Introduction

In this assignment you will recursively estimate the position of a vehicle along a trajectory using available measurements and a motion model.

The vehicle is equipped with a very simple type of LIDAR sensor, which returns range and bearing measurements corresponding to individual landmarks in the environment. The global positions of the landmarks are assumed to be known beforehand. We will also assume known data association, that is, which measurment belong to which landmark.

Motion and Measurement Models

Motion Model

The vehicle motion model recieves linear and angular velocity odometry readings as inputs, and outputs the state (i.e., the 2D pose) of the vehicle:

$$\mathbf{x}_k = \mathbf{x}_{k-1} + T \begin{bmatrix} \cos\theta_{k-1} & 0 \\ \sin\theta_{k-1} & 0 \\ 0 & 1 \end{bmatrix} \left(\begin{bmatrix} v_k \\ \omega_k \end{bmatrix} + \mathbf{w}_k \right), \quad \mathbf{w}_k = \mathcal{N}(\mathbf{0}, \mathbf{Q})$$

- $\mathbf{x}_k = [x\, y\, \theta]^T$ is the current 2D pose of the vehicle
- v_k and ω_k are the linear and angular velocity odometry readings, which we use as inputs to the model

The process noise $\mathbf{w}_k$ has a (zero mean) normal distribution with a constant covariance $\mathbf{Q}$.

Measurement Model

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

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

- x_l and y_l are the ground truth coordinates of the landmark l
- x_k and y_k and θ_k represent the current pose of the vehicle
- d is the known distance between robot center and laser rangefinder (LIDAR)

The landmark measurement noise $\mathbf{n}^l_k$ has a (zero mean) normal distribution with a constant covariance $\mathbf{R}$.

Getting Started

Since the models above 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 the motion model to produce a state and covariance estimate at a given timestep, and
- the correction step, which uses the range and bearing measurements provided by the LIDAR to correct the pose and pose covariance estimates

Unpack the Data

First, let's unpack the available data:

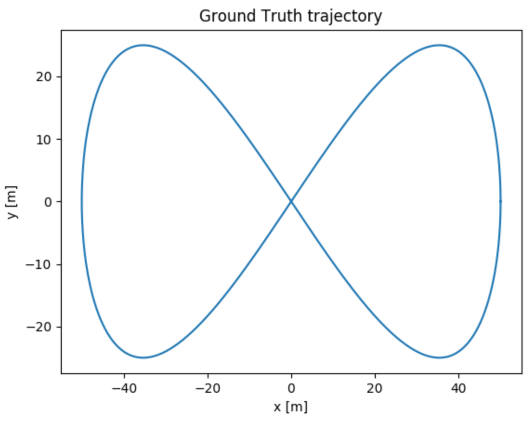
In [2]:
```
import pickle
import numpy as np
import matplotlib.pyplot as plt
import sympy

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

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

x_init  = data['x_init'] # initial x position [m]
y_init  = data['y_init'] # initial y position [m]
th_init = data['th_init'] # initial theta position [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 la
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(b.shape)
#print(r.shape)
#print(l.shape)

t_axis = np.arange(1, len(v) +1)
#plt.plot(t_axis, v, 'b')
#plt.figure(2)
#plt.plot(t_axis, om, 'g')
#plt.figure(3)
#plt.plot(t_axis, r, 'g')
#plt.figure(4)
#plt.plot(t_axis, b, 'g')
#plt.show()
print(d)
```

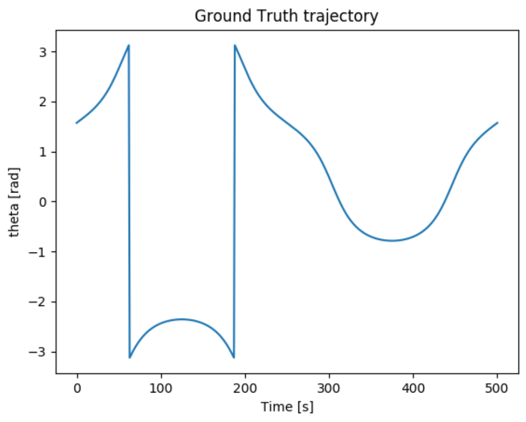
```
[0]
```

Note that distance from the LIDAR frame to the robot center is provided and loaded as an array into the `d` variable.

