Use of Kalman Filtering in State and Parameter Estimation of Diabetes Models

Cassidy Lê

Lisette de Pillis, Advisor

Blerta Shtylla, Reader

Department of Mathematics

May, 2020

Abstract

Diabetes continues to affect many lives every year, putting those affected by it at higher risk of serious health issues. Despite many efforts, there currently is no cure for diabetes. Nevertheless, researchers continue to study diabetes in hopes of understanding the disease and how it affects people, creating mathematical models to simulate the onset and progression of diabetes. Recent research in [4] has suggested that these models can be furthered through the use of Data Assimilation, a regression method that synchronizes a model with a particular set of data by estimating the system's states and parameters. In my thesis, I explore how Data Assimilation, specifically different types of Kalman filters, can be applied to various models, including a diabetes model.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to start by acknowledging that Harvey Mudd College sits upon the original homelands of the Tongvan people. Long before this institution settled here, Tongvan people have been children, students, and caretakers of this land. Despite continuous forced displacement, Tongvan efforts for proper healthcare, economic justice, and education persist. Unfortunately, the 2010 U.S. Census Bureau reports that only about 14% of Tongvans have a bachelor's degree, which is less than half that of the total U.S. population. As a student researcher of HMC and beneficiary from settler colonialism and the U.S. higher education system, I would like to show my gratitude for the Tongvan elders both past and present, honoring their lives and connection to this land. By doing so, I hope to consider my positionality in the story of colonization and the undoing of its legacy.

Additionally, I would like to express how grateful I am for the researchers who precede me, particularly Gianna Wu, Amber Nguyen, Nat Efrat-Henrici, Matt Matusiewicz, and An Do. Without your extensive research and impeccable documentation, I would not have the understanding of type 1 diabetes that I have now or the ability to dive into such a rich topic. For that, I am incredibly grateful.

Special thanks to Lindsey Tam for working with me through this challenging research. You have helped me in so many ways, both in the mathematical field and companionship. I just hope that I was able to provide the same for you as you worked through your thesis.

Most importantly, I'd like to thank Professor Blerta Shtylla and Professor Lisette de Pillis. Throughout this project, they have provided an enormous amount of guidance and support. Their passion and perseverance inspired me to produce the work that I did. Thank you so much!

Chapter 1

Background of Diabetes

According to the American Diabetes Association, every year, approximately 1.5 million Americans are diagnosed with diabetes [2], a chronic disease in which one's body exhibits excessive levels of blood glucose due to a lack of insulin. Those affected by diabetes have a higher risk of serious health issues, including neuropathy (nerve damage), nephropathy (kidney disease), ketoacidosis (excessive levels of ketones causing the blood to become acidic), and strokes (interruption of blood supply to the brain).

Unfortunately, a cure for diabetes does not currently exist [2]. However, the disease can go into remission, and oftentimes people manage it with medication and lifestyle changes [3]. Some common methods of management include taking insulin, eating healthy foods, exercising regularly, and frequently monitoring blood sugar. Depending on what type of diabetes a person has, their treatment plans may vary. The three main types of diabetes are type 1 diabetes, type 2 diabetes, and gestational diabetes. Type 1 diabetes (T1D) is an autoimmune condition that is believed to stem from genetics and/or environmental factors [18]. Type 2 diabetes (T2D) is an insulin resistance condition that is believed to be a result of genetics, lack of exercise, and/or being overweight [18]. Gestational diabetes is a result of insulin-blocking hormones produced during pregnancy and only occurs during pregnancy [18]. Gestational diabetes is not as common as T1D or T2D, so in this paper I focus on T1D and T2D.

1.1 Type 1 Diabetes

Unlike other types of diabetes, T1D is decidedly autoimmune, which causes more difficulty with potential preventions, treatments, and cures. Additionally, it is often diagnosed in early childhood. Although there is no definitive cause of T1D, scientists believe that it is a combination of genetic and environmental factors [20]. Despite its unknown cause, decades of studies and research have produced a collective understanding of the development of T1D and how it affects the autoimmune system.

T1D is characterized by an autoimmune attack on β-cells located in the pancreas, which produce insulin. Insulin is a hormone that binds with cells, allowing them to absorb glucose and consequently lowering blood glucose levels [20]. This physiological dynamic between insulin and glucose is visually portrayed in **Figure 1.1**. Additionally, biological terminology is further defined in **Appendix A**. The T1D autoimmune attack on β-cells results in a deficiency of insulin, which then causes a rise in glucose levels to a point that is unhealthy [20].

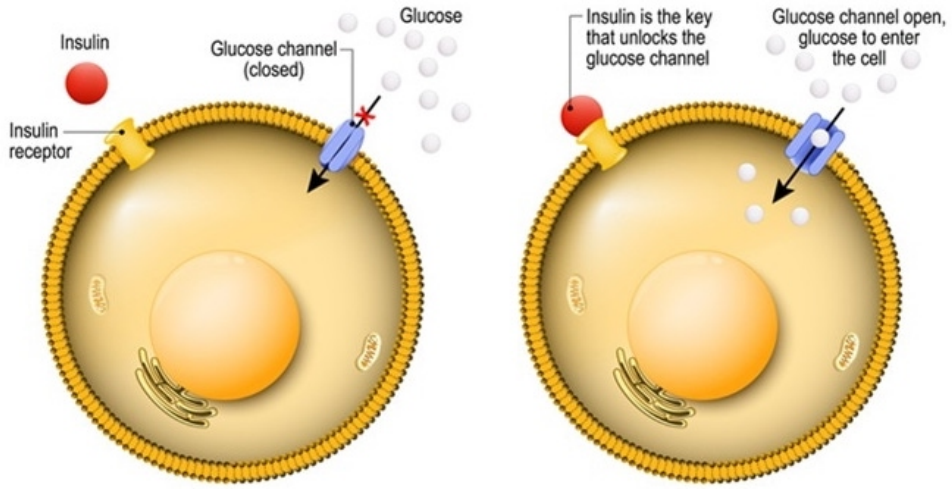

Figure 1.1 Visualization of dynamics between insulin and glucose taken from [14]. **Left**: Diabetes results in a lack of insulin to connect with the cell so it blocks the glucose channel, preventing glucose to enter the cell and be processed. **Right**: Healthy cell with sufficient insulin that connects with the cell and opens the glucose channel, allowing glucose to enter the cell and be processed.

1.2 Type 2 Diabetes

Typically, T2D initially starts as insulin resistance, which means one's body does not efficiently use insulin. As previously mentioned insulin enables cells to absorb glucose, and thus lowers blood glucose levels. See **Figure 1.1** for a visual representation of this physiological interaction.

Insulin resistance prompts the pancreas to produce more insulin in order to compensate for the inefficient use of insulin. However, at some point the pancreas cannot produce enough insulin to sufficiently supply the demand. As a result, insulin production decreases, causing a deficiency in insulin [18]. Like T1D, this then causes a rise in glucose levels to an unhealthy degree.

1.3 Diabetes Models

Before scientists can determine a cure for diabetes, they must first gain a robust understanding of the onset and progression of this disease. In order to do so, researchers have created mathematical models to simulate the immune system and relevant components in the body. These models are intended to provide proper insight for diabetes treatments, allowing scientists to test the actions and effectiveness of treatments as they are produced. Hopefully, as new discoveries are uncovered and a potential successful treatment is introduced, researchers will be able to use these models to determine the performance of a treatment on an individual with diabetes.

In this paper, I build on an existing dynamical model for T2D. In particular, I work with a T2D model proposed in [4]. In the future, I hope to extend my work with this T2D model to a T1D model from [20]. These two models are further explained in the following sections.

1.3.1 Type 1 Diabetes Model

The T1D model in [20] is a single compartment mathematical model that reflects important qualitative features of T1D progression and dynamics [20]. Through this model, there are two goals. One is to determine what influence tolerogenic dendritic cells (tDCs) have in the development of T1D in the presence of a β-cell apoptotic wave in non-obese diabetic (NOD) and non-diabetic (Balb/c) mice. The other goal is to determine the significance

that timing and dosing of tDC injections have on NOD mice's ability to escape the onset of T1D [20].

To provide more details, the single compartment model from [20] accounts for the following biological components involved in the onset and progression of T1D: dendritic cells (both tolerogenic and immunogenic), T cells (effector and regulatory), and macrophages. It is single compartment because it simulates a single, well-mixed pancreatic compartment. This model is based on an earlier model in [15]. The researchers of [20] extend the model from [15] by including healthy β-cell counts as well as antigen presenting and T-cell populations [20].

To incorporate the DC effect on the development of T1D, the researchers of [20] conduct stochastic parameter fitting techniques on human experimental DC data collected in [7]. For the overall simplified model, they fit the parameters using Maximum Likelihood Estimate (MLE), implementing the Metropolis Monte Carlo Markov-Chain (MCMC) algorithm[20].

After running the model, they find that if a mouse reduces efficacy of microphages and undergoes a wave of increased β-cell death, then said mouse enters a diseased steady state. Additionally, many of the parameters are measured or inferred, which requires a sensitivity analysis in order to assess the parameters chosen. From the sensitivity analysis, it is determined that NOD mice are much more sensitive to single parameter changes than Balb/c mice, so Balb/c mice have a more robust healthy system overall.

The strong sensitivity to parameter changes in the model from [20] suggests that there needs to be more work to stabilize parameter changes. Luckily, there are increasingly more sophisticated techniques to fit parameters. For example, one method that would be incredibly useful in predicting the correct parameter values based on the population of the data set is a method proposed in [4] called data assimilation. Data assimilation is further discussed in **chapter 2**.

1.3.2 Type 2 Diabetes Model

The T2D model that is used in [4] is one that represents simple physiologic mechanics of glucose and insulin. It is taken from an earlier study by David J Albers, George Hripcsak, and Michael Schmidt [6]. Originally, this

model was developed in [21], a study by Jeppe Sturis, Kenneth S. Polonsky, Erik Mosekilde, and Eve Van Cauter.

In general, the T2D model serves as a dynamical representation of ultradian oscillations of insulin and glucose. More colloquially, the model describes how an individual's blood sugar levels change based on that person's eating habits. Initially, researchers from [21] developed this model to determine whether the ultradian oscillations in human insulin secretion could result from the feedback loops between insulin and glucose. It consists of two major negative feedback loops: one describes the effects of insulin on the utilization of glucose, and the other describes the effects of insulin on the production of glucose. Both of these loops account for the stimulator effect of glucose on insulin secretion [21].

For the study in [4], this ultradian model is used in their research for accurate parameter fitting. Like [20], [4] notes the importance of parameter fitting in order to produce an accurate model. In the next chapter, I further discuss the research in T2D and parameter fitting detailed in [4]. Specifically, I detail their research in a parameter fitting technique called data assimilation.

Chapter 2

Data Assimilation in Bioinformatics

Mathematical modeling of biological systems, such as the T1D model from [20], is under the larger field of study called bioinformatics. Bioinformatics is an interdisciplinary field that uses computer science, mathematics and statistics to analyze biological information. To further advance studies in bioinformatics, the writers of [4] argue for an increase in the use of data assimilation (DA). In particular, they argue that DA significantly improves accuracy in research and modeling of population physiology. At the beginning of the paper, they claim that many mathematical models approximate dynamical systems and, in the case of diabetes, vary significantly by patient. To personalize these models, they advocate the use of DA.

Data assimilation is defined as a regression method that uses a particular data set to estimate the states and parameters of a mathematical model [4]. This results in a model that is accurately fitted to that set of data. It does so by combining mathematical theory, mainly dynamical systems as well as some statistical background, and human observations in the form of collected data [4][23]. In doing so, DA fits a mathematical model to a particular set of data, training the model to accurately represent a system. In the end, the model's parameters are fit in such a way that it most accurately reflects the population physiology.

This push for the use of DA in bioinformatics proves to be incredibly significant and impactful in the lives of individuals with T2D. In fact, the

results of their study are used in an app called Glucoracle. Specifically, Glucoracle uses the personalized DA algorithm that is proposed in [4] to predict an individuals blood sugar levels based on the food that they record as eaten. Users can upload blood measurements from finger prick tests as well as an estimate of their meal's nutritional content. In turn, the algorithm from [4] uses this data to forecast the user's glucose levels and returns this to the user as a post-meal prediction [9]. Consequently, Glucoracle serves as an easily accessible tool for blood sugar monitoring for type 2 diabetic individuals.

In the study from [3], which is the earlier work of [4], the researchers apply DA to a dynamical model of T2D using a dataset collected from the Mobile Access to Health Information (MAHI) [3]. For clarification, MAHI is a phone application that helps individuals with diabetes track their meals and glucose levels. As a result, the dataset consists of various glucose levels as well as the nutritional values from meal intakes.

For more comprehensive results, [3] compares two different dynamical models that represent the physiological dynamics of glucose and insulin: a simple ultradian model and a meal simulation model. The simple ultradian model represents simple physiologic mechanics and is taken from [6], an earlier study by David J Albers, George Hripcsak, and Michael Schmidt. This model is based on [21], which is a 1991 study by Jeppe Sturis, Kenneth S. Polonsky, Erik Mosekilde, and Eve Van Cauter. In **chapter 1**, I explain this model in more detail. The meal simulation model is taken from [13], a study by Chiara Dalla Man, Robert A. Rizza, and Claudio Cobelli. This model includes more digestive mechanics than the simple ultradian model, including both physiologic (living system-related) and pathophysiologic (disease-related) dynamics.

Additionally, the work in [3] produces two different implementations of DA algorithms to forecast glucose levels for the simple ultradian model: dual unscented Kalman filter and Metropolis-Hastings-within-Gibbs Markov chain Monte Carlo method [4]. For brevity's sake, I will only briefly explain these two algorithms. However, a more detailed explanation of the unscented Kalman filter algorithm can be found in **chapter 6**. The dual unscented Kalman filter computes parameters in real time as data arrives. As a brief explanation, it uses the unscented transformation, which is explained more in **chapter 6**, to choose a set of sample points around the

mean. It then uses these points to recalculate a new mean and covariance that is updated to accurately represent the data set. On the other hand, the Metropolis-Hastings-within-Gibbs Markov chain Monte Carlo method computes parameters on the whole data set in retrospect [4]. Essentially, it takes random samples that have reasonably high contribution to the expected value and uses them to determine parameter values.

In order to compare the two different dynamical models in their study, the researchers apply the dual unscented Kalman filter on both the simple ultradian model and the meal simulation model [3]. One particular assumption that they had to enforce in their study is positivity. To elaborate, the model in [3] assumes positive values. Consequently, if the algorithm produced negative values, the next iteration of points is generated using only the real part of the value. However, it is important to note that this can lead to under-fitting because it can cause premature parameter convergence [3].

To assess their results, they compare mean squared error and root mean squared error of the DA implementations to linear regression and gaussian processes. It is found that both the mean squared error and the root mean squared error are the lowest for the DA implementations, no matter the sample size of the data set. They also compare the DA forecasts to those made by certified diabetes educators, who are trained diabetes counselors. According to this comparison, it is found that both the dual unscented Kalman filter and the Metropolis-Hastings-within-Gibbs Markov chain Monte Carlo method can generate glucose forecasts that are similar to or of higher quality than those of the certified diabetes educators [3].

In conclusion, this study reveals that DA algorithms can improve forecasting accuracy to the order of 40 measures. The researchers from this study believe that this is a speed that is fast enough to be useful in context of diabetes self-management. Furthermore, they found that the dual filter in conjunction with the unscented Kalman filter substantially improves performance compared to no-filter and state-only filters [3].

These conclusions suggest that DA algorithms can be applied to other mathematical modeling of diabetes, including T1D. This is because the physiological model represents glucose and insulin dynamics for T2D, which consist of the same key components in T1D with slightly differing

dynamics. In other words, a similar method can be used to fit parameters for T1D physiological models with slight adjustments, which is my ultimate goal. In this paper, I document my findings of Kalman filtering and how it can be applied to various dynamical models with the intent of analyzing its performance on a biological model.

Chapter 3

Kalman Filter

As mentioned earlier, data assimilation (DA) is a regression method that uses machine learning to forecast a model's future states. There are many algorithms within the category of DA, some of which include Markov Chain Monte Carlo method, three-dimensional variation (3dVar), optimal interpolation, and Kalman filters. This paper focuses on Kalman filters, particularly the extended Kalman filter (EKF) and the unscented Kalman Filter (UKF).

The Kalman filter (KF) algorithm is named after its primary developer, Rudolf Kalman, who first thought of the algorithm in 1958 and later published papers about it in 1960 and 1961 [19]. It is a recursive algorithm used to estimate states and parameters of dynamical models [12]. The models on which KF can be applied are often systems that can be rewritten in linear state space format, which is explained in the next section and shown later in **Equation 3.8**.

The KF algorithm functions by first taking in data that can have some error, uncertainty, or noise. Then, it filters this data to reduce the uncertainty or noise as much as possible [19]. We can think of it in the sense of how we think of general filters. **Figure 3.1** is a visual that represents the process of the KF algorithm. Through this, we see that measured parameters as well as a set of differential equations are passed into the filter. Then, the input goes through the prediction step, where process noise is passed through as additional input. After the prediction step, the data goes to the correction step, where there is supplementary input of measurement noise and additional data. From the correction step, we get an estimated output.

This estimated output as well as the additional data mentioned before goes back to the correction step, and the cycle repeats from there. In the end, the KF algorithm outputs values that represent the state of the data.

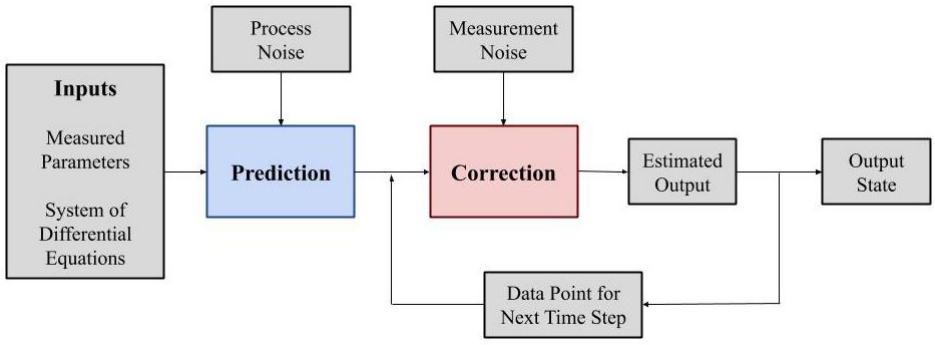

Figure 3.1 Flow chart of Kalman filter algorithm. Begins with the input of measured parameters and a set of DEs, which gets passed through the prediction step along with process noise. Then, it goes through the correction step as well as supplementary input of measurement noise and additional data, which produces an estimated output. This continues to loop through the correction step until the algorithm goes through all the time steps, where it outputs data states.

There are two main parts to the KF algorithm: the prediction and the correction. In the first part of the algorithm, it estimates, or predicts, the state of the dynamical system. In the second part, the state is corrected using known observations such that it minimizes the estimator's error covariance [12]. Consequently, KF is a form of DA that is framed through Bayesian inference [4].

Bayesian inference uses prior knowledge and the likelihood function to characterize the posterior distribution, or the distribution of the observed data. The likelihood function describes how current data and parameters map to future states and parameters [4]. In the case of the KF algorithm, the likelihood function is the state transition matrix (or state system matrix), which is often mathematically represented by F. The state transition matrix is used in both the prediction and correction steps of the KF algorithm,

which is explained in more detail later in this chapter.

3.1 Discretization of Continuous Systems

Before implementing the KF algorithm, we must first discretize the model by rewriting it in state space form, which is a set of coupled first order differential equations. Recall the difference between a discrete system and a continuous system is that a discrete system considers time as countable steps where the state variable(s) change only at each discrete step in time. On the other hand, a continuous system considers time to be fluid so the state variable(s) change continuously over time. Therefore, a discrete system would generally be written as

$$x(t_{k+1}) = x(t_k) + \Delta x(t), \tag{3.1}$$

where $t \in \mathbb{Z}$ or $t \in \mathbb{Z}^+$ and $\Delta x(t)$ is the change in x from one time step to the next. In contrast, continuous system would be written as

$$\dot{x}(t) = \frac{dx(t)}{dt}, \tag{3.2}$$

where $t \in \mathbb{R}$ or $t \in \mathbb{R}^+$. We can see how the discrete form is related to the continuous form if we manipulate **Equation 3.1** a little using Δt, where is the change in time from one time step to the next. Mathematically, Δt is

$$\Delta t = t_{k+1} - t_k.$$

More specifically, if we divide **Equation 3.1** by Δt and then take the limit as Δt approaches zero, we get

$$\frac{dx(t)}{dt} = \lim_{\Delta t \to 0} \frac{x(t_{k+1}) - x(t_k)}{t_{k+1} - t_k}.$$

This process is described a little later when we talk about the Euler method, starting at **Equation 3.5**. Essentially, the main take-away is that the discrete and continuous equations of systems are related through this limit relationship. Consequently, they only differ in that the discrete equation produces the actual value of x as a recurrence relation, whereas the continuous equation only describes the change in x rather than x itself. It is important to know this difference and relationship when constructing

and/or manipulating models on which you intend to apply Kalman filtering.

In general, the state space format for a continuous model is given by

$$\dot{x}(t) = F(t)x(t) + G(t)u(t) + w(t) \tag{3.3a}$$
$$y(t) = H(t)x(t) + v(t), \tag{3.3b}$$

where $\mathbf{x}(t)$ is the state vector, $\mathbf{y}(t)$ is the observation (or measurement) vector, and $\mathbf{u}(t)$ is the input vector. Additionally, $\mathbf{F}(t)$ is called the state transition matrix (or the state system matrix) and $\mathbf{H}(t)$ is called the observation matrix. The process noise, associated with states, is represented by $w(t)$ and the measurement noise, associated with observations, is represented by $v(t)$. In **System 3.3**, **Equation 3.3a** represents the system and **Equation 3.3b** represents the observations [8]. We can see that **System 3.3** is continuous because time is continuous, denoted by t.

As mentioned earlier, in order to apply Kalman filtering on a system, it must be discrete. Therefore, we must take a system in the form of **System 3.3** where time is continuous, solve it, and discretize it into time steps denoted as k. The general form of a discrete system in state space format is

$$x_k = e^{Ft}x_{k-1} + \int_{s=k-1}^{k} e^{F(t-s)}G(s)\,ds\; u_{k-1} + w_k \tag{3.4a}$$
$$y_k = H_k x_k + v_k, \tag{3.4b}$$

where $k \in \mathbb{Z}^+$ and $e^{\mathbf{F}t}$ is the matrix exponential of the state transition matrix F [8]. Note that **Equation 3.4a** is the solution to **Equation 3.3a**. Because computing a matrix exponential is computationally expensive, especially when F is not diagonalizable, oftentimes the Euler method is used to discretize systems.

Recall that the Euler method approximates the solution of a DE, say $f(x)$, at a certain point x_i. Consider the definition of a derivative for the function $f(x)$ at a point x_0, which is given by

$$f'(x_0) = \lim_{h\to 0} \frac{f(x_0 + h) - f(x_0)}{h}. \tag{3.5}$$

A graphical interpretation of this definition is that the derivative of a function is the slope of the secant line between the points on the function at

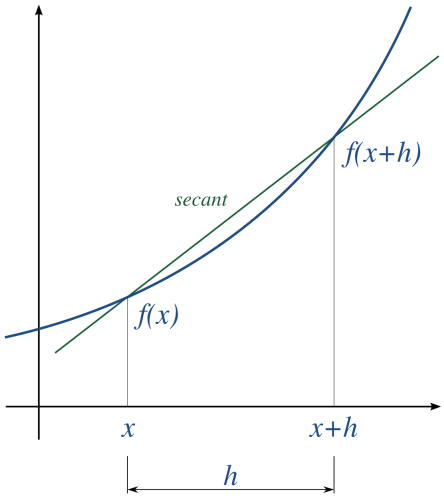

Figure 3.2 Graphical visualization of differentiation taken from [1]. Here, we see that the derivative of a function is the slope of the secant line between the points on the function at x_0 and $x + h$. The smaller that h is, the more accurate this approximation becomes.

x_0 and $x_0 + h$, or $(x_0, f(x_0))$ and $(x_0 + h, f(x_0 + h))$. As $x_0 + h$ gets closer to x_0, the slope of the secant line gets closer to the slope of the tangent line at x_0, which is equivalent to $f'(x_0)$. In other words, the smaller that h is, the more accurate this approximation becomes. A visual representation of this can be seen in **Figure 3.2**.

If we let x_0 be an arbitrary time step k, $h = \Delta x$, and $x_k = x_{k-1} + \Delta x$ in **Equation 3.5**, then we get the difference quotient of $f(x)$, which is

$$f'(x_k) \approx \frac{f(x_{k+1}) - f(x_k)}{\Delta x}. \tag{3.6}$$

We can rewrite this as a difference equation by letting F_i denote our approximation of $f(x_i)$, so at time x_k, the approximation is F_k. Then,

$$f'(x_k) = \frac{F_{k+1} - F_k}{\Delta x}$$
$$F_{k+1} = F_k + \Delta x f'(x_k).$$

In general, this means

$$F_{i+1} = F_i + \Delta x f'(x_i). \tag{3.7}$$

Although the Euler method is simple and direct, it is numerically unstable and not an accurate approximation for large time steps. As mentioned earlier, the smaller the time step Δx, the more accurate the approximation. However, using smaller time steps also requires a larger number of discretized time steps, which becomes computationally expensive. Additionally, the Euler method does not work as well for implicit DE as it does for explicit DE. Therefore, if computing the matrix exponential for **Equation 3.4** is not hard, then it is best to avoid using Euler method.

Let's try going through an example to help solidify our understanding of discretizing systems. This example is taken from Example 1 of Chapter 2 in [8]. Consider the general second order DE for Newton's 2nd law of motion:

$$m\ddot{x}(t) = u(t),$$

where m represents the mass of the moving body, $x(t)$ represents displacement (or position), and $u(t)$ represents the input function. Since $x(t)$ represents displacement, it must mean that $\ddot{x}(t)$ represents velocity and $\dot{x}(t)$ represents acceleration. Assuming that the position $x(t)$ is measured free of errors at discrete time steps, then we can write the system as

$$\begin{aligned} m\ddot{x}(t) &= u(t) \\ y(t) &= x(t). \end{aligned}$$

For this continuous system, the two state variables are $x_1(t) = x(t)$ and $x_2(t) = \dot{x}(t)$ and the measurement variable is $y(t)$. Then, it is true that

$$\begin{aligned} \dot{x_1}(t) &= \dot{x}(t) = x_2(t) \\ \dot{x_2}(t) &= \ddot{x}(t) = \frac{u(t)}{m} \\ y(t) &= x_1(t). \end{aligned}$$

The complete model can be written in matrix form:

$$\begin{aligned} \dot{x}(t) &= \begin{bmatrix} \dot{x_1}(t) \\ \dot{x_2}(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ \frac{u(t)}{m} \end{bmatrix} \\ y(t) &= x_1(t). \end{aligned}$$

Now, in vector-matrix form, the model is

$$\begin{bmatrix} \dot{x_1}(t) \\ \dot{x_2}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u(t)$$
$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}.$$

With the system in vector-matrix form, the state vector is $\mathbf{x} = [x_1 \quad x_2]^T$. Now, we have discretized the system into state space form, similar to **System 3.3** but without process noise $w(t)$ and measurement noise $v(t)$. For this system, we see that

$$F = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \qquad G = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}, \qquad H = \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

In this paper, in order to implement Kalman filtering, you do not need to manipulate the system further because there are often built-in functions to differentiate the system and solve for the solution. For practice, this example will take one step further and determine the solution to write the system in the form of **System 3.4**.

In order to solve the system and write it in the form of **System 3.4**, we must first determine the matrix exponential of F as well as $\int_{s=k-1}^{k} e^{\mathbf{F}(t-s)}\mathbf{G}(s)\, ds$. Recall that matrix exponential calculation for a diagonalizable matrix simply uses the eigenvalues. In this case, the eigenvalues of F are repeated with both equaling $\lambda_1 = 0$. Thus, the matrix exponential of F is

$$e^{Ft} = \begin{bmatrix} e^{\lambda_1 t} & te^{\lambda_1 t} \\ 0 & e^{\lambda_1 t} \end{bmatrix} = \begin{bmatrix} e^{0i} & te^{0i} \\ 0 & e^{0i} \end{bmatrix} = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix}.$$

Using this, we can calculate $\int_{s=k-1}^{k} e^{\mathbf{F}(t-s)}\mathbf{G}(s)\, ds$. In this case, we consider our time step to be the first time step where the size of each time step is t. Consequently, we calculate this integral where $k - 1 = 0$ and $k = t$.

$$\int_{s=0}^{t} e^{\mathbf{F}(t-s)}\mathbf{G}(s)\, ds = \int_{s=0}^{k} \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} ds = \int_{s=0}^{t} \begin{bmatrix} \frac{s}{m} \\ \frac{1}{m} \end{bmatrix} ds = \begin{bmatrix} \frac{t^2}{2m} \\ \frac{t}{m} \end{bmatrix}.$$

Thus, we have the discretized state space model with state vector x and observation vector y, which is

$$x_k = \begin{bmatrix} 1 & t \\ 0 & 1 \end{bmatrix} x_{k-1} + \begin{bmatrix} \frac{t^2}{2m} \\ \frac{t}{m} \end{bmatrix} u_k$$
$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} x_k.$$

Variable	Description	Dimension
x	state vector	$n \times 1$
y	observation vector	$n_y \times 1$
u	input vector	$n_u \times 1$
w	process noise vector	$n \times 1$
v	measurement noise vector	$n_y \times 1$
F	state transition matrix	$n \times n$
G	input system matrix	$n \times n_u$
H	observation matrix	$n_y \times n$

Table 3.1 Discrete time system variables for Kalman filter.

With a better understanding of how to discretize systems and rewrite them in state space form, we can now discuss the Kalman filter algorithm.

3.2 Kalman Filter Algorithm

From the previous section, we know that Kalman filtering is applied to models in discrete linear state space format. These systems would then appear as

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{3.8a}$$

$$y_k = H_k x_k + v_k, \tag{3.8b}$$

where the variables are defined in **Table 3.1** [19].

Now, we can begin the KF algorithm, which consists of two parts: the prediction and the correction.

3.2.1 Prediction Step

The prediction step calculates two things: (a) the predicted state vector of the initial state $\hat{x}_0$ and (b) the covariance matrix P_0 of the initial predicted state [19][12]. In general, the predicted state vector $\hat{x}$ represents the desired result. The values in $\hat{x}$ will be passed through and estimated by the filter. In general, $\hat{x}$ is constructed as the following:

$$\hat{x} = F_{k-1}\hat{x}_{k-1} + w_{k-1}, \tag{3.9}$$

where F_{k-1} is the state transition matrix for the previous time step, $\hat{x}_{k-1}$ is the state vector from the previous time step, and w_{k-1} is the process noise vector from the previous time step. The state transition matrix F contains the coefficients of the state terms in the dynamical model [19]. In other words, it transforms any initial state $x(k_0)$ to its corresponding state $x(k)$ at time step k[12]. Consequently, if $\hat{x}$ is an $n\times1$ vector, then F is an $n\times n$ matrix.

For the initial state, the predicted state vector $\hat{x}$ is equivalent to the initial state vector $\hat{x}_0$. This implies that in **Equation 3.9**,

$$F_0 = I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix},$$

where I is the identity matrix. In other words, the initial predicted state vector $\hat{x}_0$ consists of educated guesses that are considered the initial values. Therefore, the initial state transition matrix F_0 does not need to apply any transformation, so it is equivalent to the identity matrix.

After determining the initial predicted state vector, we must determine its corresponding initial covariance matrix P_0. We can determine the initial state error covariance matrix by simply using the values in the predicted state vector $\hat{x}$ and the statistical definition of covariance, which is

$$P = \mathbb{E}[(\hat{x} - \mathbb{E}[\hat{x}])(\hat{x} - \mathbb{E}[\hat{x}])^T], \tag{3.10}$$

where $\mathbb{E}[\hat{x}]$ is the expected value of $\hat{x}$. Recall from probability that the expected value of a vector is defined as

$$\mathbb{E}[\hat{x}] = \sum_{i=1}^{n} x_i p_i = x_1 p_1 + x_2 p_2 + \ldots + x_n p_n, \tag{3.11}$$

where $\hat{x} = \{x_1, x_2, \ldots, x_n\}$ occurring with probabilities $\{p_1, p_2, \ldots, p_n\}$ respectively. Consequently, if $\hat{x}$ is an $n \times 1$ vector, then P is an $n \times n$ matrix, which are the same dimensions as F. Thus, the initial state covariance matrix P_0 is

$$P_0 = \begin{bmatrix} var(x_1) & \cdots & cov(x_1, x_n) \\ \vdots & \ddots & \vdots \\ cov(x_n, x_1) & \cdots & var(x_n) \end{bmatrix}.$$

When the state variables in the state vector are uncorrelated, $\text{cov}(x_a, x_b) = 0$, where $a, b \in \mathbb{Z}$. Because the state variables in the state vector $\hat{x}$ are Gaussian, meaning normally distributed, if they are uncorrelated then they are independent. Consequently, if the state variables are independent, then the covariance matrix P_0 is a diagonal matrix where the diagonal consists of the variances of state variables:

$$P_0 = \begin{pmatrix} var(x_1) & & 0 \\ & \ddots & \\ 0 & & var(x_n) \end{pmatrix}.$$

3.2.2 Correction Step

Now, with our initial state vector and initial state covariance matrix, we can move on to the correction step of the KF algorithm. Similar to the prediction step, the correction step calculates the predicted state vector and its corresponding covariance matrix for the next time step k. The correction step differs from the prediction step in that it incorporates observation data that will be used to adjust the prediction state and covariance matrix. For the state vector, we can use a similar method as in the prediction step, except incorporate collected data.

$$\hat{x}_{k|k-1} = F_{k-1}\hat{x}_{k-1} + G_{k-1}\hat{u}_{k-1}, \tag{3.12}$$

where $\hat{x}_{k|k-1}$ is the state vector at time step k that we are calculating, $\hat{x}_{k-1}$ is the estimated state vector from the previous time step, $\hat{u}_{k-1}$ is the input vector that represents the collected data at the previous time step, F_{k-1} is the same state transition matrix from **Equation 3.9**, and G_{k-1} is the system input matrix from the previous time step. The system input matrix consists of coefficients of the input terms in the state dynamics. In other words, the system input matrix G consists of values that are taken from the dynamical model. These values are the coefficients in the dynamical model, specifically the coefficients of terms that are the input for the filter. Note that if $\hat{x}_{k-1}$ and $\hat{u}_{k-1}$ are vectors with length n, then $\hat{x}_{k|k-1}$ is also a vector with length n.

In order to determine the covariance matrix P, we use the following equation:

$$P_k = F_{k-1}P_{k-1}F_{k-1}^T + Q_{k-1}, \tag{3.13}$$

where P_{k-1} is the estimated state error covariance matrix for the previous time step, F_{k-1} is the state transition matrix for the previous time step, and Q_{k-1} is the process noise covariance matrix associated with the process noise vector w_{k-1}. From **Equation 3.13**, we see that P_k has the same dimensions as P_{k-1}.

With our predicted state vector $\hat{x}_k$ and predicted state covariance matrix P_k for time step k, we can now calculate the corresponding Kalman gain matrix K_k. The Kalman gain matrix describes how much you want to change your estimate by a given measurement. Mathematically, it is expressed as

$$K_k = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}, \tag{3.14}$$

where P_k is the state covariance matrix defined in **Equation 3.13**, H_k is the observation matrix at time step k, and R_k is the measurement noise covariance at time step k. The observation matrix H_k consists of the collected data values at time step k. It is important to note the difference between R and Q. R represents the measurement noise whereas Q represents the process noise. Measurement noise represents the idea that data is collected in noisy environments rather than completely controlled environments, so we must account for the noise in the environment. On the other hand, process noise represents the idea that the state of the system changes in a way that we do not know exactly, so we must account for the noise in the system.

With the Kalman gain matrix, we can now update the prediction by the appropriate amount:

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k(y_k - H_k \hat{x}_{k|k-1}), \tag{3.15}$$

where y_k is the observation (or measurement) of the output and $H_k \hat{x}_{k|k-1}$ is the predicted output (sometimes referred to as $\hat{y}_{k|k-1}$). Recall from **Equation 3.8b**, the observation (or measurement) of the output y_k at time step k is based on the previous time step $k-1$, defining it as

$$y_k = H_k \hat{x}_{k|k-1} + v_k,$$

where v_k is the "innovation." The "innovation" is also known as the measurement noise vector, which is associated with the measurement noise covariance matrix R_k. In general, the measurement noise v is normally distributed with mean zero and variance R, so $v \sim \mathcal{N}(0, R)$. In this case, since the

process noise for a given time step v_k is a vector, the variance R for that same time step is a covariance matrix R_k called the process noise covariance matrix.

Looking back at **Equation 3.15**, we can analyze how the measurement output y_k differs from the predicted output $\hat{y}_{k|k-1}$:

$$\begin{aligned} y_k - \hat{y}_{k|k-1} &= H_k\hat{x}_{k|k-1} + v_k - H_k\hat{x}_{k|k-1} \\ &= v_k. \end{aligned}$$

Therefore, the difference between the measurement output and the predicted output in **Equation 3.15** is v_k, or the "innovation." As was just mentioned, the "innovation" is also known as the measurement noise. Since the measurement output and the predicted output only differ by the measurement noise, the corrected state vector $\hat{x}_k$ in **Equation 3.15** is updated by an amount that is equivalent to scaling the measurement noise by the Kalman gain matrix K_k.

In a similar way, we also update the state error covariance matrix by

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

where I is the identity matrix.

3.2.3 Example

To help solidify the Kalman Filter algorithm, it helps to see an example, which can be found in [19]. In section 4, "A Linear Kalman Filtering Example," there is a thorough explanation on how to apply the Kalman Filter to a dynamical system for a free falling object.

3.3 Types of Kalman Filters

For linear systems, Bayesian inference methods, such as KF, work very well. However, when working with nonlinear systems, the basic KF algorithm fails to properly predict states. Thankfully, there are other prediction algorithms that can handle nonlinear systems, including derivations of KF. The most commonly used type of KF that can take in nonlinear systems is the extended Kalman filter (EKF).

The EKF algorithm determines the state distribution by analytically propagating a Gaussian random variable through the first-order linearization of the nonlinear system [22]. In other words, EKF takes a nonlinear system and approximates it to the first-order, which results in a first-order linearization. Then, it passes a normally-distributed random variable through the linearization to determine the state distribution. Because it only calculates the state distribution up to the first-order, the EKF algorithm provides only an approximation for the state estimation [22]. As a result, EKF cannot be applied to problems with high dimensional data and state spaces [22]. Otherwise, it can produce large errors in the true posterior mean and covariance of the transformed data [22]. In the next chapter, EKF is explained in more detail.

An alternative that does not produce large errors for nonlinear systems is another derivative of KF called the Unscented Kalman filter (UKF). The UKF algorithm uses a deterministic sampling approach rather than an approximation approach, which prevents the large errors that the EKF sometimes introduces. In **chapter 6**, UKF is explained in more detail.

Chapter 4

Extended Kalman Filter

The extended Kalman filter (EKF) is a type of Kalman filter that can be used to approximate states as well as parameters of nonlinear systems. The EKF algorithm was discovered by Stanley F. Schmidt shortly after Rudolf Kalman presented his Kalman filtering results in 1961. After hearing of the Kalman filter, Schdmit applied it to the upcoming Apollo program for exploration of the mankind on the moon, using KF to solve the space navigation problem. In doing so, Schmidt invented the EKF algorithm [12].

As mentioned in **chapter 3**, EKF determines the state distribution by passing a Gaussian random variable through a first-order linearization of a nonlinear system. Because EKF linearizes the system about the estimated state, the system must be represented by continuously differentiable functions. For linear systems, this could be computed efficiently and still produce fairly accurate results. However, for nonlinear systems, the EKF can become computationally time-consuming and produce large errors [12].

4.1 Extended Kalman Filter Algorithm

Like the KF algorithm, the extended Kalman filter is applied to systems in linear state space format, which is defined in **System 4.1**, where the variables are defined in **Table 4.1** [19].

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{4.1a}$$

$$y_k = H_k x_k + v_k. \tag{4.1b}$$

Variable	Description	Dimension
x	state vector	$n \times 1$
y	observation vector	$n_y \times 1$
u	input vector	$n_u \times 1$
w	process noise vector	$n \times 1$
v	measurement noise vector	$n_y \times 1$
F	state transition matrix	$n \times n$
G	input system matrix	$n \times n_u$
H	observation matrix	$n_y \times n$

Table 4.1 Discrete time system variables for extended Kalman filter.

Here, **Equation 4.1a** represents the system with state(s) x and **Equation 4.1b** represents the observation(s).

As a derivative of KF, EKF also has two steps: the prediction step and the correction step. Therefore, we can visualize EKF using the same diagram that we did for the KF algorithm in **Figure 3.1**. EKF differs from the general KF algorithm in how it linearizes the dynamical system. Unlike the KF algorithm, EKF utilizes Jacobian matrices to linearly approximate the system [8]. These Jacobian matrices are later used in the state transition matrix and the observation matrix.

Recall that a Jacobian matrix is the matrix of all first-order partial derivatives. Mathematically, if you were given an n-dimensional function $\mathbf{f}$, then the Jacobian of $\mathbf{f}$ is

$$\mathbf{J_f} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}. \tag{4.2}$$

Like the KF algorithm, we must first discretize the system of DEs, writing them in state-space form before implementing the EKF algorithm. Recall that we can do this by using the method explained in **chapter 3** and through **System 3.3**. For the system, we are often given a model that consists of the states x and the corresponding observations (or measurements) y. With a state-space system of DEs consisting of both the state equation and the

measurement equation, we can move on to the prediction step of the EKF algorithm.

4.1.1 Prediction Step

We can begin the EKF algorithm with its first step, the prediction step. The prediction step calculates two things: (a) the predicted state vector of the initial state $\hat{x}_0$ and (b) the covariance matrix P_0 of the initial predicted state [19][12]. In general, the predicted state vector represents the desired result. The values in $\hat{x}$ will be passed through and estimated by the filter. In general, $\hat{x}$ is constructed as the following:

$$\hat{x} = F_{k-1}\hat{x}_{k-1} + w_{k-1}, \tag{4.3}$$

where F_{k-1} is the state transition matrix for the previous time step, $\hat{x}_{k-1}$ is the state vector from the previous time step, and w_{k-1} is the process noise vector from the previous time step. The process noise w is normally distributed with mean zero and variance Q, so $w \sim \mathcal{N}(0, Q)$. In this case, since the process noise for the previous time step w_{k-1} is a vector, the variance Q for the previous time step is a covariance matrix Q_{k-1} called the process noise covariance matrix.

The state transition matrix F is the Jacobian of the state equations in the dynamical model [12][8]. It transforms any initial state x_{k_0} to its corresponding state x_k at time step k [12]. Consequently, if $\hat{x}$ is an $n \times 1$ vector, then F is an $n \times n$ matrix. The state transition matrix can be calculated using **Equation 4.2**.

For the initial state, the predicted state vector $\hat{x}$ is equivalent to the initial state vector $\hat{x}_0$. This implies that in **Equation 4.3**,

$$F_0 = I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix},$$

where I is the identity matrix. In other words, the predicted state vector $\hat{x}$ consists of educated guesses that are considered the initial values. Therefore, the initial state transition matrix F_0 does not need to apply any transformation, so it can be the identity matrix.

After determining the initial predicted state vector, we must determine its corresponding initial covariance matrix P_0. We can determine the initial state error covariance matrix by simply using the values in the predicted state vector $\hat{x}$ and the statistical definition of covariance, which is

$$P = \mathbb{E}[(\hat{x} - \mathbb{E}[\hat{x}])(\hat{x} - \mathbb{E}[\hat{x}])^T], \tag{4.4}$$

where $\mathbb{E}[\hat{x}]$ is the expected value of $\hat{x}$. Recall from probability that the expected value of a vector is defined as

$$\mathbb{E}[\hat{x}] = \sum_{i=1}^{n} x_i p_i = x_1 p_1 + x_2 p_2 + \ldots + x_n p_n, \tag{4.5}$$

where $\hat{x} = \{x_1, x_2, \ldots, x_n\}$ occurring with probabilities $\{p_1, p_2, \ldots, p_n\}$ respectively. Consequently, if $\hat{x}$ is an $n \times 1$ vector, then P is an $n \times n$ matrix, which are the same dimensions as F. Thus, the initial state covariance matrix P_0 is

$$P_0 = \begin{bmatrix} var(x_1) & \cdots & cov(x_1, x_n) \\ \vdots & \ddots & \vdots \\ cov(x_n, x_1) & \cdots & var(x_n) \end{bmatrix}.$$

When the state variables in the state vector are uncorrelated, $\text{cov}(x_a, x_b) = 0$, where $a, b \in \mathbb{Z}$. Because the state variables in the state vector $\hat{x}$ are Gaussian, meaning normally distributed, if they are uncorrelated then they are independent. Consequently, if the state variables are independent, then the covariance matrix P_0 is a diagonal matrix where the diagonal consists of the variances of state variables:

$$P_0 = \begin{pmatrix} var(x_1) & & 0 \\ & \ddots & \\ 0 & & var(x_n) \end{pmatrix}.$$

4.1.2 Correction Step

Now, with our initial state vector and initial state covariance matrix, we can move on to the correction step of the EKF algorithm. Similar to the prediction step, the correction step calculates the predicted state vector and its corresponding covariance matrix for the next time step k. The correction step differs from the prediction step in that it incorporates observation data

that will be used to adjust the prediction state and covariance matrix. For the state vector, we can use a similar method as in the prediction step, except incorporate collected data.

$$\hat{x}_{k|k-1} = F_{k-1}\hat{x}_{k-1} + G_{k-1}\hat{u}_{k-1}, \tag{4.6}$$

where $\hat{x}_{k|k-1}$ is the state vector at time step k that we are calculating, $\hat{x}_{k-1}$ is the estimated state vector from the previous time step, $\hat{u}_{k-1}$ is the input vector that represents the collected data at the previous time step, F_{k-1} is the same state transition matrix from **Equation 4.3**, and G_{k-1} is the system input matrix from the previous time step. Like KF, the system input matrix consists of coefficients of the input terms in the state dynamics. Note that if $\hat{x}_{k-1}$ and $\hat{u}_{k-1}$ are vectors with length n, then $\hat{x}_{k|k-1}$ is also a vector with length n.

In order to determine the covariance matrix P, we use the following equation:

$$P_k = F_{k-1}P_{k-1}F_{k-1}^T + Q_{k-1}, \tag{4.7}$$

where P_{k-1} is the estimated state error covariance matrix for the previous time step, F_{k-1} is the state transition matrix for the previous time step, and Q_{k-1} is the process noise covariance matrix associated with the process noise vector w_{k-1}. Like **Equation 4.6**, the state transition matrix F_{k-1} is the same state transition matrix from **Equation 4.3**. In other words, F_{k-1} is the Jacobian of the state equations in the dynamical model for time step $k-1$, which can be computed using the method outlined in **Equation 4.2**.

For linear systems, this is generally the same, so the implementation can be made more efficient by pre-computing the Jacobian matrix F. However, for nonlinear systems, the partial derivatives that compose F are functions of the state, so they change at every time step and, thus, must be computed with every time step [12].

With our predicted state vector $\hat{x}_k$ and predicted state covariance matrix P_k for time step k, we can now calculate the corresponding Kalman gain matrix K_k. The Kalman gain matrix describes how much you want to change your estimate by a given measurement. Mathematically, it is expressed as

$$K_k = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1}, \tag{4.8}$$

where P_k is the state covariance matrix defined in **Equation 4.7**, H_k is the observation matrix at time step k, and R_k is the measurement noise covariance at time step k.

The observation matrix H_k is the Jacobian of the measurement equations at time step k. It transforms any initial measurement y_{k_0}, which is usually written in terms of x, to its corresponding measurement y_k at time step k [8]. Consequently, if $\hat{y}$ is an $n \times 1$ vector, then H is an $n \times n$ matrix. The observation matrix can be calculated using **Equation 4.2**.

It is important to note the difference between R and Q. This was mentioned in **chapter 3**, but I will reiterate for emphasis. R represents the measurement noise whereas Q represents the process noise. Measurement noise represents the idea that data is collected in noisy environments rather than completely controlled environments, so we must account for the noise in the environment. On the other hand, process noise represents the idea that the state of the system changes in a way that we do not know exactly, so we must account for the noise in the system.

With the Kalman gain matrix, we can now update the prediction by the appropriate amount:

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k(y_k - H_k\hat{x}_{k|k-1}), \tag{4.9}$$

where y_k is the observation (or measurement) of the output and $H_k\hat{x}_{k|k-1}$ is the predicted output (sometimes referred to as $\hat{y}_{k|k-1}$). Recall from **Equation 4.1b**, the observation (or measurement) of the output y_k at time step k is based on the previous time step $k-1$, defining it as

$$y_k = H_k\hat{x}_{k|k-1} + v_k,$$

where v_k is the "innovation." The "innovation" is also known as the measurement noise vector, which is associated with the measurement noise covariance matrix R_k. In general, the measurement noise v is normally distributed with mean zero and variance R, so $v \sim \mathcal{N}(0, R)$. In this case, since the process noise for a given time step v_k is a vector, the variance R for that same time step is a covariance matrix R_k called the process noise covariance matrix.

Looking back at **Equation 4.9**, we can analyze how the measurement

output y_k differs from the predicted output $\hat{y}_{k|k-1}$:

$$\begin{aligned} y_k - \hat{y}_{k|k-1} &= H_k\hat{x}_{k|k-1} + v_k - H_k\hat{x}_{k|k-1} \\ &= v_k. \end{aligned}$$

Therefore, the difference between the measurement output and the predicted output in **Equation 4.9** is v_k, or the "innovation." As was just mentioned, the "innovation" is also known as the measurement noise. Since the measurement output and the predicted output only differ by the measurement noise, the corrected state vector $\hat{x}_k$ in **Equation 4.9** is updated by an amount that is equivalent to scaling the measurement noise by the Kalman gain matrix K_k.

In a similar way, we also update the state error covariance matrix by

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

where I is the identity matrix.

To help solidify the extended Kalman Filter algorithm, it is important to go through an example. In the following chapter, I provide examples of how the EKF algorithm estimates the states of three different systems along with the implementation for these examples.

Chapter 5

EKF State Estimation

In order to implement the EKF algorithm to estimate state values, I use source code from [8] as a foundation as well as the built-in MATLAB `extendedKalmanFilter` function. In "Chapter 11.4: Examples of non-linear fitting" of [8], there is source code for implementing the EKF algorithm on an example originally proposed by [11], which is a nonlinear system. I use this provided code as a basis for all of my implementations of EKF, except for one.

In this chapter, I go through three different implementations. The first implementation of EKF estimates the state of a linear system, which is provided in [8]. The second implementation of EKF also estimates a state variable but on a nonlinear system. For this nonlinear system, I use the provided code from [8]. The last implementation estimates the states of another nonlinear system, but this last one represents a type 2 diabetes physiological model consisting of six differential equations.

The code from [8] consists of two functions: `sim_gss` and `ekf_gss`. The first function, `sim_gss`, simulates Gaussian data for the system, which consists of state function(s) and measurement function(s). As input, it takes the variance of the process noise `Q`, the variance of the measurement noise `R`, the mean `m0`, the variance `P0`, and the number of values produced `N`. For any system, `Q` and `R` are single-valued inputs. The dimensions of `m0` and `P0` depend on the system. In general, if the system consists of n equations, then `m0` is a vector of length n and `P0` is an $n \times n$ matrix. The results of `sim_gss` represent true values for the state(s) as well as the measurement(s).

The second function, `ekf_gss`, implements the EKF algorithm and pro-

duces estimates for the given system. It takes in the same inputs as `sim_gss` with an additional two inputs, which are the outputs of `sim_gss`. The two inputs are the simulated data for the state variable(s) `x` and the simulated data for the measurement(s) `z`. The dimensions of `x` and `z` depend on the system. If the system has a state equations and b measurement equations, then `x` is a $N \times a$ matrix and `z` is a $N \times b$ matrix, where N (as mentioned before) represents the number of values produced. The results of `ekf_gss` are the EKF estimates for the state(s) and the measurement(s).

In addition to the code from [8], I also use MATLAB's pre-existing `extendedKalmanFilter` function to implement EKF on the biological model. The MATLAB `extendedKalmanFilter` function takes as input a function `StateTransitionFcn`, a function `MeasurementFcn`, a vector `InitialState`, a vector or matrix `MeasurementNoise`, and a square matrix `ProcessNoise`.

`StateTransitionFcn` calculates the state vector of the system at time step k given the state vector at time step $k-1$. `MeasurementFcn` calculates the output measurement vector of the system at time step k given the sate vector at time step k. `InitialState` represents the initial state values based on the user's knowledge of the system. Therefore, it is a vector with length n, where n represents the number of states in the system. `MeasurementNoise` represents the measurement noise covariance. It is a scalar when `HasAdditiveMeasurementNoise` is set to true and a matrix when `HasAdditiveMeasurementNoise` is set to false. If it is a matrix, then its dimensions are $v \times v$, where v represents the number of measurement noise terms. `ProcessNoise` represents the process noise covariance, and it is a square matrix with dimensions $w \times w$, where w represents the number of process noise terms.

Using the pre-existing MATLAB function `extendedKalmanFilter`, I estimate states for a biological model, specifically the T2D model discussed in **chapter 1**. I choose to use this method rather than the method proposed in [8] because the biological model is complex, which would make it difficult to implement using the method in [8]. Luckily, it can be conveniently implemented using the MATLAB function `extendedKalmanFilter`.

The code corresponding to this chapter can be found in the following GitHub repository: https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/

master/Extended_KFs/. For the linear system, the code is in the subfolder `Bolviken_Ex7`. The code corresponding to the nonlinear system is in the subfolder `Bolviken_Ex5`. The type 2 diabetes biological model's code is located in the subfolder `Albers`.

5.1 Linear System

Consider the following linear system proposed in [8]:

$$x(k) = ax(k-1) + w(k-1) \tag{5.1a}$$
$$z(k) = x(k) + v(k), \tag{5.1b}$$

where a is a constant equal to 1.

The simulated data function `sim_gss` simply rewrites the system so that it takes in values and passes it through the system to produce simulated true values for both the states x and the measurements z. The input values were guessed to be the following: `Q` = 0.01, `R` = 0.02, N = 100, `m0` = 0, `P0` = 25, and `N` = 50.

Because the initial values and noise were produced randomly, every time `sim_gss` is run a different dataset is produced, even if the input values are the same. Consequently, for reproducibility purposes, I include the dataset I used to run the next function in the GitHub repository mentioned earlier.

The `ekf_gss` implements EKF on the system. The first two inputs, `x` and `z`, were the outputs of `sim_gss` and represent what the algorithm is measuring. These results can be found in the GitHub repository mentioned earlier. The other input values are the same as the inputs for `sim_gss`. The resulting EKF estimation for the states x and z are depicted in **Figure 5.1**.

In **Figure 5.1**, there are two plots, the left for the `x` state and the right for the `z` state. For both plots, the true values (simulated data) for each state are in red while the EKF estimates are in blue. By looking at these plots, the EKF algorithm seems to perform fairly well in estimating states for this system. The EKF estimates seem to follow closely to the true values. Consequently, this example suggests that EKF performs well when estimating states of linear systems. This leads us to wonder the following: does EKF perform equally as well when estimating states of nonlinear systems?

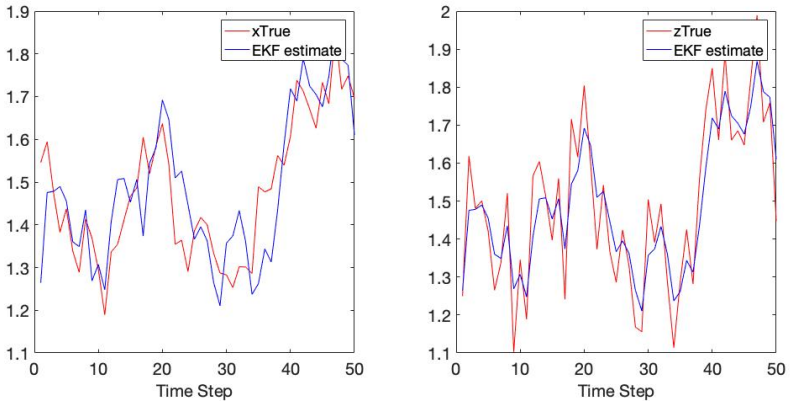

Figure 5.1 Results of EKF implementation of **System 5.1**. The true solutions for states $x(k)$ and $z(k)$ are depicted by the red curves, and the EKF estimates by the blue curves. In this example, EKF produces a reasonable approximation for the true solution (errors and simulated data not shown). Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Bolviken_Ex7/

5.2 Nonlinear System

Consider the following nonlinear system proposed in [11]:

$$x(k) = 0.5x(k-1) + \frac{25x(k-1)}{1+x^2(k-1)} + 8\cos(1.2(k-1)) + w(k-1) \tag{5.2a}$$

$$z(k) = \frac{x^2(k)}{20} + v(k). \tag{5.2b}$$

Like the linear example in the previous section, the simulated data function `sim_gss` simply rewrites the system so that it takes in values and passes it through the system to produce simulated true values for both the states x and the measurements z. The input values were provided in [8]: `Q` $= 10$, `R` $= 1$, `m0` $= 0$, `P0` $= 1$, and `N` $= 50$.

The `ekf_gss` implements EKF on the system. The first two inputs, `x` and `z`, were the outputs of `sim_gss` and represent what the algorithm is measuring. These results can be found in the GitHub repository mentioned earlier. The other input values are the same as the inputs for `sim_gss`. The resulting EKF estimation for the states x and z are depicted in **Figure 5.2**.

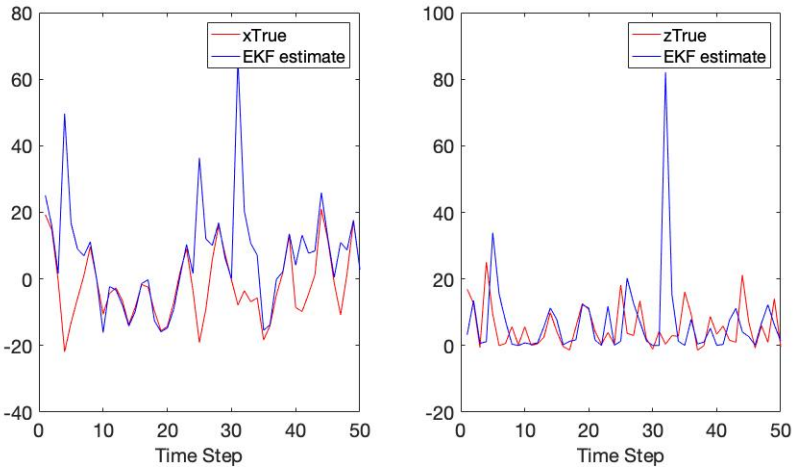

Figure 5.2 Results of EKF implementation of **System 5.2**. The true solutions for states $x(k)$ and $z(k)$ are depicted by the red curves, and the EKF estimates by the blue curves. In this example, EKF does not produce as accurate approximations for the true solution as **System 5.1** (errors and simulated data not shown). This is likely because **System 5.2** is highly nonlinear. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Bolviken_Ex5/

Figure 5.2 consists of two plots, the left for the x state and the right for the z state. For both plots, the true values (simulated data) for each state are in red while the EKF estimates are in blue. By looking at these plots, the EKF algorithm seems to have mediocre performance in estimating states for this system. The EKF estimates seem to follow the true values for the most part, but they seem to have unusual peaks for certain time steps. Consequently, this example suggests that EKF performs with moderate accuracy when estimating states of nonlinear systems.

Comparing these results to **Figure 5.1**, it seems that EKF performs better on linear states compared to nonlinear states. This was mentioned earlier in **chapter 3** and **chapter 4**. We can further investigate EKF estimation on nonlinear systems by applying EKF to the states on a highly nonlinear biological system. Given the performance of EKF on **System 5.2**, we expect the results for the next system to also perform worse than **System 5.1**, or maybe even worse given that the next system is highly nonlinear.

5.3 Type 2 Diabetes Physiologcial Model

The third system of differential equations that I implement is a nonlinear system that consisted of six ordinary differential equations (ODEs). This system of ODEs is taken from the equations David J. Albers used to model glucose/insulin through a MATLAB implementation [5]. It is supposedly the equations in the population physiology model from [6]. This model simulates glucose-insulin physiology among individuals with T2D. As mentioned in **chapter 1**, this model is based on an earlier model proposed in [21].

However, there seems to be some discrepancy between the two systems of ODEs. In particular, the two ODEs that represent the change in glucose do not seem to be the same. Interestingly, [5] seems to implement the six DEs in [21] rather than their own paper. Currently, I do not why these two systems differ and how significantly they differ. For now, we only consider the DEs from [21].

Typically, we first have to rewrite our system of DEs such that it is in state-space format. However, this glucose-insulin physiology model is already in state-space form.

The six ODEs are as follows:

$$\begin{aligned}
\frac{dI_p}{dt} &= \frac{R_m}{1 - exp(\frac{-G}{V_g C_1} + a_1)} - E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_p}{t_p} \\
\frac{dI_i}{dt} &= E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_i}{t_i} \\
\frac{dG}{dt} &= \frac{R_g}{1 + exp(\frac{0.29h_3}{V_p - 7.5})} + I_g - U_b\left(1 - exp\left(\frac{-G}{C_2 V_g}\right)\right) \\
&\quad - \frac{90}{1 + exp(-1.772 \log\left(I_i\left(\frac{1}{V_i} + \frac{1}{E t_i}\right)\right) + 7.76)} + 4 \\
\frac{dh_1}{dt} &= \frac{3(I_p - h_1)}{t_d} \\
\frac{dh_2}{dt} &= \frac{3(h_1 - h_2)}{t_d} \\
\frac{dh_3}{dt} &= \frac{3(h_2 - h_3)}{t_d}
\end{aligned}$$

The thirty different constants that appear in the system of ODEs can be found in **Appendix B**. These constants were taken from [6].

This set of six ODEs are then used to create the state vector $\hat{x}$, where the state vector is

$$\hat{x} = \begin{bmatrix} I_p \\ I_i \\ G \\ h_1 \\ h_2 \\ h_3 \end{bmatrix},$$

where I_p represents plasma insulin, I_i represents remote insulin, G represents glucose, and the last three states represent different delayed feeding cycles. By feeding cycle, we mean an individual's eating habits. Therefore, this system explores three different possible eating habits. In the system, h_1 represents the first stage linear filter feeding cycle, h_2 represents the second stage linear filter feeding cycle, and h_3 represents the third stage linear filter

feeding cycle.

In order to determine the states based on the system of ODEs, we use the MATLAB ODE-solver called `ode45`, which takes in a DE as input and returns the solution to the DE. Consequently, when applying `ode45` to the T2D model we get the state vector $\hat{x}$.

With the state vector for the glucose-insulin physiology system, we can create the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `extendedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output only the third element in the state vector, which represents glucose. By doing this, the EKF algorithm only measures glucose. This is because we mainly care about the change in glucose levels, so we choose to decrease computational cost by only measuring one state.

Additionally, we set the `InitialState` vector to be the initial values that were used in [21]. This has the initial values as

$$[200, 200, 12000, 0.1, 0.2, 0.1].$$

In other words, we set the plasma insulin to be 200 mU/min, remote insulin to be 200 mU/min, glucose to be 12000 mU/min, the first stage linear filter to be 0.1 mU/min^2, the second stage linear filter to be 0.2 mU/min^2, and the third stage linear filter to be 0.1 mU/min^2.

The MATLAB algorithm also has the capability to add an additional input to represent noise, called `MeasurementNoise` and `ProcessNoise`.

In a brief tangent, I take a moment to explain the difference between measurement noise and process noise, which are often variables incorporated in systems and models. Both are key components in accounting for noise, though they represent very different ways in which noise can enter a system. Measurement noise accounts for the uncertainty in the data measurements, whereas process noise accounts for the uncertainty in the system or model.

In this example, we let the measurement noise be additive, so

$$\hat{y}_k = \hat{x}_k + \hat{v}_k.$$

Since the measurement noise is additive, we must be sure to set `HasAdditiveMeasurementNoise` to be true. The measurement noise vector $\hat{v}_k$

has the same dimensions as the state vector $\hat{x}_k$. We set $\hat{v}_k$ to be a vector that consists of one arbitrarily-chosen value, 5. We chose this value because it adds sufficient noise to the state values for glucose, which are on the scale of a hundred. This gets input as the `MeasurementNoise`. Since we let measurement noise be additive, we must also set `HasAdditiveMeasurementNoise` in the UKF algorithm to be true.

For `ProcessNoise`, the input must be a matrix with dimensions $n \times n$, where n represents the number of states there are. In this example, the values for `ProcessNoise` were arbitrarily chosen, though intentionally chosen to be on a similar scale as the `MeasurementNoise`. As a result, the input for `ProcessNoise` is

$$\begin{bmatrix} 0.02 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.04 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}$$

Now, we have all the inputs to implement the MATLAB `extendedKalmanFilter` function. This process starts by creating simulated data measurements that incorporate some random noise that is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which was created by adding a randomly-determined shift to values that were generated using the DEs. Then, we call the `extendedKalmanFilter` function for each measured data point to update the state and covariance based on the previous estimation and the latest measured data point, which produce the "EKF estimate" values.

In **Figure 5.3**, we can see the results of applying the EKF algorithm to this T2D model. Rather than displaying all six states in the model, I only graph the results of glucose. This is because the algorithm only measures the change in glucose levels, as mentioned earlier. The graph displays the change in glucose levels in milligrams per liter over 1200 minutes. The measured data (in magenta dotted curve) is somewhat noisy while the true data (in red solid curve) is smooth. Looking at the results, it seems that the values estimated by the EKF algorithm (in blue dashed curve) are accurate at predicting the glucose levels as well as correctly smoothing the data.

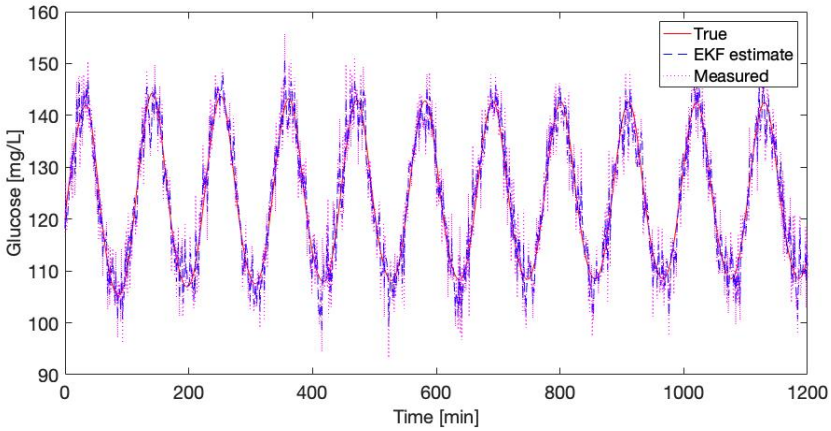

Figure 5.3 Results of EKF implementation for T2D model proposed in [6]. The true solution for the glucose state is depicted by the solid red curve, the measured data (with noise) is depicted by the dotted magenta curve, and the EKF estimates by the dashed blue curve. In this example, EKF produces a reasonable approximation for the true solution. The results for this nonlinear system seem to be better than the other nonlinear system (**System 5.1**). This could be due to the use of MATLAB's built-in `extendedKalmanFilter` function rather than manually calculating each step. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Albers/

We can further assess the performance of the EKF algorithm by analyzing the measurement residuals (or innovation) of the velocity estimation. Recall that a residual essentially represents the error in a result. In this case, a residual is calculated by finding the difference between the noisy measurement value and the algorithm's estimated value. In this case, the algorithm is EKF, so the estimated value would be the value that EKF produces as an estimate of the state(s). Mathematically, this can be written as

$$\text{measurement residual} = \text{measurement value} - \text{estimated value}. \tag{5.3}$$

The resulting measurement residuals are depicted in **Figure 5.4**. From this plot, the residuals suggest that the EKF estimation performs fairly well. This is because the residuals generally have a small magnitude, zero mean, and no autocorrelation (except at zero lag). Looking at **Figure 5.4**, we see that the residuals range between -20 and 20 centered around 0, which indicates a fairly small magnitude (considering glucose is in the scale of 100) as well as zero mean. We can infer that the residuals lack autocorrection if

they appear randomly distributed and without a pattern, which they do. Consequently, the EKF algorithm seems to perform fairly well for the type 2 diabetes biological model.

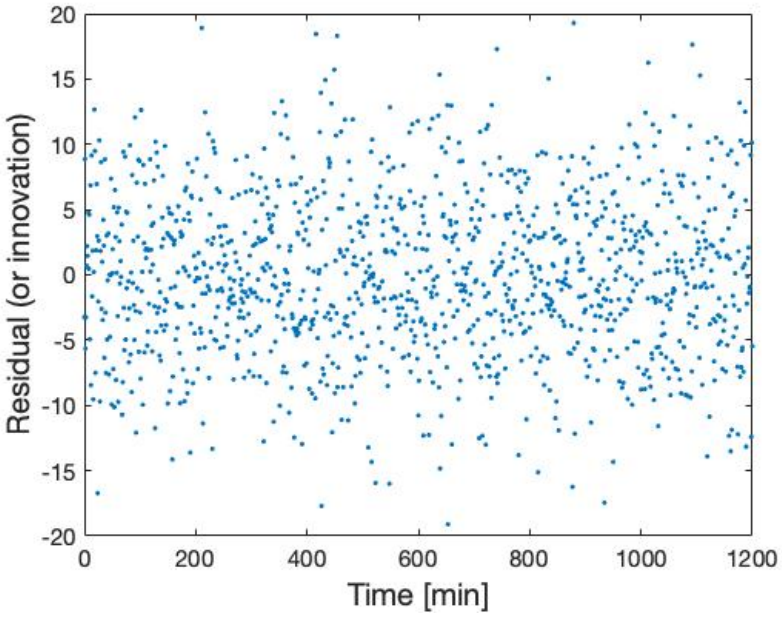

Figure 5.4 Measurement residuals of Albers' model EKF estimates. Note: measurement residual is defined in **Equation 5.3**. The residuals suggest that EKF is performing well on this system because they have a relatively small magnitude, zero mean, and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Albers/

Interestingly, these results seem to contradict the original hypothesis that EKF would not accurately estimate states for such a highly nonlinear system. This could be a result of using MATLAB's built-in `extendedKalmanFilter` algorithm. Recall that the we apply EKF to **System 5.2** by manually calculating each step, which may contribute to error. Unfortunately, we do not know MATLAB's encoding for its `extendedKalmanFilter` function, so cannot determine how this function differs from the other EKF implementation. Though, considering the performance of EKF on the nonlinear system, **System 5.2**, we may want to pursue another type of Kalman filter.

Recall that my goal is to use Kalman filtering to accurately estimate states and parameters of biological models, particularly diabetes models. These models are often nonlinear, so EKF may not be the best fit. As mentioned in **chapter 3**, the unscented Kalman filter may be a better alternative for nonlinear systems. In the following chapters, I discuss the unscented Kalman filter more.

Chapter 6

Unscented Kalman Filter

The unscented Kalman filter is a type of Kalman filter that, like the EKF, has a state distribution that is represented by a Gaussian random variable. The UKF is distinguished from other KFs by its use of the unscented transformation (UT), which was first introduced by Jeffrey Uhlmann in 1995 and further developed by Eric A. Wan and Rudolph van der Merwe [10]. Essentially, the UT attempts to completely capture the first two moments of the joint density in both the prediction and the correction steps of the KF. In other words, the UT is a method used to calculate the statistics of a random variable that is transformed nonlinearly [22]. It does so by deterministically sampling points from the joint density, applying the nonlinear dynamics to these points [10].

In other words, the UKF's state distribution is specified using a minimal set of carefully chosen sample points, which are called "sigma points" [22]. The sigma points completely capture the true mean and covariance of the state distribution. Consequently, when it is propagated through the true nonlinear system, it captures the posterior mean and covariance accurately to the second order linearization for any nonlinearity [22]. This process can be visually represented through **Figure 6.1**.

Figure 6.1 visualizes in two-dimension how the UT differs from other sampling methods that also attempt to accurately propagate the distribution of data. The left (blue) visualization shows the distribution of the

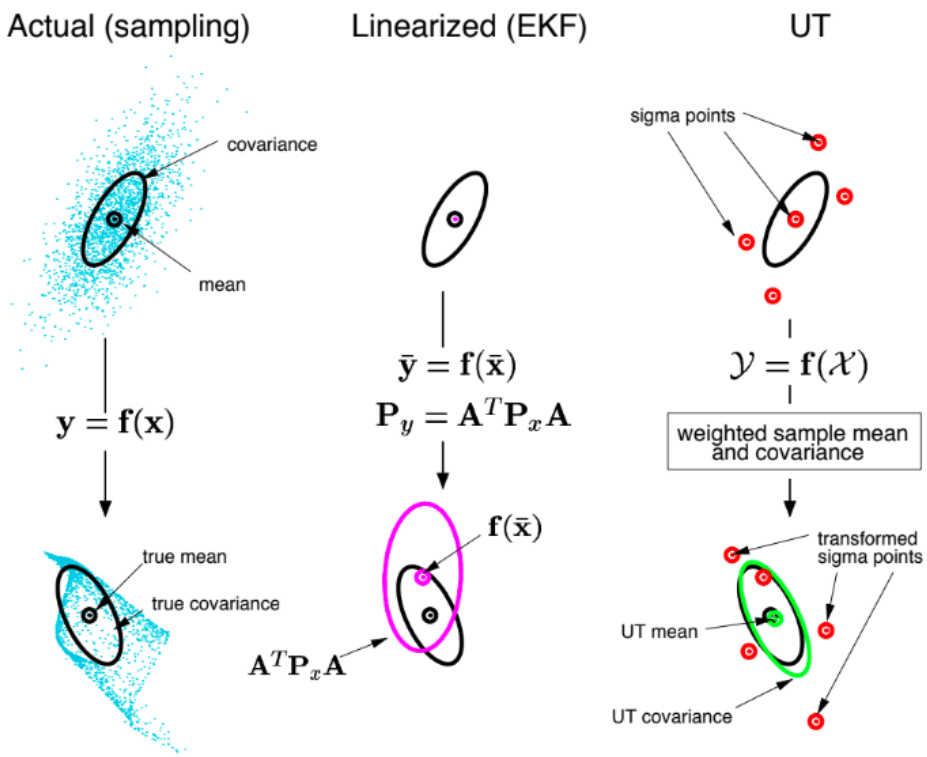

Figure 6.1 Visualization of UT from [22]. First image (left) is the actual distribution of the data as it is passed through a nonlinear system. Second (middle) image is the distribution of the data after it is passed through first-order linearization of a nonlinear transformation, and the third (right) image is the distribution of the data after it goes through UT of a nonlinear system.

actual data before and after it is propagated through the nonlinear system $y = f(x)$. Before the passing through the system, the data has a true mean and covariance associated with the distribution. After, the distribution of the data is different and, thus, the true mean and covariance have also shifted.

The middle (black and pink) visualization in **Figure 6.1** represents how using a linearization approach, such as EKF, propagates the distribution of the data. The true distribution is illustrated in black before and after it is propagated through the system. In this approach, the system is approximated with a linearization $\bar{y} = f(\bar{x})$, which produces an approximate mean $f(\bar{x})$ and approximate covariance $A^T P A$ that is depicted in pink. As

Variable	Description	Dimension
x	state vector	$n \times 1$
y	observation vector	$n_y \times 1$
u	input vector	$n_u \times 1$
w	process noise vector	$n \times 1$
v	measurement noise vector	$n_y \times 1$
F	state transition matrix	$n \times n$
G	input system matrix	$n \times n_u$
H	observation matrix	$n_y \times n$

Table 6.1 Discrete time system variables for unscented Kalman filter.

one can visually see, the linear approximation produces a distribution that is similar to the true distribution, but these approximations of the mean and covariance do not align well with the true mean and covariance (in black).

The right (black, red, and green) visualization in **Figure 6.1** displays how the UT propagates the distribution of data. As mentioned earlier, UT strategically chooses sigma points (red points) in the original data that act as representative points of the distribution of the data. Then, UT passes these points through the nonlinear system $\mathcal{Y} = f(\mathcal{X})$, which produces the transformed sigma points in red. In doing so, UT produces a mean and covariance (in green) that closely reflect the true mean and true covariance (in black).

6.1 Unscented Kalman Filter Algorithm

Similar to the KF algorithm, the unscented Kalman filter is applied to systems that can be discretized into state space format, which is defined in **System 6.1**, where the variable are defined in **Table 6.1** [19].

$$x_k = F_{k-1}x_{k-1} + G_{k-1}u_{k-1} + w_{k-1} \tag{6.1a}$$

$$y_k = H_k x_k + v_k. \tag{6.1b}$$

Additionally, like KF, UKF also consists of two main steps: the prediction step and the correction step. Therefore, we can visualize UKF using the same diagram that we did for the KF algorithm in **Figure 3.1**. As mentioned

before, the UKF algorithm is a derivative of the KF algorithm such that it utilizes the UT, applying it in the prediction step. Consequently, a diagram to represent the overall process of the UKF algorithm would not differ from that of the KF algorithm. Instead, the difference between the two algorithms can be seen in the specific steps and formulas used in the UKF algorithm.

Like the KF algorithm, we must first write the system of DEs in state-space form before implementing the UKF algorithm. Recall that we can do this by using the method explained in **chapter 3** and through **System 3.3**. With a state-space system of DEs, we can move on to the prediction step of the UKF algorithm.

6.1.1 Prediction Step

First, we must initialize our state vector $\hat{x}$ and state error covariance matrix P. To determine the initial state vector $\hat{x}_0$, we simply calculate the mean:

$$\hat{x}_0 = \mathbb{E}[x_0],$$

where $\mathbb{E}[x_0]$ is the expected value of x_0, which was defined earlier in **Equation 3.11**. To determine the initial state error covariance matrix P, we can use the same calculation as **Equation 3.10**.

With the initial state vector and state error covariance matrix, we can now calculate the sigma points, which are sample points that capture the true mean and covariance of the state distribution. First, calculate the scaling parameter λ as

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

where α and κ are constants defined in **Table 6.2**, and n is the dimension of the state vector $\hat{x}$. More specifically, α and κ are scalars that represent the spread of the sigma points around the mean state value. While α is a scalar in the range $0 < \alpha \leq 1$, κ is a scalar in the range $0 \leq \kappa \leq 3$. With λ calculated, we can now create a matrix $\mathcal{X}$ of $2n + 1$ sigma points. The sigma points $\mathcal{X}_i$ are calculated by

$$\begin{aligned} \mathcal{X}_i &= \hat{x} + (\sqrt{(n + \lambda)P})_i && i = 1, \ldots, n \\ \mathcal{X}_i &= \hat{x} - (\sqrt{(n + \lambda)P})_{i-n} && i = 1, \ldots, 2n \end{aligned}$$

Constant	Value	Representation
α	$0 < \alpha \leq 1$	spread of sigma points around mean state value
κ	$0 \leq \kappa \leq 3$	spread of sigma points around mean state value
β	$0 \leq \beta$	characterization of state distribution used to adjust weights of transformed sigma points

Table 6.2 Constants for unscented Kalman filter

Then, use the methodology described in [22] to determine the weights for the state filter. The weighting schemes W for determining the mean state estimates m and the covariances c are

$$\begin{aligned} W_0^{(m)} &= \frac{\lambda}{n+\lambda}, && i = 0 \\ W_0^{(c)} &= \frac{\lambda}{n+\lambda} + (1 - \alpha^2 + \beta), && i = 0 \\ W_i^{(m)} &= W_i^{(c)} = \frac{\lambda}{2(n+\lambda)}, && i = 1, \ldots, 2n \end{aligned}$$

where β is a constant defined in **Table 6.2**. This constant characterizes the distribution of the states and is used to adjust the weights of the transformed sigma points. β is a scalar greater than or equal to 0. It is important to note that the weights can be positive or negative and must sum to one [10]. Now, we have the sigma point set, which consists of the sigma points as well as the weights:

$$S = \{\mathcal{X}_i, W_i^{(j)} \mid i = 0, \ldots, 2n, j \in (m, c)\}.$$

For more context on the constants, α determines the spread of the sigma points around the mean state value. It is typically a small positive value. This means that smaller α values correspond to sigma points being closer to the mean state. κ is a scaling parameter that is usually set to zero. Like α, smaller κ values correspond to sigma points being closer to the mean state. β incorporates prior knowledge of the distribution of the state. For Gaussian distributions, an optimal β value is two [16].

Unlike the KF algorithm, the UKF algorithm requires more computation in the prediction step. In particular, the UKF algorithm calls for a non-

linear transformation. With our sigma point set, we apply the nonlinear transformation f to each sigma points. Mathematically, this means

$$\mathcal{Y}_i = f(\mathcal{X}_i), \qquad i = 0, \dots, 2n.$$

With the new set of nonlinear transformed sigma points, we have all the components we need to execute the correction step of the UKF algorithm.

6.1.2 Correction Step

The correction step of the UKF algorithm first calls for a re-calculation of the mean $\hat{y}$, covariance P_y, and cross covariance P_{xy}, respectively, for each time step k

$$\begin{aligned}
\hat{y} &= \sum_{i=0}^{2n} W_i^{(m)} \mathcal{Y}_i \\
P_y &= \sum_{i=0}^{2n} W_i^{(c)} (\mathcal{Y}_i - \hat{y})(\mathcal{Y}_i - \hat{y})^T + R_n \\
P_{xy} &= \sum_{i=0}^{2n} W_i^{(c)} (\mathcal{X}_i - \hat{x})(\mathcal{Y}_i - \hat{y})^T,
\end{aligned}$$

where R_n is the assumed measurement noise. Using our covariance and cross covariance from the transformed sigma points, we can compute the Kalman gain matrix for a given time step k:

$$K_k = P_{xy} P_y^{-1}.$$

With the Kalman gain matrix at time step k, we can update the state vector

$$\hat{x}_k = \hat{x}_{k|k-1} + K_k(y_k - \hat{y}_{k|k-1})$$

as well as the state error covariance matrix

$$P_k = P_{k|k-1} - K_k P_y K_k^{-1}.$$

To help solidify the unscented Kalman filter algorithm, it is important to go through an example. In the following chapter, I provide three different examples of the UKF algorithm and how to implement them to estimate state values of dynamical systems.

Chapter 7

UKF State Estimation

In order to implement the UKF algorithm to estimate state values, I use MATLAB's pre-existing `unscentedKalmanFilter` function. This function takes in the same inputs as the MATLAB `extendedKalmanFilter` function. Recall that the inputs are a function `StateTransitionFcn`, a function `MeasurementFcn`, a vector `InitialState`, a vector or matrix `MeasurementNoise`, and a square matrix `ProcessNoise`.

`StateTransitionFcn` calculates the state vector of the system at time step k given the state vector at time step $k-1$. `MeasurementFcn` calculates the output measurement vector of the system at time step k given the sate vector at time step k. `InitialState` represents the initial state values based on the user's knowledge of the system. Therefore, it is a vector with length n, where n represents the number of states in the system. `MeasurementNoise` represents the measurement noise covariance. It is a scalar when `HasAdditiveMeasurementNoise` is set to true and a matrix when `HasAdditiveMeasurementNoise` is set to false. If it is a matrix, then its dimensions are $v \times v$, where v represents the number of measurement noise terms. `ProcessNoise` represents the process noise covariance, and it is a square matrix with dimensions $w \times w$, where w represents the number of process noise terms.

Along with this pre-existing `unscentedKalmanFilter` function, MATLAB provides an example of an implementation of Van der Pol's equation that came with public source code [17]. Basing my implementation on this example, I was able to implement two different systems of differential equations. One was a simple linear system while the other was a nonlinear

system consisting of six differential equations.

The code corresponding to this chapter can be found in the following GitHub repository: https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/. For the nonlinear system, Van der Pol's equation, the code is in the subfolder `MatLab_vdp_Example`. The code corresponding to the linear system, the kinematic equations, is in the subfolder `Kinematic`. The type 2 diabetes physiological model's code is in the subfolder `Albers`.

7.1 Nonlinear System: Van der Pol Equation

As mentioned, MATLAB provides an example for how to implement the pre-existing `unscentedKalmanFilter` function [17]. In other implementations of the UKF algorithm, we use this source code as a foundation. The example uses Van der Pol's equation to implement UKF. Physically, this DE describes the dynamics of self-sustaining oscillations. One physical example of these oscillations is the heartbeat. In fact, Van der Pol's equation is often used to model the heartbeat. Mathematically, Van der Pol's equation is written as

$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = 0,$$

where x is position, $\frac{dx}{dt}$ is velocity, and $\frac{d^2x}{dt^2}$ is acceleration, and μ is the damping constant. First, we must re-write this DE in state-space form. In order to do so, we can create a state vector that represents this DE and, thus, describes the oscillations of an object. Because velocity $\frac{dx}{dt}$ and acceleration $\frac{d^2x}{dt^2}$ appear in the equation, we must include both in the state vector. Therefore, the state vector for the unscented Kalman filter is defined as

$$\begin{aligned}\hat{x}_k &= \begin{bmatrix} \frac{dx}{dt} \\ \frac{d^2x}{dt^2} \end{bmatrix} \\ &= \begin{bmatrix} \frac{dx}{dt} \\ \mu(1-x^2)\frac{dx}{dt} - x \end{bmatrix}.\end{aligned}$$

With the state vector for Van der Pol's system, we now have the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `unscentedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output only the first element in the state vector, which

represents velocity. We do this because the acceleration is based on the change in velocity, so we do not find it necessary to create measurement values for acceleration. By doing this, the UKF algorithm only measures velocity. Physically, this would represent an experiment where only velocity is measured rather than position, velocity, and acceleration.

Additionally, the `InitialState` is set to $[2, 0]$, which is given by MATLAB. This can be interpreted as setting the initial velocity to 2 meters per second and the initial acceleration to 0 meters per second^2. The MATLAB algorithm also has the capability to add an additional input to represent noise, called `MeasurementNoise` and `ProcessNoise`.

In a brief tangent, I take a moment to explain the difference between measurement noise and process noise, which are often variables incorporated in systems and models. Both are key components in accounting for noise, though they represent very different ways in which noise can enter a system. Measurement noise accounts for the uncertainty in the data measurements, whereas process noise accounts for the uncertainty in the system or model.

Let us consider a scenario where there is noise collected based on the state. Then, this noise would cause a slight uncertainty in the state measurement. As a result, we must account for this measurement noise as a vector $\hat{v}$ that is added to the state $\hat{x}$. Thus, the measurement noise function is multiplicative:

$$\hat{y}_k = \hat{x}_k(1 + \hat{v}_k).$$

Since the measurement noise is multiplicative, we must also be sure to set `HasAdditiveMeasurementNoise` as false. The measurement vector $\hat{v}_k$ has the same dimensions as the state vector $\hat{x}_k$. In this example, $\hat{v}_k$ is set as a vector that consists of one value, which is 0.2. This is the value that is input as the `MeasurementNoise`.

For `ProcessNoise`, the input must be a square matrix with dimensions $n \times n$, where n represents the number of states there are. In this example, the values for `ProcessNoise` are set as

$$\begin{bmatrix} 0.02 & 0 \\ 0 & 0.1 \end{bmatrix}$$

It is important to note that the values for `ProcessNoise` are on a similar scale as the `MeasurementNoise`. This is because a disparity in these two

inputs would cause unusual outputs or even overfitting.

With all the inputs, we can now implement the MATLAB `unscentedKalmanFilter` function. Rather than collecting data measurements, data measurements were simulated such that they incorporate some random noise, of which is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which was created by scaling values that were generated using the DEs by a randomly-determined scalar. Then, for each measured data point, we call the `unscentedKalmanFilter` function to update the state and covariance based on the previous estimation and the latest measured data point, which produce the "UKF estimate" values.

In **Figure 7.1**, we can see the results of applying the UKF algorithm to the Van der Pol system of DEs. The first graph represents the velocity of the oscillations $\frac{dx}{dt}$, and the second graph represents the acceleration of the oscillations $\frac{d^2x}{dt^2}$. As one can see, the measured data (in yellow) is quite noisy while the true data (in blue) is smooth. There is no measured data for the acceleration because we did not have the `MeasurementFcn` produce values for acceleration.

Looking at both graphs, it is evident that the values estimated by the UKF algorithm (in orange) is not only accurate but also smooths the data. From this example of Van der Pol's oscillation, we see the accurate as well as clean results of applying the UKF algorithm to a dynamical system.

We can further assess the performance of the UKF algorithm by analyzing the measurement residuals (or innovation) of the velocity estimation. Recall that residuals are explained in **chapter 5** and **Equation 5.3**. The resulting measurement residuals are depicted in **Figure 7.2**.

From this plot, the measurement residuals suggest that the UKF estimation performs fairly well. This is because the residuals generally have a small magnitude, zero mean, and no autocorrelation (except at zero lag). Looking at **Figure 7.2**, we see that the residuals range between -2 and 2 centered around 0, which indicates a small magnitude as well as zero mean. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which it does. Consequently, the UKF

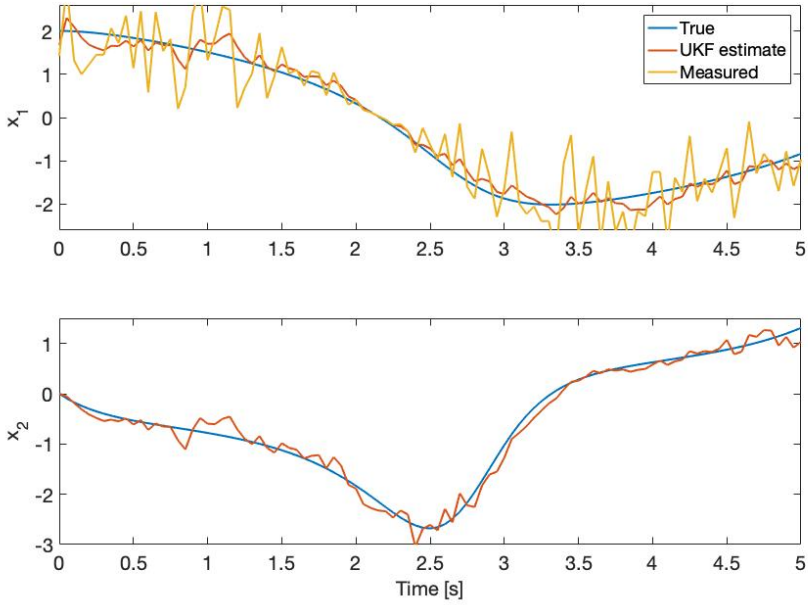

Figure 7.1 Results of UKF implementation of Van der Pol equation. The true solution for the states x_1 (velocity) and x_2 (acceleration) are depicted by the blue curves, the measured data (with noise) for velocity by the yellow curve, and the EKF estimates for x_1 and x_2 by the orange curve. In this example, UKF produces a reasonable approximation for the true solution, accurately estimating and smoothing the data. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/MatLab_vdp_Example/

algorithm seems to perform fairly well for the Van der Pol equation.

Using this example, I now have the framework to implement other dynamical systems - both simple and complex.

7.2 Linear System: Kinematic Equation

The first system of differential equations that I implemented was a simple linear system - the kinematic equations. The system of DEs for the kinematic equations can be derived from a second-order differential equation:

$$h''(t) = -g, \tag{7.1}$$

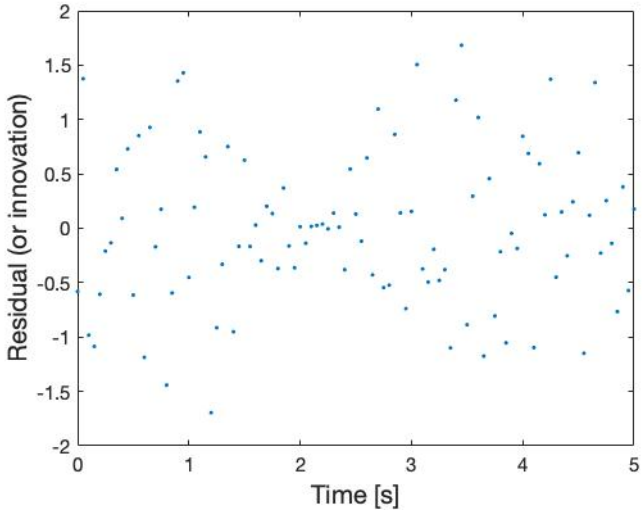

Figure 7.2 Measurement residuals (or innovation) of Van der Pol equation UKF estimates. Note: measurement residual is defined in **Equation 5.3**. The residuals suggest that UKF performs well on this system because they have small magnitudes, zero mean, and no atuocorrelation (except at zero lag). Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/MatLab_vdp_Example/

where h represents the height of an object in meters and g is the gravitational constant, 9.80665 meters per second^2. **Equation 7.1** indicates that the second derivative of height with respect to time t is equivalent to acceleration due to gravity.

Before applying the UKF algorithm to this DE, we have to rewrite the system in discretized state-space form, as explained in **3**. We can do so using the Euler method, which will result in a state-space format system that resembles **System 3.3**. First, we will rewrite **Equation 7.1** as a difference quotient.

$$h''(t_{k-1}) = \frac{h'(t_k) - h'(t_{k-1})}{\Delta t} = -g.$$

Consider only the first derivative and the gravitational constant, then we

can simplify this to

$$h'(t_k) = h'(t_{k-1}) - g\Delta t$$
$$h'_k = h'_{k-1} - g\Delta t.$$

This first-order DE represents the velocity of an object, which seems to be an implicit equation. We can integrate this to get the position of an object:

$$h(t_k) = h(t_{k-1}) + h'(t_{k-1})\Delta t - \frac{1}{2}g(\Delta t)^2$$
$$h_k = h_{k-1} + h'_{k-1}\Delta t - \frac{1}{2}g(\Delta t)^2.$$

With these kinematic expressions, we have two equations that describe the motion of an object. Now, we can create a state vector that describes the motion, or kinematics, of an object. Since the velocity equation h'_k appears in the position equation h_k, we must include both the position and the velocity equations in the state vector. Therefore, the state vector for the unscented Kalman filter is defined as

$$\hat{x}_k = \begin{bmatrix} h_k \\ h'_k \end{bmatrix} = \begin{bmatrix} h_{k-1} + h'_{k-1}\Delta t - \frac{1}{2}g(\Delta t)^2 \\ h'_{k-1} - g\Delta t \end{bmatrix}.$$

The discretization of **Equation 7.1** is explained in more detail in [19]. With the state vector for the kinematic system, we now have the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `unscentedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output only the first element in the state vector, which represents velocity. By doing this, the UKF algorithm only measures velocity. Physically, this would represent an experiment where only velocity is measured rather than velocity and acceleration.

Additionally, we set the `InitialState` to $[2, 0]$, using the same initial values as the MATLAB example for the Van der Pol oscillator. With these initial values, it represents setting the initial position to be 2 meters and the initial velocity to be 0 meters per second. The MATLAB algorithm also has the capability to add additional inputs, `MeasurementNoise` and `ProcessNoise`, to represent noise.

Similar to the previous example of Van der Pol's equation, let us consider a scenario where there is noise collected based on the position. Then, this

noise would cause a slight uncertainty in the position measurement. As a result, we must account for this measurement noise as a vector $\hat{v}$ that is added to the position $\hat{h}$ because it only shifts the position slightly. Thus, the measurement noise is additive:

$$\hat{y}_k = \hat{h}_k + \hat{v}_k.$$

Since the measurement noise is additive, we must be sure to set `HasAdditiveMeasurementNoise` to be true. The measurement vector $\hat{v}_k$ has the same dimensions as the state vector $\hat{x}_k$. Like the Van der Pol example, we set $\hat{v}_k$ to be a vector that consists of one value, which we arbitrarily choose to be 0.2. This gets input as the `MeasurementNoise`.

For `ProcessNoise`, the input must be a square matrix with dimensions $n \times n$, where n represents the number of states there are. In this example, we set the values for `ProcessNoise` to be the same values as the `ProcessNoise` in the previous example:

$$\begin{bmatrix} 0.02 & 0 \\ 0 & 0.1 \end{bmatrix}$$

Now, we have all the inputs to implement the MATLAB `unscentedKalmanFilter` function. Rather than use real data measurements, we create simulated data measurements that incorporate some random noise, of which is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which was created by adding a randomly-determined shift to values that were generated using the DEs. Then, for each measured data point, we call the `unscentedKalmanFilter` function to update the state and covariance based on the previous estimation and the latest measured data point, which produce the "UKF estimate" values.

In **Figure 7.3**, we can see the results of applying the UKF algorithm to the kinematics system of DEs. Rather than displaying both the position and velocity, I only graph the results of velocity. This is because velocity is a derivative of position and, thus, is dependent on position. The graph displays the change in velocity of an object in meters per second over the time span of five seconds. As one can see, the measured data (in yellow) is somewhat noisy while the true data (in blue) is smooth.

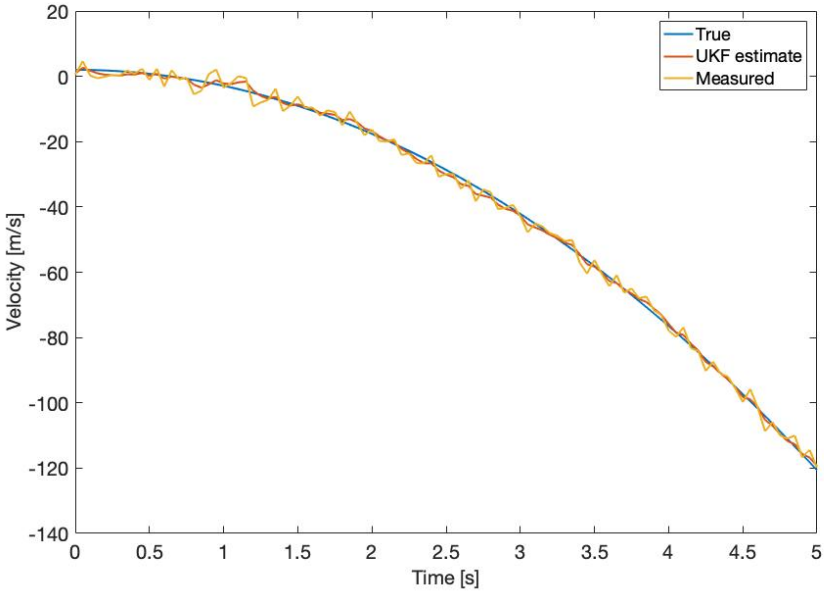

Figure 7.3 Results of UKF implementation of kinematics equation. The true solution for the velocity state is depicted by the solid blue curve, the measured data (with noise) is depicted by the solid yellow curve, and the UKF estimates by the solid orange curve. In this example, UKF produces a good approximation for the true solution of velocity, accurately estimating and smoothing the data. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Kinematic/

Looking at the results, it is evident that the values estimated by the UKF algorithm (in orange) is not only accurate but also smooths the data. The UKF estimate values accounts for noise while remaining accurate. From this example of kinematics, we see the accurate as well as clean results of applying the UKF algorithm to a simple linear system.

We can further assess the performance of the UKF algorithm by analyzing the measurement residuals of the velocity estimation. Recall that residuals are explained in **chapter 5** and **Equation 5.3**. The resulting measurement residuals are depicted in **Figure 7.4**.

From this plot, the measurement residuals suggest that the UKF estimation performs fairly well. This is because the residuals generally have a

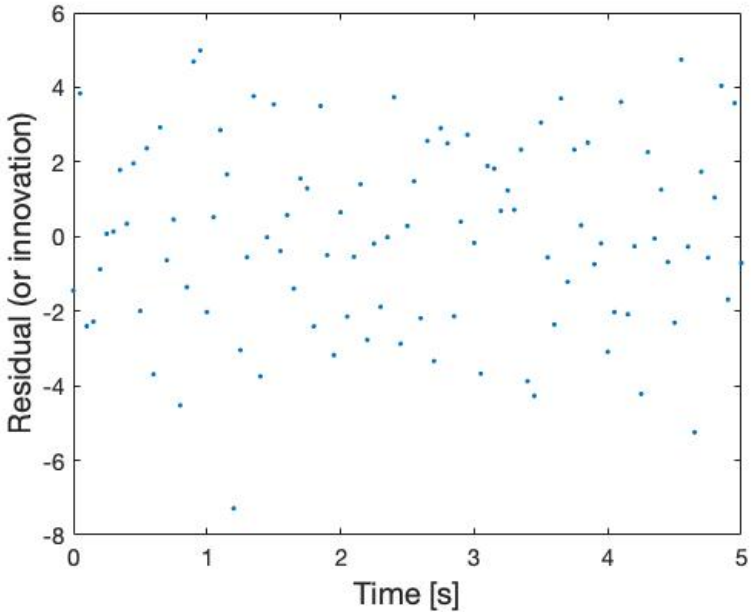

Figure 7.4 Measurement residuals (or innovation) of kinematic equation UKF estimates. Note: measurement residual is defined in **Equation 5.3**. The residuals suggest that UKF is performing well on this system because they have small magnitudes, zero mean, and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Kinematic/

small magnitude, zero mean, and no autocorrelation (except at zero lag). Looking at **Figure 7.4**, we see that the residuals range between -8 and 6 mainly centered around 0, which indicates a small magnitude as well as zero mean. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which it does. Consequently, the UKF algorithm seems to perform fairly well for the kinematic equation.

After implementing the UKF algorithm on a simple linear system, I can move onto a more complex system.

7.3 Type 2 Diabetes Physiological Model

The second system of differential equations that I implement is a biological nonlinear system that consisted of six ordinary differential equations (ODEs). This system of ODEs is taken from the equations David J. Albers used to model glucose/insulin through MATLAB [5]. It is supposedly the equations in the population physiology model from [6]. This model simulates glucose-insulin physiology among individuals with T2D. As mentioned in **chapter 1**, this model is based on an earlier model proposed in [21].

However, there seems to be some discrepancy between the two systems of ODEs. In particular, the two ODEs that represent the change in glucose do not seem to be the same. Interestingly, [5] seems to implement the six DEs in [21] rather than their own paper. Currently, I do not why these two systems differ and how significantly they differ. For now, we only consider the DEs from [21].

Typically, we first have to discretize our system of DEs and rewrite the system in state-space form. However, this glucose-insulin physiology model is already in state-space form.

The six ODEs are as follows:

$$\frac{dI_p}{dt} = \frac{R_m}{1 - exp(\frac{-G}{V_g C_1} + a_1)} - E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_p}{t_p}$$

$$\frac{dI_i}{dt} = E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_i}{t_i}$$

$$\frac{dG}{dt} = \frac{R_g}{1 + exp(\frac{0.29h_3}{V_p - 7.5})} + I_g - U_b\left(1 - exp\left(\frac{-G}{C_2 V_g}\right)\right) - \frac{90}{1 + exp(-1.772 \log\left(I_i\left(\frac{1}{V_i} + \frac{1}{Et_i}\right)\right) + 7.76)} + 4$$

$$\frac{dh_1}{dt} = \frac{3(I_p - h_1)}{t_d}$$

$$\frac{dh_2}{dt} = \frac{3(h_1 - h_2)}{t_d}$$

$$\frac{dh_3}{dt} = \frac{3(h_2 - h_3)}{t_d}$$

The thirty different constants that appear in the system of ODEs can be found in **Appendix B**. These constants were taken from [6].

This set of six ODEs are then used to create the state vector $\hat{x}_k$, where the state vector is

$$\hat{x}_k = \begin{bmatrix} I_p \\ I_i \\ G \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}.$$

where I_p represents plasma insulin, I_i represents remote insulin, G represents glucose, and the last three states represent different delayed feeding cycles. By feeding cycle, we mean an individual's eating habits. Therefore, this system explores three different possible eating habits. In the system, h_1 represents the first stage linear filter feeding cycle, h_2 represents the second stage linear filter feeding cycle, and h_3 represents the third stage linear filter

feeding cycle.

In order to determine the states based on the system of ODEs, we use the MATLAB ODE-solver called `ode45`, which takes in a DE as input and returns the solution to the DE. Consequently, when applying `ode45` to the T2D model we get the state vector $\hat{x}_k$.

With the state vector for the glucose-insulin physiology system, we can create the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `unscentedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output only the third element in the state vector, which represents glucose. By doing this, the UKF algorithm only measures glucose. This is because we mainly care about the change in glucose levels, so we choose to decrease computational cost by only measuring one state.

Additionally, we set the `InitialState` vector to be the initial values that were used in [21]. This has the initial values as

$$[200, 200, 12000, 0.1, 0.2, 0.1].$$

In other words, we set the plasma insulin to be 200 mU/min, remote insulin to be 200 mU/min, glucose to be 12000 mU/min, the first stage linear filter to be 0.1 mU/min^2, the second stage linear filter to be 0.2 mU/min^2, and the third stage linear filter to be 0.1 mU/min^2.

The MATLAB algorithm also has the capability to add an additional input to represent measurement and process noise. Similar to the kinematic example, we let the measurement noise function be an additive, so

$$\hat{y}_k = \hat{x}_k + \hat{v}_k.$$

Since the measurement noise is additive, we must be sure to set `HasAdditiveMeasurementNoise` to be true. The measurement vector $\hat{v}_k$ has the same dimensions as the state vector $\hat{x}_k$. Similar to the previous two examples, we set $\hat{v}_k$ to be a vector that consists of one arbitrarily-chosen value, 5. We chose this value because it adds sufficient noise to the state values for glucose, which are on the scale of a hundred. This gets input as the `MeasurementNoise`.

For `ProcessNoise`, the input must be a square matrix with dimensions $n \times n$, where n represents the number of states there are. In this example,

the values for `ProcessNoise` are chosen to be on a similar scale as the `MeasurementNoise`. As a result, the input for `ProcessNoise` is

$$\begin{bmatrix} 0.02 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.04 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}$$

Now, we have all the inputs to implement the MATLAB `unscented-KalmanFilter` function. This process follows the same process as the kinematic example, such that we create simulated data measurements that incorporate some random noise that is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which was created by adding a randomly-determined shift to values that were generated using the DEs. Then, we call the `unscented-KalmanFilter` function for each measured data point to update the state and covariance based on the previous estimation and the latest measured data point, which produce the "UKF estimate" values.

In **Figure 7.5**, we can see the results of applying the UKF algorithm to the Albers' et al's system of DEs. Rather than displaying all six states in the model, I only graph the results of glucose. This is because the algorithm only measures the change in glucose levels, as mentioned earlier. The graph displays the change in glucose levels in milligrams per liter over 1200 minutes. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

Looking at the results, it seems that the values estimated by the UKF algorithm (in blue solid line) are accurate at predicting the glucose levels as well as correctly smoothing the data.

We can further assess the performance of the UKF algorithm by analyzing the measurement residuals of the velocity estimation. Recall that residuals are explained in **chapter 5** and **Equation 5.3**. The resulting measurement residuals are depicted in **Figure 7.6**.

From this plot, the measurement residuals suggest that the UKF estimation performs fairly well. This is because the residuals generally have a small

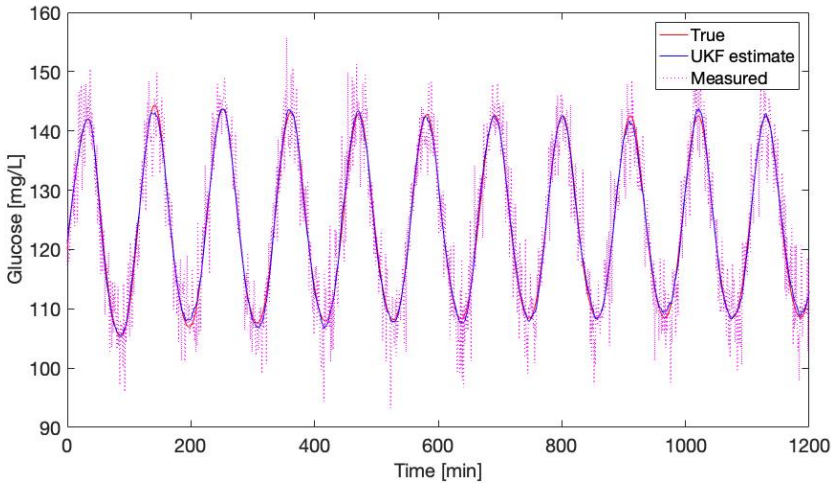

Figure 7.5 Results of UKF implementation of Albers et al's equation. The true solution for the glucose state is depicted by the solid red curve, the measured data (with noise) is depicted by the dotted magenta curve, and the EKF estimates by the solid blue curve. In this example, UKF produces a reasonable approximation for the true solution, accurately estimating and smoothing the data. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/

magnitude, zero mean, and no autocorrelation (except at zero lag). Looking at **Figure 7.6**, we see that the residuals range between -20 and 20 centered around 0, which indicates a small magnitude (considering glucose is in the scale of 100) as well as zero mean. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. Consequently, the UKF algorithm seems to perform fairly well for the type 2 diabetes system.

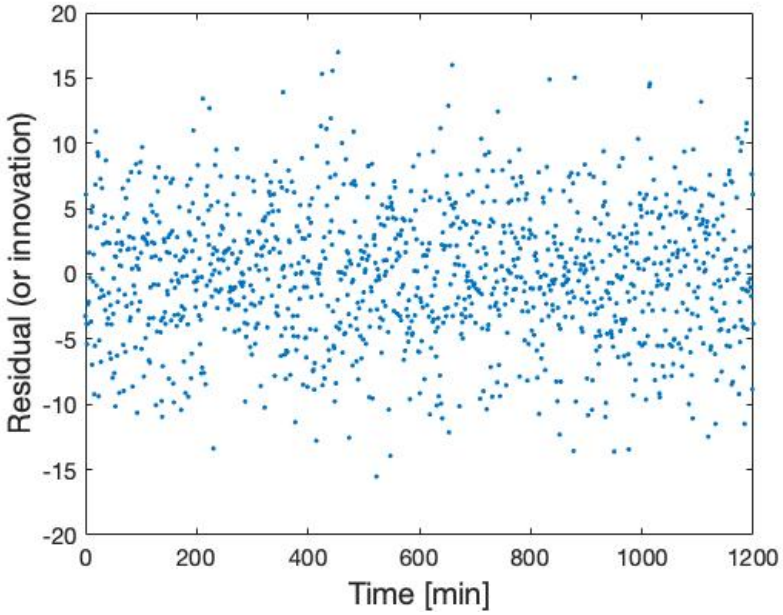

Figure 7.6 Measurement residuals (or innovation) of Albers' model UKF estimates. Note: measurement residual is defined in **Equation 5.3**. The residuals suggest that UKF is performing well on this system because they have a relatively small magnitude, zero mean, and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/

Chapter 8

Methods for State and Parameter Estimation

There are many different approaches to estimate both states and parameters of models. For instance, the Schmidt-Kalman filter indirectly estimates parameters through state estimation rather than explicitly estimating parameters. Another method is the state-dependent approach, which estimates statistically dependent unknown states and parameters using a multi-step autoregressive filtering and modeling process [10]. For Kalman filtering, the there are two main methods proposed in [22]: joint estimation and dual estimation.

Joint differs from dual in the number of filters required in order to estimate both states and parameters. Additionally, joint estimation explicitly allows for dependencies in parameters and states whereas dual estimation assumes that cross covariances are zero. Therefore, dual estimation does not explicitly estimate the cross covariances [10]. As a result, if there is suspicion that states and parameters are correlated, it can be argued that joint estimation may perform better [22].

8.1 Joint Estimation

In joint estimation, the states x and parameters a are concatenated into a single "joint" state vector $[x_k^T a_k^T]^T$ [22]. Write the state-space equation for the

joint state as the following:

$$\begin{bmatrix} x_k \\ a_k \end{bmatrix} = \begin{bmatrix} F(x_{k-1}, a_{k-1}) \\ I a_{k-1} \end{bmatrix} + \begin{bmatrix} w_k \\ b_k \end{bmatrix} \tag{8.1}$$

$$y_k = H_k \begin{bmatrix} x_k \\ a_k \end{bmatrix} + v_k, \tag{8.2}$$

where F is the linear or nonlinear transformation matrix, I is the identity matrix, w_k is the process noise associated with states, b_k is the process noise associated with parameters, y_k is the observation, H_k is the observation matrix, and v_k is the observation noise.

Recursively apply Kalman filtering on the joint state-space defined in **System 8.1** to simultaneously estimate the states x and the parameters a [22]. Joint estimation can be applied to both UKF and EKF.

8.2 Dual Estimation

In dual estimation, the states x and parameters a are represented in separate state-space equations. The state-space equation for the states x is given by

$$\begin{aligned} x_k &= F(x_{k-1}, a) + w_{k-1} \\ y_k &= H_k x_k + v_k, \end{aligned}$$

where F is the linear or nonlinear transformation matrix, w_k is the process noise associated with states, y_k is the observation, H_k is the observation matrix, and v_k is the observation noise.

The state-space representation for the parameters is written as follows:

$$\begin{aligned} a_k &= a_{k-1} + b_k \\ y_k &= F(x_{k-1}, a_k) + w_k + v_k, \end{aligned}$$

where b_k is the process noise associated with parameters, y_k is the observation, F is the linear or nonlinear transformation matrix, w_k is the process noise associated with states, and v_k is the observation noise. Note that the state-transition for the parameter is linear. This means that nonlinearity is restricted to the measurement equation only.

Recursively run two simultaneous Kalman filterings, one for the states and one for the parameters. For each time step, the most-updated estimate for the parameters is used in the KF estimation for states. Similarly, at every time step, the most-updated estimate for the states is used in the KF estimation for the parameters [22]. Dual estimation can be applied to both UKF and EKF.

In the following chapters, I explore joint estimation in Kalman filtering in different dynamical systems.

Chapter 9

Joint EKF State and Parameter Estimation

In order to implement joint EKF to estimate state and parameter values, I use the same source code that was used in **chapter 5**, which is taken from [8]. More specifically, I use the provided code from "Chapter 11.4: Examples of non-linear fitting" of this book as a basis for my implementation of joint EKF. In this chapter, I go through one implementation using this method. This implementation estimates both state and parameter of the same linear system I mentioned in **chapter 5**.

Recall that the code from [8] consists of two functions: `sim_gss` and `ekf_gss`. The first function, `sim_gss`, simulates Gaussian data for the system, which consists of state function(s) and measurement function(s). As input, it takes the variance of the process noise `Q`, the variance of the measurement noise `R`, the mean `m0`, the variance `P0`, and the number of values produced `N`. For any system, `Q` and `R` are single-valued inputs. The dimensions of `m0` and `P0` depend on the system. In general, if the system consists of n equations, then `m0` is a vector of length n and `P0` is an $n \times n$ matrix. The results of `sim_gss` represent true values for the state(s) as well as the measurement(s).

The second function, `ekf_gss`, implements the EKF algorithm and produces estimates for the given system. It takes in the same inputs as `sim_gss` with an additionally two inputs, which are the outputs of `sim_gss`. The two inputs are the simulated data for the state variables `x` and the simulated

data for the measurements `z`. The dimensions of `x` and `z` depend on the system. If the system has a state equations and b measurement equations, then `x` is a $N \times a$ matrix and `z` is a $N \times b$ matrix. The results of `ekf_gss` are the EKF estimates for the state(s) and the measurement(s).

In addition to the method used in **chapter 5**, I also use MATLAB's pre-existing `extendedKalmanFilter` function to implement joint EKF on a biological model. Recall that this function takes as input a function `StateTransitionFcn`, a function `MeasurementFcn`, a vector `InitialState`, a vector or matrix `MeasurementNoise`, and a square matrix `ProcessNoise`.

`StateTransitionFcn` calculates the state vector of the system at time step k given the state vector at time step $k-1$. `MeasurementFcn` calculates the output measurement vector of the system at time step k given the sate vector at time step k. `InitialState` represents the initial state values based on the user's knowledge of the system. Therefore, it is a vector with length n, where n represents the number of states in the system. `MeasurementNoise` represents the measurement noise covariance. It is a scalar when `HasAdditiveMeasurementNoise` is set to true and a matrix when `HasAdditiveMeasurementNoise` is set to false. If it is a matrix, then its dimensions are $v \times v$, where v represents the number of measurement noise terms. `ProcessNoise` represents the process noise covariance, and it is a square matrix with dimensions $w \times w$, where w represents the number of process noise terms.

Using the pre-existing MATLAB function `extendedKalmanFilter`, I estimate states and parameters for a biological model, specifically the T2D model discussed in **chapter 1**. I choose to use this method rather than the method used from [8] because the biological model is complex and can be conveniently implemented using the MATLAB function `extendedKalmanFilter`.

The code corresponding to this chapter can be found in the following GitHub repository: https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation. For the linear system, the code is in the subfolder `Bolviken_Ex7`. The code corresponding to the type 2 diabetes physiological model is in the subfolder `Albers`.

9.1 Linear System

Consider the the same linear system from **Equation 5.1**. We can adjust this system from a state estimation problem to a state and parameter estimation problem by defining a second state variable $x_2(k) = a$. This allows us to derive the augmented non-linear system

$$x_1(k) = x_2(k-1)x_1(k-1) + w_1(k-1) \tag{9.1a}$$
$$x_2(k) = x_2(k-1) \tag{9.1b}$$
$$z(k) = x_2(k)x_1(k) + v(k). \tag{9.1c}$$

Like the previous two examples, the simulated data function `sim_gss` simply rewrites the system so that it takes in values and passes it through the system to produce simulated true values for both the states x and the measurements z. The input values were guessed to be the following: `Q` $= 0.01$, `R` $= 0.02$, $N = 50$,

$$\texttt{m0} = \begin{bmatrix} 2 & 1 \end{bmatrix}, \qquad \texttt{P0} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The `ekf_gss` implements EKF on the system. The first two inputs, `x` and `z`, were the outputs of `sim_gss` and represent what the algorithm is measuring. These results can be found in the GitHub repository mentioned earlier. The other input values are the same as the inputs for `sim_gss`. The resulting EKF estimation for the states x and z are depicted in **Figure 9.1**.

In **Figure 9.1**, there are three plots: the left displays the results for the `x` state, the middle shows the results for the `z` state, and the right indicates the results for the parameter a. For all three plots, the true values (simulated data) for each state are in red while the EKF estimates are in blue. Looking at these plots, the EKF algorithm seems to perform fairly well in estimating states for this system because the EKF estimates for states `x` and `z` seem to follow closely to the true values.

However, for the parameter estimation in the third plot, the EKF estimates seem to vary significantly, though oscillating around the true value. From the y-axis, it looks like the estimates vary by ± 0.15. Further investigation indicates that the minimum estimation is 0.7907 and maximum is 1.8273.

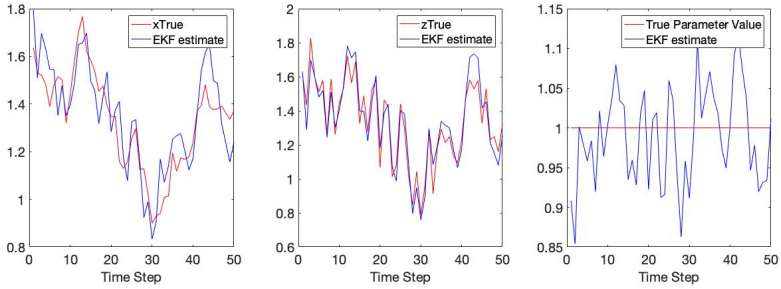

Figure 9.1 Results of EKF implementation of **System 9.1**. The true solutions for states $x(k)$ and $z(k)$ as well as the parameter $x_2 = a$ are depicted by the red curves/line, and the EKF produces a reasonable approximation for the true state solution but not the parameter estimation (errors and simulated data not shown). This may suggest that joint estimation does not perform well in predicting states and parameters. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation/Bolviken_Ex7/

The average of all estimates is 1.3363 and the median is 1.3241. After 50 times steps, the joint EKF algorithm estimates parameter a to be 1.3227. Given that the true value of a is 1, these estimates do not seem to be accurate, especially since there is such a large variance. Although joint EKF does not accurate estimate the parameter value, the estimates for the other states (x and z) suggest that EKF has fair performance when estimating states jointly with parameters for linear systems.

9.2 Type 2 Diabetes Physiological Model

The second system of differential equations that I implement is the same biological nonlinear system from **chapter 5**. This system consists of six ordinary differential equations (ODEs) that represent the states and three ODEs that represent the parameters. The three parameters we estimate are the following: exchange rate for insulin between remote compartments and plasma compartments E, insulin volume V_i, and time of remote insulin degradation t_i.

This system of ODEs is taken from the equations David J. Albers used to model glucose/insulin through MATLAB [5]. It is supposedly the equations

in the population physiology model from [6]. This model simulates glucose-insulin physiology among individuals with T2D. As mentioned earlier in **chapter 1**, this model is based on an earlier model by proposed in [21].

Typically, we first have to rewrite our system of DEs in state-space form. However, this glucose-insulin physiology model is already in state-space form.

The nine ODEs are as follows:

$$
\begin{aligned}
\frac{dI_p}{dt} &= \frac{R_m}{1 - exp(\frac{-G}{V_g C_1} + a_1)} - E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_p}{t_p} \\
\frac{dI_i}{dt} &= E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_i}{t_i} \\
\frac{dG}{dt} &= \frac{R_g}{1 + exp(\frac{0.29 h_3}{V_p - 7.5})} + I_g - U_b\left(1 - exp\left(\frac{-G}{C_2 V_g}\right)\right) \\
&\quad - \frac{90}{1 + exp(-1.772 \log\left(I_i\left(\frac{1}{V_i} + \frac{1}{E t_i}\right)\right) + 7.76)} + 4 \\
\frac{dh_1}{dt} &= \frac{3(I_p - h_1)}{t_d} \\
\frac{dh_2}{dt} &= \frac{3(h_1 - h_2)}{t_d} \\
\frac{dh_3}{dt} &= \frac{3(h_2 - h_3)}{t_d} \\
\frac{dE}{dt} &= 0 \\
\frac{dV_i}{dt} &= 0 \\
\frac{dt_i}{dt} &= 0.
\end{aligned}
$$

The different constants that appear in the system of ODEs can be found in **Appendix B**. Note that since we are estimating the three parameters, they are passed through the filter as part of the state vector rather than as constants. The true values for these three parameters are still listed in **Appendix B** for reference. The other constants were taken from [6].

This set of nine ODEs are then used to create the state vector $\hat{x}_k$, where the state vector is

$$\hat{x}_k = \begin{bmatrix} I_p \\ I_i \\ G \\ h_1 \\ h_2 \\ h_3 \\ E \\ V_i \\ t_i \end{bmatrix}.$$

In order to determine the states based on the system of ODEs, we use the MATLAB ODE-solver called `ode45`, which takes in a DE as input and returns the solution to the DE. Consequently, when applying `ode45` to the T2D model we get the state vector $\hat{x}_k$.

With the state vector for the glucose-insulin physiology system, we can create the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `extendedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output the third element in the state vector, which represents glucose, as well as the seventh, eighth, and ninth element in the state vector, which represents the three parameters E, V_i, and t_i. By doing this, the EKF algorithm only measures glucose, E, V_i, and t_i. This is because we mainly care about the change in glucose levels and parameters, so we choose to decrease computational cost by only measuring four states.

Additionally, we set the `InitialState` vector to be the initial values that were used in [21]. However, the last three initial values are offset from the true values slightly because they are the parameters we hope to estimate using joint EKF. Therefore, the initial values are

$$[200, 200, 12000, 0.1, 0.2, 0.1, 0.13, 15, 90].$$

In other words, we set the plasma insulin to be 200 mU/min, remote insulin to be 200 mU/min, glucose to be 12000 mU/min, the first stage linear filter to be 0.1 mU/min^2, the second stage linear filter to be 0.2 mU/min^2, the third stage linear filter to be 0.1 mU/min^2, the exchange rate for insulin between remote and plasma compartments to be 0.13 liters/min, the insulin vol-

ume to 15 liters, and the time for remote insulin degradation to be 90 minutes.

The MATLAB algorithm also has the capability to add an additional input to represent measurement and process noise. For this example, we let the measurement noise function be an additive, so

$$\hat{y}_k = \hat{x}_k + \hat{v}_k.$$

The measurement vector $\hat{v}_k$ has the same dimensions as the state vector $\hat{x}_k$. We set $\hat{v}_k$ to be a vector that consists of one value, 1.5. This gets input as the `MeasurementNoise`.

Interestingly, it was difficult and took quite a while to find a value that properly fit the system, ensuring that the code can run. There is no particular process used to determine this value. Instead, it is chosen through trial and error as well as intuition based on the `ProcessNoise` values.

For `ProcessNoise`, the input must be a square matrix with dimensions $n \times n$, where n represents the number of states there are. In this example, the values for `ProcessNoise` were chosen with some trial and error and intentionally on a similar scale as the `MeasurementNoise`. Some trial and error suggests that really small values for `ProcessNoise` of the three parameters work best. As a result, the input for `ProcessNoise` is

$$\begin{bmatrix} 0.02 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.04 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 \end{bmatrix}$$

Now, we have all the inputs to implement the MATLAB `extended-KalmanFilter` function. Rather than use real data, we create simulated data measurements that incorporate some random noise that is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which is created by adding a randomly-determined shift to values that were generated using the DEs. Then, we

call the `extendedKalmanFilter` function for each measured data point to update the state and covariance based on the previous estimation and the latest measured data point, which produce the "EKF estimate" values.

In **Figures 9.2, 9.3, 9.4, and 9.5**, we can see the results of applying joint EKF to the T2D system of DEs. As seen in the graphs, I use joint EKF to estimate one state (glucose) and three parameters (E, V_i, and t_i). This is because we only allow the joint EKF algorithm to measure these state and parameters.

The first graph, **Figure 9.2**, displays the estimation of parameter E over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint EKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

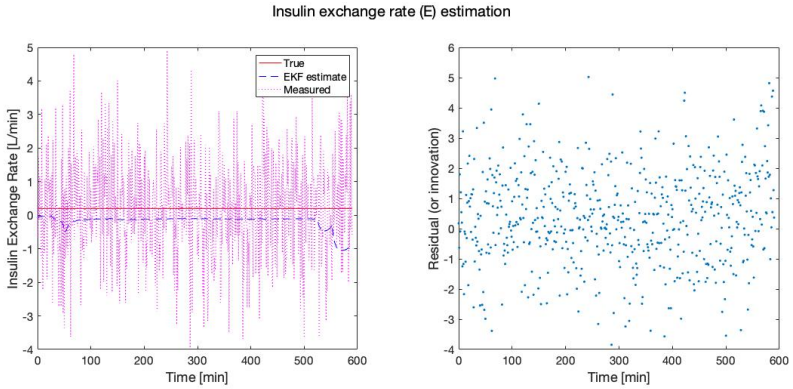

Figure 9.2 Results of joint EKF estimation for parameter E. **Left**: joint EKF estimation for E. The true solution for parameter E is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint EKF estimates by the dashed blue curve. In this example, joint EKF produces an inaccurate approximation for the true solution, diverging from the true solution at about 500 minutes. **Right**: measurement residuals (or innovation) of joint EKF estimations for E. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint EKF is not performing well because they generally have large magnitude and non-zero mean, though they do not exhibit atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation/Albers/

Looking at the results, it seems that the values estimated by the joint EKF algorithm (in blue dashed line) are not accurate at fitting the parameter E. For parameters, we want them to converge to the true value, which does not seem to be the case for E. In fact, the EKF estimate seem to be mostly unchanging until about 500 minutes, where it looks like it diverges.

Analyzing the EKF estimates for parameter E, the average estimate is -0.1869 and the median is -0.1141. Given that the true value of E is 0.2, these estimates are inaccurate. In fact, this parameter represents the exchange rate for insulin between the remote compartment and the plasma compartment. Consequently, a negative value is not physically possible. This suggests that there must be some error in the algorithm that is producing these negative values.

Furthermore, the measurement residuals suggest poor estimation. Looking at the right graph in **Figure 9.2**, we see that the residuals range between -4 and 6 centered around a value between 0 and 1. Considering the actual value of E is 0.2, these residuals seem to have a large range, which indicates large errors. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. Since the estimates have large errors and the estimates do not converge, the joint EKF algorithm does not accurately estimate the parameter E.

The second graph, **Figure 9.3**, displays the estimation of parameter V_i over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint EKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

Looking at the results, it seems that the values estimated by the EKF algorithm (in blue dashed line) are close to true value, which is 11. In fact, the average estimate for this parameter is 11.1664 and the median is 11.1507. Like E, we want the estimates for V_i to converge. However, it does not seem to be the case from the results in **Figure 9.3**, though the estimates do not diverge like they do for E.

Looking at the measurement residuals, the EKF estimates seem to stay within a somewhat large range. From the right graph in **Figure 9.3**, we see that the residuals range between -6 and 6 centered around 0 (with the

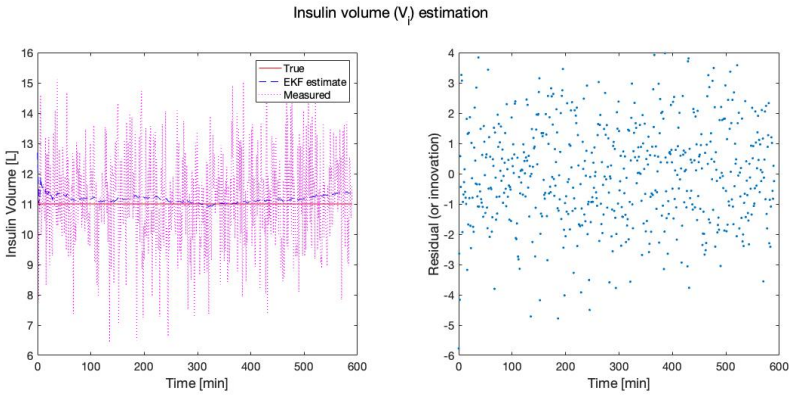

Figure 9.3 Results of joint EKF estimation for parameter V_i. **Left**: joint EKF estimation for V_i. The true solution for parameter V_i is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint EKF estimates by the dashed blue curve. In this example, joint EKF does not seem to sufficiently approximate for the true solution since the estimates do not converge. **Right**: measurement residuals (or innovation) of joint EKF estimations for V_i. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint EKF is not performing well because they generally have large magnitude, though they do seem to have zero-mean and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation/Albers/

exception with a couple outliers). Considering the actual value of V_i is 11, these residuals seem to have a fairly large range, suggesting significant errors. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. However, since the estimates have large errors and the estimates do not converge, the joint EKF algorithm does not accurately estimate the parameter V_i.

The third graph, **Figure 9.4**, displays the estimation of parameter t_i over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint EKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

Looking at the results, it seems that the values estimated by the EKF algorithm (in blue dashed line) are accurately fitting t_i. The estimates for this parameter seem to converge to the true value 100. Looking at the numerical

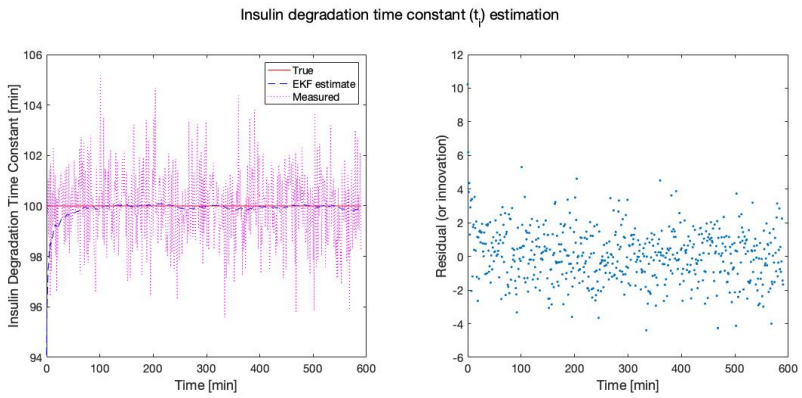

Figure 9.4 Results of joint EKF estimation for parameter t_i. **Left**: joint EKF estimation for t_i. The true solution for parameter t_i is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint EKF estimates by the dashed blue curve. In this example, joint EKF produces an accurate approximation for the true solution, converging to the true solution (100). **Right**: measurement residuals (or innovation) of joint EKF estimations for t_i. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint EKF is performing well because they have relatively small magnitude, zero mean, and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation/Albers/

estimates, the average estimate is 99.8770 and the median is 99.9601.

Additionally, looking at the measurement residuals in the right graph of **Figure 9.4**, we see that the residuals range between -6 and 6 centered around 0. Considering the actual value of t_i is 100, these residuals are small, indicating small errors. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. Thus, since the estimates have small errors and the estimates converge, the joint EKF algorithm accurately estimates the parameter t_i.

The last graph, **Figure 9.5**, displays the change in glucose levels in milligrams per liter over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint EKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

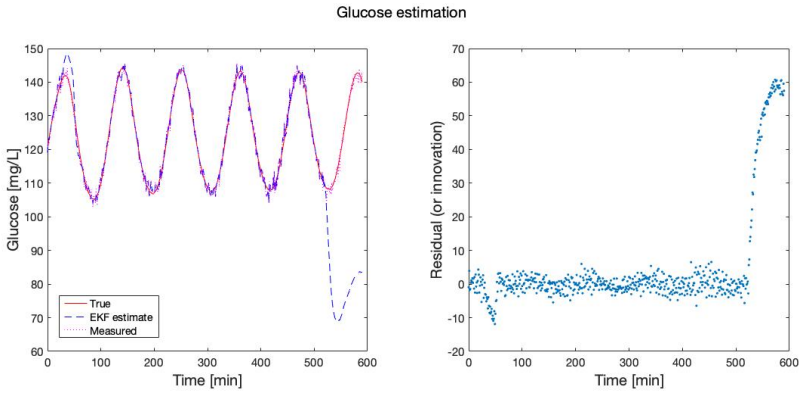

Figure 9.5 Results of joint EKF estimation for glucose. **Left**: joint EKF estimation for glucose. The true solution for the glucose state is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint EKF estimates by the dashed blue curve. In this example, joint EKF produces an inaccurate approximation for the true solution, suggesting that there is some error due to the sudden change in trend after about 500 minutes. **Right**: measurement residuals (or innovation) of joint EKF estimations for glucose. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint EKF is not performing well because they have an unusual spike after around 500 minutes, indicating that there must be some error. With further investigation, we find that there is error propagating after just a few time steps, causing the joint EKF algorithm to produce complex estimates. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Extended_KFs/Joint_Estimation/Albers/

Looking at the results, it seems that the values estimated by the EKF algorithm (in blue dashed line) are not accurate at predicting the glucose levels, though it seems to smoothing the data. In fact, after about 500 minutes, the joint EKF algorithm seems to produce estimations that suggest an inverse trend of what is expected, deviating from previous estimations that seem to follow the ultradian oscillations. To add to that, the measurement residuals, as seen in the right graph of **Figure 9.5**, exhibit an oscillatory pattern, which is not ideal.

With further investigation, we find that the joint EKF algorithm is producing complex estimates for glucose as well as the three parameters. **Table 9.1** indicates the parameters, their true values, the average estimate produced by the joint EKF algorithm, and the median estimate produced by the joint

Parameter	True Value	Mean joint EKF Estimate	Median joint EKF Estimate
E	0.2	$-0.1869 + 0.0004i$	$-0.1141 + 0.0034i$
V_i	11	$11.1664 - 0.0000i$	$11.1507 + 0.0002i$
t_i	100	$99.8770 - 0.0000i$	$99.9601 + 0.0003i$

Table 9.1 Parameter Estimates using joint EKF algorithm.

EKF algorithm. These complex values are concerning because physiological values should not be complex given that they represent real, positive values. One cause could be error propagation. Recall from **chapter 3**, the EKF algorithm uses first-order linearization to approximate nonlinear systems, such as this T2D model and many other biological models. Consequently, for high dimensional state spaces, error gets propagated after just a few time steps.

As mentioned earlier in **chapter 2**, [3] also encounters issues when applying Kalman filtering to the same T2D model. In that study, their issue is related to positivity, which is corrected by enforcing positivity. This means that if the algorithm produces negative values, the next iteration of points is generated using only the real part of the value. Clearly, numerical issues arise when applying these filters to dynamical systems like this T2D model that require forcing external constraints, such as positivity.

However, it is worth pursuing another type of Kalman filter, particularly the Unscented Kalman filter. In the following chapter, I use joint UKF to estimate states and parameters for this T2D model.

Chapter 10

Joint UKF State and Parameter Estimation

In order to implement joint UKF to estimate state and parameter values, I use MATLAB's pre-existing `unscentedKalmanFilter` function. Recall that this function takes as input a function `StateTransitionFcn`, a function `MeasurementFcn`, a vector `InitialState`, a vector or matrix `MeasurementNoise`, and a square matrix `ProcessNoise`.

`StateTransitionFcn` calculates the state vector of the system at time step k given the state vector at time step $k-1$. `MeasurementFcn` calculates the output measurement vector of the system at time step k given the sate vector at time step k. `InitialState` represents the initial state values based on the user's knowledge of the system. Therefore, it is a vector with length n, where n represents the number of states in the system. `MeasurementNoise` represents the measurement noise covariance. It is a scalar when `HasAdditiveMeasurementNoise` is set to true and a matrix when `HasAdditiveMeasurementNoise` is set to false. If it is a matrix, then its dimensions are $v \times v$, where v represents the number of measurement noise terms. `ProcessNoise` represents the process noise covariance, and it is a square matrix with dimensions $w \times w$, where w represents the number of process noise terms.

Using the pre-existing MATLAB function `unscentedKalmanFilter`, I estimate states and parameters for the same biological model detailed in **chapter 7** and used in the last chapter, **chapter 10**.

The code corresponding to this chapter can be found in the following GitHub repository: https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/Joint_Estimation.

10.1 Type 2 Diabetes Physiological Model

Recall that the system consists of six ordinary differential equations (ODEs) that represent the states and three ODEs that represent the parameters. The three parameters we estimate are the following: exchange rate for insulin between remote compartments and plasma compartments E, insulin volume V_i, and time of remote insulin degradation t_i.

Again, this system of ODEs is taken from the equations David J. Albers used to model glucose/insulin through MATLAB [5]. The nine ODEs are as

follows:

$$\begin{aligned}
\frac{dI_p}{dt} &= \frac{R_m}{1 - exp(\frac{-G}{V_g C_1} + a_1)} - E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_p}{t_p} \\
\frac{dI_i}{dt} &= E\left(\frac{I_p}{V_p} - \frac{I_i}{V_i}\right) - \frac{I_i}{t_i} \\
\frac{dG}{dt} &= \frac{R_g}{1 + exp(\frac{0.29h_3}{V_p - 7.5})} + I_g - U_b\left(1 - exp\left(\frac{-G}{C_2 V_g}\right)\right) \\
&\quad - \frac{90}{1 + exp(-1.772 \log\left(I_i\left(\frac{1}{V_i} + \frac{1}{Et_i}\right)\right) + 7.76)} + 4 \\
\frac{dh_1}{dt} &= \frac{3(I_p - h_1)}{t_d} \\
\frac{dh_2}{dt} &= \frac{3(h_1 - h_2)}{t_d} \\
\frac{dh_3}{dt} &= \frac{3(h_2 - h_3)}{t_d} \\
\frac{dE}{dt} &= 0 \\
\frac{dV_i}{dt} &= 0 \\
\frac{dt_i}{dt} &= 0.
\end{aligned}$$

The different constants that appear in the system of ODEs can be found in **Appendix B**. As mentioned in the previous chapter, since we are estimating the three parameters, they are passed through the filter as part of the state vector rather than as constants. The true values for these three parameters are still listed in **Appendix B** for reference. The other constants were taken from [6].

This set of nine ODEs are then used to create the state vector $\hat{x}_k$, where

the state vector is

$$\hat{x}_k = \begin{bmatrix} I_p \\ I_i \\ G \\ h_1 \\ h_2 \\ h_3 \\ E \\ V_i \\ t_i \end{bmatrix}.$$

In order to determine the states based on the system of ODEs, we use the MATLAB ODE-solver called `ode45`, which takes in a DE as input and returns the solution to the DE. Consequently, when applying `ode45` to the T2D model we get the state vector $\hat{x}_k$.

With the state vector for the glucose-insulin physiology system, we can create the `StateTransitionFcn` and the `MeasurementFcn` that will be fed as inputs into the `unscentedKalmanFilter` MATLAB algorithm. We set the `MeasurementFcn` to output the third element in the state vector, which represents glucose, as well as the seventh, eighth, and ninth element in the state vector, which represents the three parameters E, V_i, and t_i. By doing this, the UKF algorithm only measures glucose, E, V_i, and t_i. This is because we mainly care about the change in glucose levels and parameters, so we choose to decrease computational cost by only measuring four states.

Additionally, we set the `InitialState` vector to be the initial values that were used in [21]. However, the last three initial values are offset from the true values slightly because they are the parameters we hope to estimate using joint UKF. Therefore, the initial values are

$$[200, 200, 12000, 0.1, 0.2, 0.1, 0.13, 15, 90].$$

In other words, we set the plasma insulin to be 200 mU/min, remote insulin to be 200 mU/min, glucose to be 12000 mU/min, the first stage linear filter to be 0.1 mU/min^2, the second stage linear filter to be 0.2 mU/min^2, the third stage linear filter to be 0.1 mU/min^2, the exchange rate for insulin between remote and plasma compartments to be 0.13 liters/min, the insulin volume to 15 liters, and the time for remote insulin degradation to be 90 minutes.

The MATLAB algorithm also has the capability to add an additional input to represent measurement and process noise. For this example, we let the measurement noise function be an additive, so

$$\hat{y}_k = \hat{x}_k + \hat{v}_k.$$

The measurement vector $\hat{v}_k$ has the same dimensions as the state vector $\hat{x}_k$. We set $\hat{v}_k$ to be a vector that consists of one value, 1.5. This gets input as the `MeasurementNoise`.

Interestingly, it took quite a while to find a value that properly fit the system, ensuring that the code can run. There is no particular process used to determine this value. Instead, the `MeasurementNoise` is chosen through trial and error as well as intuition based on the `ProcessNoise` values.

For `ProcessNoise`, the input must be a square matrix with dimensions $n \times n$, where n represents the number of states there are. In this example, the values for `ProcessNoise` are chosen with some trial and error and intentionally on a similar scale as the `MeasurementNoise`. Some trial and error suggests that really small values for `ProcessNoise` of the three parameters work best. As a result, the input for `ProcessNoise` was

$$\begin{bmatrix} 0.02 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.04 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.0001 \end{bmatrix}$$

Now, we have all the inputs to implement the MATLAB `unscented-KalmanFilter` function. Rather than using real data, we create simulated data measurements that incorporate some random noise that is based on clean simulated data that represents the "true" values. The noisy simulated data represents the "measured" data, which is created by adding a randomly-determined shift to values that were generated using the DEs. Then, we call the `unscentedKalmanFilter` function for each measured data point to update the state and covariance based on the previous estimation and the

latest measured data point, which produce the "UKF estimate" values.

In **Figures 10.1, 10.2, 10.3, and 10.4**, we can see the results of applying joint UKF to the T2D system of DEs. As seen in the graphs, I use joint UKF to estimate one state (glucose) and three parameters (E, V_i, and t_i). This is because we only allow the joint UKF algorithm to measure these state and parameters.

The first graph, **Figure 10.1**, displays the estimation of parameter E over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint UKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

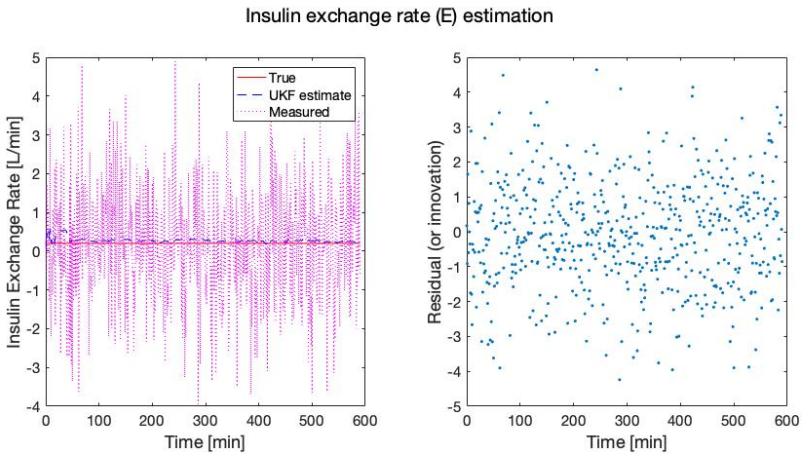

Figure 10.1 Results of joint UKF estimation for parameter E. **Left**: joint UKF estimation for E. The true solution for parameter E is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint UKF estimates by the dashed blue curve. In this example, joint UKF produces an accurate approximation for the true solution, converging to the true solution (0.2). **Right**: measurement residuals (or innovation) of joint UKF estimations for E. Note: measurement residual is defined in **Equation 5.3**. Interestingly, the measurement residuals suggest that joint UKF is not performing well because they generally have large magnitude, though they do have zero mean and lack atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/Joint_Estimation

Looking at the results, it seems that the values estimated by the joint UKF algorithm (in blue dashed line) are accurate at fitting the parameter E. For parameters, we want them to converge to the true value, which seems to be true for E. In fact, the average estimate for this parameter is 0.2767 and the median is 0.2670, which is close to the true value 0.2.

Interestingly, the measurement residuals suggest poor estimation. Looking at the right graph in **Figure 10.1**, we can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which it does. Furthermore, we see that the residuals range between -5 and 5 centered around 0. Considering the actual value of E is 0.2, the range of these residuals is fairly large, which indicates that there are large errors. This does not immediately make sense. However, further investigation provides better insight as to why this is happening.

After looking into the measurement residuals of the estimates for parameter E, we find that the minimum value is -4.2398 and the maximum value is 4.6426 while the average value for the residuals is -0.0744 and the median is -0.0688. Additionally, almost half of the residuals (47.72%) have a magnitude greater than 1, which is significant given that the true value of E is 0.2. Therefore, although the estimates for E seem to converge and there is a lack of autocorrelation, the joint UKF algorithm produces large errors when estimating the parameter E.

The second graph, **Figure 10.2**, displays the estimation of parameter V_i over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint UKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

Looking at the results, it seems that the values estimated by the UKF algorithm (in blue dashed line) are mediocre. Like E, we want the estimates for V_i to converge. However, it does not seem to be the case from the results in **Figure 10.2**. Instead, the estimates are close to the true value of 11 but showing no indication of converging. To be more precise, the average estimate for this parameter is 11.1068 and the median is 11.0898.

From the measurement residuals, the UKF estimates seem to be contained within a somewhat large range. From the right graph in **Figure 10.2**, we see

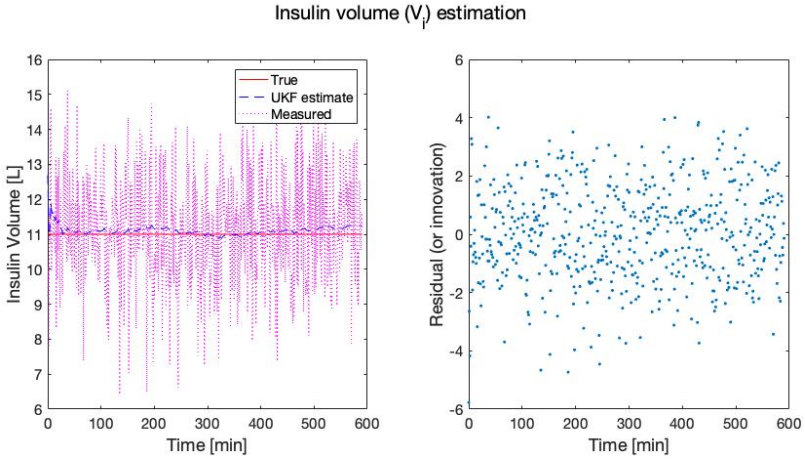

Figure 10.2 Results of joint UKF estimation for parameter V_i. **Left**: joint UKF estimation for V_i. The true solution for parameter V_i is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint UKF estimates by the dashed blue curve. In this example, joint UKF does not seem to sufficiently approximate for the true solution since the estimates do not converge, though they seem to be close to the true solution (11). **Right**: measurement residuals (or innovation) of joint UKF estimations for V_i. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint UKF is not performing well because they generally have large magnitude, though they do seem to have zero-mean and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/Joint_Estimation

that the residuals range between -4 and 4 centered around 0. Considering the actual value of V_i is 11, these residuals seem to have a fairly large range, suggesting significant errors. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. However, since the estimates have large errors and the estimates do not converge, the joint UKF algorithm does not accurately estimate the parameter V_i.

The third graph, **Figure 10.3**, displays the estimation of parameter t_i over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint UKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth.

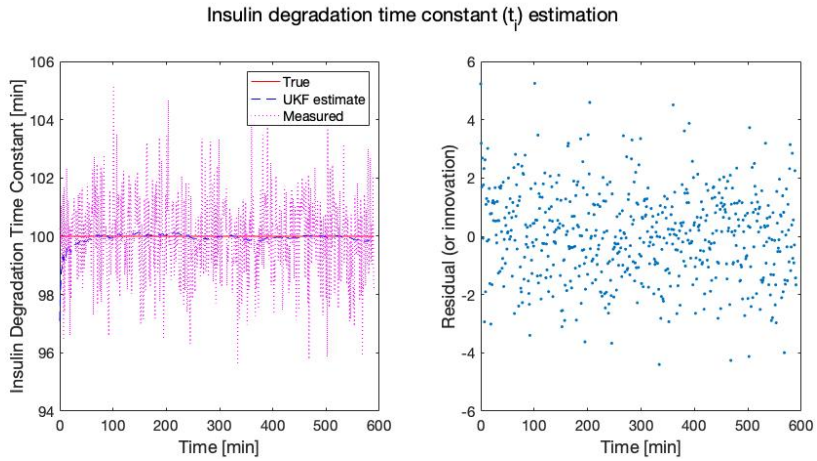

Figure 10.3 Results of joint UKF estimation for parameter t_i. **Left**: joint UKF estimation for t_i. The true solution for parameter t_i is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint UKF estimates by the dashed blue curve. In this example, joint UKF produces an accurate approximation for the true solution, converging to the true solution (100). **Right**: measurement residuals (or innovation) of joint UKF estimations for t_i. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint UKF is performing well because they have a relatively small magnitude, zero mean, and no atuocorrelation. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/Joint_Estimation

Looking at the results, it seems that the values estimated by the UKF algorithm (in blue dashed line) are accurate at fitting t_i. The estimates for this parameter seem to converge to the true value 100 with an average estimate of 99.9384 and median estimate of 99.9726.

Additionally, looking at the measurement residuals in the right graph of **Figure 10.3**, we see that the residuals range between -6 and 6 centered around 0. Considering the actual value of t_i is 100, these residuals are small, indicating small errors. We can infer that the residuals lack autocorrection if they appear randomly distributed and without a pattern, which they do. Thus, since the estimates have small errors and the estimates converge, the joint UKF algorithm accurately estimates the parameter t_i.

The last graph, **Figure 10.4**, displays the change in glucose levels in

milligrams per liter over 600 minutes as well as the corresponding measurement residuals (or innovation), which help further assess the performance of the joint UKF algorithm. The measured data (in magenta dotted line) is somewhat noisy while the true data (in red solid line) is smooth. Looking at the results, it seems that the values estimated by the UKF algorithm (in blue dashed line) are accurate at predicting the glucose levels as well as smoothing the data.

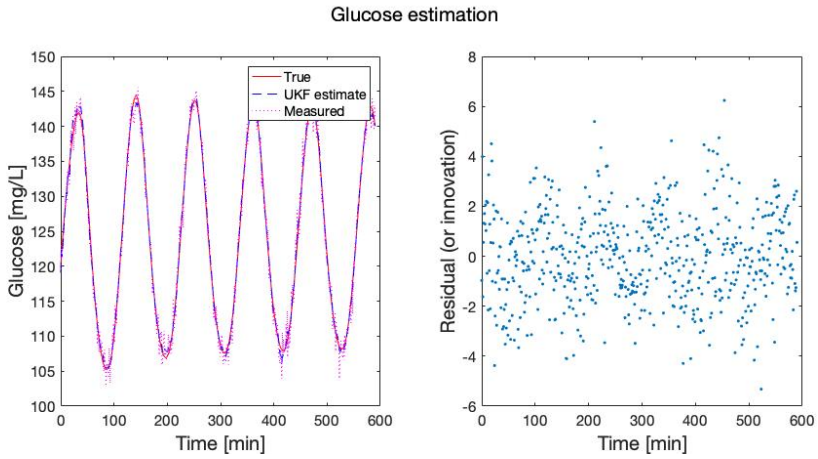

Figure 10.4 Results of joint UKF estimation for glucose. **Left**: joint UKF estimation for glucose. The true solution for the glucose state is depicted by the solid red line, the measured data (with noise) by the dotted magenta curve, and the joint UKF estimates by the dashed blue curve. In this example, joint UKF produces a reasonable approximation for the true solution, accurately estimating and smoothing the data. **Right**: measurement residuals (or innovation) of joint UKF estimations for glucose. Note: measurement residual is defined in **Equation 5.3**. The measurement residuals suggest that joint UKF is not performing well because they exhibit an oscillatory pattern, which suggests poor state estimation and possible autocorrelation in the data. Unfortunately, when the algorithm is run with a larger time interval (greater than 600 minutes), it throws a sigma point calculation error. With a little more investigation, it seems that the error stems from issues with updating the covariance matrix and, consequently, the Kalman gain. Code to reproduce figure available at https://github.com/CassidyLe98/Thesis_KalmanFilters/tree/master/Unscented_KFs/Albers/Joint_Estimation

Unfortunately, looking at the measurement residuals in the right graph of **Figure 10.4**, it seems that the residuals exhibit an oscillatory pattern, which suggests poor state estimation and possible autocorrelation in the

Parameter	True Value	Mean joint UKF Estimate	Median joint UKF Estimate
E	0.2	0.2767	0.2670
V_i	11	11.1068	11.0898
t_i	100	99.9384	99.9726

Table 10.1 Parameter Estimates using joint UKF algorithm.

data. Though, the range of the residuals is quite small with respect to the values of glucose, staying contained within a range of -6 and 6. Interestingly, when the code is run with a larger time interval (greater than 600 minutes), the algorithm throws an error that indicates a sigma point calculation error. This seems to be stemming from the updated covariance matrix and, thus, the Kalman gain matrix.

Although joint UKF produces estimates that seem to closely follow the true values for both states and parameters, the measurement residuals suggest that there needs to be a deeper investigation of what the algorithm is outputting at each time step. Just by a quick look at **Figure 10.4** and **Table 10.1**, which indicates the parameters, their true values, the average estimate produced by the joint UKF algorithm, and the median estimate produced by the joint UKF algorithm, one may conclude that joint UKF accurately estimates states and parameters. However, a deeper look into the covariance matrices produced indicates that the algorithm may not be performing as well as one may initially think.

As mentioned earlier in **chapter 2** as well as at the end of **chapter 9**, the study in [3] also encounters issues when applying UKF to the same T2D model. In that study, their issue is related to positivity, which is resolved by enforcing positivity. In other words, if the algorithm produces negative values, the next iteration of points is generated using only the real part of the value. Clearly, numerical issues arise when applying Kalman filters to systems like this T2D model that require forcing external constraints, such as positivity.

Chapter 11

Discussion

11.1 Conclusion

To summarize, in this paper, we study how Kalman filtering can be used to fit states and parameters of biological models to a data set. In particular, we explore two types of Kalman filters - the extended Kalman filter (EKF) and the unscented Kalman filter (UKF). Using a couple different methods of implementation, we estimate states as well as parameters for linear and nonlinear systems, including a type 2 diabetes model that is highly nonlinear.

For estimating states of linear systems, EKF seems to perform just as well as UKF. This is reasonable because EKF would not have to approximate the system, like it does for nonlinear systems. For estimating states of nonlinear systems, EKF does not perform as well as UKF, as seen through the results in **chapter 5** and **chapter 7**. UKF's superior performance is likely due to the fact that EKF approximates the nonlinear system by linearizing it to the first-order, which may propagate error over time. With this in mind, we suggest the use of UKF over EKF when estimating states for nonlinear systems. Therefore, because biological models tend to be highly nonlinear, we believe it is best to use UKF when estimating the states of biological models.

In addition to estimating states, we also estimate states in conjunction with parameters for both linear and nonlinear models. We do so using both joint EKF and joint UKF. For estimating states and parameters of linear systems, joint EKF performs well, as seen in **chapter 9**. However, for

nonlinear systems, joint EKF seems to propagate error after just a few time steps. In fact, the results of applying joint EKF to the type 2 diabetes model indicate that joint EKF produces complex values, which is noted in **chapter 9**. Unfortunately, although joint UKF seems to estimate states and parameters better than joint EKF, there are still issues that arise.

More specifically, the results of applying joint UKF to the type 2 diabetes model suggest that more attention is needed when running each time step. To elaborate, in **chapter 10**, we note that the algorithm seems to fail after a certain number of time steps (around 600). With further inspection, this complication stems from how the covariance matrix is updated and what the resulting covariance matrix looks like after each time step. This suggests that it may be best to use a different approach to estimate states and parameters for nonlinear systems, such as dual estimation in conjunction with Kalman filtering or even Monte Carlo methods.

Interestingly, the numerical challenges that we encounter in this paper are not unique to our implementations of Kalman filtering. In [3], their implementation of UKF for the same T2D model exhibits numerical issues related to positivity, which is corrected by enforcing positivity. In other words, if the algorithm produces negative values, the next iteration of points is generated using only the real part of the value. Clearly, numerical issues arise when applying these filters to dynamical systems like this T2D model, which require problem-dependent adjustments for each implementation.

Additionally, throughout this research, we greatly struggle to determine values for the process noise and measurement noise that worked well with each system. We believe there is a balance and threshold that exists between the process noise and measurement noise that varies based on the system. As a result, one must be careful when choosing these values. It becomes easier with more experience and a stronger foundational understanding of dynamical systems and probability. This is also an issue when determining what constants to use for the UKF algorithm, which are detailed in **Table 6.2**. Fortunately, the MATLAB `unscentedKalmanFilter` algorithm can automatically set these constant values, so the main concern is the process and measurement noise.

11.2 Future Work

It is important to note that this paper uses simulated data, which historically is cleaner and behaves better than real data sets. Therefore, the results and conclusions in this paper may not be applicable to real data sets. In the future, it would be good to see how these algorithms and systems perform on real, experimentally-based data sets. It is likely that this will cause more complications to overcome, but it would provide valuable insight as to the performance of Kalman filtering on biological models.

Another valuable extension of this project would be to apply dual EKF or dual UKF to the type 2 diabetes biological model. As mentioned earlier in the **Conclusion** section, the results of this paper revealed that joint EKF propagates error when estimating states and parameters of the T2D model. Similarly, there were issues with the joint UKF algorithm that need further investigation. Due to time constraint, we were unable to use dual EKF or dual UKF. It may be possible that the dual estimation approach works better with biological models. Although dual estimation does not account for correlations between states and parameters, it does allow for tuning of the algorithm's meta-parameters (i.e. α, κ, and β). With the flexibility to tune α, we may be able to adjust the spread of the sigma points, which may produce better estimates.

For distant future work, it would be great to apply these Kalman filtering algorithms, especially UKF, to the T1D single-compartment model from [20]. This would be dependent on the results of using real, experimentally-based data sets as well as applying dual estimation to the T2D model. If there is some success in that research, then applying some of those state and parameter estimation techniques would be highly valuable for the T1D single-compartment model from [20].

Appendix A

Terminology

Antigen: Antigens are any foreign and potentially harmful substances.

Beta-cells: Beta- (or β-) cells are located in pancreatic islets of Langerhans. They are mainly responsible for producing insulin.

Dendritic cells: Dendritic cells (or DCs) process and present antigens for recognition by T-cells, which produces an immunogenic response from the T-cells.

Endocrine system: A system in the human body that consists of glands which secrete hormones directly into the circulator system. It does so in a feedback loop process. By releasing hormones, the endocrine system regulates various functionalities of the body, some of which include metabolism, growth and development, tissue function, sexual function, reproduction, sleep, and mood.

Insulin: Insulin is a hormone that binds with cells, allowing the cells to absorb glucose. This represents the enzymatic 'lock-and-key' description.

Macrophages: Macrophages are a type of white blood cell in the immune system that engulf and digest foreign substances, microbes, cancer cells, and cellular debris through phagocytosis. When a macrophage encounters an antigen, it activates the T-cells by producing cytokines, or cellular signals.

Monocytes: Monocytes are the largest type of white blood cells. Initially,

they are found in the bloodstream and then enter body tissue after about two days. Once in body tissue, monocytes can differentiate into macrophages and dendritic cells.

Lymphocytes: Lymphocytes are a type of white blood cell that originate from stem cells in the bone marrow. They are one of the body's main types of immune cells. Two types of lymphocytes are B-cells and T-cells.

T-cells: T-cells are a type of lymphocyte that have a receptor on the cell surface, which provide them the ability to play a central role in the immune response. Initially, T-cells are classified as naive T-cells. Once a DC presents an antigen to a T-cell, the T-cell becomes immunogenic and the immune response is activated.

Appendix B

Constants for T2D Model

Constant	Value	Units	Representation
R_m	209	$\frac{mU}{min}$	linear constant affecting insulin secretion
a_1	6.67		exponential constant affecting insulin secretion
α	7.5		exponential constant affecting insulin dependent glucose utilization
β	1.77		exponent affecting insulin dependent glucose utilization
V_p	3	L	plasma volume
V_i	11	L	insulin volume
V_g	10	L	glucose space
E	0.2	$\frac{L}{min}$	exchange rate for insulin between remote/plasma compartments
t_p	6	min	time constant for plasma insulin degradation
t_i	100	min	time constant for remote insulin degradation
t_d	36	min	time delay between plasma insulin degradation and glucose production
C_1	300	$\frac{mg}{L}$	exponential constant affecting insulin secretion
C_2	144	$\frac{mg}{L}$	exponential constant affecting insulin independent glucose utilization
C_3	100	$\frac{mg}{L}$	linear constant affecting insulin dependent glucose utilization
U_b	72	$\frac{mg}{min}$	linear constant effacing insulin independent glucose utilization
U_0	4	$\frac{mg}{min}$	linear constant effacing insulin dependent glucose utilization
U_m	92	$\frac{mg}{min}$	linear constant effacing insulin independent glucose utilization
R_g	180	$\frac{mg}{min}$	linear constant effacing insulin independent glucose utilization
I_g	216	$\frac{mg}{min}$	exogenous glucose delivery rate (feeding pattern)

Bibliography

[1] 2020. Numerical differentiation. https://en.wikipedia.org/wiki/Numerical_differentiation.

[2] ADA2019. 2019. Statistics about diabetes. http://www.diabetes.org/resources/statistics/statistics-about-diabetes.

[3] Albers, David J., Matthew Levine, Bruce Gluckman, Henry Ginsberg, George Hripcsak, and Lena Mamykina. 2017. Personalized glucose forecasting for type 2 diabetes using data assimilation. *PLOS Computational Biology* 13(4):1–38. doi:10.1371/journal.pcbi.1005232. https://doi.org/10.1371/journal.pcbi.1005232.

[4] Albers, David J, Matthew E Levine, Andrew Stuart, Lena Mamykina, Bruce Gluckman, and George Hripcsak. 2018. Mechanistic machine learning: how data assimilation leverages physiologic knowledge using bayesian inference to forecast the future, infer the present, and phenotype. *JAMIA* 25(10):1392–1401. doi:10.1093/jamia/ocy106. https://doi.org/10.1371/journal.pone.0048058.

[5] Albers, DJ. 2013. Matlab code for glucose/insulin modeling. https://github.com/djalbers/glucose_dynamics_modeling.

[6] Albers, D.J., George Hripcsak, and Michael Schmidt. 2012. Population physiology: Leveraging electronic health record data to understand human endocrine dynamics. *PLOS ONE* 7(12):1–13. doi:10.1371/journal.pone.0048058. https://doi.org/10.1371/journal.pone.0048058.

[7] Albert, Matthew L., S Frieda A. Pearce, Loise M. Francisco, Birthe Sauter, Pampa Roy, Roy L. Silverstein, and Nina Bhardwaj. 1998. Immature dendritic cells phagocytose apoptotic cells via ?v?5 and cd36, and cross-present antigens to cytotoxic t lymphocytes. *JEM* 188(7):1359–1368. doi:10.1084/jem.188.7.1359. https://doi.org/10.1084/jem.188.7.1359.

[8] Bolviken, Erik, Nils Christophersen, and Geir Storvik. 1998. Linear dynamical models, Kalman filtering and statistics. Lecture notes to IN-ST 259.

[9] Center, Columbia University Medical. 2017. Diabetes app forecasts blood sugar levels. https://www.sciencedaily.com/releases/2017/04/170427141732.htm.

[10] Gove, J.H, and D.Y. Hollinger. 2006. Application of a dual unscented Kalman filter for simultaneous state and parameter estimation in problems of surface-atmosphere exchange. *Journal of Geophysical Research: Atmospheres* 111(D8). doi:10.1029/2005JD006021. https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2005JD006021.

[11] Kitagawa, Genshiro. 1987. Non-Gaussian State Space Modeling of Nonstationary Time Series. *Journal of the American Statistical Association* 82(400):1032–1041. doi:10.1080/01621459.1987.10478534.

[12] Kleinbauer, Rachel. 2004. Kalman filtering implementation with Matlab. *Universitat Stuttgart: Institute of Geodesy* .

[13] Man, Chiara Dalla, Robert A. Rizza, and Claudio Cobelli. 2008. Meal simulation model of glucose-insulin system. *IEE Transactions on Biomedical Engineering* 54(10):1740–49. doi:10.1109/TBME.2007.893506.

[14] Mandal, Ananya MD. 2019. Insulin gene. https://www.news-medical.net/health/Insulin-Gene.aspx.

[15] Marée, Athanasius, Richard Kublik, Diane T. Finegood, and Leah Edelstein-Keshet. 2006. Modelling the onset of type 1 diabetes: Can impaired macrophage phagocytosis make the difference between health and disease? *Philosophical transactions Series A, Mathematical, physical, and engineering sciences* 364:1267–82. doi:10.1098/rsta.2006.1769.

[16] Mathworks. 2019. Extended and Unscented Kalman Filter Algorithms for Online State Estimation. https://www.mathworks.com/help/control/ug/extended-and-unscented-kalman-filter-algorithms-for-online-state-estimation.html#bvgiw03.

[17] ———. 2019. unscentedKalmanFilter. https://www.mathworks.com/help/control/ref/unscentedkalmanfilter.html#bvf20x4.

[18] Pietrangelo, Ann. 2018. What are the different types of diabetes? https://www.healthline.com/health/diabetes/types-of-diabetes#causes.

[19] Rhudy, Matthew B., Roger A Salguero, and Keaton Holappa. 2017. A Kalman filtering tutorial for undergraduate students. *International Journal of Computer Science & Engineering Survey* 8(1):1–18. doi:10.5121/ijcses.2017.8101.

[20] Shtylla, Blerta, Marissa Gee, An Do, Shahrokh Shabahang, Leif Eldevik, and Lisette de Pillis. 2019. A Mathematical Model for DC Vaccine Treatment of Type I Diabetes. *Frontiers in Physiology* 10:1107. doi:10.3389/fphys.2019.01107. https://www.frontiersin.org/article/10.3389/fphys.2019.01107.

[21] Sturis, J., K.S. Polonsky, E. Mosekilde, and E. Van Cauter. 1991. Computer model for mechanisms underlying ultradian oscillations of insulin and glucose. *American Journal of Physiology-Endocrinology and Metabolism* 260(5):E801–E809. doi:10.1152/ajpendo.1991.260.5.E801.

[22] Wan, Eric A., and Rudolph Van Der Merwe. 2001. The Unscented Kalman Filter. In *Kalman Filtering and Neural Networks*, 221–280. Wiley.

[23] Wikle, Christopher, and L. Berliner. 2007. A bayesian tutorial for data assimilation. *Physica D: Nonlinear Phenomena* 230:1–16. doi:10.1016/j.physd.2006.09.017.