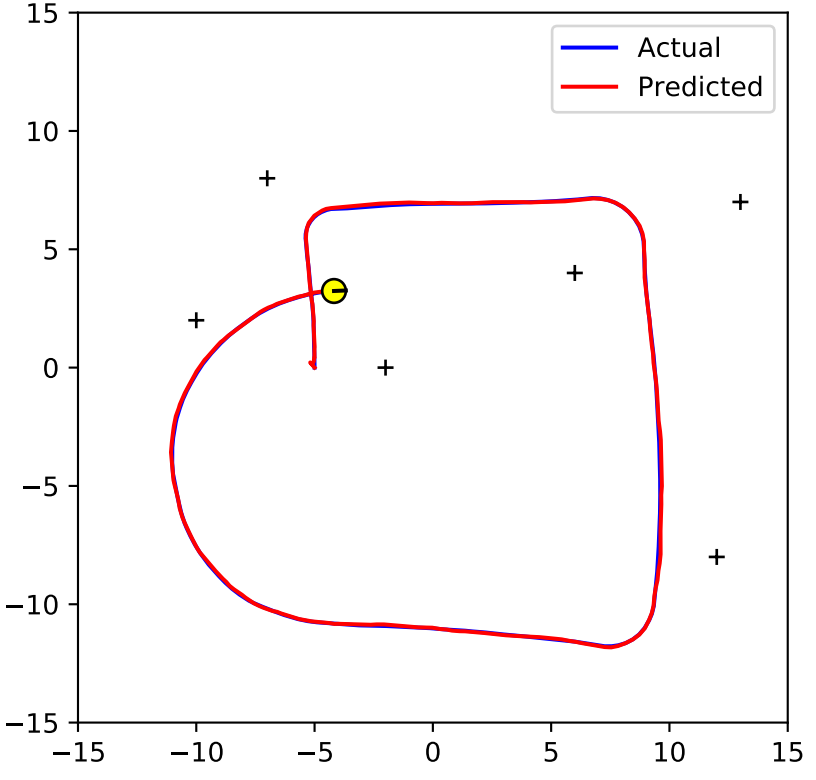
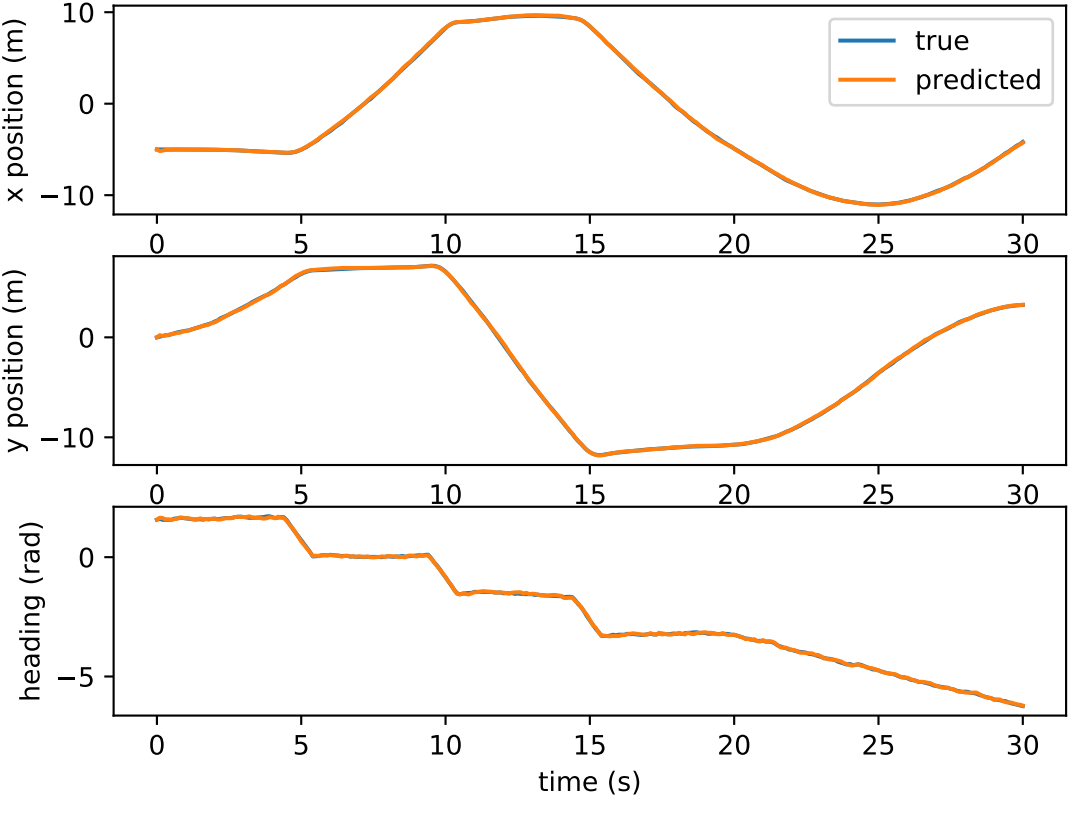
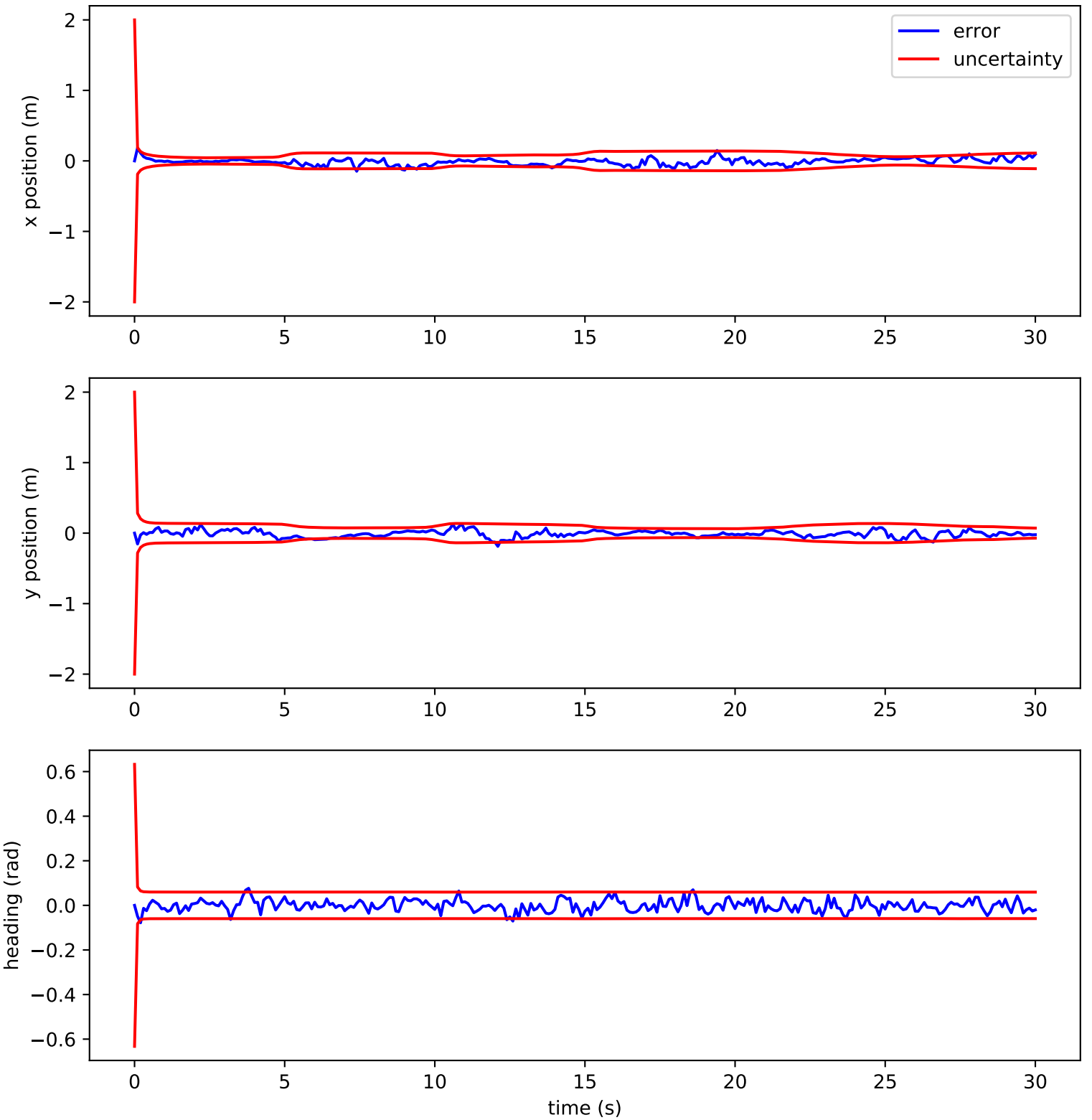
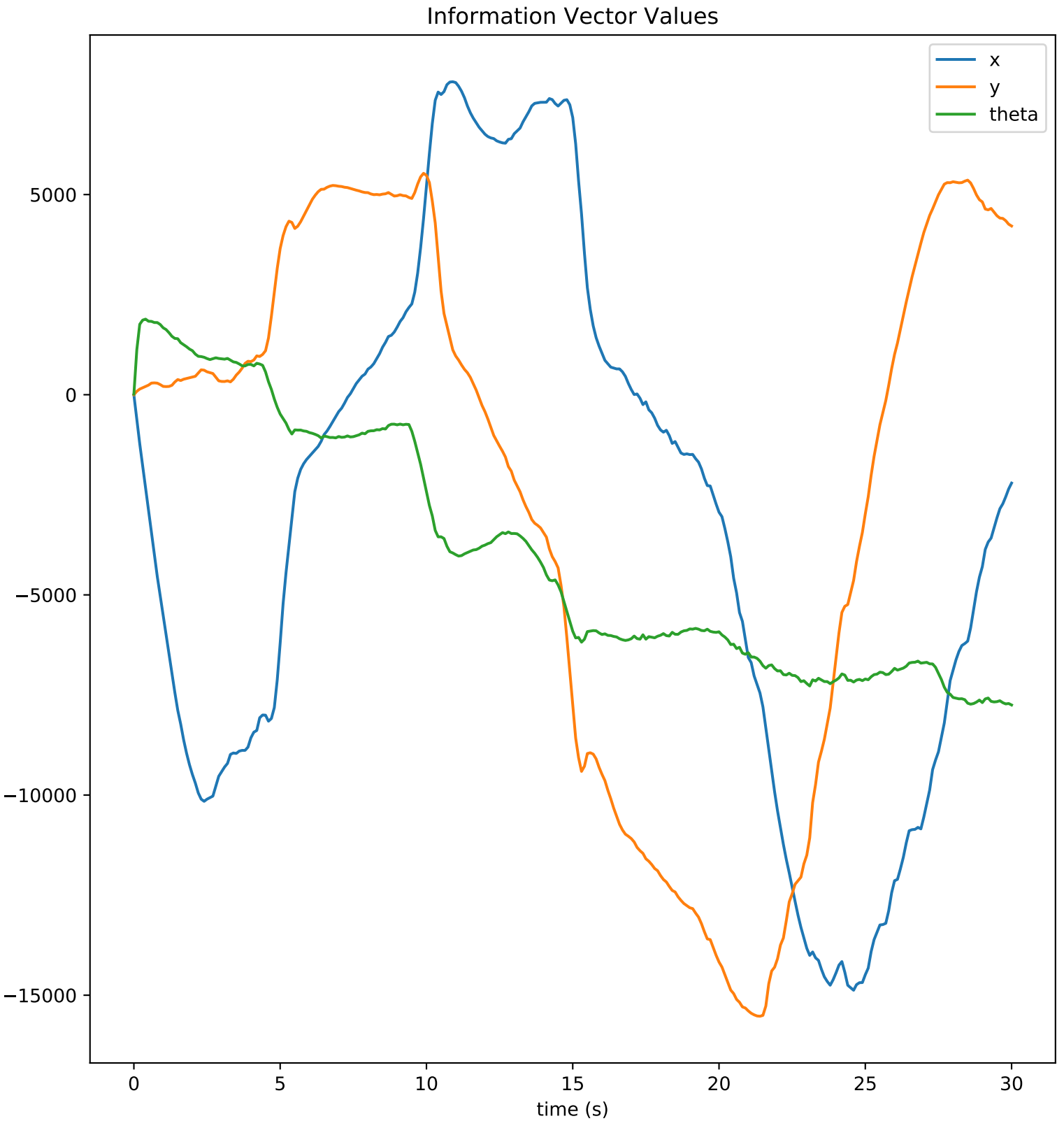

Actual
Predicted
15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15


x position (m)
y position (m)
heading (rad)
time (s)
true
predicted
10
0
−10
−5
0
5
10
15
20
25
30


error
uncertainty
x position (m)
y position (m)
heading (rad)
time (s)
2
1
0
−1
−2
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
5
10
15
20
25
30


Information Vector Values
x
y
theta
5000
0
−5000
−10000
−15000
0
5
10
15
20
25
30
time (s)

```
import matplotlib.pyplot as plt
import numpy as np
from numpy import cos, sin, arctan2
from numpy import matmul as mm
from numpy.linalg import inv as mat_inv
from scipy.io import loadmat
import pdb
from numpy.random import normal as randn
from matplotlib import animation

def animate(true_states, belief_states, markers):
    x_tr, y_tr, th_tr = true_states
    x_guess, y_guess = belief_states

    radius = .5
    yellow = (1,1,0)
    black = 'k'
    world_bounds = [-15,15]

    fig = plt.figure()
    ax = plt.axes(xlim=world_bounds, ylim=world_bounds)
    ax.set_aspect('equal')
    ax.plot(markers[0], markers[1], '+', color=black)
    actual_path, = ax.plot([], [], color='b', zorder=-2, label="Actual")
    pred_path, = ax.plot([], [], color='r', zorder=-1, label="Predicted")