Ground Truth

If available, it is useful to plot the ground truth position and orientation before starting the assignment.





Notice that the orientation values are wrapped to the $[-\pi, \pi]$ range in radians.

Initializing Parameters

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

In [47]:
```
v_var = 0.1  # translation velocity variance
om_var = 0.1  # rotational velocity variance
r_var = 0.1  # range measurements variance
b_var = 0.01  # bearing measurement variance

Q_km = np.diag([v_var, om_var]) # input noise covariance
cov_y = 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_init, y_init, th_init]) # initial state
P_est[0] = np.diag([1, 1, 0.1]) # initial state covariance


#print(x_est)
#print(x_est.shape)
#print(P_est)
#print(P_est.shape)
#print(cov_y.shape)
print(Q_km.shape)
#print(Q_km)
```

```
(2, 2)
```

Remember: that it is neccessary to tune the measurement noise variances `r_var`, `b_var` in order for the filter to perform well!

In order for the orientation estimates to coincide with the bearing measurements, it is also neccessary to wrap all estimated θ values to the $(-\pi, \pi]$ range.

In [43]:
```
# Wraps angle to (-pi,pi] range
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
```

Correction Step

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 received at a given timestep k, you should implement the following steps:

- Compute the measurement model Jacobians at $\check{\mathbf{x}}_k$

$$\mathbf{y}_k^l = \mathbf{h}(\mathbf{x}_k, \mathbf{n}_k^l)$$

$$\mathbf{H}_k = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}_k}\right|_{\check{\mathbf{x}}_k,0}, \quad \mathbf{M}_k = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{n}_k}\right|_{\check{\mathbf{x}}_k,0}.$$

- Compute the Kalman Gain

$$\mathbf{K}_k = \check{\mathbf{P}}_k \mathbf{H}_k^T \left(\mathbf{H}_k \check{\mathbf{P}}_k \mathbf{H}_k^T + \mathbf{M}_k \mathbf{R}_k \mathbf{M}_k^T\right)^{-1}$$

- Correct the predicted state

$$\check{\mathbf{y}}_k^l = \mathbf{h}\left(\check{\mathbf{x}}_k, \mathbf{0}\right)$$
$$\hat{\mathbf{x}}_k = \check{\mathbf{x}}_k + \mathbf{K}_k \left(\mathbf{y}_k^l - \check{\mathbf{y}}_k^l\right)$$

- Correct the covariance

$$\hat{\mathbf{P}}_k = \left(\mathbf{I} - \mathbf{K}_k \mathbf{H}_k\right) \check{\mathbf{P}}_k$$

