Detection and Isolation of Propeller Icing and Electric Propulsion System Faults in Fixed-Wing UAVs: Master Thesis

Ole Max Haaland

July 12, 2021

Preface

Abstract

This thesis presents a fault detection and isolation (FDI) framework for detecting propeller icing, and other propulsion faults of unmanned aerial vehicles (UAVs). Such faults are among the main causes for incidents and loss of equipment.

A theoretical framework for the proposed FDI is covered extensively. A tuning methodology and an implementation guide are also covered in detail. The method has been tested extensively using a software–in–the–loop simulator. The simulation results have proven to be very successful and this motivates future testing on real data sets.

Relation to previous work

This is a disclaimer about what is new in this thesis and what has been borrowed or reused.

Project work

There will be some content-overlap between this thesis and the project work from the fall of 2020. The project work will be quoted in the relevant parts.

Many of the core FDI ideas were developed during the project work. The main contribution of the project work was an in-depth analysis of those ideas. The analysis was aimed at limitations and possibilities. The project left off with a non-functioning method, with many open ends. This thesis presents a refined solution.

Software documentation is also covered in this thesis. The software project was started during the project work. This was not documented in the project work. By then most of the code has been refactored many times over. Further, the main data structures were developed after the project work. The project work is therefore not quoted in the software section (7).

Previous work by others

This project builds on previous projects at NTNU. Most prominent is the use of the Ardupilot-Simulink simulator. Much of the simulator was developed by A. W. Wenz, K. Gryte, A. Winter and T. A. Johansen. The simulator is described in Section 5. My contribution to the simulator was the fault dynamics and the aircraft speed control.
Some of the core ideas were originally from A. W. Wenz. Wenz never published these ideas, but the ideas are present in the project work. Thus, Wenz is credited through the project work citations.

Contents

1 Introduction

This thesis presents a framework for fault detection and isolation (FDI). The framework is applied to the aerodynamic propulsion system of an unmanned aerial vehicle (UAV). The main contribution and novelty of this thesis is the use of aerodynamic propulsion performance models and estimation for propeller icing within an FDI framework. The problem addressed here is how to automatically detect icing on the propeller of a fixed-wing UAV, and how to isolate this fault, given that there are several other plausible faults in the electric propulsion system. Please note the semantic difference between *fault detection* and *fault isolation*. A fault is detected if the system correctly realizes that an error has occurred. The fault is isolated when the system knows which error has occurred.

The development of this algorithm is motivated by the increased utilization of UAVs and their integration into non-segregated airspace. This development comes with multiple key challenges and risks. It is widely recognized that faults in the propulsion system is one of the main risks that may lead to loss of aircraft and damage to third parties.
Of particular importance is the risk related to UAV propeller icing. In fact, icing can cause a propeller to loose up to 75% of its thrust after less than 2 minutes of operation [1–3]. Notably, the effects of icing are more critical on small UAVs. This is because of small air frame sizes and slow air-speeds [4]. Missions that require beyond visual line of sight (BLOS) operation are especially prone to encountering icing conditions. This is because the mission planner has no guarantee of good weather [5, 24].
The detection and diagnosis of several types of propulsion system faults in UAVs are well studied, e.g., [8–12, 19]. However, detection of UAV propeller icing is a comparably neglected topic.

The presented FDI framework is composed of both *parallel* and *sequential* algorithms. The system utilizes a bank of Kalman and Bayes filters in parallel. These filters preform the bulk of information processing in both the *detection algorithm* and the *isolation algorithm*. These algorithms execute sequentially. Specifically, a fault detection will start the isolation algorithm.
The main idea is to employ the propulsion system dynamics [17] to formulate different models for different faults – hence *multiple models*. Statistically, the different *models* are states in a Markov chain. Each state represents a different fault hypotheses. The different models are applied to separate Kalman filters.
The algorithms for fault detection and isolation build on comparing the Kalman filter outputs in a Bayesian framework. The framework combines the Kalman filter performances with the *a priori* transition probabilities in the Markov chain. The method uses measurements of the UAV's airspeed, and the propulsion motor's angular speed and electric current. It is assumed that the measurements are noisy but bias-free. This is motivated by the reliable nature of sensors for angular speed and electric current, as well as the existence of methods that can detect and estimate faults on air speed sensors [19–22].

This approach has some similarities to methods that have been used for detection of airfoil icing, which has been studied during the last years using various methods such as model-based estimation [16], multiple-model estimation [13,14], and statistical fault diagnosis methods [15].

1.1 Thesis structure

This thesis is organized as follows: Section 2 presents an overview of the relevant scientific literature. Research materials on propeller icing detection are quite sparse. Therefore, this section will be brief.

The proceeding sections develop the theoretic framework: Section 3 introduces the propulsion model of an UAV and how icing affects this system. The model is then developed into a multiple model, state space system.

Section 4 presents the main FDI algorithm. This includes the detection algorithm, the isolation algorithms and estimators.

A systematic tuning methodology is presented in Section 6.

Section 7 presents the software implementation FDI framework.

The proposed method is tested using simulation of a fixed-wing UAV. The simulation setup is described in Section 5 and results are given in Section 9. These results are discussed in Section 10 before the conclusions in Section 11. The thesis is summarized in Section 12.

2 Literature Review

This section will outline some of the excising scientific literature on icing and icing detection. The study will focus on propeller icing detection and propeller icing dynamics.

Icing detection is a well developed field. However, most solutions are based on dedicated icing sensors. These solution are typically not feasible for small UAVs, as they require small, light and cheap sensors [29]. Thus, this study will focus on approaches that are feasible to UAVs. Furthermore, a distinction must be made between icing detection on airfoils and propellers. This thesis presents a method for *propeller* ice detection. Contrarily, most existing methods are aimed at *airfoils*. Another difference is that this project aims at only measuring airspeed, motor current and the propeller rotational speed. Therefore, few methods are available for a direct comparison.

2.1 Propeller Icing Detection

Detection and identification have become big topics and various methods has been proposed. Some of the more popular ones are *multiple models* [32] and *observer-based* methods [33] both of which are relevant to this thesis. Icing detection in UAVs have also made use of *aerodynamic coefficient* estimators [16] and *statistical methods* [31] for diagnosis. More exotic methods, such as an *unknown input observer*, described by an uncertainty model [34] works in simulations. However, most applications have not been applied to real world settings.

There are very few relevant methods for propeller icing and most of the cited studies focus on airfoil icing detection. For example, Wenz uses an *aerodynamic coefficient* estimator to monitor the icing effect on the lift [16]. Similarly, Ding [31] uses a Neyman Pearson statistical decision approach to detect airfoil icing. To achieve this, a dynamic model for icing accretion was used. However, the given dynamic model [25] is only applicable to airfoils and not to propellers. Thus, the findings cannot be applied directly to a propeller. However, the method does serve as an interesting starting point.

2.2 Icing Dynamics: Accretion and Shedding

The icing dynamics describe how the levels of ice change with time. In the literature, the dynamics are usually referred as three different phenomena - accretion, shedding and melting. This study will focus on accretion and shedding.

A dynamic icing model can serve a process model for system observers. An accurate process model offer clear advantages for detection purposes. For example, by increasing observability and the convergence rate a Kalman filter. Many of scientific findings are promising. Furthermore, I expect that most impactful, casual variables have been identified. However, an accurate dynamical propeller icing model has not been found. This limits us to use casual variables to determine statistical priors.

There is a knowledge disparity between accretion on propellers and airfoils on small aircrafts. For airfoils, a continuous icing accretion model [25] has been available for 20 years. The model introduces a time dependent ice accretion rate, $\dot{\eta}_{ice}$. The model relates $\dot{\eta}_{ice}$ to atmospheric conditions and η_{ice} itself.

In comparison, recent work on propeller ice accretion [26, 27] is limited to quantitative descriptions. Thus, accretion on airfoils is better understood than that on propellers. The disparity is mainly due to propeller accretion being hard to model. Propellers and airfoils share all relevant environmental factors such as temperature, humidity, angle of attack, etc. However, the propeller rotation introduces centrifugal forces [35] and modulated aerodynamic shear forces [26]. These forces make the icing dynamics substantially more complex.

Icing accretion on propellers is in a quantitative stage [26, 27]. The quantitative experiments are limited to measuring different icing measures at fixed intervals. Quantitative descriptions are then given the recorded data.

The casual relationship between environmental factors and icing such as temperature and humidity are well understood [29]. Further, temperature and humidity can also affect the formation of the ice. For example, cold air tends to form rime ice [26] - an ice layer that adopts the shape of the propeller rime. *Glaze* ice occurs in humid air and around the freezing point. This glaze tends to adopt complex and irregular shapes. Glaze ice has been shown to cause the largest changes in the aerodynamic surface properties [30]. We also know that propeller materials, such as wetness, can drastically impact the icing [27].

The centrifugal force also plays a part in the ice accretion as a function of spatial location. Ice accretion grows monotonically with the distance from the propeller center [26]. The loss in propeller efficiency as a function of accretion is also well established [26].

The ice accretion can be reverted by ice shedding. Ice shedding is a process where fragments of ice fall off the propeller [28]. Thus, shedding is highly relevant for a dynamic icing model. Furthermore, shedding detection is a central part of our problem space. Ideally, shedding will restore the propeller efficiency. However, uncontrolled shedding can cause unwanted events. For example, fragments can hit or fall into downstream aircraft components [28, 29]. Shedding times are related to the accretion time, temperature and de-icing heat flux [29]. Once again, only quantitative research is available. Thus, findings on the accretion and shedding can only serve as statistical priors in our model.

3 Modelling

This section starts with introducing the electric propulsion model for a fixed-wing UAV (Section 3.1). This is followed by a discussion of *icing* in Section 3.2. Subsection 3.2 formalizes *icing* and its effect on the propulsion model. Subsection 3.3 then introduces a set of icing models for feasible state estimation. Then, a state space model is used to formulate different hypotheses about the propulsion faults and propeller icing in Section 3.4. The section ends with a parameter estimation model that is formulated with a linear time-variant (LTV) state-space representation in Section 3.5.

3.1 Propulsion Model

The propulsion model of the UAV is based on modelling of the electrical, mechanical and aerodynamic subsystems. We only state the resulting equations here, and refer to [17] for a more comprehensive description of the model:

$$\Theta\dot{\omega} = k_e(I_e - I_0) - c_v\omega - Q_a \tag{1}$$

where in the nominal case we have

$$Q_a = \rho\frac{\omega^2}{4\pi^2}D^5 C_Q(J) \tag{2}$$

$$C_Q(J) = C_{Q,0} + C_{Q,1}J + C_{Q,2}J^2 \tag{3}$$

$$J = 2\pi\frac{V_a}{D\omega} \tag{4}$$

Equation (1) gives the propeller torque balance, where ω is the angular speed, Θ is the moment of inertia of the shaft and rotor including the propeller, I_e is the motor electric current, I_0 is the zero-load current, c_v is the viscous friction coefficient, and k_e is the motor constant. Equation (2) describes Q_a which is the aerodynamic torque created by propeller drag, where D is the propeller diameter and ρ is the air density. The thrust coefficient, $C_Q(J)$ is given by a second order polynomial. This polynomial is a function of the advance ratio J, where V_a is the airspeed. The use of the propulsion model was originally introduced in the project work [36].

3.2 Icing

This section introduces the concept of icing, the icing dynamics, and its effect on the propulsion model. A clear understanding of this is essential for designing a robust fault detection system. However, the nature of the icing dynamics is an open research question (as covered in Section 2). Thus, the following will also highlight the relevant gaps in our understanding. Subsections 3.3 and 3.4 cover how these gaps are compensated for.

Icing accretion refers to the formation of ice on a given surface. There are multiple ways to quantify this phenomena. For example, the *degree of icing*, ξ,

could refer to the mass of the ice $m_\xi \in \mathbb{R}^+$ or some icing coefficient $c_\xi \in [0, 1]$. The literature typically uses *icing accretion rate* [25], icing severity factor [34] or the icing edge width [26]. However, choosing a definition is not necessary here. Instead, the reader must assume that the *degree of icing*, ξ, refers to some valid quantity. Further, the *Icing dynamics* is then the time varying change of this quantity, $\dot{\xi} = f(\cdot)$.

For the present case, the surface of interest is that of a propeller. The icing phenomena can be observed through its negative effect on propeller efficiency. The energy reduction occurs because icing increases the aerodynamic torque 2. The negative effect should be clear from equation 1. An increase in Q_a will decelerate the propeller.

The aerodynamic torque increases due to an increase in the thrust coefficient C_Q. C_Q increases due to a time varying change in the coefficients $C_{Q,0}$, $C_{Q,1}$ and $C_{Q,2}$ from equation 3.

In this text, the term *icing* will casually refer to an icing induced change in C_Q. The resulting dynamics is referred to as the *Icing fault dynamics*. The icing fault dynamics can then be found as the time derivative of equation 3

$$\dot{C_Q}(\xi, J) = \dot{C}_{Q,0}(\xi) + \dot{C}_{Q,1}(\xi)J + \dot{C}_{Q,2}(\xi)J^2 + \frac{dC_Q}{dJ}\dot{J} \tag{5}$$

$$= \dot{\xi}(\frac{dC_{Q,0}}{d\xi} + \frac{dC_{Q,1}}{d\xi}J + \frac{dC_{Q,2}}{d\xi}J^2) + (C_{Q,1} + 2C_{Q,2}J)\dot{J} \tag{6}$$

where the chain rule is used.

There are many ways in which equation 6 could be developed further. A state space could include only C_Q, or $C_{Q,0}$, $C_{Q,1}$ and $C_{Q,2}$. Further, the icing dergree ξ could be modeled as a state, process noise or as an input. However, equation 6 is incomplete due to gaps in scientific knowledge. Specifically, analytical expressions of both $\dot{\xi}$ and $\frac{dC_{Q,i}}{dt}$ are missing. Furthermore, the equation becomes exceedingly complicated the time derivative of the advance ratio $\dot{J}$ is considered.

This prohibits the use of equation 6 as process model for a state observer. This imposes drastic simplifications for the observer design. To tackle this, subsection 3.4 introduces a random walk process model.

The number of independent variables on the RHS of equation 6 is also unknown. Specifically, how many parameters are needed to describe the change in the terms $\frac{dC_{Q,0}}{d\xi}$, $\frac{dC_{Q,1}}{d\xi}$ and $\frac{dC_{Q,2}}{d\xi}$ from equation 6? For example, can one parameter describe the change of all their terms? The answer has important implications for the design of the state observers. This will determine the dimensionality of the estimated state vector.

Investigation with different fault detection and identification methods has shown that it is difficult, if not impossible, to get reliable online estimates of the three coefficients of the polynomial $C_Q(J)$ when they are estimated as independent parameters [36]. The reason is that the natural variations in J are relatively small, and even with extensive airspeed changes and maneuvers designed to increase the observability of these parameters, it turns out to be difficult to get

sufficiently accurate estimates to reliably detect icing and isolate it from faults. The implication is that a state observer must use a reduced model to achieve observability. This is covered in Section 3.3.

3.3 Reduced Model of Aerodynamic Torque

This subsection will describe several methods for approximating the dynamics of the thrust coefficient C_Q, as given in equation 6. All methods are based on reducing the 2^{nd} order polynomial C_Q from three to one or two parameters. The performance of 1-parameter vs. 2-parameters methods can not be fully determined without working with real data. Nevertheless, a comparison of methods can be found in Section 5.3. The model know referred to as the *One-parameter* model is used in the final simulations.

Note that understanding the FDI framework can be understood without reading this subsection. The framework functions for all the presented methods, in both the scalar and the multivariable cases. The theoretical descriptions in the reminder of this section and Section 4 are therefore invariant to the choice of model. The variable $\boldsymbol{\theta}$ and state $\boldsymbol{x}^{(1)}$ can refer to all of the models described here.

3.3.1 One-parameter Scaling Model

This model assumes that icing has a linear scaling effect on $C_Q(J)$. The scaling is determined by the scalar parameterizer θ_1 with nominal value $\theta_1^* = 1$. The aerodynamic torque in equation 2 can then be described as

$$Q_a = Q_a^* \cdot \theta_1 \tag{7}$$

where $Q_a^* = \rho \frac{\omega^2}{4\pi^2} D^5 C_Q(J)$ is defined by the nominal (ice-free) values of the coefficients in $C_Q(J)$. Icing is then characterized by $\theta_1 > 1$. This model implies that the icing fault dynamics of all coefficients $C_{Q,i}$ of C_Q are the same:

$$\frac{dC_{Q,i}}{d\xi} = \frac{dC_{Q,j}}{d\xi}, \quad i \neq j \tag{8}$$

Preliminary findings from icing wind tunnel tests suggest that the scalar multiplicative icing model, as given in equation 7, could be well suited to describe the icing dynamics when the changes in airspeed are small.

3.3.2 Sub-space Model

This method is aimed at finding a subspace of that approximate the dynamics of C_Q in 1 or 2 dimensions. The dynamics are then parameterized in the resulting subspace. Mathematically, we have

$$C_Q(J) \approx \boldsymbol{\theta}^T \boldsymbol{Z} \begin{bmatrix} 1 & J & J^2 \end{bmatrix}^T \tag{9}$$

where $\boldsymbol{Z}$ is a 2x3 projection matrix and $\boldsymbol{\theta} = [\theta_1 \quad \theta_2]^T$ is a 2-dimensional parameter vector.

We aim at finding a projection $\boldsymbol{Z}$ that maximize some objective. Specifically, our aim is to maximize the observability of $\boldsymbol{\Theta}$.
A direct measure of observability can be found using the observability Gramian $\boldsymbol{W}^o$. Further, $\boldsymbol{W}^o$ yields such a measure for all directions in the parameter space. This can be used to identify the desired sub-space, similar to a principal component analysis (PCA). The Gramian is approximated using an empirical Gramian which is given by

$$\boldsymbol{W}^o_{k,N} = \boldsymbol{\mathcal{C}}^T_{k,N}\boldsymbol{\mathcal{C}}_{k,N} \tag{10}$$

where $\boldsymbol{\mathcal{C}}_{k,N}$ is given by

$$\boldsymbol{\mathcal{C}}_{k,N} = \begin{bmatrix} \boldsymbol{H}_k \\ \vdots \\ \boldsymbol{H}_{k-N+1} \end{bmatrix} \tag{11}$$

where the integer $N \geq 3$ defines a data window. The empirical Gramian is a symmetric positive definite matrix that can be expressed using an eigenvalue decomposition

$$\boldsymbol{W}^o_{k,N} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{12}$$

where $\boldsymbol{V}$ has columns that contain the eigenvectors and $\boldsymbol{\Sigma}$ is a diagonal matrix with eigenvalues on the diagonal. This model reduction is done as a part of the design of the fault detection and identification system. It requires an extensive dataset from the relevant UAV executing typical maneuvers under realistic conditions, leading to a large data window N. This is the reason why we have excluded the indices k and N from the matrices $\boldsymbol{V}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{Z}$.

For a given $n \in \{1, 2\}$, the most observable n-dimensional subspace is defined by the $n \times 3$ projection matrix $\boldsymbol{Z}$, having rows that contain the eigenvectors corresponding to the n largest eigenvalues. The rows of $\boldsymbol{Z}$ determine a linear combination of the original 3 polynomial terms that yield the highest observability for the given data window. This leads to the projection matrix $\boldsymbol{Z}$ that is kept constant during the online use of the system.

3.3.3 Linear Approximation

The method aims at reducing the parameter space through linearization around an operating point $\bar{J}$. The second degree polynomial in equation 3 is then approximated by a linear function as in [17]

$$\begin{aligned} C_Q(J) &= C_{Q,0} + C_{Q,1}J + C_{Q,2}J^2 \\ &\approx \bar{C}_{Q,0} + \bar{C}_{Q,1}J \end{aligned} \tag{13}$$

The model has the same form as equation (3), except that its dimension is one less since the quadratic term is neglected. The equations must be modified accordingly.

Preliminary results suggest that this method works poorly. It has therefore not been tested further and no results are presented in this thesis.

3.4 State Space Representation for Estimation

The parameters are assumed to evolve according to a random walk process. Thus, any change from time index k to $k+1$ is only attributable to process noise $\boldsymbol{v}_k$. This results in a process model given by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{v}_k \tag{14}$$

The measurement model of the system is derived from the torque balance in the system model given by equation 1. The airspeed V_a, angular velocity ω and the motor current I_e are assumed to be measured. Substituting equations 2, 3, 4 and the parameter vector $\boldsymbol{x}_k$ into equation 1 yields

$$y_k = [Q_a \quad \omega \quad k_e]\boldsymbol{x}_k + w_k \tag{15}$$

with $y_k = -k_e I_e + \Theta\dot{\omega}$. In cruise mode the UAV speed controllers will maintain an almost constant propeller speed. This motivates the simplifying assumption that

$$\dot{\omega} = 0 \tag{16}$$

which leads to $y_k = -k_e I_e$. This assumption will be used in the fault detection algorithms in this paper. The measurement model 15 forms the basis for the measurement matrices $\boldsymbol{C}_k^{(1)} = Q_a$, $\boldsymbol{C}_k^{(2)} = \omega$ and $\boldsymbol{C}_k^{(3)} = k_e$. This leads to

$$y_k = [\boldsymbol{C}_k^{(1)} \quad \boldsymbol{C}_k^{(2)} \quad \boldsymbol{C}_k^{(3)}] \begin{bmatrix} \boldsymbol{x}_k^{(1)} \\ \boldsymbol{x}_k^{(2)} \\ \boldsymbol{x}_k^{(3)} \end{bmatrix} + w_k \tag{17}$$

$$\boldsymbol{v}_k \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}), \quad w_k \sim \mathcal{N}(0, R) \tag{18}$$

Notice that y_k and $\boldsymbol{C}_k$ are *time dependant* functions of ω, V_a and I_e. This relationship is implicitly assumed throughout the paper. The variables ω, V_a and I_e will often be refereed to as a the measurement $\boldsymbol{z_k} = [\omega, V_a, I_e]$.

