V&V Report

AE3212-II SVV Flight Dynamics Assignment

Group B24

D. Stiller	5253969	A. Schelling	5107245
J. Bron	5259754	M. Fetecau	5236789
T. de Kemp	5310474	L. Gonzalez	5242231

Delft University of Technology

TU Delft

V&V Report

AE3212-II SVV Flight Dynamics Assignment

by

Group B24

D. Stiller	5253969	A. Schelling	5107245
J. Bron	5259754	M. Fetecau	5236789
T. de Kemp	5310474	L. Gonzalez	5242231

TUDelft

Contents

1 Introduction

by Lorenzo Gonzalez

Engineers face numerous challenges when designing a new aircraft, and creating a numerical model is one of them. For this project, the team was assigned to develop and design a new business jet as well as verify and validate its flight dynamics. The business jet under development is a low-wing, fuselage-mounted engine jet aircraft. The team is responsible for analyzing and predicting the flight dynamics of the vehicle as well as its stability characteristics. The aim of this project is, in fact, to improve the group's comprehension of aircraft performance, stability, and control, as well as to practice the process of verification and validation for an aerospace engineering case study.

In the initial phase of the design, a numerical model of the aircraft that accurately predicts its static and dynamic stability properties is developed and analyzed. The simulation model uses the equation of motion for both symmetric and asymmetric flight and arranges them in a state space form. The result of this is a system of differential equations that are numerically solved by the model to find the control variables of the aircraft. After doing this, a verification of the code of the numerical model is performed. The aim of this process is to check the correctness of the code, making sure that the computational model accurately implements the analytical model and its solutions. Subsequently, the model's results are compared with experimental data in a validation process, which takes into account the potential for errors in the experimental data, in order to assess the accuracy of the numerical model.

The report is organized as follows. Firstly, the necessary flight dynamics theory for developing the analytic model is explored and analyzed in a dedicated chapter. This includes discussing the reference systems used and assembling the Equations of Motion (EOM) for symmetric and asymmetric flight. This is an essential starting point in order to describe the dynamic response of the aircraft to disturbances and control inputs. This chapter also presents all the assumptions made for all the different flight conditions and their impact on the final results. The next chapter presents the analysis of the numerical model. In this part of the report, all the aspects and results of the model are analyzed and discussed. This also includes the description of the measurements, as well as their implementation in the code. The next chapter details the verification process of the model. This is initially performed for small blocks of the code through unit testing. After that, the overall code is tested, and this takes the name of integrated testing. The final part of the report discusses the validation process of the model where, the experimental data taken from a flight test performed by the group are compared with a given reference data set results. To wrap up, the report includes a final discussion on the implementation of the model.

2

Model

In this chapter the physical model on which the flight simulation is based on is presented for both symmetric and asymmetric flight. The reference axis systems, alongside with the necessary corrections for changing systems, is presented in Section 2.1. Following the establishment of the stability reference frame, the Equations of Motion (EOM) for both symmetric and asymmetric flight are assembled in Section 2.2, which are accompanied by the necessary assumptions that led to the EOM in the first place. The derivation of the state space model based on the aforementioned EOM is presented in Section 2.3, alongside the assumptions used when assembling the numerical model. Lastly, Section 2.5 will describe the limitations to be expected from the numerical model.

2.1. Reference axis systems

by Lorenzo Gonzalez

The linear model that is going to be used within the numerical model was derived using the stability axis system, as described in Figure 2.1. This reference frame is a right-handed orthogonal system with the origin in the center of gravity of the aircraft. The Z axis points perpendicular to the symmetry plane of the aircraft, and the X axis is orientated along the direction of the airspeed. This is a common reference frame in aeronautics since it is widely used to express aerodynamic data derived from computational methods, wind tunnel testing, or full-flight data. Its most important feature is that it is fixed relative to the aircraft's stability characteristics leading to lift and drag being conveniently aligned with the axes.

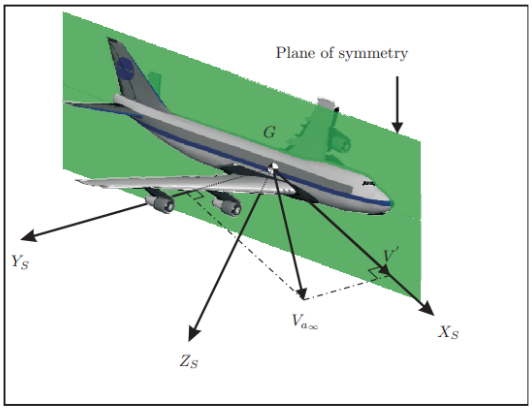


Figure 2.1: *Stability reference system*

Unfortunately, the measurements recorded during the flight test for the purposes of validation were taken considering a body axis system, which is displayed in Figure 2.2. This reference system is an orthogonal right-handed reference system, with its origin located in the centre of gravity of the aircraft. This is due to the constant gravity field assumption, that will be explained later on. On the symmetry plane of the aircraft the X_b-axis is present and faces forward along the fuselage of the aircraft. The Z_b-axis is positioned in the same symmetry plane and is directed downwards, while the Y_b-axis is perpendicular to the symmetry plane and faces right.

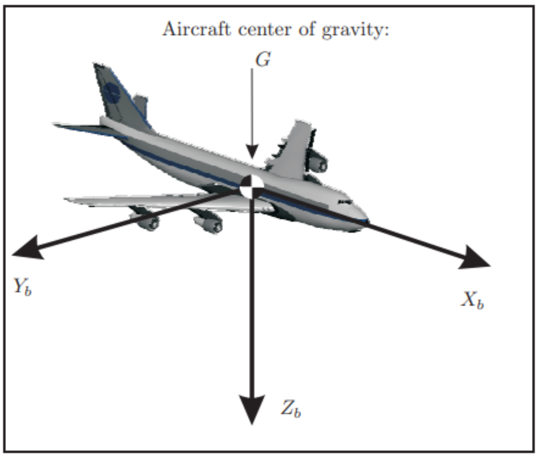


Figure 2.2: *Body reference system*

To transform the measurements from the body reference system to the stability reference system, a coordinate transformation using Euler angles needs to be performed. This is needed to couple the motions in each reference frame and to get the overall aircraft motion through time in one particular reference frame [1]. The transformation from the body-fixed reference frame (X_b, Y_b, Z_b) to the stability reference frame (X_s, Y_s, Z_s) can be performed using a sequence of three Euler angles: roll (ϕ), pitch (θ), and yaw (ψ). These represent the different rotation angles for the three axis (X, Y, Z, respectively). The matrix used to transform a given vector in the stability frame into the body-fixed one is the following:

$$\begin{bmatrix} \cos(\theta)\cos(\psi) & \cos(\theta)\sin(\psi) & -\sin(\theta) \\ \sin(\phi)\sin(\theta)\cos(\psi)-\cos(\phi)\sin(\psi) & \sin(\phi)\sin(\theta)\sin(\psi)+\cos(\phi)\cos(\psi) & \sin(\phi)\cos(\theta) \\ \cos(\phi)\sin(\theta)\cos(\psi)+\sin(\phi)\sin(\psi) & \cos(\phi)\sin(\theta)\sin(\psi)-\sin(\phi)\cos(\psi) & \cos(\phi)\cos(\theta) \end{bmatrix}$$

To use this transformation matrix, you first rotate the body-fixed frame around the Z_b-axis by the yaw angle ψ, then around the new Y-axis (Y_b after the first rotation) by the pitch angle θ, and finally around the new X-axis (Xs after the second rotation) by the roll angle ϕ.

After performing the transformation, the components of any vector expressed in the body-fixed reference frame can be expressed in the stability reference frame. This transformation is useful for analyzing the stability and control of an aircraft, as it allows for a simpler and more intuitive description of the aircraft's motion.

2.2. Analytical Model

by Lorenzo Gonzalez and Alexandra Schelling

In order to begin the development of the numerical model, it is imperative for the analytical model to be assembled first. The Equations of Motion (EOM) can be constructed using an analytical model and flight dynamics theory. For the purposes of this document, the derivation of the EOM is omitted. The final matrix form of the EOM are provided below.

