EE209AS (Fall 2018) Problem Set 3

Jingwen Zhang

October 28, 2018

Preliminaries

0(a). Github repo:
`https://github.com/Jingwen-Zhang-Aaron/Computational-Robotics.git`
The fully commented script is under the folder "pset3-KalmanFilter".

0(b). I did it individually. I sort of used github repo:
`https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python.git`
as my reference.

0(c). I coded all problems under IPython environment with Jupyter Notebook.

0(d). I contributed 100% of this work.

Robot Motion Model

When there is a mobile robot as shown in Figure 1, let the $w = 85mm$ be the distance between the wheels and $r = 20mm$ be the radius of the wheel. Assuming the robot traveled a very short distance during time Δt, the angular velocity values (ω_L, ω_R) of the left and right wheels are our inputs.

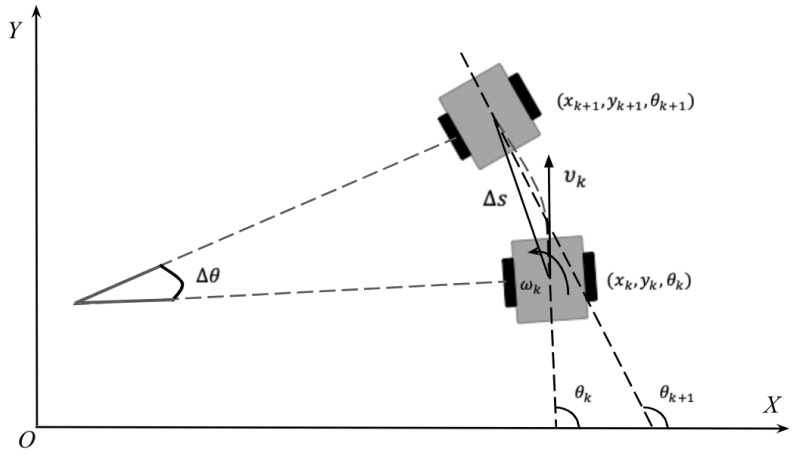


Figure 1: Model Derivation of the Laser Range Sensor

The velocity of the left and right wheel (V_L, V_R) can be calculated. Then from the left and right wheel velocity, linear velocity v_k and angular velocity ω_k of the robot can be obtained as follows:

$$
\begin{aligned}
V_L &= \omega_L r \\
V_R &= \omega_R r \\
v_k &= \frac{V_L + V_R}{2} \\
\omega_k &= \frac{V_R - V_L}{w}
\end{aligned}
$$

Finally, we can obtain the next position (x_{k+1}, y_{k+1}) and the next orientation θ_{k+1}:

$$
\begin{aligned}
\Delta s = v_k \Delta t \quad \Delta\theta &= \omega_k \Delta t \\
x_{k+1} &= x_k + \Delta s \cos(\theta_k + \frac{\Delta\theta}{2}) \\
y_{k+1} &= y_k + \Delta s \sin(\theta_k + \frac{\Delta\theta}{2}) \\
\theta_{k+1} &= \theta_k + \Delta\theta
\end{aligned}
$$

Design the State Variables

The state of our robot will consist of the 3DOF pose of the robot in 2D space:

$$
\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^T
$$

where $\theta \in [-\pi, \pi)$.

The control input will be the angular velocity values:

$$
\mathbf{u} = \begin{bmatrix} \omega_L & \omega_R \end{bmatrix}^T
$$

Design the System Model

We can model the system as a nonlinear model plus white noise.

$$
\bar{\mathbf{x}} = \mathbf{x} + f(\mathbf{x}, \mathbf{u}) + N(0, Q)
$$

where $f(\mathbf{x}, \mathbf{u})$ is already obtained in section **Robot Motion Model**. The implementation in Python is as follows:

```
def move(x, dt, u):
    """ takes current state variable and current input and returns
    the next state variable given a small dt. """
    # velocities of left and right wheels
    V_L, V_R = u[0]*r, u[1]*r

    # linear velocity and angluar velocity of the robot
    v_k = (V_L + V_R) / 2.0
    omega_k = (V_R - V_L) / w
```

```
    # linear displacement and angular displacement
    delta_s = v_k*dt
    delta_theta = omega_k*dt
    delta_x = delta_s * np.cos(x[2] + delta_theta/2)
    delta_y = delta_s * np.sin(x[2] + delta_theta/2)

    return x + np.array([delta_x, delta_y, delta_theta])
```