3.5 System Faults and Degradation Due to Icing

We will now concretize the meaning of the fault concept. We shall then formalize *the fault states* of the system.

System faults and degradation occur whenever any of the parameters of the system deviate from its nominal value. To represent this, the nominal parameter vector $\boldsymbol{x}^*$ is introduced:

$$\boldsymbol{x}^* = [\boldsymbol{\theta}^* \quad c_v^* \quad I_o^*]^T$$

The nominal vector $\boldsymbol{x}^*$ gives the parameter values of a fault-free and non-degraded system, which is related to the actual parameter vector $\boldsymbol{x}$ and the

deviation $\boldsymbol{\epsilon}$ through the equation $\boldsymbol{x} = \boldsymbol{x}^* + \boldsymbol{\epsilon}$. Note that these vectors are partitioned into:

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}^{(1)} \\ \boldsymbol{x}^{(2)} \\ \boldsymbol{x}^{(3)} \end{bmatrix} \tag{19}$$

where $\boldsymbol{x}^{(1)}$ contains the first element, $\boldsymbol{x}^{(2)}$ contains the second- and $\boldsymbol{x}^{(3)}$ contains the third element. The same notation is also used for $\boldsymbol{\epsilon}$ and similar vectors.

A fault in the propulsion system is present whenever any element in $\boldsymbol{\epsilon}$ is sufficiently different from zero. However, a simultaneous occurrence of multiple faults would be exceedingly rare. Thus, it is assumed that only one error can occur at a time. The propulsion system has 4 possible states related to faults and degradation. The following will be referred to as the *fault states* m:

0. No fault: $\boldsymbol{x} = \boldsymbol{x}^*$
1. Propeller icing: $\boldsymbol{x}^{(1)} \neq \boldsymbol{x}^{(1)*}$
2. Change in viscose friction: $\boldsymbol{x}^{(2)} \neq \boldsymbol{x}^{(2)*}$
3. Change in static friction: $\boldsymbol{x}^{(3)} \neq \boldsymbol{x}^{(3)*}$

Note that we will also refer to the much simpler binary state: **healthy** vs. **faulty**. Both states are modeled as Markov processes. We refer to the two Markov processes as the

$$\text{Health Model: health} \in \{\text{Healthy, Faulty}\}$$
$$\text{Fault Model: fault} \in \{0, 1, 2, 3\}.$$

Thus, the *health state* refers to the state of the *Health model*. Similarly, the *fault state* refers to the *Fault model*. The Markov chains are illustrated in Figure 1.

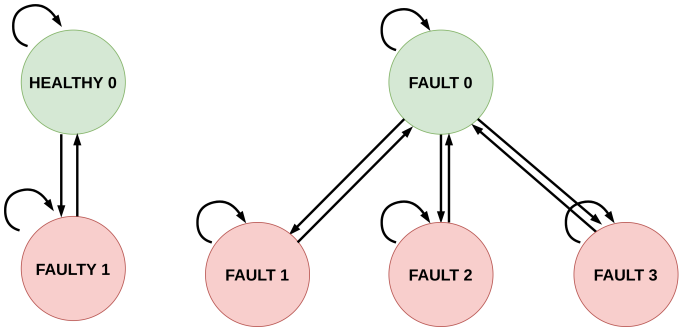

Figure 1: Left: The Markov chain of the Health state. Right: The Markov chain of the Fault state.

Both models have associated transitions probabilities. These are defined by a transition matrix

$$\Pi^{i}_{k,k_\Delta} = \begin{bmatrix} \pi_{1,1} & \dots & \pi_{1,n} \\ \vdots & \dots & \vdots \\ \pi_{n,1} & \dots & \pi_{n,n} \end{bmatrix} \in \mathcal{R}^{n \times n} \tag{20}$$

where $i \in \{f, h\}$ refers to the given model, and n is the number of states. The *Health* state has $n = 2$ and the *Fault* state has $n = 4$. $\pi_{i,j} = p(m_k = j | m_{k_\Delta} = i)$ gives the transition probability from state i to j. The first row therefore gives the transition probability from a fault free state to any other state.

4 FAULT DETECTION AND ISOLATION

We present a framework for sequentially *detecting* faults and icing, and then correctly *isolating* the fault state. The framework will therefore be presented as modules. We start by an overview of the main algorithm and the formulation of multiple relevant hypotheses. The key elements of the algorithm are then described in more detail: *the Kalman filter* and *the Bayes filter*. Then, the detection and isolation algorithms are presented.

The FDI framework presented here can be generalized and applied to other FDI problems. This allows the adoption of this algorithm to a wide range of UAV electric propulsion systems. However, some design decisions in this FDI algorithm are based on specific model assumptions.

4.1 Main Idea

This section describes the system transitions between the detection step and the isolation step. This entails giving a descriptive overview of the relevant signals and how they propagate through the system.

The detection step is concerned with the Markov process, referred to as the *Health* model. It aims to detect transitions from *healthy* to *faulty*. It therefore provides a binary hypothesis, *transition to fault* or *no fault*. This is formalized with the detection hypothesis H_D:

$$H_D \in \{\text{true, false}\} \tag{21}$$

The isolation step is concerned with the *Fault* state. It attempts to determine which transition is the most likely to have occurred. The hypothesis space is therefore

$$H_I \in \{0, 1, 2, 3\} \tag{22}$$

where $H_I = 0$ is no fault, $H_I = 1$ is fault 1 (icing), $H_I = 2$ is fault 2, and $H_I = 3$ is fault 3.

The structure of the algorithm is shown in Figure 2. It is assumed that the UAV is initially in a fault free state. The detection algorithm is initialized with the nominal state $\boldsymbol{x}^{(*)}$ and processes a stream of measurements $\boldsymbol{z_k}$. The detection algorithm executes until a fault is detected. This can be seen in the feedback loop shown in Figure 2. If a fault is detected ($H_D = \text{false}$), then the algorithm will start the identification algorithm and send a command for a small change in airspeed to the autopilot in order to enhance observability through excitation.

Inputs to the isolation algorithm are the measurement $\boldsymbol{z_k}$ and the nominal state $\boldsymbol{x}^{(*)}$. The isolation algorithm will after convergence output an hypothesis H_I, where $H_I = 0$ implies that the detection algorithm had a Type I error (false detection), whereas $H_I = i > 0$ implies that the system is in fault state i.

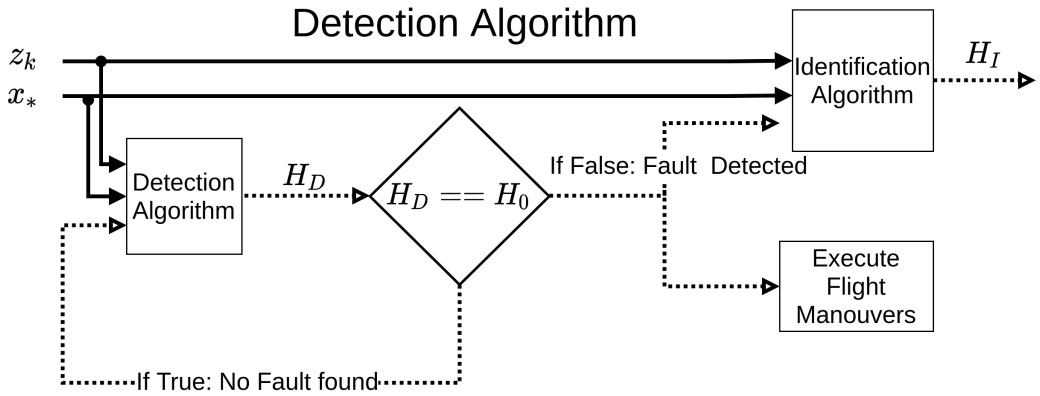

Figure 2: The data flow of the FDI algorithm. It can be seen that the detection algorithm will launch the isolation algorithm if a fault is found. Dashed lines represent a binary/integer signal. Solid lines represent continuous values.

4.2 Estimation

The Kalman filter and the Bayes Filter will now be introduced [18]. These will be used by both the detection and the isolation steps. We also formalize the concept of a static hypothesis models. The use of both the Kalman filter and the Bayes filter was suggested in the project work [36]. However, this section offer multiple refinements to the original ideas.

4.2.1 Kalman Filter

For each of the fault modes a Kalman filter is formulated. Since it is assumed that only one fault can occur at a time, the process noise is only affecting one of the sub-states $\boldsymbol{x^{(i)}}$, while the other states can be treated as constant. The state space model can therefore be rewritten as

$$\boldsymbol{x}_{k+1}^{(i)} = \boldsymbol{x}_k^{(i)} + \boldsymbol{v}_k^{(i)} \tag{23}$$

$$y_k^{(i)} = \boldsymbol{C}_k^{(i)}\boldsymbol{x}^{(i)} + [\boldsymbol{C}_k^{(j)} \quad \boldsymbol{C}_k^{(\ell)}] \begin{bmatrix} \boldsymbol{x}^{(j)*} \\ \boldsymbol{x}^{(\ell)*} \end{bmatrix} + w_k \tag{24}$$

where j and ℓ denote the two remaining modes other than i. The filters will have the same form for each mode $i \in \{1, 2, 3\}$. We refer to the Kalman filter of model i as $KF^{(i)}$. The prediction step is given by:

$$\hat{\boldsymbol{x}}_{k|k-1}^{(i)} = \hat{\boldsymbol{x}}_{k-1}^{(i)}, \qquad \in \mathcal{R}^{\delta_i} \tag{25}$$

$$\boldsymbol{P}_{k|k-1}^{(i)} = \boldsymbol{P}_{k-1}^{(i)} + \boldsymbol{Q}^{(i)}, \qquad \in \mathcal{R}^{\delta_i \times \delta_i} \tag{26}$$

$$\hat{y}_{k|k-1}^{(i)} = \boldsymbol{C}_k^{(i)}\hat{\boldsymbol{x}}_{k|k-1}^{(i)}, \qquad \in \mathcal{R}^{1} \tag{27}$$

The update step is given by

$$\nu_k^{(i)} = y_k - \hat{y}_{k|k-1}^{(i)}, \qquad \in\mathcal{R}^1 \tag{28}$$

$$S_k^{(i)} = \boldsymbol{C}_k^{(i)} \boldsymbol{P}_{k|k-1}^{(i)} \boldsymbol{C}_k^{(i)T} + R, \qquad \in\mathcal{R}^{1\times 1} \tag{29}$$

$$\boldsymbol{W}_k^{(i)} = \boldsymbol{P}_{k|k-1}^{(i)} \boldsymbol{C}_k^{(i)T} (S_k^{(i)})^{-1}, \qquad \in\mathcal{R}^{\delta_i\times\delta_i} \tag{30}$$

$$\hat{\boldsymbol{x}}_k^{(i)} = \hat{\boldsymbol{x}}_{k|k-1}^{(i)} + \boldsymbol{W}_k^{(i)} \nu_k^{(i)}, \qquad \in\mathcal{R}^{\delta_i} \tag{31}$$

$$\boldsymbol{P}_k^{(i)} = (\boldsymbol{I} - \boldsymbol{W}_k^{(i)} \boldsymbol{C}_k^{(i)}) \boldsymbol{P}_{k|k-1}^{(i)}, \qquad \in\mathcal{R}^{\delta_i\times\delta_i} \tag{32}$$

where $\delta_i = \dim(\boldsymbol{x}^{(i)})$ is the dimensionality of the state $\boldsymbol{x}^{(i)}$. Thus, the dimensions of the equations 25 and 26 depend on the mode i. This is illustrated in Figure 3, which shows 3 filters running in parallel.

Also, note that the covariance of the measurement noise R in the equation 29 is the same for all modes. This follows from the fact that all filters depend on the same physical measurements.

4.2.2 Static Hypothesis Models

We will introduce the so-called *Static Model Hypotheses*, which are models that assume that the model parameters remain fixed. For the fault free case, this gives

$$\begin{aligned} Y_k^{(0)} &= y_k - [\boldsymbol{C}_k^{(1)*} \quad \boldsymbol{C}_k^{(2)*} \quad \boldsymbol{C}_k^{(3)*}] \begin{bmatrix} \boldsymbol{x}_k^{(1)*} \\ \boldsymbol{x}_k^{(2)*} \\ \boldsymbol{x}_k^{(3)*} \end{bmatrix} + w_k \\ &= y_k - \boldsymbol{C}_k^* \boldsymbol{x}_k^* + w_k \end{aligned} \tag{33}$$

Note that we will also make use of static models for fault state $i \in \{1,\ 2,\ 3\}$. In this case it is assumed that some estimate $\hat{\boldsymbol{x}}_{k_s}^{(i)}$ was sampled at time $k_s \leq k$. This gives rise to the static measurement model

$$Y_k^{(i)} = \boldsymbol{C}_k^{(i)} \hat{\boldsymbol{x}}_{k_s}^{(i)} + [\boldsymbol{C}_k^{(j)} \quad \boldsymbol{C}_k^{(\ell)}] \begin{bmatrix} x_k^{(j)*} \\ x_k^{(\ell)*} \end{bmatrix} + w_k \tag{34}$$

The static measurement models directly output the innovation $\nu_k^{(i)}$. The covariance $S_k^{(i)}$ of $\nu_k^{(i)}$ is given by the measurement noise:

$$\nu_k^{(i)} = Y_k^{(i)} \tag{35}$$

$$S_k^{(i)} = R \tag{36}$$

R will be the same for all static filters. In the reminder, the context (i.e., the block diagrams in Figures 3 and 4) should make it intelligible when $\nu_k^{(i)}$ and $S_k^{(i)}$ are taken from the static hypothesis model and when they are taken from a Kalman filter.

4.2.3 Bayes Filter

The Bayes filter allows us to directly compare the performance of various hypotheses $\mathcal{H}_i$ corresponding to modes $i \in \{0, 1, 2, 3\}$. The filter follows directly from Bayes Theorem [18]:

$$p(\mathcal{H}_i|\boldsymbol{z}_{0:k}) = \frac{\mathcal{N}(\nu_k^{(i)}, 0, S_k^{(i)})p(\mathcal{H}_i|\boldsymbol{z}_{0:k-1}^{(i)})}{\sum_{j=0}^{M}\mathcal{N}(\nu_k^{(j)}, 0, S_k^{(j)})p(\mathcal{H}_j|\boldsymbol{z}_{0:k-1}^{(j)})} \tag{37}$$

where $\mathcal{N}(\nu_k^{(i)}, 0, S_k^{(i)})$ is the (Gaussian) likelihood of the zero-mean innovation $\nu_k^{(i)}$ given covariance $S_k^{(i)}$. The Bayes filter can compare a set of filters $Y^{(i)}, i \in \{0, 1, 2, 3\}$ and $KF^{(j)}, i \in \{1, 2, 3\}$ based on the likelihood:

$$\ell^{(i)} = \mathcal{N}(\nu^{(i)}, 0, S) \tag{38}$$

where $\nu^{(i)}$ varies with the filters. This thesis will be limited to using $S_k^{(i)} = R$ for $i \in \{0, 1, 2, 3\}$, where R is given by equation 36. Further, R will be the same for every model. This goes for both the Kalman and Bayes filters.

The innovation covariance S determines the Bayes filter sensitivity. To see this, consider the zero-mean Gaussian probability density functions (PDF) defined by S. Thus, for two (different) innovations, $\nu^{(i)} > \nu^{(j)}$ would imply $\ell(\nu^{(i)}) < \ell(\nu^{(j)})$. The smaller the $S_k^{(0)}$ is, the more narrow the distribution will be. This will increase the probability differences between the innovations. For example, the Gaussian zero mean likelihoods $\ell_n(\cdot)$ and $\ell_m(\cdot)$ have variances, S_n and S_m. $S_n < S_m$ will then imply that $\ell_n(\nu^{(i)}) - \ell_n(\nu^{(j)}) > \ell_m(\nu^{(i)}) - \ell_m(\nu^{(j)})$. Thus, smaller values of S make the algorithm *sensitive* to differences between the innovations.

The Bayes filter is recursive and initialized according to the prior $p(\mathcal{H}_i|\boldsymbol{z}_0^{(i)}) = p(\mathcal{H}_i)$. We always assume that the system starts in the fault free case. The prior is therefore given by the first row of the Markov matrix. Keep in mind that we are operating with two different Markov models. The *Health* model will initialize according to Π^h, while the *Fault* model will use Π^f.

The concept of a detection/isolation **window** L can be introduced at this point. The window L determines the length of time (or number of samples) the Bayes filter should process before returning a hypothesis. For example, $L = 10$ seconds would mean that the Bayes filter would process the data from the last 10 seconds. The mode i with the highest probability would then be returned as the hypotheses. This paper will make use of both a *detection-* and an *isolation windows*. These are the windows to be used by the respective algorithms.

We conclude our discussion by noting how this filter is different from the Magill filter [23]. This implementation uses that $S_k^{(i)} = R$. This yields a static measurement covariance. Contrarily, the Magill filter receives $S_k^{(i)}$ directly from Kalman filter $\boldsymbol{KF}^{(i)}$, as given in equation 29. Our reasoning is that using equation 29 for calculating $S_k^{(i)}$ would make filters probability differ substantially. This is because the process noise $\boldsymbol{Q}^{(i)}$ will directly affect $S_k^{(i)}$. This can be seen

in equations 26 and 29. The effect is prominent since the process noise will differ by orders of magnitude between models. Thus, different values of $S_k^{(i)}$ are likely to introduce big variations in $\ell^{(i)}$. However, the magnitude of the process noise of one filter should not make it more (or less) likely than other filters.

4.3 Fault Detection

The fault detection algorithm aims at detecting faults as defined in equation 21. The basic idea is to use a Bayes filter to compare the output of a Kalman filter, $KF^{(i)}$, against the static zero hypothesis model, $Y_k^{(0)}$. A fault is detected if some Kalman filter, $KF^{(i)}$ $i \in \{1, 2, 3\}$ outperforms the fault-free static model $Y_k^{(0)}$ during the interval $\{k - L, ..., k - 1, k\}$, where the integer L is referred to as the detection window.

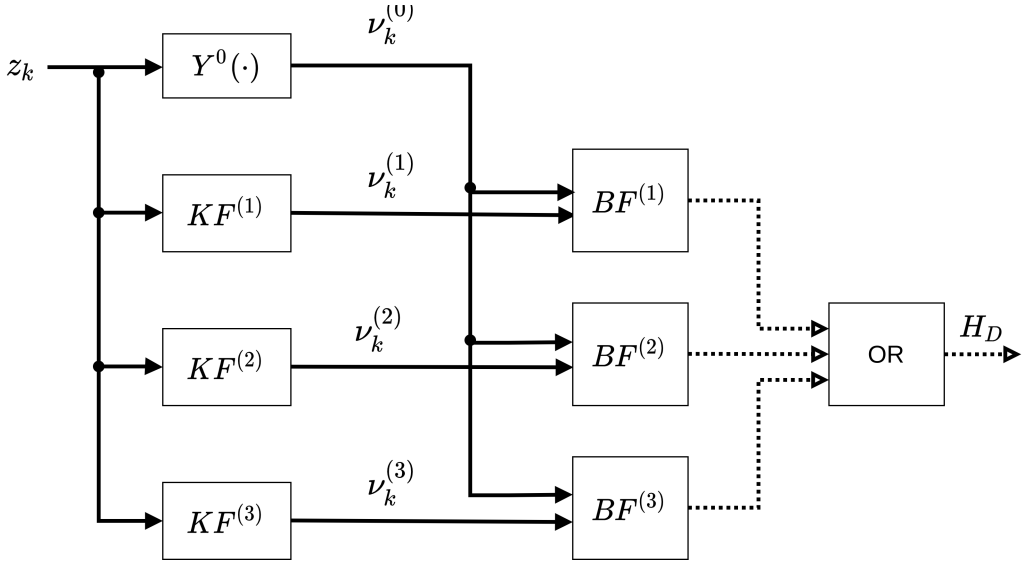

Figure 3: Block diagram with detection algorithm.

A block diagram of the algorithm is given in Figure 3. It can be seen that the measurement $\boldsymbol{z}_k$ is propagated through three computational layers before a hypothesis H_D formed. The layers are as follows:

1. The Kalman filters and static model use the measurement $\boldsymbol{z}_k$ as input and outputs:

$$(\nu_k^{(i)}) \leftarrow KF^{(i)}(\boldsymbol{z}_k), \quad i \in \{1, 2, 3\}$$
$$(\nu_k^{(0)}) \leftarrow Y_k^{(0)}(\boldsymbol{z}_k)$$

2. Each Bayes filter receives the data from *one* Kalman filter and the static

hypothesis model. They each form a hypothesis based on the L last samples

$$\mathcal{H}_i \leftarrow BF^{(i)}(\nu_k^{(0)}, \nu_k^{(i)})), \quad i \in \{1, 2, 3\}$$

3. The hypotheses are combined using a logical OR gate:

$$H_D = \mathcal{H}_1 \vee \mathcal{H}_2 \vee \mathcal{H}_3$$

Each Kalman filter, $KF^{(i)}$, is initialized to $\boldsymbol{x}^{(i)*}$. The estimate $\hat{\boldsymbol{x}}_k^{(i)}$ will then be updated as measurements $\boldsymbol{z}_k$ are made available.

