabel('y [m]')
ax.set_title('Ground Truth trajectory')
plt.show()

x_fig = plt.figure()
ax = x_fig.add_subplot(111)
ax.plot(t, (x_est[:, 0]-x_true))
ax.set_xlabel('Time [s]')
ax.set_ylabel('x [m]')
ax.set_title('Ground Truth trajectory')

y_fig = plt.figure()
ax = y_fig.add_subplot(111)
ax.plot(t, (x_est[:, 1]-y_true))
ax.set_xlabel('Time [s]')
ax.set_ylabel('y [m]')
ax.set_title('Ground Truth trajectory')

th_fig = plt.figure()
ax = th_fig.add_subplot(111)
ax.plot(t, (x_est[:, 2]-th_true))
ax.set_xlabel('Time [s]')
ax.set_ylabel('theta [rad]')
ax.set_title('Ground Truth trajectory')
plt.show()
```


Ground Truth trajectory
y [m]
x [m]


Ground Truth trajectory
x [m]
Time [s]


Ground Truth trajectory
y [m]
Time [s]



The plotted trajectory should match the ground truth. Congratulations! Assignment complete.