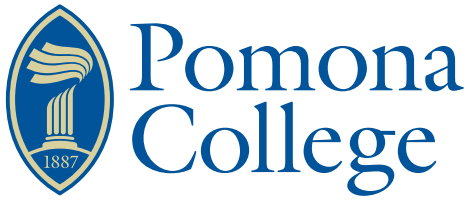


Senior Thesis in Mathematics

An Exploration of Kalman Filters

Author:
Lindsey Tam

Advisor:
Dr. Blerta Shtylla

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

March 8, 2020

Abstract

Kalman Filters are able to recursively generate predictions for linear systems; these predictions become progressively more accurate because of the system's ability to correct predictions using incoming data. Nonlinear forms of the Kalman Filter exist, including the Extended Kalman Filter and the Unscented Kalman Filter. This paper explores the theory behind each of these filters and implements a few examples of the Unscented Kalman Filter.

Contents

1 Introduction **ii**

2 Kalman Filters **iv**
2.1 Kalman Filter Algorithm vi

3 Extended Kalman Filters **xii**
3.1 Extended Kalman Filter Algorithm xii

4 Unscented Kalman Filters **xvii**
4.1 Unscented Kalman Filter Algorithm xix
4.2 Van der Pol Example xxv
4.3 Modeling a Biological System xxix

A Van der Pol Code **xxxiv**
A.1 Main function xxxiv
A.2 State function xxxvi
A.3 Measurement function xxxvii

B Metabolites Example Code **xlii**
B.1 Main function xlii
B.2 State function xlvi
B.3 Measurement function xlvii
B.4 Parameter Values xlvii

Chapter 1

Introduction

This paper explores the Kalman Filter, Extended Kalman Filter, and the Unscented Kalman Filter. The Kalman Filter is a data prediction method that estimates the value of states in linear dynamical systems. This is a recursive predictive-corrective process that enables the Kalman Filters to continuously generate predictions about state variables without relying on large amounts of initial data. In the case of nonlinear systems, alternative forms of the Kalman Filter were developed, including the Extended Kalman Filter (EKF) and the Unscented Kalman Filter (UKF). The EKF linearizes the nonlinear system around the mean by taking the Jacobean of the nonlinear function. Navigation and signal processing are the main applications of the EKF Instead of linearizing the system around a single point, the UKF utilizes many points, known as Sigma points. The Sigma points then undergo a nonlinear transformation known as the Unscented Transform. Applications of the UKF includes modeling biological systems. Though we will discuss all three versions, the exploration of the UKF will be the main focus of this paper.

In addition to understanding the theory behind these algorithms, this paper also explores applications of the UKF. An example of the UKF can be applied to the Van der Pol oscillator, which is a self sustaining nonconservative oscillator. The purpose of this simple example is to demonstrate how a simple version of UKF can be implemented on MATLAB. We will also observe how process and measurement noise influences the system. A second example includes modeling the biological pathway of metabolites, which are molecules in the human body that are the byproduct of the metabolism. This example is slightly more complex, with four different states and 18 unknown parameters. The focus of this example is to illustrate the following three things: how the UKF can correct for multiple states, working with systems in higher dimensions, and how the UKF can be applied to extract parameter values.

The ultimate goal of researching the UKF is to create a model that can forecast glucose values in real time of patients with Type 1 diabetes. This is inspired

by a separate study that was able to do real-time glucose forecasting in Type-2 diabetes patients. Using an existing model of Type 1 diabetes that includes 12 states, the UKF will be most useful in extracting parameter values. To gauge the effectiveness of the model, we will be comparing the results with mouse data. Implementing this model has the potential to improve treatment methods and the quality of life of Type 1 diabetes patients.

Chapter 2

Kalman Filters

The Kalman Filter (KF) recursively generates predictions for linear dynamical systems [10]. The basic foundations of the this algorithm include generating a prediction given some initial knowledge of the data and using actual measurements from the system to continually correct the prediction. Therefore, unlike other predictive methods like machine learning, the KF can begin generating estimates without large amounts of initial data. The KF begins by assuming the given data is noisy and Gaussian [9, 11]. The first predictive step assumes knowledge of initial states and the model process. Since we know the system is linear, the model process can be denoted as a matrix and the states can be expressed as a vector. Through matrix vector multiplication, this algorithm simulates how states are transformed after undergoing some process. The next step involves calculating the covariance in order to calculate the Kalman Gain, which is a measure of how much the estimate should be changed given actual measurements of the system. The corrective step utilizes the Kalman Gain, which is a matrix representing a weight to correct the prediction. This process can be done recursively, allowing the model to become progressively more accurate as more data is added. Overtime, it is expected that the model will converge with the actual system measurements.

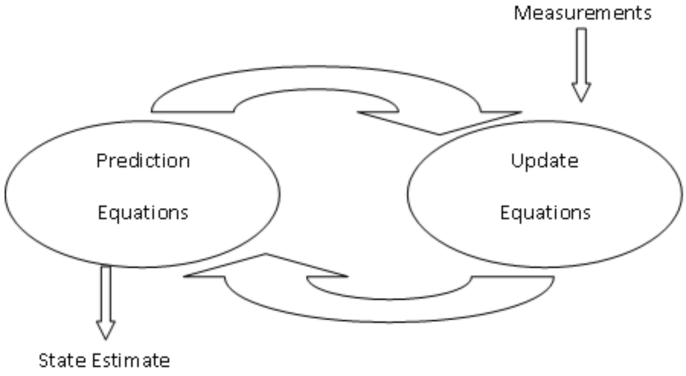


Figure 2.1: A basic diagram demonstrating the recursive nature of the Kalman Filter [6].

The most common application of KFs is in navigation, image processing, and finance. A relevant example is using computer vision to monitor and track vehicles in real time. This enables a traffic cam to know when to take a picture to capture a vehicle's license plate. Another example is the development of the Global Positioning Systems (GPS) [7]. The KF also has many aeronautical applications, which include long-distance flight and autopilot systems. In fact, the KF was created to development the navigation system in the Apollo Program [4].

The Kalman Filter is named after its developer, Rudolph Emil Kalman. Kalman was born on May 19, 1930 in Budapest, Hungary. After arriving in the United States, Kalman completed his undergraduate studies and masters degree in electrical engineering from Massachusetts Institute of Technology and completed his doctoral degree in Columbia University. He would spend the next years of his life teaching. In the 1960s to 1970s he became a professor at Stanford university [4]. In the 1970s and 1990s, Kalman spent time as professor of engineering at the University of Florida. Kalman is most known for his work on the Kalman Filter, which was developed in the late 1950s. The Kalman Filter greatly aided the United States' military projects, resulting in former President Obama to award Kalman the National Medal of Science in 2009. In addition, in 1985, Kalman was awarded the Kyoto Prize, which is the Japanese version of the Nobel Peace Prize. Kalman passed away July 2, 2016 at the age of 86 and is survived by his wife and two children [3].

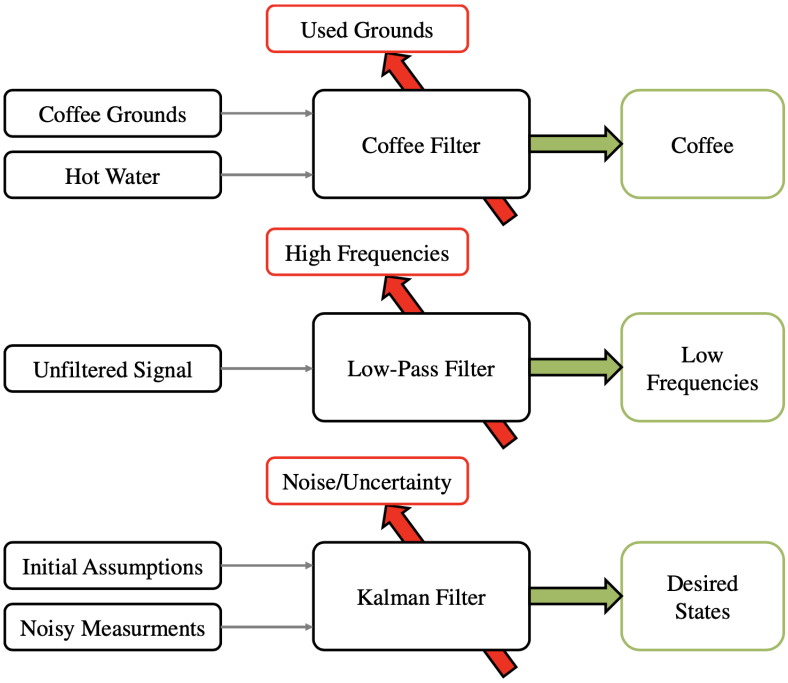


Figure 2.2: A simpler way to explore the KF is to facilitate a comparison with a coffee filter. This image is taken from a paper by Rhudy et al. comparing the Kalman Filter to a coffee filter [9].

2.1 Kalman Filter Algorithm

Before delving into the formal steps of the KF it is important to understand the underlying foundation of this algorithim. The goal of the Kalman Filter is to predict the new state of a system, after it undergoes some transformation. Let this be represented by

$$\frac{dx}{dt} = f(x) + \varepsilon',$$

where $\frac{dx}{dt}$ is the prediction of the states, f is a function that transforms the states x, and ε' is internal system noise. Since the system we are considering is linear, we can represent this transformation using matrix multiplication. Let matrix M' be used to represent the transformation f in order to approximate the equation above as

$$\frac{dx}{dt} \approx M'x + \varepsilon'.$$

The Kalman Filter works by generating predictions and making correction at various time steps. Let these time steps be denoted as k, such that k is a

nonnegative integer and let x_k be the estimate of the state variables at time k with initial assumptions of the state variables beginning at x_0 such that $x_0 = \mathcal{N}(m_0, c_0)$.

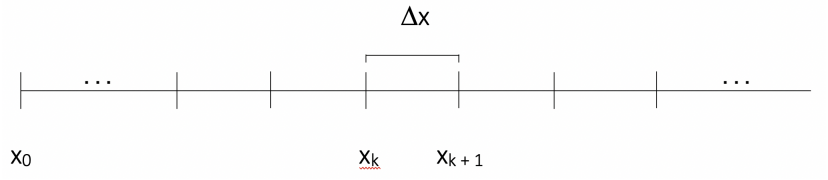


Figure 2.3: A diagram showing discretized time steps of the system. Though this diagram depicts time steps that are equal, this should not be assumed for all cases.