Note that we use 3 different Bayes filters. Each filter is initialized according to the transition probability given by the first row of Π^h. Each Bayes filter is designed to detect different faults. This partitioning avoids the situation in which different filters $KF^{(i)}$ start competing for the probability space. This could occur because multiple Kalman filters often outperform the static model $Y^{(0)}$ when an error occurs. In practice, the detection window reinitializes the filter probabilities to the prior distribution.

4.4 Fault Isolation

The fault isolation algorithm aims at isolating the true system fault after some fault has been detected. The basic idea of the algorithm is to generate a set of static model hypotheses, and alter the airspeed to introduce a perturbation. The static model of the true model hypothesis will then outperform the false ones that will drift as a consequence of the perturbation. It is important that the models have static parameters because otherwise the parameter estimation will eventually mask the faults and not be helpful to isolate them. A block diagram of the algorithm is given in Figure 4.

The isolation algorithm works as follows:

1. The airspeed V_a is increased. All Kalman filters estimates $\hat{\boldsymbol{x}}^{(i)}$ will quickly change due to the airspeed change. The estimates are then given time to stabilize. This step can be seen in Figure 2.

2. The algorithm then samples the estimates $\hat{\boldsymbol{x}}_{k_S}^{(i)}$ at some time k_s. The time k_s is given by a clock signal, as shown in Figure 4. This is used to generate the static models

$$\boldsymbol{Y}^{(i)} \leftarrow \hat{\boldsymbol{x}}_{k_s}^{(i)}, \quad i \in \{1, 2, 3\}$$

3. The airspeed V_a is decreased back to its original value.

4. The innovations $\nu^{(i)}$ of the static models are given to a Bayes filter. The filter generates a hypothesis based on the last L samples. The filter is initialized according to the first row of Π^f.

$$H_I \leftarrow BF(\nu_{k-L:k)}^{(1)}, \nu_{k-L:k)}^{(2)}, \nu_{k-L:k)}^{(3)}) \tag{39}$$

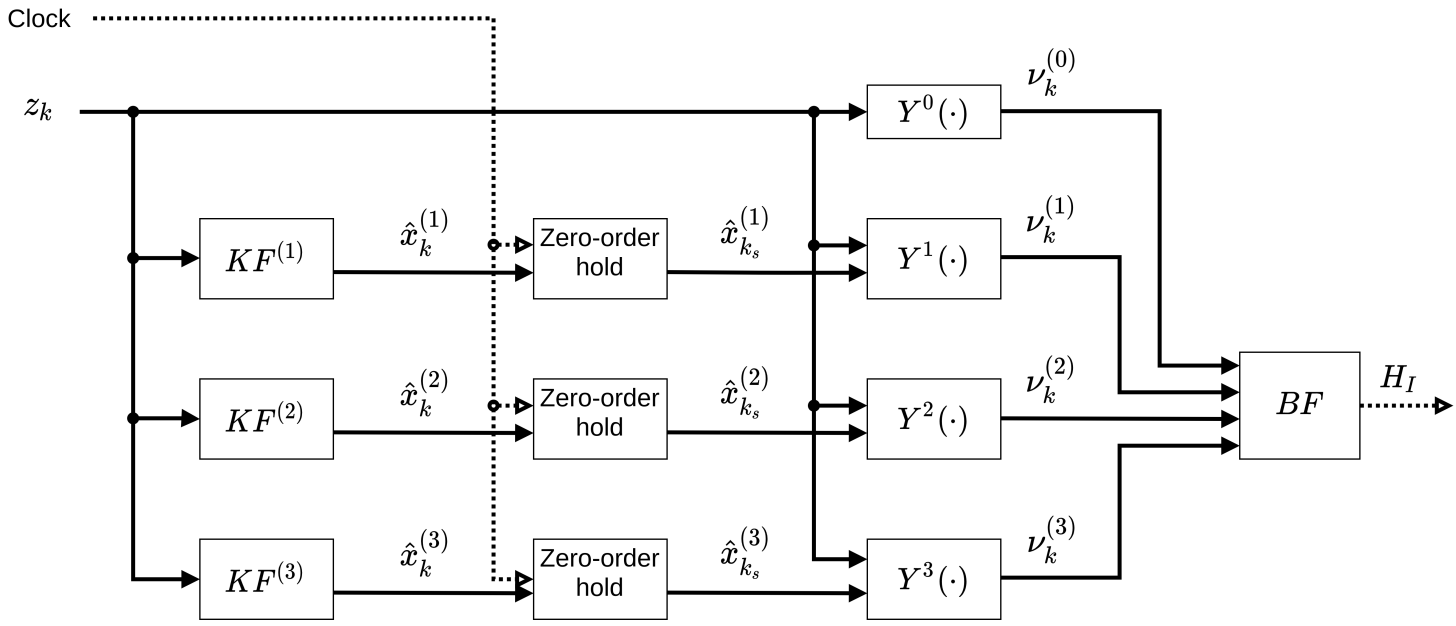

Figure 4: Block diagram with fault isolation algorithm

In practice, the isolation algorithm inherits state estimates from the detection algorithm. That is, the Kalman filters from the detection algorithm are kept running until the sampling time k_s. The isolation models then inherit these values.

5 Simulation Setup

5.1 Simulating Aircraft Control and Dynamics

The simulation of aircraft is distributed over two independent systems - Simulink for flight dynamics and Ardupilot for the UAV control.
Simulink is resposible for simulating the flight dynamics and the physical environment. The Simulink model includes the propulsion model (from equations $1 - 4$) and aerodynamics of the X8 fixed-wing UAV based on [6, 7]. This work has been further developed by the inclusion of fault dynamics. Simulink calls MATLAB scripts to calculate different dynamics.

Ardupilot manages the UAV control systems within a software-in-the-loop framework. This includes the autopilot. The autopilot is responsible for taking the UAV through a pre-defined mission. The simplest mission available - a straight line - is used for testing. Online control of the UAV is achieved using a set of LUA scripts. These scripts can change the control parameter and the mission itself during-after take-off. A set of Lua scripts communicate with Ardupilot using the MAVLINK protocol.

5.2 Fault Simulation

The MATLAB scripts can be altered to simulate fault dynamics. The propulsion model (equation 1) is used for simulating propeller dynamics. This is done by a MATLAB script that also stores the parameters of the model. Faults are implemented by changing these parameters during the simulation. The propulsion model is called in every simulation loop. Dynamic faults are then achieved by perturbing the relevant parameters by small increments in every loop. The size and timing of these perturbations is determined by some *fault dynamic* function.

Different faults are achieved by changing different parameters. The transitions are determined by parameters that change value. The implemented faults have only implemented transitions from a healthy state. The system will either stay in this state or transition to a specific fault state. This be visually understood by noting that the system always starts in the green circles in Figure 1. Naturally, future work should encompass all possible transitions. The implication of this is that transitions from a faulty to a healthy state have not been tested.

5.2.1 Simulated Fault Dynamics

The fault dynamics are simulated using a sigmoidal function. The sigmoid function was first introduced in the project work [36]. The function gradually increases over 50 seconds time span. This occurs after around 90 seconds, as illustrated in Figure 5. This is a pragmatic choice to model a system where a mathematical model is lacking. All the system faults are simulated according to

the same sigmoidal function. This goes for both the rise time and the relative growth of the fault.

The icing fault dynamics have can be simulated in numerous ways. For example, all the parameters $C_{Q,0}$, $C_{Q,1}$ and $C_{Q,2}$ from equation 3 could follow their own trajectories. This would correspond to a 3-dimensional icing model. Alternativly, a scalar icing model could be used. Which model is a more realistic one is an open question. Both models have been implemented and tested in Section 5.3. However, the simulation results in Section 9 are based on a scalar model. The noise analysis in Section 5.3 analyze both scenarios.

For a given fault, the associated variable reaches a steady state after a proportional change by a factor of 1.1. This is exemplified in Figure 5. The figure shows the fault development of θ_1.

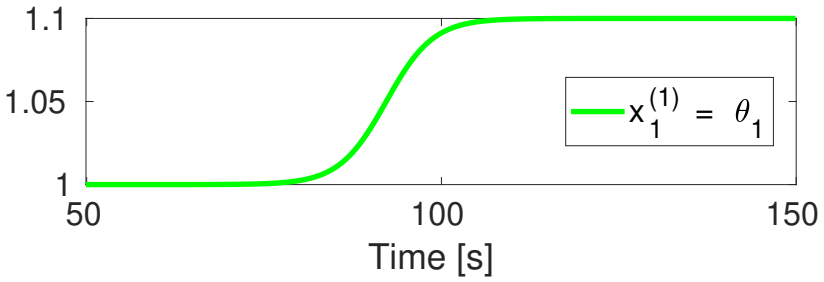

Figure 5: This plot shows simulated fault dynamics of $\boldsymbol{x}^{(1)}$.

5.3 Measurement Noise

The framework relies on measuring ω, V_a and I_e. Naturally, these measurements will be subjected to noise, w_i, w_{V_a}, and w_ω. This noise has been simulated as additive, zero mean, Gaussian noise. A simple block diagram of this is shown if Figure 6.

Tests have been conducted to evaluate how well the icing parameters can be estimated for different levels of noise. These tests are given in the following.

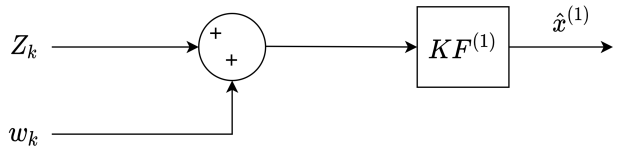

Figure 6: This block diagram shows the simple logic behind the experiments. The noise w_k is added to a noiseless measurement sequence z_k. This data is given to a Kalman filter that produces an estimate $\hat{\boldsymbol{x}}^{(1)}$.

The following noise levels were tested:

- Noise free: $\boldsymbol{R} = \mathbf{0}$
- Low noise: $R_{V_a} = 10^{-6}$, $R_\omega = 5 \cdot 10^{-3}$, $R_{I_e} = 10^{-7}$
- Moderate noise: $R_{V_a} = 7 \cdot 10^{-4}$, $R_\omega = 5 \cdot 10^{-1}$, $R_{I_e} = 10^{-5}$

Where $R = \text{diag}(R_{V_a}, R_\omega, R_{I_e})$.

The noise levels corresponding to *moderate* noise levels will be used for the FDI testing. These levels were found using the following reasoning: the standard deviation of the given noise has been chosen to lie between 0.1% and 0.2% of the mean signal values, i.e., ω has noise covariance $5 \cdot 10^{-1}$, V_a has covariance $7 \cdot 10^{-4}$, and I_e has covariance 10^{-5}.

Note that it is permissible for these noise levels to be less than that of real world sensors. This is because the model is not limited to raw measurements. For example, the FDI typically runs at a lower sampling rate than the raw measurements, which means that decimating (or averaging) several measurements would effectively reduce the noise. Moreover, the electrical model from [17] opens the door to estimating ω or I_e, or both, with a Kalman filter. The model could then make use of the less noisy $\hat{\omega}$ and $\hat{I}_e$.

The square terms in equations 3 and 4 will affect the noise. Specifically, the squaring of Gaussian noise components will result in the introduction of χ^2 noise terms [36]. Thus, the distribution of the measurement noise w_k from equation 17 will be a mixture of Gaussian and χ^2 terms. Thus, the expected value of w_k will be positive. This will introduce a bias to the system. However, for the given noise levels, this bias is small enough to be ignored.

5.3.1 Noise Testing

This section will test our ability to estimate the icing parameters for different levels of noise. The tests are conducted for 3 different levels of additive noise. This gives a picture of the possible operating range of the Kalman filter. In each scenario, the Kalman filters have been tuned based on the given levels of noise.

The noise tests have been conducted for two icing models from Section 3.3. Specifically, the *One-parameter* method and the *reduced sub-space* methods will be tested. A comparison between the two methods follows naturally. However, as noted in Section 3.3, the comparison is very limited - the models can not be falsified based on simulations alone. As a consequence, the two methods are not tested on the same data sets. Instead, they are tested on the data sets where they are assumed to perform well. Thus, the *One-parameter* model has been tested in a simulation where the fault dynamics are 1-dimensional. The subspace method has been tested in a scenario where the fault is 3-dimensional.

The one parameter model was tuned and tested for the three different noise levels. The results can be seen in Figure 7. It can be seen that the noiseless scenario is trivial to estimate. However, the low and moderate cases are much more

challenging. The same result can be found for the *reduced sub-space* method. The results can be seen in Figures 8, 9 and 10. This method differs in the fact that it tries to estimate a 2-dimensional state space. It can be seen that both dimensions can be estimated in the noiseless case. The method is designed such that $x_2^{(1)}$ is less observable than $x_1^{(1)}$. It is clear from Figure 8 that it is difficult to estimate $x_2^{(1)}$ even in the noiseless case. Figure 9 shows that the estimate $\hat{x}_2^{(1)}$ is very slow to converge in the low noise scenario. Figure 10 shows that the estimate $\hat{x}_2^{(1)}$ fails at converging to the true state. This clearly indicates that a 2 parameter model is unfit if the measurement noise is at a realistic level.

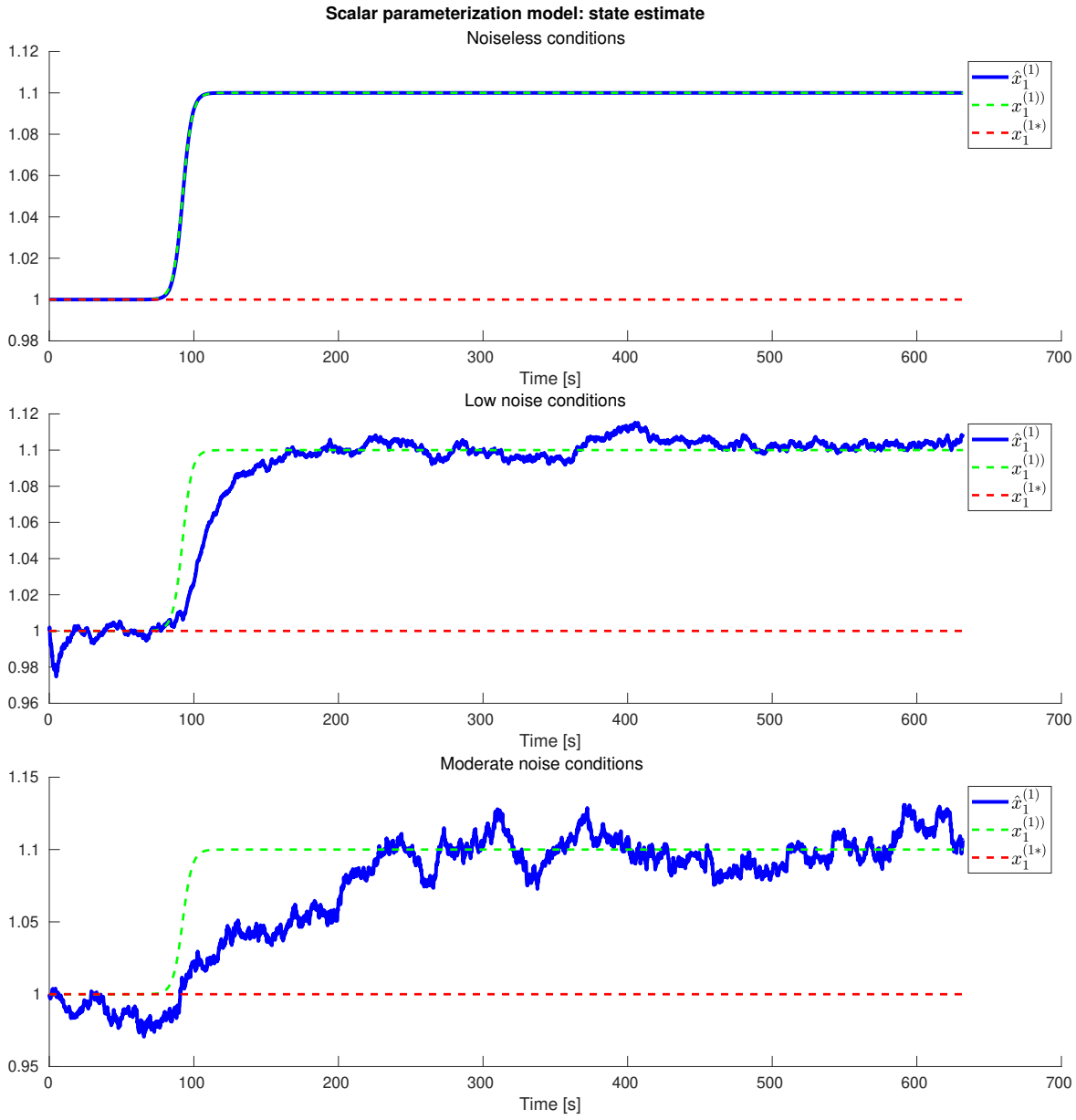

Figure 7: This figure shows the state estimate for the one-parameter model. The top row has zero noise, while the bottom row has the highest levels of the three. The simulated icing dynamics were scalar.

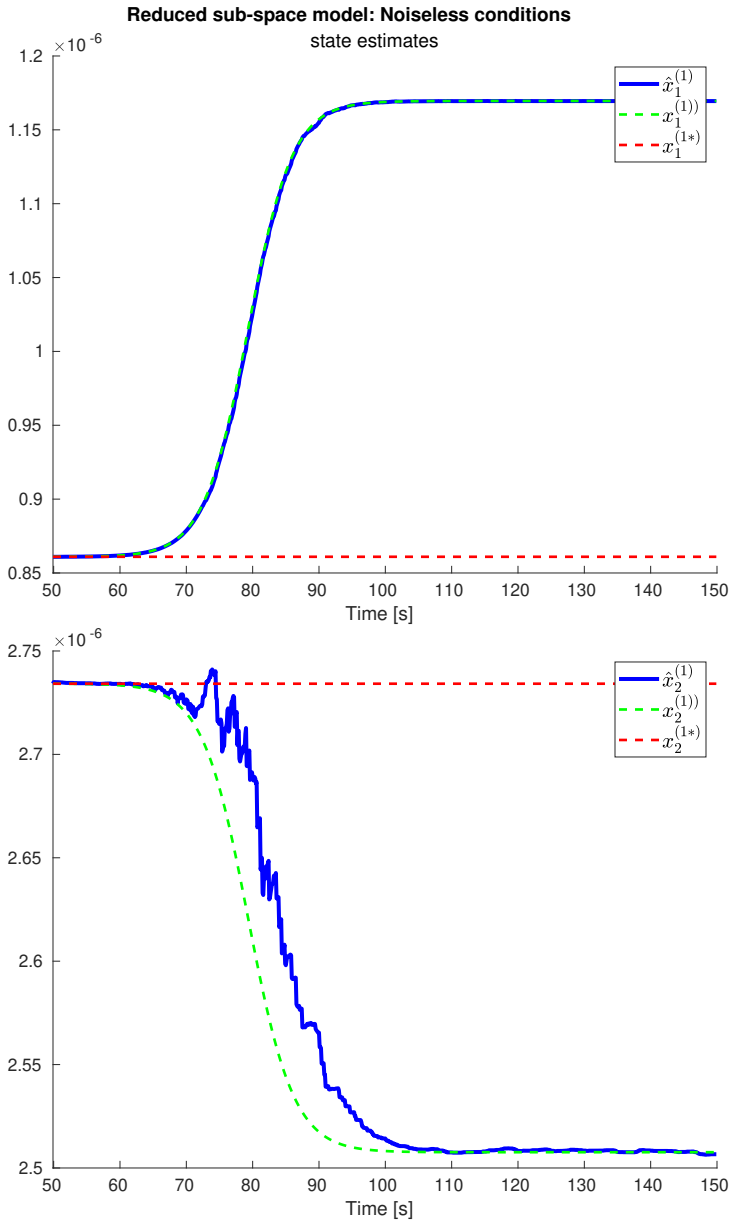

Figure 8: This figure shows the state estimates of $x^{(1)}$ and $x^{(2)}$ in noiseless conditions. The simulated icing dynamics were 3 dimensional.