    heading, = ax.plot([], [], color=black)
    robot = plt.Circle((x_tr[0],y_tr[0]), radius=radius, color=yellow, ec=black)
    ax.add_artist(robot)
    ax.legend()

    def init():
        actual_path.set_data([], [])
        pred_path.set_data([], [])
        heading.set_data([], [])
        return actual_path, pred_path, heading, robot

    def animate(i):
        actual_path.set_data(x_tr[:i+1], y_tr[:i+1])
        pred_path.set_data(x_guess[:i+1], y_guess[:i+1])
        heading.set_data([x_tr[i], x_tr[i] + radius*cos(th_tr[i])],
            [y_tr[i], y_tr[i] + radius*sin(th_tr[i])])
        robot.center = (x_tr[i],y_tr[i])
        return actual_path, pred_path, heading, robot

    anim = animation.FuncAnimation(fig, animate, init_func=init,
        frames=len(x_tr), interval=20, blit=True, repeat=False)

    plt.pause(.1)
    input("<Hit enter to close>")

def propogate(vel, angular_vel, old_state, delta_t):
    theta = old_state[2,0]
    motion_vec = np.array([
                            [vel * cos(theta) * delta_t],
                            [vel * sin(theta) * delta_t],
                            [angular_vel * delta_t]
                        ])
    return old_state + motion_vec

def wrap(angle):
    # map angle between -pi and pi
    return (angle + np.pi) % (2 * np.pi) - np.pi