Design the Measurement Model

We have two laser range sensors to get the distance to a wall in a straight line in front of the robot and the distance to a wall is a straight line to the right of the robot. At first, we consider a straight line with four boundaries. The problem we need to figure out is that what is the intersection point between the line and boundaries. As shown in Figure 2 (1), we need to decide the the intersection point is on the vertical boundary ($x = L$) or the horizontal boundary ($y = W$). The expression of the line is:

$$\frac{y_{new} - y}{x_{new} - x} = \tan\theta$$

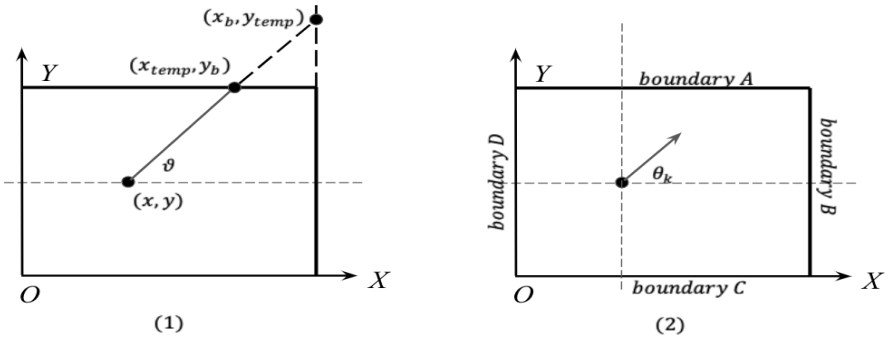


Figure 2: Model Derivation of the Laser Range Sensor

Now suppose the boundaries are:

$$x = x_b$$
$$y = y_b$$

We calculate points (x_{temp}, y_b) and (x_b, y_{temp}) as follows:

$$x_{temp} = \frac{y_b - y}{\tan\theta} + x$$
$$y_{temp} = (x_b - x)\tan\theta + y$$

Then, we can decide which point to choose. For example in the Figure 2, we should choose (x_{temp}, y_b) because the point (x_b, y_{temp}) is already out of the range. The python implementation is as follows:

```
def intersection_point(x, y, theta, boundary):
    """
    Calculate the intersection point b/n a line and the boundary
```

```
    Args:
        x, y: (x,y) is the starting point of the line.
        theta: the angle indicating the slope of the line.
        boundary: [x_b, y_b] indicating the the horizontal boundary y = y_b
              and the vertical boundary x = x_b.

    Returns:
        the position (x_r, y_r) of the intersection point.
    """
    if (theta in [0.0, -np.pi/2, np.pi/2, np.pi]):
        raise ValueError('The line is special cases')

    x_b = boundary[0]
    y_b = boundary[1]
    y_temp = (x_b - x)*math.tan(theta) + y
    x_temp = (y_b - y)/math.tan(theta) + x

    if (x_temp >= 0 and x_temp <= L):
        # the point is on the horizontal boundary y=y_b
        return [x_temp, y_b]
    else:
        # the point is on the vertical boundary x=x_b
        return [x_b, y_temp]
```

Based on this function, now we need to find the reflection point of a laser given its starting point and direction. As shown in Figure 2 (2), we can see there are 4 boundaries A, B, C and D. And based on the state of the robot, we can divide the environment into 4 areas:

$$\begin{aligned} Area1 \quad & -\pi \leqslant \theta < -\pi/2 \\ Area2 \quad & -\pi/2 \leqslant \theta < 0 \\ Area3 \quad & 0 \leqslant \theta < \pi/2 \\ Area4 \quad & \pi/2 \leqslant \theta < \pi \end{aligned}$$

Inside each area, the reflection point can only be located in two possible boundaries like we've discussed above. For example, in area 3, it's only possible for the point to show up on boundary A or B. So we can define the boundary conditions as follows:

$$\begin{cases} Area1 & (x_b, y_b) = (0, 0) \\ Area2 & (x_b, y_b) = (L, 0) \\ Area3 & (x_b, y_b) = (L, W) \\ Area4 & (x_b, y_b) = (0, W) \end{cases}$$

And then we can use the function *intersection_point* to find the reflection point:

```
def reflection_point(x, y, theta):
    """
    Find the reflection point given the position of robot and the direction of
        the laser range sensor.

    Args:
        x, y: (x,y) is the point of the robot (with laser range sensor)
```

```
    theta: the angle indicating the direction of the laser.

Returns:
    the position (x_r, y_r) of the reflection point.
"""
# normalize the angle into [-pi, pi)
theta = normalize_angle(theta)

# Area 1
if (theta >= -np.pi and theta < -np.pi/2):
    if theta != -np.pi:
        boundary = [0, 0]
        point = intersection_point(x, y, theta, boundary)
    else:
        point = [0, y]
# Area 2
if (theta >= -np.pi/2 and theta < 0.0):
    if theta != -np.pi/2:
        boundary = [L, 0]
        point = intersection_point(x, y, theta, boundary)
    else:
        point = [x, 0]
# Area 3
if (theta >= 0.0 and theta < np.pi/2):
    if theta != 0.0:
        boundary = [L, W]
        point = intersection_point(x, y, theta, boundary)
    else:
        point = [L, y]
# Area 4
if (theta >= np.pi/2 and theta < np.pi):
    if theta != np.pi/2:
        boundary = [0, W]
        point = intersection_point(x, y, theta, boundary)
    else:
        point = [x, W]
return point
```