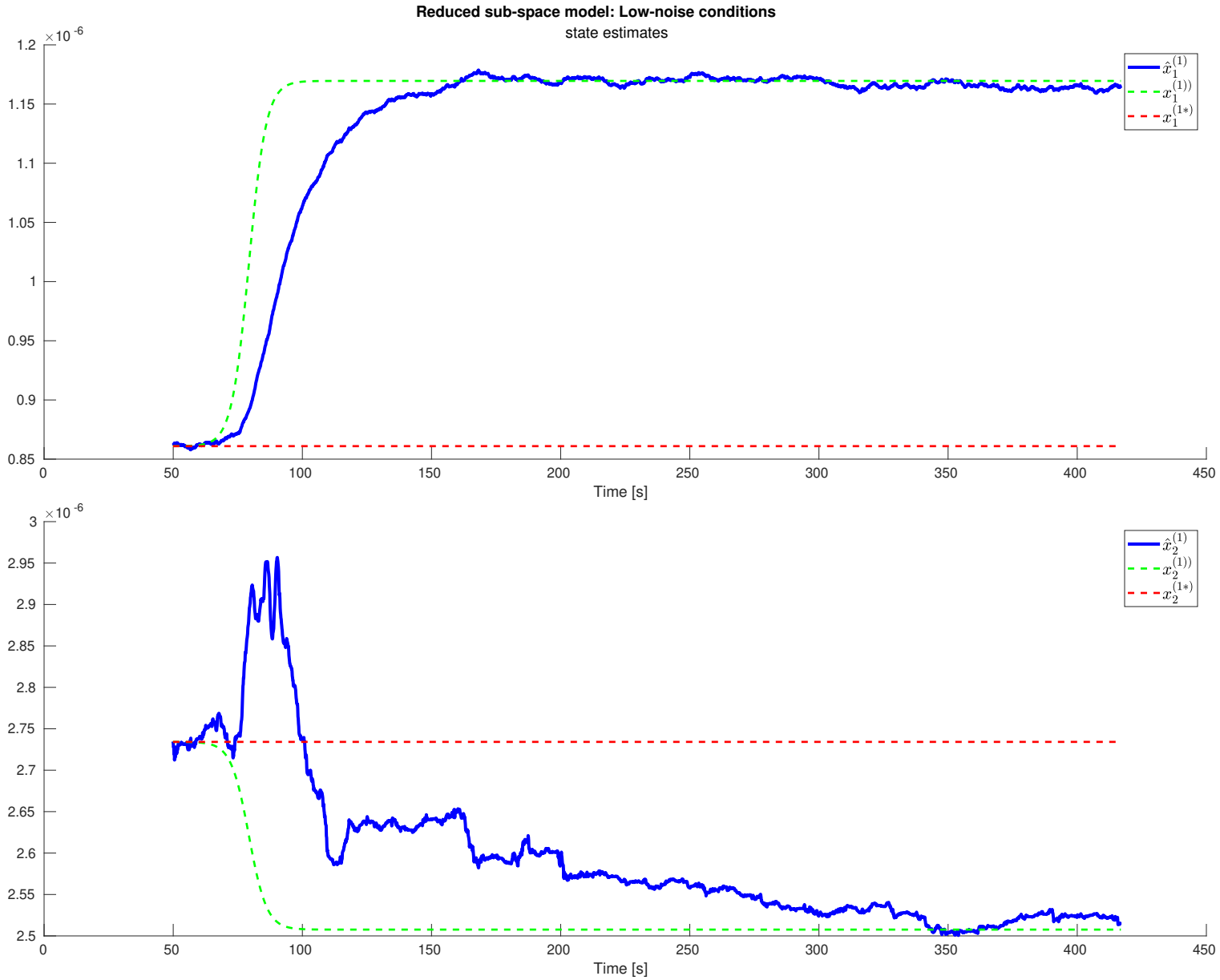

Figure 9: This figure shows the state estimates of $x^{(1)}$ and $x^{(2)}$ in low noise conditions. The simulated icing dynamics were 3 dimensional.

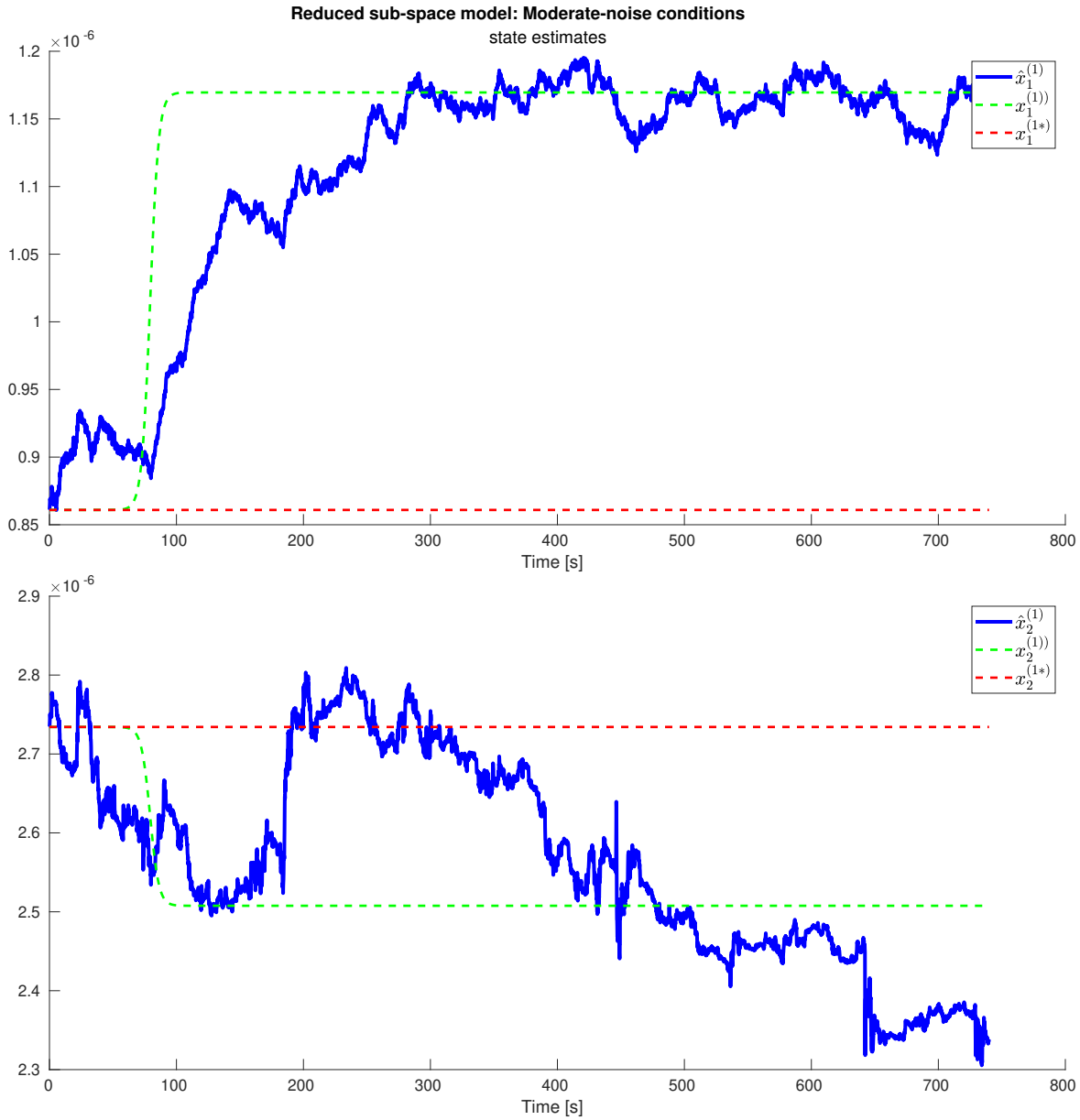

Figure 10: This figure shows the state estimates of $x^{(1)}$ and $x^{(2)}$ in moderate noise conditions. The simulated icing dynamics were 3 dimensional.

6 Tuning

The success of the **FDI** framework depends on rigours tuning. The most important tuning aspects must therefore be discussed. We will specifically cover how to tune both the Kalman filter and the Bayes filter.

6.0.1 Kalman Filter Tuning

Tuning the Kalman filters is essential for achieving a good results. However, it is assumed that the reader is familiar with tuning Kalman filters. Therefore, the three most important aspects will be covered briefly.

1. The process noise covariance $\boldsymbol{Q}_k^{(i)}$ of Kalman filter $\boldsymbol{KF}^{(i)}$ should be commensurate with the magnitude and time constant of the fault dynamics. Furthermore, different faults, e.g., $\boldsymbol{\epsilon}^{(i)}$ and $\boldsymbol{\epsilon}^{(j)}$, will tend to differ in these respects. Thus, the filters must be tuned independently.

2. The response time of a filter may be more important than its accuracy. Keep in mind that the purpose of the filters is to detect faults quickly. Furthermore, the isolation algorithm should not be executed before the Kalman filters have stabilized around some value. Thus, a quick filter is desirable. An example of this can be seen in Figure 16. The top plots show the noisy estimate of $\boldsymbol{x}^{(2)}$. The given filter was excellent at detecting faults quickly.

3. The goal is not about achieving a perfect estimate. The important point is that the correct filter outperforms all other filters. Figure 15 exemplifies this. It is easy to see that the estimate of $\boldsymbol{x}^{(1)}$ is both slow and noisy. This is due to a low signal to noise ratio. However, the filter still performs adequately for both identification and isolation.

6.0.2 Bayes Filter Tuning

The tuning of the Bayes filter determines both the success of the detection and isolation steps. The main tuning parameters are:

1. $\boldsymbol{S}_k^{(i)}$: The innovation variance of the Bayes filter (which determine the sensitivity).

2. Π^f and Π^d: The Markov matrices (which determine the prior probability distributions)

3. L: The window length (which determine how many samples the filter should use)

There will always be an interplay between these parameters, which should be kept in mind as we discuss them separately. For example, high sensitivity will allow for a shorter window length. The following will frequently refer to Figure 11. Note that this figure is the result of simulations, as covered in Section 9.

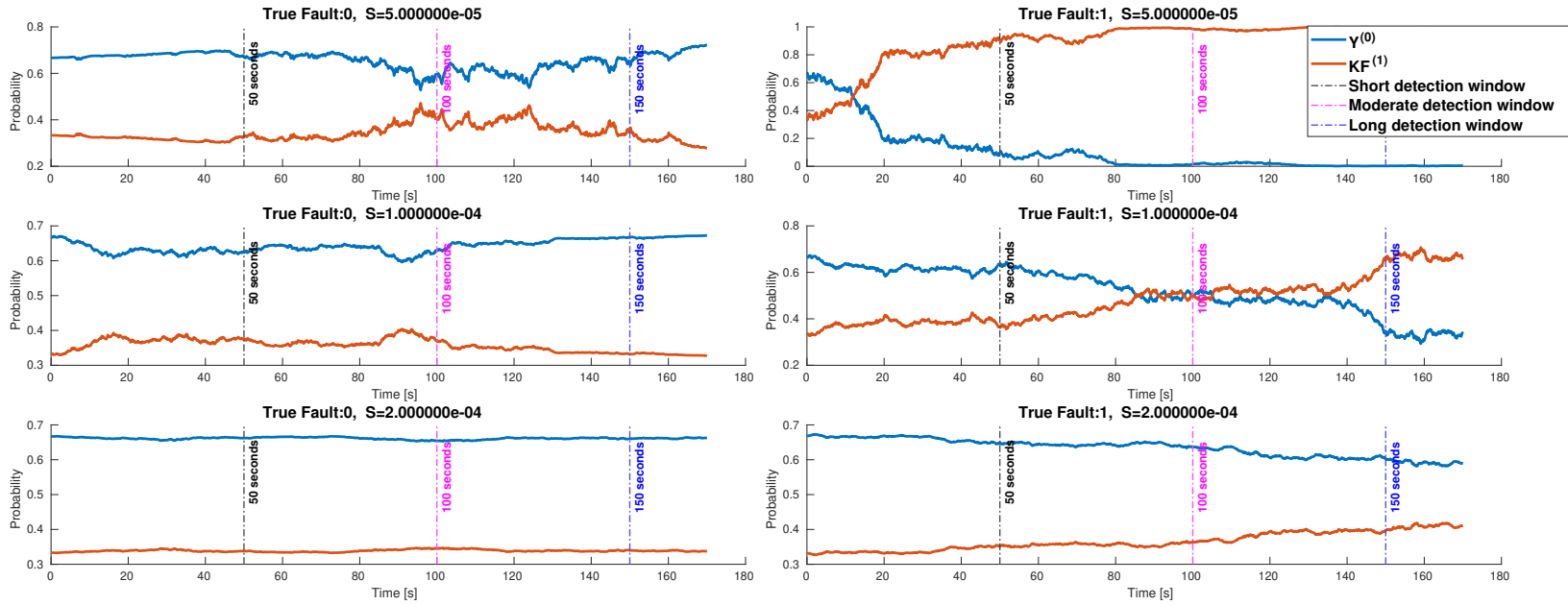

Figure 11: All plots shows the output of a Bayes filter during the detection step. **The system is healthy** (fault = 0) in all the plots in the **left** column. The blue line represents the true hypothesis in the left. **The system is faulty** (fault = 1) in all the plots in the **right** column. Thus, the red line represents the true hypothesis in the right column. The **sensitivity** of the filters is the **highest** in the **top row**. The **sensitivity** is the **lowest** in the **bottom** row. The vertical dashed lines illustrate possible values of the detection window, L.

The sensitivity is the most crucial aspect of the tuning process. Its effect can be seen in Figure 11. All the plots in the left column show scenarios where the system is fault free. The right column shows scenarios where the system is subjected to icing. In this scenario, the icing fault is fully developed for the whole time range displayed. That is, the fault is not gradually increasing, its fully developed. Thus, no change in θ_1 occurs during the execution. The plots in the top row have the highest sensitivity. The sensitivity then decreases down the rows. The innovation $\nu^{(1)}$ from Kalman filter $KF^{(1)}$ is exactly the same within each column.

It should be clear from the top left plot that high sensitivity has obvious problems: the risk of false positives increases drastically. Notice that the red line, i.e., the false hypothesis, almost surpasses the blue line multiple times. High sensitivity also comes with a clear advantage: fast convergence rates. This can be seen in the top right. The correct hypothesis is isolated in about 10 seconds. The opposite extreme is found in the bottom row. The bottom left shows that low sensitivity makes a false positive very unlikely. However, the bottom right shows that detecting the fault would take more than 3 minutes. A compromise is found in the middle row. This was the sensitivity level which functioned the best in the simulations.

The detection window length L must be adapted to the sensitivity. The horizontal lines of Figure 11 show various options for L. Keep in mind that the algorithm will choose the hypothesis with the highest value. This happens when the lines reach the end of a detection window. It should be clear that we must chose a window that is long enough for the correct solution to be chosen. At the same time, the window must not be too long. This has the aforementioned problem that a fault would not be detected fast enough. Furthermore, a core model assumption is that the system does not change state while the Bayes filter is running. For the simulations, a 100 second window was chosen.

The transition probabilities also play an important role. All the plots in Figure 11 are initialized with the same prior distribution. However, it should be easy to imagine the effect on different priors. For example, the top left plot would in many instances return a false positive if the lines started closer together. Conversely, the bottom right plot would return a true positive, if the lines started closer together.

It is straightforward to imagine more sophisticated approaches to determine both Π^h and Π^f. The transition probabilities could be, for example, varied according to environmental factors, such as power load, temperature and humidity. However, in the simulations the Markov matrices, Π^h and Π^f, were selected as simple as possible. Furthermore, only the first row of Π^h and Π^f are of interest. This follows from the fact that all simulations are initialized in a healthy state. The first row of both matrices gives the transition probabilities from healthy states. For the detection step, the first row of Π^h was set to:

$$\Pi^h_1 = \begin{bmatrix} \frac{2}{3} & \frac{1}{3} \end{bmatrix} \tag{40}$$

Thus, remaining in a healthy state is considered twice as likely as transitioning

to a faulty state. The isolation step assumes a discrete uniform distribution:

$$\Pi_1^f = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix} \tag{41}$$

It is often assumed that the probability of remaining in the current state has the highest probability. However, it should be accounted for that Π^f is only used if the detection step has found a transition. This transition introduces a prior that should be accounted for in Π^f.

7 Implementation

The FDI framework from Section 4 has been implemented in MATLAB. This section will cover the most important features of the design, interface and architecture of this software. This sections assumes that the reader is familiar with the algorithms from Section 4. It is also assumed that the reader is familiar with the MATLAB language and UML diagrams. In its entirety, the software spans over a thousand lines of code, spread over multiple scripts. Thus, this review must be limited to high level concepts and explanations. Nevertheless, a few central topics have received disproportionate attention. This is to equip future developers with the tools to work with the code.
Section 7.1 explores the design principles behind the software project. This section will also give the reader a rudimentary understanding of the system structure. This documentation is a general guide to the FDI framework. The focus is therefore on the algorithms from Section 4 and not on the propulsion model from Section 3. A guide on how to implement systems, such as the propulsion model, is given in Section 3.5.
The *model class* is covered in Section 7.3. This introduces a data structure that is essential for understanding the system architecture. The implementation of the FDI framework can then be introduced in Section 7.4. Limitations and pitfalls are covered in Section 7.5.

7.1 Design Principles and Overview

The code has been written with *scalablity* and *generality* in mind. Specifically, the idea is to make it very easy to:

1. Augument and modify the existing FDI framework.
2. Adopt the FDI framework to other systems. For example, it might be worthwhile to test the framework for icing detection on airfoils.

These goals are achieved by working with *classes*. The FDI framework is implemented as the *FDI class*. This class contains all the algorithms from Section 4. All the algorithms and support functions are implemented as separate *methods*. Further, the low level methods function independently and therefore are easy to assemble into new arrangements. Such methods are also written to be invariant of state dimensions. This makes the functions easy to adapt and apply to different systems. The FDI class is covered extensively in Section 7.4.

The *FDI class* is compatible with any system that can be expressed in the state space form. It is therefore system *invariant*. The invariance is achieved by separating the system equations (such as the propulsion model) and the FDI-framework into independent classes. The result is a *system class* and an *FDI class*. This is analogous to the split between the *MODELLING* Section 3 and the *FDI* Section 4 of this text.

The introduction of the *system class* offers two main advantages:

- New systems are *quickly* adopted by writing a *system class* script for the given system. The template for this is given in Section 7.2.6.
- Existing systems are easy to modify - without minding the FDI class.

The system class is covered extensively in Section 7.2

7.2 System Class Implementation

This section will present how systems, such as the propulsion model, can be implemented as *multiple models* in the FDI framework. For each such system, a new script must be written. These scripts define *system classes*. This subsection will show how to implement such a class. Each script will have the same structure.

These efforts are somewhat involved. The explanation has therefore been partitioned into different steps. We start with simple explanations of what information the system class should contain and how to represent this. Example cases **case 1** and **case 2** are then introduced. The subsection is concluded with a general template.

7.2.1 System Models

The system class defines a set of `system models`. The system models define the system equations that underlie a model in the *multiple model* framework.

The system models define the information needed for running Kalman filters and computing innovations for the given model. The system models are passed as *types* to the FDI class when it generates the model class. This concept will be elaborated in the Section 7.3.

The the system models contain the following data:

- The **fault state** that the system model corresponds to. This is given as a positive integer.
- The **method** used to model the fault state. This is necessary because different system models might be used to model the same hypothesis. This is given as a positive integer.
- The **innovation** $\nu^{(\cdot)}$ given as a function handle. This is used by the Kalman filter and the Bayes filter.
- The **time variant measurement matrix** $C^{(\cdot)}$ given as a function handle. This is used by the Kalman filter.
- The **state dimension** d of $x^{(\cdot)}$.

Where `C` is the measurement matrix $C^{(1)}$ and the innovation $\nu^{(1)}$ is given by $\nu^{(1)} = y - C^{(1)}x^{(1)}$. This forms the basis for estimation using a Kalman filter. Keep in mind that the process model $\boldsymbol{A}$ is determined by the state dimension d. This is because $\boldsymbol{A} = \boldsymbol{I}$ where $\boldsymbol{I} \in \mathcal{R}^{dXd}$.

The difference between the *fault state* and the *method* property can be difficult to grasp. The answer lies in the fact that some faults might have many possible representations. For example, subsection 3.3 showed how 3 different methods can model the icing fault state. These methods are implemented as separate system models. These system models then point to the same fault state.

7.2.2 Enumeration Types

The system class is implemented using MATLAB enumeration types. The enumeration types define a set of *system models*. Each enumeration type maps to a system model. This makes the implemented *system models* easy to work with. For example, if a system class has been saved by the name `system_name`, then a system model can be declared by

```
model_1 = System_name.name_of_system_model
```

Properties of a system model can then be accessed using

```
model_1.some_property
```

The naming convention should follow from the hypothesis and the system model itself, for example, the propulsion model

```
fault_2 = propulsion_model.change_in_viscous_friction
```

Now, the innovation of `fault_2` can then be accessed using

```
innovation = fault_2.nu
```

7.2.3 A General System

We will now look at how to find the necessary properties to define a system model. It is assumed that a few things have to be done before the implementation:

- The system equations are reformulated as a zero-mean innovation.
- The *nominal* parameter values are saved to some local file.
- The different system models and hypothesis have been analytically defined.
- The measurement variables z_i are identified.

The system models are then implemented as follows:

Step 1: Formulate the system innovation as a *function handle*. This is straight forward in MATLAB and is shown in the code bellow

```
load(' nominal_values}', ' x_1_nom',...,' x_m_nom', ' c_1',
  ..., ' c_h')
nu = @(z_1, ..., z_n, x_1, ... , x_m) expression

```

Listing 1: Formulating the innovation as function handle

where "`expression`" refers to the system equations. The variables `x_1, ..., x_m` are the potentially faulty parameters. The variables `c_1, ..., c_h` are the parameters that are guaranteed not to change. These are included in "`expression`" and not used further. `z_1, ..., z_n` represent the time variant coefficients of the system. These are not estimated by the FDI-framework but given as inputs.

Step 2: Formulate the innovation `nu`. Further, generalize the number of time variant coefficients z_i by introducing the vector $\boldsymbol{z} = [z_1, z_2, z_3]^T$.

```
nu = @(z, x_1, ...,x_m) nu(z(1), ..., z(n), x_1, ..., x_m)

```

Listing 2: Vectorizing the innovation input.

The new function handle is now a function of the vector $\boldsymbol{z}$. This allows the *FDI class* to be independent of the dimension of `z`. Notice how the original function handle can be used to define the new function handle.