if __name__ == "__main__":
    ''' read data from matlab file '''
    mat_file = loadmat('midterm_data.mat')
    # true states (x, y, theta)
    X_tr = mat_file['X_tr']
    # measurements for true states (with measurement noise)
```

```
    range_tr = mat_file['range_tr']
    bearing_tr = mat_file['bearing_tr']
    # command velocities
    v_c = mat_file['v_c']
    om_c = mat_file['om_c']
    v_c = v_c[0,:]
    om_c = om_c[0,:]
    # actual velocities (with noise)
    v = mat_file['v']
    om = mat_file['om']
    v = v[0,:]
    om = om[0,:]
    # times
    t = mat_file['t']
    t = t[0,:]

    # timestep
    dt = t[1] - t[0]

    # landmarks (x and y coordinates)
    lm_x = [6, -7, 12, -2, -10, 13]
    lm_y = [4, 8, -8, 0, 2, 7]
    assert(len(lm_x) == len(lm_y))
    num_landmarks = len(lm_x)

    # initial state & lists of state predictions
    state = np.array([[-5], [0], [np.pi/2]])
    x_pred = [state[0,0]]
    y_pred = [state[1,0]]
    th_pred = [state[2,0]]

    # initial uncertainty & lists of parameter uncertanties (95% confidence interval)
    sigma = np.array([
                        [1, 0, 0],  # x
                        [0, 1, 0],  # y
                        [0, 0, .1]  # theta
                    ])
    bound_x = [np.sqrt(sigma[0 , 0]) * 2]
    bound_y = [np.sqrt(sigma[1 , 1]) * 2]
