MECH_ENG 469: ML & AI In Robotics

Report 0: Filtering Algorithms
UKF Implementation; Dataset 1

Michael Jenz

10-6-2025

Table of Contents

1 Introduction

This report describes how I have implemented an Unscented Kalman Filter Algorithm, a probabilistic state estimation filter, and tested it on a provided robot dataset (data set #1). The goal of implementing this filter was to accurately track the position of the robot using control data, as well as measurement data recorded by the robot. The robot in this data set was a differential drive robot that moved around a space using coded beacons to navigate. The robot configuration or state is represented as a three dimensional vector $[x, y, \theta]$ describing the robots x and y position in the world frame and the robots heading about the z-axis. The control commands are defined as a two dimensional vector $[v, w]$. This configuration can be seen below in Figure 1.

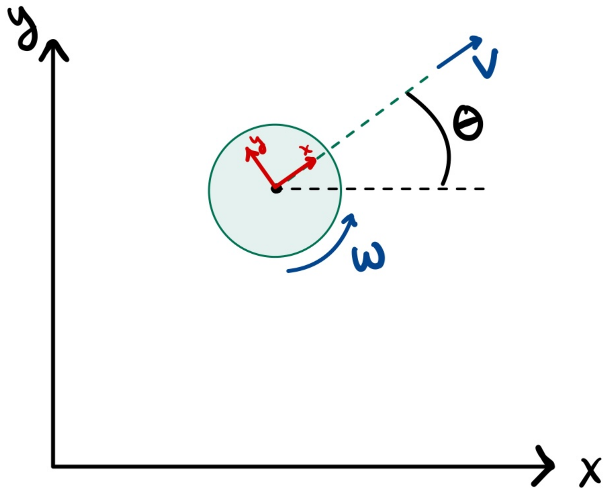


Figure 1: State and commands as defined for a differential drive robot represented by a green circle.

As a brief overview, the UKF algorithm has two main components: the motion, and the measurement model. These functions are used to transform sampled points from the state distribution and are a critical element of the design process. Thus, I will relay how I have designed the motion model, measurement model, as well as the overall UKF algorithm. Finally, I will present my overall results and findings in the last subsection. One important note is that I have switched the order of Question 4 to the UKF section, as it made more sense to discuss the particulars of this filter later in the report.

2 Part A

2.1 The Motion Model

The motion model estimates the future state of the robot. My measurement model uses a simple Euler's estimation algorithm. Using the velocity provided by the control data file and the change in time since the last command, I can estimate how much the robot's position had changed.

2.1.1 Design (Q1)

The inputs to my motion model are robot commands given in the robot coordinate frame, it is a vector containing the linear and angular velocity $[v, \omega]$. The motion model outputs the next state of the robot as defined by the vector $[x, y, \theta$ which describes its configuration in the world frame. To derive the next state of the robot in the world frame, I need to transform the linear velocity command into the world frame. I used simple trigonometry to achieve this. As shown below in Figure 2, the motion model uses the robot state heading to transform the command velocity into x and y components.

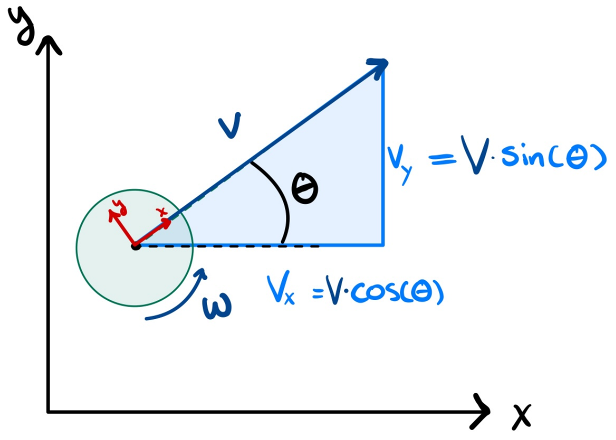


Figure 2: Transformation trigonometry to transform linear velocity command into world frame.