Step 3: For each hypothesis, find the innovation $\nu^{(\cdot)}$. This is done by parameterizing `nu`. For hypothesis $1 < i < m$, we get the following code

```
% x_1_nom and x_m_nom are constants
% x_i is a new variable
nu_i = @(Z, x_i) nu(Z, x_1_nom, ...,  x_i, ..., x_m_nom)

%nu_i is now a function of Z and x_i!

system_model.state_dim = length(x_i_nom)

```

Listing 3: Formulating the innovation $nu^{(i)}$ of model i.

Further, the measurement transition $C^{(\cdot)}$ must be found. This amounts to finding measurement coefficients of the state. This is a bit more involved - especially in the presence of non-linear terms such as $z_1 x_i x_j$. However, a general solution is found with a simple rule

$$C(z) = \nu(z, \boldsymbol{x}^{(i)} = 1, \boldsymbol{x}^{(j)} = \boldsymbol{x}^{(j)*}) - \nu(z, \boldsymbol{x}_i = 0, \boldsymbol{x}^{(j)} = \boldsymbol{x}^{(j)*}), \quad i \neq j \tag{42}$$

Notice that $\boldsymbol{x}^{(i)}$ is substituted with 1 in the first term and 0 in the second. This solution will be valid as long as $\boldsymbol{x}^{(i)}$ is linear with itself. In terms of code, this is given by

```
C_i = nu(Z, x_1_nom, ... 1, ..., x_m_nom) - nu(Z, x_1_nom,
 ..., 0, ..., x_m_nom)

```

Listing 4: A general approach to finding the measurement coefficients of the state $\boldsymbol{x}^{(i)}$.

The state dimension is readily obtainable from the nominal values:

```
    state_dim = length(x_i_nom)

```

Listing 5: Formulating the innovation $nu^{(i)}$ of model i.

Step 4: Formulate a *switch* for assigning each system model to a fault state. The fault free system model is also defined. This model only necessitates the the nominal values. The fault free system model is also dimensionless and does not need the measurement matrix C. A template for formulating the switch is given in the following code:

```
switch fault_state
case 0  %Fault free
    system_model.nu = @(Z) nu(Z, x_1_nom, ...,  x_i_nom, ...,
    x_m_nom)
      ...
case 1
  system_model.C = nu(Z, x_1, ... x_i_nom, ..., x_m_nom) - nu(Z, 0,
     ..., x_i_nom, x_m_nom)
  system_model.nu = @(Z, x_1) nu(Z, x_1, ...,  x_i_nom, ...,
    x_m_nom)
  system_model.state_dim = length(x_1_nom)

...

case i
  system_model.C = nu(Z, x_1_nom, ... 1, ..., x_m_nom) - nu(Z,
    x_1_nom, ..., x_i, x_m_nom)
  system_model.nu = @(Z, x_i) nu(Z, x_1_nom, ...,  x_i, ...,
    x_m_nom)
  system_model.state_dim = length(x_i_nom)
...

case m
  system_model.C = nu(Z, x_1_nom, ... x_i_nom, ..., 1) - nu(Z,
    x_1_nom, ..., x_i_nom, 0)
  system_model.nu = @(Z, x_m) nu(Z, x_1_nom, ...,  x_i_nom, ...,
    x_m)
  system_model.state_dim = length(x_m_nom)
end
```

Listing 6: Formulating a MATLAB switch for assigning system models to fault states.

Some system classes might require different methods for representing the same fault state. For example, Section 3.3 presents 3 methods for tracking icing. Comparing the different methods is an important part of the experimentation process. This has therefore been included as part of the class structure.

The code solution is very simple - a switch inside the existing switch. The `fault_state` will determine the outer switch, while the `method` will determine the inner switch. This is shown in the code bellow:

```
switch fault_state
  case 0  %Fault free
```

```
    system.nu = ...
  case 1
    switch method
      case 1
        system_model.nu = ...
        system_model.C = ...
        system_model.state_dim = ...
      case 2
        system_model.nu = ...
        system_model.C = ...
        system_model.state_dim = ...
      ...
      case n
        ...
    end
  case 2
    ...
  case m
end
```

Listing 7: This code shows a template for including multiple methods for the same fault type.

Fault states that do not require a method argument can just ignore the `method` type and skip the inner switch case.

7.2.4 Case 1: Multiple Models

The following illustrates a generic example of such a class. We will start by looking at how to formulate the system equations. For the following example, the system is given by

$$z_3 - c_1 z_1 = c_1 c_2 z_2$$

Let us say that we are trying to detect faults in c_1 and c_2. Thus, we have two possible models. These parameters have nominal values c_i^*. Further, z_1 and z_2 are measured state variables. The zero mean innovation is found by moving all variables to the RHS.

$$0 = c_1 z_1 + c_1 c_2 z_2 - z_3 \tag{43}$$

The system can then be described using a MATALB *function handle*

```
nu = @(z_1, z_2, z_3, c_1, c_2) c_1*z_1 +c_1*c_2*z_2-z_3;
```

The function handle `nu` can now be evaluated based on function inputs. For example, `h(1,1,1,1,1)` returns 1.
The measurement terms should then be z_i gathered into a vector $\boldsymbol{z} = [z_1, z_2, z_3]^T$.

```
nu = @(z, c_1, c_2) nu(z(1), z(2), z(3), c_1, c_2)
```

This function handle is the basis for defining a set of new models. We now want to create a model where only c_1 can change with time. c_1 is therefore referred to as the state $x^{(1)}$ of the new system model. The corresponding measurement model is then given by two new function handles

```
load('nominal_values', 'c_1_nominal', 'c_2_nominal')

nu_1 = @(z, x) nu(z, x, c_2_nominal,)
C_1 = @(z) nu(z, 1 , c_2_nominal) -  nu(z, 0, c_2_nominal)
```

The last line is easy to verify

$$\begin{aligned} C^{(1)} &= 1 \cdot z_1 + 1 \cdot c^* z_2 - z_3 - (0 \cdot z_1 + 0 \cdot c_2^* z_2 - z_3) \\ &= z_1 + c_2^* z_2 \end{aligned}$$

The same approach can be applied to c_2. This yields *multiple system models*

```
switch fault_state
  case 0  %Fault free
    system_model.nu = @(z) nu(z, c_1_nominal, c_2_nominal)
  case 1
    system_model.C = nu(z, 1, c_2_nominal) - nu(z, 0, c_2_nominal)
    system_model.nu = @(z, x) nu(z, x, c_2_nominal)
  case 2
    system_model.C = nu(z, c_1_nominal, 1) - nu(z, c_1_nominal, 0)
    system_model.nu = @(z,x) nu(z, c_1_nominal, x)
end
```

Listing 8: This code shows the model switch for case 1.

7.2.5 Case 2: Multiple Methods for the Same Model

This subsection will show an example of how to implement different methods for the same fault state. This example will cover how to implement two of the methods that were introduced in subsection 3.3. Before reading further, keep in mind that multiple system models will not be used in a live application. This step is only necessary in a development phase.

This example assumes a system of the following form

$$0 = \boldsymbol{z}_{1:3}\boldsymbol{x}^{(1)} + z_4 x^{(2)} \tag{44}$$

$$= [z_1, z_2, z_3] \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} + z_4 x^{(2)} \tag{45}$$

In this case, faults can occur in either $\boldsymbol{x}^{(1)}$ or $x^{(2)}$. This gives 3 fault states (including the fault free state).

Equation 44 can be expressed as the function handle

```
nu = @(z, x_1, x_2) z(1:3)*x_1+z(4)*x_2;
```

where the step vectorization was skipped.
This function handle works well for making a system model for the full state space. However, in this case it could be hard to estimate the full state space of $\boldsymbol{x}^{(1)}$. Therefore, the 1-parameter parametrization and the reduced state-space system models from subsection 3.3 will also be tested. To reiterate, the 2 system models do the following:

1. **The 1-parameter scaling model** will scale the nominal values $x^{(1)*}$ by the parameter p. Thus, the expression will now be $\boldsymbol{x}^{(1)} = \boldsymbol{x}^{(1)*} \cdot p \in \mathcal{R}$.

2. **The subspace model** will estimate the subspace of the transformed variable $\tilde{\boldsymbol{x}}^{(1)}$. The subspace is in 1 or 2 dimensional system. The transformed variable $\tilde{\boldsymbol{x}}^{(1)} = [\tilde{x}_1^{(1)}, \tilde{x}_2^{(1)}, \tilde{x}_3^{(1)}]$ is ordered by observability. For example, $\tilde{x}_1^{(1)}$ will be easier to observe than $\tilde{x}_3^{(1)}$ and so forth. The system model assumes that only $\tilde{x}_1^{(1)}$ and $\tilde{x}_2^{(1)}$ change with time. The transformation is given by T. For nominal values, we get $\boldsymbol{x}^{(1)*} = \boldsymbol{T}\tilde{\boldsymbol{x}}^{(1)*}$. We assume that the transformation T is available prior to initialization.

These system models require a different function handles than the one above. The key to handling this lies in the function handle formulation. The naive solution is to introduce a new function handle expression for every system model. Fortunately, this is not necessary. In fact, only one modification is needed for the function handle - the introduction of a *flexible* parameter `P`

```
nu = @(z, P, x_1, x_2) z(1:3)*P*x_1+z(4)*x_2;
```

Listing 9: This code introduces the *flexible* variable P.

The power of this approach arises from the fact that `P` can be substituted with scalars, vectors and matrices. This is why `P` is referred to as *flexible*. The 1-parameter model will substitute `P` for the scalar parameter p. In this case, the parameter `P` takes the role of the state $x^{(1)}$. Contrarily, the reduced model will substitute the parameter `P` for transformation T. In this case, `P` is simply a non-changing transformation. We see that `P` can serve widely different purposes. Other system models can ignore `P` by using `P=1`. This is allowed because the choice of system model has no effect on the nominal values. That is

$$p^* \boldsymbol{x}^{(1)*} = T\tilde{\boldsymbol{x}}^{(1)*}, \text{ with } p^* = 1 \tag{46}$$

This methodology will now be implemented as code. All the `case` blocks bellow are assumed to be within a `method` switch. This is showed in the code bellow

```
switch fault_state
  case 0 %Fault free case
    system_model.nu = ...
  case 1
    load('nominal_values.m', ' x_1_nominal', ' x_2_nominal')
    switch method
      %All bellow examples are here
    end
end
```

First, the simplest case - the full state model - $\boldsymbol{x}^{(1)} \in \mathcal{R}^3$. This gives

```

case 1 %Full state space}
  P_nominal = 1;
  system_model.nu = @(z, x) nu(z, P_nominal, x, x_2_nominal)
```

```
  state_dim = 3;
  system_model.C = @(z) ...
   [nu(z, P_nominal, [1;0;0], x_2_nominal)-nu(z, P_nominal,
    [0;0;0], x_2_nominal), ...
   nu(z, P_nominal, [0;1;0], x_2_nominal)-nu(z, P_nominal, [0;0;0],
     x_2_nominal), ...
   nu(z, P_nominal, [0;0;1], x_2_nominal)-nu(z, P_nominal, [0;0;0],
     x_2_nominal)]
```

Listing 10: Implementing the full state space.

Note that `system_model.C` is now a vector valued function. This follows directly from $\boldsymbol{x}^{(1)}$ being a vector. The 1-parameter system model uses `P` as a state and will be substituted with the variable `x`.

```
case 2 %One-parameter model
  system_model.nu = @(z, x) nu(z, x, x_1_nominal, x_2_nominal)
  state_dim = 1;
  system_model.C = @(z) nu(z, 1, x_1_nominal, x_2_nominal) ...
          - nu(z, 0, x_1_nominal, x_2_nominal)
```

Listing 11: Implementing the One-parameter model.

Notice that only the nominal value of $\boldsymbol{x}^{(1)*}$ is used. Further, `system_model.C` is now a scalar valued function. The subspace model is presented in the code bellow.

```
case 3 %Square subspace system_model
  load('nominal_values.mat', ' T')

  %Find the nominal transformed variable
  x_nom_trf = T * x_1_nominal;

  system_model.nu = @(z, x) nu([z, T, [x;x_1_nom_trf(3)],
    x_2_nominal)
  state_dim = 2;

  subtraction_term = nu([z;0], P_nominal, [0;0;x_1_tld(3)],
    x_2_nominal)

  system_model.C = @(z) ...
        [nu([z;0], P_nominal, [1;0; x_1_nom_trf(3)], x_2_nominal)-
    subtraction_term, ...
        nu([z;0], P_nominal, [0;1; x_1_nom_trf(3)], x_2_nominal)-
    subtraction_term]
```

Listing 12: This codes show how the sub-space model has been implemented.

The nominal values of the transformed variable $\tilde{\boldsymbol{x}}^{(1)}$ are calculated after loading T. $\tilde{x}_3^{(1)*}$ is a fixed value in both expressions. The resulting state space is two dimensional. Note that this state space should be initialized accordingly to $\tilde{\boldsymbol{x}}_{1:2}^{(1)*}$. The scalar variant of the model reduction is implemented in a similar way. The difference is that $\tilde{x}_2^{(1)}$ is also held constant.

7.2.6 Implementing the MATLAB Enumeration Type

The exact way to implement the system class using MATLAB enumeration types is covered here. The script has three segments: *enumeration*, *properties* and *the constructor*.

- **Enumeration**: A lists of all allowed system models. Each system model is specified by its *unique* name. Each name has a set of input arguments. These arguments are enclosed in parenthesis on the same line.

```
  name_of_model   (argument_1, argument_2)
```

  These are used by the class constructor in the given order.

- **Properties**: A list of all variables that *can* be associated with each system model. This includes the function handles.

- **The Constructor** assigns properties values to the different enumeration types. The constructor takes the enumeration arguments as inputs.

```
  system_name(argument_1, argument_2)
```

A template is given bellow:

```

classdef system_name % <- pick a name
   enumeration
       %Type_name          (fault_state, method)
       fault_free           (0, 1) %  <- Constructor arguments
       name_1      (1, NaN) % <- Fill in a name
     ...
       name_mn     (m, n)
   end
   properties
   %Constants
   system_model_idx; hypothesis_idx; state_dim; innovation_dim
   %Function handles
   nu; C
   end
   methods
       function system_model = system_name(hypothesis,
   system_model_idx)
     load(' nominal_values', ' x_1_nom',...,' x_m_nom', ' c_1',
   ..., ' c_h')

     nu_orginal = @(z_1, ..., z_n, x_1, ... , x_m) expression
     nu = @(z, x_1, ...,x_m) nu_orginal(z(1), ..., z(n), x_1, ...,
    x_m)

     switch fault_state
         case 0 %Fault free
               system_model.nu = @(Z) nu(Z(1), ..., Z(n), x_1_nom,
    ...,  x_i_nom, ..., x_m_nom)
          case 1
          system_model.C = nu(Z, x_1, ... x_i_nom, ..., x_m_nom)
   ...
```

```
                        - nu(Z, 0, ..., x_i_nom, x_m_nom)
                system_model.nu = @(Z, x_1) nu(Z(1), ..., Z(n), x_1,
    ...,  x_i_nom, ..., x_m_nom)
                system_model.state_dim = length(x_1_nom)
              ...
              case m
      switch method
        case 1
        ...
        case n
      end
      end
        system_model.index_idx = system_model_idx;
      system_model.hypothesis_idx = hypotheses;
      % The innovation dimension is the same for all system_models
      system_model.innovation_dim = some number; % <- set the
    innovation dimension
        end
    end
end
```

Listing 13: This code gives a template for implementation of the enumeration types.

The template assumes that some fault states only have one method. For example, `fault_state=1` only uses one method. The first system model, `name_1`, has the fault state argument `fault_state=1`. The `method` argument has therefore been set to `NaN`. This is not the case for the fault state `m` witch has `n` types. The system model `name_mn` has the fault state `m` and the method type `n`.

7.3 Model Class and Models Container

The `model` class and `models` container will now be presented. In this context, *model* has an extended meaning from what we saw in Section 4. In this context, `model` refers to a class that stores all data associated with the model. For example, a detection model is associated to the state estimate $\hat{\boldsymbol{x}}^{(i)}$. Thus, the corresponding `model` stores both the state $\hat{\boldsymbol{x}}^{(i)}$ and the the fault state i.
The `models` container stores all the `models` that the FDI class has stored. `models` is a data structure within the FDI class. A `model` is itself a super class and it contains a *system model* (from the system class) and a *model type*. The model types are independent of the system class. The different types are *fault free*, *detection* and *isolation*. The model class structure has been visualized as an UML class diagram in Figure 7.3. It can be seen that the *model class* has an *aggregated* relationship with the *system class* and the *model type*. An instance of the *model class* is required to contain one system model and one model type. The figure also shows the relationship between the *model class* and the *models* container. It can be seen that there is a *composition* relationship between the *model class* and the *models* container - if the container is deleted, then all the model instances are deleted as well. Further, the container can contain an unlimited number of instances. It is important to note that the models

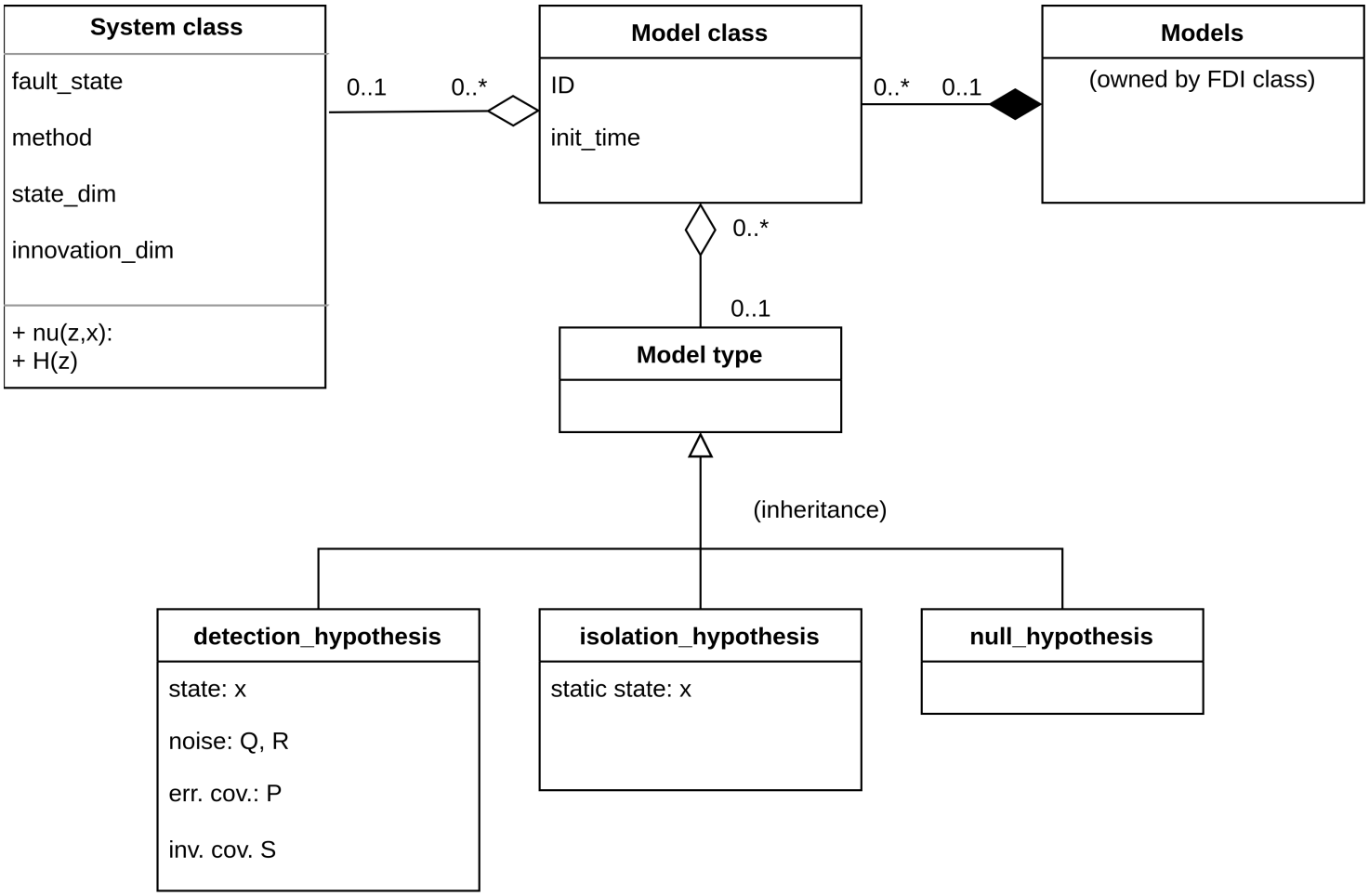

Figure 12: A UML class diagram. The diagram shows the structure of the *model class*.

container is owned by the FDI class. It can be seen in Figure 7.3 that the `model` super class has 4 main properties:

- `id`: A unique ID for every model.
- `init_time`: The time when the model was spawned.
- `model_type`: An *aggregated* class.
- `system_class`: An *aggregated* class (as discussed in subsection 7.2).

The model types, *detection* and *isolation*, are used in the detection or isolation steps, respectively. The fault free model is used in both steps. The different steps require different data. That is why the different model types have different sets of properties. All the properties and methods needed for the fault free model are embedded in `system_type`. The *fault free* model type is empty.

The detection type is the most involved. It includes everything needed to initialize a Kalman filter:

- `x`: The state (which starts at the initial value).
- `Q` and `R`: The process and measurement noise covariances.
- `P` and `S`: The state and innovation covariance.

In fact, one Kalman filter is spawned for every detection model. The isolation class is comparably sparse. It only contains the static estimate `x`. The isolation model doesn't need its own Kalman filter - it is initialized based on the estimates contained in previous detection models.

The `models` container has no limit to the amount `model` instances it can store. Further, the FDI class will function with any number of model. The FDI method from Section 4 would only require 3 detection models and 3 isolation models. Thus, the FDI class gives flexibility beyond the presented method. The flexibility has multiple advantages. From a tuning perspective, this allows for rapid development. For example, many model with the same `system type` can be spawned concurrently. The only difference may be the Kalman tuning parameters Q and R. It is easy to compare different tuning options when the algorithm terminates. This simplifies the tuning effort drastically. Furthermore, different system types, with the same model id, are also easy to compare.

The flexibility also makes the FDI class is easy to modify. For example, a new FDI algorithm might require 3 detection models for possible fault, e.g. three icing models. This new feature can be added by initializing the FDI class differently. No changes needed.

7.4 FDI class

This section will focus on the methods of the FDI class. This involves seeing how the algorithms from section 4 has been structured into separate functions. The interaction between these functions has been detailed as UML sequence

diagrams. These can be seen in figure 7.4.1 and 7.4.1. After this, we'll look at how the main tuning parameters of the FDI class. the section is finished with a brief overview of how to the FDI class is initialized and run.

7.4.1 FDI functions and sequencing

The main functions of the FDI class is given in the 2 lists bellow. The list bellow covers the *interface* of the FDI class. These functions are responsible for implementing the algorithms covered in sections 4.1, 4.3 and 4.4.

- `update_model`: This is the main function of the FDI class. It implements the logic from section 4.1. It is responsible for coordinating all the other functions during execution. This function is given the measurement $\boldsymbol{z}$ from the UAV interface. The innovation of all `model classes` is computed every time a measurement $\boldsymbol{z}$ is received. The function will also executes the Kalman filters associated to the detection models. The function will periodically *update* the detection or isolation models by executing the FDI algorithm. The length of this period is determined by the `update_rate`.
- `runDetection`: This function is responsible for detecting faults in the system. This answers to the algorithm presented in section 4.3.
- `runIsolation`: This function is responsible for isolating faults in the system. This answers to the algorithm presented in section 4.4.
- `state_transition`: This function is responsible for handling the transition between algorithm states. For the current implementation, the only transition is from detection to isolation. For this transitions, the function is responsible for initializing new isolation models when a fault has been detected. This is done by calling the function `add_model`. The function also determine the constant k_s (as covered in 4.4).

The main calculations are done by the following functions:

- `kalmanFilter`: An implementation of the Kalman filter equations as given in section 4.2.1. Note that the function is *general*. It adapts to the *system type*, as specified in the system class. Thus, the function is not rewritten for every new model.
- `likelihoods`: A function that calculates the likelihoods of all models, based their innovations. The function assumes a multivariate Gaussian distribution.
- `bayesFilter`: An implementation of the Bayes filter equation as given in section 4.2.3.

The interaction between the functions can be seen in the two UML sequence diagrams in figure 7.4.1 and 7.4.1. The sequence diagrams follow each other, such that the content of figure 7.4.1 should be understood as the continuation

of figure 7.4.1. The diagrams have been split to ensure readability. However, the partition follows naturally from the structure of `update_detection`. The logic in figure 7.4.1 is executed every time a new measurement is available. Contrarily, the contents of figure 7.4.1 only execute periodically.

`run_detection` and `run_isolation` are not detailed at length here. These functions are the same as what was presented in section 4.

7.4.2 FDI tuning parameters

The main tuning parameters of FDI algorithm must be specified before initialized. Most of these tuning were detailed in section 4. They will therefore just be briefly covered here:

1. `update_rate`: The frequency that the FDI class should run detection and isolation algorithms. Specified as the number of samples.
2. `detection_window`: The length of the detection window L_d. Specified as the number of samples.
3. `isolation_window`: The length of the isolation window L_i. Specified as the number of samples.
4. `S_sensitivity`: The covariance used by the Bayes Filter. This gives the sensitivity. Defined as a square matrix or scalar.
5. `transition_prob`: The transition probability between fault states. This is given by a square matrix.

7.4.3 Initialization and execution

It will now be shown how the FDI algorithm can be initialized and executed. Initialization will be covered first. The main functions involved during initialization is are

- `FDI_class`: The constructor function
- `add_nominal_model`: A function for adding the a fault free model to the `models` container.
- `add_model`: A function for adding new model to the `models` container. This function is also used by the `state_transition` function - after initialization.

`FDI_class` constructor require all the tuning parameters are given as inputs. Further, the constructor needs information about the system class being used. Specifically, the *fault free* model, as specified in subsection 7.2. The constructor will call `add_nominal_model` and pass the fault free *type* and *system model*. An example initialization can then be done as follows:

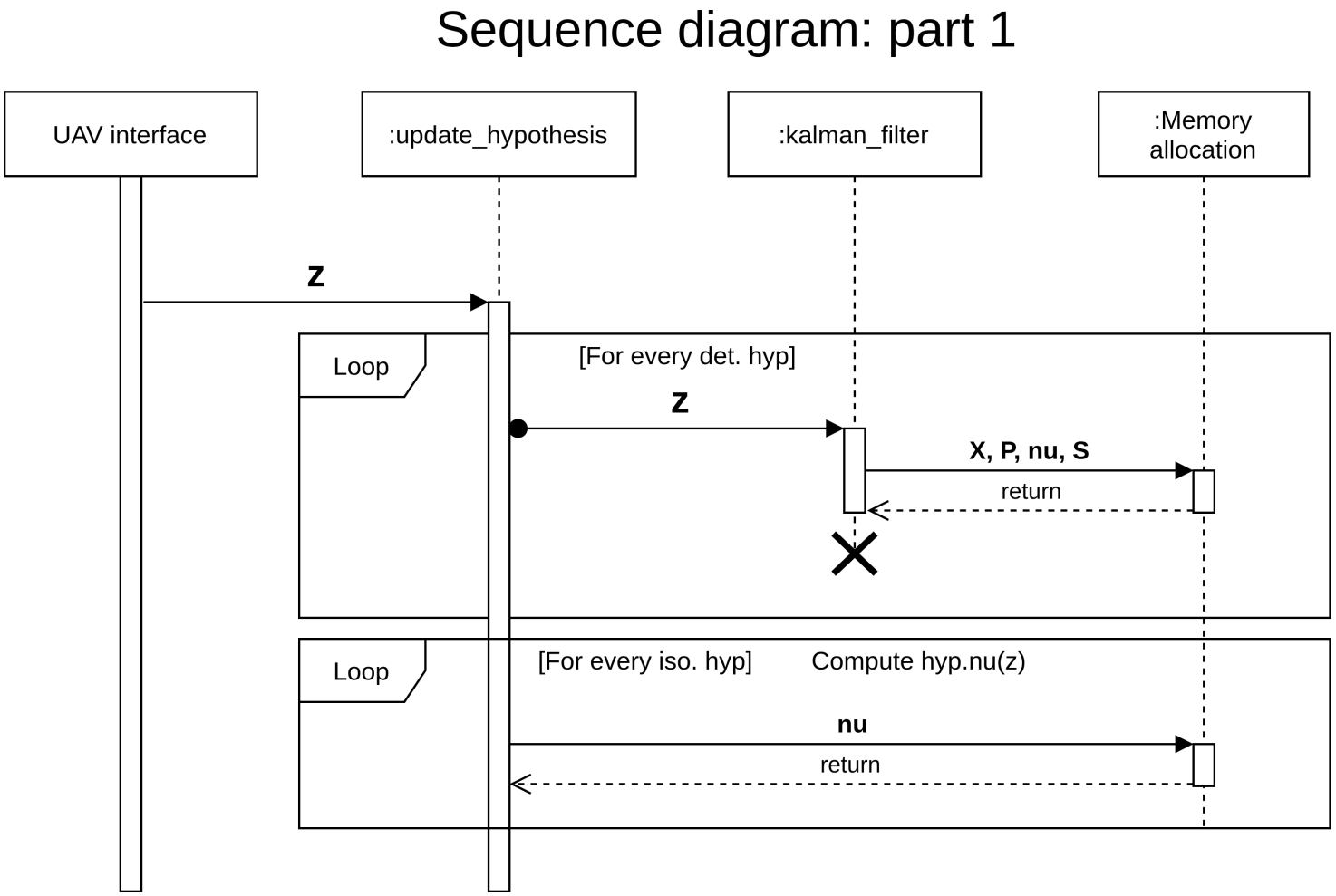

Figure 13: This figure shows a the first part of a UML sequence diagram. The diagram shows the FDI class handle incoming measurements from the UAV interface. The actions are executed every time a measurement is received.

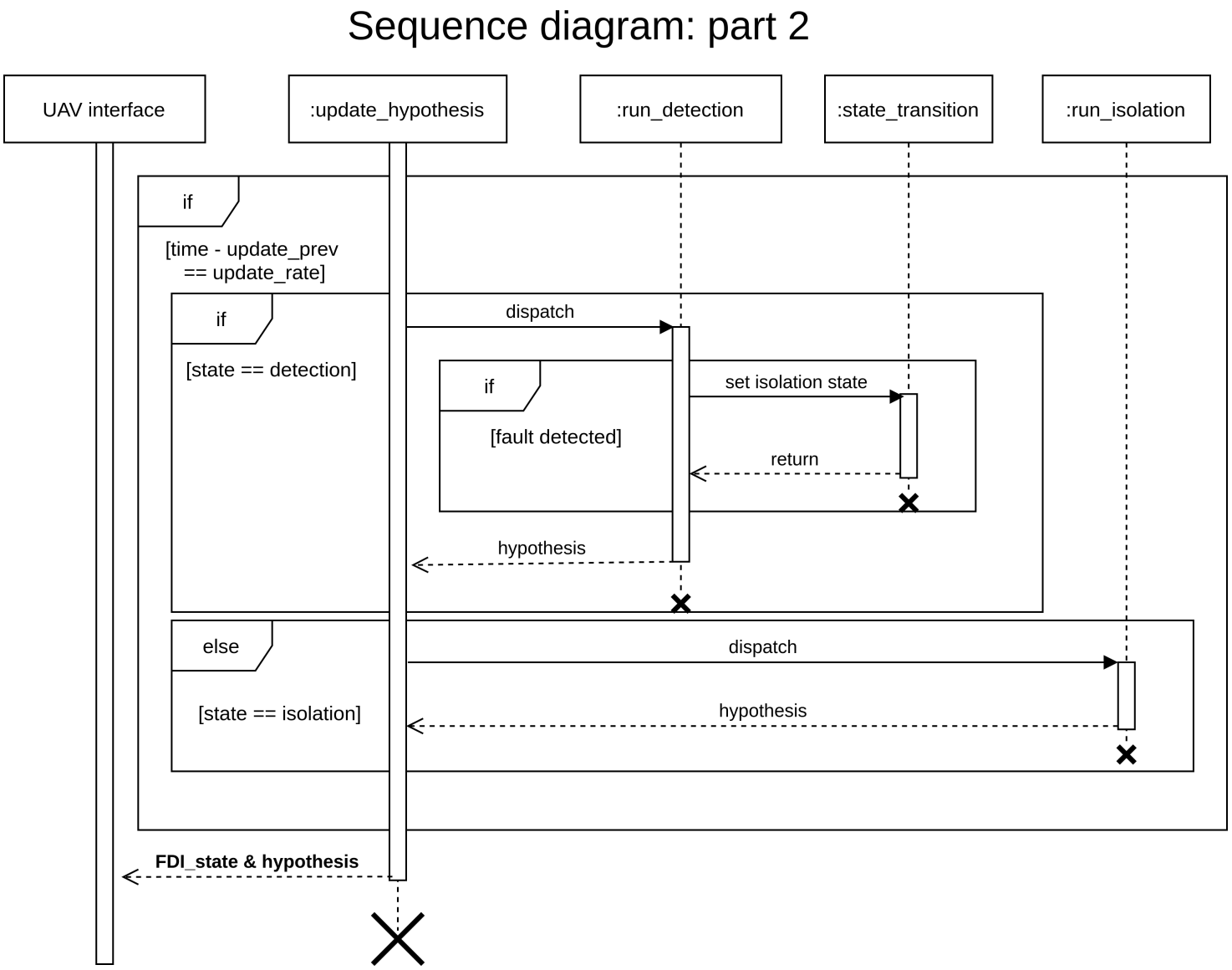

Figure 14: This figure shows the second part a UML sequence diagram. The diagram shows how the FDI class react to a measurement from the UAV interface. The actions are only executed periodically

```
FDI_prop = FDI_class(update_rate, detection_window,
    isolation_window, S_sensitivity, transition_prob,
    propulsion_model.fault_free, mission_length)
```

Listing 14: Initialization of the FDI class. All the FDI tuning parameters are given as constructor inputs.

`FDI_prop` is now an instance of the `FDI_class`. Now, the possible system faults must be added. `add_model` takes care of this. The `model_type` and `system_model` must be specified for each `model`. At initialization, all models will be of type *detection*. As seen in section 7.3, the detection model require the specifications for a Kalman filter - Q, R and P_0. In code, new models are added as displayed bellow.

```
% Icing scalar param
Q = 2e-4; R = 5e-4; P = 1;
FDI_prop.add_model(FaultType.icing_scalar_param, HypothesisType.
    detection, Q, R, P, p_nominal, start_index);

% Air viscosity
Q = 1e-14; R = 3e-11; P = 1e-15;
FDI_prop.add_model(FaultType.air_viscosity, HypothesisType.
    detection,  Q, R, P, phi_nominal(4), start_index);

%  Zero load current
Q = 1e-6; R = 3e-9; P = 5e-4;
FDI_prop.add_model(FaultType.zero_load_current, HypothesisType.
    detection, Q, R, P, phi_nominal(5), start_index);
```

Listing 15: An example of how to add new models to the FDI class - prior to execution.

The system can now be executed in a loop

```
while some_condition
    if UAV.new_measurements_received()
        z = UAV.get_current_measurement();
        FDI_prop.update_model(z);

        if FDI_prop.fault_detected
                UAV.export(FDI_prop.k_s);
        end
    end
end
```

Listing 16: This code exemplify of how the FDI framework can be executed.

Note that there is currently no `UAV` class. As a consequence, all mission data is prerecorded and k_s is set in advance. Thus, the code above is not implemented in its entirety.

7.5 Limitations and drawbacks

The presented software framework has its limitations and drawbacks. This follows naturally from resource constraints and the size the project. This information is vital for future development. Naturally, knowing every bug and

limitation is impossible in advance. However, the following will present the most important drawbacks that are currently understood.

7.5.1 Application for real time systems

The FDI class is optimized for rapid development and prototyping - not for execution speed. Further, the developed software has been used with pre-recorded data and never in a real time environment. Applying the FDI class to real time data is straight forward with the instructions from section 7.4. However, in doing so, the execution times might not be fast enough. This is a direct result of the flexibility of the software. Thus, speed has been traded for flexibility.

For example, the current implementation allows for quick generation of any number of hypotheses. This makes tuning and testing very easy. Oppositely, a real time system should operate with a fixed number of models. This makes it difficult to optimize the the matrix multiplications, e.g. through vectorization. Similarly, using a fixed amount of models would allow for parallelization of the computations. Both vectorization and parallelization would yield high performance gains. However, this is difficult with the current implementation.
The FDI class also stores a lot of data. This data is used for analyzing and plotting the algorithm performance data. A real time system would spend significantly less resources on such storage. Such resources is better spent on increasing execution speeds.
There are some ways to increase the code performance without changing the overall structure. For example by building a minimal version of the current implementation. Another way would be to translate the MATLAB code to some faster language, e.g. C++. However, state of the art solution would require a redesign of the system architecture.

7.5.2 Ardupilot interface

The current MATLAB implementation does not interface with the Ardupilot autopilot. This makes the FDI framework unable to access the UAV motion control. Thus, the airspeed changes, as described in section 4.4, are not executed by the FDI algorithm. Instead this airspeed changes were pre-timed in an external LUA script and executed during simulation. The FDI-class was then executed on using the recorded data. The problem is therefore that the interface around the FDI class is incomplete. For example, the FDI class can determine a propper sampling time k_s, but this cannot be used to control the airspeed. Thus, a communication package must be developed before real time projects can be undertaken.

8 Evaluation Criteria

Specific evaluation criteria has been set to evaluate the algorithm. The *criteria* specify a baseline for what the algorithm must achieve in order to be considered functional.

The evaluation criteria define a binary set of requirements: These are as follows:

1. False Alarm rate should be zero: The algorithm should not conclude that a healthy system has transitioned to a *faulty* state.
2. Fault detection rate should be 100%: The algorithm should always conclude that a faulty system has transitioned to a *faulty* state.
3. Fault Isolation should have 100% accuracy: The algorithm should always isolate the correct fault state of the system.

The presented algorithm is still at low technology readiness level. The presented results is therefore focused on satisfying these criteria using simulations. This allow us to determine proof of concept. Future work should focus on finding the operating range in witch these criteria are met. For example, how much noise the system can be subjected to.

9 Simulation results

The FDI framework has been tested vigorously in the simulation environment. This section will detail the main results from this testing.

Four different simulation scenarios are used to illustrate the performance of the presented fault detection and isolation system, one for each of the fault states listed in Section 3.5. For the given implementation, the system satisfies the criteria we defined in section 8 for all four scenarios.

The results of the detection and isolation are covered separately. However, one should keep in mind that the FDI algorithm is only successful if both steps succeed. In the results we show a series of *probability* plots over time. Such plots can be seen in Figure 11, for example. These plots show the result of the Bayes filter iterating over a window of length L. The graph with the highest value at the end of the window is chosen as the most likely hypothesis. Note that the Bayes filter is generally not run over the entire data set. Instead, it is reinitialized to its priors and executed periodically.

9.1 Detection

It has been found that all fault scenarios can be detected using our method. We also found that false positives can be avoided through good tuning.

9.1.1 No-fault scenario

The algorithm was successful in avoiding false alarms. The algorithm did not show any false alarms in the chosen scenario. Note that this is the result of tuning for this scenario and should be analyzed over a wider range of scenarios.

9.1.2 Propeller icing scenario

The propeller icing $\boldsymbol{x}^{(1)}$ was successfully detected. This can be seen in the bottom row of Figure 15. The plots shows various Bayes Filter outputs throughout the simulation. The columns represent slices in time. Each row shows the probability of a particular Kalman filter. The goal here is for the red line to climb above the blue line (no fault). This signifies a successful detection. It can be seen that the it takes over 100 seconds (after the fault occur) before the error is detected. The slow detection time is in large part due to the slow convergence of the state estimate $\boldsymbol{x}^{(1)}$, as can be seen in the top plot of 15. It can be seen in the top plots of Figure 16 and 17 that $\hat{\boldsymbol{x}}^{(2)}$ and $\hat{\boldsymbol{x}}^{(3)}$ converge much faster. This results in shorter detection times.

An overview of $KF^{(1)}$ can be seen in the top plots of Figure 15. The top plot shows the true parameter and the Kalman filter estimates. The plot also show the Fault free (nominal) parameters. It should be evident that the filter estimates $\hat{\boldsymbol{x}^{(1)}}$ with moderate success. This is a result of the low signal to noise ratio between aerodynamic thrust coefficients and the measurement noise. The middle plot show that the error covariance $\boldsymbol{P}^{(1)}$ stops decreasing after about 300 seconds.

9.1.3 Change in viscous friction scenario

The detection algorithm detected the change in viscose friction $\boldsymbol{x}^{(2)}$ quickly. This can be seen in Figure 16. The top plot shows the dynamics of $\boldsymbol{x}^{(2)}$ and the estimate $\hat{\boldsymbol{x}}^{(2)}$. It can be seen that the estimate $\hat{\boldsymbol{x}}^{(2)}$ responds quickly to the change. This allows the fault to be detected very quickly. The fast detection can be seen in the 3 bottom plots of Figure 16. The Bayes filter becomes very confident 30 seconds after the error occurs.

Note that the estimate $\hat{\boldsymbol{x}}^{(2)}$ looks noisier than the estimate $\hat{\boldsymbol{x}}^{(3)}$ (see figure 17). However, $\boldsymbol{x}^{(2)}$ is many orders of magnitude smaller than $\boldsymbol{x}^{(3)}$. Furthermore, both estimates are subjected to the same measurement noise.

9.1.4 Change in static friction scenario

The change in static friction $\hat{\boldsymbol{x}}$ was successfully detected. The results of the estimation can be seen in Figure 17. Note that both the results and their plots are more or less the same as that of the change in viscous friction in Section 9.1.3.

9.2 Isolation

The isolation algorithm manages to find the correct fault for all fault scenarios. This can be seen in Figure 18. The plot shows the output of the Bayes filter. It can be seen that the algorithm was successful in every case. However, it is clear that the icing fault is the hardest one to isolate. This can be seen in the top plot of Figure 18. Notice that $Y^{(1)}$ only marginally outperforms $Y^{(0)}$. This means that the icing was hard to isolate from the fault free case. The two plots below show $Y^{(2)}$ and $Y^{(3)}$. The Bayes filter was executed 40 seconds after the second air speed change (see Figure 19).