    bound_theta = [np.sqrt(sigma[2 , 2]) * 2]

    info_matrix = mat_inv(sigma)
    info_vector = mm(info_matrix, state)

    # noise for control inputs
    std_dev_v = .15 # m/s
    std_dev_om = .1 # rad/s

    # noise for measurement
    std_dev_range = .2      # m
    std_dev_bearing = .1    # rad
    var_range = std_dev_range * std_dev_range
    var_bearing = std_dev_bearing * std_dev_bearing
    Q_t = np.array([
                    [ var_range, 0 ],
                    [ 0,         var_bearing ]
                ])

    # information vector information
    info_x = [info_vector[0,0]]
    info_y = [info_vector[1,0]]
    info_th = [info_vector[2,0]]

    ''' run EIF '''
    for i in range(1,t.size):
        prev_state = mm(mat_inv(info_matrix), info_vector)
        theta = prev_state[2,0]

        G_t = np.array([
```

```
                    [ 1, 0, -v_c[i]*sin(theta)*dt ],
                    [ 0, 1,  v_c[i]*cos(theta)*dt ],
                    [ 0, 0,  1 ]
                ])
        V_t = np.array([
                    [ cos(theta)*dt, 0 ],
                    [ sin(theta)*dt, 0 ],
                    [ 0,                dt ]
                ])
        M_t = np.array([
                    [ std_dev_v, 0 ],
                    [ 0,         std_dev_om ]
                ])
        R_t = mm( mm(V_t, M_t), np.transpose(V_t) )