2.2.1. Analytical model assumptions

To develop an analytical model being as accurate as possible, it is crucial to formulate all the main assumptions considered and their consequences. In this section, all the main assumptions used for the analytical model will be presented, as well as their effect on the desired result. Every assumption will be stated with a corresponding code type based on the following format: **ASS-TYPE-NUMBER**.

ASS-MAIN-01 The Earth is assumed not to rotate. By neglecting the angular velocity of the Earth (which is the normal Earth fixed reference frame rotation with respect to the inertial reference frame), the influence of the Coriolis acceleration and the centripetal one is neglected. The effects of this assumption are not relevant for measurements over a short time span. However, when the measurements last for several hours, errors become relevant. This assumption aims further to simplify the equations of motion of the aircraft.

ASS-MAIN-02 Flat Earth assumption. For the purposes of this assignment, the considered distances covered by the aircraft are assumed to be small. This leads to a simplification of the equations of motion since the curvature of the Earth is neglected. However, for larger distances, this assumption is not valid anymore and results in large changes and errors in the vehicle's dynamic state.

ASS-MAIN-03 The combination of **ASS-MAIN-01** and **ASS-MAIN-02** leads to a very powerful assumption being the fact that the Earth's radius is assumed to be infinite. This has implications for the kinematic position equations that simplify.

ASS-MAIN-04 The aircraft is assumed to be a rigid body. Given the structural stiffness of the aircraft frame, it is possible to neglect the small deflections encountered during flight.

ASS-MAIN-05 It is assumed the aircraft has a plane of symmetry. This allows the orientation of the body-fixed reference frame to be aligned with the principal axis of the vehicle in the symmetry plane. With this assumption, I_{xy} and I_{yz} value becomes zero.

ASS-MAIN-06 It is assumed that the aircraft follows a standard configuration. With this, it is meant that the aircraft has one main wing, a horizontal and vertical stabilizer, ailerons, elevators, and one rudder.

ASS-MAIN-07 No gusts and turbulences are assumed. With this assumption, wind is assumed to be undisturbed and the aircraft will not encounter gusts or turbulences and thus simplifying the flight case.

ASS-MAIN-08 The gravity field is assumed to be constant for all altitudes. This means that the value of $g = [0, 0, 9.81 m/s^2]$ for all cases.

ASS-MAIN-09 Resultant thrust lies in the symmetry plane. This means that the thrust only influences the symmetric aerodynamic forces in the X, Z directions and the symmetric aerodynamic moment M. Influence of the thrust causing moments around the Z axis is neglected.

ASS-MAIN-10 It is assumed that the symmetric and asymmetric flight tests can be completely decoupled into two distinct parts of the flight test as long as the deviations and disturbances remain small. This assumption provides the basis of the experimental set-up for the validation process.

ASS-MAIN-11 The aerodynamic coefficients are assumed to be linear. Making this assumption may have several consequences on the accuracy of aircraft performance prediction and control. This, in fact, may lead to an over- or under-estimation of the aircraft lift and drag coefficients. Moreover, this assumption may be responsible for limiting the accuracy of simulations and modeling tools used to design and analyze the aircraft performance.

ASS-MAIN-12 It is assumed that the on-board indicators present the calibrated airspeed.

ASS-MAIN-13 It is assumed that all flight is conducted in steady fashion. By this term it is meant a precise flight condition of the aircraft in which speed, altitude and angle of attack are kept constant leading to equilibrium. This assumption is crucial for the model that is going to be developed since this type of motion sets the basis for the description of different other more complicated conditions.

ASS-MAIN-14 It is assumed that the resultant thrust lies in the symmetry plane. With this assumption only the aerodynamic forces X, Z and the symmetric aerodynamic moment M are influenced.

ASS-MAIN-15 It is assumed that for steady flight, the sideslip angle is equal to zero. This is a valid assumption due to the watervane stability given by the vertical stabilizer. This assumption will disregard any contribution of wind gusts.

ASS-MAIN-16 A parabolic drag polar is assumed, thus neglecting the higher order C_L dependencies on the C_D.

2.2.2. Symmetric flight

In this section, the equations of motion for the symmetric flight are presented. By this condition, it is meant the state of an aircraft's motion that flies in a straight and fixed path, with its left and right wings generating equal amounts of lift and its left and right engines producing equal amounts of thrust. In this flight condition, the aircraft is balanced and stable and does not experience any rolling or yawing moments. In addition to the already mentioned assumptions, for this flight condition, some further assumptions will be stated. These are listed below:

ASS-SYMM-01 It is assumed that C_N and C_L are approximately equal. This assumption is derived from the small angle approximation applied on the Angle of Attack (AOA) α. For the purposes of this assignment and test flight, the AOA can be considered small during symmetric flight.

ASS-SYMM-02 All derivatives, lateral velocity and angular velocities are assumed to be zero. These conditions are fundamental for the simplifications that lead to symmetric aircraft.

ASS-SYMM-03 It is assumed that C_L follows a parabolic distribution with respect to C_D. This is fundamental for the derivation of the Oswald efficiency factor and the C_{D_0}

ASS-SYMM-04 The aircraft is not subject to any external forces or moments, such as engine failure or crosswind.

ASS-SYMM-05 The weight of the aircraft is evenly distributed between both wings. This is done especially to avoid dealing with unwanted moments that may be caused by the weight distribution of the aircraft.

ASS-SYMM-06 It is assumed rotating masses, such as the engines create no gyroscopic effects. These effects result from the conservation of angular momentum, which causes a rotating object to resist changes in the direction of its axis of rotation. In an aircraft, the engines are a significant source of rotating mass, and the gyroscopic effects they create can have an impact on the aircraft's handling and stability. On the other hand, if a small vehicle is used and its operations are not pushed to the limits, the effects of rotating masses may be small enough to be negligible.

These assumptions are necessary to simplify the analysis of the aircraft behaviour and its corresponding analytical model. The equations of motion for the symmetric flight case are presented in Equation (2.1), with the distinction that the term D_c is representing the non-dimensional symmetric flight derivative with respect to time. Accordingly, the only controllable variables used are the relative airspeed $\hat{u}$, normalized after the initial true airspeed V_{t_0}, the AOA α, the pitch angle θ, the pitch rate $q\bar{c}/V$. These equations can be used to analyze the aircraft's stability and control characteristics, such as its response to control inputs, the effect of aerodynamic forces and moments, and the aircraft's natural modes of motion.

$$\begin{bmatrix} C_{x_u} - 2\mu_c D_c & C_{x_\alpha} & C_{Z_0} & C_{X_q} \\ C_{Z_u} & C_{Z_\alpha} + (C_{Z_{\dot{\alpha}}} - 2\mu_c) D_c & -C_{X_0} & C_{Z_q} + 2\mu_c \\ 0 & 0 & -D_c & 1 \\ C_{m_u} & C_{m_\alpha} + C_{m_{\dot{\alpha}}} D_c & 0 & C_{m_q} - 2\mu_c K_{yy}^2 D_c \end{bmatrix} \begin{bmatrix} \hat{u} \\ \alpha \\ \theta \\ \frac{q\bar{c}}{V} \end{bmatrix} = \begin{bmatrix} -C_{X_\delta} \\ -C_{Z_\delta} \\ 0 \\ -C_{m_\delta} \end{bmatrix} \delta_e \tag{2.1}$$

2.2.3. Asymmetric flight

Asymmetric flight occurs when an aircraft experiences imbalanced aerodynamic forces and moments due to a mechanical failure or loss of power in one of its engines. This type of situation can also arise from sudden wind direction changes or turbulence in the airflow. As a consequence, the resulting aerodynamic forces cause significant changes in the aircraft's moments (yaw, pitch, roll), leading to an unstable and challenging-to-control situation. This being said, to have a complete aircraft model, it is crucial to consider this flight condition, too, in addition to the more conventional ones. As already done for the symmetric case, also for this condition, assumptions were made to simplify the calculations of the EOM.

ASS-ASYMM-01 The aircraft is assumed to be in stable flight conditions before the asymmetric event occurs. This comes from the already mentioned **ASS-MAIN-10** where for small deviations and disturbances both the symmetric and asymmetric flight conditions result uncoupled.