Next, since I have the velocity components, I can simply multiply the velocity components by the change in time and add this to the previous position to get the estimated next robot (x, y) position. However, I must also estimate the heading of the robot. This is done in similar fashion by multiplying the angular velocity by the change in time and adding this to the previous state. This estimation does not require a transformation since the z-axes of the world and robot frames are always aligned. These calculations can be seen below in Equations 1- 3. The motion model described has non-linear inputs because the angular velocity component ω is rotational.

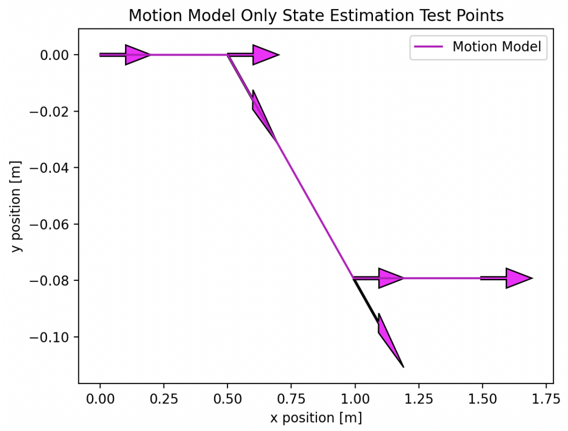
$$x_{next} = x_{prev} + v_x \cdot \Delta t \quad (1)$$

$$y_{next} = y_{prev} + v_y \cdot \Delta t \quad (2)$$

$$\theta_{next} = \theta_{prev} + \omega \cdot \Delta t \quad (3)$$

2.1.2 Implementation (Q2)

My implementation of the motion model can be seen below in Figure 3. I used a separate function for these simulation tests, since this is separate from the function of implemented filter. The plot below shows clearly the effect of the commands as the robot moves straight, turns, and repeats this process. The arrows indicate the heading of the robot at that point in time.



$$[v=0.5\frac{m}{s},\ \omega=0\frac{rad}{s},\ t=1s]$$
$$[v=0\frac{m}{s},\ \omega=\frac{-1}{2\pi}\frac{rad}{s},\ t=1s]$$
$$[v=0.5\frac{m}{s},\ \omega=0\frac{rad}{s},\ t=1s]$$
$$[v=0\frac{m}{s},\ \omega=\frac{1}{2\pi}\frac{rad}{s},\ t=1s]$$
$$[v=0.5\frac{m}{s},\ \omega=0\frac{rad}{s},\ t=1s]$$

Figure 3: Visualization (left) of example points (right) after they are passed through my implementation of the motion model.

2.1.3 Test (Q3)

Next, I tested my motion model on the complete command sequence from the dataset, as seen below in Figure 4. To understand how well the model performed, I plotted it alongside the ground truth position of the robot during operation. It is clear the motion model alone is not capable of accurately estimating the state of the robot during operation, as the estimated path diverges quickly from the actual and does not resemble the ground truth or follow any parallel path during the simulation. In summary, this figure shows the insufficiency of an approach using only input control data.

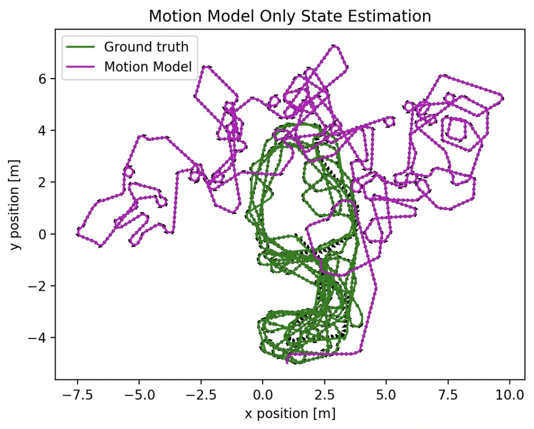


Figure 4: Comparison between the motion model state only state estimation and the ground truth robot location over time.

2.2 The Measurement Model

The measurement model returns the distance and heading (angle) to a specific beacon. This is useful because the robot can use this estimate as a way of checking if it is in the state that it thinks it is in. For instance, if the state estimation algorithm says that the robot is in a state $[x_1, y_1, \theta_1]$, it would expect the measurement to be $[d_1, \phi_1]$. If the measurement is different from this expectation, it shows that the robot is not where it predicted, and it can adjust its estimation from this measurement data. Therefore, we need a way to make this measurement guess.