        a = mm( mm(G_t, mat_inv(info_matrix)), np.transpose(G_t) )
        bar_info_matrix = mat_inv(a + R_t)

        mu_bar = propogate(v_c[i], om_c[i], prev_state, dt)

        bar_info_vector = mm(bar_info_matrix, mu_bar)

        for j in range(num_landmarks):
        # for j in range(1):
            m_j_x = lm_x[j]
            m_j_y = lm_y[j]
            bel_x = mu_bar[0 , 0]
            bel_y = mu_bar[1 , 0]
            bel_theta = mu_bar[2 , 0]

            # get predicted measurement and jacobian
            diff_x = m_j_x - bel_x
            diff_y = m_j_y - bel_y
            q = (diff_x ** 2) + (diff_y ** 2)
            z_hat = np.array([
                                [np.sqrt(q)],
                                [arctan2(diff_y, diff_x) - bel_theta]
                            ])
            H_t = np.array([
                            [-diff_x / np.sqrt(q), -diff_y / np.sqrt(q), 0],
                            [diff_y / q, -diff_x / q, -1]
                            ])

            a = mm( mm(np.transpose(H_t), mat_inv(Q_t)), H_t )
            bar_info_matrix += a

            z_t = np.array([
                            [ range_tr[i,j] ],
                            [ bearing_tr[i,j] ]
                            ])
            z_diff = z_t - z_hat
            z_diff[1,0] = wrap(z_diff[1,0]) # wrapping here is IMPORTANT!!!