In [84]:
```
def measurement_update(lk, rk, bk, P_check, x_check):

    # 1. Compute measurement Jacobian
    x  = x_check[0,0]
    y  = x_check[0,1]
    th = wraptopi(x_check[0,2])

    xl = lk[0]
    yl = lk[1]

    star1 =xl - x - d*np.cos(th)
    star2 =yl - y - d*np.sin(th)
    frac = star1**2+star2**2
    den  = np.sqrt(frac)

    Hk=np.array([
                [-star1/den,     -star2/den,     (star1*d*np.sin(th)-star2*d*np.c
                [star2/frac,     -star1/frac,    -1- (d*np.sin(th)*star2/frac) - 
                ])

    Hk = Hk.reshape((2,3))
    Mk=np.identity(2)

    # 2. Compute Kalman Gain
    #print('Hk shape is    ' + str(Hk.shape))
    #print('cov_y shape is ' + str(cov_y.shape))
    #print('Mk shape is    ' + str(Mk.shape))

    S  = np.matmul(Hk, np.matmul(P_check, Hk.T)) + np.matmul(Mk, np.matmul(cov_y,
    #S  = np.matmul(Hk, np.matmul(P_check, Hk.T)) + cov_y
    Kk = np.matmul(np.matmul(P_check, Hk.T), np.linalg.inv(S) )
    #print('Kk shape is    ' + str(Kk.shape))

    # 3. Correct predicted state (remember to wrap the angles to [-pi,pi])
    yk=np.array([den, wraptopi(np.arctan2(star2,star1)-wraptopi(th))]).reshape((2
    ym=np.array([rk,wraptopi(bk)]).reshape((2,1))

    temp = (ym-yk)
    #print('ym shape is    ' + str(ym.shape))
    #print('yk shape is    ' + str(yk.shape))
    #print('temp shape is  ' + str(temp.shape))

    #print('residual correction is ' + str(temp))
    x_check = x_check + np.matmul(Kk, temp).T
    #print('x_check shape is    ' + str(x_check.shape))
    x_check[0,2]=wraptopi(x_check[0,2])

    # 4. Correct covariance
    P_check= np.matmul( (np.identity(3)-np.matmul(Kk,Hk)) , P_check )

    return x_check, P_check
```

Prediction Step

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

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

Where

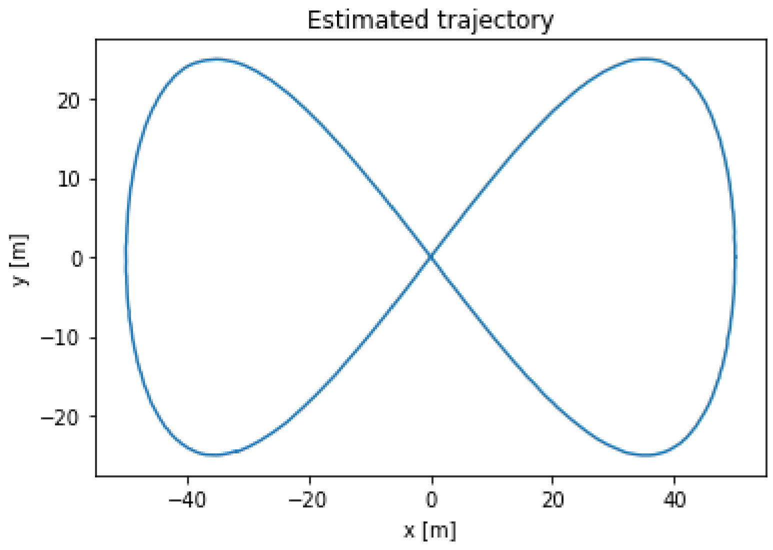
$$\mathbf{F}_{k-1} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}_{k-1}}\right|_{\hat{\mathbf{x}}_{k-1},\mathbf{u}_k,0}, \quad \mathbf{L}_{k-1} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{w}_k}\right|_{\hat{\mathbf{x}}_{k-1},\mathbf{u}_k,0} .$$