2.2.1 Design (Q5)

The measurement model takes in a state of the robot $[x, y, \theta]$ and a subject ID which is a number that corresponds to a beacon. These beacons are towers located at known points that the robots can identify. With the subject ID, the measurement model can access the known position of the beacon to use in its calculations. The measurement model returns the estimated distance d and heading ϕ to the beacon in the robot reference frame as a vector $[d, \phi]$. This means that the heading is relative to the state of the robot, not to the world frame.

The operation of the measurement model is simple. To find the distance, it uses the simple distance formula, which is based on the geometric properties of a triangle. Since the position of the robot and beacon is known, we can draw a right triangle between the two points. The length of the perpendicular sides is equal to the difference in x and y coordinates. Then, using the Pythagorean theorem, we can find the length of the hypotenuse which is the distance between the two points. These calculations are in Equations 4- 6. The heading is then simply the angle of the right triangle closest to the robot, minus the robots current bearing. This angle can be calculated using the function $arctan$ as seen in Equations 7. These geometric relationships and variable definitions are shown in Figure 5.

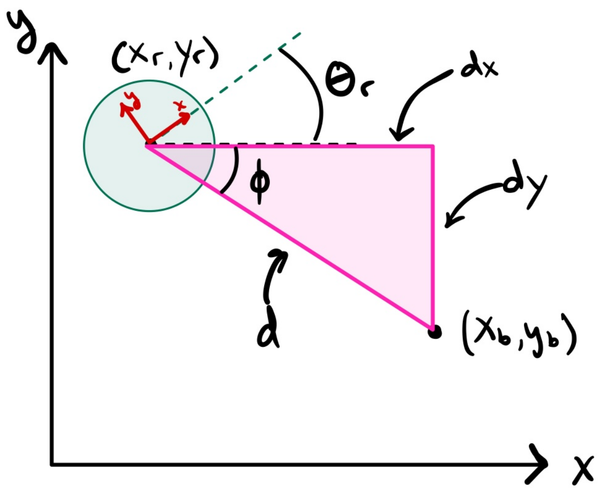


Figure 5: Geometric definitions used in the design of the measurement model that finds the distance and heading from the robot to a beacon.

$$d_x = x_b - x_r \tag{4}$$

$$d_y = y_b - y_r \tag{5}$$

$$d = \sqrt{d_x^2 + d_y^2} \tag{6}$$

$$\phi = arctan(\frac{d_y}{d_x}) - \theta_r \tag{7}$$

An important note is that the function $arctan$ is not continuous. Traditionally, it returns values between $[\frac{-\pi}{2}, \frac{-\pi}{2}]$. In my python implementation, I used the special $atan2$ function which extends the space around the complete unit circle to $[-\pi, \pi$ which ensures that I do not need to account for angles in quadrants II and III. However, I need to ensure that my state angles stay within this range of $atan2$, therefore I implemented angle wrapping as seen below in Equation 8 in both my measurement and the motion model to be safe. The $\%$ operator is known as the modulus operator and returns the remainder of the division between the dividend and the divisor.

$$\phi = (\phi + \pi)\%(2\pi) - \pi \tag{8}$$

2.2.2 Test (Q6)

The results of the measurement model test are shown in Table 1 and visualized in Figure 6.

Position (x, y, θ)	Landmark $(Subject\#)$	Distance (d)	Heading (phi)
(2,3,0)	6	8.57	-1.58
(0,3,0)	13	4.13	-0.729
(1,-2,0)	17	5.22	1.97

Table 1: Measurement model test results using given points and landmarks.