            a = z_diff + + mm(H_t, mu_bar)
            b = mm(np.transpose(H_t), mat_inv(Q_t))
            bar_info_vector += mm(b, a)

            # update mu_bar to reflect information gain (useful for next landmark)
            mu_bar = mm(mat_inv(bar_info_matrix), bar_info_vector)

        info_matrix = bar_info_matrix
        info_vector = bar_info_vector

        # save info for plotting later
        x_pred.append(mu_bar[0,0])
        y_pred.append(mu_bar[1,0])
        th_pred.append(mu_bar[2,0])
        sigma = mat_inv(info_matrix)
        bound_x.append(np.sqrt(sigma[0 , 0]) * 2)
        bound_y.append(np.sqrt(sigma[1 , 1]) * 2)
```

```
        bound_theta.append(np.sqrt(sigma[2 , 2]) * 2)
        info_x.append(info_vector[0,0])
        info_y.append(info_vector[1,0])
        info_th.append(info_vector[2,0])

    ''' algorithm is done; plotting and animating results '''

    # plot the states over time
    p1 = plt.figure(1)
    plt.subplot(311)
    plt.plot(t, X_tr[0,:], label="true")
    plt.plot(t, x_pred, label="predicted")
    plt.ylabel("x position (m)")
    plt.legend()
    plt.subplot(312)
    plt.plot(t, X_tr[1,:])
    plt.plot(t, y_pred)
    plt.ylabel("y position (m)")
    plt.subplot(313)
    plt.plot(t, X_tr[2,:])
    plt.plot(t, th_pred)
    plt.ylabel("heading (rad)")
    plt.xlabel("time (s)")
    plt.draw()

    # plot the uncertainty in states over time
    p2 = plt.figure(2)
    plt.subplot(311)
    plt.plot(t, np.array(X_tr[0,:]) - np.array(x_pred), color='b', label="error")
    plt.plot(t, bound_x, color='r', label="uncertainty")
    plt.plot(t, [x * -1 for x in bound_x], color='r')
    plt.ylabel("x position (m)")
    plt.legend()
    plt.subplot(312)
    plt.plot(t, np.array(X_tr[1,:]) - np.array(y_pred), color='b')
    plt.plot(t, bound_y, color='r')
    plt.plot(t, [x * -1 for x in bound_y], color='r')
    plt.ylabel("y position (m)")
    plt.subplot(313)
    plt.plot(t, np.array(X_tr[2,:]) - np.array(th_pred), color='b')
    plt.plot(t, bound_theta, color='r')
    plt.plot(t, [x * -1 for x in bound_theta], color='r')
    plt.ylabel("heading (rad)")
    plt.xlabel("time (s)")
    plt.draw()

    # plot information vector values
    p3 = plt.figure(3)
    plt.plot(t, info_x, label="x")
    plt.plot(t, info_y, label="y")
    plt.plot(t, info_th, label="theta")
    plt.xlabel("time (s)")
    plt.title("Information Vector Values")
    plt.legend()
    plt.draw()

    animate( (X_tr[0,:], X_tr[1,:], X_tr[2,:]) , (x_pred, y_pred) , (lm_x, lm_y) )
```