ASS-ASYMM-02 The aircraft is assumed to have symmetrical flight characteristics, meaning that the aerodynamic forces and moments on each side of the aircraft are roughly equal under normal flight conditions. This assumption is related to the previous one. The aircraft is assumed to flight in symmetric stable flight before the asymmetric forces start acting.

ASS-ASYMM-03 The aircraft is assumed to have sufficient control surfaces and engine power to counteract the asymmetric forces and moments. This comes from the fact that the aircraft to be modeled needs to be able to counteract these unstable effects and come back to stable conditions.

ASS-ASYMM-04 The aircraft's weight and balance are assumed to be within limits specified in the aircraft's loading diagram. This means that the asymmetric condition is not caused by any change in the weight distribution of the aircraft.

The EOM for the asymmetric flight are provided in Equation (2.2), where, similarly to the symmetric case, D_b is the non-dimensional asymmetric flight derivative with respect to time. As previously mentioned in **ASS-MAIN-10**, it is assumed that the symmetric and asymmetric flight cases can be decoupled completely. Accordingly, the controllable variables, in this case, are the side slip angle β, the roll angle ϕ, the roll rate $\frac{pb}{2V}$ and the yaw rate $\frac{rb}{2V}$. Similarly, the according deflections for the vertical stabilizer δ_r and the ailerons δ_a can be calculated in order to achieve steady flight.

$$\begin{aligned}
&\begin{bmatrix} C_{Y_\beta} + (C_{Y_{\dot\beta}} - 2\mu_b)D_b & C_L & C_{Y_p} & C_{Y_r} - 4\mu_b \\ 0 & -\frac{1}{2}D_b & 1 & 0 \\ C_{l_\beta} & 0 & C_{l_p} - 4\mu_b K_{xx}^2 D_b & C_{l_r} + 4\mu_b K_{xz} D_b \\ C_{n_\beta} + C_{n_{\dot\beta}} D_b & 0 & C_{n_p} + 4\mu_b K_{zz}^2 D_b & C_{n_r} - 4\mu_b K_{zz}^2 D_b \end{bmatrix} \begin{bmatrix} \beta \\ \phi \\ \frac{pb}{2V} \\ \frac{rb}{2V} \end{bmatrix} = \\
&= \begin{bmatrix} -C_{y_{\delta_a}} \\ 0 \\ -C_{l_{\delta_a}} \\ -C_{n_{\delta_a}} \end{bmatrix} \delta_a + \begin{bmatrix} -C_{y_{\delta_r}} \\ 0 \\ -C_{l_{\delta_r}} \\ -C_{n_{\delta_r}} \end{bmatrix} \delta_r
\end{aligned} \tag{2.2}$$

2.3. Numerical Model

by Lorenzo Gonzalez

A numerical model of an aircraft is a mathematical representation of its behavior, used for simulations, analysis, and design purposes. The numerical model used for this project implements a state space approach, where the EOMs are rewritten to follow the format presented in Equation (2.3). Note that for this model, the desired output of the simulation $\bar{y}$ is the state vector $\bar{x}$, with zero direct feedthrough. In order to achieve that, the D matrix will need to equal the null matrix, while the C matrix will equate to the identity matrix.

$$\begin{aligned} \dot{\bar{x}} &= A\bar{x} + B\bar{u} \\ \bar{y} &= C\bar{x} + D\bar{u} \end{aligned} \tag{2.3}$$

A simple approach to obtain these matrices is to rewrite the EOM under the form presented in Equation (2.4). Further matrix manipulation will yield to the state matrix A and the control matrix B.

$$C_1\dot{\bar{x}} + C_2\bar{x} + C_3\bar{u} = 0 \tag{2.4}$$

Note that, due to the linearization of the EOM about the steady flight conditions, and due to the change in reference systems, all simulation inputs, both the initial state of the system and the control input vectors, are taken as deviations from this steady state. Accordingly, the initial state vector will always be the null vector. In order to be able to compare the simulation results to the measured in-flight data, the conversion back to the body reference system will be necessary. This is done by adding the initial steady state values to each individual simulated state.

2.3.1. Symmetrical flight

For the simulation of maneuvers performed using symmetric flight, the state vector that is going to be used will follow the form $[\hat{u}, \alpha, \theta, q]^T$, while the control input vector will only be formed by the elevator deflection $[\delta_e]$. Using the description of the symmetric EOM in Equation (2.1), the intermediary C matrices can be computed as presented in Equation (2.5), Equation (2.6), and Equation (2.7), respectively.

$$C_1 = \begin{bmatrix} -2\mu_c \frac{\bar{c}}{V} & 0 & 0 & 0 \\ 0 & (C_{Z_{\dot{\alpha}}} - 2\mu_c)\frac{\bar{c}}{V} & 0 & 0 \\ 0 & 0 & -\frac{\bar{c}}{V} & 0 \\ 0 & C_{m_{\dot{\alpha}}}\frac{\bar{c}}{V} & 0 & -2\mu_c K_{yy}^2 (\frac{\bar{c}}{V})^2 \end{bmatrix} \tag{2.5}$$

$$C_2 = \begin{bmatrix} C_{X_u} & C_{X_\alpha} & C_{Z_0} & C_{X_q}\frac{\bar{c}}{V} \\ C_{Z_u} & C_{Z_\alpha} & -C_{X_0} & (C_{Z_q} + 2\mu_c)\frac{\bar{c}}{V} \\ 0 & 0 & 0 & \frac{\bar{c}}{V} \\ C_{m_u} & C_{m_\alpha} & 0 & C_{m_q}\frac{\bar{c}}{V} \end{bmatrix} \tag{2.6}$$

$$C_3 = \begin{bmatrix} C_{X_\delta} \\ C_{Z_\delta} \\ 0 \\ C_{m_\delta} \end{bmatrix} \tag{2.7}$$

2.3.2. Asymmetrical flight

In the case of asymmetric flight, the preferred state vector takes the form of $[\beta, \phi, p, r]^T$, while the control input vector will consist of the aileron and rudder deflections $[\delta_a, \delta_r]^T$. Following the expression of the asymmetric EOM in Equation (2.2), the intermediary C matrices can be computed as in Equation (2.8), Equation (2.9), and Equation (2.10), respectively.

$$C_1 = \begin{bmatrix} (C_{Y_{\dot{\beta}}} - 2\mu_b)\frac{b}{V} & 0 & 0 & 0 \\ 0 & -\frac{b}{2V} & 0 & 0 \\ 0 & 0 & -2\mu_b K_{xx}^2 (\frac{b}{V})^2 & 2\mu_b K_{xz} (\frac{b}{V})^2 \\ C_{n_{\dot{\beta}}}\frac{b}{V} & 0 & 2\mu_b K_{xz} (\frac{b}{V})^2 & -2\mu_b K_{zz}^2 (\frac{b}{V})^2 \end{bmatrix} \tag{2.8}$$

$$C_2 = \begin{bmatrix} C_{Y_\beta} & C_L & C_{Y_p}\frac{b}{2V} & (C_{Y_r} - 4\mu_b)\frac{b}{2V} \\ 0 & 0 & \frac{b}{2V} & 0 \\ C_{l_\beta} & 0 & C_{l_p}\frac{b}{2V} & C_{l_r}\frac{b}{2V} \\ C_{n_\beta} & 0 & C_{n_p}\frac{b}{2V} & C_{n_r}\frac{b}{2V} \end{bmatrix} \tag{2.9}$$

$$C_3 = \begin{bmatrix} C_{Y_{\delta_a}} & C_{Y_{\delta_r}} \\ 0 & 0 \\ C_{l_{\delta_a}} & C_{l_{\delta_r}} \\ C_{n_{\delta_a}} & C_{n_{\delta_r}} \end{bmatrix} \tag{2.10}$$

2.4. Center of gravity range

By Timo de Kemp and Alexandra Schelling

The location of the center of gravity is deduced from the Basic Empty Mass (BEM), the weight of the fuel and the payload (which include the passengers, the crew and their baggage). Its range depends on the fuel usage and the shifting of the passengers.