The effectiveness of the isolation algorithm can also be seen in Figure 19. The true fault is given by $H_I = 2$. The left most bottom plot shows that finding the correct hypothesis, prior the identification step, is a challenging problem. In this plot, we see that the wrong hypothesis would have been chosen. One should also note the horizontal green lines in the two plots in the middle. The two horizontal green lines shown in the two plots indicate the parameter values chosen in the static models $Y^{(2)}$ and $Y^{(3)}$. These are generated right before the decrease in airspeed. Observe that the estimate $\hat{\boldsymbol{x}}_k^{(2)} \approx \hat{\boldsymbol{x}}_{k_s}^{(2)}$ and thus remains stable for all $k > k_s$. Contrary to this, it is found that both $\hat{\boldsymbol{x}}_k^{(3)}$ diverges from $\hat{\boldsymbol{x}}_{k_s}^{(3)}$ and $\hat{\boldsymbol{x}}_k^{(1)}$ diverges from $\hat{\boldsymbol{x}}_{k_s}^{(1)}$. It is these divergences that results from the perturbation of V_a that allows the isolation algorithm to find the true fault. The bottom right plot shows how the Bayes Filter confidently isolate returns $H_I = 2$. Note that this scenario, the fault dynamics only gave a 1% increase in $\boldsymbol{x}^{(2)}$.

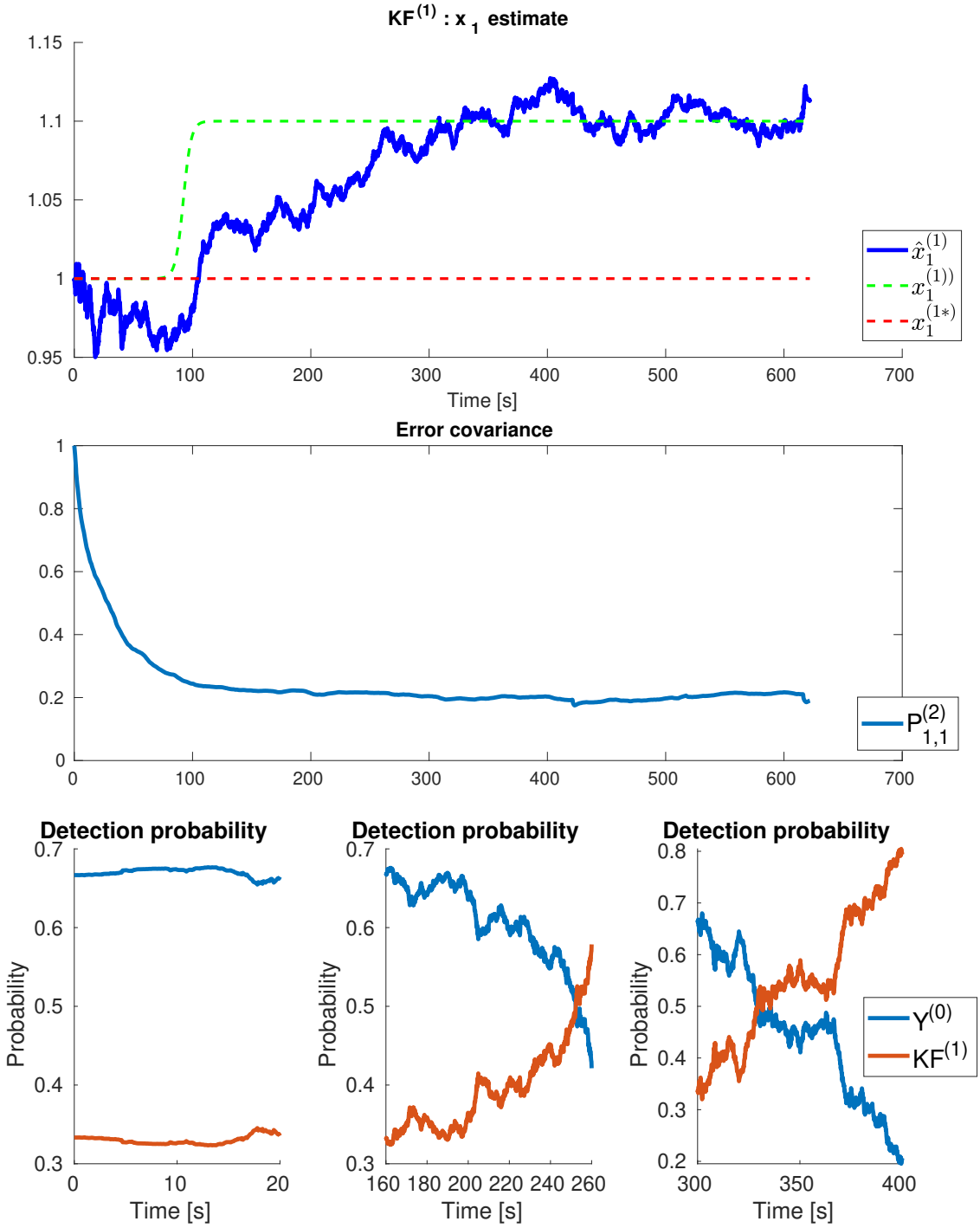

Figure 15: Fault scenario 1: Top plots show state and its estimate. The middle plot show the error covariance $\boldsymbol{P}$. The bottom row display detection probabilities at different slices in time.

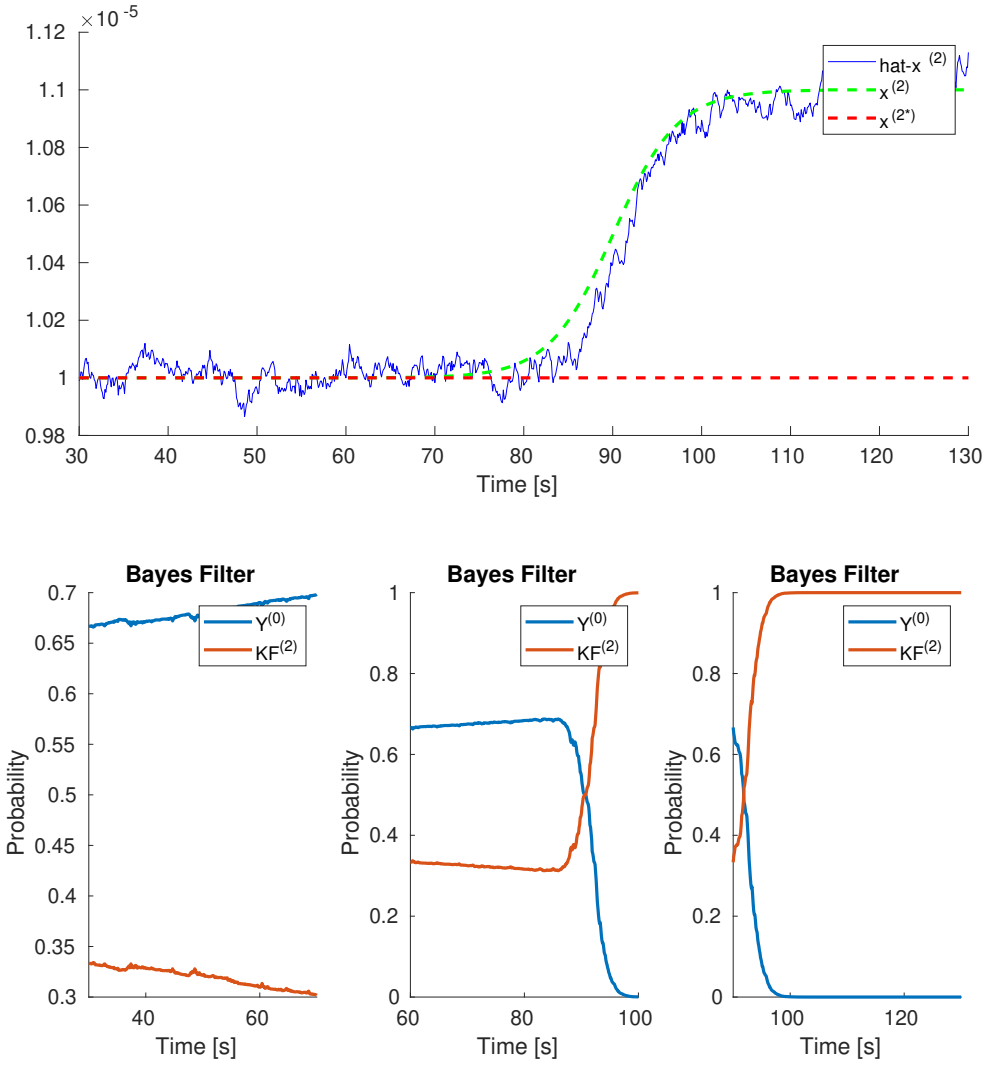

Figure 16: Fault scenario 2. Top: True value and estimate of $\boldsymbol{x}^{(2)}$. Bottom: Probability plots from Bayes Filter at various time instances.

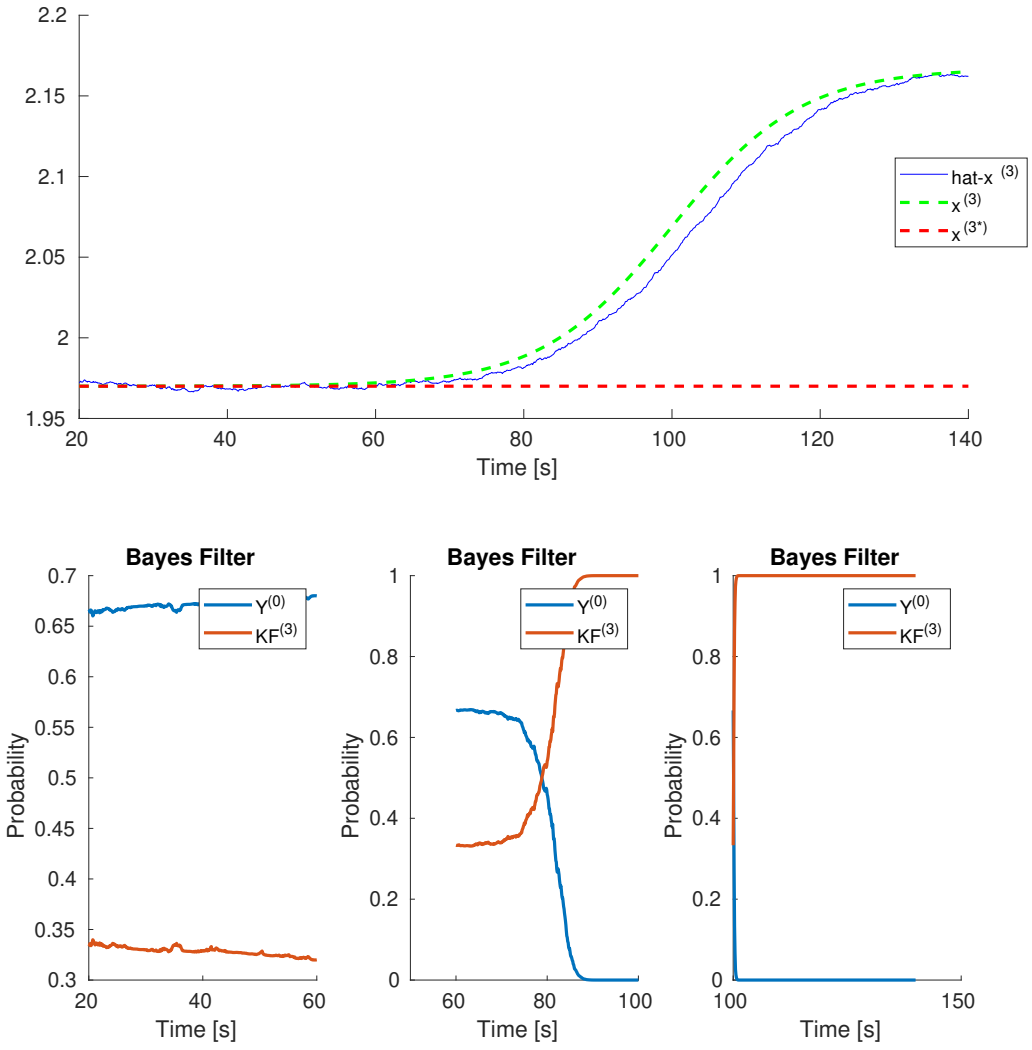

Figure 17: Fault scenario 3. Top: True value and estimate of $\boldsymbol{x}^{(3)}$. Bottom: Probability plots from Bayes Filter at various time instances.

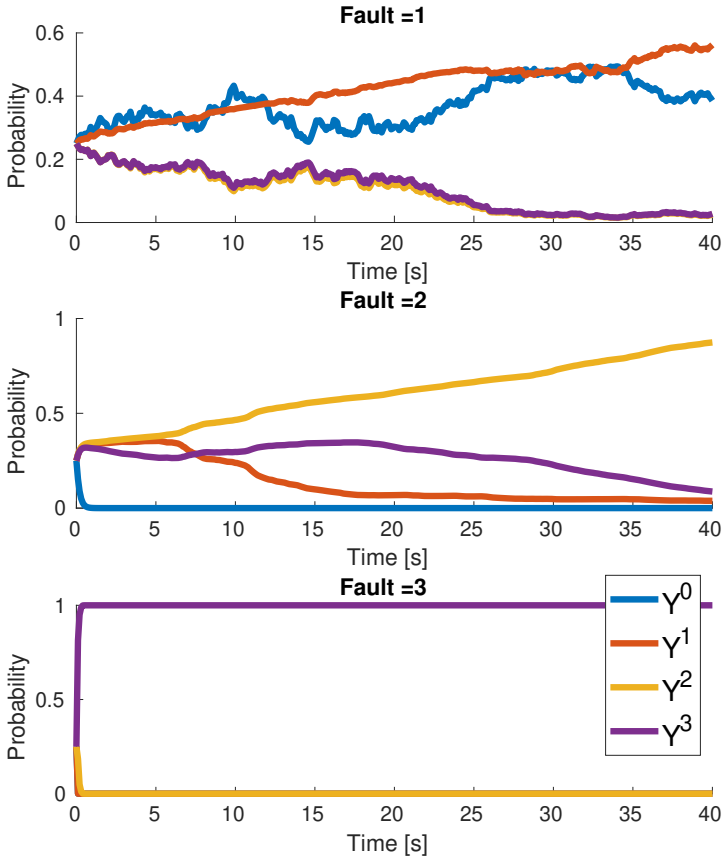

Figure 18: Results of isolation algorithm for all faults. Left: True Fault = 1, Isolation hypothesis = 1. Middle: True Fault = 2, Isolation Hypothesis = 2, Right: True Fault = 3, Isolation Hypothesis = 3

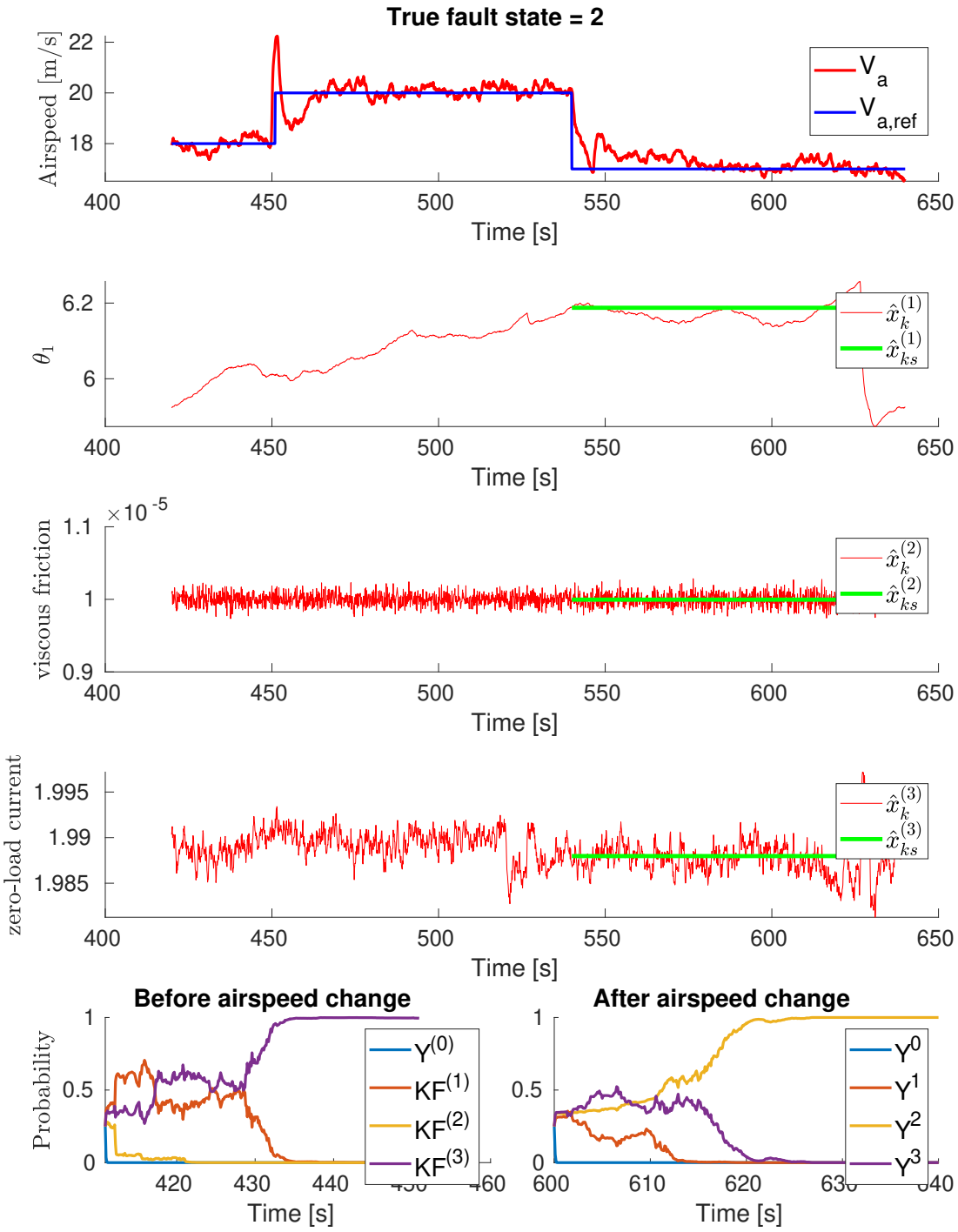

Figure 19: Illustration of the isolation algorithm. The top plot shows the airspeed and its reference. The plots in the middle shows dynamic and static estimates. The bottom plots show outputs of the Bayes filter before and after the static filters are generated.

10 Discussion

This section offers a give a brief discussion of the results. This will include an interpretation of the results and its implications. The limitations of these findings are then discussed. The section ends with recommendations for future work.

10.1 Interpretation

The results warrants a positive interpretation. In every scenario, both the detection and isolation algorithm were successfully. Furthermore, all the evaluation criteria were satisfied.
It is clear that icing was the slowest to detect. However, icing isolation was achieved once it had been detected. Isolating icing from the other fault models was much easier than isolating the icing from the fault free case. This is probably because the fault free case is unaffected by changes in airspeed.

10.2 Implications

The results have two main implications. First, the results is strong validation of the theoretical correctness of the FDI framework. The methodology does inded work as predicted. Second, such results warrants further exploration of the method.

Such exploration can take two forms

1. The method can be tested on real data. This would be straight forward, given that the system dynamics for the X8 fixed wing UAV have been implemented. Further, real wind tunnel data is already available. The only challenge is the to format this data to fit the framework.

2. The method can be applied to new systems. This could include similar systems, such as airfoils, or different systems, such as the wheel of a car.

10.3 Limitations

The results does suggest some limitations. It is quite clear that the method is sensitive to measurment noise. In fact, noise is expected to be the main limitation of the FDI framework. The result strongly suggest that the drifts with the greatest magnitude were the easiest to detect. Thus, upper limits to noise will depend on both the noise and the magnitude on the parameter drift, that is, the signal to noise ratio.
Both of the noise levels and drift magnitude are currently uncertian. The amount of noise in the system will depend on both the measurment equipment and our ability to filter the measurement. The drift magnitudes will hopefully be better understood when a real data set is analyzed.

10.4 Recomendations

Future development on this project should be focused on testing the FDI framework on real data. A substantial part of this work should be devoted to a rigours analysis of noise. These effors should also be directed towards filtering the measurements. Further, the motor current I_e should receive the bulk of this attention. This is because the of I_e has the biggest effect on the system.

11 Conclusion

This thesis set out to detect and isolate icing on a UAV propulsion system. This is a difficult problem. My own project work [36] found that propulsion model has limited obeservability of icing when in cruising speed. The project also found that it is impossible to tell icing apart from other faults while in cruising mode. An initial literature study also found that icing detection on UAV propellers is underdeveloped. Among the problems is the absence of dynamic models of propeller icing accretion. This thesis has presented solutions and workarounds to these challenges. Part of the solution lies in novel system modeling, airspeed control and the introduction of a FDI framework.

It has been shown that a *Random walk* model can substitute for a dynamic icing model. The random walk models has been integrated into a different Kalman filters. These filters has been shown to successfully track many different types of parameter drifts - including icing. The fault dynamics was based on a Sigmoid function. It is fair to assume that other dynamic responses could be tracked as well. Thus, propeller icing can be tracked, without an dynamic icing model. However, a dynamic icing model would still be useful and could potentially improve performance further. The inclusion of such a model would not change the structure of the FDI algorithm.

The propulsion model has limited observability and imposes estimation restrictions - at most 2 dimensions of a state space can be estimated [36]. This thesis shows that such reduction still allows for icing detection. However, this conclusion is based simulating icing using a scalar model. The result might be different if the icing is significantly more complex. However, preliminary findings at NTNU suggest that a 1 parameter model is adequate. If so, then the presented results should be well suited for accurate estimation.