After finding reflection points p_1 of the front-side laser range sensor and p_2 of the right-side laser range sensor, we can compute range r_{front} and r_{right} as follows:

$$r_{front} = \sqrt{(p_{1x} - x)^2 + (p_{1y} - y)^2}$$
$$r_{right} = \sqrt{(p_{2x} - x)^2 + (p_{2y} - y)^2}$$

For the output of the IMU, we can just use the absolute bearing. Suppose the direction of magnetic north is +x. So:

$$\phi = \theta$$

Thus our measurement function is

$$\begin{aligned}\mathbf{z} &= h(\mathbf{x}, \mathbf{P}) + N(0, R) \\ &= \begin{bmatrix} \sqrt{(p_{1x} - x)^2 + (p_{1y} - y)^2} \\ \sqrt{(p_{2x} - x)^2 + (p_{2y} - y)^2} \\ \theta \end{bmatrix} + N(0, R)\end{aligned}$$

It is reasonable to assume that the range and bearing measurement noise is independent, hence

$$R = \begin{bmatrix} \sigma^2_{range1} & & \\ & \sigma^2_{range2} & \\ & & \sigma^2_{bearing} \end{bmatrix}$$

The measurement model is implemented as follows:

```
def Hx(x):
    """ takes a state variable and returns the measurement
    that would correspond to that state. """
    # Reflection point of the front side
    p1 = reflection_point(x[0], x[1], x[2])
    # Reflection point of the right side
    p2 = reflection_point(x[0], x[1], x[2]-np.pi/2)

    r_front = np.sqrt((p1[0] - x[0])**2 + (p1[1] - x[1])**2)
    r_right = np.sqrt((p2[0] - x[0])**2 + (p2[1] - x[1])**2)
    angle = normalize(x[2])
    return np.array([r_front, r_right, angle])
```

Choose Kalman Filter

We have two candidates, one is Extended Kalman Filter (EKF) while another is Unscented Kalman Filter (UKF). I chose UKF mainly based on following reasons:

(1) The implementation of UKF is much easier, especially with the existing python library (I used *filterpy.kalman*) while the EKF for the same problem requires fairly difficult mathematics. EKF needs to compute the Jacobians for the state and measurement models which is always difficult even with some simple motion model. But UKF just requires the nonlinear state model $f(x)$ and measurement model $h(x)$.

(2) There are many cases where the Jacobian cannot be found analytically. If so, you need to use special numerical methods to compute the Jacobian in every predict-and-update loop. In this problem, it is not hard to get the analytical Jacobian of the state model $f(x)$. But for the $h(x)$, I didn't find a good way to do the linearization. It's highly nonlinear due to the nonlinear change of the reflection point of the laser range sensors.

(3) From my perspective, I think UKF is more accurate than EKF. The EKF works by linearizing the system model and measurement model at a single point while the UKF uses $2n + 1$ points.

Unscented Kalman Filter (UKF) Implementation

Although I use the library "FilterPy" as follows, I will introduce the equations behind the code. We already know Monte Carlo approach and we can use the mean and variance of a distribution to represent a nonlinear function without knowing it analytically. Sampling requires no specialized knowledge and does not require a closed form solution. No matter how nonlinear or poorly behaved the function is, as long as we sample with enough points we will build an accurate output distribution. That's how UKF works.

```
from filterpy.kalman import MerweScaledSigmaPoints
from filterpy.kalman import UnscentedKalmanFilter as UKF

def ukf_build(x, P, sigma_range, sigma_bearing, dt=1.0):
   """ construct Unscented Kalman Filter with the initial state x
   and the initial covaiance matrix P

   sigma_range: the std of laser range sensors
   sigma_bearing: the std of IMU
   """
   # construct the sigma points
   points = MerweScaledSigmaPoints(n=3, alpha=0.001, beta=2, kappa=0,
                       subtract=residual)

   # build the UKF based on previous functions
   ukf = UKF(dim_x=3, dim_z=3, fx=move, hx=Hx,
       dt=dt, points=points, x_mean_fn=state_mean,
       z_mean_fn=z_mean, residual_x=residual,
       residual_z=residual)

   # assign the parameters of ukf
   ukf.x = np.array(x)
   ukf.P = P
   ukf.R = np.diag([sigma_range**2, sigma_range**2, sigma_bearing**2])
   ukf.Q = np.eye(3)*0.0001
   return ukf
```

1. Build Sigma Points

Sampling points in UKFs are called sigma points which are from a specific distribution instead of totally random. Actually, there're many different algorithms to generate sigma points from a distribution to improve the performance of UKFs. I used Van der Merwe's Scaled Sigma Point Algorithm. The set of sigma points χ and corresponding weights W^m which is used to compute the mean and W^c which is used to compute the covariance are generated:

$$\chi = sigma\_fn(x, P, parameters)$$
$$W^m, W^c = weight\_fn(n, parameters)$$

where $x = \mu$ is the mean of initial guess, $P = \Sigma$ is the covariance matrix.