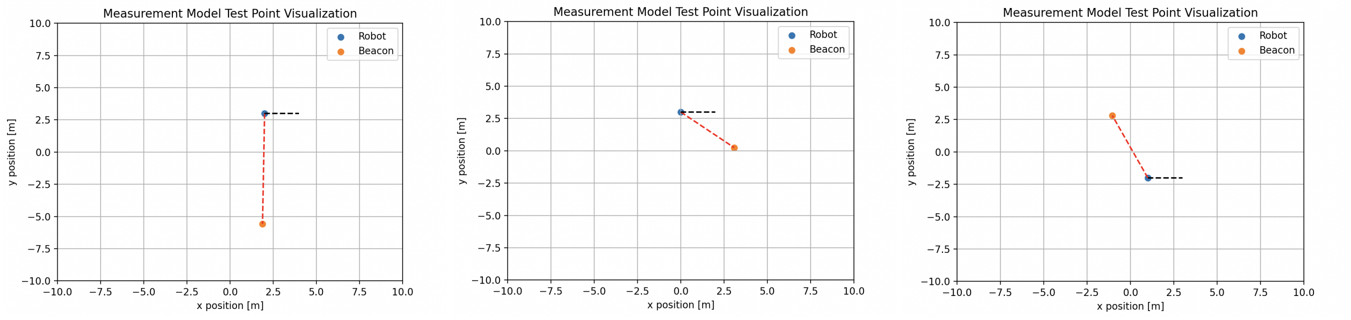


Figure 6: Graphical representation of measurement model calculating distance and heading to beacons from robot position and bearing.

3 Part B

3.1 The Unscented Kalman Filter

3.1.1 Description (Q4)

The Unscented Kalman Filter (UKF) is a probabilistic method of static estimation that allows for the use of non-linear functions through sampling. The UKF represents the state of the robot as a random variable within a multi-dimensional Gaussian distribution with mean μ and covariance Σ. The filter preserves the Gaussian distribution as it is passed through non-linear functions by sampling points known as *sigma points*, and recovers the distribution after passing these points through non-linear functions. The Gaussian is preserved because of the way that the sigma points are sampled. The two non-linear functions that the UKF uses are the process $f(x)$ and observation models $h(x)$, which in this case are the motion and measurement models that I have already developed. In this explanation, the state distribution is represented as x and the measurement distribution as z.
The first step in the UKF algorithm is to sample the sigma points of the distribution of x_{t-1}, representing the random variable of the previous state. We sample the sigma points according to Equations 9- 11 below, so there are $1 + 2n$ total sigma points where n is the dimension of the state space. In essence, to sample the distribution, we select the mean (x^a_{t-1}) and then points equally spaced around the mean. We define a scaling parameter λ in Equation 12 with parameters α and κ which determines how far apart these sigma points are. Typically, $\alpha = 10^-3$ and $\kappa = 0$ [1], [2].

$$X^0_{t-1} = x^a_{t-1} \tag{9}$$

$$X^i_{t-1} = x^a_{t-1} + (\sqrt{(n+\lambda)\Sigma})_i \tag{10}$$

$$X^{i+n}_{t-1} = x^a_{t-1} - (\sqrt{(n+\lambda)\Sigma})_i \tag{11}$$

$$\lambda = \alpha^2(n+\kappa) - n \tag{12}$$

In addition, we need to calculate the weights w^m, w^c to preserve the distribution for both the sigma points and the covariance. The weights can be calculated as in Equations 13- 16 where β is another parameter typically equal to 2 [3].

$$w^m_0 = \frac{\lambda}{n+\lambda} \tag{13}$$

$$w^m_i = \frac{1}{2(n+\lambda)} \tag{14}$$

$$w^c_0 = w^m_0 + (1 - \alpha^2 + \beta) \tag{15}$$

$$w^c_i = \frac{1}{2(n+\lambda)} \tag{16}$$

Now, we pass these sigma points through the nonlinear process model (motion model) $f(x)$ as in Equation 17. We then recover the Gaussian distribution using the following Equations 18- 19 where Q is noise. This transformed distribution is known as the forecast and is analogous to our use of the motion model earlier to track the state of the robot.

$$x_t^{f,j} = f(x_{t-1}^j) \tag{17}$$

$$x_t^f = \sum_{j=0}^{2n} w^{m,j} x_t^{f,j} \tag{18}$$

$$\Sigma_t^f = \sum_{j=0}^{2n} (x_t^{f,j} - x_t^f)(x_t^{f,j} - x_t^f)^T + Q_{t-1} \tag{19}$$

Next, we complete a very similar process, passing the same sigma points through the non-linear observation model as below in Equation 20.

$$z_t^{f,j} - 1 = h(x_{t-1}^j) \tag{20}$$

Then, we would repeat the distribution recovery step as seen previously, but using the sigma points as transformed above. As such, this step is not shown, the only difference is that the noise is adjusted and denoted as R_t.This gives us the innovation mean and covariance, essentially a distribution of expected measurements.