To calculate the location of the center of gravity (CG), the following equation is used:

$$x_{cg} = \frac{M_{BEM}X_{BEM} + M_{fuel}X_{fuel} + M_{payload}X_{payload}}{M_{BEM} + M_{fuel} + M_{payload}} \tag{2.11}$$

BEM and the payload mass are known, as they are weighted beforehand. The BEM location is known too, however $X_{payload}$ has to be determined yet. This value depends on where the baggage of the passengers is stashed and the allocation of the passengers. To compute $X_{payload}$ the following equation is used:

$$X_{payload} = \frac{\sum_i M_i X_i}{\sum_i M_i} \tag{2.12}$$

Where M_i is the weight of passenger or baggage i and X_i is the arm to passenger or baggage i (measured from the datum line). Finally the contribution of the fuel needs to be determined. In [2] a table with fuel mass and the mass moment due to the fuel is given. Plotting this resulted into a linear relationship, therefore linear regression is used to calculate the moments for masses not given in the table. From the linear regression, a slope (a) and an intersect (b) is determined. To determine $M_{fuel}X_{fuel}$ Equation (2.13) is used.

$$M_{fuel}X_{fuel} = a \cdot M_{fuel} + b \tag{2.13}$$

2.5. Limitations

by Lorenzo Gonzalez

Limitations are an important consideration in the development of any model, as they place restrictions on the accuracy and applicability of the model. Numerical models, including aircraft models, represent an abstraction of reality, and as a result, they have inherent limitations that must be taken into account when using them to make predictions or draw conclusions. While aircraft models can be highly efficient, it is essential to recognize and address their limitations when interpreting their results.

In this section of the report, we will outline the key limitations of the aircraft model that have been identified.

LIM-01 Linear aerodynamics assumption: the aerodynamic forces acting on the aircraft are assumed to be linearly related to the aircraft's state variables, such as angle of attack, sideslip angle, and control surface deflection. This have important effects on the accuracy of the final results and for this reason, it needs to be taken into account.

LIM-02 Simplifying assumptions: previously, several assumptions were made to simplify the formulations and calculations of the EOM. These are all listed in the previous sections. The combination of all of them may be translated into large deviations of the model from the real-life situation, therefore, every assumption must also present the effect that it has on the results.

LIM-03 It is not possible for the model to simulate stick-free controls without adding calculations that include the hinge moment. This is mainly due to the hinge moment being a fundamental aspect of how the aircraft control surfaces behave. If the model does not account for the hinge moment, it will not be able to simulate the behaviour of the control surfaces accurately, and therefore it will not be able to accurately simulate stick-free controls.

LIM-04 The model can't simulate the effect of the yaw damper for the Dutch roll manouver. This limitation comes from the fact that the model is not accurate enough to consider these types of controls. A yaw damper can either be mechanical or fly-by-wire control system, and its implementation in the computational model would have requested a significant amount of time and additional information from the flight test operators.

3

Analysis

In this chapter of the report, the analysis of the model data and results is presented. Initially the description of the stationary and dynamic measurements is presented. After that the process of loading and implementing them into the code is described. The reduction process is then shown, with the estimation of all the main aerodynamic parameters.

3.1. Description of stationary measurements

By Timo de Kemp

The stationary measurements are done to determine the missing aerodynamic and control derivatives. From the measurements, C_{L_α}, C_{D_0}, e, C_{m_δ}, and C_{m_α} are determined in Sections 3.6 and 3.8. This is done by measuring the following parameters when the aircraft was steady for different conditions.

- Pressure height
- Indicated airspeed
- Angle of attack
- Fuel flow (left and right engine)
- Fuel used
- Total air temperature
- Elevator deflection angle
- Elevator trim deflection angle
- Stick force

3.2. Description of dynamic measurements

By Timo de Kemp

For the eigenmotions that are used to validate the model, dynamic measurements are taken at 10 Hz. During the flight only the times of the start of the motion were written down to be able to find the motions in the data. The following parameters are used to validate and improve the simulation.

- Roll rate
- Yaw rate
- True airspeed
- Pitch rate

3.3. Data loading

By Dominik Stiller

To ensure that all analysis and simulation steps use the same and correct values, data loading is unified. Data loading consists of reading files, applying appropriate conversions, calculating derived quantities, and providing the data to the other modules of the software. Therefore, the other modules will always receive their input data from the data loading module. The data consists of time series from the Flight Test Instrumentation System (FTIS), time series from the post-flight data sheet (PFDS), and unstructured data such as timestamps and mass distribution. Time series will be stored as Pandas DataFrames, and unstructured data such as passenger masses as class attributes.

3.3.1. Loading of FTIS measurements

The FTIS measurements consist of 48 time series, timestamped and recorded at 20 Hz (although the provided reference data are sampled at 10 Hz). These are provided as MATLAB structs but can be loaded in Python.

These time series are first converted to SI units as described in Section 3.4. Doing this before the data touch any other parts of the program prevents hard-to-spot unit mistakes. Next, unnecessary columns are removed, and retained columns are renamed to clearer names. Finally, the time-dependent mass, which changes from the initial mass due to fuel flow, is added to the DataFrame.

3.3.2. Loading of post-flight data sheet

	time	h	cas	alpha	fuel_flow_left	fuel_flow_right	fuel_used	T_total	time_min	m	...	T_left	T_right
0	1317.666667	3058.160	119.179630	0.028507	0.079001	0.089563	226.720586	267.116667	21.961111	6582.163674	...	2922.306822	3530.407124
1	1410.833333	3057.144	109.319444	0.037234	0.073058	0.080303	239.874765	265.700000	23.513889	6569.009495	...	2706.849611	3122.269507
2	1494.500000	3057.652	98.687593	0.052069	0.066086	0.069530	250.534186	263.650000	24.908333	6558.350075	...	2443.392673	2640.535362
3	1586.000000	3056.636	83.254259	0.084648	0.053213	0.058316	259.832829	261.800000	26.433333	6549.051431	...	1868.781162	2169.729663
4	1716.666667	3080.004	66.877778	0.139335	0.051386	0.056132	272.533416	259.700000	28.611111	6536.350845	...	1938.822910	2233.977169
5	1821.666667	3103.372	58.818148	0.188205	0.045485	0.051197	282.210053	258.550000	30.361111	6526.674207	...	1660.679908	2023.098918

6 rows × 28 columns

Figure 3.1: *Example of processed PFDS time series (stationary $C_L - C_D$ measurements) loaded into a Pandas DataFrame.*

The PFDS contains a mix of time series and unstructured data, provided as Excel sheet with consistent field positions. The PFDS time series are processed similarly to those of the FTIS; an example is shown in Figure 3.1. From the masses of pilots, observers, block fuel and basic empty weight, the ramp mass m_{ramp} is calculated. Similarly, the CG position is found as described in Section 2.4.

Next, the time series for stationary measurements are extracted and organized, then converted to SI units. Then, the values from all PFDS (in our case six) of one flight test are averaged and compared against each other to detect outliers. This helped us to find some manual errors early on.

Finally, time-dependent derived columns are added to the DataFrame. This includes the mass, thermodynamic quantities, and thrust, which depends on altitude and Mach number. To integrate thrust calculation, which is rather involved, seamlessly into our program, we translated the given Excel sheet into Python and made the code available to other groups. Also, velocity and elevator deflection are reduced to standard conditions as described in Section 3.5.

3.4. Unit conversion

By Joachim Bron and Timo de Kemp

Some of the measured quantities are in Non-SI units. For consistency in the calculations, these will be converted to SI units. Table 3.1 shows the non-SI units (left column) and their corresponding SI units (middle column). The method to convert the non-SI into their SI equivalent is shown in the last column.

3.5. Reduction of measurements to standard conditions

By Joachim Bron and Timo de Kemp

To compare the data for different conditions, standard conditions are introduced. Uncontrollable and adjustable variables have to be corrected to standard values. The fully controllable variables are set as parameters. The standard variables were given in [2] to be as presented in Table 3.2.

Table 3.1: *Non-SI to SI units conversion*

Non-SI	SI	Conversion formula
lbs	kg	[kg] = 0.45359237*[lbs]
kts	m/s	[m/s] = 1852/3600*[kts]
ft	m	[m] = 0.3048*[ft]
°C	K	[K] = [°C] + 273.15
deg	rad	[rad] = pi/180*[deg]
lbs/hr	kg/s	[kg/s] = 0.45359237/3600*[lbs/hr]
psi	Pa	[Pa] = 6894.757*[psi]
ft/min	m/s	[m/s] = 0.3048/60*[ft/min]

Table 3.2: *Notation and numerical value of standard values*

Parameter	Notation	Standard value
Standard aircraft mass	W_S	60500 N
Standard engine fuel flow per engine	$\dot{m}_{f_s}$	0.048 kg/s
Standard air density	ρ_0	1.225 kg/m^3

Reduced equivalent airspeed

To calculate the reduced equivalent airspeed Equation (3.1) can be used the standard weight, W_s from Table 3.2. The weight can be calculated as explained in Section 3.6.

$$\tilde{V}_e = V_e\sqrt{\frac{W_s}{W}} \tag{3.1}$$

To determine the equivalent airspeed the true airspeed(V_t) and the air density need to be determined as can be seen in Equation (3.2).

$$V_e = V_t\sqrt{\frac{\rho}{\rho_0}} \tag{3.2}$$

To calculate the V_t, the mach number and speed of sound at the condition have to be determined, as $V_t = M \cdot a$. The speed of sound can be determined as usual using $a = \sqrt{\gamma RT}$. As T needs to be the static temperature for this equation, the measured total air temperature, T_m has to be corrected for ram rise using Equation (3.3).

$$T = \frac{T_m}{1 + \frac{\gamma-1}{2}M^2} \tag{3.3}$$

For this equation and to calculate V_t the mach number is needed, the mach number can be calculated using Equation (3.4).

$$M = \sqrt{\frac{2}{\gamma-1}\left[\left(1+\frac{p_0}{p}\left\{\left(1+\frac{\gamma-1}{2\gamma}\frac{\rho_0}{\rho}V_c^2\right)^{\frac{\gamma-1}{\gamma}}-1\right\}\right)^{\frac{\gamma-1}{\gamma}}-1\right]} \tag{3.4}$$

The only unknown left in this equation is the pressure as this is a function of the pressure altitude, Equation (3.5), which is measured this can also be determined.

$$p = p_0\left[1+\frac{\lambda h_p}{T_0}\right]^{-\frac{g_0}{\lambda R}} \tag{3.5}$$

Reduced stick force

The Reduced stick force can be calculated from the stick force which is measured and the standard weight and weight from Equation (3.6)

$$F^*_{e_{aer}} = F_{e_{aer}} \cdot \frac{W_s}{W} \tag{3.6}$$

Reduced elevator deflection

The reduced elevator deflection can be calculated from Equation (3.7), the measured elevator deflection, the previously determined C_{m_δ}, the given $C_{m_{T_c}}$ and T_{c_s} and T_c, still have to be determined.

$$\delta^*_{e_{eq}} = \delta_{e_{eqmeas}} - \frac{1}{C_{m_\delta}} C_{m_{T_c}} \left(T_{c_s} - T_c\right) \tag{3.7}$$

To determine T_{c_s} and T_c Equation (3.8) is used, for T_c the real conditions are used. However for T_{c_s} the standard mass flow is used to calculate the thrust in Equation (3.8).

$$T_c = \frac{T}{\frac{1}{2}\rho V^2 S} \tag{3.8}$$

3.6. $C_L-\alpha$, $C_D-\alpha$ and ${C_L}^2-C_D$ plots and estimation of C_{L_α}, C_{D_0} and e

By Joachim Bron, Timo de Kemp and Alexandra Schelling

Plotting of $C_L-\alpha$ and estimation of C_{L_α}

For every time step, C_L can be computed using Equation 3.9

$$C_L = \frac{W}{\frac{1}{2}\rho V^2 S} = C_{L_\alpha}(\alpha - \alpha_0) \tag{3.9}$$

The mass m can be obtained from the mass as a function of time given by Equation 3.10. Furthermore a filled in mass balance sheet of the flight can be found in Appendix A.1.

$$m(t_1) = m_{ramp} - \int_0^{t_1} \dot{m}_{f_{l+r}} \mathrm{d}t \tag{3.10}$$

To compute the dynamic pressure q, we can use the equivalent airspeed V_e and the ISA air density at sea level ρ_0 since we have

$$q = \frac{1}{2}\rho V^2 = \frac{1}{2}\rho_0 V_e^2 \approx \frac{1}{2}\rho_0 V_c^2 \tag{3.11}$$

in our case, $V_c \approx V_e$, so we can use the calibrated airspeed V_c (given as a measurement) instead of the equivalent airspeed V_e (see [2] on for more details). The wing surface S is also given. Using these measurements, C_L can be computed for every time step. This can then be combined with α, also measured for every time step, and a scatter plot can be made of C_L vs α.

To estimate C_{L_α}, the scatter plot of $C_L-\alpha$ is investigated visually and pre-processed in such a way as only to include the linear region. This is done to have the best possible approximation of C_{L_α}. Then, using only the data from the linear part (and assuming C_{L_α} is constant and C_L vs α linear for α not close to stall), a best-fit line is found using linear regression. It was chosen over simple linear regression due to its robustness. The slope of this best-fit line is our best estimate for C_{L_α}. To fully characterize the $C_L-\alpha$, the α_0 and C_{L_0} parameters can also be extracted from the line of best fit.
The plots can be seen for the data from the flight in Figure 3.2a and for the reference data in Figure 3.3a. From these graphs, it can be seen that the typical linear behaviour also holds for the conditions flown.

Plotting of $C_D - C_L$ and estimation of C_{D_0} and e

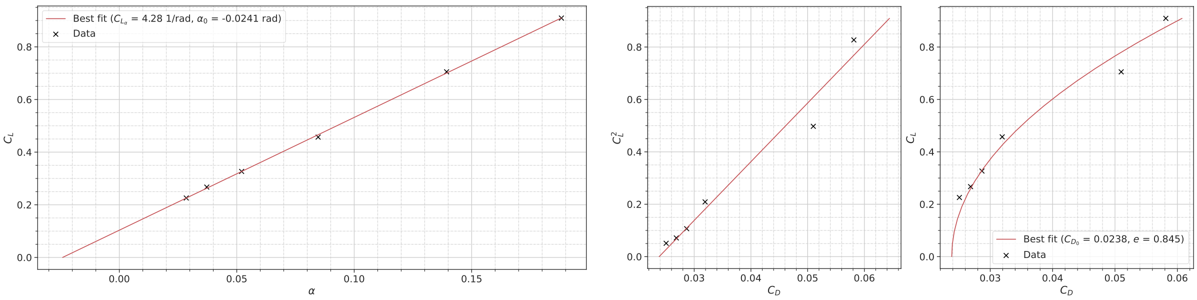
Similarly to C_L, C_D can be computed using Equation 3.12

$$C_D = \frac{T}{\frac{1}{2}\rho V^2 S} = C_{D_0} + \frac{C_L^2}{\pi A e} \tag{3.12}$$

Here again we use q and S in the same way as described for C_L. In this case however, we also require the thrust T. This is obtained from an executable as the computation is quite involved.

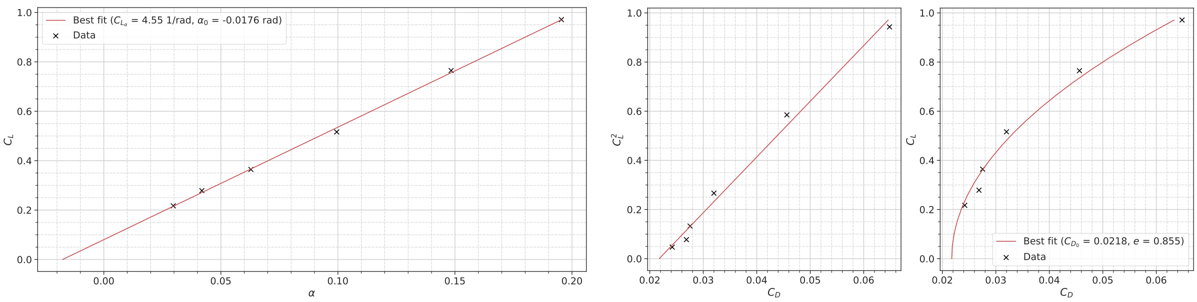
Furthermore, for the improvement of the numerical model we also require the C_{D_0} and e parameters. To obtain these, we note that by plotting C_D vs C_L^2, the value of C_D at $C_L = 0$ corresponds to C_{D_0} (y-intercept) and that the slope corresponds to $\frac{1}{\pi A e}$. If we find the slope, e can easily be found by rearranging, since A is known. To estimate C_{D_0} and e, we again use linear regression and find the line of best fit. Once C_{D_0} and e are known, they are used to plot the $C_L - C_D$ curve.