Next, we can discretize the system and approximate $\frac{dx}{dt}$ using the limit definition of derivative:

$$\begin{aligned}
\frac{x_{k+1} - x_k}{\Delta t} &\approx M' x_k + \varepsilon'_k \\
x_{k+1} - x_k &\approx M' x_k \Delta t + \varepsilon'_k \Delta t \\
x_{k+1} &\approx M' x_k \Delta t + \varepsilon'_k \Delta t + x_k \\
x_{k+1} &\approx (M' \Delta t + I) x_k + \varepsilon'_k \Delta t.
\end{aligned}$$

From here, substitute M for $M'\Delta t + I$ and ε_k for $\varepsilon'_k \Delta t$ to get the dynamics model:

$$x_{k+1} \approx M x_k + \varepsilon_k,$$

where $\varepsilon_k = \mathcal{N}(0, c_1) or Gaussian white noise$. This general equation is significant to the generation of predictions and will continue to be used in different versions of the Kalman Filter.

Another important aspect of the KF is the incorporation of state measurements. We cannot assume that we will always have measurements for all states. More likely, we will only have measurements for a subset of states, Therefore, it is necessary to transform the prediction into a format that can be compared with state values. The transfomed estimate of the the system measurement at the next time step, call it y_{k+1}, is given by

$$y_{k+1} \approx H x_{k+1} + v_{k+1},$$

where H is the matrix observation function and v_k is measurement noise vector at time step k such that $v_k = \mathcal{N}(0, c_2) or Gaussian white noise$. H enables the state variables to be linearly transformed to match the outputs of the system. The dimensions of H reflects which state variables have measurable values in the system. It is not assumed that every state variable is measurable, so H

allows us to compare the measurable state variables to x_k. Simple applications of H include creating matrices with 0's and 1's, with 1's denoting that a state variable is measurable and a 0's representing non-measurable states. In other cases, H is an integer used as a scaling factor.

Now that the dynamics model and data model are clarified, we can begin discussing steps of the KF in greater depth. As a general overview, the KF algorithm consists of three major components:

1. Initialize state variables
2. Generate a prediction
3. Update prediction with measurements from the system.

The recursive component of the filter consists of repeating steps 2 and 3 repeatedly, while step 1 only needs to be done once. Details about each step are explored further below.

1. Begin by initializing the state estimate and the initial state covariance matrix. The state estimate, x_0, is a column vector containing state variables, call them $x_a, x_b, \ldots, x_n$, such that $x_0 = [x_a, x_b, \ldots, x_n]^T$, where T is the transpose. The state estimate can be found by taking the expected value of the data, which is normally distributed. If the system states are finite, the expectation is denoted by

$$\mathbb{E}[x_0] = \sum_{i=a}^{n} x_i p_i = [x_a p_a + x_b p_b + \ldots + x_n p_n]^T,$$

where $p_a, p_b, \ldots, p_n$ is the respective probability of getting each state variable. The state covariance matrix, P_0, is a square matrix whose contents are the covariance of the pairwise elements [1]. A state covariance matrix is a symmetrical positive semi-definite square matrix whose diagonals correspond to the variance of a variable at location i and elsewhere is the covariance of the pairwise elements. In practice, covariance matrices help us better understand the spread of data. For the case of the KF, calculating the state covariance is necessary for computing Kalman Gain and can be done by the following

$$P_0 = \begin{bmatrix} var(x_a) & \ldots & cov(x_a, x_n) \\ \vdots & \ddots & \vdots \\ cov(x_n, x_a) & \ldots & var(x_n) \end{bmatrix}.$$

Enough should be known about the modeled system to generate these values. In the case where one knows the true value of the initial states,

[1] Recall that the covariance of a variable with itself is the variance of the variable

the state covariance would have all 0 values. On the other hand, if one is unsure of the initial values, values in P_0 are expected to be higher. It is important to initialize the model with values close to the true value, else the system will converge at a slow rate.

2. After initializing the state estimate and state covariance, a prediction can be generated. The estimate of the system at the next time step, x_{k+1}, is given by

$$x_{k+1} = Fx_k + w_k,$$

where F is the state transition matrix and w_k is the process noise vector. Every state variable contained in x_k is defined by a linear differential equation. These linear differential equations can be used to generate the F matrix. Therefore, F should be a square matrix whose dimension is equal to the number of states variables.

w_k is the process noise vector at time k. Process noise can be thought of as the model's accuracy. When process noise is 0, it implies that the model is perfectly accurate and does not have to correct for incoming system measurements. On the other hand, high process noise will essentially restart the system based on incoming measurements. w_k has the same dimensions as x_k, allowing us to identify whether or not to adjust the equations for the state variables.

3. Next, correct the prediction with incoming measurements from the system. Begin by calculating the state covariance matrix in order to calculate Kalman Gain. The state covariance matrix at time step k given the last time step, is

$$P_{k|k-1} = FP_{k-1}F^T + Q_{k-1},$$

where F^T is the transpose of F, and Q_k is the process noise covariance of w_k. The Kalman Gain at time step k, is given by

$$K_k = P_{k|k-1}H^T(HP_{k|k-1}H^T + R_k)^{-1},$$

where H is the observation matrix and R is measurement noise covariance matrix.

From this equation, one can see that balancing Q and R is critical for model performance. Larger values of Q indicate higher modeling error, which leads to a higher Kalman Gain and increased model correction. On the other hand, large values of R imply high measurement error, leading

to a lower Kalman Gain and less model correction.

The Kalman Gain is the main component of the corrective aspect of the KF. The Kalman Gain is a measure of how much to change the model based on incoming data. Low values of the Kalman Gain imply the model is accurate while higher values indicate the model should adjust based on the incoming data.

Next, calculate the transformed prediction in order to correct the prediction. The transformed state vector, $\hat{y}_k$, is given by

$$\hat{y}_k = Hx_k + v_k,$$

where v_k is measurement noise, which is added to account for measurement error. The corrected prediction, x_k, is given by

$$x_k = x_{k-1} + K_k(y_k - \hat{y}_k),$$

where y_k is the actual measurements of the system, K_k is the Kalman Gain matrix at time step k, and x_{k-1} are the values of the state variable at the last time step. The quantity $y_k - \hat{y}_k$ is also known as measurement residual or innovation. Later on, we will see how this value is used to gauge model performance.

The final step is to update the state covariance matrix, P_k, through the equation

$$P_k = (I - K_k H)P_{k|k-1},$$

where I is the identity matrix, K_k is the Kalman Gain at time step K, H is the observation matrix, and $P_{k|k-1}$ is the state covariance at time step k given the last time step. P_k will be used in the next iteration of the filter.

Table 2.1: Description of all variables in the Kalman Filter

Variables in the Kalman Filter		
Variable	Description	Dimensions
x	State variables	$x \times 1$
$\hat{y}$	Transformed state vector	$y \times 1$
y	Actual system measurement	$y \times 1$
v	Measurement noise	$y \times 1$
w	Process noise	$x \times 1$
F	State function	$x \times x$
H	Observation function	$y \times x$
K	Kalman Gain	$x \times y$
Q	Process noise covariance	$x \times x$
R	Measurement noise covariance	$y \times y$
P	Covariance matrix	$x \times x$

Chapter 3

Extended Kalman Filters

The Extended Kalman Filter (EKF) is the non-linear version of the Kalman Filter. For the most part, the EKF algorithm is nearly identical to the KF algorithm. The critical difference is in linearizing the non-linear state and observation function. The EKF uses the Jacobian to linearly approximate the non-linear function around the mean of the Gaussian distribution. Skipping this step would result in the transformed data being non-Gaussian; though taking the Jacobian enables the transformation to remain Gaussian, it is not exact, resulting in some generalization. Linear approximation through a single point also makes the EKF inefficient when dealing with complex, higher order systems. Because of this, the model is highly subject to error, which can be somewhat reduced by setting accurate initial values. Though these flaws exist, the EKF performs strongly with applications of real time spatial fields, including navigation and positioning systems.

For the purpose of my thesis, I will not be applying the EKF to any examples. The purpose of incorporating it here is to demonstrate one of the simpler non-linear versions of the KF. For more information, including MATLAB code, the following can be referenced [1, 9].

3.1 Extended Kalman Filter Algorithm

Similar the the KF, the EKF has two main models:

$$\begin{aligned} &\text{state model: } x_k = f(x_{k-1}) + w_{k-1} \\ &\text{data model: } \hat{y}_k = h(x_k) + v_k, \end{aligned}$$

where x_k is the state vector at time k, f is the vector-valued non-linear transformation function, w_k is the process noise at time k such that $w_k = \mathcal{N}(0, Q)$ or gaussian white noise with covariance Q, h is the vector-valued nonlinear observation function, and v_k is the measurement noise at time k such that $v_k = \mathcal{N}(0, R)$

or gaussian white noise with covariance R.

The main difference between these models and the models from the KF is that f and h are vector-valued non-linear functions. A prediction can be obtained by taking results from the last time step and transforming them using f, which is the non-linear set of ODEs used to describe the system. Similarly, the transformed prediction can be generated by taking the ODEs of states that are measurable, h, to transform the prediction and then add measurement noise.

Although using the non-linear function works for the state and data model, a non-linear function cannot be used in the correction step, such as calculating covariance of the state or the Kalman Gain. To overcome this, the EKF linearizes around the non-linear system by calculating the Jacobian matrix of f and h. Recall that the Jacobian matrix allows for the linear approximation of a non-linear system by taking first-order partial derivatives. Let F, the Jacobian of f, be given by

$$F = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix},$$

where f_i is the ODE corresponding to the ith state variable and x_i is the ith state variable. One can see that this results in F being a square matrix. In addition, let H be the Jacobian of h, given by

$$H = \frac{\partial h}{\partial x} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial x_1} & \cdots & \frac{\partial h_m}{\partial x_n} \end{bmatrix},$$

where h_i is the ODE corresponding to the ith measurable state variable and x_i is the ith state variable.

Using these linearized versions of f and h, the correction step can be applied, making the algorithim similar to the KF. Recall that the algorithm consists of three major components:

1. Initialize the state variables
2. Generate a prediction
3. Update prediction with measurements from the system.

The recursive component of the filter consists of repeating steps 2 and 3 repeatedly, while step 1 only needs to be done once. For the most part, the EKF is going to be very similar to the KF, with a few exceptions. Details are explained more in depth below.

1. Begin by initializing the state estimate, x_0, and the initial covariance matrix, P_0. Similar to the KF, the initial state estimate can be found by

$$\mathbb{E}[x_0] = \sum_{i=a}^{n} x_i p_i = [x_a p_a + x_b p_b + \ldots + x_n p_n]^T,$$

where $x_a, x_b, \ldots, x_n$ are the state variable, $p_a, p_b, \ldots, p_n$ is the probability of getting each state variable, and T is the transpose. We also want to initialize the covariance matrix by

$$P_0 = \begin{bmatrix} var(x_a) & \ldots & cov(x_a, x_n) \\ \vdots & \ddots & \vdots \\ cov(x_n, x_a) & \ldots & var(x_n) \end{bmatrix}.$$

Often, P_0 will be initialized as a diagonal matrix with the diagonals being the variance of each state variance and every other value being set to 0.

2. After initializing the model, a prediction can be generated. The prediction step deviates from the Kalman Filter because we can no longer represent the transformation through matrix multiplication because the system is no longer linear. Instead, the prediction is generated by directly applying the last state estimate to the non-linear transformation f and adding process noise w_k, which is given by

$$x_k = f(x_{k-1}) + w_{k-1}.$$

3. The correction step requires transforming the prediction and calculating the Kalman Gain. However, before we can do that we must linearize both f and h by calculating the Jacobian. Let F be the Jacobian of f and let H be the Jacobian of h, both of which are given by

$$F = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix},$$

$$H = \frac{\partial h}{\partial x} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_m}{\partial x_1} & \cdots & \frac{\partial h_m}{\partial x_n} \end{bmatrix},$$

where f_i is the ODE corresponding to the ith state variable, h_i is the ODE corresponding to the ith measurable state variable, and x_i is the ith state variable.

Using F and H, one can now calculate the state covariance and the Kalman Gain. Recall that the Kalman Gain is a method to determine whether to place more weight on incoming measurements or the system's model. $P_{k|k-1}$, can be found by

$$P_{k|k-1} = FP_{k-1}F^T + Q_{k-1},$$

where F is the Jacobian of the nonlinear state function, P_{k-1} is the state covariance at the last time step, T is the transpose, and Q is the process noise covariance. Using $P_{k|k-1}$, calculate the Kalman Gain by

$$K_k = P_{k|k-1}H_K^T(H_kP_{k|k-1}H_K^T + R_k)^{-1},$$

where H is the Jacbian of the nonlinear observation matrix, and R is the measurement noise covariance. Recall that H and F are linear approximations of f and h. Therefore, it is assumed that there is some amount of error when calculating the Kalman Gain.

The next component of the correction step is to transform the prediction, $\hat{y}_k$, to compare with incoming measurements from the system, y_k. The transformed prediction, $\hat{y}_k$, is given by

$$\hat{y}_k = h(x_k) + v_k,$$

where h is the vector-valued non-linear observation function and v_k is measurement noise. Using $\hat{y}_k$, we can calculate the residual, which is a measure of how close the estimate is the measured values. The equation can be updated by

$$x_k = x_{k-1} + K_k(y_{k-1} - \hat{y}_{k-1}).$$

Finally, update the covariance matrix, P_k, which will be used in the next iteration of the filter. P_k can be updated by

$$P_k = (I - K_kH)P_{k|k-1},$$

where I is the identity matrix, K_k is the Kalman Gain at time k, $P_{k|k-1}$ is the covariance matrix given the last time step, and H is the Jacobian of the nonlinear observation matrix.

Table 3.1: Description of all variables in the Extended Kalman Filter

Variables in the Extended Kalman Filter		
Variable	Description	Dimensions
x	State variables	$x \times 1$
$\hat{y}$	Transformed state vector	$y \times 1$
y	Actual system measurement	$y \times 1$
v	Measurement noise	$y \times 1$
w	Process noise	$x \times 1$
f	Non linear state function	$x \times 1$
F	Jacobian of f	$x \times x$
h	Non linear observation function	$y \times 1$
H	Jacobian of h	$y \times x$
K	Kalman Gain	$x \times y$
Q	Process noise covariance	$x \times x$
R	Measurement noise covariance	$y \times y$
P	Covariance matrix	$x \times x$

Chapter 4

Unscented Kalman Filters

The Unscented Kalman Filter (UKF) is another nonlinear version of the Kalman Filter and was developed to address the shortcomings of the EKF. As opposed to using the Jacobian to linearly approximate around a single point, the UKF uses the Unscented Transform (UT) to approximate around multiple points, known as sigma points. The UT is a method of approximating probability distributions that have undergone a non linear transformation using limited statistics. The UT uses these sigma points, which are represented in a Sigma Point Matrix, to represent the normal distribution of the data. Then, the sigma points undergo a non-linear transformation, resulting in a posterior distribution that is not normal [10, 12] . We are able to approximate the normal distribution of the posterior distribution using the weights and covariance that were calculated prior to the transformation. This process enables the Kalman Filter to be applied to more complex non linear problems.

Unlike the Kalman Filter and the Extended Kalman Filter, the UKF also has a set of parameters. Explanations of each parameter and their default values can be found in Table 4.2. For the UKF, parameters are necessary for controlling the spread of sigma points. This was not needed for the EKF, since the EKF was only linearizing around the mean. expand here on how to tune parameters - have not had a chance to look into this into depth yet

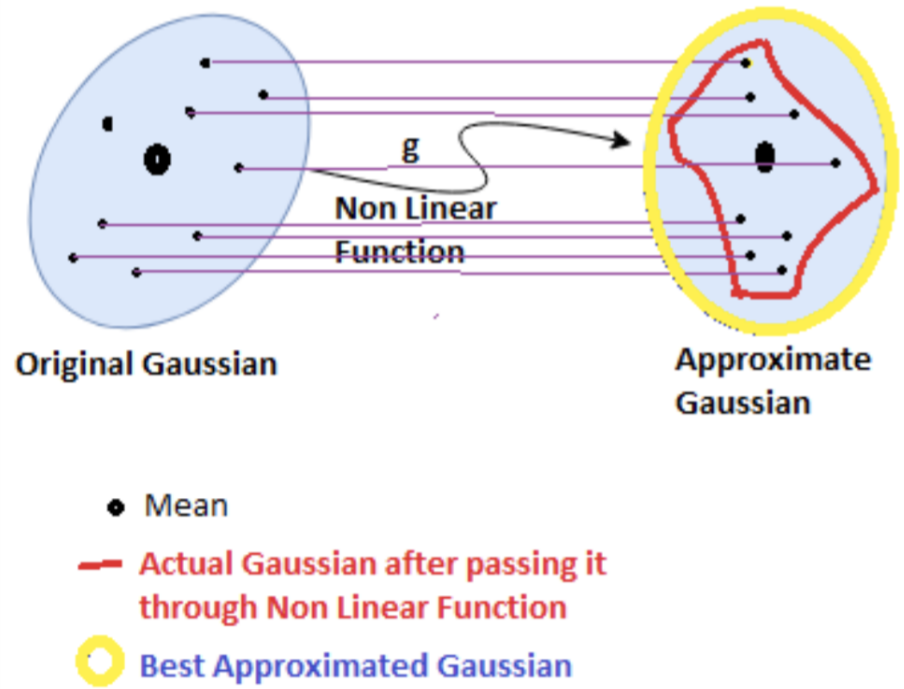


Figure 4.1: A depiction of the general overview of the UKF. Here we see that sigma points are sampled from a Gaussian distribution and are propagated through a non-linear function, called g. Though the output is non-Gaussian, an approximation of the Gaussian distribution can be obtained.

The term 'unscented' was arbitrarily coined by the developer of the UKF, Jeffrey Uhlmann. In an interview, he shares:

> "Initially I only referred to it as the "new filter." Needing a more specific name, people in my lab began referring to it as the "Uhlmann filter," which obviously isn't a name that I could use, so I had to come up with an official term. One evening everyone else in the lab was at the Royal Opera House, and as I was working I noticed someone's deodorant on a desk. The word "unscented" caught my eye as the perfect technical term. At first people in the lab thought it was absurd—which is okay because absurdity is my guiding principle—and that it wouldn't catch on. My claim was that people simply accept technical terms as technical terms: for example, does anyone think about why a tree is called a tree?"

4.1 Unscented Kalman Filter Algorithm

Before delving into the details of the algorithm, we will explore a high-level overview of the UKF process. As with the KF and the EKF, the UKF follows the three major components of:

1. Initializing the model's states
2. Generating a prediction
3. Updating prediction with measurements from the system.

In the first step, initializing the model requires calculation of sigma points. These sigma points characterize the normal distribution of the data.

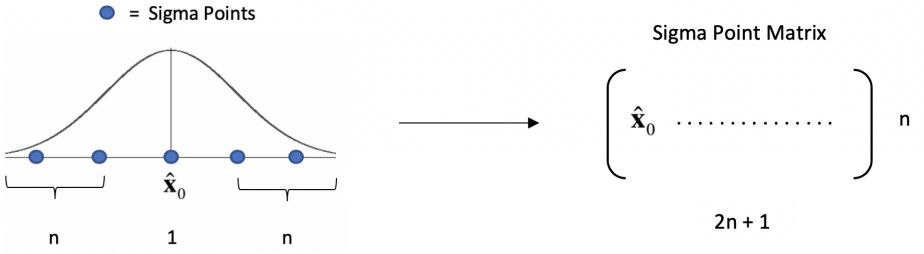


Figure 4.2: A brief illustration on how sigma points are used to characterize the Gaussian distribution of the data and how these sigma points are used to generate the Sigma Point Matrix.

After the sigma points undergo a nonlinear transformation, the resulting transformed sigma point matrix can be used to approximate the normal distribution of the data, which we can use to generate a prediction.

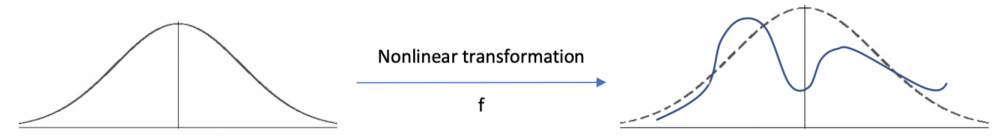


Figure 4.3: After the Gaussian-distributed data undergoes a nonlinear transformation, call it f, the result is no longer linear

One can see that this follows the basic outline of the previous versions of the KF, but the process of doing each step varies. More details about each step are explained below.

1. The first step is to initialize state vector, x_0, and covariance, P_{x_0}, which is exactly similar to both the KF and the EKF. Since we know that the x_0 is normally distributed, the expectation can be calculated by

$$x_0 = \mathbb{E}[x_0] = \sum_{i=a}^{n} x_i p_i = [x_a p_a + x_b p_b + \ldots + x_n p_n]^T,$$

where $x_0 = [x_a, x_b, \ldots, x_n]^T$, $p_a, p_b, \ldots, p_n$ is the respective probability of obtaining each state variable and T is the transpose. Next initialize the state covariance, by

$$P_{x_0} = \mathbb{E}[(x_0 - \hat{x}_0)(x_0 - \hat{x}_0)^T].$$

Unlike the KF and EKF, the UKF requires the calculation of sigma points, which is a way characterizing the distribution of the data. The number of sigma points is deterministic and depends on the dimensions of the system. In general, a UFK will have $2x + 1$ sigma points, where x represents the dimension of the state vector [10–12]. All sigma points are stored in a sigma point matrix, called χ. To get a general idea about the distribution of the data, we use λ, which is a scalar value that determines how spread out the sigma points are from the mean. λ can be calculated by

$$\lambda = \alpha^2(x + \kappa) - x,$$

where x is a scalar representing the dimension of the state vector, and α and κ are both parameters that control for the spread of sigma points around the mean value of the state. The spread of the sigma points is proportional to α. For both α and κ, the smaller the values are, the closer the sigma points are to the mean.

Another parameter of the UKF is β, which uses information regarding state distribution to adjust sigma points. β has a default value of 2 if the data is Gaussian (which is an assumption we will make throughout this paper).

Since the goal of this step is to characterize the distribution, set one of these sigma points to the mean, $\chi_{\ 0,k-1}$ which can be expressed as

$$\chi_{\ 0,k-1} = x_{k-1},$$

where 0 is a row on χ and $k-1$ is the time step. Then allocate half of the remaining points will be smaller than the mean and the other half will be larger than the mean by

$$\chi_{\ i,k-1} = \hat{x}_{k-1} + \left(\sqrt{(x+\lambda)P_{x_{k-1}}}\right)_i \qquad i = 1, \ldots, x,$$

$$\chi_{\ i,k-1} = \hat{x}_{k-1} - \left(\sqrt{(x+\lambda)P_{x_{k-1}}}\right)_{i-n} \qquad i = x+1,\ldots,2x+1,$$

where x is the dimension of the state vector, $(\sqrt{(d_x+\lambda)P_{x_{k-1}}})$ is a matrix, and the i subscript is the i^{th} column of the matrix. Recall that the square root of a matrix, satisfies the following condition: $A = B^2$, where A and B are both matrices.

Next, we must calculate a weight for each sigma point. Weights are scalars used to calculate posterior sigma points after they have undergone a nonlinear transformation. One set of weights, $W^{(m)}$, will be used to calculate the posterior mean while another set of weights $W^{(c)}$ will be used to calculate the posterior covariance. Weights can have positive or negative values, but will ultimately sum to 1 [2].

The initialized weight for the mean, $W_0^{(m)}$ can be found by

$$W_0^{(m)} = \frac{\lambda}{x+\lambda}.$$

Similarly, the weight for the covariance at the initial time step, $W_0^{(c)}$, is given by

$$W_0^{(c)} = \frac{\lambda}{x+\lambda} + (1-\alpha^2+\beta).$$

In later time steps, $W_i^{(m)}$ and$W_i^{(c)}$ follow the same equation, given by

$$W_i^{(m)} = W_i^{(c)} = \frac{\lambda}{2(d_x+\lambda)} \qquad i = 1,\ldots,2x.$$

2. A prediction can be generated by performing a nonlinear transformation on the sigma point matrix, χ, in order to generate a prediction and provide an update on the covariance matrix. After χ undergoes a nonlinear transformation, f, the result is a transformed sigma point matrix at time step k given the last time step, $k-1$, which will is called $\chi_{k|k-1}$ and is given by

$$\chi_{k|k-1} = f(\chi).$$

Though χ has a Gaussian distribution, $\chi_{k|k-1}$ does not because it has been transformed by the nonlinear state function f. The sum of the columns in $\chi_{k|k-1}$ and the weights calculated in step 3 will then be used to generate a prediction of the state variables, x_k, by

$$x_k = \sum_{i=0}^{2x} W_i^{(m)} \chi_{i,k|k-1}.$$

Adding the weights help approximate the Gaussian distribution of the state variables after they undergo a transformation.

3. Now that the prediction component of the filter is completed, we move on to the correction step, which begins by calculting the transformed covariance matrix and then transforming our predictions into a format that can be compared with system measurements. Calculation of the posterior (also called augmented) sigma points is necessary for converting system measurements into a format that can be compared with the state variables. System measurements must undergo a non linear transformation, h, resulting in a non Gaussian distribution. Therefore, the Unscented Transform is used again. Begin by calculating the posterior covariance matrix for the state variable, which is necessary for updating the state covariance later on by

$$P_{x,k|k-1} = \sum_{i=0}^{2x} W_i^{(c)} (\chi_{i,k|k-1} - x_k)(\chi_{i,k|k-1} - x_k)^T + Q,$$

where Q is process noise that provides the error in our model f, T is the transpose, and $W_i^{(c)}$ are the weights calculated earlier.

Then calculate augmented sigma points, $\chi^{(aug)}$, by

$$\chi_{0,k|k-1}^{(aug)} = x_k$$

$$\chi_{i,k|k-1}^{(aug)} = x_k \pm \left(\sqrt{(x+\lambda) P_{x_k}} \right)_i \qquad i = 1, \ldots, 2x$$

Recall that λ was calculated in step 2. Since λ is not time dependent, we can use the same value used earlier.

Now we calculate a sigma point matrix that represents the transformation of the prediction so that it can be compared with the states. This is necessary, especially in cases where measurements are not being obtained for all state variables. This sigma point matrix, $\mathcal{Y}_{k|k-1}$, can be obtained by having the $\chi_{k|k-1}$ undergo nonlinear transformation h, by

$$\mathcal{Y}_{k|k-1} = h(\chi_{k|k-1}^{(aug)}).$$

$\mathcal{Y}_{k|k-1}$ is a transformed sigma point matrix. While in this format, it cannot be compared with the state variables. However, $\mathcal{Y}_{k|k-1}$ can be used to convert the system measurements into a format, y_k, which can be found by

$$y_k = \sum_{i=0}^{2x} W_i^{(m)} \mathcal{Y}_{i,k|k-1}.$$

Now, the prediction can be compared with actual system measurements.

Then, we are able to calculate the Kalman Gain in order to determine how much to correct the model.

Unlike previous versions of the KF, in addition to calculating the covariance of the state variables, calculations is also done for the covariance of observations, P_y, and for state variables with observations, P_{xy}. When generating the covariance matrix for P_y the covariance of measurement noise is added. P_y can be found by

$$P_{y,k|k-1} = \sum_{i=0}^{2x} W_i^{(c)} (\mathcal{Y}_{i,k|k-1} - y_k)(\mathcal{Y}_{i,k|k-1} - y_k)^T + R,$$

where R is the covariance of measurement noise and $W_i^{(c)}$ are the weights calculated in step 3. Next, calculate P_{xy} by

$$P_{xy,k|k-1} = \sum_{i=0}^{2x} W_i^{(c)} (\chi_{i,k|k-1}^{(aug)} - x_k)(\mathcal{Y}_{i,k|k-1} - y_k)^T.$$

Finally, using P_y and P_{xy}, calculate the Kalman Gain, by

$$K_k = P_{xy,k|k-1}(P_{y,k|k-1})^{-1}.$$

Finally, we are able to correct the prediction and updating the covariance matrix, which will both be used in the next iteration. Similar to the UKF and the KF, the prediction of the state variables at the next time step, x_{k+1}, is given by

$$x_{k+1} = x_k + K_k(\hat{y}_k - y_k),$$

where $\hat{y}_k$ is system measurements. Conclude this iteration of the filter by preparing $P_{x,k}$ for the next iteration. $P_{x,k}$ can be found by

$$P_{x,k} = P_{x,k|k-1} - K_k(P_{y,k|k-1}){K_k}^T.$$

Table 4.1: Description of all variables in the Unscented Kalman Filter

Variables in the Unscented Kalman Filter		
Variable	Description	Dimensions
x	State variables	$x \times 1$
y	Transformed prediction	$y \times 1$
$\hat{y}$	Sytem measurements	$y \times 1$
χ	Sigma point matrix for states	$x \times (2x + 1)$
$\mathcal{Y}$	Sigma point matrix for obs	$y \times (2x + 1)$
f	Nonlinear state function	$x \times 1$
h	Nonlinear observation function	$y \times 1$
P_x	Covariance of states	$x \times x$
P_y	Covariance of observations	$y \times y$
P_{xy}	Covariance of states/obs	$x \times y$
Q	Process noise covariance	$x \times x$
R	Measurement noise covariance	$y \times y$
$W^{(m)}$	Weight for posterior mean	scalar
$W^{(c)}$	Weight for posterior covariance	scalar

Table 4.2: Description of all parameters in the Unscented Kalman Filter

Parameters in the Unscented Kalman Filter			
	Description	Bounds	Default
α	Controls spread of sigma points	$0 < \alpha \leq 1$	.001
β	Adjust sigma point weight	$\beta \geq 0$	2
κ	Sigma point weighting constant	$0 \leq \kappa \leq 3^*$	0

* Some use $\kappa = 3 - x$

4.2 Van der Pol Example

Applying the UKF to the Van der Pol oscillator will be used as a simple example to demonstrate the impacts of measurement and process noise. The Van der Pol equation, so named after its developer Balthasar Van der Pol, describes a self sustaining oscillator that create energy at small amplitudes and remove energy from large amplitudes. The Van der Pol equation describes a nonconservative oscillator (also known as a relaxation oscillator). Applications of the Van der Pol oscillators include circuits, vacuums, and modeling biological systems [13]. The Van der Pol oscillator is represented by a nonlinear second order differential equation:

$$\frac{d^2y}{dt^2} + \mu(y^2 - 1)\frac{dy}{dt} = 0$$

where μ is a damping coefficient, $\frac{d^2y}{dt^2}$ is acceleration, $\frac{dy}{dt}$ is velocity, and y is position. Therefore, for all $\mu < 0$, dampening occurs and the system tends to 0. The rate at which the system converges to zero is dependent on the size of μ, with larger values taking longer to converge and smaller values converging faster. If $\mu = 0$, the system becomes a simple harmonic oscillator, where motion is periodic. Lastly, if $\mu > 0$, the system enters a limit cycle, which is an isolated closed trajectory. [5].

The UKF can be applied to this nonlinear system to determine where the system will be at a point in time. To do so, relevant state variables include position and velocity. [1]

$$x_k = \begin{bmatrix} y \\ v \end{bmatrix}$$

Transforming the Van der Pol equation from a second order differential equation to a first order differential equation makes it easier to define f, the state function of the system. By substituting v for $\frac{dy}{dt}$, and $\frac{dv}{dt}$ for $\frac{d^2y}{dt^2}$ one can rewrite the Van der Pol equation as

$$\frac{dv}{dt} + \mu(y^2 - 1)v + y = 0.$$

From this, we get the differential equations associated with the state variables to generate the nonlinear transformation function f. For the sake of simplicity, assuming $\mu = 1$, we get

$$\dot{x}_k = \begin{bmatrix} \frac{dy}{dt} \\ \\ \frac{d^2y}{dt^2} \end{bmatrix} = \begin{bmatrix} v \\ \\ \frac{dv}{dt} \end{bmatrix} = \begin{bmatrix} v \\ \\ -1(y^2 - 1)v - y \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \\ 1 - y^2 - 1 - y \end{bmatrix} x_k = f.$$

[1] In Matlab, x(1) and x(2) represent position and velocity, respectively

In this particular example, the only measurement received from the system is position. Therefore, this filter is continually correcting for the position state state variable through measurement function h. Here, measurements of the system are simulated by adding noise to the position state variable. Even though there are only measurements for one state variable, we can still generate estimates of both state variables. In terms of parameters α, κ, and β, default values were used.

To simulate a Van der Pol Oscillator, an ordinary differential equation solver can be applied to state function f to generate true values of the system. Of course, systems do not perform perfectly; variations in model performance can be captured by randomly adding noise to the system.

All of this can be modeled on Matlab; all source code is from Matlab and can be referenced in Appendix A [**?**]

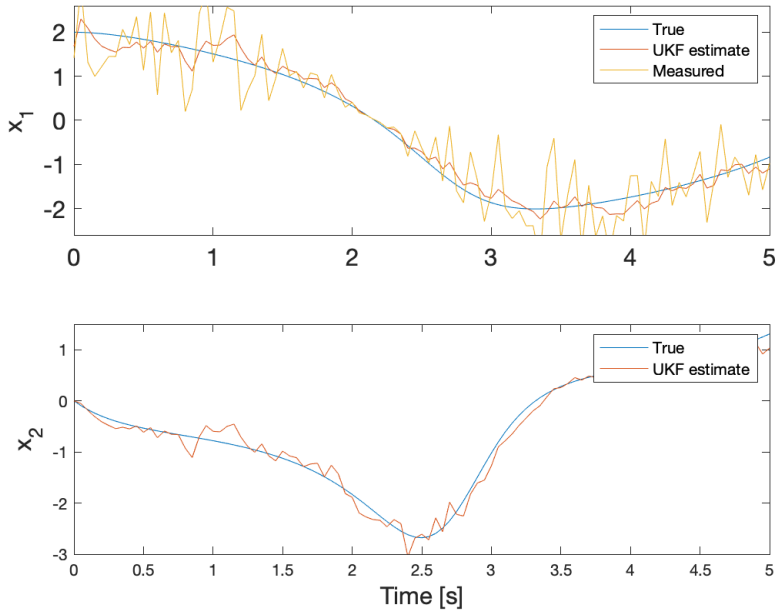


Figure 4.4: Performance of the UKF with R = 0.2, and Q = (0.02 0.1). As expected, the model converges on the true values of the system for both state variables. In this case, the only measurements we are receiving from the system are position, which is why the second state variable has no measured values.

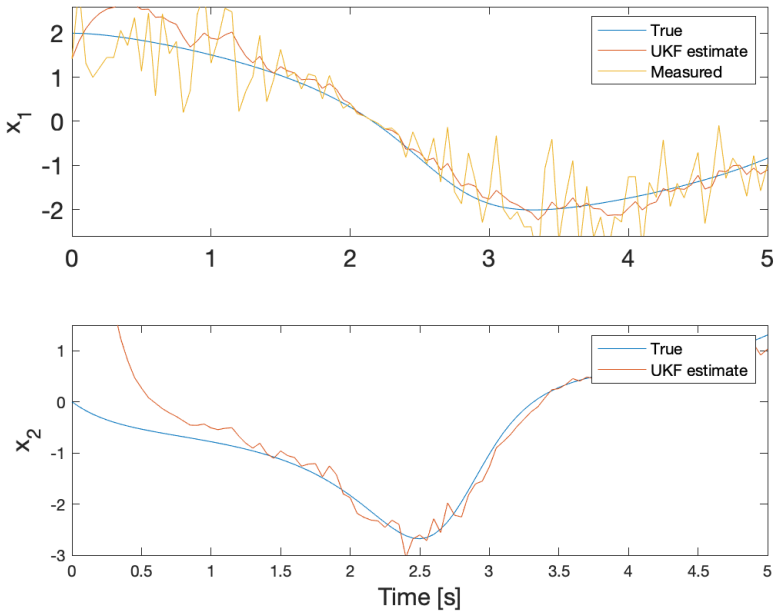
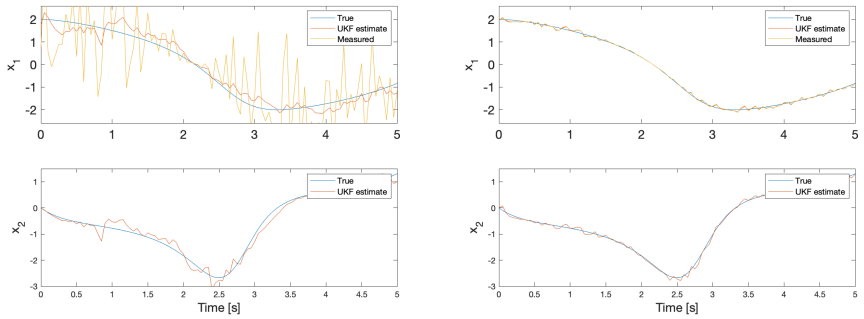
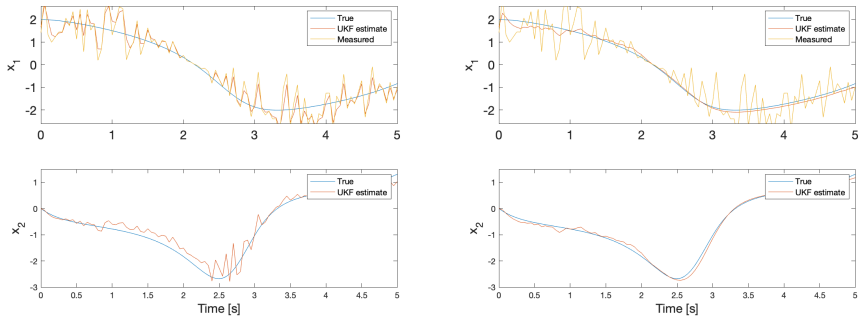


Figure 4.5: Performance of the UKF with poor initial conditions: (1, 7) instead of (2,0). Recall that inaccurate intial conditions cause convergence to take place more slowly. This seems to be the case, especially in the state variable that is not being corrected for.



(a) UKF with high measurement noise (0.9) (b) UKF with low measurement noise (0.002)

Figure 4.6: UKF on VDP oscillator with difference values of measurement noise. The model's behavior changes in response to the different values of measurement noise. Even when measurement noise is high, the UKF continues to perform well. Though, the rate of convergence appears to be slower. On the other hand, when measurement noise is low, the UKF seems to converge instantly with the measured values. The velocity state variable also quickly converges with the true value of the system.



(a) UKF with high process noise (0.9 0.8) (b) UKF with low process noise (0.0001 0.0001)

Figure 4.7: UKF on VDP oscillator with different values of process noise. Recall that process noise measures errors in the model. For the purpose of this example, process error was set to extreme high and low values. In theory, the Van der Pol equation should have small process error because it is a well used equation.

4.3 Modeling a Biological System

This example was inspired by a paper that uses an adaptation of the UKF, called the Iterative Unscented Kalman Filter (IUKF), to model the biological pathway of metabolites, which are molecules that are the byproduct of the body's metabolism. This model contained four different states, or metabolites, which contained 18 unknown parameters. These researchers utilized the IUKF for parameter fitting and was useful in enabling the model converge faster by resetting the covariance to re-excite the model. Utilizing the IUKF on this model was effective, because data regarding metabolites is highly influenced by noise, which is a factor that makes other approaches, such as regression and annealing, fail [8]

By skipping this step (as is done in the UKF), the three state variables without measurements converge significantly slower. A metabolite is small bodily structure that is the byproduct of the metabolism. Data regarding metabolites is highly influenced by noise, which is a factor that makes other approaches, such as regression and annealing, fail [8].

In the paper, researchers had access to their own data sources and used an approach that was adapted from the UKF. Though we are not using their exact dataset, we will be simulating data using the same approach as the previous example. However, this example will be following UKF algorithm, as opposed to the IUKF algorithm. By doing so, the state variables without incoming measurements converge significantly slower than the measurable states. Ultimately, the three goals of this example is to demonstrate how the UKF works on higher dimensional and more complex systems, how the UKF can correct for multiple variables or states, and how parameters can be adjusted to fit the model.

In this example, four metabolites, or states, are have the following differential equations:

$$\begin{aligned}
\dot{x}_1 &= \alpha_1 x_3^{g_{13}} x_5^{g_{15}} - \beta_1 x_1^{h_{11}}, \\
\dot{x}_2 &= \alpha_2 x_1^{g_{21}} - \beta_2 x_2^{h_{22}}, \\
\dot{x}_3 &= \alpha_3 x_2^{g_{32}} - \beta_3 x_3^{h_{33}} x_4^{h_{34}}, \\
\dot{x}_4 &= \alpha_4 x_1^{g_{41}} - \beta_4 x_4^{h_{44}},
\end{aligned}$$

where there are 18 unknown parameters ($\alpha_1, \ldots, \alpha_4, \beta_1, \ldots, \beta_4, g_{13}, g_{15}, g_{21}$, $g_{32}, g_{41}, h_{11}, h_{22}, h_{33}, h_{34}, h_{44}$). We have a guess as to what these values might be and we will use the UKF to find parameter values that best fit model to some state. In both the original example as well as this one, sampling time will be 0.1 seconds for 5 seconds, totaling 50 UKF estimates.

Since we did not have access to the original example's dataset, data was simulated on MATLAB. Similar to the previous example, the system's true output was simulated using an ODE solver and incoming system measurements were

Table 4.3: True Parameter Values

True Parameter Values										
	α_i	g_{i1}	g_{i2}	g_{i3}	g_{i4}	β_i	h_{i1}	h_{i2}	h_{i3}	h_{i4}
x_1	20.0	0	0	-0.8	0	10.0	0.5	0	0	0
x_2	8.0	.5	0	0	0	3.0	0	0.75	0	0
x_3	3.0	0	0.75	0	0	5.0	0	0	0.5	0.2
x_4	2.0	.5	0	0	0	6.0	0	0	0	0.8

calculated by adding noise to the true value. The model was initialized by setting the state variable to $x_0 = [4, 1, 3, 4]^T$ and the state covariance to $P_0 = .01I$. Table 4.3 shows the true values of the parameters and Figure 4.8 shows the results of the UKF given the true parameter values.

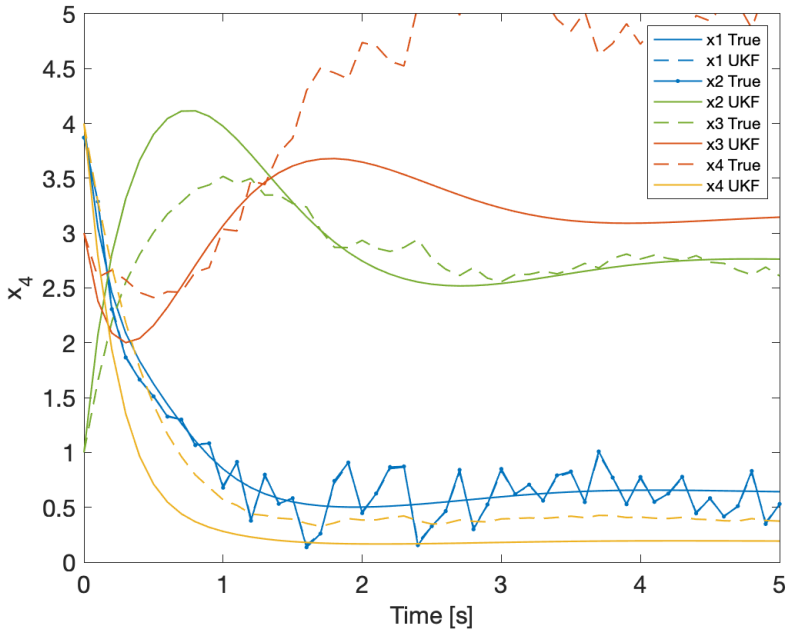


Figure 4.8: Performance of all state variables with the following conditions: R = 0.001, Q = (0.02 0.01 .03 .04) and initial values = $[4, 1, 3, 4]^T$.

Since the first state, $x1$ is the only state that has incoming system measurements, we have a 3 different blue lines in this figure. From this figure, it appears that the UKF prediction perfectly overlaps with the measured values. However, upon closer inspection, it actually does not. This is likely because there is only a

small amount of measurement noise to the system. Figure 4.9 highlights the same information as Figure 4.8, but separates the states into different graphs, enabling a clearer illustration of each state's behavior.

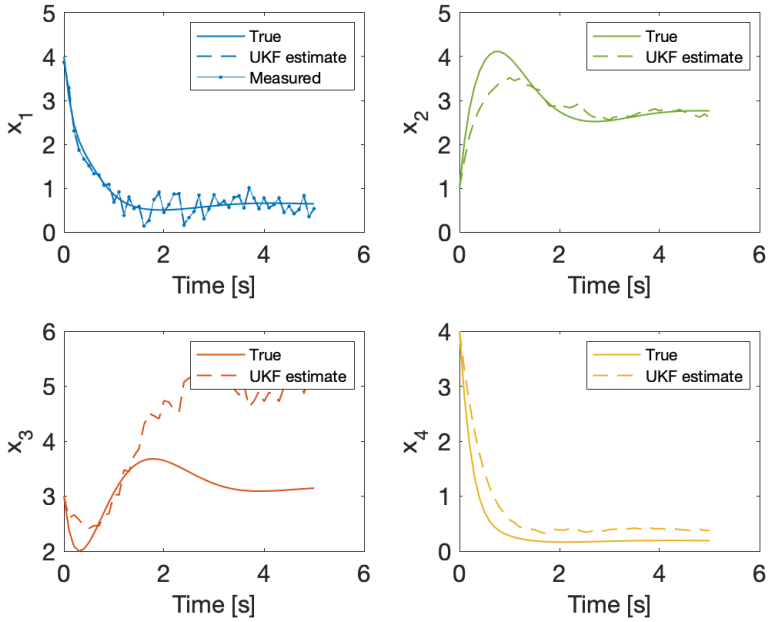


Figure 4.9: Performance of all four state variables with one corrected state. The first state variable (blue) is the only one that is receiving measurements from the system. This explains why it is more accurate and converges better with the system, as compared with the other three states.

In the original example, researchers used the following parameter values: $\epsilon = 1, \kappa = -14$. In theory, values of κ can be negative, but negative values cannot be inputted into MATLAB. Therefore, the parameter values used in this example are set to default values ($\alpha = 1e-3, \beta = 2, \kappa = 0$). Expand here on tweaking parameters.

Residuals, also known as the innovation, are one way to access the model's performance. Recall that the residual is the difference between the actual measurement values and the predicted measurement values. Since only one state, $x1$ had incoming measurements, the residual graph in Figure 4.10 is specifically for $x1$. Generally, a strong residual graph has

- a symmetrical distribution that is clustered toward the center,
- values that are close to 0,

- a random or unclear pattern.

Figure 4.10 exhibits these three characteristics, indicating strong model performance.

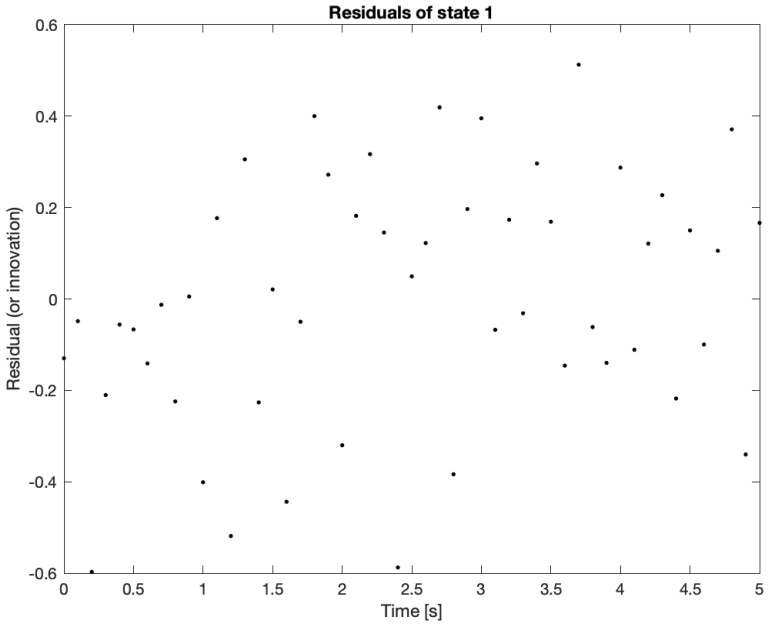


Figure 4.10: This scatterplot provides information regarding the residual, or innovation, of the first state, which is the only state recieving incoming system measurements.

Expand here on correcting for 1+ states, currently we are having technical challenges in completing this step

Bibliography

[1] Yi Cao. Learning the extended kalman filter - file exchange - matlab central, Jan 2008.

[2] J. Gove and DY Hollinger. Application of a dual unscented kalman filter for simultaneous state and parameter estimation in problems of surface-atmosphere exchange. *Journal of Geophysical Research*, 111:21 PP.–21 PP., 04 2006.

[3] Remembering rudolf e. kalman (1930 – 2016), Jul 2016.

[4] Professor rudolf emil kalman.

[5] Shuichi Kinoshita. Van der pol oscillator, 2013.

[6] David Kohanbash. Kalman filtering – a practical implementation guide (with code!), Jan 2014.

[7] Chot Hun Lim, Lee Yeng Ong, Tien Sze Lim, and Voon Chet Koo. Kalman filtering and its real-time applications. *Real-time Systems*, Jun 2016.

[8] Nader Meskin, Hazem Nounou, Mohamed Nounou, and Aniruddha Datta. Parameter estimation of biological phenomena: An unscented kalman filter approach. *IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM*, 10:537–43, 03 2013.

[9] Matthew Rhudy, Roger Salguero, and Keaton Holappa. A kalman filtering tutorial for undergraduate students. *International Journal of Computer Science & Engineering Survey*, 08:01–18, 02 2017.

[10] Esra Saatci and Aydin Akan. *Dual Unscented Kalman Filter and Its Applications to Respiratory System Modeling*, page 4. intechopen, 04 2009.

[11] Eric Wan and Ronell Merwe. The unscented kalman filter for nonlinear estimation. In *The Unscented Kalman Filter for Nonlinear Estimation*, volume 153-158, pages 153 – 158, 02 2000.

[12] Eric A. Wan and Rudolph Van Der Merwe. The unscented kalman filter. In *Kalman Filtering and Neural Networks*, pages 221–280. Wiley, 2001.

[13] Eric W Weisstein. Van der pol equation, Dec 2019.

Appendix A

Van der Pol Code

A.1 Main function

```
% Taken from https://www.mathworks.com/help/control/ug/nonlinear-state-estimation-using-unscented-kalman-filter.html

initialStateGuess = [2;0]; % xhat[k|k-1]
% Construct the filter
ukf = unscentedKalmanFilter(...
    @vdpStateFcn,... % State transition function
    @vdpMeasurementNonAdditiveNoiseFcn,... % Measurement function
    initialStateGuess,...
    'HasAdditiveMeasurementNoise',false);

R = 0.2; % Variance of the measurement noise v[k], original
%R = 0.002; % little measurement noise
%R = .9; % high measurement noise
ukf.MeasurementNoise = R;

ukf.ProcessNoise = diag([0.02  0.1]); %original
%ukf.ProcessNoise = diag([0.9  0.8]); %model is corrected alot
%ukf.ProcessNoise = diag([0.0001  0.0001]); %model is corrected little

T = 0.05; % [s] Filter sample time
timeVector = 0:T:5;
[~,xTrue]=ode45(@vdp1,timeVector,[2;0]);
```

```
23
24 rng(1); % Fix the random number generator for
       reproducible results
25 yTrue = xTrue(:,1);
26 yMeas = yTrue .* (1+sqrt(R)*randn(size(yTrue))); % sqrt(R
       ): Standard deviation of noise
27
28 Nsteps = numel(yMeas); % Number of time steps
29 xCorrectedUKF = zeros(Nsteps,2); % Corrected state
       estimates
30 PCorrected = zeros(Nsteps,2,2); % Corrected state
       estimation error covariances
31 e = zeros(Nsteps,1); % Residuals (or innovations)
32
33 for k=1:Nsteps
34     % Let k denote the current time.
35     %
36     % Residuals (or innovations): Measured output -
           Predicted output
37     e(k) = yMeas(k) - vdpMeasurementFcn(ukf.State); % ukf
           .State is x[k|k-1] at this point
38     % Incorporate the measurements at time k into the
           state estimates by
39     % using the "correct" command. This updates the State
            and StateCovariance
40     % properties of the filter to contain x[k|k] and P[k|
           k]. These values
41     % are also produced as the output of the "correct"
           command.
42     [xCorrectedUKF(k,:), PCorrected(k,:,:)] = correct(ukf
           ,yMeas(k));
43     % Predict the states at next time step, k+1. This
           updates the State and
44     % StateCovariance properties of the filter to contain
            x[k+1|k] and
45     % P[k+1|k]. These will be utilized by the filter at
           the next time step.
46     predict(ukf);
47 end
48
49
50 figure();
51 subplot(2,1,1);
52 plot(timeVector,xTrue(:,1),timeVector,xCorrectedUKF(:,1),
       timeVector,yMeas(:));
53 set(gca, 'FontSize', 15);
```

```
54 e=legend('True','UKF estimate','Measured')
55 e.FontSize = 10;
56 ylim([-2.6  2.6]);
57 ylabel('x_1', 'FontSize', 15);
58 subplot(2,1,2);%
59 plot(timeVector,xTrue(:,2),timeVector,xCorrectedUKF(:,2))
       ;
60 f=legend('True','UKF estimate')
61 f.FontSize = 10;
62 set(gca, 'FontSize', 10);
63 ylim([-3  1.5]);
64 xlabel('Time [s]', 'FontSize', 15);
65 ylabel('x_2', 'FontSize', 15);
66
67 %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
       VDP_highMN_highPN.png')
```

A.2 State function

```
1  function x = vdpStateFcn(x)
2  % vdpStateFcn Discrete-time approximation to van der Pol
       ODEs for mu = 1.
3  % Sample time is 0.05s.
4  %
5  % Example state transition function for discrete-time
       nonlinear state
6  % estimators.
7  %
8  % xk1 = vdpStateFcn(xk)
9  %
10 % Inputs:
11 %    xk - States x[k]
12 %
13 % Outputs:
14 %    xk1 - Propagated states x[k+1]
15 %
16 % See also extendedKalmanFilter, unscentedKalmanFilter
17
18 %   Copyright 2016 The MathWorks, Inc.
19
20 %#codegen
21
22 % The tag %#codegen must be included if you wish to
       generate code with
23 % MATLAB Coder.
24
```

```
% Euler integration of continuous-time dynamics x'=f(x)
    with sample time dt
dt = 0.05; % [s] Sample time
x = x + vdpStateFcnContinuous(x)*dt;
end

function dxdt = vdpStateFcnContinuous(x)
%vdpStateFcnContinuous Evaluate the van der Pol ODEs for
    mu = 1
dxdt = [x(2); (1-x(1)^2)*x(2)-x(1)];
end
```

A.3 Measurement function

```
initialStateGuess = [4;1;3;4]; % xhat[k|k-1]
%initialStateGuess = [1;4;2;5]; % bad intials

% Construct the filter
ukf = unscentedKalmanFilter(...
    @MeskinStateFcn,... % State transition function
    @MeskinMeasurementNonAddFcn,... % Measurement
        function
    initialStateGuess,...
    'HasAdditiveMeasurementNoise',false)%,...
    %'alpha', 0.9,...
    %'kappa', 3,...
    %'beta', 2);

R = .001; % Variance of the measurement noise v[k]
ukf.MeasurementNoise = R;

ukf.ProcessNoise = diag([0.02 0.01 .03 .04]); %stores the
    process noise covariance

T = 0.1; % [s] Filter sample time
timeVector = 0:T:5;
[~,xTrue]=ode45(@Meskin1,timeVector,initialStateGuess);

rng(1); % Fix the random number generator for
    reproducible results
yTrue = xTrue(:,1);
yMeas = yTrue + (sqrt(R)*randn(size(yTrue)));
% sqrt(R): Standard deviation of noise
% randn(size(yTrue)): randomly sample 51 elements from
    ytrue
%yMeas = yTrue; Assumes no noise
```

```
29
30 for k=1:numel(yMeas)
31     % Let k denote the current time.
32
33     % Residuals (or innovations): Measured output -
           Predicted output
34     %e(k) = yMeas(k) - MeskinMeasurementFcn(ukf.State); %
           ukf.State is x[k|k-1] at this point
35
36     % Incorporate the measurements at time k into the
           state estimates by
37     % using the "correct" command. This updates the State
           and StateCovariance
38     % properties of the filter to contain x[k|k] and P[k|
           k]. These values
39     % are also produced as the output of the "correct"
           command.
40
41     [xCorrectedUKF(k,:), PCorrected(k,:,:)] = correct(ukf
           , yMeas(k));
42
43     % Predict the states at next time step, k+1. This
           updates the State and
44     % StateCovariance properties of the filter to contain
           x[k+1|k] and
45     % P[k+1|k]. These will be utilized by the filter at
           the next time step.
46     predict(ukf);
47
48 end
49
50
51 figure();
52 color1=[0,0.4470, 0.7410]; %blue
53 color2=[0.4660 0.6740 0.1880]; %green
54 color3=[0.8500 0.3250 0.0980]; %orange
55 color4=[0.9290 0.6940 0.1250]; %yellow
56
57 subplot(2,2,1);
58 plot(timeVector, xTrue(:,1), 'Color', color1, 'LineStyle'
       , '-',...
59     'LineWidth', 1)
60 hold on
61 plot(timeVector,xCorrectedUKF(:,1), 'Color', color1, '
       LineStyle', '--',...
62     'LineWidth', 1)
```

```
63  hold on
64  plot(timeVector,yMeas, 'Color', color1, 'Marker', '.',...
65      'LineWidth', 1)
66
67  set(gca, 'FontSize', 15);
68  set(gcf, 'Color', 'None');
69  a = legend('True','UKF estimate','Measured')
70  a.FontSize = 10;
71  ylim([0 5]);
72  ylabel('x_1', 'FontSize', 15);
73  xlabel('Time [s]', 'FontSize', 15);
74
75  subplot(2,2,2);
76  plot(timeVector, xTrue(:,2), 'Color', color2, 'LineStyle'
        , '_',...
77      'LineWidth', 1)
78  hold on
79  plot(timeVector,xCorrectedUKF(:,2), 'Color', color2, '
        LineStyle', '--',...
80      'LineWidth', 1)
81  set(gca, 'FontSize', 15);
82  b = legend('True','UKF estimate')
83  b.FontSize = 10;
84  ylim([0 5]);
85  xlabel('Time [s]', 'FontSize', 15);
86  ylabel('x_2', 'FontSize', 15);
87  xlabel('Time [s]', 'FontSize', 15);
88
89  subplot(2,2,3);
90  plot(timeVector, xTrue(:,3), 'Color', color3, 'LineStyle'
        , '_',...
91      'LineWidth', 1)
92  hold on
93  plot(timeVector,xCorrectedUKF(:,3), 'Color', color3, '
        LineStyle', '--',...
94      'LineWidth', 1)
95  set(gca, 'FontSize', 15);
96  c = legend('True','UKF estimate')
97  c.FontSize = 10;
98  %ylim([-2.6 2.6]);
99  ylabel('x_3', 'FontSize', 15);
100 xlabel('Time [s]', 'FontSize', 15);
101
102 subplot(2,2,4);
103 plot(timeVector, xTrue(:,4), 'Color', color4, 'LineStyle'
        , '_',...
```

```
104       'LineWidth', 1)
105   hold on
106   plot(timeVector,xCorrectedUKF(:,4), 'Color', color4, '
          LineStyle', '--',...
107       'LineWidth', 1)
108   set(gca, 'FontSize', 15);
109   d = legend('True','UKF estimate','Measured')
110   d.FontSize = 10;
111   %ylim([-2.6 2.6]);
112   ylabel('x_4', 'FontSize', 15);
113   xlabel('Time [s]', 'FontSize', 15);
114
115   %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
          Meskin_states_badIntial.png')
116   %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
          Meskin_states.png')
117
118   subplot(1,1,1);
119
120   plot(timeVector, xTrue(:,1), 'Color', color1, 'LineStyle'
          , '-',...
121       'LineWidth', 1)
122   hold on
123   plot(timeVector,xCorrectedUKF(:,1), 'Color', color1, '
          LineStyle', '--',...
124       'LineWidth', 1)
125   hold on
126   plot(timeVector,yMeas, 'Color', color1, 'Marker', '.',...
127       'LineWidth', 1)
128   hold on
129   plot(timeVector, xTrue(:,2), 'Color', color2, 'LineStyle'
          , '-',...
130       'LineWidth', 1)
131   hold on
132   plot(timeVector,xCorrectedUKF(:,2), 'Color', color2, '
          LineStyle', '--',...
133       'LineWidth', 1)
134   hold on
135   plot(timeVector, xTrue(:,3), 'Color', color3, 'LineStyle'
          , '-',...
136       'LineWidth', 1)
137   hold on
138   plot(timeVector,xCorrectedUKF(:,3), 'Color', color3, '
          LineStyle', '--',...
139       'LineWidth', 1)
140   hold on
```

```
141 plot(timeVector, xTrue(:,4), 'Color', color4, 'LineStyle'
        , '-',...
142     'LineWidth', 1)
143 hold on
144 plot(timeVector,xCorrectedUKF(:,4), 'Color', color4, '
    LineStyle', '--',...
145     'LineWidth', 1)
146
147 set(gca, 'FontSize', 15);
148 set(gcf, 'color', 'none');
149 set(gca, 'color', 'none');
150 e = legend('x1 True','x1 UKF','x2 True', 'x2 UKF', 'x3
    True','x3 UKF','x4 True', 'x4 UKF')
151 e.FontSize = 10;
152 ylim([0 5]);
153 ylabel('x_4', 'FontSize', 15);
154 xlabel('Time [s]', 'FontSize', 15);
155
156 %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_overall_badIntial.png')
157 saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_overall.png')
```

Appendix B

Metabolites Example Code

B.1 Main function

```
initialStateGuess = [4;1;3;4]; % xhat[k|k-1]
%initialStateGuess = [1;4;2;5]; % bad intials

% Construct the filter
ukf = unscentedKalmanFilter(...
    @MeskinStateFcn,... % State transition function
    @MeskinMeasurementNonAddFcn,... % Measurement
        function
    initialStateGuess,...
    'HasAdditiveMeasurementNoise',false)%,...
    %'alpha', 0.9,...
    %'kappa', 3,...
    %'beta', 2);

R = .001; % Variance of the measurement noise v[k]
ukf.MeasurementNoise = R;

ukf.ProcessNoise = diag([0.02 0.01 .03 .04]); %stores the
    process noise covariance

T = 0.1; % [s] Filter sample time
timeVector = 0:T:5;
[~,xTrue]=ode45(@Meskin1,timeVector,initialStateGuess);

rng(1); % Fix the random number generator for
    reproducible results
yTrue = xTrue(:,1);
yMeas = yTrue + (sqrt(R)*randn(size(yTrue)));
```

```
% sqrt(R): Standard deviation of noise
% randn(size(yTrue)): randomly sample 51 elements from ytrue
%yMeas = yTrue; Assumes no noise

for k=1:numel(yMeas)
    % Let k denote the current time.

    % Residuals (or innovations): Measured output - Predicted output
    %e(k) = yMeas(k) - MeskinMeasurementFcn(ukf.State); % ukf.State is x[k|k-1] at this point

    % Incorporate the measurements at time k into the state estimates by
    % using the "correct" command. This updates the State and StateCovariance
    % properties of the filter to contain x[k|k] and P[k|k]. These values
    % are also produced as the output of the "correct" command.

    [xCorrectedUKF(k,:), PCorrected(k,:,:)] = correct(ukf, yMeas(k));

    % Predict the states at next time step, k+1. This updates the State and
    % StateCovariance properties of the filter to contain x[k+1|k] and
    % P[k+1|k]. These will be utilized by the filter at the next time step.
    predict(ukf);

end


figure();
color1=[0,0.4470, 0.7410]; %blue
color2=[0.4660 0.6740 0.1880]; %green
color3=[0.8500 0.3250 0.0980]; %orange
color4=[0.9290 0.6940 0.1250]; %yellow

subplot(2,2,1);
plot(timeVector, xTrue(:,1), 'Color', color1, 'LineStyle', '-',...
    'LineWidth', 1)
```

```
60  hold on
61  plot(timeVector,xCorrectedUKF(:,1), 'Color', color1, '
        LineStyle', '--',...
62      'LineWidth', 1)
63  hold on
64  plot(timeVector,yMeas, 'Color', color1, 'Marker', '.',...
65      'LineWidth', 1)
66
67  set(gca, 'FontSize', 15);
68  set(gcf, 'Color', 'None');
69  a = legend('True','UKF estimate','Measured')
70  a.FontSize = 10;
71  ylim([0 5]);
72  ylabel('x_1', 'FontSize', 15);
73  xlabel('Time [s]', 'FontSize', 15);
74
75  subplot(2,2,2);
76  plot(timeVector, xTrue(:,2), 'Color', color2, 'LineStyle'
        , '-',...
77      'LineWidth', 1)
78  hold on
79  plot(timeVector,xCorrectedUKF(:,2), 'Color', color2, '
        LineStyle', '--',...
80      'LineWidth', 1)
81  set(gca, 'FontSize', 15);
82  b = legend('True','UKF estimate')
83  b.FontSize = 10;
84  ylim([0 5]);
85  xlabel('Time [s]', 'FontSize', 15);
86  ylabel('x_2', 'FontSize', 15);
87  xlabel('Time [s]', 'FontSize', 15);
88
89  subplot(2,2,3);
90  plot(timeVector, xTrue(:,3), 'Color', color3, 'LineStyle'
        , '-',...
91      'LineWidth', 1)
92  hold on
93  plot(timeVector,xCorrectedUKF(:,3), 'Color', color3, '
        LineStyle', '--',...
94      'LineWidth', 1)
95  set(gca, 'FontSize', 15);
96  c = legend('True','UKF estimate')
97  c.FontSize = 10;
98  %ylim([-2.6 2.6]);
99  ylabel('x_3', 'FontSize', 15);
100 xlabel('Time [s]', 'FontSize', 15);
```

```
101
102 subplot(2,2,4);
103 plot(timeVector, xTrue(:,4), 'Color', color4, 'LineStyle'
    , '-',...
104     'LineWidth', 1)
105 hold on
106 plot(timeVector,xCorrectedUKF(:,4), 'Color', color4, '
    LineStyle', '--',...
107     'LineWidth', 1)
108 set(gca, 'FontSize', 15);
109 d = legend('True','UKF estimate','Measured')
110 d.FontSize = 10;
111 %ylim([-2.6 2.6]);
112 ylabel('x_4', 'FontSize', 15);
113 xlabel('Time [s]', 'FontSize', 15);
114
115 %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_states_badIntial.png')
116 %saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_states.png')
117
118 subplot(1,1,1);
119
120 plot(timeVector, xTrue(:,1), 'Color', color1, 'LineStyle'
    , '-',...
121     'LineWidth', 1)
122 hold on
123 plot(timeVector,xCorrectedUKF(:,1), 'Color', color1, '
    LineStyle', '--',...
124     'LineWidth', 1)
125 hold on
126 plot(timeVector,yMeas, 'Color', color1, 'Marker', '.',...
127     'LineWidth', 1)
128 hold on
129 plot(timeVector, xTrue(:,2), 'Color', color2, 'LineStyle'
    , '-',...
130     'LineWidth', 1)
131 hold on
132 plot(timeVector,xCorrectedUKF(:,2), 'Color', color2, '
    LineStyle', '--',...
133     'LineWidth', 1)
134 hold on
135 plot(timeVector, xTrue(:,3), 'Color', color3, 'LineStyle'
    , '-',...
136     'LineWidth', 1)
137 hold on
```

```
plot(timeVector,xCorrectedUKF(:,3), 'Color', color3, '
    LineStyle', '--',...
     'LineWidth', 1)
hold on
plot(timeVector, xTrue(:,4), 'Color', color4, 'LineStyle'
    , '-',...
     'LineWidth', 1)
hold on
plot(timeVector,xCorrectedUKF(:,4), 'Color', color4, '
    LineStyle', '--',...
     'LineWidth', 1)

set(gca, 'FontSize', 15);
set(gcf, 'color', 'none');
set(gca, 'color', 'none');
e = legend('x1 True','x1 UKF','x2 True', 'x2 UKF', 'x3
    True','x3 UKF','x4 True', 'x4 UKF')
e.FontSize = 10;
ylim([0 5]);
ylabel('x_4', 'FontSize', 15);
xlabel('Time [s]', 'FontSize', 15);

%saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_overall_badIntial.png')
saveas(gcf,'\Users\lindseytam\Desktop\thesis\
    Meskin_overall.png')
```

B.2 State function

```
function x = MeskinStateFcn(x)
% Trying Meskin in two dimensions
dt = 0.05;
t = 0; % dummy time variable for MeskinODE
x = x + MeskinODE(t, x)*dt;
end

%{
function dxdt = MeskinStateFcnContinuous(x)
parameter_values;

dxdt = [a_1 * x(3)^g_13 - b_1 * x(1)^h_11;
        a_2 * x(1)^g_21 - b_2 * x(2)^h_22;
        a_3 * x(2)^g_32 - b_3 * x(3)^h_33 * x(4)^h_34;
        a_4 * x(1)^g_41 - b_4 * x(4)^h_44];
end
%}
```

B.3 Measurement function

```
function yk = MeskinMeasurementNonAddFcn(xk,vk)
% Author: Ltam
% Date: November 17, 2019
% Summary:
% Inputs:    xk = states at time k, x[k]
%            vk = measurement noise vector at time, k v[k]
% Outputs:   yk = measurements at time k
%
% The measurement is the first state with multiplicative
    noise
yk = xk(1)+vk;
%yk =xk*(1+vk);
end
```

B.4 Parameter Values

```
%This script saves the baseline parameter values for our
    model

a_1 = 20;
a_2 = 8;
a_3 = 3;
a_4 = 2;

b_1 = 10;
b_2 = 3;
b_3 = 5;
b_4 = 6;

g_11 = 0;
g_21 = 0.5;
g_31 = 0;
g_41 = 0.5;

g_12 = 0;
g_22 = 0;
g_32 = 0.75;
g_42 = 0;

g_13 = -0.8;
g_23 = 0;
g_33 = 0;
g_43 = 0;
```

```
27
28  g_14 = 0;
29  g_24 = 0;
30  g_34 = 0;
31  g_44 = 0;
32
33  h_11 = 0.5;
34  h_21 = 0;
35  h_31 = 0;
36  h_41 = 0;
37
38  h_12 = 0;
39  h_22 = 0.75;
40  h_32 = 0;
41  h_42 = 0;
42
43  h_13 = 0;
44  h_23 = 0;
45  h_33 = 0.5;
46  h_43 = 0;
47
48  h_14 = 0;
49  h_24 = 0;
50  h_34 = 0.2;
51  h_44 = 0.8;
```