Finally, we complete the UKF by calculating the cross covariance between x_t^f and z_{t-1}^f as in Equation 21. Then, using this cross covariance we can find the Kalman gain, which is the last piece remaining before we find the posterior mean and covariance as in Equations 22- 24.

$$\Sigma_{cross} = \sum_{j=0}^{2n} (x_t^{f,j} - x_t^f)(z_{t-1}^{f,j} - z_{t-1}^f)^T \tag{21}$$

$$K_t = \Sigma_{cross} \Sigma_{innovation} \tag{22}$$

$$x_t^a = x_t^f + K_t(z_t - z_{t-1}^f \tag{23}$$

$$\Sigma_t = \Sigma_t^f - K_t \Sigma_{innovation} K_t^T \tag{24}$$

3.1.2 Implementation (Q7)

My full implementation of the UKF algorithm in Python is available in the run.py file located in the same compressed file as this PDF. This file will produce all of the requried plots from parts 1-9.

Implementing the above algorithm faced practical issues during the coding. Mainly, there was the question of command and measurement data coming in a different speeds and times. I solved this by going through the control data and using the measurement data to make corrections when it was available. Therefore, on iterations where there is no measurement, the fore-casted distributed was used as the posterior. Furthermore, at times the robot would record multiple measurements simultaneously. In this case, I would use all measurements of beacons and perform multiple corrections until all measurements at that timestep were used.

3.2 Results

3.2.1 Performance Comparison (Q8)

The best way to benchmark the improvement of the UKF algorithm is to compare it to the performance of the motion model only results. First, if we examine how the UKF performs using the list of given commands from Q2, we can see that there is not a substantial difference in performance from the motion model alone as seen in Figure 7. There

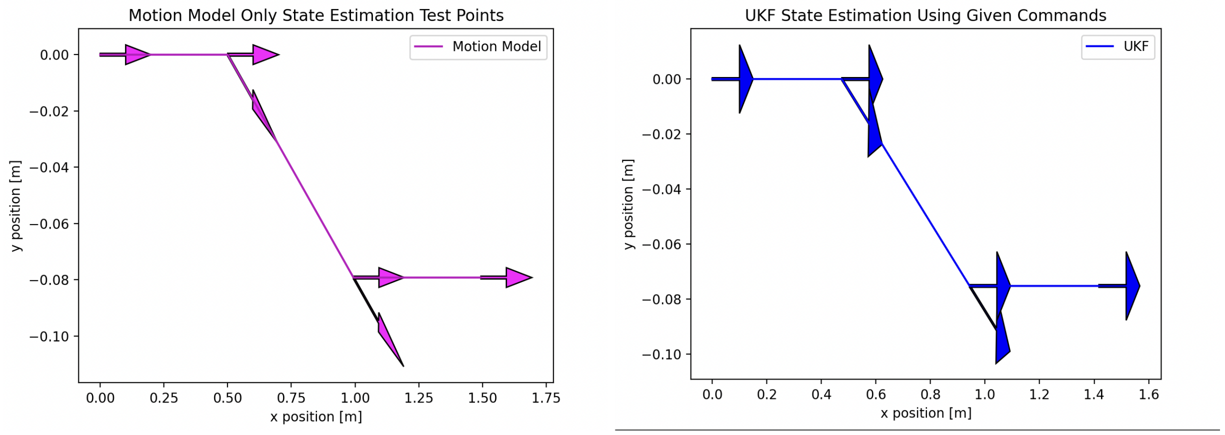


Figure 7: Comparison between the performance of the motion model only state estimation and the UKF state estimation on the given points from Q2.

is little discernible difference between these two plots because the UKF algorithm is doing the same thing as the motion model alone. Without measurement data to make corrections to the distribution, the UKF algorithm is just passing points through the same motion model as in step 2.