In Van der Merwe's Scaled Sigma Point Algorithm, the $weight\_fn$ is:

$$\begin{aligned}
\lambda &= \alpha^2(n+\kappa) - n \\
W_0^m &= \frac{\lambda}{n+\lambda} \\
W_0^c &= \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta \\
W_i^m &= W_i^c = \frac{1}{2(n+\lambda)} \qquad i = 1 \cdots 2n
\end{aligned}$$

and the $sigma\_fn$ is:

$$\begin{aligned}
\chi_0 &= \mu \\
\chi_i &= \mu + \left[\sqrt{(n+\lambda)\Sigma}\right]_i \qquad i = 1 \cdots n \\
\chi_i &= \mu - \left[\sqrt{(n+\lambda)\Sigma}\right]_{i-n} \quad i = (n+1) \cdots 2n
\end{aligned}$$

That's what we did with the code:

```
points = MerweScaledSigmaPoints(n=3, alpha=0.001, beta=2, kappa=0,
                         subtract=residual)
```

where n is the number of states, $\kappa = 3 - n$, $\beta = 2$ for Gaussian problems and $0 < \alpha < 1$.

2. Predict Step

Pass each sigma point through the state model $\bar{x} = f(x, \Delta t)$ forming the new prior X:

$$\mathrm{X} = f(\chi, \Delta t)$$

With the unscented transform, we can get the mean and covariance of the prior:

$$\begin{aligned}
\bar{x} &= \sum_{i=0}^{2n} W_i^m X_i \\
\bar{P} &= \sum_{i=0}^{2n} W_i^c (X_i - \bar{x})(X_i - \bar{x})^T + Q
\end{aligned}$$

That's what we did with the code:

```
ukf.predict(u=u)
```

3. Update Step

Convert the sigma points of the prior into measurements model $z = h(x)$:

$$\mathrm{Z} = h(X)$$

Similarly, compute the mean and covariance of Z:

$$\begin{aligned}
\mu_z &= \sum_{i=0}^{2n} W_i^m \mathrm{Z}_i \\
P_z &= \sum_{i=0}^{2n} W_i^c (\mathrm{Z}_i - \mu_z)(\mathrm{Z}_i - \mu_z)^T + R
\end{aligned}$$

Then compute the residual and Kalman gain:

$$y = z - \mu_z$$
$$K = \left[\sum_{i=0}^{2n} W_i^c (X_i - \bar{x})(Z_i - \mu_z)^T\right] P_z^{-1}$$

With it, we can get the mean and covariance of the posterior:

$$x = \bar{x} + Ky$$
$$P = \bar{P} - KP_zK^T$$

That's what we did with the code:

```
ukf.update(z)
```

4. Implementation Issues

As I described above, we need to do compute the residual $y = z - h(x)$ and also use $(X_i - \bar{x})$. In this problem, we have the angle in $z[2]$ and $x[2]$. Suppose we have an angle of $1°$ and another of $361°$. What's the difference? Without any modification, you will get $360°$. But actually it's 0. So we need to normalize all the angle into $[-\pi,\ \pi)$ as follows:

```
def normalize_angle(x):
   x = x % (2 * np.pi) # force in range [0, 2pi)
   if x >= np.pi:    # move to [-pi, pi)
      x -= 2 * np.pi
   return x
```

And use it to modify the residual function:

```
def residual(a, b):
   """ compute residual and
   normalize to get the correct angular difference"""
   y = a - b
   y[2] = normalize_angle(y[2])
   return y
```

Another issue is that we need to compute the average of the state x and measurement z, each with an angle. The average of $359°$ and $1°$ will be $180°$ while the actual value is 0. So we need to use:

$$\bar{\theta} = a\tan 2(\sum_{i=0}^{2n} W_i \sin\theta_i, \sum_{i=0}^{2n} W_i \cos\theta_i)$$

```
def state_mean(sigmas, Wm):
   """ takes the sigma points and their corresponding weights
   and return the state mean of these sigma points"""
   x = np.zeros(3)
   sum_sin = np.sum(np.dot(np.sin(sigmas[:, 2]), Wm))
   sum_cos = np.sum(np.dot(np.cos(sigmas[:, 2]), Wm))
   x[0] = np.sum(np.dot(sigmas[:, 0], Wm))
   x[1] = np.sum(np.dot(sigmas[:, 1], Wm))
   x[2] = math.atan2(sum_sin, sum_cos)
```

```
    return x

def z_mean(sigmas, Wm):
    """ takes the sigma points and their corresponding weights
    and return the measurement mean of these sigma points"""
    z = np.zeros(3)
    sum_sin = np.sum(np.dot(np.sin(sigmas[:, 2]), Wm))
    sum_cos = np.sum(np.dot(np.cos(sigmas[:, 2]), Wm))
    z[0] = np.sum(np.dot(sigmas[:, 0], Wm))
    z[1] = np.sum(np.dot(sigmas[:, 1], Wm))
    z[2] = math.atan2(sum_sin, sum_cos)
    return z
```

So with these modifications, we can build the UKF:

```
    # build the UKF based on previous functions
    ukf = UKF(dim_x=3, dim_z=3, fx=move, hx=Hx,
        dt=dt, points=points, x_mean_fn=state_mean,
        z_mean_fn=z_mean, residual_x=residual,
        residual_z=residual)
```

5. Parameters

The main parameters for the UKF needed to be defined is x, P, Q and R. I will talk more later in Simulation about the real values that I used for these parameters. Here for the UKF, the initial x is the mean of your initial Gaussian distribution of the robot's initial state, P is the covariance matrix of the initial Gaussian distribution. They described how much you know about the initial state. And Q is the covariance matrix of noise model $N(0, Q)$ while R is the covariance matrix of noise model $N(0, R)$, which I already discussed in Motion Model and Measurement Model.

```
    ukf.x = np.array(x)
    ukf.P = P
    ukf.R = np.diag([sigma_range**2, sigma_range**2, sigma_bearing**2])
    ukf.Q = np.eye(3)*0.0001
```

Simulation

1. Build Simulated Models of Actuator and Sensor

Within the simulation, models of robot and sensor should do some modifications, especially including noise to generate realistic sensor outputs and control inputs. The range sensor output should include a normal distribution with zero mean and a standard deviation ($range\_std$), and the IMU absolute bearing output should include the same noise with $bearing\_std$. As for control inputs, resulting effective angular speed of the wheels is normal with a standard deviation vel_std, which is 5% of the max motor speed. The modified models is shown as follows:

```
# Build simulation models of sensor and actuator response
from numpy.random import randn

def noisy_reading(x, range_std, bearing_std):
```

```
    """ Return the sensor output with simulated noisy"""
    sensor_output = Hx(x)
    sensor_output[0] += randn() * range_std
    sensor_output[1] += randn() * range_std
    sensor_output[2] += randn() * bearing_std
    return np.array(sensor_output)


def noisy_move(x, dt, u, vel_std):
    """ Return next state of robot with
    simulated angular velocity noisy"""

    V_L = (u[0] + randn()*vel_std)*r
    V_R = (u[1] + randn()*vel_std)*r

    # linear velocity and angluar velocity of the robot
    v_k = (V_L + V_R) / 2.0
    omega_k = -(V_L - V_R) / w

    # linear displacement and angular displacement
    delta_s = v_k*dt
    delta_theta = omega_k*dt
    delta_x = delta_s * np.cos(x[2] + delta_theta/2)
    delta_y = delta_s * np.sin(x[2] + delta_theta/2)

    return x + np.array([delta_x, delta_y, delta_theta])
```

To find the exact value of $range\_std$, $bearing\_std$ and vel_std, I used the datasheets for reference. The max motor speed is $130rpm$ at $6V$. So we can get the standard deviation of the motor speed:

$$
\begin{aligned}
f &= \frac{130}{60} Hz \\
\omega_{\max} &= 2\pi f \\
\sigma_{vel} &= 5\% \omega_{\max}
\end{aligned}
$$

And the standard deviation of range sensors is 3% of the max range. The standard deviation of IMU is $0.1°$. I assigned them in code:

```
rpm_motor = 130
omega_max = 2*np.pi*rpm_motor/60.0 # max motor speed
sigma_vel = omega_max*0.05 # std = 5% of max motor speed
sigma_range = 1200*0.03    # std of range sensors
sigma_bearing = np.radians(0.1) # std of IMU absolute bearing
```

