Jacob H. Dahl

Model-based control of UAV in icing conditions

Masteroppgave i Kybernetikk og Robotikk
Veileder: Morten D. Pedersen
Januar 2023

Jacob H. Dahl

Model-based control of UAV in icing conditions

Masteroppgave i Kybernetikk og Robotikk
Veileder: Morten D. Pedersen
Januar 2023

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Model-based control of UAV in icing conditions

Author
Jacob H. Dahl

Supervisor
Morten D. Pedersen

January 16, 2023

Department of Engineering Cybernetics
NTNU

Abstract

With the exponential increase in computer power over the last decades, data-driven algorithms have become more and more common in different applications. This includes control systems. Model-based control systems require a model of the plants they are intended to control. Traditionally, this was done using the laws of physics and mathematically modelling, but data-driven methods allow for the use of data to find the desired model instead.

This thesis implements a Sparse Identification of Nonlinear Dynamics (SINDy) estimator. With the goal of applying the estimated model on a dynamic inversion controller to control the pitch rate of an Unmanned aerial vehicle (UAV). This model is then evaluated in simulation of an aircraft under three different scenarios: clean airfoil, fully iced airfoil, and the transition between them.

Sammendrag

Den eksponensielle økningen av datakraft over de siste tiår har gjort datadrevne algoritmer mer og mer vanlig i et mangfold av applikasjoner, inkludert kontrollsystemer. Modellbaserte kontrollsystemer anvender en modell av systemet de er ment å kontrollere. Tradisjonelt er dette gjort ved modellering via fysikkens lover of matematisk modelleringsprinsipper, men datadrevne metoder tillater bruk av data til å finne ønskede modeller i steden.

Denne oppgaven implementerer en SINDy estimator med mål om å bruke estimatoren på en dynamic-inversion-kontroller for å kontrollere stampraten (pitch rate) til en UAV. Modellen er så evaluert i simulering av et luftfartøy i tre forskjellige scenarioer: ren aerofoil, fullt iset aerofoil, og overgangen mellom dem.

Preface

This thesis is the result of five years at the Norwegian University of Science and Technology (NTNU). Two of my years at NTNU were spent in the organization Revolve NTNU. It was an amazing experience I would not be without, and a big thanks goes out to them, their sponsors, and all team members from 2019 and 2020.

I would also like to thank my supervisor Morten Dinhoff Pedersen for guidance in the project, and Bendik Holm, Viktor Korsnes, Anders Fagerli, and Jon Nordby for proof-reading and tips.

Contents

List of Tables

List of Figures

Acronyms

3-DOF Three-degrees-of-freedom. 11

6-DOF Six-degrees-of-freedom. 8

CFD Computational Fluid Dynamics. 2, 17, 36

ECEF Earth Centered Earth Fixed coordinate system. 9

LASSO Least Absolute Shrinkage and Selection Operator. 29

MPC Model Predictive Control. 67

MSE Mean Squared Error. 25, 40, 49, 52, 55, 63

NACA National Advisory Committee for Aeronautics. 18

NED North East Down coordinate system. 7–9, 11

PID Proportional Integral Derivative. 3, 20, 43, 58, 59, 63, 69–71

ROS Robot Operating System. 29, 30, 39

SINDy Sparse Identification of Nonlinear Dynamics. i, ii, 4, 26, 27, 29, 31, 33, 34, 36, 39, 40, 42–50, 52–58, 60–63, 65–72

SR3 Sparse Relaxed Regularized Regression. 29, 40

STLSQ Sequentially-Thresholded Least-Squares algorithm. 29, 40, 65, 66

UAV Unmanned aerial vehicle. i, ii, 1–3, 5–8, 30, 32, 54, 58, 65, 66, 68, 70, 71

Chapter 1

Introduction

1.1 Motivation and problem description

Unmanned aerial vehicle (UAV) icing has recently emerged as a research topic because of its limiting factor on the operation of airborne vehicles [2]. UAV's have previously seen most use in military operations, but recently, more commercial and research applications have become prominent [3]. For package delivery, as an example, the aircraft needs to be able to operate during long stints at a time during a wide variety of weather conditions without any significant performance loss. Atmospheric icing is the phenomenon where sub-zero temperature water droplets are present in the atmosphere. When a UAV enters such an area of the atmosphere, icing of the airfoils can occur [4][5][6]. When icing occurs, the dynamics of the aircraft will change. The icing of the airfoils will change their geometry, reduce lift, and increase drag [6]. When this happens, a sound control system is a system that handles the change in dynamics and manages to operate normally.

The modelling of icing is still a topic under research, and to give an accurate aerodynamic model of any UAV requires extensive testing or Computational Fluid Dynamics analysis [7]. Icing makes modelling more difficult. To accurately predict the dynamics during icing, a model of how ice accumulates is necessary. In addition, specialized CFD analysis tools are needed to get meaningful models of icing, [2]. When accurate empirical models are challenging to find, a solution can be data-driven modelling.

Data-driven modelling has the goal of giving a model of a system based on data points of the input and output of the system. There are many techniques which achieve this goal, but a downside to data models is the tradeoff between complexity and accuracy [8]. To get accurate models from standard data-driven, we usually need a complex model. The more complex a model is, the difficulty in explaining what exactly the model learns increases. This can also lead to what is known as ”overfitting” the model [8]. This is especially undesirable in a safety-critical control system on an aircraft because a change in atmospheric disturbances, icing, or noisy measurements can completely alter the output of the estimated model. When applying a model to a dynamic inversion controller, it needs to be invertible and should be a representation of the underlying dynamics so it is able to handle input data which is taken out of the learned distribution.

We already know models which describe the dynamics of certain UAV’s like the ”Aerosonde” presented in Beard and McLain [7]. However, this model is only valid for this specific aircraft. If we wanted to apply the same model-based control system for a different aircraft, we would need to do the modelling work again for it specifically. Data-driven modelling can reduce the workload of the modelling, but testing would still be required to find an accurate representative model of the aircraft. Hence, a desirable UAV data-driven model estimating system will be able to:

- Find an accurate representation of the underlying dynamics

- Have the found model be able to handle out-of-distribution input
- Do this online during flight to reduce the amount of testing needed
- Invert the estimated model

The goal of this thesis is to use a data-driven method for model discovery of a UAV system which fulfils these four requirements. The resulting system will be tested in a self-made simulator which simulates a UAV during icing. It will be compared to a model-error estimating neural network controller [1], and a non-model based successive loop closure Proportional Integral Derivative controller [7].

1.2 Thesis outline

The structure of the thesis is as follows. Chapter two will go through important theory topics for the simulator and control systems. Chapter three will be on implementation steps needed to make the algorithm work, and a high-level description of how the algorithm is implemented in code. Chapter four will present the results, and chapter five will contain a discussion, conclusion and further work.

1.3 Literature study

Dynamic inversion is a commonly used control scheme for aircraft control. Adaptive dynamic inversion methods using neural networks are explored in Calise [9], Lakshmikanth et al. [10], Liu [11], and Johnson [12], and were used as a basis for the controller developed in Dahl [1] and used here.

For accurate dynamic inversion control, a accurate model is required. Modern system identification algorithms include Dynamic mode decomposition [13]. It has been applied to control problems in Proctor et al. [14] with high.dimensional data. Juang et al. [15] implements system identification via an observer with computed Markov parameters. Juang and Pappa [16] implements a model reduction method to reduce higher dimensional data to a lower order for estimation. All these techniques are meant for high-dimensional data, but in our case, the data is in low dimensions. For this a different algorithm is preferred.

SINDy is a relatively new algorithm developed in Brunton [17]. It has been applied to control problems in Morrison and Kutz [18], Kaiser et al. [19], Fasel et al. [20], and Brunton et al. [21]. Morrison and Kutz [18] implements control on low-dimensional biological systems, and Hopfield memory networks. Kaiser et al. [19] implements a model predictive controller for different systems using SINDy with low amounts of data available. Fasel et al. [20], and Brunton et al. [21] are about control in general with no specific application being the focus, but Brunton et al. [21] uses a pitch controller for an aircraft as an example of an application for SINDy with a model predictive control scheme. No research has been found on the application of SINDy on a dynamic inversion controller.

Since SINDy is optimal for low-dimensional data, has been applied to control before, but not with a dynamic inversion scheme, this method was chosen as the system identification method. SINDy is also easily available to implement the Python package Pysindy [22].

Chapter 2

Theory

2.1 Introduction

In this chapter, the theory behind the development of the simulator and the control systems is presented. The chapter also includes some information on some useful tools that were used in the development.

2.2 UAV

The UAV parameters used for simulation and testing of the control systems developed is the Aerosonde UAV. The Aerosonde is an UAV developed by Textron systems. It is primarily used for atmospheric research, but has also been used in border patrol and wildlife monitoring [23], [24]. The Aerosonde is a long-endurance UAV, capable of flying for up to 24 hours at a time, and can fly at altitudes up to 6000 meters. It is equipped with a variety of sensors, including weather radar, and can be controlled remotely or programmed to fly autonomously using GPS navigation. The Aerosonde is known for its reliability, versatility, and ability to operate in harsh en-

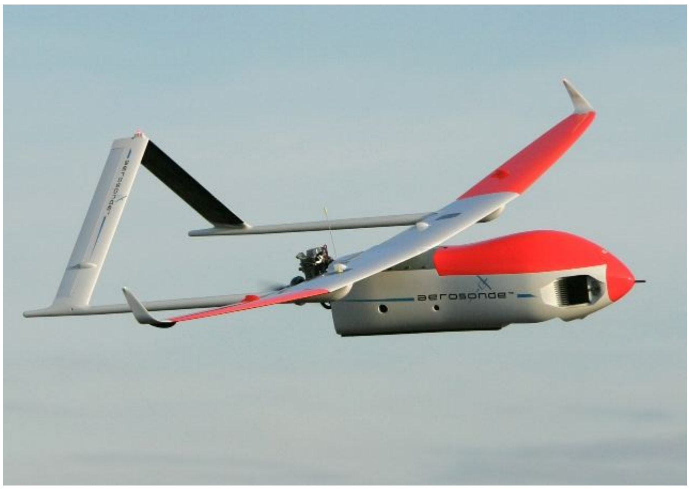
Figure 2.1: Aerosonde UAV [26].

vironments [25]. In 2.1 we can see an image of the Aerosonde during a mission.

2.3 Control systems

Control systems in general are necessary when flying an aircraft since they allow the craft to remain stable. The control system receives inputs from sensors, like accelerometers, gyroscopes, GPS, barometers, and so on, and uses the information to adjust the control surfaces of the aircraft to achieve the desired state (position, velocity, pitch angle, etc.). In addition to control surfaces, UAV's need a system to generate thrust.fulfills

The common control surfaces available in an aircraft are rudders, flaps, ailerons and elevators, as can be seen in 2.2. The Aerosonde does not have conventional rudders or elevators like the example in 2.2. It has the ailerons and flaps on each side combined into one control surface which also does the work of an elevator control surface. On its rear wing it has some rudders at an angle. However, the controller developed is only for the pitch axis,

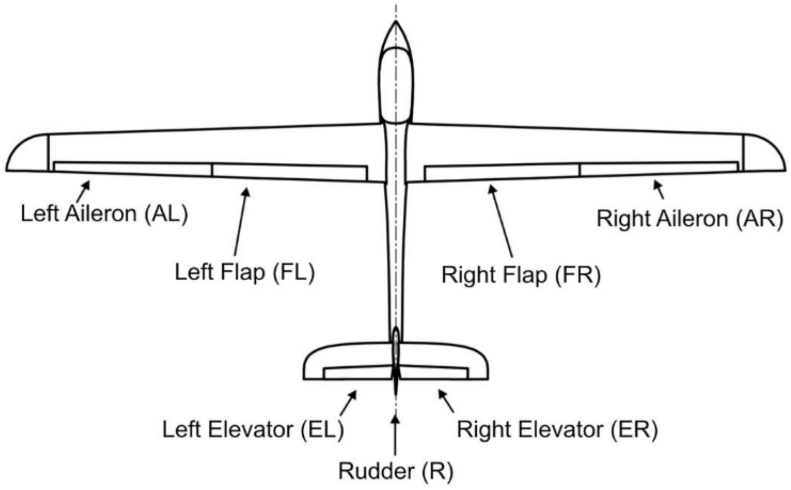

Figure 2.2: Normal control surfaces for an UAV [27].

thus the elevator is the only control surface we consider. The Aerosonde uses a propeller system to generate thrust, and this also needs to be a part of the control system of the vehicle. It is not further mentioned here, but is developed in Dahl [1].

2.4 Modelling

To be able to simulate the Aerosonde UAV, we need a mathematical model of how it behaves. The sections from here and including section 2.7 develop this mathematical model. The simulator is the same is the one developed in previously in Dahl [1] , so for further details about its implementation, see Dahl [1].

The notation used in modelling is mainly sourced from [7], and repeated here in short form.

In local navigation problems, the North East Down coordinate system (NED), shown in 2.3 is commonly used for displacement and the three Euler angles, yaw, pitch, roll, shown in 2.4, are representations of angular rotation [28]. This constitutes a Six-degrees-of-freedom (6-DOF) system. To model the UAV in 6-DOF, we combine them in a state vector η 2.2. Here p_n, p_e, p_d are the displacement in the north-, east-, down direction, and ϕ, θ, ψ are the rotations about those axes, named yaw, pitch, roll, respectively.

$$\eta = \begin{bmatrix} p_n \\ p_e \\ p_d \\ \phi \\ \theta \\ \psi \end{bmatrix} \tag{2.1}$$

The derivatives of the states are also of interest, and are put into their own state vector. However, the equations of motion which will be used to update the linear velocities are better expressed in other frames than NED. The equations of motion will be expressed in the body frame of the vehicle. We set the NED frame as the inertial frame which does not move, and the body frame moves relative to the NED frame. To transform between these states we use a rotation matrix $\mathbf{R}$, and to express a state in a frame, we use a superscript of the frame on the value. The rotation matrix has a subscript that describes which frame it transforms from and to.

$$\begin{bmatrix} \dot{p_n} \\ \dot{p_e} \\ \dot{p_d} \end{bmatrix} = \mathbf{R}_{\mathbf{b}}^{\mathbf{i}}(\phi, \theta, \psi) \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{2.2}$$

In this 2.2 we are rotating the linear velocities u, v, w of the vehicle in x, y, z from the body frame to the inertial frame (NED). The rotation matrix is given by 2.3 where c_x and s_x are given by $\cos x$ and $\sin x$.

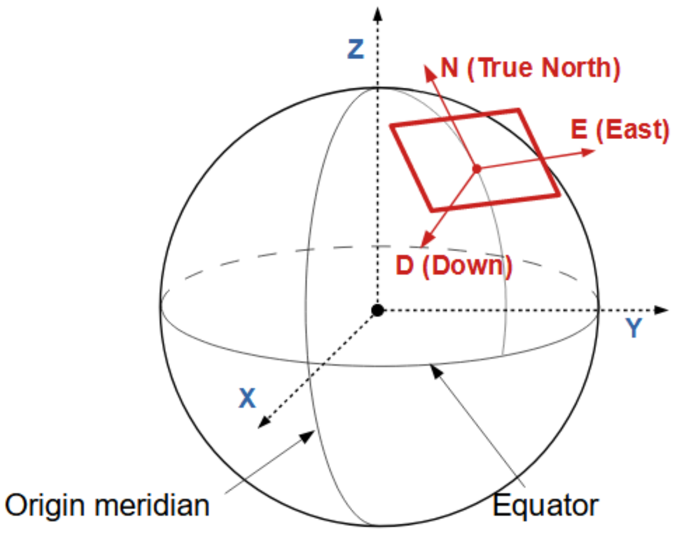

Figure 2.3: NED compared to ECEF. [29]

$$\mathbf{R_b^i}(\phi,\theta,\psi) = \begin{bmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta c_\psi - s_\phi c_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{bmatrix} \tag{2.3}$$

The Euler angle velocities are the same in both frames and are given by 2.4.

$$\begin{bmatrix} \dot\phi \\ \dot\theta \\ \dot\psi \end{bmatrix} = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{2.4}$$

2.5 Longitudinal dynamics

The longitudinal dynamics are only considering the p_n, p_d, and θ states, and only the control surfaces and actuators which affect these states. This results in a nonlinear ordinary differential equation.

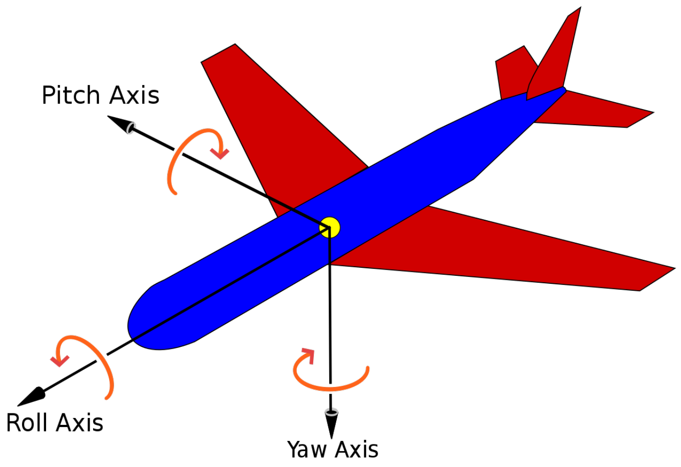

Figure 2.4: Euler angles of an aircraft. [30]

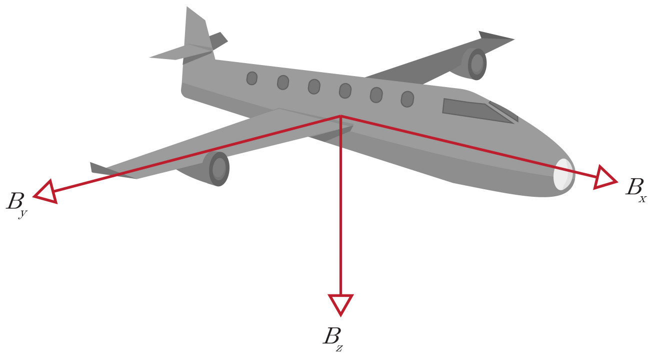

Figure 2.5: Body frame of an aircraft. [31]

A general nonlinear model is given by 2.5 which maps a state and control input to a state derivative,

$$\dot{x} = f(x, u) \tag{2.5}$$

with x being the state vector and u being the control input.

We can rewrite this to a 3-DOF differential equation for our aircraft,

$$\dot{\nu} = \mathbf{A}(\nu)\nu + \frac{1}{m}\mathbf{F}(\nu, u) \tag{2.6}$$

where ν are is the velocity state vector in the body frame $[u, w, q]^T$.

Using a rigid body model for our vehicle we can write our model in 2.7, 2.8, and 2.9 [7]

$$\dot{u} = -qw + \frac{f_x}{m} \tag{2.7}$$

$$\dot{w} = qu + \frac{f_z}{m} \tag{2.8}$$

$$\dot{q} = \frac{M}{J_y} \tag{2.9}$$

Here, f_x, f_z are the forces in the x and z direction, M is the pitching moment about the center of gravity, m is the vehicle's mass, and J_y is the moment of inertia about the y-axis

f_x and f_z are divided into three different parts: the gravitational force f_g and aerodynamic force f_a in both directions, and propeller thrust f_t in the x direction.

We model gravity in NED with a constant acceleration of $g = 9.81\ ms^{-2}$

$$\begin{bmatrix} f_{g_x} \\ f_{g_z} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} 0 \\ mg \end{bmatrix} \tag{2.10}$$

Thrust can be modeled as a front mounted propeller as in [7].

$$Fx_{prop} = \frac{1}{2}\rho S_{prop} C_{prop}((k_{motor}\delta_t)^2 - V_a^2) \tag{2.11}$$

Where ρ is the air density, S_{prop}, C_{prop} and k_{motor} are propeller specific constants. δ_t is the throttle command given which ranges from no throttle (0), to max throttle (1).

V_a is the air speed of the surrounding air. This is influenced by the speed of the vehicle relative to the ground and the wind. We model it as 2.12 with u_r and w_r defined in 2.13.

$$V_a = \sqrt{u_r + w_r} \tag{2.12}$$

$$\begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = \begin{bmatrix} u_g \\ v_g \\ w_r \end{bmatrix} - \begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} \tag{2.13}$$

Here we express the relative velocities to the wind. u_r is the relative velocity in x, u_g is the x-velocity perceived from a stationary ground, and u_w is the wind velocity in the x-direction. The same applies for the y- and z directions as well. All velocities are expressed in the body frame.

The aerodynamic surfaces are responsible for the remaining forces and moments. Here it is helpful to introduce another coordinate system frame, the stability frame. The stability frame is defined to align with the drag- and lift forces of the aircraft [7]. To transform between the body frame and stability frame, we rotate by the angle of attack,

$$\alpha = atan2(\frac{w_r}{u_r}) \tag{2.14}$$

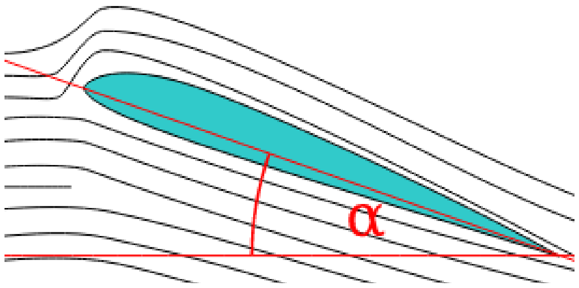

Figure 2.6: Angle of attack. Black lines are incoming air stream, and the blue geometry is the wing. [32]

which can also be seen in figure 2.6.

With the angle of attack, we can rotate the expressions for drag and lift in stability frame, to body frame by the rotation matrix

$$\mathbf{R}_{\mathbf{s}}^{\mathbf{b}}(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{2.15}$$

The model for aerodynamic lift, drag, and pitching moment are given by 2.16, 2.17 and 2.18.

$$F_{lift} = \frac{1}{2}\rho V_a^2 S C_L(\alpha, q, \delta_E) \tag{2.16}$$

$$F_{drag} = \frac{1}{2}\rho V_a^2 S C_D(\alpha, q, \delta_E) \tag{2.17}$$

$$M_\theta = \frac{1}{2}\rho V_a^2 S c C_M(\alpha, q, \delta_E) \tag{2.18}$$

S is the planform area of the wings 2.7 and c is the mean aerodynamic chord of the wings 2.8. δ_E is the control input to the elevator control surface. C_L,

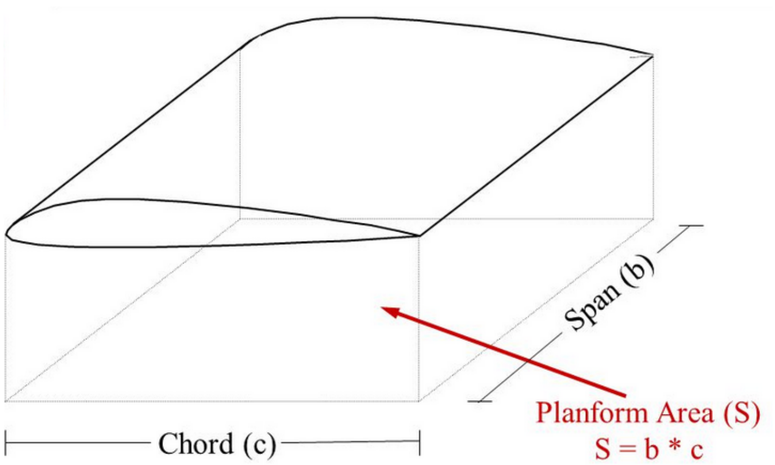

Figure 2.7: Planform area of aircraft wing. [33]

C_D, and C_M are the nonlinear functions of lift, drag, and pitching moment, respectively. These functions will capture the aerodynamic model of the aircraft and will vary from aircraft to aircraft. These are commonly found numerically through simulation or with wind tunnel testing for high fidelity simulations [7] [34].

Linearizing the functions gives 2.19, 2.20 and 2.21. The $C_{Lx/Dx/Mx}$ values are constants called stability derivatives [7]. These give a linear model of the coefficients of lift/drag/pitching moment which will be approximately correct for small values of α, q, and δ_e.

$$F_{lift} = \frac{1}{2}\rho {V_a}^2 S[C_{L0} + C_{L\alpha}\alpha + C_{Lq}\frac{c}{2V_a}q + C_{L\delta e}\delta_e] \tag{2.19}$$

$$F_{drag} = \frac{1}{2}\rho {V_a}^2 S[C_{D0} + C_{D\alpha}\alpha + C_{Dq}\frac{c}{2V_a}q + C_{D\delta e}\delta_e] \tag{2.20}$$

$$M_\theta = \frac{1}{2}\rho {V_a}^2 Sc[C_{M0} + C_{M\alpha}\alpha + C_{Mq}\frac{c}{2V_a}q + C_{M\delta e}\delta_e] \tag{2.21}$$

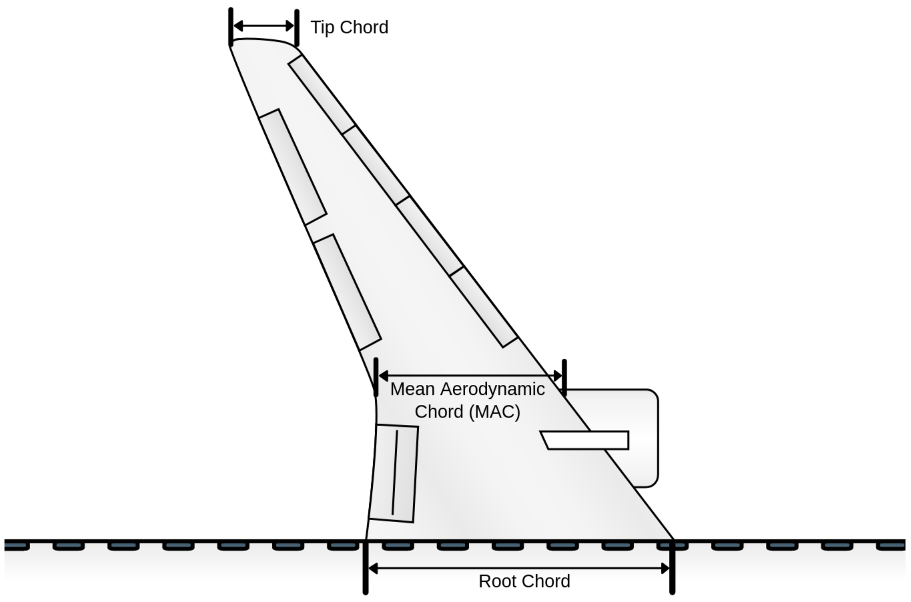

Figure 2.8: Mean aerodynamic chord of aircraft wing. [35]

For a higher fidelity model, nonlinear lift/drag/pitching moment coefficients are needed.

2.6 Wind

Wind is an important factor to model when simulating an aircraft. As shown in 2.12 and 2.14, we model the airspeed and angle of attack with the velocity of the aircraft relative to the surrounding air. Which ultimately means we model the aerodynamic forces of the vehicle relative to the surrounding air.

We model wind in two separate parts, steady flow, and gusts.

Steady flow is simple, we just model a constant speed in a direction. The direction is commonly directly facing of directly following the aircraft.

The gusts are more complicated. There are a plethora of gust models, but the one used here is the Dryden turbulence model [36] [7]. This model is based on transfer functions with white noise as input in body frame.

$$H_u(s) = \sigma_u \sqrt{\frac{2V_a}{L_u}} \frac{1}{s + \frac{V_a}{L_u}} \tag{2.22a}$$

$$H_v(s) = \sigma_v \sqrt{\frac{3V_a}{L_v}} \frac{(s + \frac{V_a}{\sqrt{3}L_v})}{(s + \frac{V_a}{L_v})^2} \tag{2.22b}$$

$$H_w(s) = \sigma_w \sqrt{\frac{3V_a}{L_w}} \frac{(s + \frac{V_a}{\sqrt{3}L_w})}{(s + \frac{V_a}{L_w})^2} \tag{2.22c}$$

2.7 Nonlinear aerodynamics

There are two approaches taken here on nonlinear aerodynamics. Coefficient look-up-tables, and higher fidelity models. Beard and McLain [7] and Vepa [34] develop higher fidelity models. The one developed in [7] is repeated in short form here.

2.7.1 Higher fidelity model

The resulting model for $C_L(\alpha)$ is given by 2.23.

$$C_L(\alpha) = (1 - \sigma(\alpha))[C_{L_0} + C_{L_\alpha}\alpha] + \sigma(\alpha)[2sign(\alpha)^2(\alpha)\cos(\alpha)] \tag{2.23}$$

This function merges the linear stability derivative model $C_{L0} + C_{L\alpha}$ with a nonlinear function $2sign(\alpha)\sin^2(\alpha)\cos(\alpha)$, while the other stability derivatives are identical.

The merging function, $\sigma(\alpha)$, used is given by 2.24

$$\sigma(\alpha) = \frac{1 + e^{-M(\alpha-\alpha_0)} + e^{M(\alpha+\alpha_0)}}{(1 + e^{-M(\alpha-\alpha_0)})(1 + e^{M(\alpha+\alpha_0)})} \tag{2.24}$$

with α_0 as the linear-/nonlinear cutoff, and M as a merging factor.

The coefficient of drag is a simpler model 2.25.

$$C_D(\alpha) = C_{D_p} + \frac{(C_{L0} + C_{L\alpha}\alpha)^2}{\pi e b^2/S} \tag{2.25}$$

with C_{D_p} is the shear stress or parasitic drag, S is the planform area of the wing, and b is the wingspan [7].

2.7.2 Look-up-tables

Look-up-tables are generated from experimental- or CFD data. Airfoiltools.com [37] is a large database of look-up-tables for different airfoils. XFLR5 [38] is an open-source software that can generate look-up-tables for different geometries of airfoils. Using these tools we can generate a set of different look-up-tables to simulate a certain airfoil. An example airfoil look-up-table can be seen in figure 2.9.

2.8 Airfoil icing

Airfoil icing can be categorized by different kinds of ice [6]

- Rime ice
- Glaze ice
- Mixed ice
- Supercooled large droplets

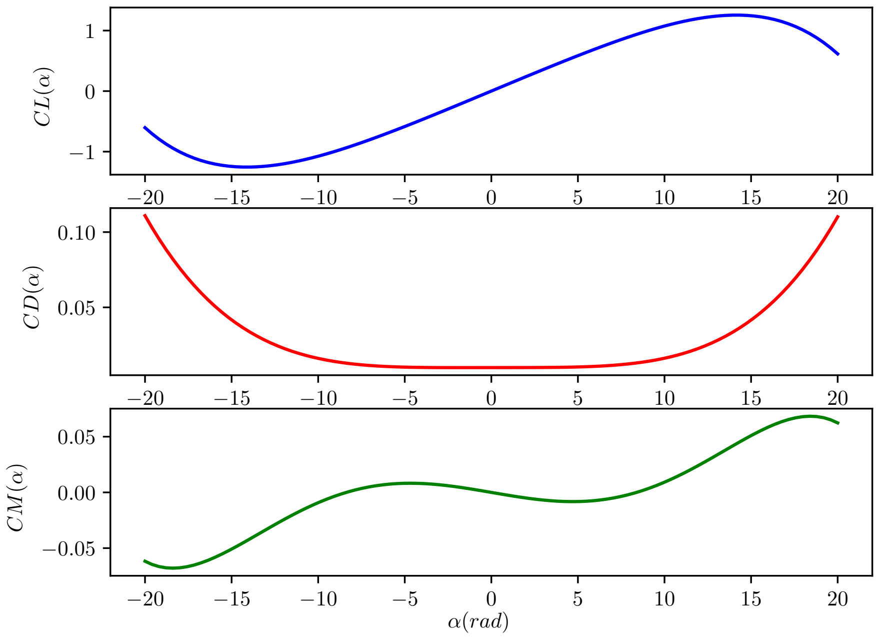

Figure 2.9: Example look-up-tables of NACA0011 airfoil. ([39])

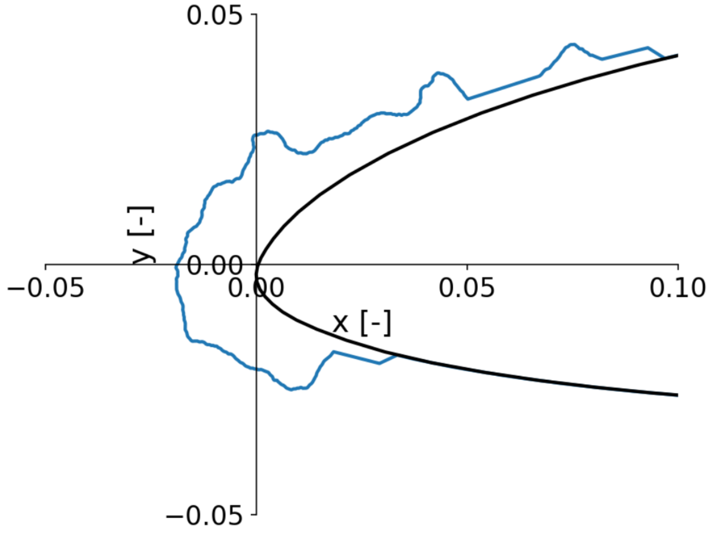

Figure 2.10: Example of glaze ice geometry (blue) on cross-section of airfoil (black) [2].

- Snow and ice crystals

All have different effects on the aircraft, but according to Hann [6], the type with the highest degradation on aerodynamic performance is glaze ice. Hence, we will use glaze ice in our simulations, and when referring to ”ice” we mean glaze ice.

In figures 2.10, 2.11 we see examples of glaze ice on airfoils. Although the details of how an airfoil actually generates lift are disputed [40][41], there

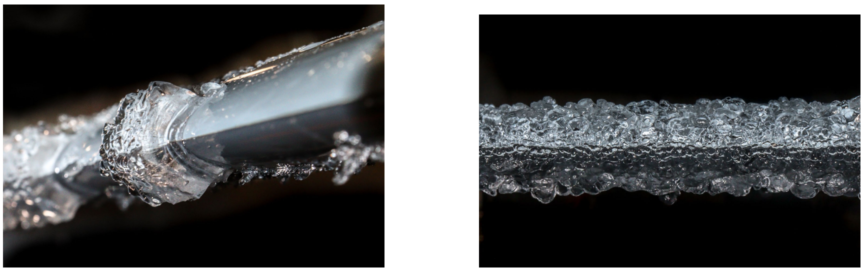

Figure 2.11: Pictures of glaze ice on leading edge of airfoil [6].

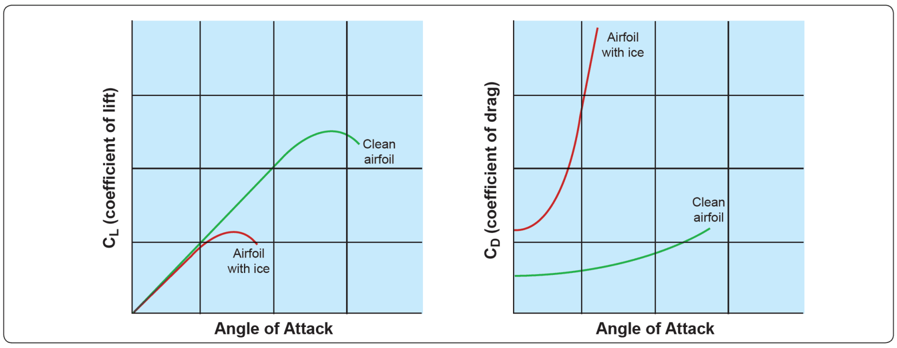

Figure 2.12: The general impact of icing on lift- and drag curves [42].

is consensus that the airfoil needs smooth curves to reduce separation if the air. With ice on the wings, the resulting aerodynamic performance will in general be: higher drag, lower lift, and stall conditions at lower angles of attack. This is illustrated in 2.12.

2.9 PID controller

The development of the PID controller is described in detail in Beard and McLain [7], but repeated here in short form.

$$\theta^c = k_{p_h}(h^c - h) + k_{i_h} \int (h^c - h)dt \tag{2.26}$$

$$\delta_e = k_{p_\theta}(\theta^c - \theta) - k_{d_\theta} q \tag{2.27}$$

The controller is summarized in the two equations 2.26, 2.27. Equation 2.26 is a proportional integral controller(PI), where θ^c is the desired pitch angle, and ^{c} and h are the desired height and current height, respectively. The k terms in both equations are gain parameters for tuning the response. Equation 2.27 is a proportional derivative controller(PD). This takes the desired pitch angle θ^c as an input, the current pitch angle θ, and the current pitch rate q. It outputs the elevator control input δ_e.

The core idea of the controller is to have two controllers in series, meaning the output of the PI controller is the input to the PD controller. This only works if the inner-loop controller, in our case the PD controller, is much faster than the outer loop controller. Another point to take note of is that this controller makes no assumptions of the plant model it is meant to control. It is not a model-based controller like the dynamic inversion controller is.

2.10 Dynamic inversion

Dynamic inversion gained traction in the 80s and 90s as a robust nonlinear control method which did not require expensive testing and gain scheduling to be effective. It could also be applied to different aircraft with different characteristics without needing a complete redesign [43].

The basic theory of dynamic inversion is briefly presented in Dahl [1], which will not be repeated here, but expanded upon further.

For a linear system described by the equations 2.28, it is often desired to follow a trajectory $r(t)$, which gives the error dynamics equation 2.29.

$$\dot{x} = \mathbf{A}x + \mathbf{B}u \tag{2.28a}$$

$$y = \mathbf{C}x \tag{2.28b}$$

$$e(t) = r(t) - y(t) \tag{2.29}$$

$$u = (\mathbf{CB})^{-1}(\dot{r} + \mathbf{K}e - \mathbf{CA}x) \tag{2.30}$$

2.30 will stabilize the system as long as the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are known to sufficient accuracy [43], [44], and the gain matrix, $\mathbf{K}$ is positive definite [45].

The system in 2.28 is linear, but dynamic inversion expands to nonlinear systems as well.

A general nonlinear differential equation with control input can be written as

$$\dot{x} = \mathbf{f}(x) + \mathbf{g}(x)u \tag{2.31}$$

It can also be written in companion- or canonical controllable form

$$\begin{bmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_{n-1} \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} x_2 \\ \vdots \\ x_n \\ b(x) \end{bmatrix} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ a(x) \end{bmatrix} u \tag{2.32}$$

Now all the nonlinear terms have been moved to the final state and keeping the rest linear. Making a virtual control input

$$v = b(x) + a(x)u \tag{2.33}$$

and inverting it to gain a control input. Now we can control this variable, and the rest of the states are controlled linearly.

$$u = a^{-1}(x)(v - b(x)) \tag{2.34}$$

Now we can insert the same error-term from 2.29 with a constant positive definite gain matrix $\mathbf{K}$ to control the system.

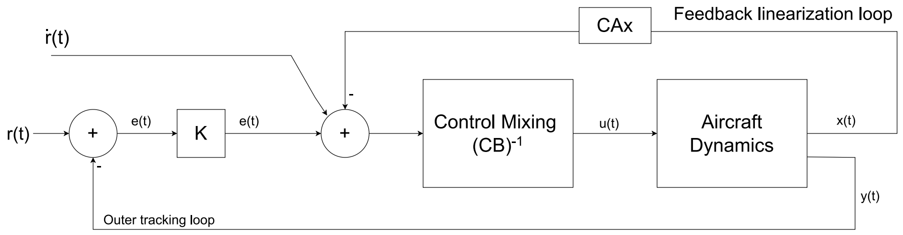

Figure 2.13: Dynamic inversion flow chart [1].

$$u = a^{-1}(x)(\mathbf{K}e - b(x)) \tag{2.35}$$

2.11 Artificial neural networks

Artificial neural networks are a broad class of machine learning techniques for estimation, classification, and data generation. Classical programs have a pre-defined set of instructions on what to do with a certain input, but machine learning programs learn the instructions based on the data it receives. This can happen in a supervised fashion, which means the algorithm is trained on labeled data, or unsupervised, which means the network has to infer some structure out of the training data by itself.

Neural networks have been proposed for over half a century [46], but only recently, with the continuing improvement of computer hardware, have become a prominent method across multiple fields of research. The simplest form of a neural network is feed-forward network.

2.11.1 Feed-forward neural networks

A feed-forward network is a network with an input layer, output layer, and n amount of hidden layers in between. The number of nodes in each layer, the number of hidden layers, choice of cost function, and choice of activation function are design parameters for the user. It has been shown that with two hidden layers and a nonlinear activation function, a network can approximate any nonlinear function [47]. Hornik [48] showed that a network with a single hidden layer can approximate any nonlinear mapping from one finite dimensional space to another.

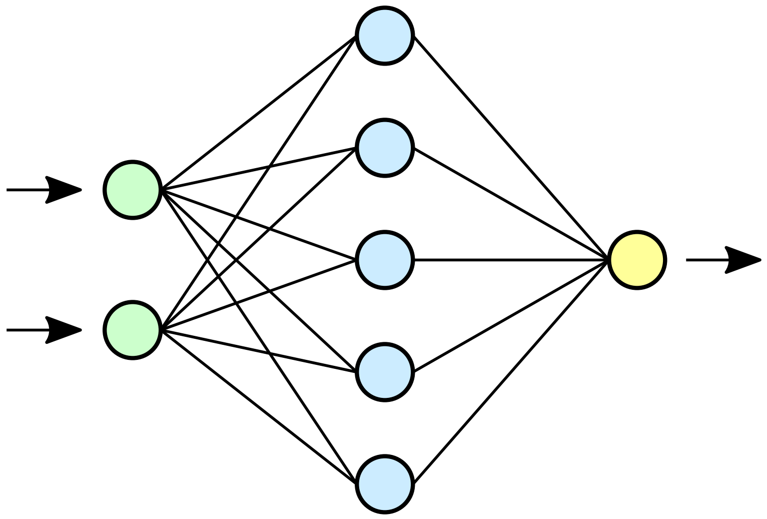

Figure 2.14: Neural network visualization. Input layer(left), hidden layer(middle), and output layer(right).

The mathematical expression for a single node in a hidden- and output layer is shown in 2.36. Here, z is the resulting node value, x_i are the inputs from the previous layer, b_i is the bias for this node, and w_i are the weights from the previous layer to this one node.

$$z = \sum_{i=1}^{n} x_i w_i + b_i \tag{2.36}$$

Expanding the expression for a whole layer, it becomes 2.37.

$$\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \tag{2.37a}$$

Or in vector variable form:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.37b}$$

However, to be able to fit nonlinear data, we need to add a nonlinear activation function. A common activation function is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.38}$$

Meaning our complete expression for a fully connected feed forward network becomes:

$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.39}$$

Feeding forward is how the network predicts data based on its inputs. When training the network however, we need to change the weights and biases of the network based on how well the network did on a prediction. A common algorithm for doing this is called back propagation.

Backpropagation is to apply gradient descent on a cost function comparing a networks prediction with the labeled, correct, value ([49]). There are many such cost functions, but mean squared error (MSE) is a common one.

$$C = \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2 \tag{2.40}$$

Here $\hat{y}$ is the network output, y is the correct value, and n is the amount of outputs from the network. (If the network is a single-value estimator, n would be 1.)

The ultimate goal of the backpropagation algorithm is to change the weights so the cost function reaches a satisfactory local- or global minimum. Using gradient descent with a learning rate parameter α (not to be confused with the angle of attack), we can tune how fast this convergence happens. The resulting algorithm is expressed in 2.41 for each weight in each layer. This has to be done for every weight and bias in every layer, but the basic expression is the same for all.

$$w_i \leftarrow w_i - \alpha \frac{\partial C}{\partial w_i} \tag{2.41}$$

For a deeper explanation, see [49].

2.12 Symbolic Regression and SINDy

Sparse identification of nonlinear dynamics (SINDy) is a symbolic regression algorithm proposed in Brunton [17]. It is a data-driven machine learning algorithm with the goal of producing the underlying differential equations which the input data is sampled from. This is similar to different classes of neural networks (i.e. feed-forward networks, recurrent neural networks, etc.), but SINDy produces short-form explainable differential equations [17]. This is different to neural networks which often have a larger number of parameters, and its behavior is difficult to explain. The difference from other regression techniques, is that SINDy allows a weighing of sparsity. Sparsity allows for the resulting model to be more resistant to overfitting noise. It has also been shown that SINDy can generalize better to out-of-distribution data, meaning it generalizes beyond the training set better than neural networks.

SINDy is presented in Brunton [17], but repeated in short form here.

We start with gathering data on the plant we wish to model. If the derivative of the state $\dot{\mathbf{x}}$ is available to measure, we use that. If only the state $\mathbf{x}$ itself is available, we input the state and estimate the derivative with finite differences or some other method ([22]).

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^T(t_1) \\ \mathbf{x}^T(t_2) \\ \vdots \\ \mathbf{x}^T(t_m) \end{bmatrix} = \begin{bmatrix} x_1(t_1) & x_2(t_1) & \dots & x_n(t_1) \\ x_1(t_2) & x_2(t_2) & \dots & x_n(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ x_1(t_m) & x_2(t_m) & \dots & x_n(t_m) \end{bmatrix} \tag{2.42}$$

$$\dot{\mathbf{X}} = \begin{bmatrix} \dot{\mathbf{x}}^T(t_1) \\ \dot{\mathbf{x}}^T(t_2) \\ \vdots \\ \dot{\mathbf{x}}^T(t_m) \end{bmatrix} = \begin{bmatrix} \dot{x_1}(t_1) & \dot{x_2}(t_1) & \dots & \dot{x_n}(t_1) \\ \dot{x_1}(t_2) & \dot{x_2}(t_2) & \dots & \dot{x_n}(t_2) \\ \vdots & \vdots & \ddots & \vdots \\ \dot{x_1}(t_m) & \dot{x_2}(t_m) & \dots & \dot{x_n}(t_m) \end{bmatrix} \tag{2.43}$$

Next, we define a library of potential candidate functions. These are operations we conjecture that could be done to the input to give an accurate answer. If we know it is a linear differential equation, we add only linear terms. If we know it is trigonometric, we add only trigonometric functions. If the useful operations are unknown, we have to experiment to find a suitable set. A standard polynomial library is a good start. If the SINDy process is done online and only needs to be valid locally, a polynomial library can approximate a Taylor expansion of the actual dynamics, and be accurate enough locally [50].

$$\boldsymbol{\Theta}(\mathbf{X}) = \begin{bmatrix} | & | & | & | & & | & | & \\ \mathbf{1} & \mathbf{X} & \mathbf{X}^{P_2} & \mathbf{X}^{P_3} & \dots & \sin(\mathbf{X}) & \cos(\mathbf{X}) & \dots \\ | & | & | & | & & | & | & \end{bmatrix} \tag{2.44}$$

We also merge all linear combinations of states. As an example, the second order polynomial candidate library would look like 2.45.

$$\mathbf{X}^{P_2} = \begin{bmatrix} x_1^2(t_1) & x_1(t_1)x_2(t_1) & \dots & x_2^2(t_1) & \dots & x_n^2(t_1) \\ x_1^2(t_2) & x_1(t_2)x_2(t_2) & \dots & x_2^2(t_2) & \dots & x_n^2(t_2) \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ x_1^2(t_m) & x_1(t_m)x_2(t_m) & \dots & x_2^2(t_m) & \dots & x_n^2(t_m) \end{bmatrix} \tag{2.45}$$

The sparsity is enabled through a vector of sparsity coefficients 2.46. These are the coefficients we run the regression over to determine which candidate functions are active ([17]).

$$\Xi = \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_n \end{bmatrix} \tag{2.46}$$

The resulting regression problem is state as 2.47.

$$\dot{\mathbf{X}} = \boldsymbol{\Theta}(\mathbf{X})\boldsymbol{\Xi} \tag{2.47}$$

Once the regression problem is solved and $\boldsymbol{\Xi}$ is found, each state variables differential equation can be written as

$$\dot{\mathbf{x}}_k = \mathbf{f}_k(\mathbf{x}) = \boldsymbol{\Theta}(\mathbf{x}^{\mathbf{T}})\xi_k \tag{2.48}$$

[17]

The regression problem posed in 2.47 can be solved by different tools. The original SINDy paper [17] proposes to use least absolute shrinkage and selection operator (LASSO). However, future developments implemented in PySINDy [22], give more methods. Sparse Relaxed Regularized Regression (SR3) and Sequentially-Thresholded Least-Squares algorithm (STLSQ) are the recommended ones to use.

PySINDy is a python library by [22] which implements the SINDy algorithm described above. With different tools to improve its robustness and methods to solve it.

PYSR [51] is a python wrapper for the Julia [52] library SymbolicRegression.jl [53]. It also implements symbolic regression in a similar way to PySINDy, but uses a genetic algorithm instead of regression to find the appropriate model. The benefit of using PYSR over PySINDy is dependent on the problem. PYSR is much slower, so not fit for real-time use, but has higher accuracy if the resulting expression has nested functions and complicated expressions and if the candidate functions are not known. PySINDy is a better option for short-form expressions and real-time use if the functional basis is known [54].

2.13 Robot Operating System

Robot operating system (ROS) is an open-source framework for robotics development. It comes with interfaces and plugins for different simulation- and visualization software. It also takes care of running multiple programs, or in ROS terms, nodes, simultaneously. Which offloads the task of multi-threading and passing thread-safe messages between nodes from the user. This allows for quick development and quickly deployable software. ROS has native support for the Python- and C++ programming languages, but

3rd party software is available to develop with ROS in many other languages as well. A benefit of developing the control software in ROS is that the road from a prototype in a simulator to production in the real world is short. The control system only cares about what input it receives and produces an output. ROS allows for switching between real-world feedback from sensors, and simulated ones in a simulator seamlessly, since all messaging between nodes is done via ROS topics. A topic is just a messaging pipeline that allows nodes to listen to its messages. ROS shortens the number of steps needed to take to go from a pure simulation project to applying it to a real-world system. This allows the developed control system to be tested on an actual UAV in the future without refactoring the entire program.