In Figures 3.2b and 3.3b, B24 and reference data respectively, $C_D - C_L^2$ and $C_D - C_L$ can be seen. From the $C_D - C_L^2$ curve the expected linear behavior can be seen, the data is not exactly linear, a parabolic drag polar is assumed, ASS-MAIN-16. From the $C_D - C_L$ curve it can be seen that the calculation for C_{D_0} and e is a good estimation as the curve follows the data nicely.



(a) *Lift curve $C_L - \alpha$.* **(b)** *Drag polar $C_D - C_L$.*

Figure 3.2: *Lift curve and drag polar from the B24 dataset.*



(a) *Lift curve $C_L - \alpha$.* **(b)** *Drag polar $C_D - C_L$.*

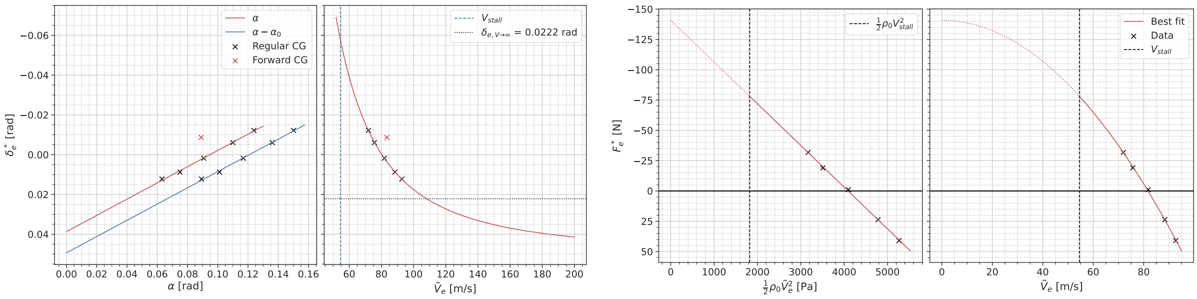
Figure 3.3: *Lift curve and drag polar from the reference dataset.*

3.7. Elevator trim curve and elevator control force curve

By Joachim Bron and Alexandra Schelling

For the investigation of the static stability in stick-free and stick-fixed conditions, the elevator trim and elevator control force curves are needed. From the elevator trim curve $\delta_e^* - \tilde{V}_e$, C_{m_α} can be estimated, and C_{m_δ} is calculated from the shift in cg, as explained in Section 3.8. The (reduced) elevator trim curve plot $\delta_e^* - \tilde{V}_e$ is shown in Figure 3.4a. The (reduced) elevator control force curve $F_e^* - \tilde{V}_e$ is shown in Figure 3.4b.

By qualitatively analyzing the plots, multiple observations can be made. First of all, for Figure 3.4a, it can clearly be seen from the curve fits that there is a linear relation between δ_e^* and α, and an inverse quadratic



(a) *Elevator trim curve $\delta_e^* - \tilde{V}_e$.*

(b) *Elevator control force curve $F_e^* - \tilde{V}_e$.*

Figure 3.4: *Elevator $\delta_e^* - \tilde{V}_e$ and $F_e^* - \tilde{V}_e$ curves (clean configuration + normal flap angle) from the B24 dataset.*

relation between δ_e^* and $\tilde{V}_e$. Furthermore, since the δ_e^*-α curve has negative slope (i.e. $\frac{d\delta_e}{d\alpha} < 0$), and since $C_{m_{\delta_e}}$ is always negative, it follows that $C_{m_\alpha} < 0$ and the aircraft is statically stable. The slope of the δ_e^*-$\tilde{V}_e$ is always positive, which further confirms the observation that $C_{m_\alpha} < 0$ and thus the static longitudinal stability of our aircraft. For both of these plots, a δ_e point is also plotted for the same AOA and V_e but different c.g. location. In our case, the c.g. location was moved forward by moving one of the passengers forward. It can be seen that for δ_e^*-α, δ_e becomes more negative for a forward-moving center of gravity. This makes sense since a forward-moving cg increases the stability margin. In other words, for a more forward c.g., the force from the elevator required for stability decreases. Similar reasoning can be applied to δ_e^*-$\tilde{V}_e$. For δ_e^*-$\tilde{V}_e$, it can also be seen that the calculated asymptote for $V_e \to \infty$ does not correspond to the graphical asymptote of the fitted curve. The calculated asymptote was based on data given to the group, and thus it would be difficult to judge its accuracy. There is, however, a significant difference in their values, the cause of which is not immediately apparent and would need to be further investigated.

For the force control curve given in Figure 3.4b, it can be seen that the F_e^*-$\frac{1}{2}\rho_0\tilde{V}_e^2$ exhibits a linear character and F_e^*-$\tilde{V}_e$ exhibits a quadratic character, as expected. The intercept at $V_e^* = 0$ being negative indicates that the aircraft is stick-free and statically stable. Since $\left(\frac{dF_e}{dV_e}\right)_{F_e=0} > 0$ at the trim speed, the aircraft can be said to exhibit elevator control force stability in this configuration.

3.8. Estimation of C_{m_δ} and C_{m_α}

By Timo de Kemp

To estimate C_{m_δ} and C_{m_α}, during the flight test, the center of gravity is moved by moving one of the observers to a different position. The elevator deflection angle is recorded for each of the tests. This change in elevator deflection will cause a change in the moment coefficient, as this relation is assumed to be linear the following relation is true.

$$\Delta C_m = C_{m_\delta} \cdot \Delta\delta_e \tag{3.13}$$

Furthermore, the shift in center of gravity also changes the moment created around this point due to the normal force. The change due to this change in moment is only due to the change in moment arm as the weight and thus normal force is assumed to be unchanged between these two measurements. The change in moment can be expressed as follows.

$$\Delta C_m = C_N \frac{\Delta cg}{\bar{c}} \tag{3.14}$$

From Equation (3.13) and Equation (3.14), Equation (3.15) can be calculated as these moment changes have to cancel each other out to remain in moment equilibrium.

$$C_{m_\delta} = -\frac{1}{\Delta\delta_e} \frac{W}{\frac{1}{2}\rho V^2 S} \frac{\Delta x_{cg}}{\bar{c}} \tag{3.15}$$

The following equation can be derived from the required deflection angle for equilibrium. This equation shows that the change of elevator deflection due to a change in the angle of attack is a function of C_{m_δ} and C_{m_α}. This makes sense as moment equilibrium has to be obtained after a change in one of these variables.

$$\frac{\mathrm{d}\delta_e}{\mathrm{d}\alpha} = -\frac{1}{C_{m_\delta}} C_{m_\alpha} \tag{3.16}$$

which must be < 0 for stability. As both the elevator deflection and the angle of attack can be easily measured during flight and is known for all times. It is assumed that the relationship is linear, therefore, linear regression is put and the value of $\frac{\mathrm{d}\delta_e}{\mathrm{d}\alpha}$ is calculated. With this and the value for C_{m_δ}, C_{m_α} can be calculated.

3.9. Estimation of τ, P and $T_{1/2}$

By Alexandra Schelling

Using the eigenvalues calculated before for the different motions performed during the flight test, estimations can be made for the time constant τ, the period P, and the time to damp to half amplitude $T_{1/2}$. For the aperiodic roll and the spiral, which are the aperiodic motions, the time constant will be estimated. The period and the time to damp to half amplitude will be estimated for the Phugoid, the Dutch Roll and the short period.