The difference in performance becomes discernible when measurements are used. In Figure 8 we see how the UKF algorithm performs compared to the motion model alone and the ground truth.

As you can see, the motion model only quickly diverges from the ground truth while the UKF implementation closely tracks the robots position over time. To reinfore this point, that the measurement correction and thus use of the Kalman gain and posteriors is critical, see Figure 10 (left) below which shows a similar plot to the one above, but the UKF algorithm is deprived of measurement corrections. It performs even worse than the motion model alone. This is likely because the noise introduced by the UKF algorithm in the distribution recovery steps now only serves to make the algorithm worse at following the robots position. To put an even finer tip on this point, see Figure 10 (right) which shows the UKF performance without measurements against our benchmark comparisons with noise removed from the algorithm. We can see that without noise it indeed performs extremely similarly to the motion

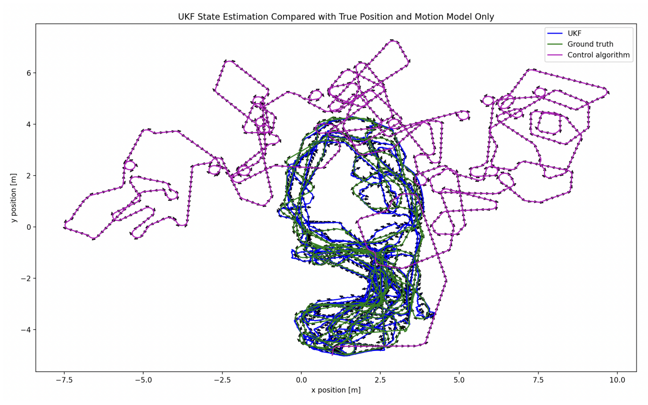


Figure 8: Comparison between the UKF implementation, ground truth, and motion model only implementation on the complete dataset.

model alone, but not accurately as it did with measurements.

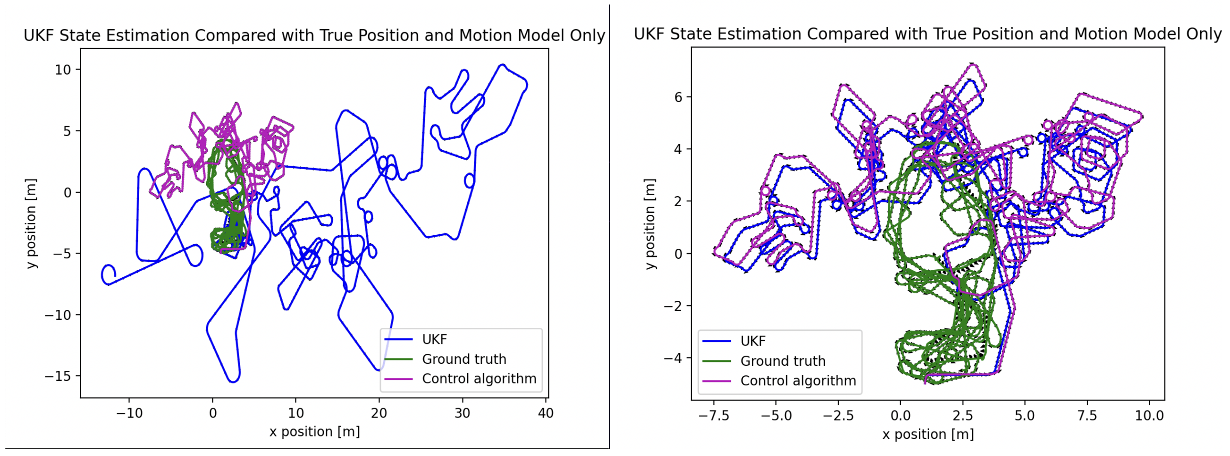


Figure 9: UKF performance without measurements (left) and without measurements and noise (right) against benchmark.

In summary, this section shows how the UKF's use of measurement to correct the motion model is critical in its performance. Next, I will analyze the use of noise in this algorithm and how that affects performance.

3.2.2 Uncertainty Analysis (Q9)

Noise plays a critical role in the UKF algorithm. While I was implementing my algorithm, I noticed that the magnitude of the noise had a huge impact on the performance of the algorithm. Too much noise and the UKF would get confused and lose the robot. Too little noise and the UKF would drift off if it did not have a measurement correction because it was not accounting for all possible states that the robot might be in.