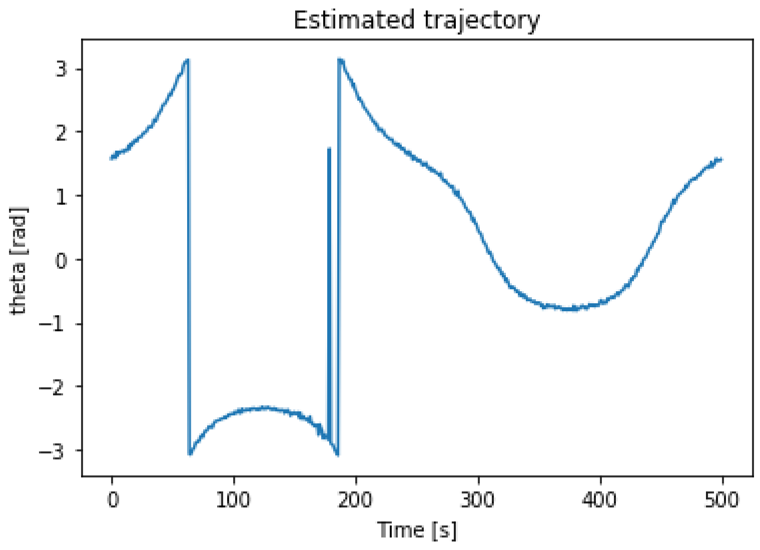
In [90]:
```
#### 5. Main Filter Loop ##########################################################
x_check = x_est[0,:]
x_check = x_check.reshape((1,3))
P_check = P_est[0]

#print(x_check.shape)
print(x_check)
#print(P_check.shape)

for k in range(1, len(t)):
#for k in range(1, 4):

    delta_t = t[k] - t[k - 1]  # time step (difference between timestamps)
    #print('x_check is    ' + str(x_check))
    theta   = wraptopi(x_check[0,2])
    Fu      = np.array([
                        [np.cos(theta), 0],
                        [np.sin(theta), 0],
                        [0,1]
                       ])
    inputc = np.array([v[k-1],om[k-1]])
    #print('inputc is    ' + str(inputc))

    #1.predict state using non-linear equation
    x_check = x_check + delta_t * (np.matmul(Fu, inputc.T)).T
    x_check[0,2]=wraptopi(x_check[0,2])
    #print('x_check after prediction is    ' + str(x_check))

    #propagating the covariance
    Fx       = np.array([
                        [1,0, -delta_t*np.sin(theta) * v[k-1]],
                        [0,1, delta_t*np.cos(theta)  * v[k-1]],
                        [0,0,1]
                        ])



    Lx = delta_t * Fu
    #Lx = np.array([
     #              [1,0],
     #              [1,0],
     #              [0,1]
     #              ])

    #print(Lx.shape)
    P_check = np.matmul(Fx, np.matmul(P_check, Fx.T)) + np.matmul(Lx, np.matmul(Q
    #P_check = np.matmul(Fx, np.matmul(P_check, Fx.T)) + Q_km
    #print('P_check shape is    ' + str(P_check.shape))

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


    # Set final state predictions for timestep
    x_est[k, 0] = x_check[0,0]
    x_est[k, 1] = x_check[0,1]
    x_est[k, 2] = wraptopi(x_check[0,2])
    P_est[k, :, :] = P_check
```

```
[[50.          0.          1.57079633]]
```

Let's plot the resulting state estimates:

In [91]:
```
e_fig = plt.figure()
ax = e_fig.add_subplot(111)
#ax.plot(x_est[0:4, 0], x_est[0:4, 1])
#ax.plot(x_est[0:4, 0], x_est[0:4, 1], 'ro')
ax.plot(x_est[:, 0], x_est[:, 1])
ax.set_xlabel('x [m]')
ax.set_ylabel('y [m]')
ax.set_title('Estimated trajectory')
plt.show()

e_fig = plt.figure()
ax = e_fig.add_subplot(111)
#ax.plot(t[0:4], x_est[0:4, 2])
ax.plot(t[:], x_est[:, 2])
ax.set_xlabel('Time [s]')
ax.set_ylabel('theta [rad]')
ax.set_title('Estimated trajectory')
plt.show()
```





Are you satisfied wth your results? The resulting trajectory should closely resemble the ground truth, with minor "jumps" in the orientation estimate due to angle wrapping. If this is the case, run the code below to produce your solution file.

In [92]:
```
with open('submission.pkl', 'wb') as f:
    pickle.dump(x_est, f, pickle.HIGHEST_PROTOCOL)
```

In [ ]: