PID Control Simulation and Kalman Filter State Estimation of HIT-SI Injector Flux Circuit

Matthew Kraske

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Aeronautics & Astronautics

University of Washington

2014

Reading Committee:

Thomas R. Jarboe, Chair

Brian Nelson

Program Authorized to Offer Degree:
UW Aeronautics and Astronautics

University of Washington

Abstract

PID Control Simulation and Kalman Filter State Estimation of HIT-SI Injector Flux Circuit

Matthew Kraske

Chair of the Supervisory Committee:
Professor Thomas R. Jarboe
Aeronautics and Astronautics

In order to implement an optimal modern control system on the HIT-SI injector voltage and flux circuits, it is first necessary to apply state estimation techniques, allowing the physical system to be observed by the controller. To test these estimation methods prior to implementation on the experiment, a simulation must be developed which accurately represents the dynamics and behavior of the experiment. Kalman filter state estimation is implemented using a circuit dynamics model which yields observable state tracking with very low error. Extended Kalman filter estimation is implemented for circuit parameter estimation and for sine wave fitting but requires additional development.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

HIT-SI: the Helicity Injected Torus - Steady Inductive experiment at the University of Washington.

KALMAN FILTER: an algorithm which utilizes assumed process and measurement error to estimate system. states

PROCESS ERROR: the error introduced to system states during time evolution. This may be due to an incomplete model.

MEASUREMENT ERROR: the error introduced to the system measurements. This may be due to known inaccuracies in the measurement method.

LINEAR QUADRATIC REGULATOR: LQR, a control which works by seeking to minimize a cost function.

PULSE WIDTH MODULATION: PWM, a means of actuating a system where the amplitude of a driving force is held constant while the duration of the force application is varied.

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to University of Washington; his faculty mentor, Brian Nelson; and his colleagues in the controls discipline at UW. Without your assistance this work would not have been possible. He would also like to thank the physics department at the University of Washington. Your support has been appreciated.

DEDICATION

to my family

Marie, Will, Cindy, Rach, & Ben

Chapter 1
INTRODUCTION

1.1 The HIT-SI experiment

The Helicity Injected Torus-Steady Inductive (HIT-SI) experiment at the University of Washington is a spheromak current drive experiment. The research being done in this group seeks to better understand the dynamics of plasma in the spheromak configuration. Ultimately, the information gathered here may help develop feasible means of generating electricity through nuclear fusion. Unlike many other experiments aimed at unlocking fusion, HIT-SI seeks to inductively drive current in an electrodelss, steady-state fashion. This current drive is sought through utilizing a phenomenon known as magnetic helicity injection. Magnetic helicity is the tendency for magnetic fields to intertwine helically, as do the strands of a rope. While eventual steady-state operation is desirable, HIT-SI operates for short periods of time. These periods are called 'shots' and typically last between 2000 and 4000 microseconds.

The confinement volume can be visualized as a bow-tie rotated about its center knot. This can be seen in Figure 1.1. Both the magnetic helicity and the toroidal current are introduced through two jug-handle shaped injectors mounted to the sides of the machine, injector X and injector Y. These semi-toroidal injectors contain two separate sets of coils. The poloidal injector windings are used to induce a flux in the injector and as such are referred to as the 'flux coil'. The toroidal windings act as the primary of a transformer. These serve to induce a toroidal voltage within the injector and are referred to as the 'voltage coil'. The currents and fluxes generated by these injectors loop with the spheromak, driving the current. The inductance and impedance of the flux coil varies throughout the shot as the

core of the inductor changes from vacuum to a time-dependent concentration of plasma.

Generating the desired voltages and fluxes in the injectors takes large currents. To supply the necessary power, each injector has a corresponding power system consisting of a capacitor bank, an array of switching power amplifiers, a large capacitor, and the cabling needed to connect all of these parts. The switching power amplifiers (SPAs) have optically-isolated triggering signals, and use use H-bridge insulated gate bi-polar transistors (IGBTs). The SPAs can apply varying width positive and negative voltage pulses from the capacitor bank to the remainder of the circuit. These pulses are adjusted in length to control the circuit. This method of control is known as pulse width modulation (PWM). The capacitor is implemented to create an LC or 'tank' circuit that can be driven off-resonance to increase the ratio of coil current to switching power supply current while not being limited by the capacitor bank voltage. The circuit can be visualized as in Figure 1.2.

Since, the capacitor bank voltage is set prior to the shot, the flux and voltage coil circuits cannot be controlled with continually variable supply voltage. Thus, the aforementioned PWM method is implemented via the switching power amplifiers. The helicity injector's voltage and flux circuits are oscillating and can be controlled by slightly varying the length of the PWM voltage pulses applied to the injector coils.

A control algorithm has been developed to operate the switching power amplifiers via switching signals. This system is discussed in the subsequent section.

1.2 Control System Overview

In order for the experiment to achieve maximum performance, feedback control on the flux and voltage coils is desirable. The voltage circuit switching power amplifiers are presently driven by a pre-programmed signal while there is a Proportional Integral Derivative (PID) control regime instated on the flux power amplifiers. The control code resides and runs on single board microcontrollers in the room adjacent to the experiment (for added electro-magnetic isolation).

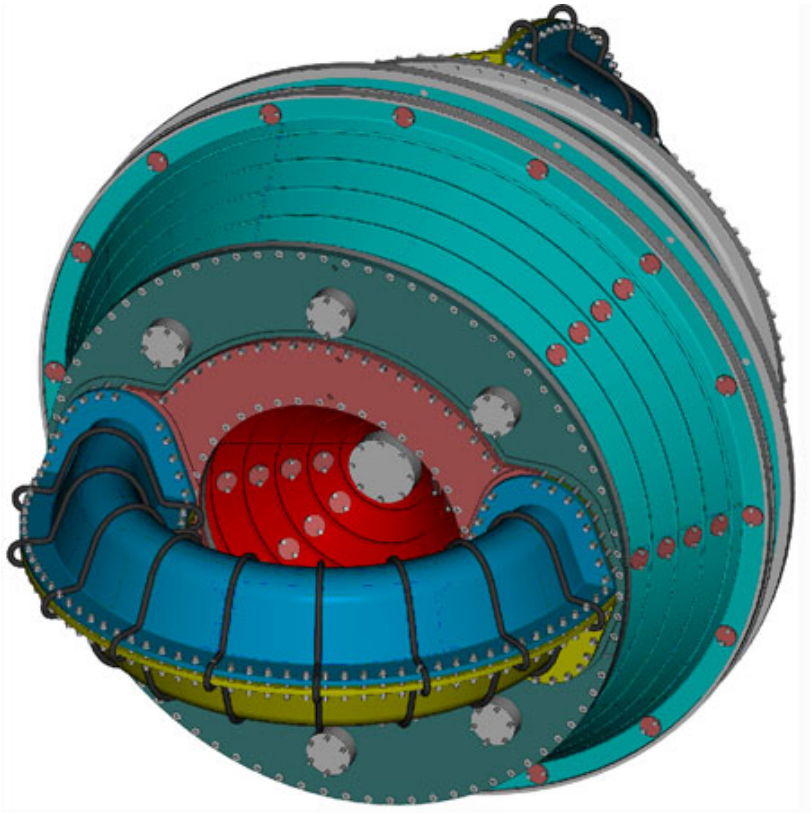

Figure 1.1: A CAD model of the HIT-SI confinement volume with helicity injectors and injector flux windings

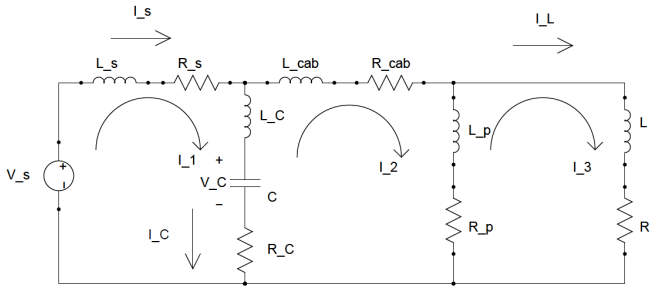

Figure 1.2: A Visualization of the Injector LC Circuit

1.2.1 Control Hardware

The single board microcontroller utilized on HIT-SI are Blackfin built by Analog Devices. Each device is equipped with several peripherals including Ethernet, GPIO (general purpose input-output) pins, and a single ADC (analogue to digital converter), as well as a number of hardware-level digital signal processing instructions that can be used to speed up operation. The main limitation of these devices is their single ADC. This requires additional Blackfins to be used wherever monitoring more than one channel is required. This will be discussed later. During a shot, the majority of the Blackfin device communication capabilities are locked out by masking all interrupts, to improve deterministic behavior of the control algorithm.

1.2.2 Control Algorithm

The flux circuit control algorithm uses a pre-programmed ramp stage paired with a feedback controlled stage. The pre-programmed stage runs for a predetermined length of time at the beginning of the shot. After a user defined number of oscillation cycles, the pre-programmed control ends, transitioning directly into the feedback control stage. This occurs around the time that plasma breaks down. The ADC on the flux circuit control Blackfin is used to measure and digitize injector flux for for feedback control.

The voltage circuit uses two unique pre-programmed control stages. The first stage runs during the vacuum period of the shot. When plasma breaks down, the second stage is activated for the remainder of the shot. The ADC on the voltage control Blackfin is used to monitor and digitize the current through the injector, allowing for the program switch to occur when plasma becomes present. (Later iterations of the HIT experiment utilize a single Blackfin for current detection in addition to one each for feedback control of the flux and voltage circuits.)

As discussed previously, the circuit behavior is controlled through a PWM method where the SPAs are used to switch the application of capacitor bank voltage to the in-

jector flux and voltage coils. As such, the controller output is an injector pulse width. The circuit is actually driven at 14.5 kHz, just below its natural resonance at 15.8 kHz as the ratio of flux coil to SPA current reduces to unity as the resonant frequency is approached from lower frequencies. (The ratio is less than unity above the resonant frequency.)

The feedback control algorithm presently in place on the flux circuit utilizes an unconventional "dual" PID controller to both eliminate DC injector flux offset and to reach target current gains. Each of these PID controllers is treated as a separate system and only has control over one half of the injector current waveform, either positive (above the mean) or negative (below the mean). This dual system can be visualized as two independent waves being controlled about target amplitudes (mirrored across the x-axis). It is depicted in Figure 1.3.

Over the duration of the shot, these individual PID controllers attempt to drive the deviation between measured injector flux and the demand (termed, DCP Plasma) to zero. This can again be visualized as driving the peak in Figure 1.3 to match the demand. These PID controllers operate alternately with each other on a common set of gains.

The PID algorithms used are:

$$PW = \frac{error^{-} \times G_p + error^{-}\, sum \times G_i + (error^{-} - error^{-}\, prev) \times G_d}{fp\, factor}$$

$$PW = \frac{error^{+} \times G_p + error^{+}\, sum \times G_i + (error^{+} - error^{+}\, prev) \times G_d}{fp\, factor}$$

where the terms are defined in table 3.1.

One of the unique aspects of this system is that the PID controllers operate completely independent of each other. The controller acting on the positive pulse widths only feeds back based on the positive pulse information and likewise for the negative controller.

In order to maximize the effectiveness of the HIT-SI plasma experiment, it is necessary to ensure that the control algorithms used to operate the experiment are capable of minimizing error and avoiding instabilities. Work in this area is ongoing. However, many areas where improvements can be made have been identified.

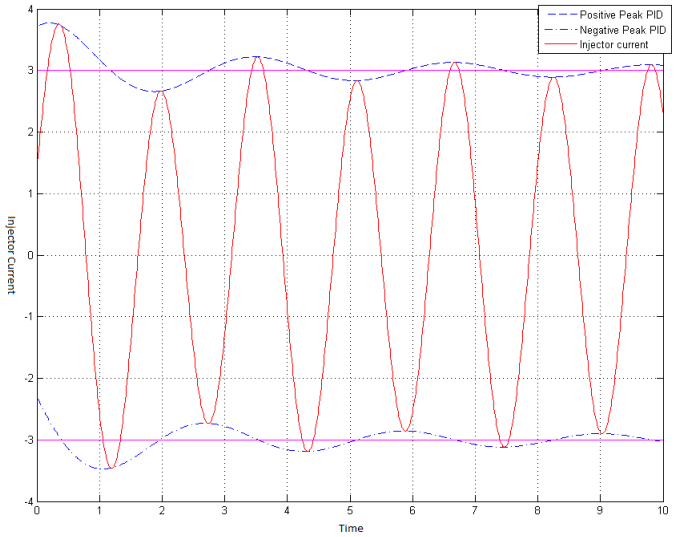

Figure 1.3: ”Dual” PID Control of a sine wave with target aplitudes (generated wave; for illustrative purposes only)

1.3 Algorithm Improvements

The existing algorithm and experiment system has been analyzed, and emphasis has been placed on identifying areas where modern control code can be implemented to improve control of the experiment. Many of these areas have been identified. One is the compensation for the drop in capacitor bank voltage toward the end of the shot. This research, however, focuses a different major improvement: the implementation of advanced filtering methods on the Blackfin ADC flux samples. The aim of this research is to make the feedback data more accessible while the experiment is running.

1.3.1 Kalman Filtering of ADC Samples

To properly drive the plasma within the confinement volume, it is necessary to induce flux via injectors. Achieving ideal plasma behavior is contingent on being capable of inducing flux which oscillates with a specific amplitude, frequency, and phase. Driving flux in a

PW	controlled pulse width
G_p	proportional gain term
G_i	integral gain term
G_d	derivative gain term
$fpfactor$	scaling factor applied to gain
$error^-$	error for present negative half pulse
$error^-sum$	sum of error for all negative half pulses to present
$errpr^-prev$	error for previous negative half pulse
$error^+$	error for present positive half pulse
$error^+sum$	sum of error for all previous positive half pulses
$error^+prev$	error for previous positive half pulse

Table 1.1: System parameter values and descriptions

manner so as to create the desired wave, requires an accurate means of measuring the flux. The existing system takes flux amplitude measurements (samples) at predefined time points during the oscillation cycle. However, these measurements are only used by the control code as scalar values and not a direct measurement of the rate of change of flux. Furthermore, the cycle period of the Blackfin does not exactly overlap the period of the flux waveform, due to the inherent phase delay between the PWM drive and the flux coil current. As such, it is not easy to determine where the sample points lie on the flux waveform. In the present control algorithm the sample points are simply summed to give an average value of the flux waveform for a half oscillation period.

Fortunately, there are methods of determining more about the flux waveform from these sample points. Thanks to the mathematical model of the flux circuit system dynamics developed by Moser and Nelson in their unpublished paper on the subject, this problem is an ideal candidate for Kalman filtering. This method utilizes the mathematical system model,

actual control inputs, and actual system samples to track the behavior of the actual system and converge the model with it. While preliminary research on this subject yielded mixed results, improvements in system simulation have allowed for new progress in this area. The entire flux waveform can now be made visible to the controller allowing, with additional fitting methods, not only for amplitude and DC offset adjustments, but adjustments in phase between injectors. This method and its implementation will be discussed further in subsequent chapters.

Chapter 2

CONTROL SYSTEMS BACKGROUND

2.1 Introduction to Control Systems

Control theory has always revolved around a desire to exact a specific behavior from a dynamic system. In the past this has been accomplished largely with basic feedback control algorithm like proportional (P), proportional-integral (PI), and proportional-integral-derivative (PID) control. However, over the past 80 years control theory has made substantial advances and now offers more optimal methods.

2.2 State Space Representation

The single largest difference between modern control and classical control is the use of state space representation. The behavior of a given system can generally be described by a set of ordinary differential equations (ODEs). By isolating the derivative term on the left hand side of the equation, we can place linear ODEs in the form

$$\dot{x} = Ax + g$$

In the event that the system is of an order higher than one, or contains more than a single dimension of movement, x becomes a vector $\mathbf{x}$ of length n and A becomes a matrix of dimension $n \times n$. This is the footwork for state space representation. In this very basic form $\mathbf{x}$ is our vector of variables or 'states' that can be evolved through time and A is called the state or system matrix.

Inputs can be added to this model in the form of an additional vector $\mathbf{u}$ of dimension

$m \times 1$ where m is the number of unique inputs. This input vector is multiplied by an imput matrix, B, of dimension $n \times m$ and is added to the product of $A\mathbf{x}$.

This model can be furthered by placing constraints on the states that can be measured. An output vector, $\mathbf{y}$ is defined as being the sum of two products: the product of an output matrix, C, and our state vector $\mathbf{x}$, and the product a feedforward matrix D with our input vector, $\mathbf{u}$. In their most general form, these equations appear as

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$$

$$\mathbf{y} = C\mathbf{x} + D\mathbf{u}$$

where $\mathbf{x}$ and $\mathbf{u}$ are clearly time dependent. Additionally, the system, input, output, and feedforward matrices can also be time dependent. In this case, the state space representation takes the form

$$\dot{\mathbf{x}} = A(t)\mathbf{x} + B(t)\mathbf{u} \tag{2.1}$$

$$\mathbf{y} = C(t)\mathbf{x} + D(t)\mathbf{u} \tag{2.2}$$

This state space representation is not applicable in matrix form to nonlinear ODEs, however, it is a very useful notation for linear time-varying systems. Further information on state space representation and linear systems can be found in Chen's work, Linear System Theory and Design [1].

2.3 Basic Controller Design

A very simple modern control model would appear as in Figure 2.1. Here the flow of state values can be seen in the center as well as their relation to output and input values. The feedback gain, K, seen in the matrix multiplication block at the bottom of the figure scales the output values and returns them to the system as an input.

While controller structure and the means of calculating the gain matrix, $\mathbf{K}$, will vary from implementation to implementation, they all rely on output data for the system. For the

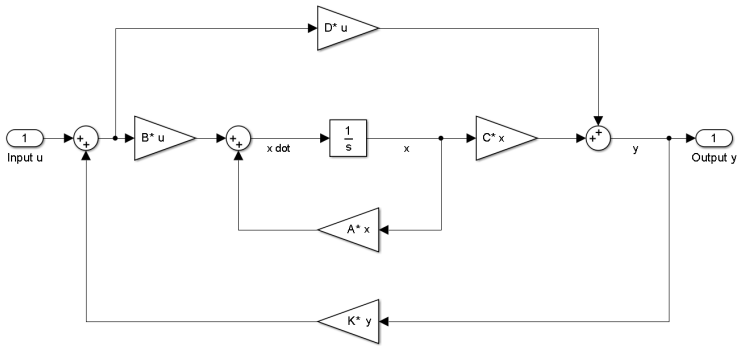

Figure 2.1: A Basic State Space with Output Feedback and Input Feedforward

controller to work properly, the output must be accurate and provide sufficient information about the states of the system that are to be controlled. In theory this is a simple matter. The output matrix can take the form of an identity matrix of dimension n, yielding each state as an output. In practice, however, this is not so easy. Specific system states are not always physically observable, and both measurement and process noise further complicates the situation. In these instances, filtering of outputs may become necessary.

2.4 Introduction to Optimal Estimation

When attempting to control a dynamic system using feedback, it is necessary to use measurements of specific system parameters. Often, however, these parameters contain noise making them hard to use, or are missing completely. Process noise is error added to the states while the system is evolved through time. Measurement noise is error added to the output when the states are measured. Both can lead to difficulties in computing the control input of a system.

The output of a system is rarely composed of individual states. Generally, the output is a set of some combination of two or more states. In these situations, a state observer can be used to compute the states using the state space matrices, the input, and the output for a given time, t. Extracting a state from these components, however, requires that the state be

observable.

For all of the states of a system to be observable, it is required that the 'observability matrix' for the system have the same rank as the number of states. This observability matrix can be constructed from the state and output matrices (A and C respectively) as follows

$$\mathscr{O} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

In the event that the rank of $\mathscr{O}$ is n, then the entire system is observable and a state observer will be capable of producing the states at a given time. If the rank is less than n, then some of the states of the system are not inherently observable. Modal Decomposition can be used to determine which states are observable and which are not.

Taking the eigenvector matrix, P, and multiplying it by the observability matrix, C, yields the unobservable states. They will be represented by zeros in the product. These unobservable states correspond to the the eigenvalue in the same index when the product $P^{-1}AP$ is computed.

Once the observable states have been determined, a state observer can be implemented.

2.5 The Kalman Filter

While state observers are useful, they fall short in several areas. For instance, they are often not capable of producing accurate state estimates given the process and measurement noise. Additionally, they are not capable of producing information for a given state when that state is unobservable (as discussed previously). For these reasons, it is desirable to implement a Kalman filter (or variation thereof).

The Kalman filter, also called Linear Quadratic Estimator, is a two-step algorithm designed specifically to eliminate error due to process and measurement noise. It has the

added benefit of being capable, in many applications, of determining states that are not observable. In concise terms, the Kalman filter contains a model of the real system. It evolves this model through time simultaneously with the real system making comparisons and adjusting when data are available. The states of the Kalman filter model are denoted as $\hat{\mathbf{x}}$.

The Kalman filter algorithm can be broken down into four important parts: initialization, the gain computation phase, the update phase, and the propogation phase. The initialization of the Kalman Filter (abbreviated as KF henceforth) consists of setting an initial guess of the real system states to initialize the model at the start time for the filter, t_0. In addition, an initial covariance matrix, P_0 (not to be confused with the matrix of eigenvectors from the previous section) is also established using the expected state process error. An accurate initial state guess can hasten state estimate convergence drastically. The initial covariance matrix should be generated based on the process error in the system. However, in practice it can generally be set to zeros. Both of those terms take the following form:

$$\hat{\mathbf{x}}(t_0) = \hat{\mathbf{x}}_0$$

$$P_0 = E\left\{\tilde{x}(t_0)\tilde{x}^T(t_0)\right\}$$

where $\tilde{\mathbf{x}}(t_0) = \hat{\mathbf{x}}(t_0) - \hat{\mathbf{x}}_0$ and $E\left\{\tilde{x}(t_0)\tilde{x}^T(t_0)\right\}$ is the expected value of $\tilde{x}(t_0)\tilde{x}^T(t_0)$. The expected value is the weighted average of all possible values which yields the most probable value for any given time.

The remaining phases of the KF represent the looping portion of the algorithm that runs for the duration of the control system (if the filter is to be used in real time). The discrete Kalman filter uses subscripts, such as x_k or x_{k+1}, to denote the loop iteration. As it is a two step algorithm, additional notation must be used to differentiate between the first and second step of each loop iteration. To show this we use the x^- and x^+ superscripts, respectively. The gain computation phase takes the previous covariance matrix, P, and uses

it to calculate the gain, K, which is to be used in updating the state estimate, $\hat{\mathbf{x}}$. This is one of the more computationally intensive steps in the algorithm as it involves a matrix inversion. Fortunately, this inversion is dimensionally identical to the number of outputs. Thus, for many systems it is small. The gain is computed as

$$K_k = P_k^- C_k^T \left[C_k P_k^- C_k^T + R_k \right]^{-1}$$

where C is the output matrix, P is the covariance matrix, and R is the measurement noise.

The update phase involves updating the estimates of both the state and the covariance matrix for the current time step. The new state estimate becomes a weighted average of the previous estimate and the error between the previous estimate of the output and the actual output. Both the state estimate and the covariance matrix are updated based on the gain calculated in the previous phase. Here the gain is used to adjust the model system's reliance between its previous iteration and the value measured from the real system. $h_k \hat{\mathbf{x}}$ is the term containing the sample of the real system.

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + K_k \left[\tilde{y}_k - h_k \hat{\mathbf{x}}_k^- \right]$$

$$P_k^+ = \left[I - K_k H_k \right] P_k^-$$

The final stage of the KF algorithm is the propagation phase wherein the state and covariance matrices are both evolved through time in order to produce an estimate at the next time point in the system. In the case of a discrete system, the stepping is trivial. In the case of a continuous time system, many different finite difference methods may be used to advance the system.

$$\hat{\mathbf{x}}_{k+1}^- = \Phi_k \hat{\mathbf{x}}_k^+ + \Gamma_k \mathbf{u}_k$$

$$P_{k+1}^- = \Phi_k P_k^+ \Phi_k^T + \gamma_k Q_k \gamma_k^T$$

In this case Φ and Γ correspond to the discrete time equivalents of the A and B matrices respectively. γ and Q are both terms relating to the process error of the system.

2.6 Kalman Filter Variants

There are several variations of the Kalman filter that extend the principle to address different problems. One particular variant, the Extended Kalman Filter (EKF), is designed to be used on nonlinear ODEs.

The EKF is built on the same architecture as the basic Kalman Filter. It revolves around a looped three-step process. Gain computation, estimate update, and estimate propagation. The difference here lies in the computation of the system dynamics Jacobian matrices computed for this filter. Rather than simply taking the A matrix from the state space representation, the matrix must be calculated for every timestep. Thus, given the system

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t)$$

$$\tilde{\mathbf{y}}(t) = \mathbf{h}(\mathbf{x}(t), t)$$

the filter algorithm includes Jacobian terms

$$F(t) = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}(t), \mathbf{u}(t)}$$

$$H(t) = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}(t)}$$

For a discrete system these terms are slightly altered

$$F(\hat{\mathbf{x}}_k) = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_k, \mathbf{u}(t)} \tag{2.3}$$

$$H_k(\hat{\mathbf{x}}_k) = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_k} \tag{2.4}$$

By inserting the above equations into the Kalman filter, the discrete Extended Kalman Fitler algorithm is realized.

The gain phase takes the form found in equations 2.5 and 2.6. Following the gain phase is the update phase which takes the form of equations 2.7 and 2.8. The propagation phase takes the form seen in equations 2.9 through 2.11. Note that in place of the system matrices used in the discrete time Kalman filter, the EKF uses the Jacobian matrices calculated in 2.3 and 2.4.

$$K_k = P_k^- H_k^T(\hat{x}_k^-)\left[H_k(\hat{x}_k^-)P_k^- H_k^T(\hat{x}_k^-) + R\right]^{-1} \tag{2.5}$$

$$H_k(\hat{x}_k^-) = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k^-} \tag{2.6}$$

$$\hat{x}_k^+ = \hat{x}_k^- + K_k\left[\tilde{y}_k - h(\hat{x}_k^-)\right] \tag{2.7}$$

$$P_k^+ = [I - K_k H]P_k^- \tag{2.8}$$

$$\hat{x}_{k+1}^- = f(\hat{x}_k^+) \tag{2.9}$$

$$P_k^- = F(\hat{x}_k^+)P_k^+ F^T(\hat{x}_k^+) + \Upsilon Q \Upsilon^T \tag{2.10}$$

$$F(\hat{x}_k^+) = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}_k^-, u(t)} \tag{2.11}$$

In equations 2.5 and 2.10, R and Q are terms used to add process and measurement noise. The Υ is a method of weighting the application of process noise for different states. The derivation of these two filters is covered in Crassidis and Junkins optimal estimation text [2].

Chapter 3

CONTROL SYSTEM SIMULATION

3.1 Flux Circuit Simulation

While the controller improvements are poised to make a large difference in the operation of the HIT-SI experiment, several more steps must be undertaken prior to implementation. Due to the high cost of operation and even higher cost of repairs to the containment vessel and the control power supply system, it is necessary to test any control code changes extensively in simulation prior to implementing them. To facilitate this testing, a reliable and empirically accurate simulation of the flux circuit must be constructed.

This simulation is written using MatLab and seeks to replicate the physical system as much as possible. Several external functions have been developed simultaneously with the main simulation code. These functions are utilized in modeling the system dynamics of the flux circuit. The complete code can be found in the appendix.

Expanding on the flux and voltage circuit dynamics developed by Nelson and Moser (May 10, 2011), a piecewise state space representation for the flux circuit can be created in the form of equations 2.1 and 2.2.

The mesh equations are rather lengthy so it is helpful to define several repeated quantities in advance. These are defined in equations 3.1 through 3.13. The values of parameters expressed in these equations can be found in table 3.1 along with their descriptions.

$$a = \frac{1}{L_s + L_c} \tag{3.1}$$

$$b = \frac{1}{L_{cab} + L_p + L_c} \tag{3.2}$$

$$eh = a(R_s + R_c) \tag{3.3}$$

$$f = aR_c \tag{3.4}$$

$$g = aL_c \tag{3.5}$$

$$h = bR_c \tag{3.6}$$

$$k = b(R_{cab} + R_p + R_c) \tag{3.7}$$

$$l = bR_p \tag{3.8}$$

$$m = bL_c \tag{3.9}$$

$$n = bL_p \tag{3.10}$$

$$p = \frac{R_p}{L + L_p} \tag{3.11}$$

$$q = \frac{R + R_p}{L + L_p} \tag{3.12}$$

$$v = \frac{L_p}{L + L_p} \tag{3.13}$$

These quantities are used in the construction of the A and B matrices for our system state space. The complete syste state space can be seen in equations 3.14 and 3.15. The

Quantity	Value	Description
L_s	2.47 μH	series inductance (with source)
R_s	5 mΩ	series resistance (with source)
L_c	100 nF	capacitor leg inductance
C	95 μF	capacitance
R_c	2 mΩ	capacitor leg resistance
L_{cab}	0 H	cabling inductance
R_{cab}	0 Ω	cabling resistance
L_p	8.21 μF	parallel (to load) inductance/primary inductance
R_p	5 mΩ	parallel (to load) resistance/primary resistance
L	1.85 μH	coil inductance/(transformed) plasma inductance (vacuum)
R	20 mΩ	coil resistance/(transformed) plasma resistance (vacuum)

Table 3.1: System parameter values and descriptions

C matrix for this system produces only the injector flux value. This differs from Nelson and Mosers original system and reflects the limited observable output of the actual system. This matrix is shown in equation 3.16.

$$A = \frac{1}{1-mg-nv}\begin{bmatrix} gh-eh(1-nv) & f(1-nv)-g(k-np) & g(l-nq) & gb-a(1-nv) \\ h-meh & mf-k+np & l-nq & b-ma \\ v(h-meh) & p(1-mg)+v(mf-k) & vl-q(1-mg) & v(b-ma) \\ \frac{1}{C}(1-mg-nv) & \frac{-1}{C}(1-mg-nv) & 0 & 0 \end{bmatrix} \tag{3.14}$$

$$B = \frac{1}{1-mg-nv}\begin{bmatrix} a(1-nv) \\ ma \\ vma \\ 0 \end{bmatrix} \tag{3.15}$$

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.16}$$

$$\mathbf{x} = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ V_c \end{bmatrix} \tag{3.17}$$

$$u = [V_s] \tag{3.18}$$

3.2 Simulation Flow

3.2.1 Initialization

The main simulation code can be broken down into several sections: initialization, experiment, and post-processing. The initialization phase contains the definition of all constants

used in the simulation, the construction of the target plasma duty cycle percentage (referred to as DCP_Plasma), the initialization of all the arrays used to store output during the simulation, and the initialization of the Kalman filter used to observe the states.

The simulation parameters which are defined in this section include those required for operating the experiment. Some of these parameters are user-specified on the actual experiment (such as the number of ADC samples per oscillation half cycle) while others are inherent physical properties of the machine (such as the clock speed of the Blackfin processors). Any parameters which are adjustable by the operator partianing to the execution of the flux circuit circuit code can be found and adjusted in this section. The DCP_Plasma signal is one such parameter.

Unlike most of the other parameters defined in the initialization section, DCP_Plasma is an input array. While the other parameters are typically scalar in nature, this parameter can be visualized as a waveform envelope. It will be referred to as a signal hereto forth. Unlike scalar values which can simply be read, this parameter must be constructed using an array of time points and corresponding amplitude values. Once the waveform has been constructed, it is time scaled and discretized such that the number of samples corresponds to the number of half oscillations of the control system for a given shot.

After the DCP_Plasma signal is interpolated onto the simulated Blackfin timebase, all of the storage arrays are initialized. These arrays are needed to to log various parameters during the simulated shot so they can be reviewed and analyzed after the experimental loop has run. These include the system states, the timebase, simulated ADC samples, and estimated states returned by the KF, to name a few. The sizes of all of these arrays are dependent on the number of cycles for which the shot is programmed to run.

The Kalman Filter is initialized after the storage arrays are allocated. This involves several different components. Storage arrays much like those mentioned previously are initialized to store filter parameters including the gain, covariance, and estimated states. Additionally, the starting covariance matrix and state estimate must be defined. Finally, a discretization of the flux circuit must be created so that the estimated states and covarience

matrix can be evolved through time (as outlined in chapter 2).

3.2.2 Execution

The simulation execution is composed of four main components in a loop. The loop executes for the predefined number of half cycles in the shot.

The first component of the simulation execution section is the feedback control system. This component includes a case structure with two different modes corresponding to vacuum and plasma. The vacuum mode is simply a pre-programmed ramping waveform. This mode is selected when the number of oscillation half cycles has not exceeded the predefined number corresponding to vacuum power ramping. Plasma is assumed to break down approximately when this predefined number of half cycles has completed. Once this occurs, the case switches into plasma feedback control. This mode is a basic PID based feedback controller. A detailed explanation of the workings of this code are discussed in Chapter 1. This component of the code features switches to allow for easy variation of the control scheme. A new control method can be implemented in a separate case and enabled prior to execution, allowing for comparison of control method effectiveness. However, beyond the addition of these switches, care was taken to replicate the actual operational C code as closely as possible. This was done with the goal of accurately reproducing historic shots with a high degree of accuracy in mind. Both the preprogrammed ramping and the PID feedback controller provide a pulse duration which is used by the system time evolution component of the execution loop.

Once the control component has executed, the loop moves to the system dynamics time evolution component. Since the system input is pulse width modulated, the input takes the form of a fixed non-zero input value lasting a specified duration. Prior to any time evolution, this component of the execution code calculates key time points for the next half cycle. These points are the beginning of the half cycle, the time when the PWM input pulse begins, the time when the PWM input pulse ends, and the time when the half cycle ends. Once these points are calculated, the half cycle is broken into three time spans (divided by

these key time points). Each of these time spans is evolved in a single pass with a specific input value (0 for the first and last parts, and input voltage for the center part).

This evolution is accomplished using an ODE solver (geneally ode45, a Runge-Kutta 4-5 finite difference method) and one of three system dynamics functions. As the interior of the injector changes from vacuum to plasma, and then throughout the duration of the plasma, the dynamics equations that describe the behavior of the flux injector must also change. Specifically, the resistance and inductance of the injector flux coils will change, much in the same way the inductance of a coil will change depending on the core. The function dynamicsfuncvac.m is a linear first order ODE used to represent the system without plasma present. This function is used to evolve the system states through the three time spans of any given half cycle with the corresponding input voltage values. Once the predefined number of pre-plasma cycles have been completed, dynamicsfuncplas.m is used. After the powered portion of the shot is completed, dynamicsfuncspaoff.m is used to allow the system to relax back to a zero state in the same manner as the actual machine. This is accomplished by modifying the resistance and inductance values of the flux circuit leg corresponding to the SPAs. To simulate the SPAs shutting off and opening that portion of the circuit, the resistance value is set very high (within reason to prevent the system from becoming stiff and difficult to evolve with speed).

Once the system has been time evolved, the outputs of all three time spans are concatenated and stored for sampling and analysis.

The sampling component of simulation execution occurs next. Given a number of ADC samples to be taken per half cycle (defined by the operator in the initialization phase), this piece of code determines the time step between samples. It uses this time step to search the concatenated time evolution output for the index with the closest time value. The state at each of these indices is then saved to a storage matrix. These sample values are also added together and temporarily stored for use by the feedback system on the next loop iteration.

The final phase of the execution matrix is the Kalman Filtering portion. This follows the basic outline given in Chapter 2 for a discrete-discrete Kalman filter. Due to the time

dependence of the system matrix, this KF implementation uses two separate propagation steps which are contained within a case structure just like the actual system evolution.

3.2.3 Post Processing

The final stage of the simulation is a basic post-processor. This portion of the code is responsible for returning a graphical representation of the experiment and conducting basic error analysis of the Kalman Filter results. All system states, inputs, and outputs can be read from the simulation with minor modification. At this time, flux, system input, and PID error are plotted by default.

In order to gauge the effectiveness of the Kalman Filter, it is necessary to examine the difference between the predicted state and the actual state for a given point. This difference is called the error. The $\pm 3\sigma$ bounds, corresponding to the values of the three standard deviations of the measurements, are generally plotted on the same figure. As long as the error appears to remain within the $\pm 3\sigma$ bounds, the filter is said to be effective. As the $\pm 3\sigma$ bounds lines appear to converge on a value which represents the convergence of the filter to within a specific range.

3.3 Simulation Results

For the purpose of verification the simulation has been set up with the proper constants and signals to emulate shot 129499 of HIT-SI operation. While this shot actually involved both the X and Y injectors, the simulation will only produce data for a single injector. Actual shot data for the X injector were imported to the simulation for comparison purposes.

The flux circuit dynamics model mentioned previously, did not provide values for vacuum stage coil inductance or resistance. As such, values were estimated and the simulation was run. This initial run proved rather disappointing, as the simulation output waveform varied considerably from the recorded waveform of the actual shot. After investigating the discrepancy, it was determined that the values provided for the plasma stage coil inductance

and resistance were actually the appropriate values for these parameters during the vacuum stage. The plasma stage parameters were not given.

A minor foray into determining the plasma stage parameters was undertaken. However, it soon became evident that the values were time dependent and, as such, the coefficients would not be easy to guess. Approximate starting values were arrived upon but these values proved to not drain sufficient energy from the system. This manifested itself in continuously growing flux amplitudes dissimilar from the plateauing values seen in the actual experiment. Running the entire shot under preprogrammed control minimized these issues and provided a wave that more closely matched the experimental data. As such the remainder of the research was conducted with the control system running the entire shot in preprogrammed mode.

Determining the unknown time-dependent plasma stage system parameters lends itself well to a parameter estimation implementation of the Kalman Filter. This became an extension of the research being conducted. A discussion of the parameter estimation using an Extended Kalman Filter is covered in Chapter 5.

3.3.1 Preprogrammed control

Overall, the simulation was found to exhibit behavior very similar to that of the actual system in pre-programmed mode. The conversion of the target waveform amplitude is conducted quickly and produces a waveform of proper dimensionality. This can be seen in Figure 3.1.

The flux waveform builds from an amplitude of zero to the point where plasma breaks down. At this point we see a very slight shift in phase and the amplitude begins to grow once again. This can be seen in Figure 3.2.

The simulation feedback sample points are denoted in this figure with a magenta 'x'. The beginning and end of each Blackfin half cycle is indicated with a cyan 'o'.

When compared with the apriori flux data from this shot, it can be seen that the simulation waveform matches quite well. There are some minor differences between the acutual

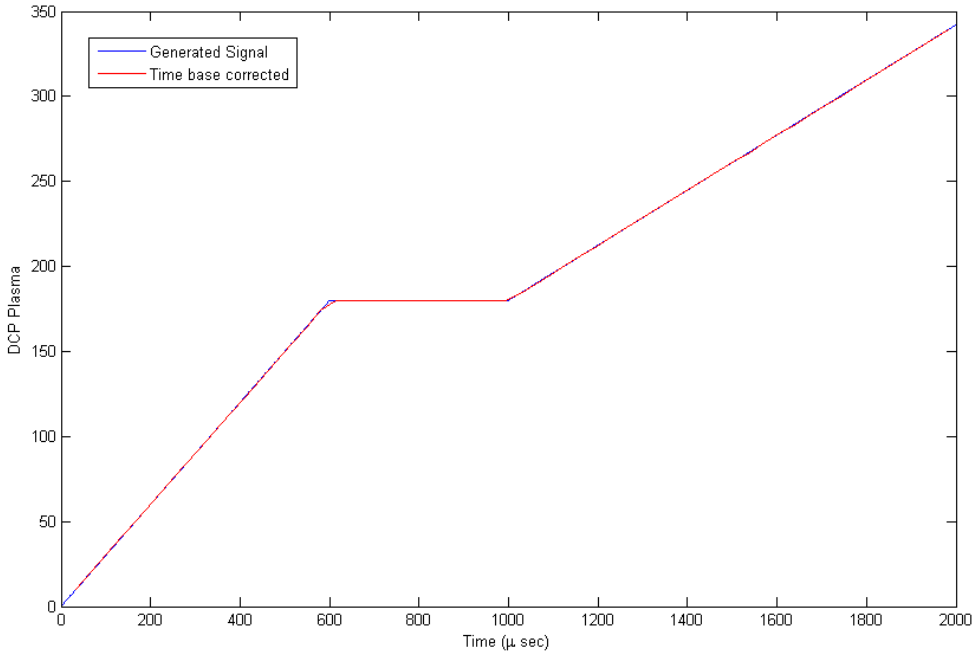

Figure 3.1: Simulation flux demand waveform and time base corrected version

experimental data and the simulation, however. The switch to plasma in the simulation removes less energy from the system than the corresponding switch in the apriori data however, this effect is minimized as the apriori flux wave grows much more quickly than the simulation wave. These differences may be due to unknown or poorly estimated plasma stage circuit inductances and resistances.

The simulation stages where the circuit parameters are known are nearly identical. The preprogrammed vacuum ramping stage matches particularly well with a negligible phase and frequency difference, and a very slight amplitude difference. This can be seen in Figure 3.3.

The stage at the termination of the shot, when the SPAs are shut off, also matches well, exhibiting similar, if not identical phase and frequency. Again, a slight disparity between amplitudes is present. This can be seen in Figure 3.4.

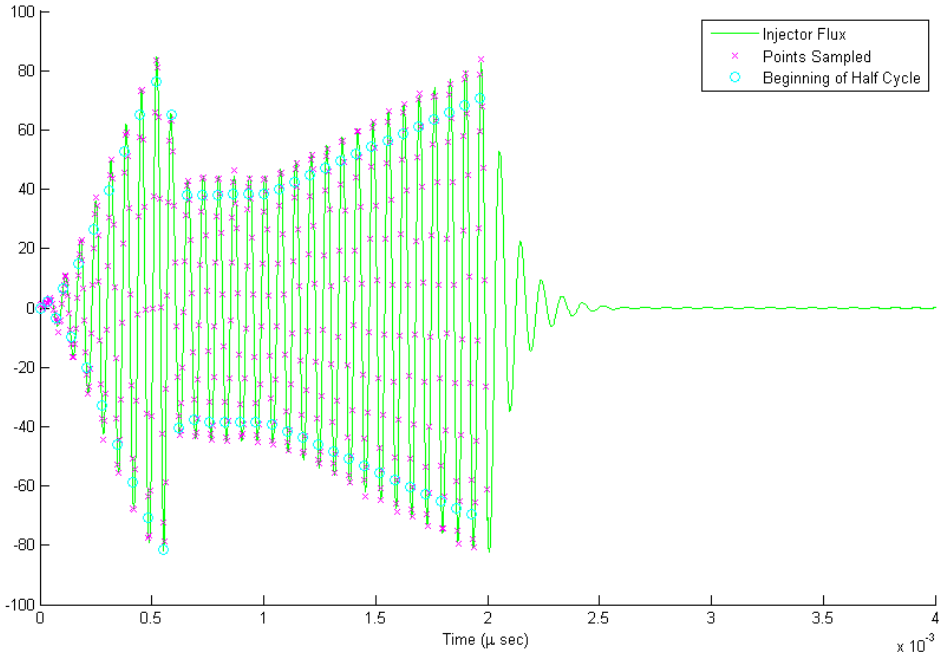

Figure 3.2: Simulated flux waveform with simulated sample points and simulated Blackfin half cycle boundaries

3.3.2 PID control

The PID control results are as equally promising as those of the preprogrammed control scheme. As with the preprogrammed control, the simulation waveform plasma stage doesn't perfectly match the experimental plasma stage data. This is again due to the unknown change in circuit inductance due to plasma presence in the injectors.

To provide the amplitude matching seen in the preprogrammed control ramping stage, the ADC sample values were scaled by $\frac{1}{10}$. As such, it is logical to assume that the ADC sample sums used by the PID system should be scaled by the same amount. This serves to eliminate the scaling introduced by the Blackfin ADC (the simulation samples the wave directly so there is no scaling applied by default).

With a scaling value of .1 the PID control system is seen to compensate for the addi-

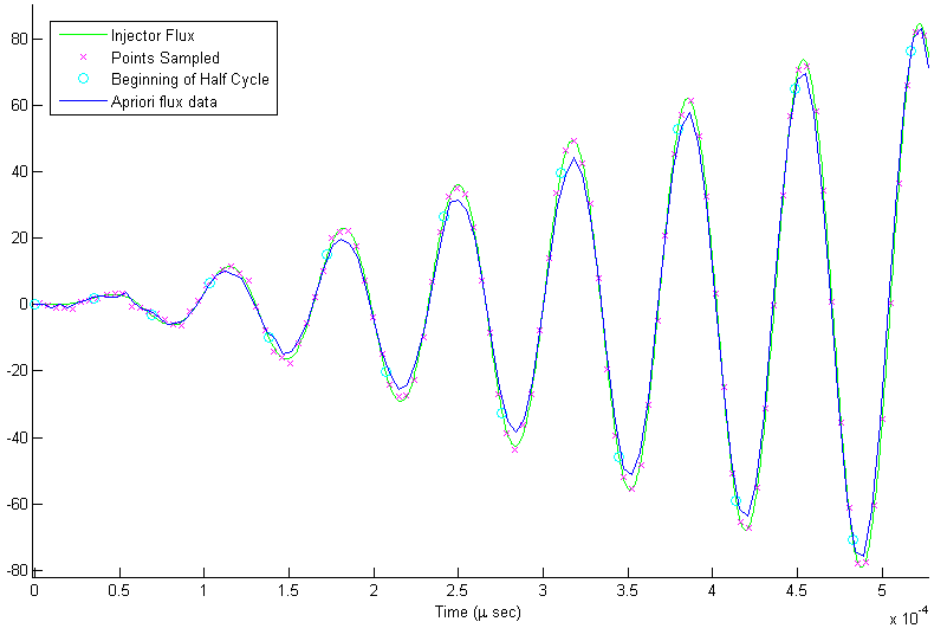

Figure 3.3: The simulated and experimental "a priori" flux waveform ramp stage

tional loading introduced when plasma breaks down. Ultimately, it is able to drive the flux waveform amplitude much higher than the preprogrammed control. This can be seen in Figure 3.5.

In addition to driving higher flux amplitudes, the PID control scheme is able to remove DC bias. Closer examination of the peak simulation flux waveform amplitudes just after plasma breakdown ($\approx 5.5\mu s$) indicate the beginning of a DC bias. This is largely eliminated by the PID control within the next full cycle (see Figure 3.6).

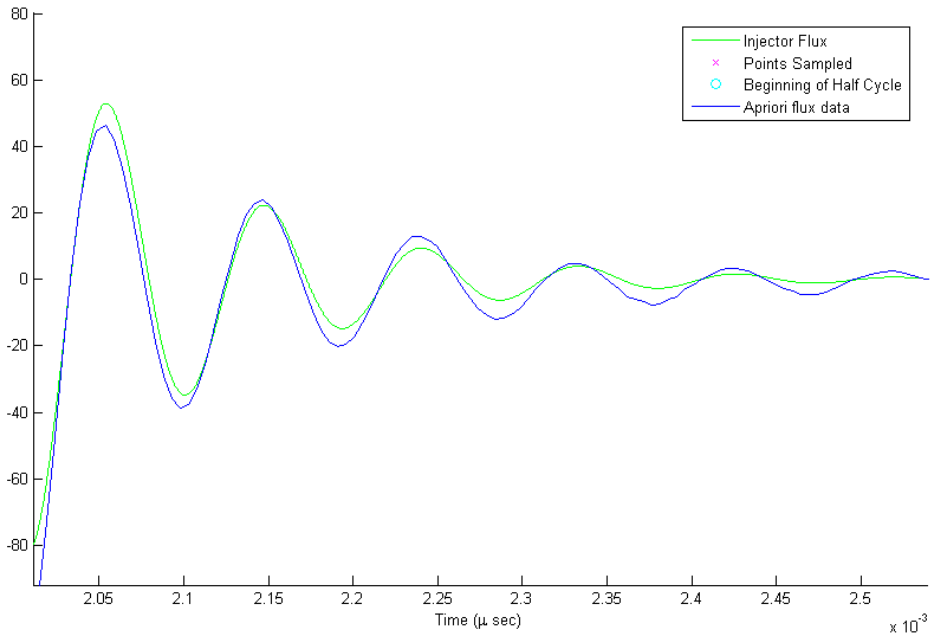

Figure 3.4: The simulated and experimental “a priori” flux waveform SPA shutoff stage

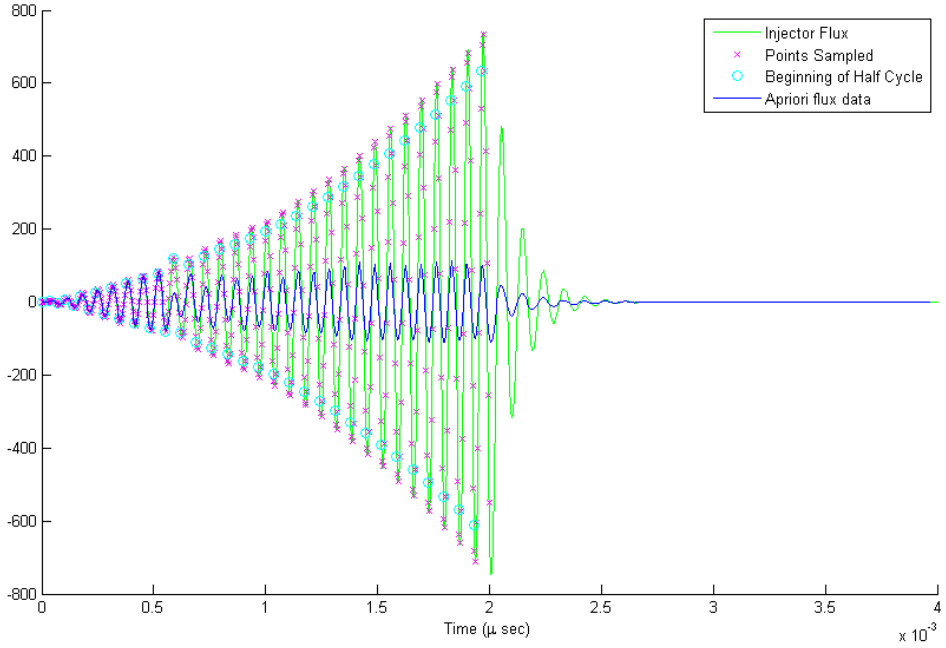

Figure 3.5: The simulated flux waveform with experimental data, PID control scheme

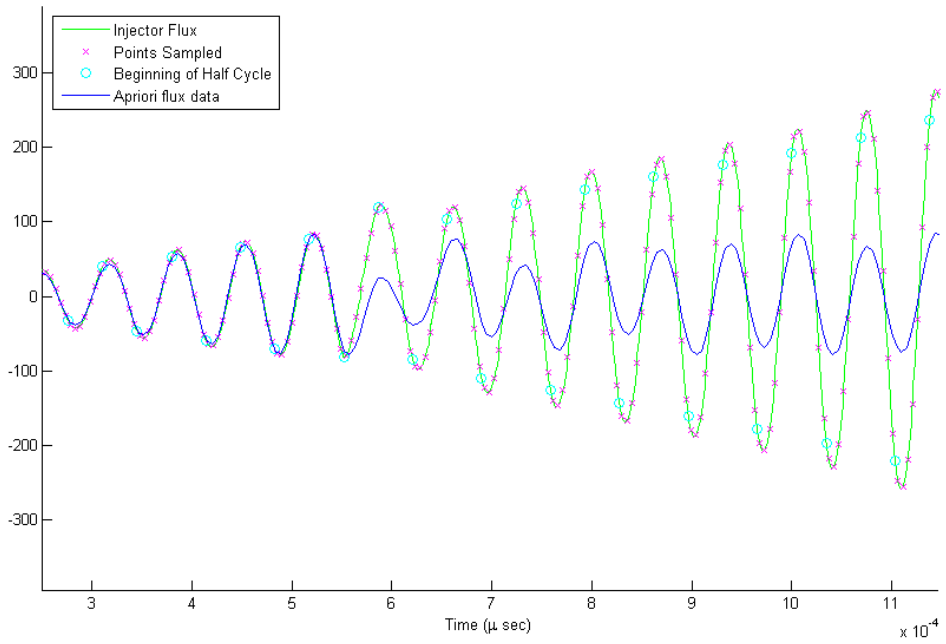

Figure 3.6: The simulated flux waveform with experimental data, PID control scheme

Chapter 4

KALMAN FILTERING OF FLUX SAMPLES

4.1 The Case for Kalman Filtering

The Kalman Filter is an example of a state estimation method that has arisen out of the development of modern control. It is considered an advanced method but its uses are widespread in signal processing and data analysis.

Thus far, the usefulness of this filter has been explored in two similar real-time-operation implementations: state estimation using the full flux circuit system dynamics, and state estimation using a sine wave with basic parameter estimation.

The motivation for the state estimation using the full system dynamics model is fairly evident. At the very least, it would allow the control system to correctly interpolate data points for the flux waveform between the sparsely placed data samples taken by the Blackfin ADCs. At most, it could provide useful data about the other system states, enabling capacitor bank voltage to be calculated without the need of an additional ADC. With additional work, this could lead to a voltage drop optimized control system producing either higher toroidal currents or longer shots.

The most basic task of the KF is better flux wave interpolation. This can produce more accurate data for peak wave amplitude. Variations of the KF used for sine wave fitting will attempt to converge amplitude, frequency, and phase values for every step of the wave. While the absolute value of this phase estimation has little intrinsic value (it only indicates the wave phase relative to an arbitrary sine or cosine wave), the relative phase between different injectors could be determined in real time using this method. This would allow for on-the-fly phase adjustments between injectors.

4.1.1 Downfalls of KF Method

The major disadvantages of the Kalman Filter arise in time or computational resource sensitive environments. While the code involved in the basic KF is fairly straightforward, utilization requires a matrix inversion to compute the gain prior used in the update phase. Additionally, a function evaluation is required prior to computing the Jacobian matrix of the system dynamics at a given time step. All of these operations are inherently computationally intensive and will slow the code even if the resources and speed are present to compute them in the first place.

For linear time independent system dynamics, where the system can be put into the form

$$\dot{x}(t) = Ax(t) + Bu(t)$$

this issue is minimized. In this case, a function evaluation is not required as the Jacobian matrix with respect to the states is simply the matrix A. Additionally, since the A matrix is time independent, the matrix inversion can be calculated once for all time steps.

Unfortunately, in the case of nonlinear or time dependent system dynamics (like the systems in equations 4.1 and 4.2), this is not the case. In these cases, the function evaluation and Jacobian calculation must occur every time step or whenever the states or system parameters change. Additionally, the matrix inversion used to calculate the update gain must be computed every time step. As such the EKF method, like that used for sine wave fitting, becomes very computationally intensive and much less ideal for real-time use.

$$\dot{x}(t) = f(x(t), t) \tag{4.1}$$

$$\dot{x}(t) = A(t)x(t) \tag{4.2}$$

4.2 Full-Dynamics KF

This research sought to implement a Kalman Filter for state estimation of the flux waveform. To do this, a KF algorithm was modified and implemented just after the data sampling portion of the Flux Circuit simulation code.

4.2.1 Implementation

The Kalman Filter code implemented on the flux circuit accounts for approximately 29 lines of code not counting initialization. The actual filter can be reduced to a mere eight lines of code. However, with the necessary sampling loops, the fully realized algorithm requires more space.

In order to imitate the actual continuous time system, the simulation code extracts the system state at specific intervals. This state state extraction emulates the ADC sampling that occurs on the actual system. Due to the variable time-stepping nature of the simulation code, it is not easy to find the real state index corresponding to a specific time. To find the index most closely corresponding to the desired time step, a simple search algorithm has been implemented. The Matlab function `interp` can be modified to accomplish the same effect, but is not as easily ported to the C language. This search algorithm identifies the index with the closest time stamp to the desired time, and returns its index number. This way, state values for this index can be easily found. The search algorithms and an additional six lines that are used exclusively for storing results compose the remainder of the KF algorithm.

In order to accommodate the change in system dynamics that occurs when the plasma breaks down $\frac{1}{3}$ of the way into the shot, two separate propagation conditions exist. The first one is for vacuum propagation while the second is for plasma propagation. These two conditions account for the different propagation equations that must be used to advance both the state estimate and the covariance matrix.

4.2.2 Results

Overall the results of this filter are very promising. The primary function of the filter in this implementation is to provide a more complete picture of the flux wave. For these to data to be useful, the filter must converge upon the actual data prior to the control system switching to feedback mode. Observing Figure 4.1 it can be seen that this takes place.

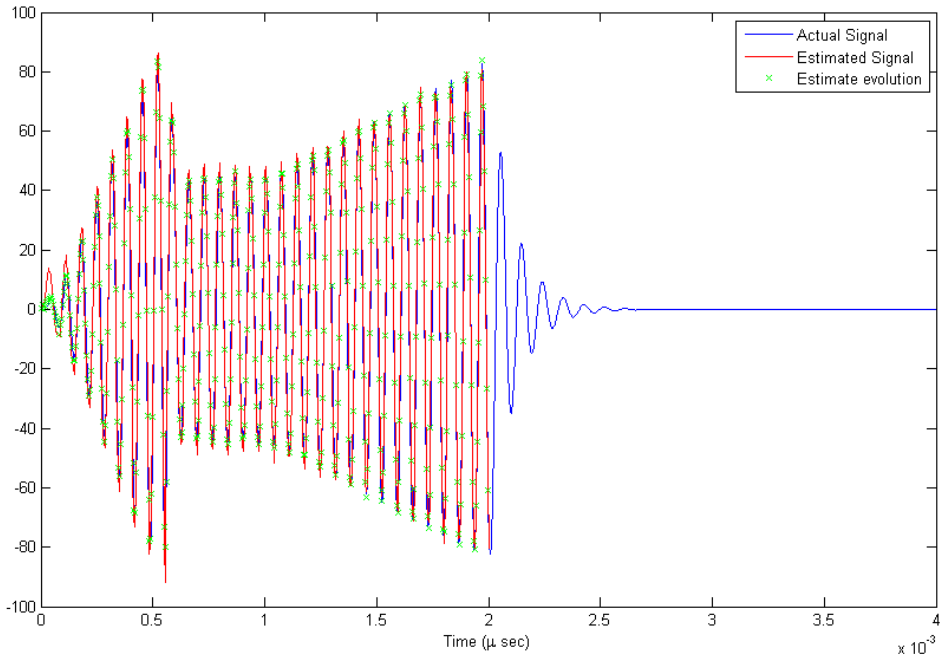

Figure 4.1: Kalman state two estimate before (estimated signal) and after update phase (estimate evolution), and simulated flux waveform

Furthermore, by more closely examining the first 500 nano seconds of the shot, it can be seen that the filter effectively converges within the first 5 cycles of the wave (Figure 4.2).

The other states, capacitor voltage and other currents, are less crucial to the PID controller and could be observed through the addition of low cost sensors. However, it is worth noting that 2 of the remaining three also converge very quickly. The estimate for state three, the current through the third circuit leg, converges in approximately the same time as state

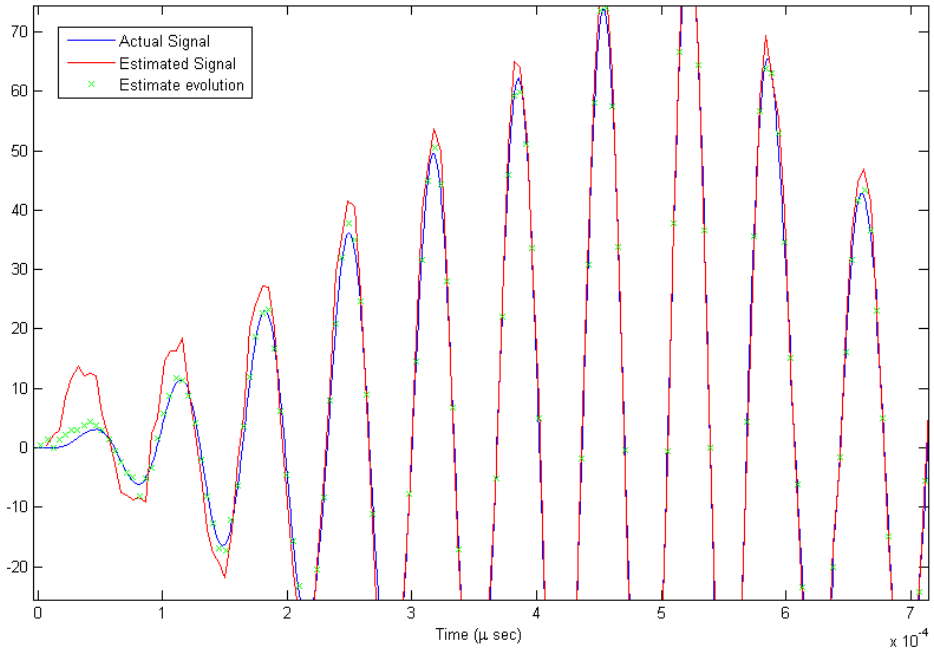

Figure 4.2: Kalman state two estimate before and after update phase, and simulated flux waveform

two. This can be seen in Figure 4.3. This closely tied convergence time is likely due to state three representing a portion of the flux encompassed completely by state two.

State four, the flux tank circuit capacitor voltage, converges somewhat slower. However, the estimate of this state is still able to converge upon the real value by the time plasma breaks down (see Figure 4.4).

The convergence of Kalman Filters is frequently gauged by the error between the estimated state and the actual state. If the error lies between positive 3σ and negative 3σ, which is the three times the expected standard deviation of that time step, it is said to have converged on the actual solution and be tracking properly. From Figure 4.5 it can be seen that the state two, injector flux, estimate error is in fact bounded by $\pm 3\sigma$.

Similarly, it can be seen that the other two states which successfully converge are

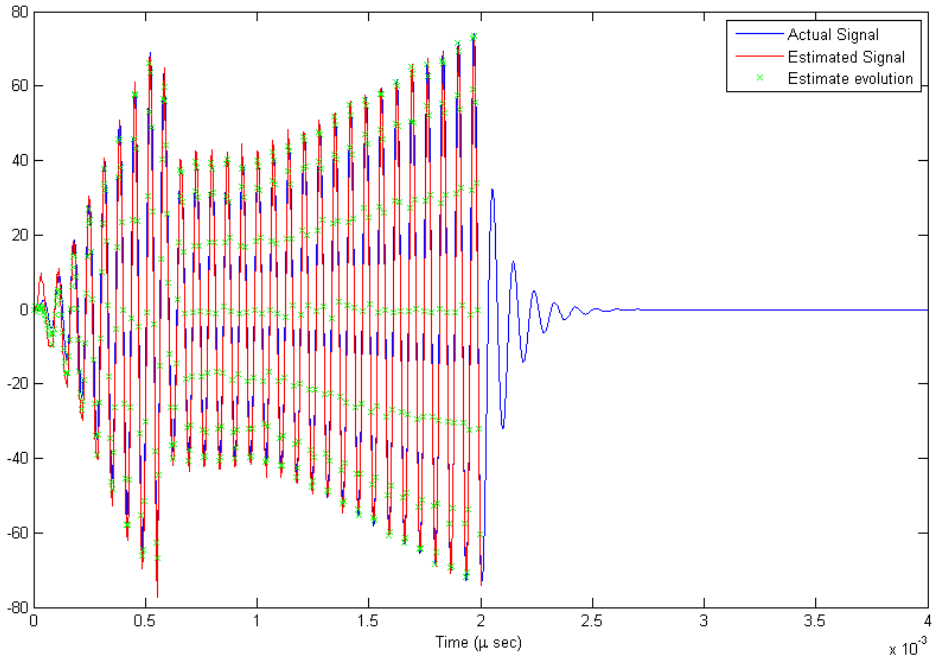

Figure 4.3: Kalman state three (leg three current) estimate and simulation flux waveform

bounded above and below by $\pm 3\sigma$. State four converges very nicely as seen in Figure 4.6. State three, on the other hand, has error beyond the normal measurement noise initially. This error is quickly corrected resulting in basic measurement noise which falls well within the 3σ bounds. Figure 4.7 depicts the error wave residing within the 3σ bounds.

In order to better test the full state dynamics Kalman filter, Gaussian noise was added to to the samples to simulate measurement noise. While this noise manifested itself in the state errors previously plotted, the filter was still able to converge to proper values of three states.

4.3 Sine Fitting KF

The full state dynamics Kalman filter is an excellent algorithm that has been shown to allow for better interpolation of the system states. However, it lacks some desirable attributes.

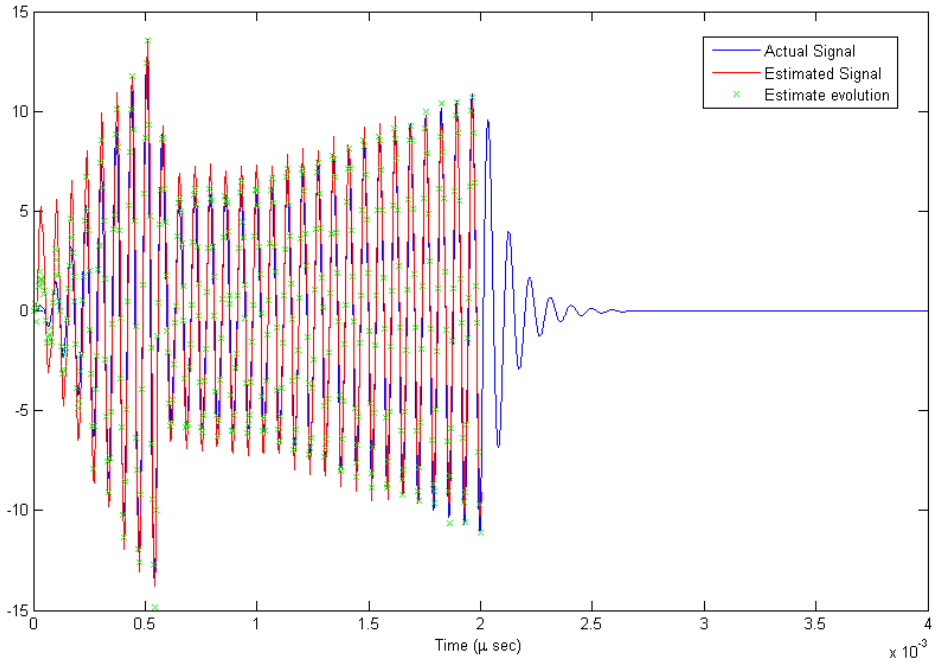

Figure 4.4: Kalman state four (capacitor voltage) estimate and simulation flux waveform

For example, it may, in the future, become useful to know the phase of the injector flux waveform relative to some fixed point. This would allow for phase adjustment between the injectors. There are many ways to process the sampled data to determine phase. One of these is a parameter estimating Kalman filter.

Like the full state dynamics filter, the sine parameter estimating filter operates in real time, using output samples as they are taken. However, instead of estimating the states of the complete flux tank circuit, the filter estimates the parameters of a sine wave.

4.3.1 Implementation

The sine fitting extended Kalman filter algorithm follows the outline discussed in the previous chapter. Unlike the full dynamics KF, in this filter each state corresponds to a different wave parameter. State one corresponds to the value of the sine wave at time t in terms of

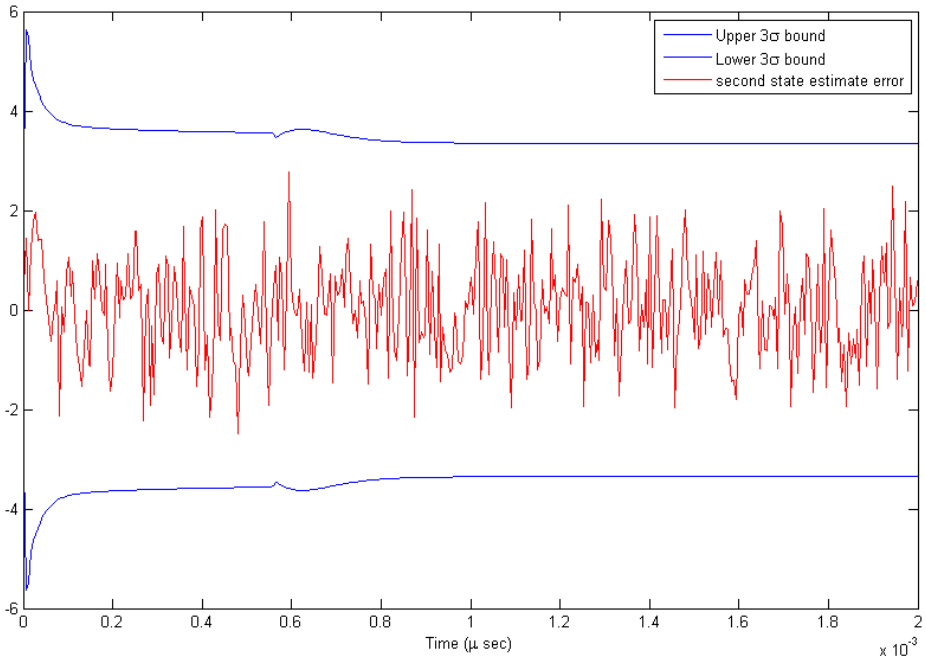

Figure 4.5: Kalman state two estimate error and $\pm 3\sigma$ bounds

the other three states. The second state corresponds to the frequency of the wave. State three corresponds to the amplitude of the wave. The fourth state is the phase value for the wave. These variables can be assembled into a nonlinear system as follows:

$$x_1 = x_3 * \sin(x_2 t + x_4)$$

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_3 * \sin(x_2 t + x_4) \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

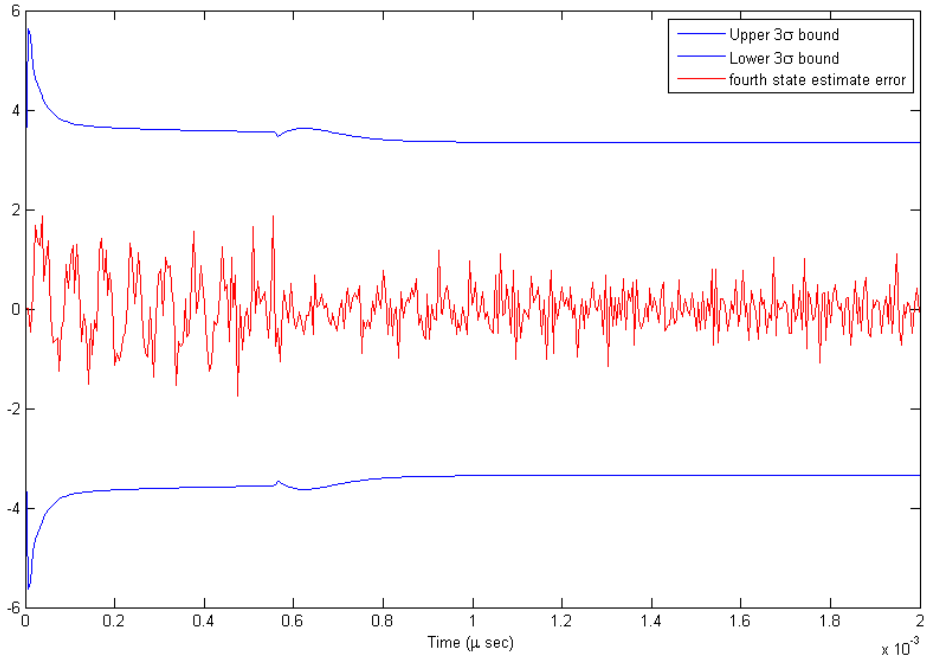

Figure 4.6: State four estimate error and $\pm 3\sigma$ bounds.

The Jacobian of the system then takes the form

$$A = \begin{bmatrix} 0 & x_3 t \sin(x_2 t + x_4) & \sin(x_2 t + x_4) & x_3 \cos(x_2 t + x_4) \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The above are used in place of the full state dynamics in the implementation of an extended Kalman filter. Because the original state dynamics are no longer being used, the parameter switch which was necessary in the full filter is no longer needed.

4.3.2 Results

The sine fitting EKF (Extended Kalman Filter) implemented in this application does not appear to correctly converge the true value of amplitude, phase, and frequency. The most

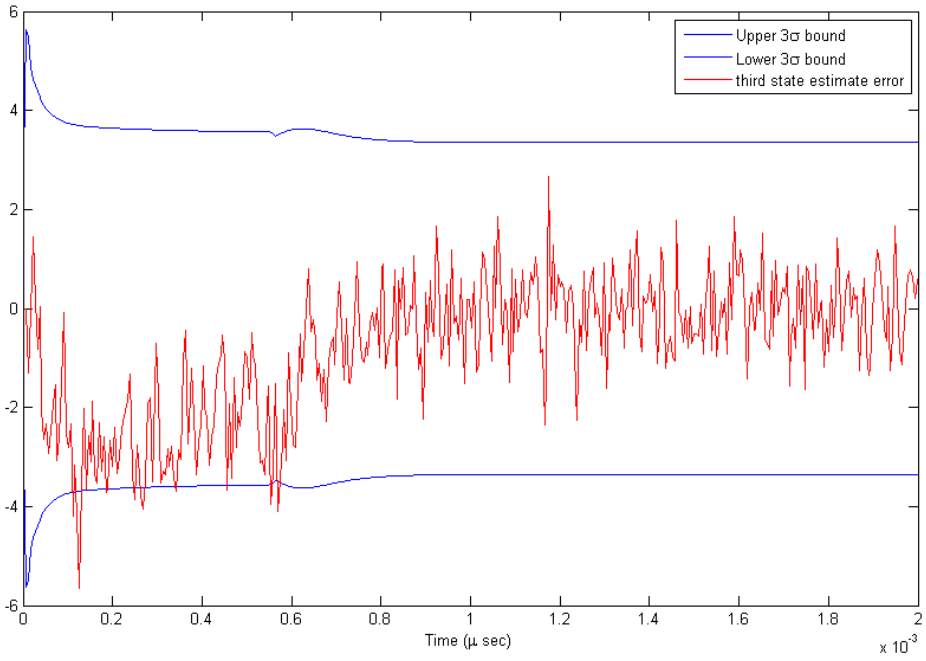

Figure 4.7: State three estimate error and $\pm 3\sigma$ bounds.

obvious indication of this can be seen in Figure 4.8. It should be noted that the frequency graph in this figure is in radians per second. A quick comparison of the amplitude and wave graphs shows that the scales are very different and the amplitude dip present in the wave around 60 μs is not reflected in the amplitude graph. The filter however does converge, with the wave position accurately tracking the injector flux waveform. This is illustrated in Figure 4.9. With some further work it should be possible to converge this filter to the correct values of phase and frequency.

It is likely that the convergence issue present in this filter is due to a lack of input information. With only a single input, the solution is not unique at all points. It is possible that this situation can be remedied using only the existing ADCs and samples. By augmenting the system with a second order derivative of the wave position, an additional state can be added. This state can be approximately measured from the single ADC sample by

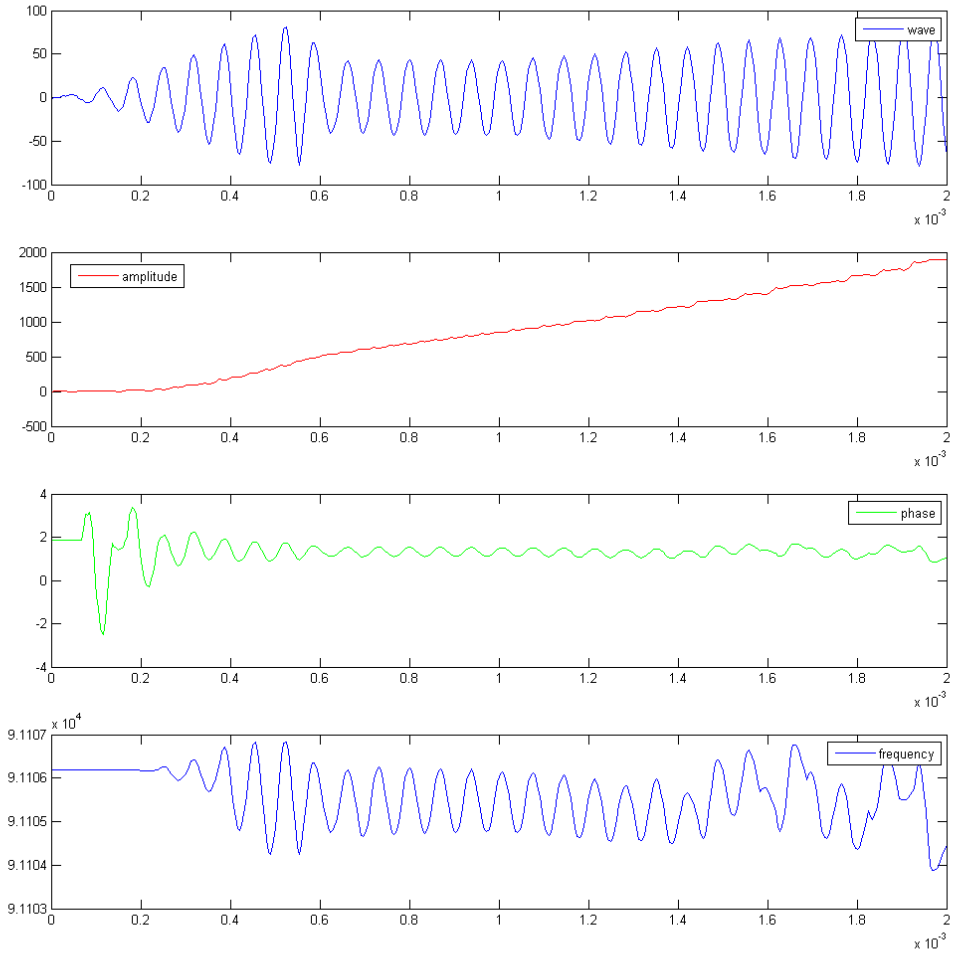

Figure 4.8: Sine wave fitting parameters

finding the difference between consecutive samples. A similar effect could be achieved by summing ADC samples over time and adding a term to represent the integral of the wave position.

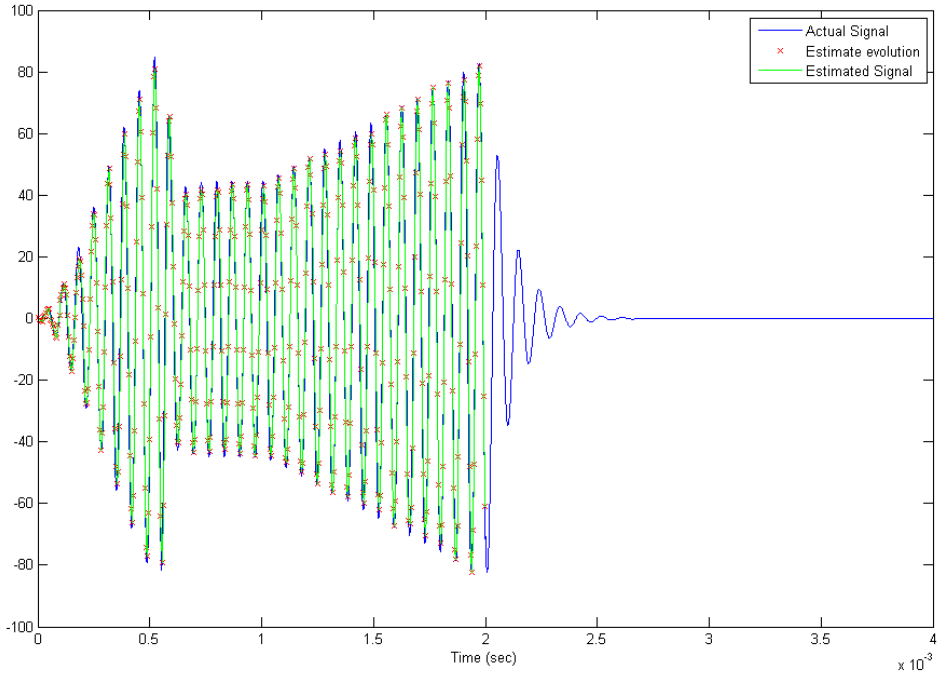

Figure 4.9: Sine fitting filter wave and actual simulation wave

Chapter 5

PARAMETER ESTIMATION

In addition to the implementation of the Kalman Filter for real-time state estimation, it can be used to estimate system parameters. This attribute of the filter makes it ideal for determining the unknown flux circuit parameters.

5.1 Estimation of Flux Circuit Inductance

A comparison of simulation outputs to actual shot flux data for various 'plasma' stage resistances shows that the flux circuit resistance likely undergoes little change during the course of the shot. The flux however, appears to change continuously throughout the 'plasma' stage of the shot. No constant value was found to represent circuit inductance for the duration of this stage. The closest constant value yielded the plots seen in Figure 5.1. It therefore stands to reason that the inductance value must be time dependent. While the exact values may not be possible to determine analytically, an empirical approximation can be found through the filtering of previous experimental data. While a higher order polynomial that exactly represents the evolution of inductance with respect to time would seem desirable, there are several reasons not to filter for such a relation. The time resolution of flux sample data is limited. This is the result of physical limitations on the hardware and software limits. Data collected in real time by Blackfin ADCs, rather than being digitized after the experimental operation, such as injector flux measurements, are necessarily collected at slower time intervals. In addition, to prevent uncontrolled scaling (the effect of having different numbers of samples in subsequent half cycles), the control system specifies a fixed number of samples to be taken per half cycle. Additionally, since the Blackfins have only a single ADC, they are only capable of having a single dimension of measured

output to feed back into the control system.

The combination of this feedback single-dimensionality and the low time resolution of these samples makes computing a higher order fit effectively impossible. Even in the event that a higher order relation were available, it is unlikely that the relation would hold for more than a single shot. The flux circuit inductance variation is caused by changes in the plasma present within the injector. While the plasma changes with time, it is unlikely to change in a completely consistent manner. It does, however, exhibit a general trend of growth throughout the shot. As such, it makes sense to search for a basic linear or quadratic fit.

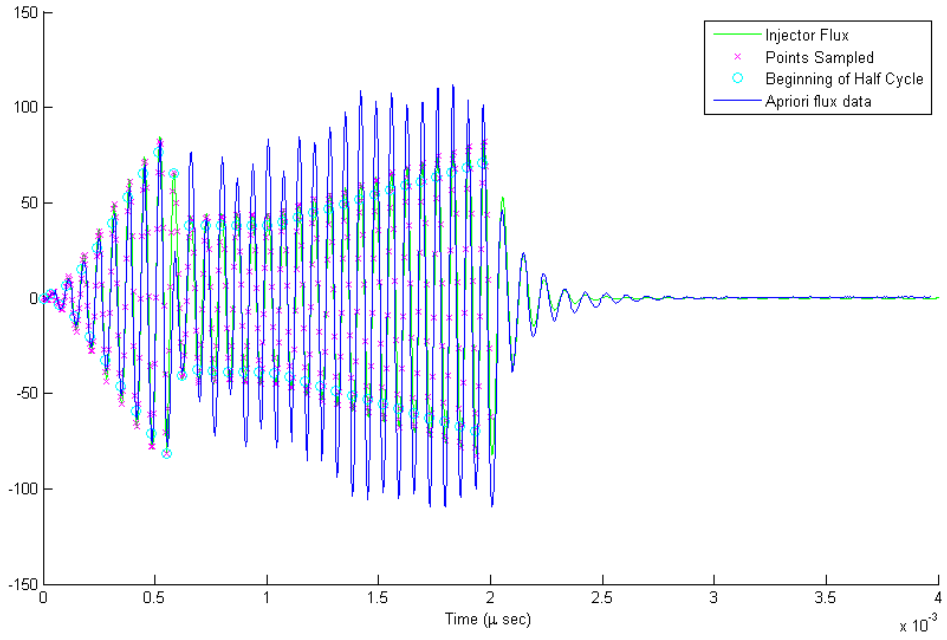

Figure 5.1: Simulation output vs experimental data under preprogrammed control with switching dynamics parameters

5.2 Kalman Algorithm Modification

The basic Kalman Filter which has been discussed in depth undergoes little modification to accommodate this new use. The majority of the modification occurs in the form of state space augmentation. In the case of parameter estimation for the flux circuit, the state space must also include parameters which encompass the changing induction and resistance of the circuit over time. As discussed previously, the estimation is simplified by assuming that the resistance is essentially time-invariant. The inductance is now represented by a linear time dependence with two unknown parameters: $L = L_1 + L_2 t$. Estimating these paramters is the goal of this EKF.

The original system state space took the found in equations 3.14, 3.15, and 3.16 forms the majority of the new system. However, with the addition of the two states, L_1 and L_2, the system becomes nonlinear. Thus it can no longer be represented in the form of $\dot{x} = Ax + Bu$. Equation 3.14 is multiplied by the state vector, x, and becomes the $6x1$ matrix found in 5.1. Similarly 3.15 and 3.16 become 5.3 and 5.4.

$$A = \frac{1}{1-mg-nv} \tag{5.1}$$

$$\begin{bmatrix} x_1(gh-eh(1-nv))+x_2(f(1-nv)-g(k-np))+x_3(g(l-nq))+x_4(gb-a(1-nv)) \\ x_1(h-meh)+x_2(mf-k+np)+x_3(l-nq)+x_4(b-ma)) \\ x_1(v(h-meh))+x_2(p(1-mg)+v(mf-k))+x_3(vl-q(1-mg))+x_4(v(b-ma)) \\ x_1\left(\frac{1}{C}(1-mg-nv)\right)+x_2\left(\frac{-1}{C}(1-mg-nv)\right) \\ L_1 \\ L_2 \end{bmatrix} \tag{5.2}$$

$$B = \frac{1}{1-mg-nv}\begin{bmatrix} a(1-nv) \\ ma \\ vma \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.3}$$

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.4}$$

5.2.1 Implementation

Due to the complexity inherent of estimating additional states, this estimator is best run using only previously obtained experimental data. Thus operation requires an independent script which will operate the estimator using injector flux data from previous shots. To minimize the possibility of introducing additional error via the control system, SPA voltages measured from previous shots were also used. Shot 129499 was chosen because the SPA voltage and injector flux data for this shot was already in the correct format, and all of the additional machine parameters for this shot are well documented.

The script for this filter can be broken into three major components: initialization, filtering, and post processing.

Initialization

This script begins by initializing the system parameters needed to run the simulation. Unlike the control system simulation, this script is limited to analysis of existing data. As such, much of the initialization used to set up the PID and pre programmed control can be forgone. Time stepping, however, does need to be initialized so that the filter can attempt to construct a time dependence for the inductance. Thus, crucial values such as the δt between injector flux samples, and the total experimental run time are defined.

The experimental data are also imported during this stage. While this is a straightforward procedure, it is worth noting that the time bases for the two key signals are not the same. This makes for some additional necessary computation during the estimator operation.

The final component of the initialzation phase is the definition of the system dynamics, defined in equations 5.1, 5.3, and 5.4. These systems are defined for both the vacuum and plasma stage of the shot. The operation of the estimator requires that the Jacobian of the system with respect to its states be taken and evaluated each loop iteration. Since the form of the system does not change throughout the time evolution, this Jacobian can be taken in the initialization and simply evaluated each estimator iteration. This cuts computational intensity significantly, though a function evaluation is still required in each iteration.

Filtering

The extended Kalman filter algorithm is implemented in a loop which cycles through all of the available injector flux samples. At each discrete time point, the loop iterates using the corresponding SPA voltage value. As mentioned previously, these SPA values are in a different time base than the injector flux values. Thus, a brief segment of code is needed

to search for the value most closely corresponding to the experiment time associated with the injector flux sample of the iteration. Once the closest value is found, it is used as the system input for the propagation phase of the algorithm.

Three separate propagation cases are present: one for the vacuum system dynamics, one for plasma dynamics, and one for the freewheel stage at the end of the shot. Separate Jacobians are computed for each of these stages to account for the projected change in the inductance parameters.

Post Processing

The post processing section of this script consists simply of several plotting functions. Plots are generated for the four system states as in the control system simulation script. Additionally, a plot of the inductance parameters over time is generated. A plot of the estimated output overlaid with the experimental output is also generated.

5.3 Results

While the premise of this script is simple and straightforward, it is plagued with complications which make it very difficult to implement successfully. At this point in time, the estimator is not able to progress beyond approximately 100 iterations. This is due in large part to the nonlinear nature of the system. As the estimator iterates, several covariance matrix terms rapidly approach infinity and exceed the computable number range for Matlab. The root of this issue likely lies with the propagation stage of the estimator. The Discrete-discrete EKF requires the Jacobian of the discrete system evaluated at each time step to propogate this covariance matrix. The calculation of the Jacobian is manageable, however, discretizing the system is a different matter.

In linear systems discretization is a trivial task involving a matrix exponential and basic calculations. For nonlinear PDEs, however, the task is considerable more difficult. Often the system can be linearized near a given time or position and discretized once this has been

completed. Unfortunately, in practice this involves linearizing the system at each iteration of the estimator loop. This method would be computationally intensive and therefore has not been pursued.

An alternative would be to implement a continuous-discrete extended Kalman filter. This method is likely to be the best solution. Unfortunately, this method also has inherent difficulties. The continuous-discrete extended Kalman filter returns rates of change for both the state vector and covariance matrix. While evolving these discretely should be a simple matter of multiplying the rate of change by the timestep and adding it to the previous set of values. In practice, however, this method does not produce convergent results. With additional time developing this estimator and its propogation phase, useful results should be attainable.

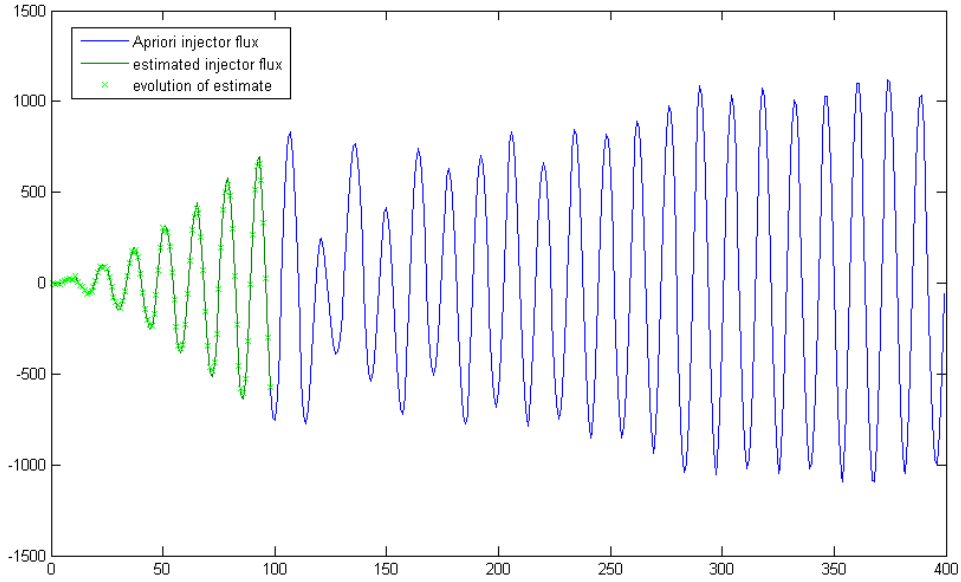

Figure 5.2: Estimate of parameter estimation state two, injector flux coil current, under preprogrammed control and experimental flux waveform

By examining the first 100 iterations of this estimator, it can be seen that it tracks very

well. The evolution of the estimated states tracks. This can be seen in Figure 5.2 for state two, the current generating the injector flux. The estimates of parameters $L1$ and $L2$ remain relatively constant during this period (see Figure 5.4). This is probably due to the inductance values remaining stable during the vacuum portion of the shot. All of the states can be seen in figures 5.3 and 5.4

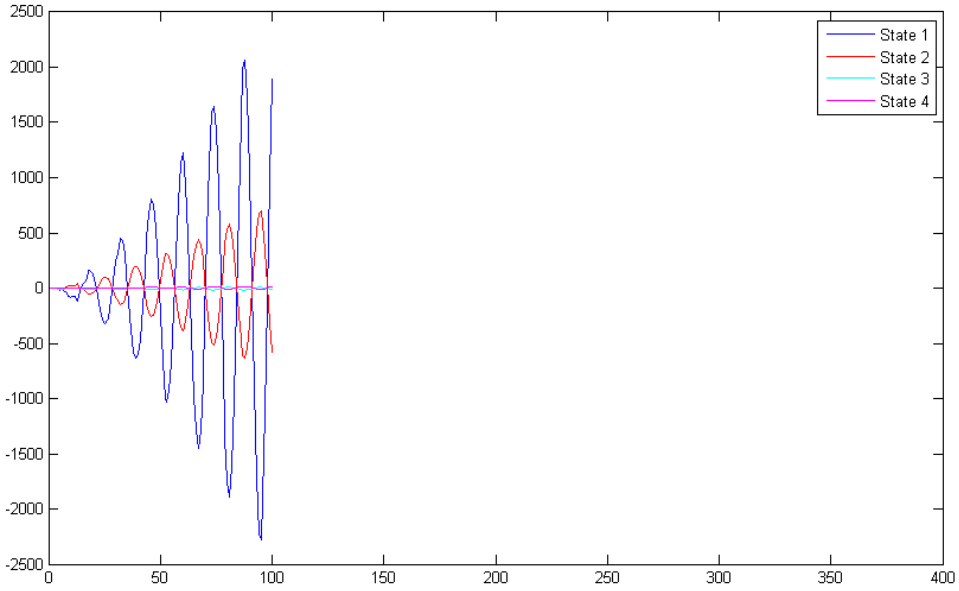

Figure 5.3: Estimated system states 1 through 4: loop current 1, loop current 2, loop current 3, and capacitor voltage

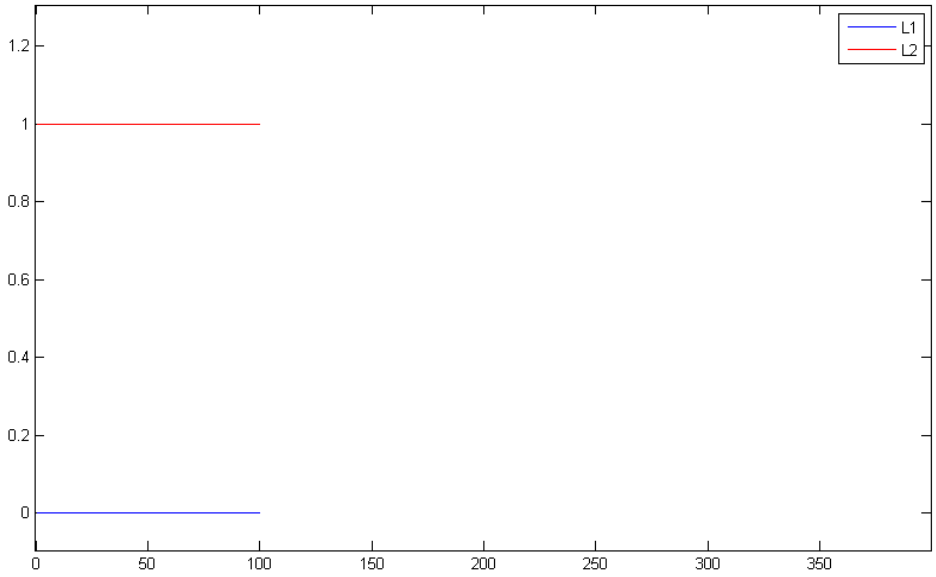

Figure 5.4: Estimate of $L1$ and $L2$

Chapter 6
CONCLUSIONS AND FUTURE WORK

The work conducted developing the control system simulation and state estimation code for the HIT-SI experiment has made several contributions toward the implementation of a viable modern control system. Prior to this research, a working model of the complete PID feedback control system had not been developed. This PID control simulation functions as desired and, with appropriate values for plasma resistance and inductance will generate waveforms that closely match those seen in the actual experiment.

6.1 Conclusions

The full state dynamics Kalman filter successfully converges three of the four system states, the three circuit loop currents. While this appears to be a trivial matter, it is impressive given that the observability of the system is limited to two states; the second and third. Thus the convergence of the fourth state is very impressive and is an excellent demonstration of the usefulness of known measurement noise.

The injector flux coil current is one of the successfully tracked states. Once implemented, if the state estimator exhibits similar behavior to that seen in simulation, it will lock onto the injector flux waveform before the experiment has transitioned from vacuum to plasma stage. This will allow for full visibility of the injector flux in plasma mode and will allow for more advanced computations to take place without the limitiations of the flux sample resolution. While the sine wave fitting is not currently successful, it shows promise and can likely be successfully implemented with some additional development. This would provide a means of identifying flux wave amplitude and phase instantaneously without lengthy additional computations. Both of these filtering methods can form the first

portion of a linear quadratic Gaussian control system. This system is discussed later in this chapter.

Investigation into the time varying injector inductance will need to be continued to allow the simulation model to be verified to a high degree of precision. While the filter presently does not execute fully and halts with errors, the additional work required to bring it to full functionality should be managable. Once this filter functions properly, it should be able to shed new light on the dynamic resistance and inductance of the injectors with plasma present.

6.2 Future Work

Now that a functioning state estimator has been developed, the process of control system development can continue. The next major step will be to identify system parameters which are to be optomized and to construct cost functions (or functionals) which reflect these. The most obvious system might prioritize time optimality (to reduce DC flux offset as quickly as possible) while also weighing capacitor bank voltage droop. Another option might be to target higher flux demand while attempting to minimize overloading of any one circuit component.

Once the goal of the control system is identified, the real-time state estimators can be tweaked or modified to provide information which will be most useful for the feedback gain calculation. One feedback gain calculation (controller) that would work well in this situation would be a Linear-Quadratic Regulator (LQR). Development of this code should be relatively straightforward once the desired goals have been established. This combined with the Kalman filter will provide the experiment with what is called Linear-Quadratic Gaussian control.

While this falls beyond the scope of this research, work in this direction can be carried out with minimal additional manpower and should be within the reach of additional graduate work.

6.3 Additional Reading

Derivations of the Kalman filters used in this paper can be found in Crassidis and Junkins text on optimal control [2]. This text also includes discussion and development of various additional filters and filtering methods. An expanded discussion of linear system modeling and control can be found in Chen's linear systems text [1]. This text also includes more in-depth analysis of observability and a discussion of controllability in linear systems.

BIBLIOGRAPHY

[1] Chi-Tsong Chen. *Linear System Theory and Design, Third Edition*. Oxford University Press, 1999.

[2] John L. Junkins John L. Crassidis. *Optimal Estimation of Dynamic Systems, Second Edition*. CRC Press, 1986.

Appendix A

SIMULATION AND ESTIMATION CODE

A.1 Control Simulation Code

A.1.1 Main Script

breaklines

```
%Matt Kraske
%HIT-SI lab, University of Washington - 2014
%Control system development Jan 9, 2014
%script to simulate control dynamics for HIT-SI Flux circuit time
  evolution

clear variables; clf; close all;
clc;


format long;

% Set the PID connectivity
PID_connectivity='none'; % 'none', 'fullperiod', 'gain aid', 'binary'

%ybig(10) is I_2-I_3, ybig(6) is I_2

%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%    Initialization Phase %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Set up initial parameters --set as per "fbtank_129499_flux_x"

SCLK=1.25e8;          %Blackfin Clock frequency (125 MHz)
shot_length=2000;     %shot length in microseconds (us)
freq_SIHI=14500;      %tank resonant frequency (Hz)
T_period_cts=1/freq_SIHI/2*SCLK     %blackfin ticks per half period was =4310;
n_pulses=shot_length*freq_SIHI*2/1000000;        %number of pulses in shot length was 117
if mod(n_pulses,2)==0
    n_pulses=n_pulses+1;
end
n_adcs_per_bf_period=7; %number of blackfin sample events per SIHI period
dead_time=1;          %no clue
offset=2081;          %ADC sample offset (used for adjusting pos/neg offset --was 1026
vac_cycles=16;        %Number of ramping cycles (8 periods)


T_half_period=1/freq_SIHI/2*SCLK;

fp_gain=410;          %proportional gain for PID
fi_gain=62;           %integral gain for PID
fd_gain=0;            %derivative gain for PID
fp_factor=1000;       %scaling factor for some part of PID
```

```
dem_factor=200;     %scaling factor for some part of PID
min_width_counts=20;    %limit for coercing ramp demand

plasma_period=1/freq_SIHI;
half_plas_per=plasma_period/2;

scaling=180; %scaling factor for DCP_plasma

adc_delta=(T_half_period*2)/n_adcs_per_bf_period;
adc_half_delta=adc_delta/2;




%%%    building DCP Plasma with us timestep
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
values=[0,1,1,1.9,1.9]*scaling;     %target values
timepoints=[0,600,1000,2000,shot_length];   %timepoints in
  microseconds (us)
%stages
fillvalues=[];              %creates empty matrix to expand each
  iteration
valueold=values(1);         %creates an ititial demand value based on
  first desired value in values array
for ind_stage=1:length(values)-1        %loop with iterations for each
   major slope stage
    slope=(values(ind_stage+1)-values(ind_stage))/...
        (timepoints(ind_stage+1)-timepoints(ind_stage));    %calculate
           slope
    for ind_step=timepoints(ind_stage)+1:timepoints(ind_stage+1) %loop
       for each timepoint (us)
        value=valueold+slope;           %changes demand based on
          previous demand and target slope
        valueold=value;                 %saves current value for use
```

```
          next iteration above
69        currenttimeindex=[value;ind_step];          %builds vector
          with demand and timestep
70        fillvalues=[fillvalues,currenttimeindex];   %places vector at
          end of matrix
71    end
72 end
73 fillvalues; %rename
74 figure(1)
75 %subplot(2,1,1)
76 plot(fillvalues(2,:),fillvalues(1,:))   %plot to test
77 hold on
78
79
80 %%%    building DCP Plasma with cycle timestep
   %%%%%%%%%%%%%%%%%%%%%%%%%%
81 uspercycle=4000/117;
82 dcplasma=[];
83 for ind_sort=1:n_pulses-1
84     cycle=fillvalues(:,round(ind_sort*uspercycle));
85     dcplasma=[dcplasma,cycle];
86 end
87 plot(dcplasma(2,:),dcplasma(1,:),'r')   %plot to test
88 ylabel('DCP Plasma');
89 xlabel('Time (sec)');
90 legend('Generated Signal','Time base corrected');
91 drawnow
92
93
94 %%%    Initialize loop variables
   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
95 T_max_half_width=T_half_period-dead_time-min_width_counts;
96 min_width_counts=20; %1.6e-7;
```

```
97  T_half_width=1;
98  neg_error_sum=0;
99  pos_error_sum=0;
100 neg_error_prev=0;
101 pos_error_prev=0;
102 n_samples=n_adcs_per_bf_period;
103 sample_step=half_plas_per/n_samples;
104 sample_delta=half_plas_per/(n_samples);
105 N=n_adcs_per_bf_period*n_pulses;
106
107 X0=[0;0;0;0;0];   %no energy initial condition
108 t0=0;                        %start at time=0
109
110 %%%    The following matrices are initialized null and expaned via
        %%%
111 %%%    concatenation. While this is not "correct" it is done buecause
     %%%
112 %%%    the variable interval solver used has no constraint on number
   of %%%
113 %%%    timesteps.
                                                                  %%%
114 utotal=[];
115 xtotal=[];
116 ytotal=[];
117 ttotal=[];
118 curpol=false(1);
119 pos_sum=zeros(1,n_pulses);
120 samplepts=[];
121 pos_error=[];
122 neg_error=[];
123 sample_timer=0;
124 kferror=[];
125 kfxtotal=[];
```

```
126 kfKtotal=[];
127 kfdenomtotal=[];
128 sigmatotal=[];
129 kfxoldtotal=[];
130 ekfxoldtotal=[];
131
132 % kfxtotal=zeros(5,N);
133 % kfKtotal=zeros(5,N);
134 % kfdenomtotal=zeros(1,N);
135 % sigmatotal=zeros(5,N);
136
137
138 startmarker=[];
139
140 %%%    Initialize Kalman Filter
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
141 kfx=zeros(4,1);
142 kfxold=zeros(4,1);
143 kfP=eye(4);
144 kfH=[0,1,0,0];
145 kfups=ones(4,1);
146 Rnoise=1;
147 Qnoise=1;
148
149 %Tank Circuit Dynx Constants (vacuum)
150 L_s=2.47e-6;    %series inductance (with source)
151 R_s=5e-3;   %series resistance (with source)
152 L_c=100e-9; %capacitor leg inductance
153 C=95e-6;    %capacitance
154 R_c=2e-3;   %capacitor leg resistance
155 L_cab=0;    %cabling inductance
156 R_cab=0;    %cabling resistance
157
```

```
158 L_p=8.21e-6;    %parallel (to load) inductance/primary inductance
159 R_p=5e-3;   %parallel (to load) resistance/primary resistance
160 L=1.85e-6;  %coil inductance/(transformed) plasma inductance (vacuum)
161 R=20e-3;    %coil resistance/(transformed) plasma resistance (vacuum)
162
163 %Tank Circuit Dynx derived quantities
164 a=1/(L_s+L_c);
165 b=1/(L_cab+L_p+L_c);
166 eh=(R_s+R_c)*a;
167 f=R_c*a;
168 g=L_c*a;
169 h=R_c*b;
170 k=(R_cab+R_p+R_c)*b;
171 l=R_p*b;
172 m=L_c*b;
173 n=L_p*b;
174 p=R_p/(L+L_p);
175 q=(R+R_p)/(L+L_p);
176 v=L_p/(L+L_p);
177
178 %Tank Circuit Dynx SS (no flux)
179 A1=[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-a*(1-n*v)];
180 A2=[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
181 A3=[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b-m*a)];
182 A4=[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];
183
184 kfAvac=(1/(1-m*g-n*v)).*[A1;A2;A3;A4];
185 kfBvac=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0];
186
187 %Tank Circuit Dynx Constants (plasma)
188 L_p=8.21e-6;    %parallel (to load) inductance/primary inductance
189 R_p=5e-3;   %parallel (to load) resistance/primary resistance
190 L=.9e-6;    %coil inductance/(transformed) plasma inductance (plasma)
```

```
R=10e-2;    %coil resistance/(transformed) plasma resistance (plasma)

%Tank Circuit Dynx derived quantities
a=1/(L_s+L_c);
b=1/(L_cab+L_p+L_c);
eh=(R_s+R_c)*a;
f=R_c*a;
g=L_c*a;
h=R_c*b;
k=(R_cab+R_p+R_c)*b;
l=R_p*b;
m=L_c*b;
n=L_p*b;
p=R_p/(L+L_p);
q=(R+R_p)/(L+L_p);
v=L_p/(L+L_p);

%Tank Circuit Dynx SS (no flux)
A1=[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-a*(1-n*v)];
A2=[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
A3=[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b-m*a)];
A4=[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];

kfAplas=(1/(1-m*g-n*v)).*[A1;A2;A3;A4];
kfBplas=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0];

sysvac=ss(kfAvac,kfBvac,kfH,[0]);
sysplas=ss(kfAplas,kfBplas,kfH,[0]);

whyscale=1;
dsysvac=c2d(sysvac,whyscale*sample_delta,'zoh');
dsysplas=c2d(sysplas,whyscale*sample_delta,'zoh');

```

```
224 %sine EKF initialization
225 ekfxtotal=[];
226 ekfx=zeros(4,1);
227 Upsilon = [ .001 0 0 0;
228       0 0.1 0 0;  %freq
229       0 0 10 0;       %amp
230       0 0 0 .1];    %phase
231
232 %   Upsilon = [ 1 0 0 0;
233 %       0 0.1 0 0;
234 %       0 0 0.1 0;
235 %       0 0 0   1];
236 % Upsilon = [ 1 0 0 0;
237 %       0 0.001 0 0;  %freq
238 %       0 0 10 0;       %amp
239 %       0 0 0 .01];    %phase
240
241
242 x_apri = zeros(4,N);
243 x_apost = zeros(4,N);
244 P_apri = zeros(4,4,N);
245 P_apost = zeros(4,4,N);
246 Pcov = zeros(4,N);
247 yekf = zeros(1,N);
248 Kekf = zeros(4,N);
249
250 ekfx=[ 0;
251       14500*2*pi;   %      14500*2*pi;
252       0;
253       .6*pi];
254 Qekf = [.001,0,0,0;
255         0,.001,0,0;
256         0,0,.001,0;
```

```
257         0,0,0,.001];
258 Rekf = .10;
259 P_apri(:,:,1) = eye(4);
260 Pcov(:,1) = diag(P_apost(:,:,1));
261 wekf = randn(4,N);      %state noise
262 vekf = randn(1,N);      %measurement noise
263
264 fekf = @(x,w,t) [ x(3)*sin(x(2)*t+x(4));
265              x(2);
266              x(3);
267              x(4)] + w;
268 % Fekf = @(x,t) [  0   x(3)*t*cos(x(2)*t+x(4)) sin(x(2)*t+x(4)) x(3)*
    cos(x(2)*t+x(4));
269 %                 0              0              0         0;
270 %                 0              0              .1         0;
271 %                 0              0              0         .01];
272
273
274 Fekf = @(x,t) [  0   x(3)*t*cos(x(2)*t+x(4)) sin(x(2)*t+x(4)) x(3)*cos
    (x(2)*t+x(4));
275    0              1              0         0;
276    0              0              1         0;
277    0              0              0         1];
278
279
280 %%%    Import actual flux waveform
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
281 realflux = importdata('fbtank_129499_flux_x.dat',' ',20);
282 realfluxindex=realflux.data(:,1);
283 realfluxvalue=realflux.data(:,2);
284 realfluxtimebase=linspace(0,.004-(half_plas_per),(4000*freq_SIHI
    *2/1000000)*n_samples)'; %+(.25*plasma_period)
285 %realfluxtimebase=2000/SCLK:adc_half_delta/SCLK:shot_length+2000/SCLK;
```

```
%phase shifted
%realfluxtimebase=0:adc_half_delta/SCLK:shot_length; %not phase shifted
realfluxvalue=realfluxvalue(1:end-7,1);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Experiment Execution Code %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% tank circuit iteration
%this phase takes place using variable step solvers to evolve through time.
%These solvers are halted at events so that inputs can be changed

for ind_cycle=1:n_pulses-1
    display(' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ')
    display(' %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ')
    cycle=ind_cycle


    %%%    Feedback control system
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if ind_cycle<n_pulses
        level=pos_sum;
        target=dcplasma(1,ind_cycle);
        if ind_cycle>0%<vac_cycles+1   %vacuum ramping stage
            display('PPG')

            T_half_width=dcplasma(1,ind_cycle)*dem_factor/1000; % T_max_half_width; %
            if T_half_width<=0; %0
                T_half_width=min_width_counts;
            end
            if T_half_width>T_max_half_width
                T_half_width=T_max_half_width;
            end
            %          if(mod(cycle,2)==1)
            %              T_half_width_prev_neg=T_half_width;
            %          else
            %              T_half_width_prev_pos=T_half_width;
            %          end

            T_half_width
        else    %plasma stage
            display('PID')
            if(mod(ind_cycle,2)==1)
                display('even') %even cycle
                last_cycle=ind_cycle-2;
                pos_sum(last_cycle)
                pos_error(ind_cycle)=dcplasma(1,ind_cycle)-pos_sum(last_cycle);
                display('pos_error')
                display(pos_error(ind_cycle))
                pos_error_sum=pos_error_sum+pos_error(ind_cycle);
```

```
335                 T_half_width=(pos_error(ind_cycle)*fp_gain)/fp_factor
                      +(pos_error_sum*fi_gain)/fp_factor...
336                     -((pos_error(ind_cycle)-pos_error(ind_cycle-2))*
                          fd_gain)/fp_factor;
337                 pos_error_prev=pos_error(ind_cycle);
338
339                 switch PID_connectivity
340                     case 'none'
341
342                     case 'fullperiod'
343                         %not defined
344                     case 'gain aid'
345                         %not defined
346                     case 'binary'
347                         if pos_sum(ind_cycle-1)<=0
348                             T_half_width=0;
349                         end
350                 end
351                 if T_half_width<=0
352                     T_half_width=min_width_counts;
353                 end
354                 if T_half_width>T_max_half_width
355                     T_half_width=T_max_half_width;
356                 end
357                 error(1,ind_cycle)=dcplasma(1,ind_cycle)-pos_sum(1,
                      last_cycle);
358             else
359                 display('odd')  %odd cycle
360                 last_cycle=ind_cycle-2;
361                 pos_sum(last_cycle)
362                 neg_error(ind_cycle)=dcplasma(1,ind_cycle)+pos_sum(
                      last_cycle);
363                 display('neg_error')
```

```
364                 display(neg_error(ind_cycle))
365                 neg_error_sum=neg_error_sum+neg_error(ind_cycle);
366                 T_half_width=(neg_error(ind_cycle)*fp_gain)/fp_factor
                      +(neg_error_sum*fi_gain)/fp_factor...
367                       -((neg_error(ind_cycle)-neg_error(ind_cycle-2))*
                            fd_gain)/fp_factor;
368                 neg_error_prev=neg_error(ind_cycle);
369
370                 switch PID_connectivity
371                     case 'none'
372
373                     case 'fullperiod'
374                         %not defined
375                     case 'gain aid'
376                         %not defined
377                     case 'binary'
378                         if pos_sum(ind_cycle-1)<=0
379                             T_half_width=0;
380                         end
381                 end
382                 if T_half_width<=0; %0
383                     T_half_width=min_width_counts;
384                 end
385                 if T_half_width>T_max_half_width
386                     T_half_width=T_max_half_width;
387                 end
388
389                 error(1,ind_cycle)=pos_sum(1,last_cycle)+dcplasma(1,
                      ind_cycle);
390             end
391             error(2,ind_cycle)=tbig(1)+half_plas_per;
392             display('width')
393             display(T_half_width)
```

```
        end
    end


    %1- calculate the two crucial timespoints: 'v_on', and 'v_off'
    pulse_duration=T_half_width/SCLK;
    v_on=t0+(half_plas_per-pulse_duration)/2;
    v_off=v_on+pulse_duration;
    tf=t0+half_plas_per;


    % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% The following code snipped was implemented to investigate the option %%
    %%% of adjusting controller phase once the plasma had broken down. It is %%
    %%% no longer being used but has been left for future testing.          %%
    % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % if ind_cycle==vac_cycles
    %     phaseshift=t0+half_plas_per/2;
    %     [t4,y4]=ode23(@(t,X) dynamicsfuncplas(t,X),[t0,phaseshift],X0);
    %     X0=y4(end,:)';
    %     t0=phaseshift;
    % else
    %     t4=[];
    %     y4=[];
    % end
```

```
419
420     %%%    Flux Circuit System Dynamics Time Evolution
          %%%%%%%%%%%%%%%%%%%%%
421     if ind_cycle<=vac_cycles    %vacuum ramping stage    ==0% >0%
422         display('vac')
423         %2- section 1: pre powered pulse (t=o to 'v_on')
424         U=0;                          %zero voltage section
425         X0(5,1)=U; %Troubleshooting method
426         [t1,x1]=ode45(@(t,X) dynamicsfuncvac(t,X),[t0,v_on],X0);
427
428         %3- section 2: during powered pulse ('v_on' to 'v_off')
429         swt=1;
430         X0=x1(end,:)';                %begin where the last section
              ended
431         if(curpol==1)    %odd cycle
432             U=-450*swt;                    %voltage active section
433         else
434             U=450*swt;                   %voltage active (negative)
                section
435         end
436         X0(5,1)=U; %Troubleshooting method
437         %!!-EVENTUALLY NEED TO IMPLEMENT TIME DEPENDENT U TO SIM VOLT
              DROOP-!!
438         [t2,x2]=ode45(@(t,X) dynamicsfuncvac(t,X),[v_on,v_off],X0);
439
440         %4- section 3: post powered pulse ('v_off to t=half_plas_per)
441         X0=x2(end,:)';                %begin where last section ended
442         U=0;                          %zero voltage section
443         X0(5,1)=U; %Troubleshooting method
444         [t3,x3]=ode45(@(t,X) dynamicsfuncvac(t,X),[v_off,tf],X0);
445     else ind_cycle<=n_pulses%%%%
446         display('plas')
447         %2- section 1: pre powered pulse (t=o to 'v_on')
```

```
448         U=0;                          %zero voltage section
449         X0(5,1)=U; %Troubleshooting method
450         [t1,x1]=ode45(@(t,X) dynamicsfuncplas(t,X),[t0,v_on],X0);
451
452         %3- section 2: during powered pulse ('v_on' to 'v_off')
453         swt=1;
454         X0=x1(end,:)';                  %begin where the last section
              ended
455         if(curpol==1)    %odd cycle
456             U=-450*swt;                    %voltage active section
457         else
458             U=450*swt;                   %voltage active (negative)
                section
459         end
460         X0(5,1)=U; %Troubleshooting method
461         %!!-EVENTUALLY NEED TO IMPLEMENT TIME DEPENDENT U TO SIM VOLT
              DROOP-!!
462         [t2,x2]=ode45(@(t,X) dynamicsfuncplas(t,X),[v_on,v_off],X0);
463         %4- section 3: post powered pulse ('v_off to t=half_plas_per)
464         X0=x2(end,:)';                %begin where last section ended
465         U=0;                          %zero voltage section
466         X0(5,1)=U; %Troubleshooting method
467         [t3,x3]=ode45(@(t,X) dynamicsfuncplas(t,X),[v_off,tf],X0);
468     end
469
470
471     %%%    Sampling portion
              %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
472     %5- post process
473     tbig=[t1;t2;t3];
474     xbig=[x1;x2;x3];
475     startmarker=[startmarker,[xbig(1,2);tbig(1,1)]];
476
```

```
%%%    Observe system %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
in1=zeros(length(t1),1);
if(curpol==1)    %odd cycle
    in2=-450*swt*ones(length(t2),1);                    %voltage active section
else
    in2=450*swt*ones(length(t2),1);                     %voltage active (negative) section
end
in3=zeros(length(t3),1);
ubig=[in1;in2;in3];

%sample_delta=(tbig(end)-tbig(1))/(n_samples);

%6- utilize Kalman filter for signal tracking
%store this info for use in PID calculation
%!!-APPENDING WITH Cont/Disc EKF-!!

%%%    sampling and KF %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sample_timer=tbig(1);
pos_sum1=0;
for ind_samp=1:n_samples
    sample_timer=sample_timer+sample_delta/2;
    for ind_search=1:length(tbig)
        if tbig(ind_search,1)>=sample_timer || ind_search==length(tbig)
            if ind_search==1;
                sample_ind=ind_search;
                break
            elseif tbig(ind_search)-sample_timer>tbig(ind_search-1)-sample_timer
```

```
504                     sample_ind=ind_search;
505                 else
506                     sample_ind=ind_search-1;
507                 end
508                 break
509             end
510         end
511         sample_timer2=sample_timer+sample_delta;
512         for ind_search2=1:length(tbig)
513             if tbig(ind_search2,1)>=sample_timer2 || ind_search2==
                  length(tbig)
514                 if ind_search2==1;
515                     sample_ind2=ind_search2;
516                     break
517                 elseif tbig(ind_search2)-sample_timer2>tbig(
                      ind_search2-1)-sample_timer2
518                     sample_ind2=ind_search2;
519                 else
520                     sample_ind2=ind_search2-1;
521                 end
522                 break
523             end
524         end
525                 sample_timer3=sample_timer-sample_delta;
526                 for ind_search3=1:length(tbig)
527             if tbig(ind_search3,1)>=sample_timer3 || ind_search3==
                  length(tbig)
528                 if ind_search3==1;
529                     sample_ind3=ind_search3;
530                     break
531                 elseif tbig(ind_search3)-sample_timer3>tbig(
                      ind_search3-1)-sample_timer3
532                     sample_ind3=ind_search3;
```

```
533                 else
534                     sample_ind3=ind_search3-1;
535                 end
536                 break
537             end
538         end
539         pt1=xbig(sample_ind,2)+1*(randn(1,1)+0); %;%
540         samptime=tbig(sample_ind,1);
541         samplepts=[samplepts,[pt1;samptime]];
542         pos_sum1=pos_sum1+pt1;
543
544         %begin KF iteration  --linear D-D KF
545         kfdenom=kfH*kfP*kfH'+Rnoise;
546         kfK=kfP*kfH'/kfdenom;
547         steperror=pt1-kfH*kfx;
548         kfx=kfx+kfK*(steperror);
549         kfP=(eye(4)-kfK*kfH)*kfP;
550
551         kferror=[kferror,[steperror;samptime]];
552         kfxtotal=[kfxtotal,[kfx;samptime]];
553         kfKtotal=[kfKtotal,[kfK;samptime]];
554         kfdenomtotal=[kfdenomtotal,kfdenom];
555         sigmatotal=[sigmatotal,[3*sqrt(diag(kfP));samptime]];
556
557         evoinput=ubig(sample_ind:sample_ind2,1);
558         evoinputav=mean(evoinput)*.08;%<-------!!not sure why scaling
              needed!!
559
560         if ind_cycle<=vac_cycles %vac stage     >0%
561             kfxapost=dsysvac.a*kfx+dsysvac.b*evoinputav; %*ubig(
                  sample_ind,1); %
562             %kfx=kfx+(kfAvac*kfx+kfBvac*ubig(sample_ind,1))*
                  sample_delta;
```

```
563             kfP=dsysvac.a*kfP*dsysvac.a'+kfups*Qnoise*kfups';
564             %kfP=kfAvac*kfP*kfAvac'+kfups*Qnoise*kfups';
565         else    %plas stage
566             kfxapost=dsysplas.a*kfx+dsysplas.b*evoinputav; %*ubig(
                  sample_ind,1); %
567             %kfx=kfx+(kfAplas*kfx+kfBplas*ubig(sample_ind,1))*
                  sample_delta;
568             kfP=dsysplas.a*kfP*dsysplas.a'+kfups*Qnoise*kfups';
569             %kfP=kfAplas*kfP*kfAplas'+kfups*Qnoise*kfups';
570         end
571         kfx=kfxapost;
572         kfxoldtotal=[kfxoldtotal,[kfxapost;samptime+sample_delta]];
573         sample_timer=sample_timer+sample_delta/2;
574
575
576
577
578         %sine wave EKF  --nonlinear cont-disc (with Fwd Euler cont
579         %evolution) EKF
580         Hx=[1 zeros(1,3)];    % Measurement-state Jacobian
581
582         Sekf=(Hx*P_apri(:,:,ind_cycle)*Hx'+Rekf);   % Update Kalman
              gain
583         Kekf(:,ind_cycle)=P_apri(:,:,ind_cycle)*Hx'/Sekf;
584
585         ekfx=ekfx+Kekf(:,ind_cycle)*(pt1-ekfx(1));   % Update
              aposteriori state estimate
586         P_apost(:,:,ind_cycle)=(eye(4)-Kekf(:,ind_cycle)*Hx)*P_apri
              (:,:,ind_cycle);  % Update aposteriori error covariance
              estimate
587         %ekfx=ekfx+[ekfx(3)*sin(ekfx(2)*(samptime)+ekfx(4));0;0;0].*
              sample_delta;% Update apriori estimate
588
```

```
589         ekfxtotal=[ekfxtotal,[ekfx;samptime]];
590
591         Fx=Fekf(ekfx,samptime+sample_delta);    % Update state
              Jacobian %(ind_cycle)*sample_delta
592         fx=fekf(ekfx,0,samptime+sample_delta);
593         ekfxapost=ekfx+Fx*[1;1;1;1].*sample_delta;% Update apriori
              estimate
594         %ekfxapost=fx;
595         %P_apri(:,:,ind_cycle+1)=P_apost(:,:,ind_cycle)+(Fx*P_apost
              (:,:,ind_cycle)+P_apost(:,:,ind_cycle)*Fx'+Upsilon*Qekf*
              Upsilon').*sample_delta;   % Update aprioiri error
              covariance estimate
596         P_apri(:,:,ind_cycle+1)=Fx*P_apost(:,:,ind_cycle)*Fx'+Upsilon*
              Qekf*Upsilon';   % Update aprioiri error covariance estimate
597
598         Pcov(:,ind_cycle+1)=diag(P_apost(:,:,ind_cycle+1))';
599         ekfx=ekfxapost;
600         ekfxoldtotal=[ekfxoldtotal,[ekfxapost;samptime]];%+
              sample_delta]];
601
602     end
603     current=pos_sum1/3;
604     scale=.1;   %%% SCALING TERM FOR ADC SIMULATION
605     %%%    -> .10 seems to be correct based on a comparison of the ADC
              data for
606     %%%    an entire shot vs the vacuum only pre-programmed only
              values for the
607     %%%    complete simulation. Using a scaling factor of 1/10 on the
              real ADC
608     %%%    graph, it was found to match the simulation almost
              perfectly for the
609     %%%    preprogrammed stage. See figure ()
610     pos_sum(ind_cycle)=pos_sum1*scale;
```

```
    sample_timer=sample_timer+sample_delta;

    %loop finalization steps
    curpol=not(curpol);
    X0=xbig(end,:)';
    t0=tbig(end);
    utotal=[utotal;ubig];
    xtotal=[xtotal;xbig];
    %ytotal=[ytotal;ybig];
    ttotal=[ttotal;tbig];
end

X0=xtotal(end,:)';
U=0;                         %zero voltage section
X0(5,1)=U; %Troubleshooting method
[t1,x1]=ode23(@(t,X) dynamicsfuncspaoff(t,X),[ttotal(end),.004],X0);

u1=zeros(length(t1),1);
utotal=[utotal;u1];
xtotal=[xtotal;x1];
ttotal=[ttotal;t1];
%H=[0,0,L,0]; comm'd 1/8/14 MK (this piece was an observability
  handicap for KF research AA549)
H=[1,0,0,0;0,1,0,0;0,0,1,0;0,0,0,1;1,-1,0,0;0,1,-1,0];

ytotal=zeros(6,length(xtotal));
for indobs=1:length(xtotal)
    ytotal(:,indobs)=H*(xtotal(indobs,1:4)');
end
ytotal=ytotal';
%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Post-Processing
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%
xerrortotal=zeros(2,length(kfxtotal));

for inderror=1:length(kfxtotal)
    timepoint=kfxtotal(5,inderror);
    for inderrorsearch=1:length(ttotal)
        if ttotal(inderrorsearch,1)>=timepoint || inderrorsearch==
          length(ttotal)
                if inderrorsearch==1;
                    errorind=inderrorsearch;
                    break
                elseif ttotal(inderrorsearch)-timepoint>ttotal(
                  inderrorsearch-1)-timepoint
                    errorind=inderrorsearch;
                else
                    errorind=inderrorsearch-1;
                end
                break
        end
    end
    xerrortotal(1:5,inderror)=kfxtotal(:,inderror)-xtotal(errorind,:)
      ';
    xerrortotal(6,inderror)=timepoint;
end
```

```
666
667 figure(2)
668 %subplot(3,1,1)
669 %plot(ttotal,ytotal(:,5),'b')
670 %subplot(3,1,1)
671 %subplot(2,1,2)
672 %plot(ttotal,utotal/4.5,'r');
673 hold on
674 plot(ttotal,xtotal(:,2),'g' )
675 %plot(ttotal,ytotal(:,2),'m:' )
676 plot(samplepts(2,:),samplepts(1,:),'m x')
677 plot(startmarker(2,:),startmarker(1,:),'c o')
678 %subplot(3,1,2)
679 %plot(error(2,:),error(1,:))
680 xlabel('Time (sec)');
681 %legend('Injector Flux','Points Sampled','Beginning of Half Cycle');
682
683 %subplot(3,1,3)
684 plot(realfluxtimebase,realfluxvalue/10)
685 legend('Injector Flux','Points Sampled','Beginning of Half Cycle','
    Apriori flux data');
686
687 %Simulation vs KF results
688 figure(3)
689 %subplot(4,1,1)
690 plot(ttotal,xtotal(:,1),'b',kfxoldtotal(5,:),kfxoldtotal(1,:),'r',
    kfxtotal(5,:),kfxtotal(1,:),'xg')
691 xlabel('Time (sec)');
692 legend('Actual Signal','Estimated Signal','Estimate evolution');
693
694 figure(4)
695 %subplot(4,1,2)
696 plot(ttotal,xtotal(:,2),'b',kfxoldtotal(5,:),kfxoldtotal(2,:),'or',
```

```
  kfxtotal(5,:),kfxtotal(2,:),'xg');%,ekfxtotal(5,:),ekfxtotal(1,:),'g
  ')
xlabel('Time (sec)');
legend('Actual Signal','Estimated Signal','Estimate evolution');

figure(5)
%subplot(4,1,3)
plot(ttotal,xtotal(:,3),'b',kfxoldtotal(5,:),kfxoldtotal(3,:),'r',
  kfxtotal(5,:),kfxtotal(3,:),'xg')
xlabel('Time (sec)');
legend('Actual Signal','Estimated Signal','Estimate evolution');

figure(6)
%subplot(4,1,4)
plot(ttotal,xtotal(:,4),'b',kfxoldtotal(5,:),kfxoldtotal(4,:),'r',
  kfxtotal(5,:),kfxtotal(4,:),'xg')
xlabel('Time (sec)');
legend('Actual Signal','Estimated Signal','Estimate evolution');

%kf error and 3-sigma bounds
figure(7)
%subplot(2,1,1)
plot(kfKtotal(5,:),kfKtotal(1:4,:));
xlabel('Time (sec)');
legend('First State Gain','Second State Gain','Third State Gain','
  Fourth State Gain');
figure(8)
%subplot(2,1,2)
plot(sigmatotal(5,:),sigmatotal(1,:),'b',sigmatotal(5,:),-sigmatotal
  (1,:),'b',xerrortotal(6,:),xerrortotal(2,:),'r'); %,kferror(2,:),
  kferror(1,:),'g');
xlabel('Time (sec)');
legend('Upper 3\sigma bound','Lower 3\sigma bound','second state
```

```
  estimate error');
723
724 figure(11)
725 %subplot(2,1,2)
726 plot(sigmatotal(5,:),sigmatotal(1,:),'b',sigmatotal(5,:),-sigmatotal
    (1,:),'b',xerrortotal(6,:),xerrortotal(1,:),'r'); %,kferror(2,:),
    kferror(1,:),'g');
727 xlabel('Time (sec)');
728 legend('Upper 3\sigma bound','Lower 3\sigma bound','first state
    estimate error');
729 figure(12)
730 %subplot(2,1,2)
731 plot(sigmatotal(5,:),sigmatotal(1,:),'b',sigmatotal(5,:),-sigmatotal
    (1,:),'b',xerrortotal(6,:),xerrortotal(3,:),'r'); %,kferror(2,:),
    kferror(1,:),'g');
732 xlabel('Time (sec)');
733 legend('Upper 3\sigma bound','Lower 3\sigma bound','third state
    estimate error');
734 figure(13)
735 %subplot(2,1,2)
736 plot(sigmatotal(5,:),sigmatotal(1,:),'b',sigmatotal(5,:),-sigmatotal
    (1,:),'b',xerrortotal(6,:),xerrortotal(4,:),'r'); %,kferror(2,:),
    kferror(1,:),'g');
737 xlabel('Time (sec)');
738 legend('Upper 3\sigma bound','Lower 3\sigma bound','fourth state
    estimate error');
739
740 %EKF converged values
741 figure(9)
742 subplot(4,1,1)
743 plot(ekfxtotal(5,:),ekfxtotal(1,:))
744 legend('wave')
745 subplot(4,1,2)
```

```
746 plot(ekfxtotal(5,:),ekfxtotal(3,:),'r')
747 legend('amplitude')
748 subplot(4,1,3)
749 plot(ekfxtotal(5,:),ekfxtotal(4,:),'g')
750 legend('phase')
751 subplot(4,1,4)
752 plot(ekfxtotal(5,:),ekfxtotal(2,:),'b')
753 legend('frequency')
754
755 figure(10)
756 plot(ttotal,xtotal(:,2),'b',ekfxoldtotal(5,:),ekfxoldtotal(1,:),'xr',
    ekfxtotal(5,:),ekfxtotal(1,:),'g')
757 xlabel('Time (sec)');
758 legend('Actual Signal','Estimate evolution','Estimated Signal');
```

A.1.2 Dynamics Functions

breaklines

```
1  %Matt Kraske
2  %HIT-SI lab, University of Washington - 2014
3  %Control system development Jan 23, 2014
4  %Function for HIT-SI Flux circuit time evolution with PID control
5
6  function F=dynamicsfuncvac(t,X)
7
8  state=X(1:4,1);
9  U=X(5,1);
10 %input breakdown:
11 %X(1)=I_1    loop 1 current
12 %X(2)=I_2    loop 2 current
13 %X(3)=I_3    loop 3 current
14 %X(4)=V_C   capacitor voltage
15 %U=V_s    supply voltage
```

```
16
17 %Tank Circuit Dynx Constants
18 L_s=2.47e-6;    %series inductance (with source)
19 R_s=5e-3;   %series resistance (with source)
20 L_c=100e-9; %capacitor leg inductance
21 C=95e-6;    %capacitance
22 R_c=2e-3;   %capacitor leg resistance
23 L_cab=0;    %cabling inductance
24 R_cab=0;    %cabling resistance
25
26 L_p=8.21e-6;    %parallel (to load) inductance/primary inductance
27 R_p=5e-3;   %parallel (to load) resistance/primary resistance
28 L=1.85e-6;  %coil inductance/(transformed) plasma inductance (plasma)
29 R=20e-3;    %coil resistance/(transformed) plasma resistance (plasma)
30 % L=1e9;    %coil inductance/(transformed) plasma inductance (vac) !
     was e4 MK 4-7-14
31 % R=1e9;    %coil resistance/(transformed) plasma resistance (vac)
32 % L=1e4;    %coil inductance/(transformed) plasma inductance (vac)
33 % R=2e4;    %coil resistance/(transformed) plasma resistance (vac)
34
35 % L_p=1/(1/8.21e-6+1/1.85e-6);
36 % R_p=1/(1/5e-3+1/20e-3);
37 % L=0;
38 % R=1e20;
39
40
41 % %Tank Circuit Dynx Constants (modified)
42 % L_s=0.634e-6; %series inductance (with source)
43 % R_s=2e-3; %series resistance (with source)
44 % L_c=50e-9;    %capacitor leg inductance
45 % C=191e-6; %capacitance
46 % R_c=2e-3; %capacitor leg resistance
47 % L_cab=0;  %cabling inductance
```

```
% R_cab=0;  %cabling resistance
% L_p=2.4e-6;   %parallel (to load) inductance/primary inductance
% R_p=3e-3; %parallel (to load) resistance/primary resistance
% L=1e4;    %coil inductance/(transformed) plasma inductance
% R=2e4;    %coil resistance/(transformed) plasma resistance

%Tank Circuit Dynx derived quantities
a=1/(L_s+L_c);
b=1/(L_cab+L_p+L_c);
eh=(R_s+R_c)*a;
f=R_c*a;
g=L_c*a;
h=R_c*b;
k=(R_cab+R_p+R_c)*b;
l=R_p*b;
m=L_c*b;
n=L_p*b;
p=R_p/(L+L_p);
q=(R+R_p)/(L+L_p);
v=L_p/(L+L_p);

%Tank Circuit Dynx SS (no flux)
A1=[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-a*(1-n*v)];
A2=[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
A3=[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b-m*a)];
A4=[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];

A=(1/(1-m*g-n*v)).*[A1;A2;A3;A4];
B=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0];

F=[A*state+B*U;U];
```

breaklines

```
%Matt Kraske
```

```
%HIT-SI lab, University of Washington - 2014
%Control system development Jan 23, 2014
%Function for HIT-SI Flux circuit time evolution with PID control

function F=dynamicsfuncplas(t,X)

state=X(1:4,1);
U=X(5,1);
%input breakdown:
%X(1)=I_1    loop 1 current
%X(2)=I_2    loop 2 current
%X(3)=I_3    loop 3 current
%X(4)=V_C   capacitor voltage
%U=V_s    supply voltage

%Tank Circuit Dynx Constants
L_s=2.47e-6;    %series inductance (with source)
R_s=5e-3;   %series resistance (with source)
L_c=100e-9; %capacitor leg inductance
C=95e-6;    %capacitance
R_c=2e-3;   %capacitor leg resistance
L_cab=0;    %cabling inductance
R_cab=0;    %cabling resistance

L_p=8.21e-6;    %parallel (to load) inductance/primary inductance
R_p=5e-3;   %parallel (to load) resistance/primary resistance
L=.9e-6;    %coil inductance/(transformed) plasma inductance (plasma)
R=10e-2;    %coil resistance/(transformed) plasma resistance (plasma)

% %Tank Circuit Dynx Constants (modified)
% L_s=0.634e-6; %series inductance (with source)
% R_s=2e-3; %series resistance (with source)
% L_c=50e-9;    %capacitor leg inductance
```

```
% C=191e-6; %capacitance
% R_c=2e-3; %capacitor leg resistance
% L_cab=0;  %cabling inductance
% R_cab=0;  %cabling resistance
% L_p=2.4e-6;   %parallel (to load) inductance/primary inductance
% R_p=3e-3; %parallel (to load) resistance/primary resistance
% L=.9e-6;  %coil inductance/(transformed) plasma inductance
% R=250e-3; %coil resistance/(transformed) plasma resistance

%Tank Circuit Dynx derived quantities
a=1/(L_s+L_c);
b=1/(L_cab+L_p+L_c);
eh=(R_s+R_c)*a;
f=R_c*a;
g=L_c*a;
h=R_c*b;
k=(R_cab+R_p+R_c)*b;
l=R_p*b;
m=L_c*b;
n=L_p*b;
p=R_p/(L+L_p);
q=(R+R_p)/(L+L_p);
v=L_p/(L+L_p);

%Tank Circuit Dynx SS (no flux)
A1=[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-a*(1-n*v)];
A2=[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
A3=[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b-m*a)];
A4=[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];

A=(1/(1-m*g-n*v)).*[A1;A2;A3;A4];
B=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0];

```

```
68 F=[A*state+B*U;U];
```

breaklines

```
1  %Matt Kraske
2  %HIT-SI lab, University of Washington - 2014
3  %Control system development Jan 23, 2014
4  %Function for HIT-SI Flux circuit time evolution with PID control
5
6  function F=dynamicsfuncspaoff(t,X)
7
8  state=X(1:4,1);
9  U=X(5,1);
10 %input breakdown:
11 %X(1)=I_1    loop 1 current
12 %X(2)=I_2    loop 2 current
13 %X(3)=I_3    loop 3 current
14 %X(4)=V_C   capacitor voltage
15 %U=V_s    supply voltage
16
17 %Tank Circuit Dynx Constants
18 L_s=2.47e-6;    %series inductance (with source)
19 R_s=5e1;    %series resistance (with source)was -3
20 L_c=100e-9; %capacitor leg inductance
21 C=95e-6;    %capacitance
22 R_c=2e-3;   %capacitor leg resistance
23 L_cab=0;    %cabling inductance
24 R_cab=0;    %cabling resistance
25
26 L_p=8.21e-6;    %parallel (to load) inductance/primary inductance
27 R_p=5e-3;   %parallel (to load) resistance/primary resistance
28 L=2.85e-6;  %coil inductance/(transformed) plasma inductance (plasma)
29 R=7e-2; %coil resistance/(transformed) plasma resistance (plasma)
30
31 % %Tank Circuit Dynx Constants (modified)
```

```
% L_s=0.634e-6; %series inductance (with source)
% R_s=2e-3; %series resistance (with source)
% L_c=50e-9;    %capacitor leg inductance
% C=191e-6; %capacitance
% R_c=2e-3; %capacitor leg resistance
% L_cab=0;  %cabling inductance
% R_cab=0;  %cabling resistance
% L_p=2.4e-6;   %parallel (to load) inductance/primary inductance
% R_p=3e-3; %parallel (to load) resistance/primary resistance
% L=.9e-6;  %coil inductance/(transformed) plasma inductance
% R=250e-3; %coil resistance/(transformed) plasma resistance

%Tank Circuit Dynx derived quantities
a=1/(L_s+L_c);
b=1/(L_cab+L_p+L_c);
eh=(R_s+R_c)*a;
f=R_c*a;
g=L_c*a;
h=R_c*b;
k=(R_cab+R_p+R_c)*b;
l=R_p*b;
m=L_c*b;
n=L_p*b;
p=R_p/(L+L_p);
q=(R+R_p)/(L+L_p);
v=L_p/(L+L_p);

%Tank Circuit Dynx SS (no flux)
A1=[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-a*(1-n*v)];
A2=[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
A3=[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b-m*a)];
A4=[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];

```

```
A=(1/(1-m*g-n*v)).*[A1;A2;A3;A4];
B=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0];

F=[A*state+B*U;U];
```

A.2 Estimation Script

breaklines

```
%Matt Kraske
%HIT-SI lab, University of Washington - 2014
%Control system development May 3, 2014
%script to estimate the plasa inductance for HIT-SI Flux circuit time evolution
%for use with plasma inductance estimation

close all; clear all; clc;

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Initialization of Parameters %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

%% Set up initial parameters --set as per "fbtank_129499_flux_x"

```

```
18 SCLK=1.25e8;         %Blackfin Clock frequency (125 MHz)
19 shot_length=2000;    %shot length in microseconds (us)
20 freq_SIHI=14500;     %tank resonant frequency (Hz)
21 T_period_cts=1/freq_SIHI/2*SCLK;     %blackfin ticks per half period
     was=4310;
22 n_pulses=shot_length*freq_SIHI*2/1000000;        %number of pulses in
     shot length was 117
23 if mod(n_pulses,2)==0
24     n_pulses=n_pulses+1;
25 end
26 n_adcs_per_bf_period=7; %number of blackfin sample events per SIHI
     period
27 dead_time=1;         %no clue
28 offset=2081;         %ADC sample offset (used for adjusting pos/neg
     offset --was 1026
29 vac_cycles=16;       %Number of ramping cycles (8 periods)
30
31
32 T_half_period=1/freq_SIHI/2*SCLK;
33
34 fp_gain=410;         %proportional gain for PID
35 fi_gain=62;          %integral gain for PID
36 fd_gain=0;           %derivative gain for PID
37 fp_factor=1000;      %scaling factor for some part of PID
38 dem_factor=200;      %scaling factor for some part of PID
39 min_width_counts=20;     %limit for coercing ramp demand
40
41 plasma_period=1/freq_SIHI;
42 half_plas_per=plasma_period/2;
43
44 scaling=180; %scaling factor for DCP_plasma
45
46 adc_delta=(T_half_period*2)/n_adcs_per_bf_period;
```

```
adc_half_delta=adc_delta/2;

%%%    Initialize loop variables %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
T_max_half_width=T_half_period-dead_time-min_width_counts;
min_width_counts=20; %1.6e-7;
T_half_width=1;
neg_error_sum=0;
pos_error_sum=0;
neg_error_prev=0;
pos_error_prev=0;
n_samples=n_adcs_per_bf_period;
sample_step=half_plas_per/n_samples;
sample_delta=half_plas_per/(n_samples);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Construct Jacobian %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

syms x1 x2 x3 x4 L1 L2 t U;


%Tank Circuit Dynx Constants
L_s=2.47e-6;    %series inductance (with source)
R_s=5e-3;   %series resistance (with source)
L_c=100e-9; %capacitor leg inductance
C=95e-6;    %capacitance
R_c=2e-3;   %capacitor leg resistance
```

```
74 L_cab=0;     %cabling inductance
75 R_cab=0;     %cabling resistance
76
77 L_p=8.21e-6;     %parallel (to load) inductance/primary inductance
78 R_p=5e-3;    %parallel (to load) resistance/primary resistance
79 L=L1+L2*t;  %coil inductance/(transformed) plasma inductance (plasma)
80 R=10e-2;     %coil resistance/(transformed) plasma resistance (plasma)
81
82 %Tank Circuit Dynx derived quantities
83 a=1/(L_s+L_c);
84 b=1/(L_cab+L_p+L_c);
85 eh=(R_s+R_c)*a;
86 f=R_c*a;
87 g=L_c*a;
88 h=R_c*b;
89 k=(R_cab+R_p+R_c)*b;
90 l=R_p*b;
91 m=L_c*b;
92 n=L_p*b;
93 p=R_p/(L+L_p);
94 q=(R+R_p)/(L+L_p);
95 v=L_p/(L+L_p);
96
97 %Tank Circuit Dynx SS (no flux)
98 A1=(1/(1-m*g-n*v)).*[g*h-eh*(1-n*v),f*(1-n*v)-g*(k-n*p),g*(l-n*q),g*b-
     a*(1-n*v)];
99 A2=(1/(1-m*g-n*v)).*[h-m*eh,m*f-k+n*p,l-n*q,b-m*a];
100 A3=(1/(1-m*g-n*v)).*[v*(h-m*eh),p*(1-m*g)+v*(m*f-k),v*l-q*(1-m*g),v*(b
     -m*a)];
101 A4=(1/(1-m*g-n*v)).*[(1/C)*(1-m*g-n*v),(-1/C)*(1-m*g-n*v),0,0];
102 A5=zeros(1,4);
103 A6=zeros(1,4);
104
```

```
105 A=[A1;A2;A3;A4;A5;A6];
106 B=(1/(1-m*g-n*v)).*[a*(1-n*v);m*a;v*m*a;0;0;0];
107
108 nonlin=A*[x1,x2,x3,x4]'+B*U;
109 Jac1=jacobian(nonlin,[x1,x2,x3,x4,L1,L2]);
110
111 thadynamics=@(x1,x2,x3,x4,L1,L2,t,U) nonlin;
112
113 nonlin2=A*[x1,x2,x3,x4]'+B*U;
114 nonlin2(5,1)=L1;
115 nonlin2(6,1)=L2;
116 Jac2=jacobian(nonlin2,[x1,x2,x3,x4,L1,L2]);
117
118 thadynamics2=@(x1,x2,x3,x4,L1,L2,t,U) nonlin2;
119
120 %%
121 %%%    Import actual flux waveform
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
122 realflux = importdata('fbtank_129499_flux_x.dat',' ',20);
123 realfluxindex=realflux.data(:,1);
124 realfluxvalue=realflux.data(:,2);
125 % realfluxtimebase=linspace(0,.004-(half_plas_per),(4000*freq_SIHI
      *2/1000000)*n_samples)'; %+(.25*plasma_period)
126 % %realfluxtimebase=2000/SCLK:adc_half_delta/SCLK:shot_length+2000/
      SCLK; %phase shifted
127 % %realfluxtimebase=0:adc_half_delta/SCLK:shot_length; %not phase
      shifted
128 % realfluxvalue=realfluxvalue(1:end-7,1);
129
130 realvolt = importdata('fbtank_129499_spav_x.dat');
131 % realfluxindex=realflux.data(:,1);
132 % realfluxvalue=realflux.data(:,2);
133 % realfluxtimebase=linspace(0,.004-(half_plas_per),(4000*freq_SIHI
```

```
*2/1000000)*n_samples)'; %+(.25*plasma_period)
% %realfluxtimebase=2000/SCLK:adc_half_delta/SCLK:shot_length+2000/SCLK; %phase shifted
% %realfluxtimebase=0:adc_half_delta/SCLK:shot_length; %not phase shifted
% realfluxvalue=realfluxvalue(1:end-7,1);
%
% plot(realvolt(:,2),realvolt(:,3))


% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Initialize EKF parameters %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
N=400; %length(realfluxindex);
Hx=[0,1,0,0,0,0];
K=zeros(6,N);

realvoltsamples=zeros(1,N);
x=zeros(6,N);
xold=zeros(6,N);
xfine=zeros(6,length(realvolt));
%x=zeros(6,1);
Upsilon = diag([1,.1,.1,.1,1,1]);
%N=n_adcs_per_bf_period*n_pulses;

x_apri = zeros(6,N);
x_apost = zeros(6,N);
P_apri = zeros(6,6,N);
```

```
P_apri(:,:,1) = eye(6);
P_apost = zeros(6,6,N);
Pcov = zeros(6,N);
Pcov(:,1) = diag(P_apost(:,:,1));
yekf = zeros(1,N);
Sekf=zeros(1,N);

x(:,1)=[0,0,0,0,1.85e-6,1]';

Qekf=diag([.1,.1,.1,.1,.1,.1]);
Rekf=0.01;

wekf=randn(6,N);     %state noise
vekf=randn(1,N);     %measurement noise

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%    Loop for Parameter Estimation %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%

for indmain=2:(N-1)
    sample_timer=(indmain-2)*sample_delta;

    %%%    Find realvolt value that corresponds to time step
```

```
      %%%%%%%%%%%%%
185   for ind_search=1:length(realvolt)
186       if realvolt(ind_search,2)>=sample_timer || ind_search==length(
            realvolt)
187           if ind_search==1;
188               sample_ind=ind_search;
189               break
190           elseif realvolt(ind_search,2)-sample_timer>realvolt(
              ind_search-1,2)-sample_timer
191               sample_ind=ind_search-1;
192           else
193               sample_ind=ind_search;
194           end
195           break
196       end
197   end
198   realvoltsamples(1,indmain-1)=realvolt(sample_ind,3);
199
200   %%%    Update Kalman gain
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
201   Sekf(1,indmain)=(Hx*P_apri(:,:,indmain-1)*Hx'+Rekf);
202   K(:,indmain)=P_apri(:,:,indmain-1)*Hx'/Sekf(1,indmain);
203
204   %%%    Update aposteriori state estimate
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%
205   x(:,indmain)=x(:,indmain-1)+K(:,indmain)*(realfluxvalue(indmain-1)
        -Hx*x(:,indmain-1));
206
207   %%%    Update aposteriori error covariance estimate
        %%%%%%%%%%%%%%%%
208   P_apost(:,:,indmain)=(eye(6)-K(:,indmain)*Hx)*P_apri(:,:,indmain
        -1);
209
```

```
210     %%%    Find input index that corresponds to end of evolution
          %%%%%%%
211     for ind_search2=1:length(realvolt)
212         if realvolt(ind_search2,2)>=sample_timer+sample_delta ||
              ind_search2==length(realvolt)
213             if ind_search2==1;
214                 end_ind=ind_search2;
215                 break
216             elseif realvolt(ind_search2,2)-sample_timer>realvolt(
                  ind_search2-1,2)-sample_timer
217                 end_ind=ind_search2-1;
218             else
219                 end_ind=ind_search2;
220             end
221             break
222         end
223     end
224
225     %%%    Coerce L values back to where they should be for Vac
          %%%%%%%%
226     if indmain<=112
227         x(5:6,1)=[.9e-6,1]';
228     end
229
230     %%%    Set evolution start and end points from determined indices
              %%
231     evolvestart=sample_ind;
232     evolveend=end_ind;
233     xfine(:,evolvestart)=x(:,indmain);
234
235     %%%    Update state Jacobian %(ind_cycle)*sample_delta
          %%%%%%%%%%%%%%
236     t=sample_timer;
```

```
237     x1=x(1,indmain);
238     x2=x(2,indmain);
239     x3=x(3,indmain);
240     x4=x(4,indmain);
241     L1=x(5,indmain);
242     L2=x(6,indmain);
243     U=realvolt(evolvestart,3);
244
245     if indmain<=112
246         Fx=eval(Jac2);
247         for ind_evolve=evolvestart:evolveend-1
248             xfine(:,ind_evolve+1)=xfine(:,ind_evolve)+(eval(
                  thadynamics2(x1,x2,x3,x4,L1,L2,t,realvolt(ind_evolve,3))
                  )).*(3.33e-7);% Update apriori estimate
249         end
250     elseif indmain<=112
251         Fx=eval(Jac2);
252         for ind_evolve=evolvestart:evolveend-1
253             xfine(:,ind_evolve+1)=xfine(:,ind_evolve)+(eval(
                  thadynamics2(x1,x2,x3,x4,L1,L2,t,realvolt(ind_evolve,3))
                  )).*(3.33e-7);% Update apriori estimate
254         end
255
256     else
257         for ind_evolve=evolvestart:evolveend-1
258             xfine(:,ind_evolve+1)=xfine(:,ind_evolve)+(eval(
                  thadynamics2(x1,x2,x3,x4,L1,L2,t,realvolt(ind_evolve,3))
                  )).*(3.33e-7);% Update apriori estimate
259         end
260     end
261     x(:,indmain+1)=xfine(:,evolveend);    % Update apriori estimate
262     xold(:,indmain+1)=xfine(:,evolveend);
263     P_apri(:,:,indmain)=Fx*P_apost(:,:,indmain)*Fx'+Upsilon*Qekf*
```

```
      Upsilon';   % Update aprioiri error covariance estimate
264   %P_apri(:,:,indmain)=P_apost(:,:,indmain)+(Fx*P_apost(:,:,indmain)
        +P_apost(:,:,indmain)*Fx'+Upsilon*Qekf*Upsilon')*sample_delta;
          % Update aprioiri error covariance estimate
265
266   Pcov(:,indmain+1)=diag(P_apost(:,:,indmain+1))';
267 end
268
269 figure(1)
270 plot(1:1:(N-1),x(1,1:end-1),'b',1:1:(N-1),x(2,1:end-1),'r',1:1:(N-1),x
    (3,1:end-1),'c',1:1:(N-1),x(4,1:end-1),'m')
271 legend('State 1','State 2','State 3','State 4')
272
273 figure(2)
274 plot(1:1:(N-1),x(5,1:end-1),'b',1:1:(N-1),x(6,1:end-1),'r')
275 legend('L1','L2')
276
277 figure(3)
278 plot(realfluxindex(1:400),realfluxvalue(1:400),1:1:(N-2),x(2,3:end)
    ,1:1:(N-3),xold(2,4:end),'xg')
279 legend('Apriori injector flux','estimated injector flux','evolution of
     estimate')
```

VITA

Matt Kraske is a graduate student in the controls emphasis at the University of Washington in Seattle Washington. Any inquiries into the research documented here can be directed to him at kraskm@uw.edu