2. Simulation Implementation

Due to the design of the motion model which is derived based on the small displacement assumption. So I will design the UKF so that Δt is small. So if the robot is moving slowly our model will give a reasonably accurate prediction. I used $\Delta t = 1s$. The full implementation is as follows:

```
from filterpy.stats import plot_covariance_ellipse
import matplotlib.pyplot as plt
```

```
# Simulation with perfect knowledge of the initial state
dt = 1.0
def run_simulation(cmds, x, P, sigma_vel, sigma_range, sigma_bearing,
                   ellipse_step=1, step=2):
    """
    Args:
        cmds: a list containing all control inputs, angluar vels of left and
            right wheels
        x, P: mean and covariance of the knowledge about initial state
        sigma_vel, sigma_range, sigma_bearing: simulated standard deviation of
            input,
        range sensors and IMU.
        step: how many times the robot position is updated a second.

    Returns:
        the actual trajectory and state trajectory estimated by ukf
    """
    plt.figure(figsize=(12,8))
    ukf = ukf_build(x, P, sigma_range, sigma_bearing, dt=1.0)
    sim_pos = ukf.x.copy()

    # Plot boundaries
    plt.plot([0, 0], [0, W], 'k', linewidth = '10')
    plt.plot([0, L], [0, 0], 'k', linewidth = '10')
    plt.plot([L, L], [0, W], 'k', linewidth = '10')
    plt.plot([0, L], [W, W], 'k', linewidth = '10')

    np.random.seed(1)
    # actual trajectory
    traj = []
    # states given by ukf
    xs = []

    for i, u in enumerate(cmds):
        # move the robot with noisy input
        sim_pos = noisy_move(sim_pos, dt/step, u, sigma_vel)
        traj.append(sim_pos)

        if i % step == 0:
            # do the ukf predict step
            ukf.predict(u=u)
            # plot the covariance ellipse of the prior
            if i % ellipse_step == 0:
                plot_covariance_ellipse((ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2],
                    std=6,
                                        facecolor='k', alpha=0.3)
            # do the ukf update step with the noisy sensor output
            z = noisy_reading(sim_pos, sigma_range, sigma_bearing)
            ukf.update(z)
            # plot the covariance ellipse of the posterior
            if i % ellipse_step == 0:
                plot_covariance_ellipse((ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2],
                    std=6,
```

```
                                facecolor='g', alpha=0.8)
        xs.append(ukf.x.copy())

    # plot the actual trajectory and state trajectory estimated by ukf
    xs = np.array(xs)
    traj = np.array(traj)
    plt.plot(traj[:, 0], traj[:, 1], color='k', linewidth=2)
    plt.plot(xs[:, 0], xs[:, 1], 'r--', linewidth=2)
    plt.title("UKF Robot Localization")
    plt.grid()
    plt.axis('equal')
    plt.show()
    return xs, traj
```

In this implementation, the general loop is that move the robot according to the commands ($cmds$), then do the UKF predict step to get the prior, and do the measurement after moving, then do the UKF update step to get the posterior. The loop can be shown as follows:

$$\begin{array}{ccc} noisy\_move & \rightarrow & ukf.predict(u) \\ \uparrow & & \downarrow \\ ukf.update(z) & \xleftarrow{z} & noisy\_reading \end{array}$$

Other codes are mainly used to plot the simulation. I used *plot_covariance_ellipse* to plot the covariance ellipse at each state so that we can see the iteration of the prior and the posterior. And the parameter "step" is used to control how many times I moved the simulated robot per Δt.

3. Experiment with Constant Inputs

At first simulation, I just make angular velocities of the left and right wheels constant all the time as follows:

```
cmds = [np.array([2.0, 2.1])] * 30 # constant inputs
```

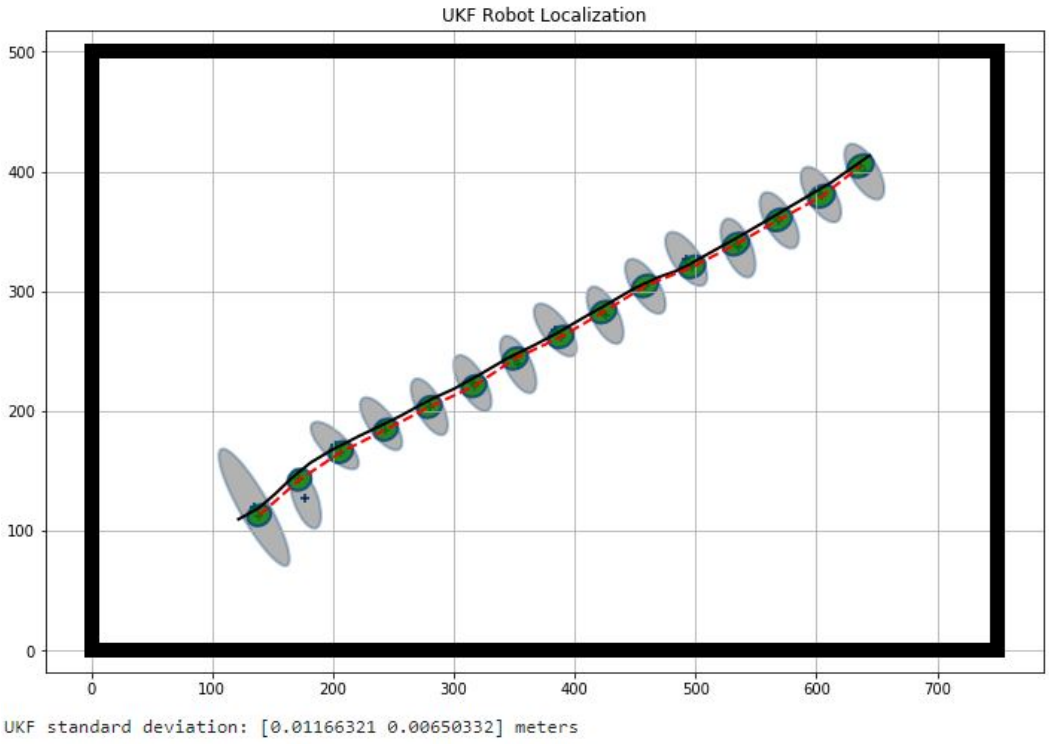
And I supposed the initial position of the robot is $x = 100mm, y = 100mm$ and the heading angle is $0.5rad$. And I have perfect knowledge of this initial state so that the covariance P could be comparatively small.

```
x = np.array([100, 100, 0.5]) # initial state
P = np.diag([2, 2, 0.05]) # initial covariance
```