Icing is not the only fault that can occur in the propulsion system. A *multiple models* approach has therefore been used - one model for each fault. Each model is tracked using a separate Kalman filter. The performance of the Kalman filters were compared to a the performance a static fault free model (the null hypothesis). The fault free model outperformed the Kalman filters when no fault had occurred. Furthermore, it was found that the Kalman filters outperformed the static model when a fault occurred. Thus, the detection algorithm was a success for the given set of faults.

In all fault scenarios, *all* the Kalman filters would, after some time, outperform the static model. This is because all models were *excited* at by an error occur. This supports the conclusion that the correct fault is not guaranteed to outperform other models, while in during speed [36]. However, well tuned filters resulted in the correct model having the highest performing Kalman filter. Thus, the correct fault could be identified running the isolation algorithm. However, this results is of limited usefulness - the filters were individually tuned to faults know in advance. The faults cannot be know in advance. Therefore, the filters cannot be expected to be as well tuned. Nevertheless, the result is positive. For example, in some cases there might not be enough time to run the

isolation algorithm.

The isolation step was found to be successful in every scenario. This was the case for all the fault magnitudes that were tested. Furthermore, the isolation step is very good at correcting false positives from the detection step. This allows for the use of a *sensitive* detection algorithm. It is better to allow the detection step "detect" one fault to many, than to not detect anything at all. The isolation steps then serve as a good tool for correcting potential mistakes.

The FDI framework was found to satisfy all the criteria that were introduced in Section 8. Thus, the isolation step never had to correct for false positives.

The biggest drawback of the method is noise sensitivity. The method is highly dependent of good estimates of V_a, ω and I_e. Noisy estimates can disrupt our ability to detect icing. This will be among the primary challenges when working with real data.

In conclusion, the FDI framework must be considered a success. At least from a theoretical stand point. The results are positive. Nevertheless, a healthy dose of skepticism encouraged. In the simulation the propulsion model and the icing dynamics are perfectly described by equation 1 and the One-parameter icing model. Thus, the system implementation and tuning process was based on perfect system knowledge. Naturally, real data will introduce much more model uncertainties. The FDI framework might not perform as well in such circumstances. Consequently, the acquired results might not extrapolate well to real data.

12 Summary

This project introduced a fault detection and isolation (FDI) framework for detecting propeller icing, and other propulsion faults of UAVs. Icing accretion is among the leading sources of UAV system failures.

12.1 Literature Study

This thesis focuses on detecting icing on a propeller UAV. Within the field of aeronautics, *icing* is short hand for *icing accretion* - the formation and accumulation of ice. Icing detection is well studied in aeronautics, but this is not the case for the propellers of small UAVs. Propeller icing is both a difficult and relatively underdeveloped topic. It is therefore natural to look for solutions in related domains.
The closest relation can be found to airfoil icing on UAVs which is a comparably well developed topic. Here, we find many methods for icing detection. Some methods of interest include the use of *multiple models* [32], *observer-based* methods [33], *aerodynamic coefficient estimators* [16] and statistical methods [31]. However, the mechanics of airfoil are comparably simple. In the propeller case, both centrifugal forces and modulated aerodynamic shear forces act on the ice. Tellingly, icing accretion on propellers is currently poorly understood and current research is limited to quantitative analysis. This makes the mechanics difficult and direct comparisons infeasible. Nevertheless, studying published work on airfoil icing is highly useful.

12.2 Modeling

The presented method is based on working with a propulsion model for fixed wing UAVs. The model relates the propeller angular acceleration $\dot{\omega}$ to variables such as airspeed V_a, motor electric current I_e, rotational speed $\dot{\omega}$ and the aerodynamic torque Q_a. Aerodynamic torque will increase as icing accrete. This offers the possibility to detect icing by only measuring ω, V_a and I_e.

Propeller icing and its effect on the aerodynamic torque are open research questions. The chosen approach is to model the icing as *random walk* processes. Further, aerodynamic torque could in theory be a high degree polynomial, as a function of icing. This method demands a low order system due to limited observability.

Therefore, many models were available to model it. The best candidates are reduced state space model and a scalar parameterization. The majority of the text assumes that the one parameter model is used. The reason for this is two fold. Firstly, this seems to fit well with preliminary results from experimental data. Secondly, a scalar model yields a very simple system. The other faults are much easier to model and are therefore given less attention.

A fault occurs whenever a parameter drifts from its nominal value. Many such errors can occur in a system. This framework uses a *multiple models*

approach to model different faults. The goal is to determine which model out of the many is correct. In each model, the associated parameter represents the state. All other parameters are kept constant. The exception is the fault free model. This model holds all parameters constant.

The framework assumes that only one model can be true at a time. Thus, multiple models can be referred to as the *states* of a state machine. In this machine, the current state will only depend on the previous state. Further, transitions between states is a statistical question. This results in a *Markov process*.

The prior probability that a fault will occur can now be considered as the *transition* probability in the state machine. The goal is then to predict when the Markov process transitions.

The model can be represented as state space systems. The process model of the state space is then given by the random walk model. The measurement model of the system is derived from the propulsion model. This allows for state observers to estimate the state. The state space can then be estimated using Kalman filters. The state of model i is denoted by $\hat{x}^{(i)}$. Generally, $\cdot^{(i)}$ denotes any variable associated to model i. For example, the Kalman filter of model i is denoted as $KF^{(i)}$.

It is important to realize that every model is based on the same set of system equations. In this case, all models use the same propulsion model to derive the measurement equation, $y_k^{(i)} = C_k^{(i)}\hat{x}_k^{(i)} + \textit{static terms}$. For model i the *static terms* are the terms that do not depend on $x^{(i)}$.

12.3 The Fault Detection and Isolation Framework

The FDI framework is partitioned into two separate steps, *detection* and *isolation*. The *detection* step is the default algorithm. In this step, the goal is limited to detecting that a fault has occurred. This is done while the UAV is operating in cruise mode, i.e., at a constant speed. If a fault is detected, then the algorithm will issue two commands concurrently. **1.** The UAV will be requested to make a series of jumps in the airspeed. **2.** The isolation will be activated. The isolation algorithm aims at determining which fault that has occurred. The jump in airspeed is necessary in order to tell the different faults apart.

12.3.1 Estimation

Both steps of the isolation and the detection algorithms depend on state observers for states estimation. In this case, Kalman filters are used. Each Kalman filter maps to exactly one model. Thus, if a models assumes that some parameter α is varying with time, then a Kalman filter will estimate α. The filters are tuned based on the characteristics of the parameters they are estimating. For

example, if a parameter tends to have large drifts, then the filter process noise covariance Q must be large as well. Thus, different Kalman filters will be tuned differently.
The filter performance is determined by the innovation $\nu_k^{(i)} = y_k^{(i)} - C_k^{(i)}\hat{x}_k^{(i)} +$ *static terms*. The innovation gives the difference between the measurement and the predicted measurement.
Not all models require a Kalman filter. The fault free model assumes that parameters are fixed and thus do not require estimation. This motivates the introduction of *static hypotheses models*. Static hypotheses models are used to represent models that assume all parameters to be static. Static hypothesis are initialized according to a set of parameters and these do not change. A static model will still return an innovation: $\nu_k^{(i)} = y_k^{(i)} -$ *static terms*. Thus, dynamic and static models can be compared by their innovations. The model with the smallest innovation will tend to represent the best hypothesis.

Both the detection and isolation algorithms use a *Bayes filter* to chose between competing hypotheses. The detection step aims to find the detection hypothesis $H_D \in \{\text{healthy, faulty}\}$. Similarly, the isolation step aims to find the isolation hypothesis $H_I \in \{\text{no fault, icing, ...}\}$.

Using a Bayesian framework follows naturally from the probabilistic nature of the problem. Furthermore, using Bayes reasoning allows us get the most out of the available information. From before, the Markov process defines a set of transition probabilities. This corresponds to prior probabilities. Further, the conditional model probability, given the measurements, can be found based on the Kalman filter performance.

The *Bayes filter* always initializes with a prior and receives innovations from a set of models. The filter then calculates which model has the highest probability based on Bayes rule. The filter will terminate after receiving a specified number of innovation samples. This number is referred to as the *window*.

12.3.2 The Detection Algorithm

The detection algorithm is aimed at answering a binary question: Is the system faulty? This is formulated as the detection hypothesis $H_D \in \{healthy, faulthy\}$. In this case, the number of models will not equal the possible detection outcomes. Instead, one model is generated for each possible fault. These models have associated state spaces and Kalman filters. A static model is spawned for the fault free scenario.

A fault is detected if any of fault models become statistically more probable than the fault free model. This comparison is done using a set of Bayes filters. The i^{th} Bayes filter, $BF^{(i)}$ takes the innovations from the fault free model $\nu^{(0)}$ and the i^{th} Kalman filter $\nu^{(i)} \leftarrow KF^{(i)}$. This gives the i^{th} hypothesis $H_i = BF^{(i)}(\nu^{(0)}, \nu^{(1)}) \in \{0, 1\}$. The detection hypothesis can then be determined by a Boolean summation:

$$H_D = \mathcal{H}_1 \vee \mathcal{H}_2 \vee \mathcal{H}_3$$

12.3.3 Isolation

The isolation algorithm is aimed at answering the question: What is the system fault? This is formulated as the isolation hypothesis $H_I \in \{\textit{fault free}, \textit{icing} \text{etc...}\}$. In this case, the number of models equals the possible isolation outcomes. The isolation algorithm will only execute if a fault has been detected. The FDI framework will also command the airplane to execute a set of airspeed jumps. These jumps are designed to make the faults easier to isolate.

The isolation algorithm assumes that all models are static. The static models initialize according to the estimates from the existing Kalman filters. These are Kalman filters that were used by the dynamic models in the detection step. Naturally, the inheritance occurs between models of the same type. For example, the static model for icing will inherit its estimate from the model that estimated icing. The isolation algorithm also includes the fault free model from before.

The probability of the different models can then be compared. All models are compared using *one* Bayes filter. The filter takes in the innovation from all the models and returns the isolation hypothesis directly. The propulsion model has 3 possible faults and a fault free scenario. This gives four models. The detection hypothesis can then be found directly using:

$$H_I = BF^{(i)}(\nu^{(0)}, \nu^{(1)}, \nu^{(2)}, \nu^{(3)})$$

12.4 Simulation Environment

The simulation of a propulsion system is distributed over two independent systems - Ardupilot and Simulink. Ardupilot manages the UAV control using a software-in-the-loop framework. Simulink is responsible for simulating the physical environment. This includes the Aerodynamic properties of the UAV.
Faults are also simulated within Simulink. This is done by altering the parameters of the propulsion model.
Measurement noise is added to the recorded data. The added noise was Gaussian.

12.5 Tuning

The FDI framework is dependent on a rigours tuning process. This amounts to tuning Kalman filters and Bayes filters.

12.5.1 Kalman Filter Tuning

Well-tuned Kalman filters are essential for achieving good results. This process is the same as that of tuning any other Kalman filters. However, 3 points are important for the present case:

1. The process noise covariance $\boldsymbol{Q}_k^{(i)}$ of Kalman filter $\boldsymbol{KF}^{(i)}$ must be updated to the expected magnitude of fault i. The filters will therefore tend to be differently tuned.

2. The estimate response time is often more important than the quality. This is because faults most often can be detected quickly.

3. The goal is not about achieving a perfect estimate. The important point is that the correct filter outperforms all other filters.

12.6 Bayes Filter Tuning

The Bayes filter has the following tuning parameters:

1. $\boldsymbol{S}_k^{(i)}$: The innovation variance of the Bayes filter (which determines the sensitivity). This parameter will determine how sensitive the filter is to differences in the model innovation. Let us consider two models with the same prior probability. For a given time step, model i has a very small innovation $\nu^{(i)} \approx 0$ and j has a large innovation $\nu^{(j)} >> 0$. If the filter is sensitive, then the Bayes filter will update the probabilities such that model i has a much larger probability. Contrarily, an insensitive filter will barely update the probabilities. The sensitivity must be sufficiently high for the models to converge in time. At the same time, too high sensitivity makes the filter vulnerable to measurement outliers.

2. Π^f and Π^d: The Markov matrices (which determine the prior probability distributions) of the Markov processes.

3. L: The window length (which determines how many samples the filter should use). The window must be long enough for the Bayes filter to converge to the true model. However, the window should be much longer than the response times of the faulty parameters. For example, if ice accretion builds up the time scale of minutes, then an hour long window would catch the fault too slowly.

12.7 Implementation

The FDI framework has been implemented in MATLAB. It is written to be general and highly flexible. This allows for quick development and experimentation. The two main classes of the system are:

1. The system class: This class implements the system specific equations. For example, for the current project, this class contains the propulsion model. The class also contains the state space formulations of the different models. New systems can be added by following the template in 7.2.6.

2. The model class: This class implements the different models. The class contains all the data that a model needs to function in the FDI framework. The class receives a state space from the system class. It also inherits many properties based on the sub-types `detection, isolation, fault free`. For example, a detection model stores the specifications for tuning a Kalman filter.

3. The FDI class: This class implements the FDI framework. This includes the detection algorithm, isolation algorithm, the Kalman filter and the Bayes filter.

The code implementation comes with two main drawbacks:

1. The code is not optimized for speed. This might be problem when working with real time systems.

2. Real time execution demands that the FDI framework communicates with the internal control unit of the UAV. However, the required communication package has not been developed.

12.8 Simulation Results

The FDI framework is found to work satisfactory for all the simulated scenarios. The faults in these scenarios are 10% of the nominal parameter values. These results are found while applying *moderate* noise levels.

12.9 Detection

The detection algorithm function is satisfactory in all scenarios. When no fault is present, the algorithm avoids false positives. Further, a fault is detected in all fault scenarios.
The icing scenario takes the longest to detect. This is because the underlying parameters have the smallest magnitude. A 10% drift in these parameters is therefore smaller than a change in other parameters.

12.10 Isolation

The isolation algorithm function is satisfactory in all scenarios. The algorithm is capable of isolating the true fault in every scenario.

12.11 Conclusion

All evaluation criteria for the FDI framework were met. The framework is ready for testing on real data sets.

References

[1] Yang Liu, Linkai Li, Zhe Ning, Wei Tian and Hui Hu, Experimental Investigation on the Dynamic Icing Process over a Rotating Propeller Model *J. Propulsion and Power*, Vol. 34, pp. 933–946, 2018

[2] Yang Liu, Linkai Li, Wenli Chen, Wei Tian, Hui Hu, An experimental study on the aerodynamic performance degradation of a UAS propeller model induced by ice accretion process, *Experimental Thermal and Fluid Science*, vol. 102, pp. 101-112, 2019

[3] N. Müller, R. Hann, T. Lutz, The Influence of Meteorological Conditions on the Icing Performance Penalties on a UAV Propeller, Deutscher Luft- und Raumfahrtkongress (DLRK), 2020

[4] R. Hann, T. A. Johansen, UAV Icing: The Influence of Airspeed and Chord Length on Performance Degradation, *Aircraft Engineering and Aerospace Technology*, 2021

[5] B. C. Bernstein, C. A. Wolff, F. McDonough, An Inferred Climatology of Icing Conditions Aloft, Including Supercooled Large Drops. Part I, *Journal of Applied Meteorology and Climatology*, vol. 46, pp. 1857–1878, 2007

[6] A. Winter, R. Hann, A. Wenz, K. Gryte, T. A. Johansen, Stability of a Flying Wing UAV in Icing Conditions, 8th European Conference for Aeronautics and Space Sciences (EUCASS), Madrid, 2019

[7] K. Gryte, R. Hann, M. Alam, J. Rohác, T. A. Johansen, T. I. Fossen, Aerodynamic modeling of the Skywalker X8 Fixed-Wing Unmanned Aerial Vehicle, International Conference on Unmanned Aircraft Systems, Dallas, 2018

[8] G. Ducard, H. P. Geering, Efficient Nonlinear Actuator Fault Detection and Isolation System for Unmanned Aerial Vehicles, *J. Guidance, Control and Dynamics*, Vol. 31, pp.225-237, 2008

[9] E. Baskaya, M. Bronz, D. Delahaye, Fault detection and diagnosis for small UAVs via machine learning, IEEE/AIAA 36th Digital Avionics Systems Conference (DASC), St. Petersburg, FL, 2017

[10] P. Freeman, R. Pandita, N. Srivastava and G. J. Balas, Model-Based and Data-Driven Fault Detection Performance for a Small UAV, *IEEE/ASME Transactions on Mechatronics*, vol. 18, no. 4, pp. 1300-1309, 2013

[11] H. M. Odendaal, T. Jones, Actuator fault detection and isolation: An optimised parity space approach, *Control Engineering Practice*, Vol. 26, pp. 222-232, 2014

[12] A. Hasan, T. A. Johansen, Model-Based Actuator Fault Diagnosis in Multirotor UAVs, International Conference on Unmanned Aircraft Systems, Dallas, 2018

[13] A. Cristofaro, A. P. Aguiar, T. A. Johansen, Icing Detection and Identification for Unmanned Aerial Vehicles using Adaptive Nested Multiple Models, *International Journal of Adaptive Control and Signal Processing*, Vol. 31, pp. 1584–1607, 2017

[14] M. M. Seron, T. A. Johansen, J. De Dona, A. Cristofaro, Detection and Estimation of Icing in Unmanned Aerial Vehicles using a Bank of Unknown Input Observers, Australian Control Conference, 2015

[15] K. L. Sørensen, M. Blanke, T. A. Johansen, Diagnosis of Wing Icing Through Lift and Drag Coefficient Change Detection for Small Unmanned Aircraft, Proc. 9th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes, Paris, 2015

[16] A. W. Wenz, T. A. Johansen, Icing Detection for Small Fixed Wing UAVs using Inflight Aerodynamic Coefficient Estimation, IEEE Aerospace Conference, Big Sky, Paper 2636, 2019

[17] E. M. L. Coates, A. W. Wenz, K. Gryte, T. A. Johansen, Propulsion System Modeling for Small Fixed Wing UAVs, International Conference on Unmanned Aircraft Systems (ICUAS), Atlanta, 2019

[18] S. Särkkä, Bayesian Filtering and Smooting. Cambridge University Press, 2013

[19] S. Hansen, M. Blanke, J. Adrian, A Framework for Diagnosis of Critical Faults in Unmanned Aerial Vehicles, IFAC Proceedings Volumes (IFAC World Congress), Vol. 47, pp 10555-10561, 2014

[20] S. Hansen and M. Blanke, Diagnosis of Airspeed Measurement Faults for Unmanned Aerial Vehicles, *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, pp. 224-239, 2014

[21] T. A. Johansen, A. Cristofaro, K. L. Sørensen, J. M. Hansen, T. I. Fossen, On estimation of wind velocity, angle-of-attack and sideslip angle of small UAVs using standard sensors, International Conference on Unmanned Aircraft Systems, Denver, 2015

[22] A. W. Wenz, T. A. Johansen, Moving Horizon Estimation of Air Data Parameters for UAVs, *IEEE Transactions on Aerospace and Electronic Systems*, vol. 56, pp. 2101-2121, 2020

[23] W. Chaer, R. Bishop, J. Ghosh, A mixture-of-experts framework for adaptive Kalman filtering, *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics*, vol. 27. pp. 452-64, 1997

[24] E. F. L. Narum, R. Hann, T. A. Johansen, Optimal Mission Planning for Fixed-Wing UAVs with Electro-Thermal Icing Protection and Hybrid-Electric Power Systems, Int. Conf. Unmanned Aircraft Systems, Athens, 2020

[25] Melody, J. Hillbrand, T. Perkins, W., "Hβ Parameter identification for inflight detection of aircraft icing", December 2001, Control Engineering Practice, 9(12):1327-1335

[26] Yang. Liu, Linkai. Li, Zhe. Ning, Wei. Tian, Hui. Hu, "Experimental Investigation on the Dynamic Icing Process over a Rotating Propeller Model", Journal of Propulsion and Power, 2018.

[27] Yang. Liu, Linkai, Li. Hui. Hu, "An experimental study of surface wettability effects on dynamic ice accretion process over an UAS propeller model", Aerospace Science and Technology, 2017.

[28] ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT, "Ice Accretion Simulation", AGARD ADVISORY REPORT 344, 1997.

[29] R. Hann, "Atmospheric Ice Accretions, Aerodynamic Icing Penalties, and Ice Protection Systems on Unmanned Aerial Vehicles", Doctoral theses at NTNU, 2020.

[30] Bragg, M., Gregorek, G., and Lee, J., "Airfoil Aerodynamics in Icing Conditions," Journal of Aircraft, 1986

[31] D. Ding, W. Q. Qian, Q. Wang, "Aircraft Inflight Icing Detection Based on Statistical Decision Theory"

[32] P. Lu, L. Van Eykeren, E. Van Kampen, QP. Chu. "Selective-reinitialization multiple-model adaptive estimation for fault detection and diagnosis". Journal of Guidance, Control, and Dynamics 2015.

[33] P. Kabore, H. Wang, "Design of fault diagnosis filters and fault-tolerant control for a class of nonlinear systems" IEEE Transactions on Automatic Control 20.

[34] D. Rotondo, A. Cristofaro, T. A. Johansen, F. Nejjari, V. Puig, "Robust fault and icing diagnosis in unmanned aerial vehicles using LPV interval observers", Int. journal of robust and nonlinear control, 2018.

[35] Han, Y., Palacios, J., and Schmitz, S., "Scaled Ice Accretion Experiments on a Rotating Wind Turbine Blade," Journal of Wind Engineering and Industrial Aerodynamics, Vol. 109, Oct. 2012

[36] O. M. Haaland, "Propeller icing detection: Project work", NTNU, 2020.