Chapter 3

Method

3.1 Introduction

This chapter will explain the implementation-specific actions that were taken to achieve the results in the next chapter. The issues that are needed to solve to make SINDy work on our specific problem are:

- Choice of coordinate system for the input data
- Modify the input data to make the solver converge easier
- Process the icing data into a model possible for SINDy to find

Additionally, a section is added specifically on how the algorithm is implemented in code. There are considerations needed to make it function in a real-time system, and a solution is presented.

3.2 Continuation of previous work

Dahl [1] developed a dynamic inversion controller based on a neural network architecture to estimate the error in the linear model. Results were varying, and the controller had large discrepancies in tracking small differences in input. The further work proposed in [1] included deeper neural network architecture, recurrent neural networks, and symbolic regression as possible improvements to the system. Here, the symbolic regression solution is further explored.

3.3 Model parameters

The table in 3.1 are the values used in the simulation of the UAV. These are taken from the Aerosonde parameters in [7].

Parameter	Value	Longitudinal Coef.	Value
m	$25kg$	C_{L_0}	0.23
J_y	$2.135kgm^2$	C_{D_0}	0.043
S	$0.55m^2$	C_{m_0}	0.0135
b	$2.8956m$	C_{L_α}	-0.98
c	$0.18994m$	C_{D_α}	0.30
S_{prop}	$0.2027m^2$	C_{m_α}	-2.74
ρ	$1.2682kgm^{-3}$	C_{L_q}	0
k_{motor}	80	C_{D_q}	0
k_{T_p}	0	C_{m_q}	-38.21
k_Ω	0	$C_{L_{\delta_E}}$	-0.36
e	0.9	$C_{D_{\delta_E}}$	0
		$C_{m_{\delta_E}}$	-0.99
		C_{prop}	1.0
		M	50
		α_0	0.4712
		ϵ	0.1592
		C_{D_p}	0
		$C_{n_{\delta_r}}$	-0.032

Table 3.1: Table of used parameters used in the simulation of the Aerosonde UAV.

3.4 SINDy

3.4.1 Change of coordinates

We require an accurate model of the pitch dynamics of the aircraft to control it. Looking at the model in 2.9 and 2.21 we see it is not necessarily too sparse. The model has four separate terms added together, and the maximum order of a term is the C_{m_α} term which is multiplied by both V_a^2 and α. This gives a third order model with interacting terms. However, the terms themselves are functions of the state variables, as shown in 2.12 and 2.14. Having a model of the third degree with interacting terms, square roots, and a trigonometric function is not sparse. Hence, we need to express the dynamics of the model in a different coordinate system than the state variables.

By changing the input to the SINDy algorithm, we do not get an output of our state variables. Pysindy tries to find a sparse representation of the input variables + control inputs, meaning, if we change the inputs to better accommodate the pitch dynamics, we need to do this separately for each state variable we wish to estimate.

The best change of coordinates we can do for the pitch dynamics is to have the inputs to the system be V_a, α, q, and δ_e as a control input. SINDy will now estimate $\dot{q}$, $\dot{V_a}$ and $\dot{\alpha}$. While $\dot{V_a}$ and $\dot{\alpha}$ contain information on $\dot{u}$ and $\dot{w}$, their expressions are even less sparse than the pitch dynamics. They contain rotation from the stability frame to the body frame as well, thus further transformations of the results are needed, or SINDy will need to try and estimate these rotations as well.

Either way, for pitch control, we only require the pitch rate dynamics, and thus we can use the coordinate transformation to V_a and α.

Further transformations can be done. Since the pitching moment 2.21 is directly proportional to V_a^2, we can divide $\dot{q}$ by it, to reduce the model dynamics to second order. The downside of this is that we need to take care of our $\dot{q}$-data ourselves. Pysindy includes different methods to numerically differentiate the input data to be used as ground truth for the SINDy algorithm. If we wish to pre-divide all $\dot{q}$ data points by V_a^2, we need to do this manually, and feed in the ground truth data directly. Another downside is that this action presumes prior knowledge of the dynamics we wish to model. If we wish to apply SINDy to model a totally unknown system, we can not base the usage of the algorithm on prior knowledge.

Some prior knowledge methods like pre-dividing are useful in this application, but some other modifications to the inputs can be an important factor in a general application of SINDy.

3.4.2 Input modification

Pysindy tries to find the coefficients of the terms in the model which have the largest impact on the result. Considering our goal of modelling the pitch dynamics, this corresponds to some aerodynamic coefficients we want to find. But since the whole expression in 2.21 is reliant on V_a^2(2.12), the coefficients would all need to be quite small. With a $V_{a_{trim}} \approx 35m/s$, $V_a^2 \approx 1200$ while $|\dot{q}| \approx 1$, would require the desired coefficients for SINDy to find be on the order of $1/1000$. This poses a problem for pysindy, since it rarely finds such low coefficients, and will just find a larger combination of higher valued coefficients instead. To combat this, we need to introduce some tuning variables. These can be inferred from educated guesses of the desired dynamic model, and the desire is to shift all the coefficients to the same order of magnitude above 1.

Using the values for simulation given in [1], the linear pitching moment equation becomes

$$\dot{q} = 0.00042V_a^2 - 0.085\alpha V_a^2 - 0.11qV_a - 0.031\delta_e V_a^2 \tag{3.1}$$

These coefficients are distributed over three orders of magnitude which makes the sparsification of the regression problem more difficult. If we augment the inputs to the model we are trying to fit, we can make the problem easier.

$$\bar{V_a} = \frac{V_a}{n} \tag{3.2}$$

Changing our inputs to augmented inputs in the form of 3.2 for some/all of α, δ_e and q as well, reformulates the differential equation to

$$\dot{q} = 0.00042\frac{V_a^2}{n_1^2} - 0.085\frac{\alpha}{n_2}\frac{V_a^2}{n_1^2} - 0.11\frac{q}{n_3}\frac{V_a}{n_1} - 0.031\frac{\delta_e}{n_4}\frac{V_a^2}{n_1^2} \tag{3.3a}$$

$$\dot{q} = 0.00042\bar{V_a^2} - 0.085\bar{\alpha}\bar{V_a^2} - 0.11\bar{q}\bar{V_a} - 0.031\bar{\delta_e}\bar{V_a^2} \tag{3.3b}$$

Now, the only possible way for $\dot{q}$ to be the same, is if the coefficients are increased by the same amount the inputs were divided by.

By applying this augmentation to the inputs, and using $n_1 = 200$, $n_2 = 0.01$, $n_3 = 1$ and $n_4 = 0.001$, we get the system in 3.4.

$$\dot{q} = 16.755V_a^2 - 34.006\alpha V_a^2 - 22.518qV_a - 12.287V_a^2\delta_E \tag{3.4}$$

Here, all coefficients are of the same order of magnitude, and the problem is now much easier for Pysindy to solve.

There is obviously a great advantage to knowing the model the simulator uses to tune these augmentation values. In other cases where the model dynamics are completely unknown, this becomes more difficult. However, in most applications, there should be some prior knowledge of the system which can be used to tune these values. But in the end, it does introduce some additional tuning parameters that need to be considered when using the SINDy method. Pysindy does have the option to use proprietary optimizers similar in form to SciPy optimizers[55]. Using this, it should be possible to make an optimizer that will not require this augmentation step to be successful, but this was not further explored in this thesis.

3.5 Airfoil icing

Under icing conditions, it is difficult to get experimental or CFD simulation data. Oswald et al. [2] has published data on icing conditions for an RG-15 airfoil. While a different airfoil than the one used on the Aerosonde, its experimental data is still functional as a basis for simulation of the Aerosonde. To make alterations to this data to fit the Aerosonde would be highly speculative. Thus, it has been used here without any alterations.

$$C_m(\alpha) = -0.00042418\alpha^2 + 0.00392625\alpha - 0.0039956 \tag{3.5}$$

In 3.1, Oswald et al. [2] presents experimental data for the pitching moment derivative vs. angle of attack at a Reynolds number [57] of $2.0 \cdot 10^5$, meaning turbulent flow. Comparing the plot to the data found in 3.1 and models in 3.2, we see some similarities in especially the clean foil. Though the linear approximation is a rough one, with a larger set of angles of attack, we would likely see a similar picture to the one in 3.3. Looking at the icing wing, we see the same general shape in the higher angles of attack, but without any obvious symmetry around $\alpha = 0$. This means at angles of attack of 0 and below, the $C_m(\alpha)$ coefficient induces some instability to

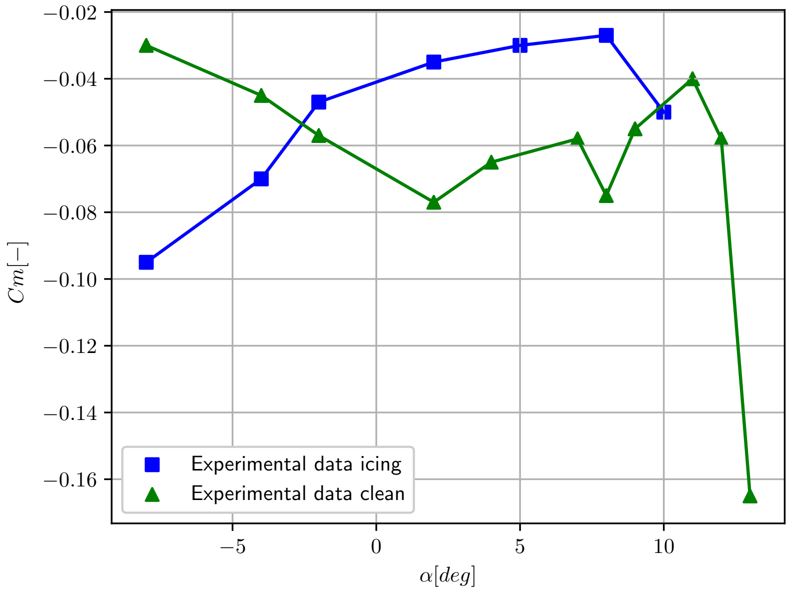

Figure 3.1: Experimental icing- and clean data from Oswald et al. [2].

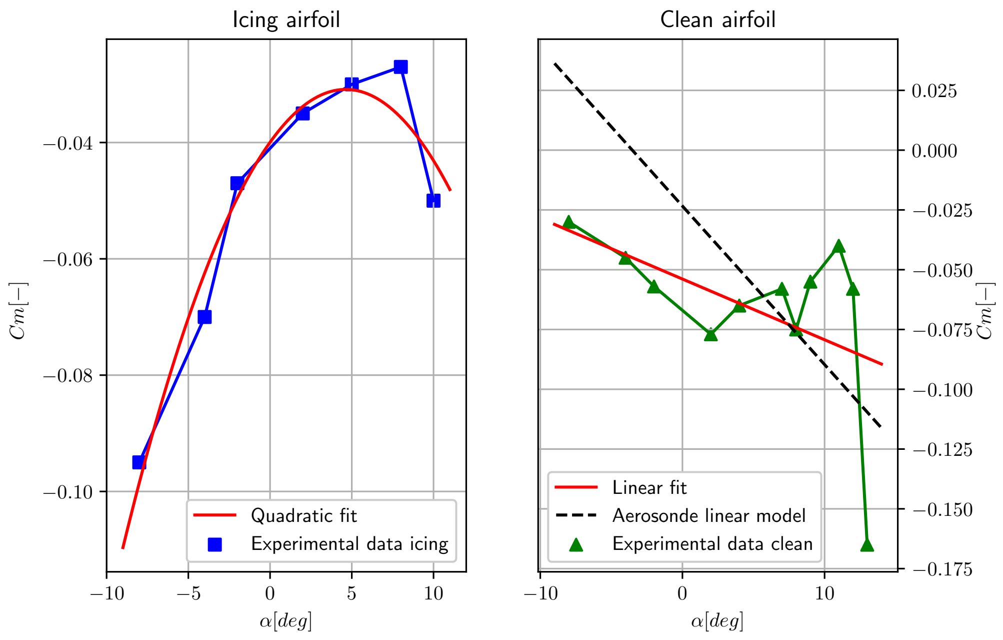

Figure 3.2: Curve fitting the icing data (left). Comparing Non-iced data with the Aerosonde linear model (right).

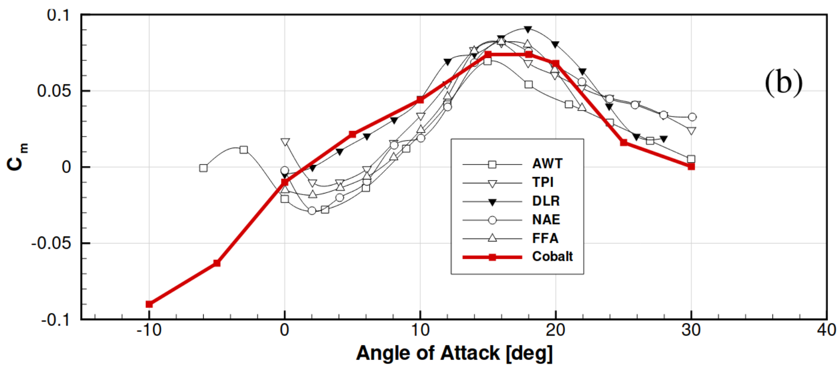

Figure 3.3: Pitching moment coefficients found through wind tunnel testing in [56].

the system, as can be seen in the pitch dynamics in 2.9 and 2.21. Negative angle of attack and a negative coefficient gives positive addition to $\dot{q}$, which can give instability if the other derivatives do not reduce it.

For this reason, the model in figure 3.2 and equation **??** was chosen to be used for testing if the control systems can keep control during icing of the airfoils.

3.6 Algorithm implementation

There are certain goals of the algorithm we need to consider when implementing it.

- The algorithm should be able to generate a new control model while running.
- It should be easy to switch from one controller to another mid flight.
- Computational overhead for processing large amounts of data real-time during flight.
- The estimation algorithm has to be separated from the simulation software to best model a real-world flight.

To achieve these goals, we create a separate ROS node, or without ROS, a separate thread. The separate node runs simultaneously with the simulation and is only started once the simulation has gathered enough data to start the model training. There is no lower limit on the data one can use, so this limit should not be the source of much consideration. Next, we need to consider how much processing power the algorithm should consume. The amount of data available for training increases at the frequency of the sensors, but storing and training on all the data during long flights quickly becomes infeasible using current implementations of SINDy on current computer hardware. Thus, a rolling window of past training data has to be used. The size of this window needs to be evaluated based on priorities: A larger window gives longer processing times for new models, but the models received should be more accurate. A smaller window is the other way around.

SINDy implemented with Pysindy has access to multiple solvers for the regression problem 2.47. The recommended ones are sequentially-thresholded least-squares algorithm (STLSQ) [58], and sparse relaxed regularized regression (SR3) [59]. Both algorithms include a threshold tuning variable which sets the minimum included coefficient value. If a coefficient found is below the threshold, then it is not included in the model. Working on completely unknown data, it can be difficult to know which value to use as the threshold, so doing the optimization over a range of thresholds, and picking the model with the lowest error-metric increases the quality of the output. Some issues with this method are that it is prone to overfitting if the test-set is part of the training set. Thus, a separate test set with un-seen data is helpful. Using an error-metric like MSE on the test set does not guarantee a sparse model either. If we run the optimization over five different thresholds, we wish to use the model with the lowest MSE with the testing data, which is the most sparse, and is better than the current model we are currently using. From this, a new metric was made, which punishes test set error, punishes a high amount of terms in the found model compared to the other found models, and updates the MSE on the estimation done by the model currently in use. In addition, a small punishment is given to a model which has been in use for a long time, since the assumption is that newer data gives a better representation of the dynamics than the older data.

A consideration needed when applying dynamic inversion is that the implementation has to have logic ready to handle a completely wrong model, but with a low error-metric. If a model is found which has a term with a quadratic-, cubic-, or higher order control input, the model can not be straightforwardly inverted. We have two options when implementing: Checking the model input for resulting square-, cube- or higher order roots that end up in the inversion to ensure they never take a negative root or limit the library functions available for the control input. The latter is much easier to implement in the pitch rate model, but the former is more generalizable to different models. However, if a model is suspected to have a higher order control input, it might be easier to modify the input and limit the control input library since Pysindy easily facilitates this.

The resulting algorithm flowchart can be seen in figure 3.4.

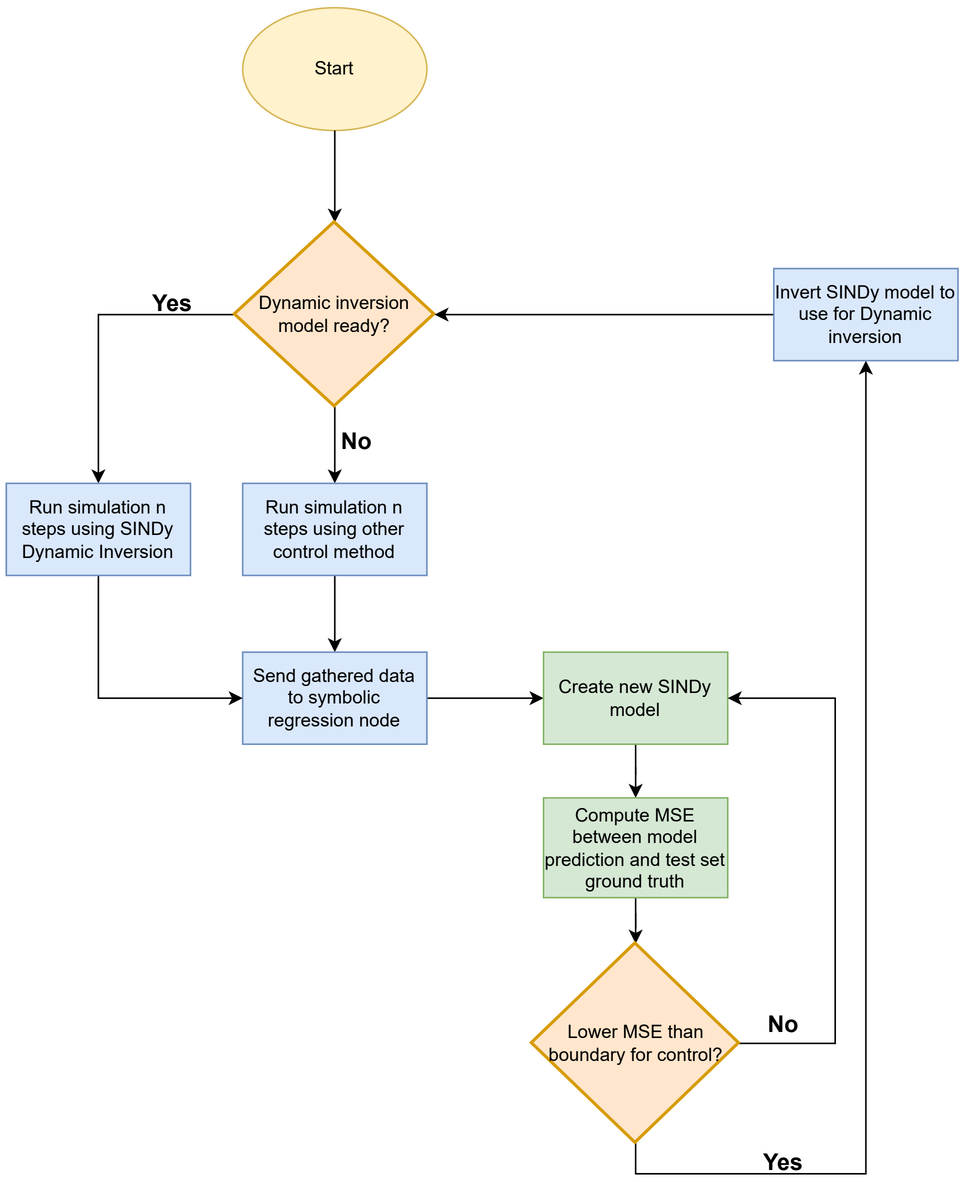
Start
Dynamic inversion model ready?
Yes
No
Invert SINDy model to use for Dynamic inversion
Run simulation n steps using SINDy Dynamic Inversion
Run simulation n steps using other control method
Send gathered data to symbolic regression node
Create new SINDy model
Compute MSE between model prediction and test set ground truth
Lower MSE than boundary for control?
No
Yes

Figure 3.4: Flowchart of the basic implementation of the online SINDy control algorithm. Blue nodes are running in the main thread of the program and green nodes run in a separate thread.

Chapter 4

Results

Presented are the results of multiple simulations done on the different test cases. The tracking set points are generated using a random walk [60], which are then low-pass filtered for smoothing, and lastly, some random step-functions are added in for diversity. When comparing multiple control schemes together, the random seed is set constant to get identical tracking set points. The random walk was also used to generate the test set data for the SINDy algorithm. It used a different seed to generate the data than it was run on, so it is only unseen data.

Some plots only contain SINDy and the neural network-based controller. This is because the two dynamic inversion controllers are made to track pitch rate, while the PID controller, tracks pitch. The plots that include all three controllers are the height tracking plots. These could be used as a basis for all of the plots, but it is often easier to see the difference in performance between the controllers in pitch rate. Hence, when comparing the SINDy controller and the neural network controller, pitch rate is the value of interest.

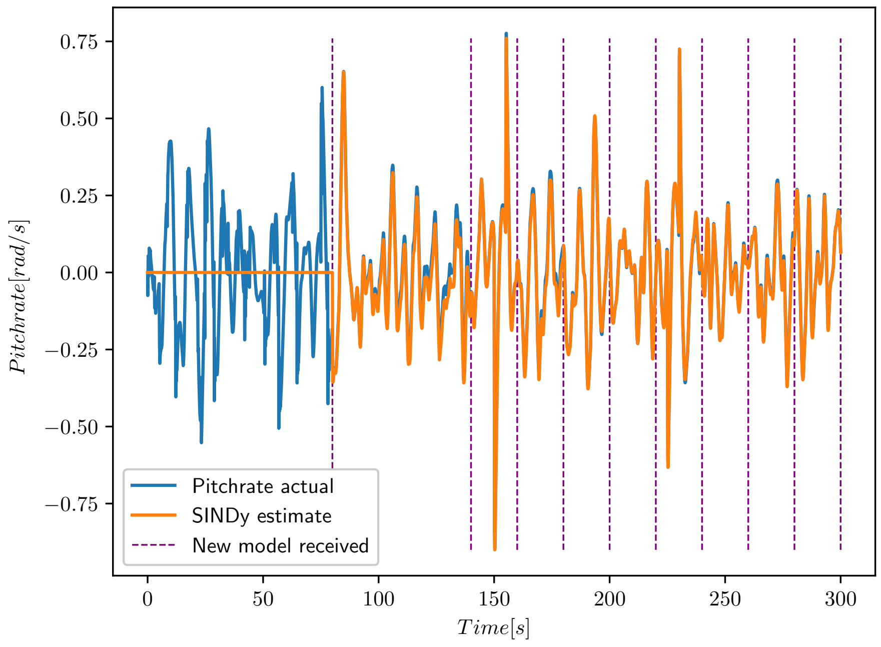

Figure 4.1: SINDy tracking of pitch rate running online.

4.1 Non-icing conditions

In figure 4.1 a varying height flight path is simulated with an online estimation of the pitch rate model. The model is updated at specific times, but only kept if it has improved performance on the testing data. We clearly see it converging toward the correct model.

The pitch rate derivative is shown in figure 4.2. Zooming in on figure 4.3 we can see some offsets in the pitch rate. Though small, they add up when integrating the values, and this is the result we in figure 4.1 where the tracking diverges.

The resulting model from SINDy gave the model in 4.1, while the actual

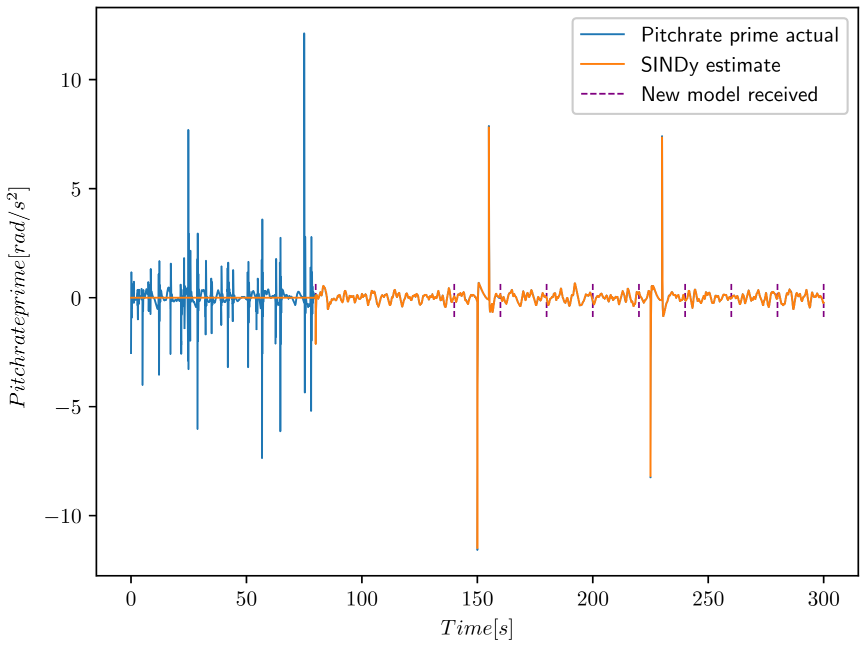

Figure 4.2: SINDy tracking of pitch rate prime running online.

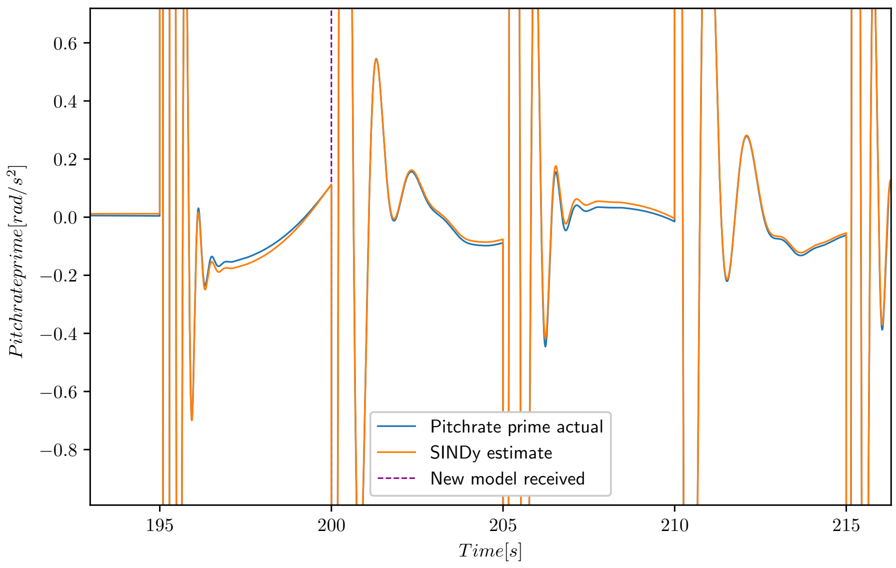

Figure 4.3: SINDy tracking of pitch rate prime zoomed in. Some small offsets are clearly present. Integrating these offsets up to pitch rate gives a larger offset in pitch rate over time.

model used in the simulation was the model in 3.4. These are identical in form with only small numerical differences. Across multiple runs, small additional terms can be included in the estimated model. In testing these appear most commonly as airspeed V_a terms to a higher degree. Most common is CV_a^3, with C being some small constant less than 0.1. Keeping in mind this model is with the shifted input arguments, V_a is less than 0.5, making these terms insignificant compared to the output.

$$\dot{q} = 16.743V_a^2 - 33.756\alpha V_a^2 - 22.189qV_a - 12.185V_a^2\delta_E \tag{4.1}$$

Since this is a simulation, SINDy had access to perfect data. But adding wind to the model using the technique presented in section 2.6 adds noisy

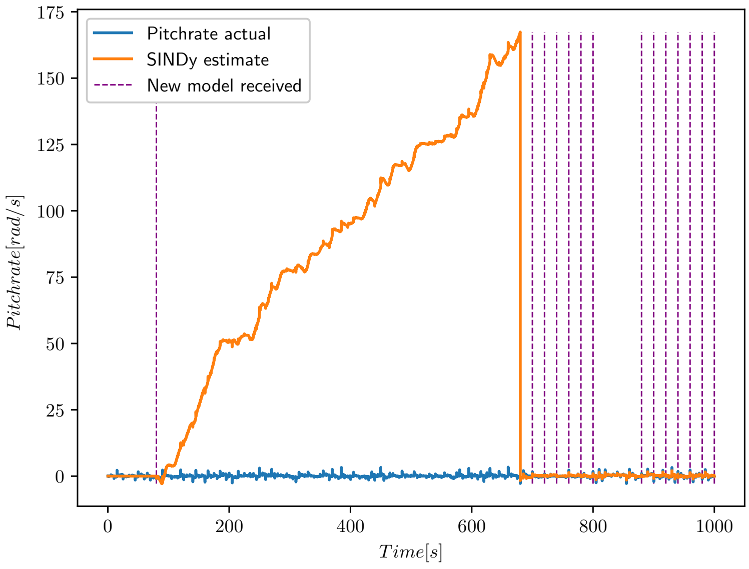

Figure 4.4: Pitch rate estimates using SINDy with added wind to the simulation. The model converges, but much slower than with perfect measurement data.

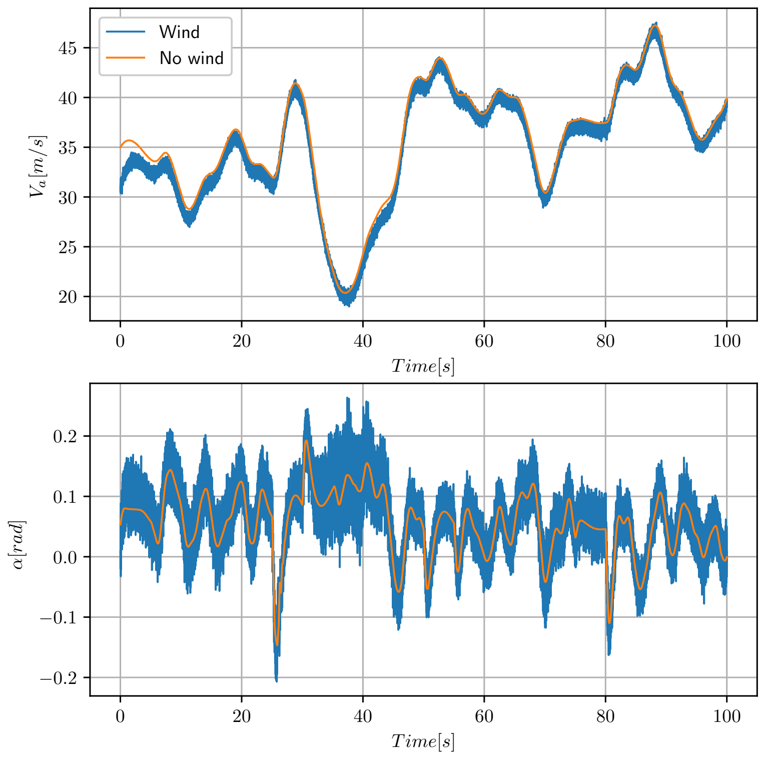

Figure 4.5: Measurements received by SINDy when wind is applied vs. when no wind is applies. Top plot is the airspeed (2.12). Bottom is angle of attack (2.14). The constant term in the wind model is set to $-3m/s$ in x and $-0.2m/s$ in z. This gives the approximately symmetric noise around α, while the added wind measurements gives a negative shift in air speed plus some added noise.

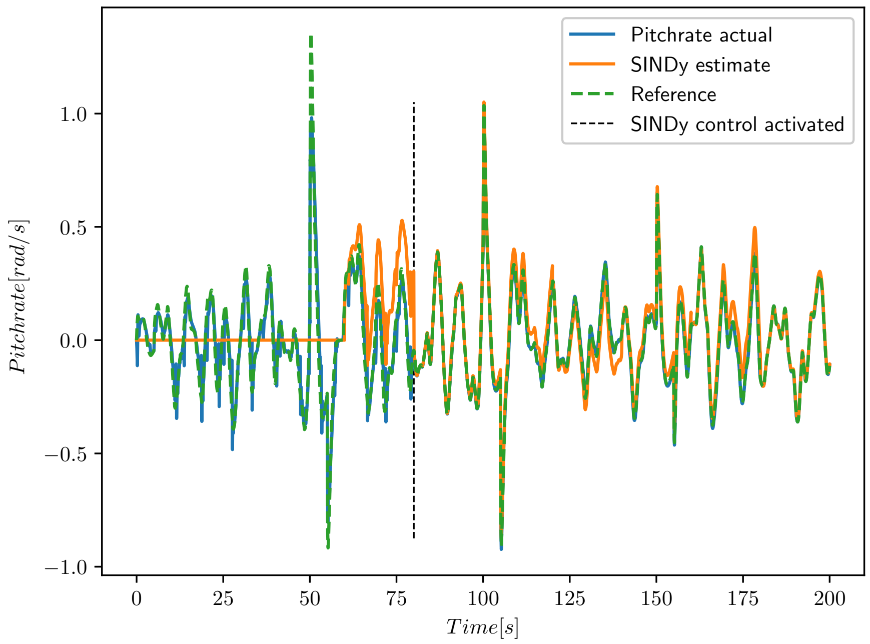

Figure 4.6: Added a reference to track. The first part of the model is using pitch rate control from a different controller.

measurements to the problem which can be seen in figure 4.5. Using a constant wind of $-3m/s$ in x and $-0.2m/s$ in z, and the Dryden turbulence, we get the resulting pitch rate tracking in figure 4.4. We can see it converges much slower, and diverges completely during a large time period. This was caused by SINDy using the error-metric not finding a better model during the time span, but towards the end, the algorithm converged to an expression similar to 4.1.

Given an accurate estimate of the pitch dynamics, we can apply it to pitch control.

In figure 4.6 we run the online SINDy algorithm to estimate the model. Once the model MSE on the test data is lower than a set amount, in this case

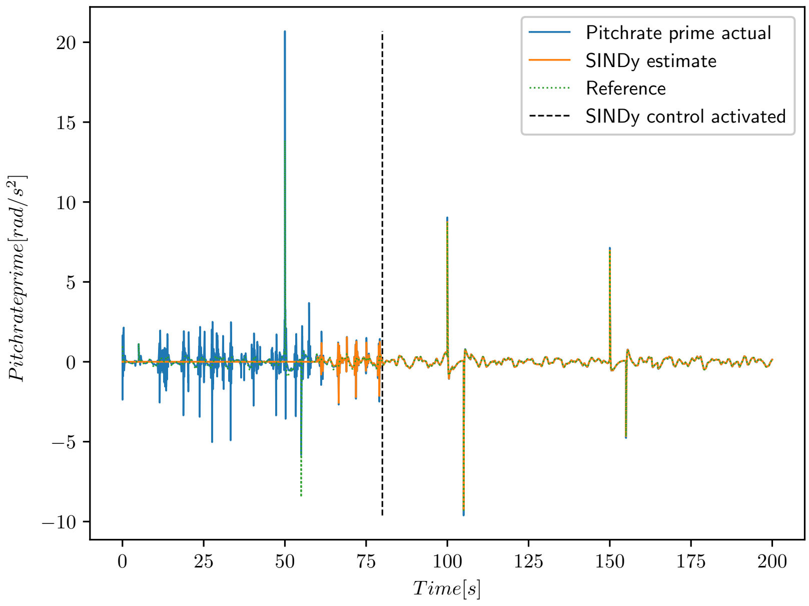

Figure 4.7: Clearer visible shift in where the SINDy model starts estimating, and where it is deemed good enough for control purposes.

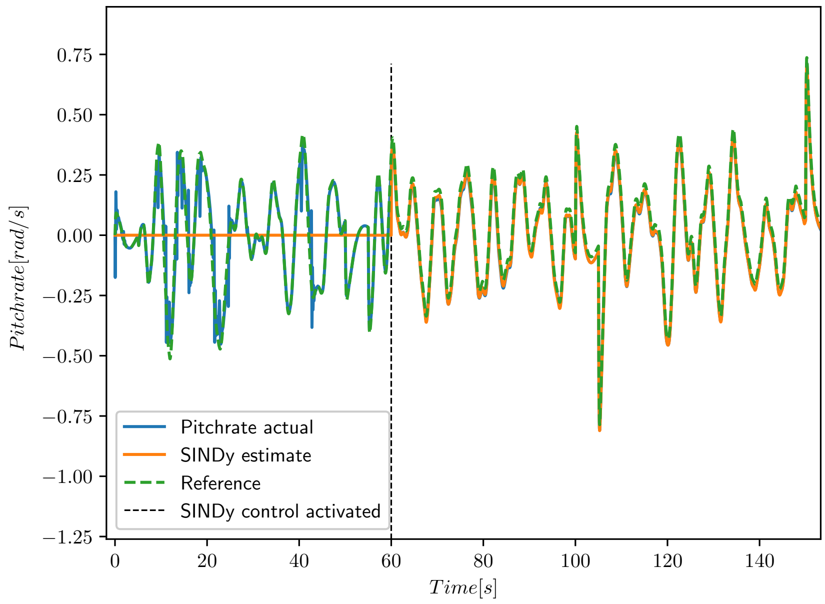

Figure 4.8: Pure open loop control using the found model. Given a perfect estimated model, the tracking should be perfect, but the used model had small numeric errors. We can see some small errors in tracking the peaks and crevices around the 100 second mark.

0.001, the model is used as the model in dynamic inversion with a low gain feedback of $K_p = 0.1$. In figure 4.7 the shifts are even easier to see. From $t \approx 60$ to $t \approx 80$, a SINDy model is found, but deemed too inaccurate for control by having a higher MSE than the threshold set at 0.01. At $t \approx 80$, a new model is found with MSE below 0.01, and the rest of the run uses it for the control scheme. Figure 4.8 shows the control algorithm running without any feedback or adaption, only open loop control.

4.2 Full icing conditions

The most effective way to apply SINDy for control during icing was not to apply the error-metric with a test set, length punishment and time punishment. What instead gave better results was to always take the latest model given by SINDy. When applying the found model to control, using the error-metric allows for a model based on older data to newer data, which could easily diverge. In figure 4.9 we see the compared models where one is updated with each new found model, and one is only updated when the error-metric is lower. Thus, for future SINDy models, the error-metric was not used.

In figure 4.10 we look closer at the performance of the SINDy model over time. While the found model usually gets the shape of the model correctly, it does occasionally introduce an offset in the pitch rate. For control purposes, this means we can not rely completely on the model found for control as we did in figure 4.6. If we add a simple proportional control term, where we multiply the pitch rate error with a small gain and include it in the controller, we can combat this small offset. The neural network controller also uses this proportional control term, but as we can see in figure 4.11 the necessary gain is different.

In the graph in figure 4.11 we compare the dynamic inversion scheme using

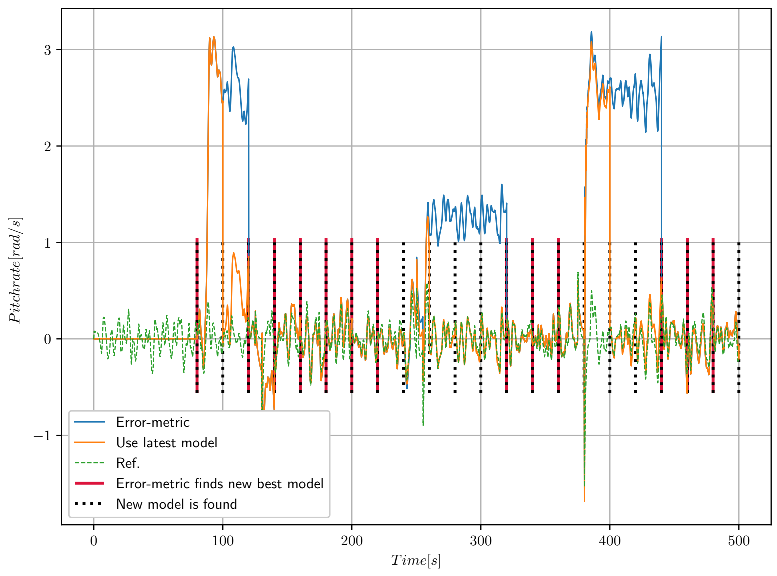

Figure 4.9: Comparing the estimation given by SINDy when using the error-metric to always taking the latest available model. The vertical lines indicate where the respective model is updated. Using the most recently found model performs better, as when the error-metric calculates an older model to be better, the performance is worse. This is clearly seen around the 300 second mark where one found model diverges, but is saved at the next found model. The error-metric still deems this model to have the lowest error and continues to diverge.

the neural network, and SINDy for control. The performance of both is quite similar, but the SINDy model reacts better to large jumps in reference. The major difference is the amount of gain applied. The neural network needs over 10 times the amount of gain to be able to compete with the SINDy model. It is preferable to have a smaller amount of proportional gain since the gain will amplify measurement noise to the system. It can also lead to non-physical control inputs which either can not be reached, or they can damage the actuator [61].

By turning down the gain used in the neural network-based controller, we run into a different issue. The neural network can sometimes diverge and output some high-frequency noisy control inputs, as was also mentioned in Dahl [1]. In figure 4.12 we see the raw output pitch rate of the UAV when using 0.5 gain. With this low gain, the network can occasionally output a high frequency, noisy signal to the control input which results in the pitch rate seen in the plot. To combat this, we can add a low-pass filter to the control input. By doing this we get the clean signal in the plot. However, a low-pass filter induces some delay to the system [45], which is visible in figure 4.13. Here we see a clear time delay on the controller using a low-pass filter.

4.2.1 Model found by SINDy

The benefit to SINDy is that we can actually discover the models of the underlying dynamics. However, the real world is complex with icing, atmospheric disturbances, magnetic disturbances, shifting center of mass, non rigid-body dynamics, and a multitude of other influences which are not modelled in a simulation or captured by a sparse equation. Even if all the influences could be captured in an equation, it would probably not be of low order or sparse. Which is why, an interesting application to SINDy is if it manages to capture the higher order dynamics using lower order terms.

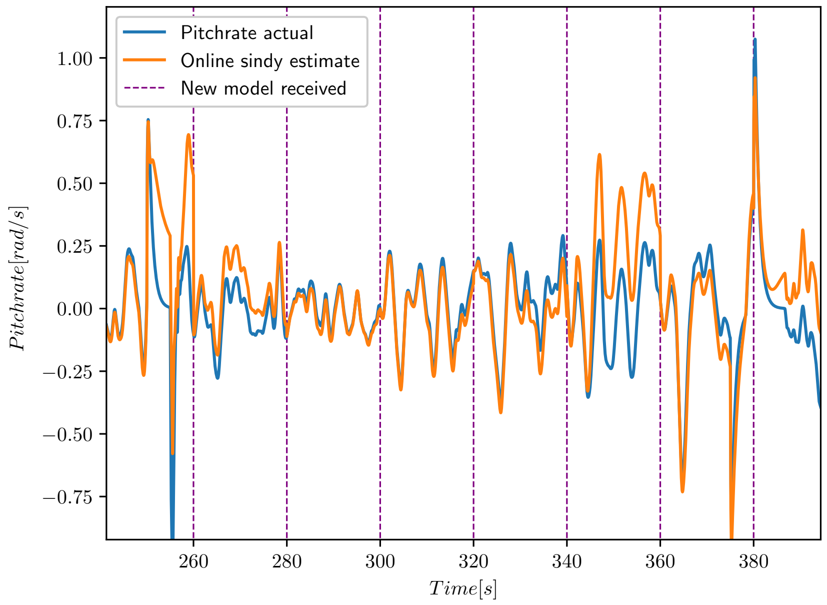

Figure 4.10: SINDy online model estimation. Here, the best of three models are chosen each iteration, but the model from the last iteration will not be used again if it has lower MSE.

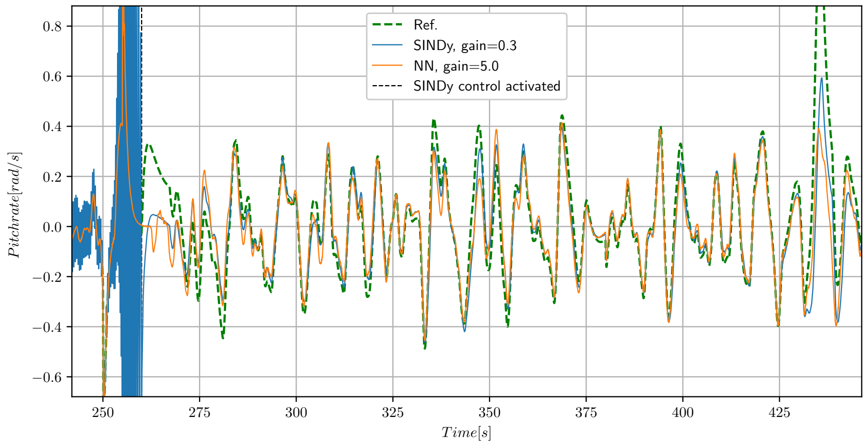

Figure 4.11: Comparing the control methods after the SINDy control algorithm is activated at $t = 260$. The tracking performance between the controllers is similar, but the neural network-based controller requires a magnitude higher proportional feedback gain than the SINDy based controller.

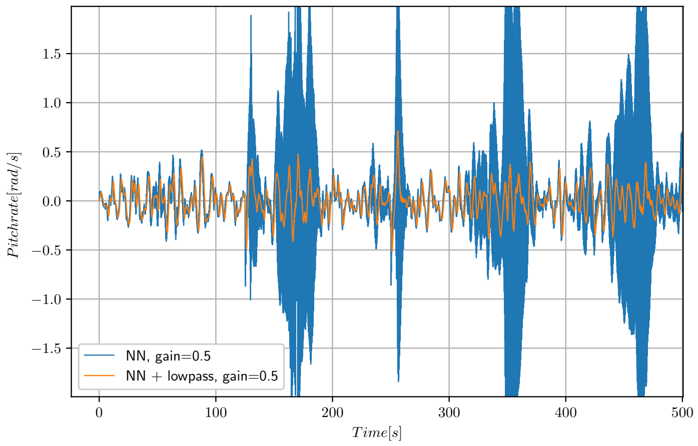

Figure 4.12: The raw output of the simulator when the neural network diverges, and the output when the control-input is low-pass filtered

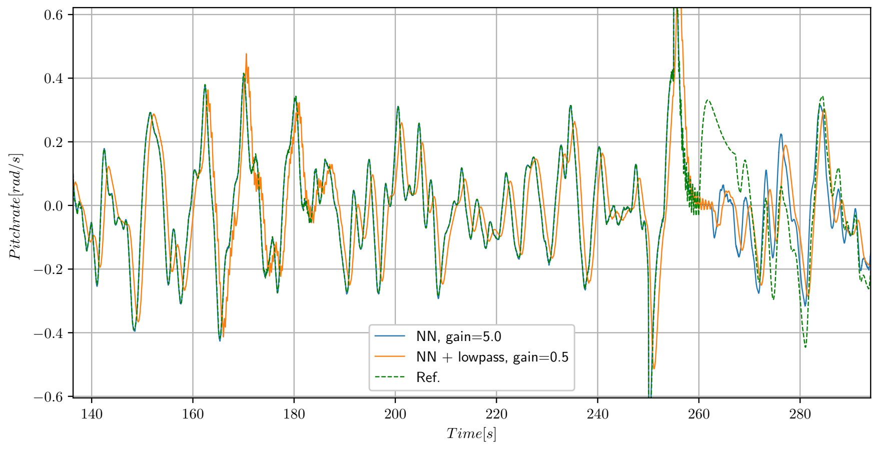

Figure 4.13: Comparing a high gain neural network controller with a low gain low-pass filtered control input neural network controller. Here we see the time delay introduced by the low-pass filter.

This has been tested in the icing example where the $CM(\alpha)$ coefficient function is interpolated as a quadratic function shown left in 3.2. This leads to a fourth order model for the pitch dynamics, and inserting the other coefficients in table 3.1, and the input augmentation from 3.4.2, with the values $n_1 = 100$, $n_2 = 10$, $n_3 = 1$ and $n_4 = 0.1$, we get the model in 4.2.

$$\dot{q} = -13.16\alpha^2 V_a^2 + 12.182\alpha V_a^2 - 1.24V_a^2 - 11.26qV_a - 30.72\delta_e V_a^2 \quad (4.2)$$

Increasing the order of the candidate library to the fourth order to hopefully find the model 4.2 never worked well. The combinatorics of a fourth-order candidate library makes it much larger than the third-order, so the best feasible way to capture the dynamics in a model was to use a third-order library and allow SINDy a less sparse model to compensate for the missing fourth order term. This third order model is used in all SINDy based results for icing, but with an added linear control input library to hinder the possibility of a squared control input model, as discussed in section 3.6.

$$
\begin{aligned}
\dot{q} = -458.4\alpha^2 q + 448.7\alpha^2\delta_e - 25008.7\alpha^2 V_a + 3817.1\alpha^2 \\
- 3.479\alpha q^2 - 9.86\alpha q\delta_e - 45.1\alpha q V_a \, 4.79\alpha q - 9.65\alpha\delta_e \\
+ 130.5\alpha V_a^2 + 319.7\alpha V_a - 49.1\alpha - 0.029 q\delta_e V_a + 2.29 q V_a^2 \\
- 0.53\delta_e V_a - 0.55 V_a^3 - 0.83 V_a^2 - 11.3 q V_a - 28.1\delta_e V_a^2 \quad (4.3)
\end{aligned}
$$

One model found is shown in 4.3. It is huge, but captures the dynamics well. If we look at the last three terms in the found model, we recognize the last three terms from 4.2, which are quite close to correct. Also, we have to keep in mind that these are the coefficients for the modified input, which means the coefficients including V_a and α terms will be substantially smaller in reality, and the terms including δ_e will be larger.

4.2.2 Height- and angle tracking results

The references shown in the plots above are generated through a PID-estimator made for tracking a height reference for the UAV. This is described further in [1]. The ultimate result we are interested in is how well the aircraft can fly. In the longitudinal model, we are then interested in how accurately and quickly the UAV can reach a certain height set point.

The results of the height tracking are shown in figure 4.14 and zoomed in closer in figure 4.15. The performance between the controllers is similar, but zoomed in, we notice the SINDy controller is a bit faster during the later part of the run. This neural network controller also utilized the higher gain value, but is still not reaching the set point faster.

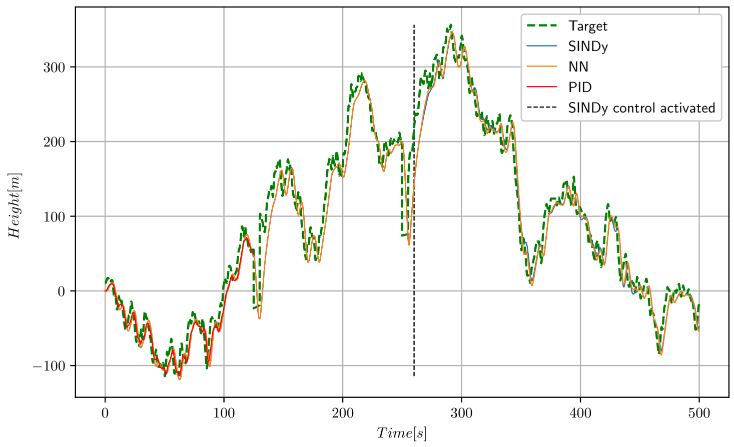

Figure 4.14: The performance on height tracking for the different controllers. The PID-controller diverges after $t = 125s$. The neural network controller and SINDy controller use a gain of 5.0 and 0.3, respectively.

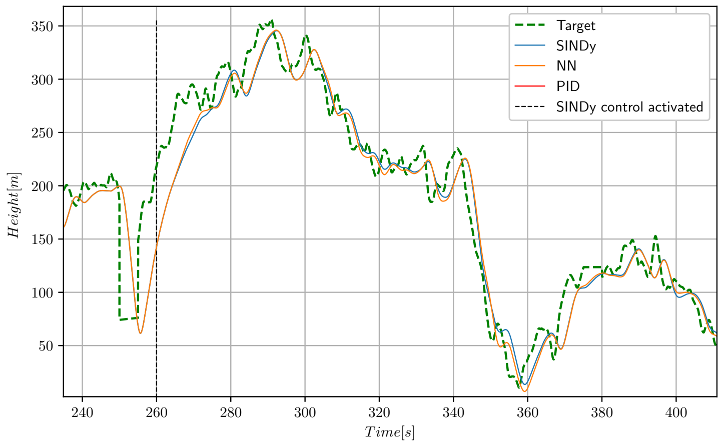

Figure 4.15: The performance on height tracking for the different controllers, zoomed in for higher detail.

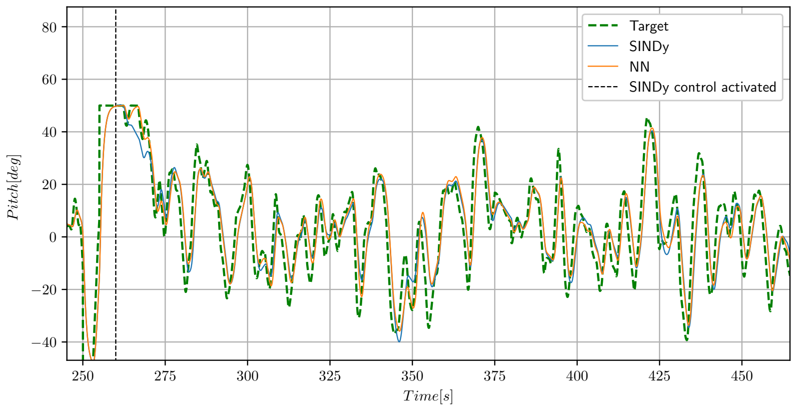

Figure 4.16: The performance on angle tracking for the different controllers.

The height set points are made into angle set points which are then made into a pitch rate reference.

The results in figure 4.16 show the resulting angle tracking. Results are very similar to the height set points, and the SINDy controller is faster and more accurate than the neural network controller. The results are similar to the height tracking, and this comes from the fact that these set points are generated from the set points of height, but with an added saturation of $\pm 50°$.

4.3 Transitioning conditions

An important aspect of handling icing on an aircraft is how the craft is controlled during the transition from a clean wing to an icy one. We have looked at the results for a clean wing and an icy wing separately, but transitioning from clean to icing poses difficulties for data-driven models. As we have seen in previous results, SINDy relies on a relatively large amount of data to converge with noisy measurements, but during icing, the older data becomes less and less representative and SINDy could end up discovering

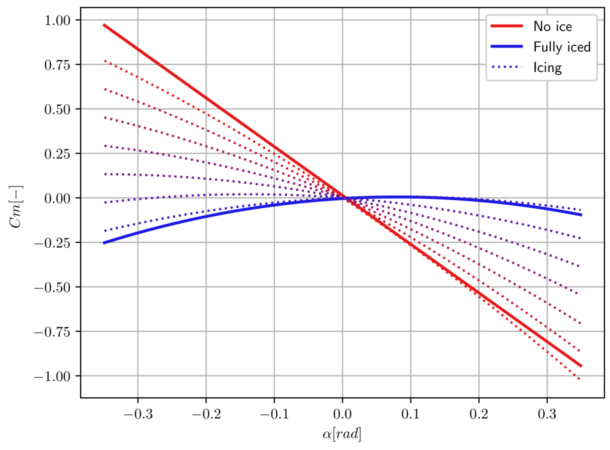

Figure 4.17: $C_m(\alpha)$ transitioning from a linear model (no icing) to the curve fit icing data from 3.2.

an incorrect model.

The transition model is generated by merging the linear model 2.21 with the icing model **??**, using the sigma merging function in 2.24. This gives a linear merging of the two functions which will increasingly become the icing model. Some sample data from the merging function can be seen in figure 4.17, with the color indicating how close it is to the clean- and icing model.

Applying the merging function in the simulation, and transitioning over a range of time spans, we get the following results. In figure 4.18 we see the SINDy algorithm applied to the transitioning data over four different time spans. The model is not fully the icy model until the end, but the dotted line marks where the transition started, so three of the models have a time period of just the linear model in the beginning. In the bottom row, we can clearly see the impact the transitioning data has on the model. It diverges completely once the pure linear model ends.

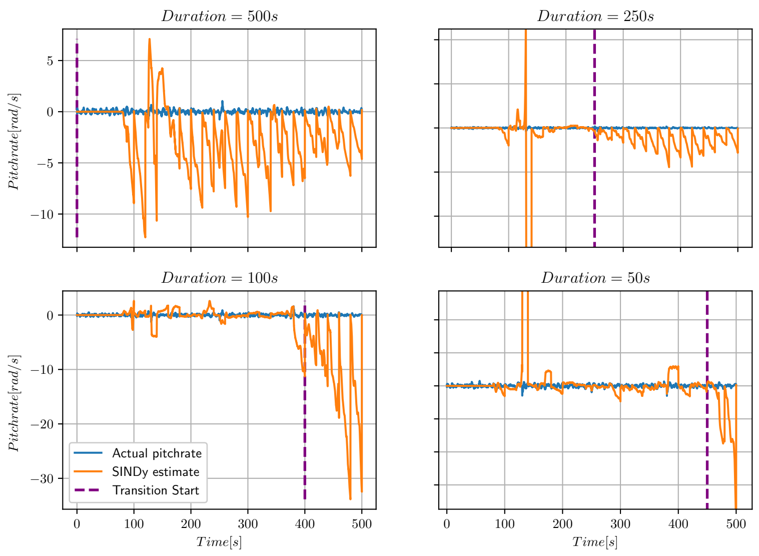

Figure 4.18: The performance of the SINDy algorithm on an airfoil transitioning from clean to icy. The dotted line represents when the transition from clean to icing starts, but no model is fully iced until $t = 500s$.

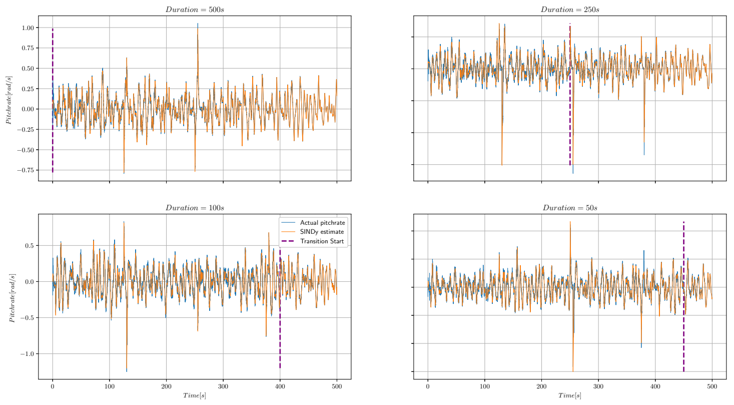

Figure 4.19: The performance of the neural network algorithm on an airfoil transitioning from clean to icy. The dotted line represents when the transition from clean to icing starts, but no model is fully iced until $t = 500s$.

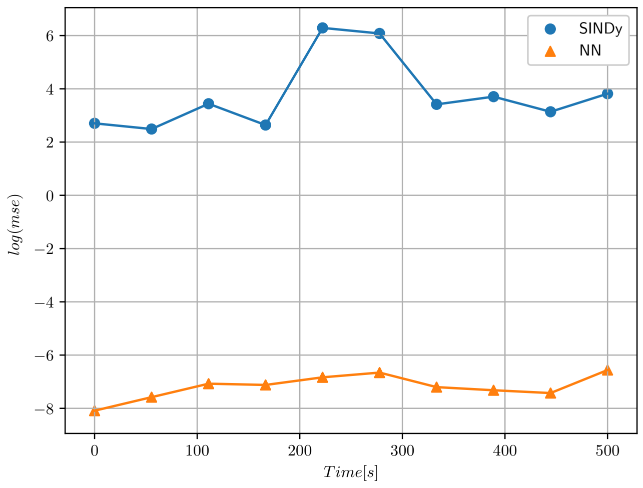

Figure 4.20: Log of MSE of the SINDy and neural network control schemes. Each data point is the resulting $\log(mse)$ of the tracking error to the reference with the transition to icing starting at t.

The plot in 4.19 shows the same scenario as in 4.18, but the neural network handles the change of model much better. It follows the reference similar to the results in the pure icing case. The difference in MSE between the SINDy model and the neural network model can be seen in figure 4.20, where the error is almost ten orders of magnitude larger for the SINDy model.

Finally, we have the PID controller. It performed surprisingly well, as can be seen in figure 4.21. In the pure icing case, the PID controller diverged at the first step in reference, but here we see it handles the transition case well. There is no clear visible difference between any of the four plots in the image, but the image does not represent the pure icing case. Meaning, the pitching moment used in the pure icing case seems to be on the limit of what the controller can handle. But transitioning to the ice model, it handles well.

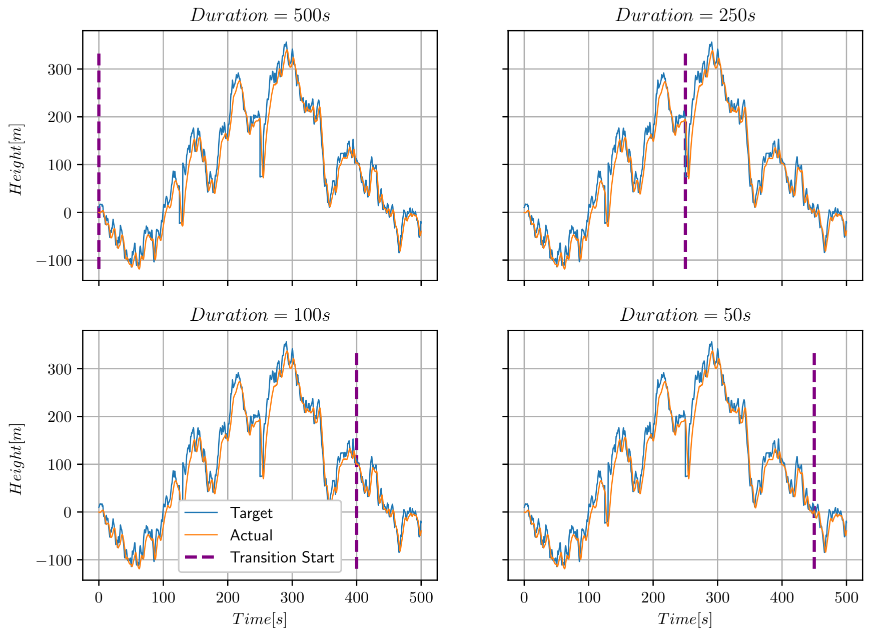

Figure 4.21: PID height tracking at different transition times.

Chapter 5

Discussion

5.1 SINDy

SINDy is proposed as a middle-ground between the pure black box machine learning models, and standard physical modelling. Ideally, the algorithm gives the best of both worlds: an explainable, simple model to describe the dynamics of a system without doing the work of manually modelling it. In this thesis, we applied the SINDy algorithm to control an UAV by dynamics inversion. The controller was applied to three different scenarios: a clean airfoil, a transitioning airfoil, and an icy airfoil.

The results when applied to the clean airfoil were promising. The underlying model was found, but it required some tuning of the input variables. Extra tuning variables are not desirable if they can be avoided, but with the underlying coefficients being orders of magnitude different, the STLSQ solver could not find a satisfactory sparse solution. Hence, the input modification was necessary. Another downside of having to augment the input to the solver is that this augmentation is difficult to know beforehand, and it is highly probable that it will need to change if the underlying dynamics change. As we saw in the transition case, once the underlying dynamics change, SINDy struggles to converge to something sensible. A small change like wind, it was able to handle perfectly, which makes it robust against noise. However, the larger shifts in dynamics were too much for a sensible result.

Another issue with using SINDy for this application, is the fact that the dynamics of an UAV is not that sparse. The simplest model of the pitch dynamics we used, is still a third-order model with four terms. STLSQ was able to find the third-order dynamics, but when the order increased to fourth, the pool of candidate functions became too large for a sparse result. However, we also saw that we managed to approximate the fourth order model with a third-order one by increasing the number of terms it had. A downside to this is that the resulting approximation to the fourth order model is overfitted to the data. We see this once the transition phase was initiated from the linear one. If the underlying dynamics were found, the small shift in pitch dynamics should not cause the system to diverge immediately, but an overfitted system will behave this way. When we keep in mind that a real-world UAV is not rigid, has a shifting center of pressure which is not at the center of mass, varying mass, pressure difference at different heights, and additional sources of uncertainty which are not modelled here, the poor results from the transition into icing do not suggest a stable control method for real-world application. An upside to the overfitted results is that we can have a way to inspect the resulting model and see it is overfitted by the number of terms and the size of the coefficients in front. This is not possible in a traditional neural network. As an example, in the overfitted model in 4.2, we have terms orders of magnitude higher than others, $-25008.7\alpha^2 V_a$ and $-0.53\delta_e V_a$ in the same equation points to a non-physical model and overfitting. By applying and extending this principle to some heuristic measurement of how probable it is have resulting model dynamics with a certain amount of coefficients and the span of coefficient values, there can be derived a measurement of how probable it is for a model to be overfitted. A model like the one hypothesised could be a useful metric and sanity test when applying SINDy to some system.

The problems with the SINDy algorithm can probably be solved by introducing new solvers, data-processing, and by adding more heuristic solutions to cover up its shortcomings. Thus, a SINDy-based control system in a real-world application would be possible, but this implies a much more exhaustive design of the system. The purpose of this examination of the SINDy algorithm applied to control was to explore it as a neural network, but with an explainable solution. However, this is not the case.

The upsides of SINDy, when compared to a neural network, are many. It can discover dynamics of unknown systems online while in use. It also covers up a big short-coming of a neural network, which is the stability proof. A sparse model of dynamics allows for the use of common frameworks for nonlinear stability analysis, like the common Lyapunov stability analysis for nonlinear systems[62].

SINDy has been applied with MPC-control before [20], [19], but dynamic inversion is much less computationally heavy. The Pysindy algorithm itself requires large computing power, but the controller does only simple matrix multiplication. If the controller is supposed to run on an embedded computer, online SINDy should not be used, but the simplicity of the dynamic inversion controller opens up another possibility. Recording data from the system using a simple controller first, analyzing the data from the run using SINDy, and then activating the dynamic inversion controller with the found model for another run. This gives a simple method for finding an appropriate model for a general nonlinear dynamic inversion controller. Even with an imperfect model, some standard lightweight feedback terms can be added to compensate for the modelling error from the SINDy model. But the main point is that the SINDy model will be much more accurate than a standard linearized one, and this will hopefully improve performance and reduce the additional control terms needed to apply a dynamic inversion controller. This method does however not work with the icing wings as shown in 4.18. By changing the underlying dynamics of the model, SINDy's found model is rendered less and less useful the more the dynamics change, and thus cannot be recommended to use on an icing UAV drastically reducing the amount of data needed for it to converge. We saw in the linear case that SINDy converged much faster without noisy measurements, so reducing the measurement noise by some pre-processing method could reduce the amount of data needed for convergence.

5.2 Neural network

The function of the network is fundamentally different from the SINDy algorithm. While both are used here to apply dynamic inversion, the neural network does not try to estimate the pitch dynamics, but tries to learn the error dynamics between the actual model and the linearized one. The performance of the network was high if it used a higher gain-term in the controller as well. Reducing it from 5 to 0.5 gave rise to some unwanted oscillations. It can be remedied by introducing some delay to the system, which is generally unwanted, but depending on the application, can work perfectly fine.

In most of the results presented, the SINDy algorithm gave better system control than the neural network solution, especially when considering the fact that it had a tenth of the gain in the controller. The SINDy controller has closer relation to common control methods. Thus it is preferable since it allows for further analysis using common tools, like those mentioned above in section 5.1. The neural network method did however excel in the most realistic simulation of icing performed in figure 4.19. Since the neural network is adapted at a high frequency and is not reliant on previous data, it will not be affected by the changing of underlying dynamics like the SINDy model is.

5.3 PID

PID-control is a common control scheme in most applications. The implementation used here is presented in Beard and McLain [7] and in short form in chapter 2 as the successive loop closure controller. PID control is not model-based, meaning we do not need a model of the system to apply it, it uses only error feedback to adjust the input to the system. The lack of a model simplifies the implementation, and makes the controller generally applicable to any system. When applied to the icing problem, it had mixed results. When the empirical model for the RG-15 airfoil during icing in figure 3.2 was used in the simulation, the PID controller would diverge at large jumps in tracking reference. The controller was tuned to function in non-icing conditions, and re-tuning it would probably give better results. However, the scenario is a realistic one: having a controller tuned to a standard mode of operation, but also needing to work during edge cases like icing. In this sense, the PID controller fails.

More can be done to improve the performance of the controller. A common technique for adapting the tuning the weights online during flight is gain scheduling [63]. This will improve the performance of the controller, but increases the complexity and amount of tuning needed. The upside for the successive loop closure PID controller was the transition case from no icing to icing. It managed to track the reference at the same performance as in the pure no icing case. This hints that the icing pitch dynamics are just on the edge of what the controller can handle. Seeing as that data was the worst-case scenario, a further developed PID controller should be able to handle it without increasing the complexity to a model-based controller.

A PID controller is commonly used as a first iteration in control. If it has unsatisfactory results, a more specialized controller can be applied. Comparing the results with the two other controllers that were tested here, we see the reason for this. The performance of the controller was excellent in the common scenario it was applied to. The no icing case is what most UAV's experience 99% of the time, which makes the PID controller a good option. In addition, the controller is extremely simple to implement and tune.

5.4 Conclusion

In this thesis, SINDy was implemented in a simulation of the longitudinal dynamics of an UAV. It was compared to the neural network adaptive controller implemented in Dahl [1], and a successive loop closure PID controller from Beard and McLain [7]. The different controllers were applied and compared on three different test cases for flight: non-icing airfoil, icing airfoil, and the transition between them.

Results show a promising use-case for SINDy in control during stable dynamics, but when introducing a large change in the underlying dynamics, the SINDy model diverged and became infeasible for control purposes. SINDy was posed as an explainable machine learning technique, and the hypothesis was that it would out-perform a neural network approach in both simplicity and performance. While it did outperform the neural network controller on stable dynamics (fully iced or no ice), it did not in the transition case. Additionally, it required some additional tuning parameters and coordinate shifts to converge. The adaptive technique utilizing a neural network handled all three test cases well, but with the caveat of utilizing a much higher gain for stable results. The occasional diverging and noisy control inputs produced by the neural network were present, which made the network need additional processing to be viable. The classic technique of PID control was also applied to compare the model-based techniques with a simple output feedback system. It performed well in both the non-icing case and the transition-case, but diverged during normal operation with a fully iced airfoil. But for nominal use, its performance was similar to the other controllers, while being much simpler to implement and tune. In conclusion, to control a UAV during icing, none of the controllers presented in their current forms are viable. They all have short-comings that require additional development to be viable options in a safety critical system, but the PID controller with some different tuning is the best option if the system does not require pitch rate tracking. It is reliable, simple to implement, and simple to understand.

5.5 Further work

Simulation is only an approximation of reality, not a replacement. To be able to say anything surely about these techniques, they need to be tested on a real UAV. This is common for all three methods.

For the SINDy control method. Further improvements could be made by implementing a different solver which does not use a threshold on the model coefficients to make the model sparse. The threshold-based approach poses problems for systems like the pitch dynamics used here, where the coefficients span a multiple orders of magnitude. Further work could also be done on implementing a better error metric for the algorithm. A fundamental problem with the implementation of SINDy for control used here is the large amount of data needed for the model to converge. If this could be reduced by some pre-processing, the model would update more frequently and the model could handle a change in dynamics better.

The neural network method worked well on all test cases. Still, there are improvements to be made. Ideally, the network would converge to a perfect model of the error dynamics, and no feedback gain would be needed. But the results do not show this. A large amount of gain (compared to the SINDy model) was needed for a stable result. This is indicative of the neural network not converging, and not being able to approximate the differential equation. Recurrent neural networks [64], long-short-term memory [65] and transformers [66] are, respectively, the next few steps in time-series forecasting. Each being more complex than the next, but with higher performance on time-based data. Further developing the network to one of these different architectures should increase the performance, but at the cost of complexity and computational power.

Bibliography

[1] J. Dahl. Artificial neural network-based model error estimation for aircraft control. 2022.

[2] Johannes W Oswald, Adriana Enache, Richard Hann, Gertjan Glabeke, and Thorsten Lutz. Uav icing: Experimental and numerical study of glaze ice performance penalties on an rg-15 airfoil. In *AIAA SCITECH 2022 Forum*, page 1976, 2022.

[3] Hazim Shakhatreh, Ahmad H. Sawalmeh, Ala Al-Fuqaha, Zuochao Dou, Eyad Almaita, Issa Khalil, Noor Shamsiah Othman, Abdallah Khreishah, and Mohsen Guizani. Unmanned aerial vehicles (uavs): A survey on civil applications and key research challenges. *IEEE Access*, 7:48572–48634, 2019. doi: 10.1109/ACCESS.2019.2909530.

[4] Ben C Bernstein, Cory A Wolff, and Frank McDonough. An inferred climatology of icing conditions aloft, including supercooled large drops. part i: Canada and the continental united states. *Journal of Applied Meteorology and Climatology*, 46(11):1857–1878, 2007.

[5] Ben C Bernstein and Christine Le Bot. An inferred climatology of icing conditions aloft, including supercooled large drops. part ii: Eu-

rope, asia, and the globe. *Journal of applied meteorology and climatology*, 48(8):1503–1526, 2009.

[6] Richard Hann. Atmospheric ice accretions, aerodynamic icing penalties, and ice protection systems on unmanned aerial vehicles. 2020.

[7] Randal W Beard and Timothy W McLain. *Small unmanned aircraft: Theory and practice*. Princeton university press, 2012.

[8] Witten D-Hastie T Tibshirani R James, G. *An Introduction to Statistical Learning with Applications in R, 2nd ed.* Springer, 2021.

[9] Rysdyk R.T. Calise, A.J. Nonlinear Adaptive Flight Control Using Neural Networks. 1998.

[10] Geethalakshmi S Lakshmikanth, Radhakant Padhi, John M Watkins, and James E Steck. Adaptive flight-control design using neural-network-aided optimal nonlinear dynamic inversion. *Journal of Aerospace Information Systems*, 11(11), 2014.

[11] Lara-Rosano-F. Chan C.W. Liu, X.J. Model-Reference Adaptive Control Based on Neurofuzzy Networks. 2004.

[12] Calise-A.J Johnson, E.N. Limited Authority Adaptive Flight Control for Reusable Launch Vehicles. 2003.

[13] Jonathan H Tu. *Dynamic mode decomposition: Theory and applications*. PhD thesis, Princeton University, 2013.

[14] Joshua L Proctor, Steven L Brunton, and J Nathan Kutz. Dynamic mode decomposition with control. *SIAM Journal on Applied Dynamical Systems*, 15(1):142–161, 2016.

[15] Jer-Nan Juang, Minh Phan, Lucas G Horta, and Richard W Longman. Identification of observer/kalman filter markov parameters-theory and

experiments. *Journal of Guidance, Control, and Dynamics*, 16(2): 320–329, 1993.

[16] Jer-Nan Juang and Richard S Pappa. An eigensystem realization algorithm for modal parameter identification and model reduction. *Journal of guidance, control, and dynamics*, 8(5):620–627, 1985.

[17] Proctor-J.L.-Kutz J.N. Brunton, S.L. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. 2016.

[18] Megan Morrison and J Nathan Kutz. Nonlinear control of networked dynamical systems. *IEEE Transactions on Network Science and Engineering*, 8(1):174–189, 2020.

[19] Eurika Kaiser, J Nathan Kutz, and Steven L Brunton. Sparse identification of nonlinear dynamics for model predictive control in the low-data limit. *Proceedings of the Royal Society A*, 474(2219):20180335, 2018.

[20] Urban Fasel, Eurika Kaiser, J Nathan Kutz, Bingni W Brunton, and Steven L Brunton. Sindy with control: A tutorial. In *2021 60th IEEE Conference on Decision and Control (CDC)*, pages 16–21. IEEE, 2021.

[21] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Sparse identification of nonlinear dynamics with control (sindyc). *IFAC-PapersOnLine*, 49(18):710–715, 2016.

[22] Brian M. de Silva, Kathleen Champion, Markus Quade, Jean-Christophe Loiseau, J. Nathan Kutz, and Steven L. Brunton. Pysindy: A python package for the sparse identification of nonlinear dynamics from data. 2020.

[23] Zongyao Yang, Xueying Yu, Simon Dedman, Massimiliano Rosso, Jingmin Zhu, Jiaqi Yang, Yuxiang Xia, Yichao Tian, Guangping Zhang, and Jingzhen Wang. Uav remote sensing applications in marine monitoring: Knowledge visualization and review. *Science of The Total Environment*, 2022.

[24] Cross-border monitoring and surveillance campaign deployed over the eastern baltic sea. URL `nordicunmanned.com`. Accessed: 2023-01-13.

[25] textronsystems.com. URL `textronsystems.com/products/aerosonde`. Accessed: 2023-01-13.

[26] Armyrecognition.com, 2022.

[27] Antoni Kopyt and Marcin Żugaj. Analysis of pilot interaction with the control adapting system for uav. *Journal of Aerospace Engineering*, 2020.

[28] Thor I Fossen. *Handbook of marine craft hydrodynamics and motion control*. Wiley, 2021.

[29] sbg-systems.com, 2022.

[30] wikipedia.org by user auawise, 2022.

[31] vectornav.com, 2022.

[32] wikimedia.org, 2022.

[33] Marcus McKenzie. Aero engineering 315, 2022.

[34] R. Vepa. *Flight Dynamics, Simulation, and Control for Rigid and Flexible Aircraft*. CRC Press, 2014.

[35] wikipedia.org by user antilived, 2006.

[36] Hugh L Dryden. A review of the statistical theory of turbulence. *Quarterly of Applied Mathematics*, 1(1):7–42, 1943.

[37] Airfoiltools. URL `airfoiltools.com`. Accessed: 2022-11-29.

[38] André. Xflr5. `http://www.xflr5.tech/xflr5.htm`, 2003–2021.

[39] Von Doenhoff-A.E. Stivers Jr L. Abott, I.A. Summary of Airfoil Data. 1945.

[40] N Landell-Mills. How airplanes generate lift is disputed. *Pre-Print DOI*, 10, 2019.

[41] Charles N Eastlake. An aerodynamicist's view of lift, bernoulli, and newton. *The Physics Teacher*, 40(3), 2002.

[42] flight-study.com, 2022.

[43] Bugajski D.-Hendrick R. Stein G. Dale, E. Dynamic Inversion: an evolving methodology for flight control design. 1994.

[44] Lewis-Frank L Johnson Eric N Stevens, Brian L. *Aircraft control and simulation; 3rd ed.* Wiley, 2016.

[45] Andresen-Trond Foss Bjarne A Balchen, Jens G. *Reguleringsteknikk; 6th ed.* Institutt for teknisk kybernetikk, NTNU, 2016.

[46] F Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of the Brain Mechanisms.* Spartan Books, 1962.

[47] R. Hecht-Nielsen. Theory of the backpropagation neural network. 1989.

[48] Stinchombe M.-White H. Hornik, K. Multilayer feedforward networks are universal approximators. 1989.

[49] Hinton G.E.-Williams R.J. Rumelhart, D.E. Learning representations by back-propagating errors. 1986.

[50] George Corliss and YF Chang. Solving ordinary differential equations using taylor series. *ACM Transactions on Mathematical Software (TOMS)*, 8(2):114–144, 1982.

[51] Miles Cranmer. Pysr: Fast parallelized symbolic regression in python/julia. 2020. URL `https://zenodo.org/record/4052869`.

[52] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1): 65–98, 2017. URL `https://doi.org/10.1137/141000671`.

[53] A. Battaglia P. Xu R. Cranmer K. Spergel D. Ho Shirley. Cranmer, M. Sanchez-Gonzalez. Discovering Symbolic Models from Deep Learning with Inductive Biases. 2020.

[54] Miles Cranmer Alvaro Sanchez-Gonzalez Peter Battaglia Rui Xu Kyle Cranmer David Spergel Shirley Ho. Discovering symbolic models from deep learning with inductive biases. 2020. URL `https://www.youtube.com/watch?v=HKJB0Bjo6tQ`.

[55] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa,

Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

[56] Stefan Schmidt and Daniel M Newman. Estimation of dynamic stability derivatives of a generic aircraft. In *Proceedings of the 17th Australasian Fluid Mechanics Conference*, 2010.

[57] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

[58] Kadierdan Kaheman, J Nathan Kutz, and Steven L Brunton. Sindy-pi: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics. *Proceedings of the Royal Society A*, 476(2242): 20200279, 2020.

[59] Peng Zheng, Travis Askham, Steven L Brunton, J Nathan Kutz, and Aleksandr Y Aravkin. A unified framework for sparse relaxed regularized regression: Sr3. *IEEE Access*, 7:1404–1423, 2018.

[60] Gregory F Lawler and Vlada Limic. *Random walk: a modern introduction*, volume 123. Cambridge University Press, 2010.

[61] Karl Johan Åström, Chang Chieh Hang, Patrik Persson, and Weng Khuen Ho. Towards intelligent pid control. *Automatica*, 28 (1):1–9, 1992.

[62] Hassan K Khalil. *Nonlinear systems; 3rd ed.* Prentice-Hall, 2002.

[63] Anders Helmersson. *Methods for robust gain scheduling*. PhD thesis, Linköping University Electronic Press, 1995.

[64] L.C. Jain L.R. Medsker. *Recurrent neural networks design and application*. CRC Press, 2001.

[65] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural computation*, 31(7):1235–1270, 2019.

[66] Francesco Giuliari, Irtiza Hasan, Marco Cristani, and Fabio Galasso. Transformer networks for trajectory forecasting. In *2020 25th international conference on pattern recognition (ICPR)*, pages 10335–10342. IEEE, 2021.