Then I used these parameters to run the simulation with $step = 2$, which means I updated the simulated robot position 2 times a second but run the UKF only once:

```
xs, traj= run_simulation(cmds, x, P, sigma_vel, sigma_range, sigma_bearing,
    ellipse_step = 1, step=2)
print('UKF standard deviation:', np.std(xs[:,0:2]-traj[:,0:2],axis=0)/1000,
    'meters')
```

The simulation result is shown as follows. The gray ellipse is the covariance of the prior after each movement while the green ellipse is the covariance of the posterior after considering the measurement. I plotted 6σ for visualization. And the black line is the actual trajectory while the read dashed line shows the state trajectory estimated by the UKF.



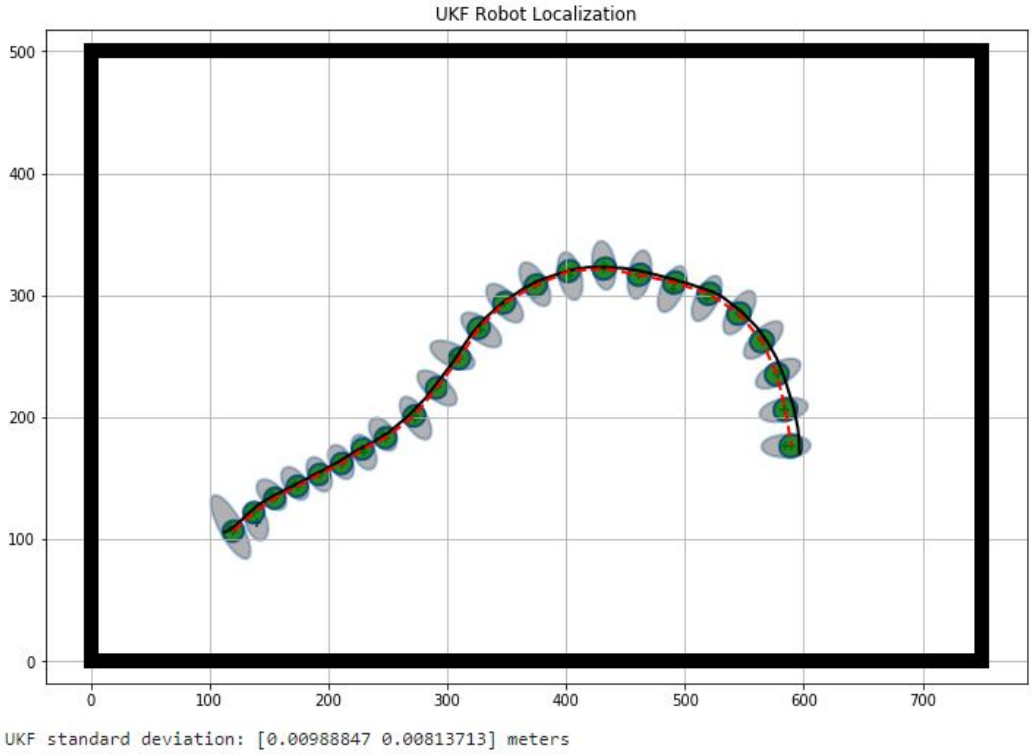
From the result, we can see that the initial covariance of the position is big and then the ellipse will become small (green) considering the measurement. Then after each movement, the green ellipse will become the next gray ellipse which indicates the uncertainty becomes bigger. After the measurement, the gray ellipse will shrink into the green ellipse which indicates the uncertainty becomes smaller with measurement. And the standard deviation of error between the actual trajectory and estimated trajectory is about $[0.0117m, 0.0064m]$.