For the aperiodic motions, such as the aperiodic roll and the spiral, the eigenvalues are real. Using the relationship in Equation (3.17):

$$x(t+\tau) = \frac{1}{e} x(t) \tag{3.17}$$

The following equation can be used to make an estimation for the time constant:

$$\tau = -\frac{1}{\lambda_c} \frac{\bar{c}}{V} \tag{3.18}$$

The Phugoid, Dutch Roll and short period are oscillatory motions and thus have complex eigenvalues. For the oscillatory motions the time to damp to half amplitude and the period is estimated.

For the symmetric motions, the Phugoid and the short period, $T_{1/2}$ and P can be estimated by the following equations, Equation (3.19) and Equation (3.20) respectively:

$$T_{1/2} = \frac{\ln \frac{1}{2}}{\xi_c} \frac{\bar{c}}{V} \tag{3.19}$$

$$P = \frac{2\pi}{\eta_c} \frac{\bar{c}}{V} \tag{3.20}$$

For the Dutch Roll, which is an asymmetric motion, Equation (3.19) and Equation (3.20) can be used, however, read 'b' for 'c' and '$\bar{c}$'. Note that our state vector is dimensional, thus we get eigenvalues λ instead of λ_c and multiplying by $\bar{c}/V$ is not necessary.

3.10. Summary of results

By Dominik Stiller

Table 3.3: *Estimated aerodynamic parameters.*

	ref_2023	B24	% Abs. Diff. B24 w.r.t. ref_2023
C_{L_α}	4.55	4.28	5.93
α_0	-0.0176	-0.0241	36.93
C_{D_0}	0.0218	0.0238	9.17
C_{m_α}	-0.542	-0.546	0.74
C_{m_δ}	-1.208	-1.34	10.93
e	0.855	0.845	1.17

The primary goal of the analysis of stationary measurements is the estimation of aerodynamic parameters as described in Sections 3.6 and 3.8. The results are shown in Table 3.3 for the reference dataset and our own. The corresponding lift curve and drag polar of the B24 dataset are shown in Figure 3.2, while those of the reference dataset are shown in Figure 3.3.

All estimates match in magnitude and sign, which gives confidence in their correctness. The parameters C_{m_α} and e, in particular, are virtually identical. C_{m_δ} is also remarkably similar, given that only two measurements (forward and aft CG) were used. The biggest differences are visible in C_{L_α}, C_{D_0} and α_0. Fortunately, α_0 is not used in the simulation. Comparing the fits in Figures 3.2 and 3.3, both have low residuals with respect to the measured data. To find better estimates, more measurement series should be taken and averaged.

Table 3.4: *Characteristics of eigenmotions. Cells with "–" could not be determined from data.*

	ref_2023		B24	
Periodic	P [s]	$T_{1/2}$ [s]	P [s]	$T_{1/2}$ [s]
Phugoid	41	79	41	69
Short period	–	–	–	–
Dutch roll	3.4	4.6	3.3	2.7
Dutch roll YD	3.1	2.6	3	2.6
Aperiodic	τ [s]		τ [s]	
Aperiodic roll	–		–	
Spiral	-51		–	

The characteristics of the eigenmotion were determined visually from the FTIS measurements and are shown in Table 3.4. Phugoid and Dutch roll (without and with yaw damper) characteristics agree reasonably well between our data and the reference data. The Dutch roll in our data is much more damped (shorter $T_{1/2}$) than the reference flight. However, due to the exponential nature of damping, this estimate is very sensitive, which also explains the difference in $T_{1/2}$ for the phugoid.

Note that we could not identify the characteristics for short period and the aperiodic modes. The short period has no clear periodic–damped response, while the aperiodic modes show no exponential growth from which a time constant could be determined.

Table 3.5: *Eigenvalues for the asymmetric and symmetric A matrices using data from our B24 flight test.*

Symmetric	**Eigenvalue**	P **[s]**	$T_{1/2}$ **[s]**	τ **[s]**
Phugoid	-0.0053+0.14j	46	132	
Short period	-1.3+1.9j	3.3	0.52	
Asymmetric	**Eigenvalue**	P **[s]**	$T_{1/2}$ **[s]**	τ **[s]**
Dutch roll	-0.28+2.1j	3.0	2.5	
Aperiodic roll	-4.8			0.21
Spiral	0.005			-192

The eigenvalues for the symmetric and asymmetric state space matrix A is given in Table 3.5. For each eigenmotion, the matrix is created with proper initial conditions taken from the FTIS data, therefore these do not correspond to eigenvalues of the same instance of A.

The periods of phugoid and Dutch roll match closely to what we observed in Table 3.4. However, the Dutch roll is damped much more than in actual flight, while the phugoid is much less damped. For the spiral, the time constant derived from the eigenvalues is much longer, and for the other modes we have no data for comparison. As expected, the spiral is the only unstable eigenmode but has a very large time constant, which means that the pilot has enough time to react, or will not even notice the aircraft is in a spiral.

4
Verification

This chapter of the report outlines the various steps and tests conducted to verify the flight dynamics model. Verification is a crucial aspect in the design of any structure as it ensures that the computational model is accurate and consistent with the mathematical one. The verification process began with a thorough static code analysis of the program, followed by the creation of several unit tests to verify specific parts of the code. An integrated testing process was then conducted to evaluate the code's overall functionality in different aspects. Additionally, an investigation into the effect of eigenfrequencies on various aspects was conducted.

4.1. Unit tests

by Lorenzo Gonzalez

In this section of the report, the unit tests for the functions implemented in the model's code are presented. Performing these types of tests is crucial to verify the correctness of the model and improve the code's quality. Unit testing represents, in fact, a very efficient way to identify bugs and errors already in the early development process of the model. Moreover, unit tests can help facilitate code maintenance by making it easier to identify areas of code that are affected by changes. Especially in big projects, this can help reduce the risk of introducing new bugs or breaking existing functionality when making changes to the code. All the unit tests performed can be found in the code Appendix A.

4.2. Integrated testing

By Timo de Kemp

After performing the unit tests for the individual parts of the code, it is crucial to test the code as a whole. This is done by means of integrated tests. Integrated testing is a type of software testing that evaluates the behaviour of multiple software components when they are integrated with each other. The individual components that have already passed unit tests are combined and tested as a group to ensure they function correctly together. To test the simulation as a whole, the response of the selected aircraft states to disturbances is investigated. The response will be plotted and visually verified.

VER-INT-1 Positive pulse input for δ_e should result in a positive pulse response of α. In Figure 4.1 the initial response to the pulse in elevator deflection can be seen, this is a negative reaction as expected. Furthermore the recovery can also be seen as the value over time will go to zero. In the plot it seems to be increasing this is an oscillation that is very small and will further die out as time goes to infinity.

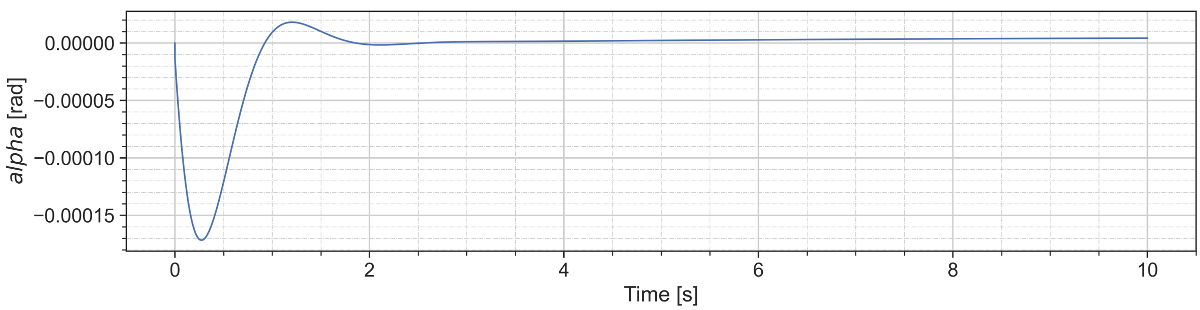
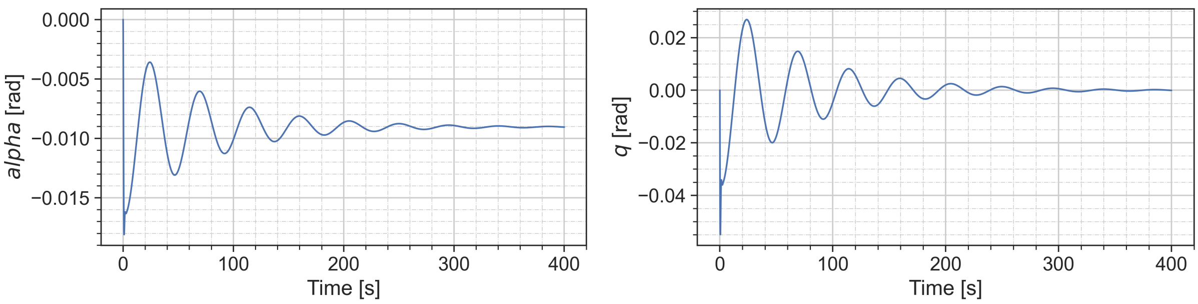