Figure **??** below shows how increasing a decreasing the gains Q and R can affect the algorithm.

It is clear from the figure above that having noise that is too high will cause the algorithm to be extremely jittery. This creates high error scenarios and in effect reduces the accuracy of the kalman gain when you are involving

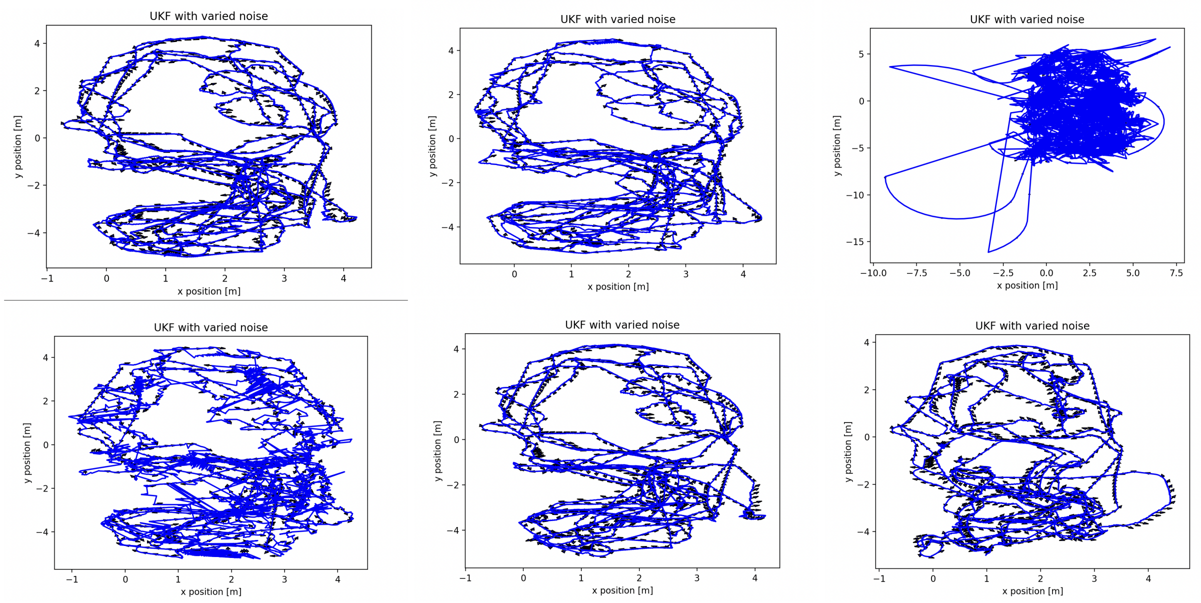


Figure 10: UKF algorithm with noise varied from small (left) to large (right). Q noise is varied in the top now and R noise is varied in the bottom row.

sigma points that are not possible states of the robot. Furthermore, we see that when noise is too small the UKF also becomes jittery. I believe that this effect is due to the absence of measurements. We can see this especially if you compare the results of the two most left plots. The low noise on the motion model transform does not have a huge negative effect on performance, but when there is low noise in the measurement model, the algorithm struggles to effectively sample the space and get useful sigma points to help explore the space around it and make use of the measurement correction.

In summary, these tests show the sometimes unpredictable and hard to discern effects of uncertainty in the algorithm.

References

[1] G. A. Terejanu, “Unscented kalman filter tutorial,” Department of Computer Science and Engineering, University at Buffalo, Buffalo, NY 14260, Technical Report, YearHere, Accessed: 2025-10-07.

[2] *Unscented kalman filter — ahrs 0.4.0 documentation*, Readthedocs.io, 2019. [Online]. Available: https://ahrs.readthedocs.io/en/latest/filters/ukf.html (visited on 10/07/2025).

[3] J. Han, *The unscented kalman filter (ukf): A full tutorial. ps. sampling methods are amazing*, YouTube, Aug. 2023. [Online]. Available: https://www.youtube.com/watch?v=c_6WDC66aVk (visited on 10/07/2025).