4. Steering the Robot

The simulation above only considers constant inputs. Now let's consider varying angular velocities of wheels. I built the commands as follows:

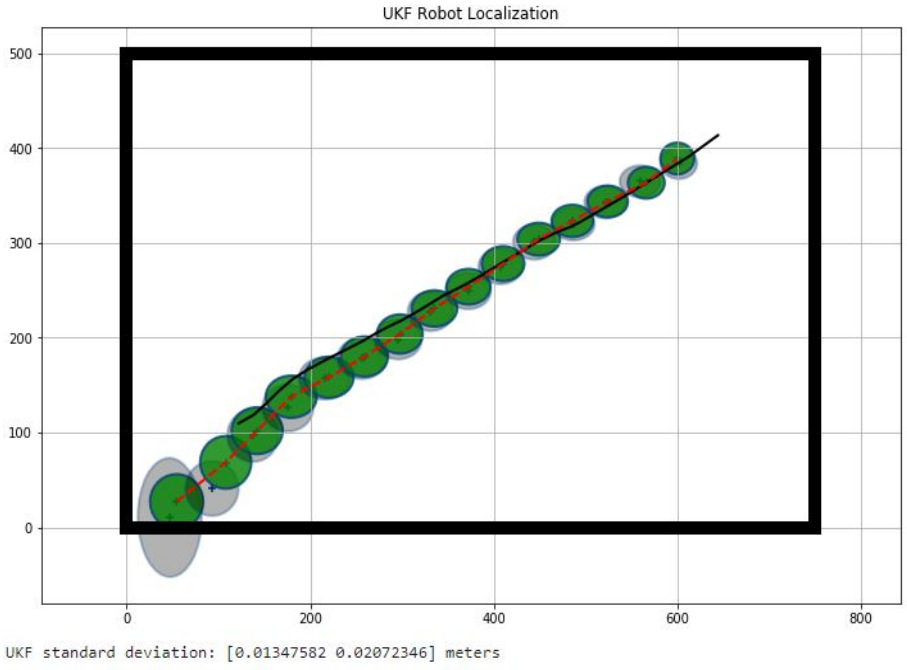
```
# Generate a sequence of inputs (not constant input)
cmds = [np.array([1.0, 1.1])] * 15
cmds.extend([np.array([1.0, 2.0])])
cmds.extend([cmds[-1]]*5)
cmds.extend([np.array([2.0, 1.0])])
cmds.extend([cmds[-1]]*25)
```

Still with good confidence of the initial state, we get the following result. The UKF still gave a good estimation. And the standard deviation of error between the actual trajectory and estimated trajectory is about $[0.0099m, 0.0081m]$. The accuracy doesn't change a lot.



5. No Knowledge of the Initial State

Now let's consider the case that we don't know where the initial state of robot is. The actual initial state is still $[100mm, 100m, 0.5rad]$. Supposing we don't know it, I just used a random guess $x = [10, 10, 0]$. Now with no confidence of this guess, we need to assign the covariance matrix P big enough so that the initial distribution can cover the actual state.



```
x = np.array([10, 10, 0])    # initial state (far away from the real one)
P = np.diag([100, 100, 0.2]) # large covariance
cmds = [np.array([2.0, 2.1])] * 30
xs, traj= run_simulation(cmds, x, P, sigma_vel, sigma_range, sigma_bearing,
    ellipse_step = 1, step=2)
print('UKF standard deviation:', np.std(xs[:,0:2]-traj[:,0:2],axis=0)/1000,
    'meters')
```

So you can see that the estimation of early stage is not accurate but the UKF will drag the estimation back to the actual trajectory gradually, which indicates the UKF can still work with no knowledge of the initial state. Also the accuracy decreased a little bit due to bad estimation during early stage.

Conclusions

1. My UKF state estimator works well both with perfect knowledge of the initial state and with no knowledge of it. These two cases always happen during robot localization. So this state estimator is a good candidate. The general standard deviation of the error is about $0.01m$ ($10mm$) which can be seen as a good estimator. In realistic, we can expand the outline of the robot with the estimation error so that it can avoid obstacles more confidentially in robot applications.

2. Another case is that the robot maybe move very fast. In this case, this UKF state estimator will fail quickly because if Δt is large, the motion model will become very inconsistent with the real system. If so, we need to implement it using a more sophisticated numerical integration technique such as Runge Kutta.

3. The choice of sigma points is very important for UKFs. The Van der Merwe's Scaled Sigma Point Algorithm is not the only choice and maybe not the best choice. It is case by case. Each technique uses a different way of choosing and weighting the sigma points so that they have different advantages and drawbacks.