Figure 4.1: *α versus time, for a pulse input of 0.01 radian on the elevator deflection.*

VER-INT-2 Positive step input for δ_e should result in a negative step response of α, which should give a negative q transient which approaches 0 for $t \to \infty$. In Figure 4.2a the step in angle of attack can be seen after initial oscillations, as expected to be a negative step. Furthermore in Figure 4.2b the initial oscillations can also be seen which decay and tend to 0 for large times. The oscillations for the step input take a lot longer to die out compared to the pulse input which is due to the aircraft having to find a new equilibrium state for the step input.



(a) *α versus time curve.* **(b)** *q versus time curve.*

Figure 4.2: *Responses due to a step input of 0.01 radian elevator deflection*

VER-INT-3 Positive pulse input for δ_r should result in a negative response of r (yaw rate). Figure 4.3 shows an initial negative response to the yaw rate as expected. After some oscillations in yaw rate related to the dutch roll the yaw rate tends to zero.

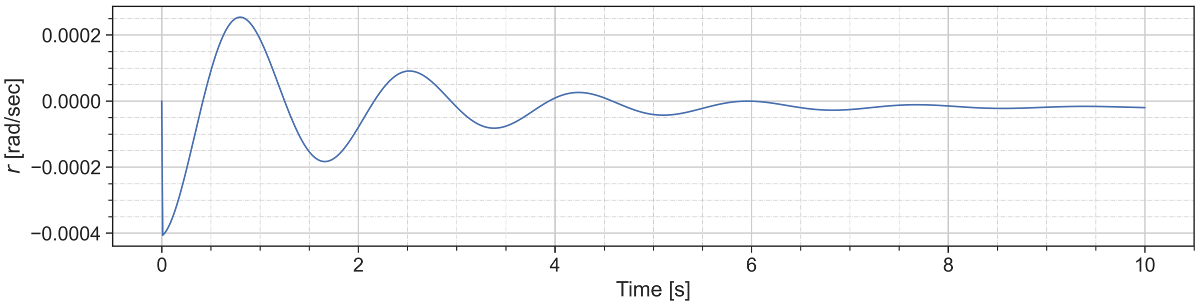


Figure 4.3: *r versus time, for a pulse input of 0.01 radian on the rudder deflection.*

VER-INT-4 Positive pulse input for δ_a should result in a negative response of p (roll rate), going to a constant. In Figure 4.4 the initial response to the input is negative as expected. Furthermore it can be seen that the roll rate goes to a constant close to or equal to zero.

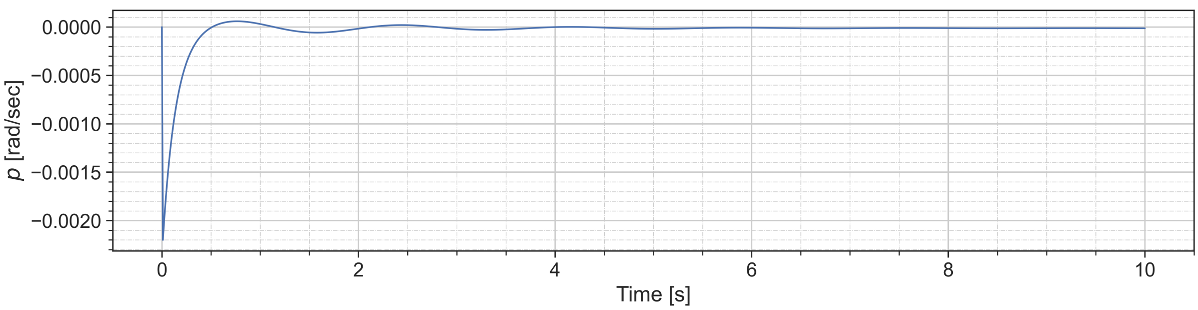


Figure 4.4: *p versus time, for a pulse input of 0.01 radian on the aileron deflection.*

VER-INT-5 Positive step input for δ_a should result in a negative response of p (roll rate), increasing in magnitude over time. Figure 4.5 shows the roll rate due to a step aileron input. As can be seen the roll rate generates a negative response that is ever increasing in magnitude, this is as expected from theory.

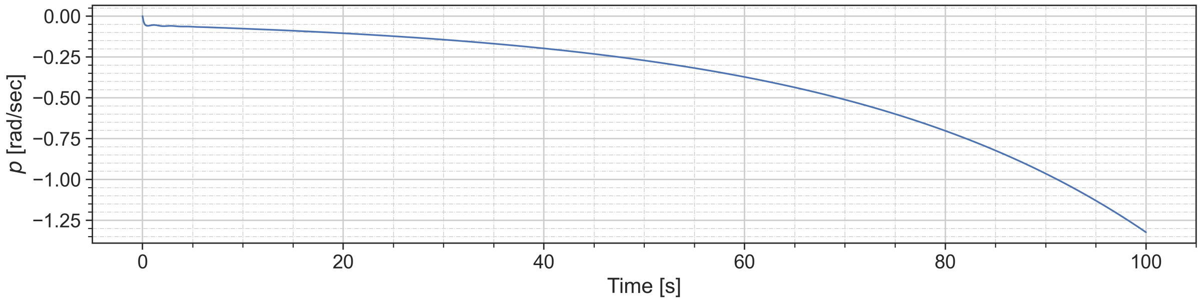
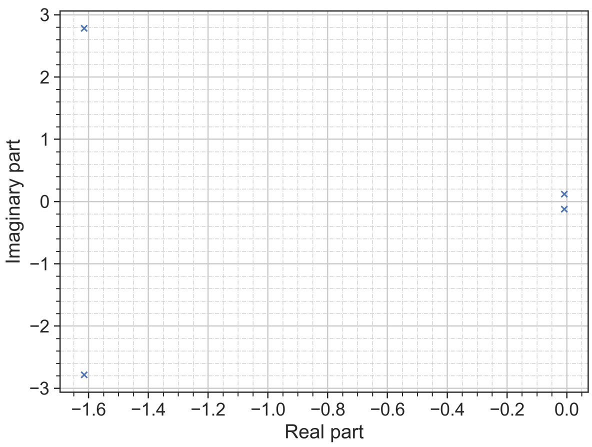


Figure 4.5: *r versus time, for a step input of 0.01 radian on the aileron deflection.*

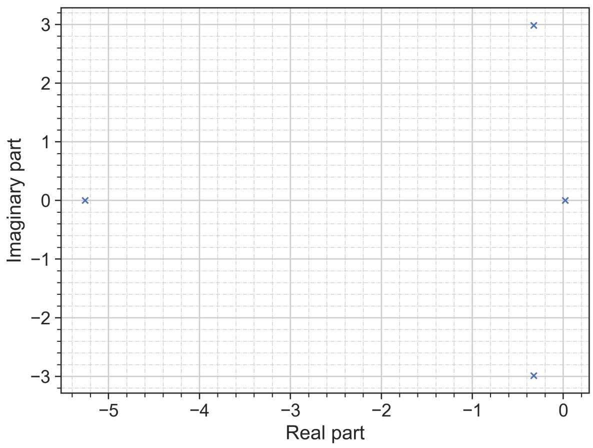
4.3. Eigenvalues

By Timo de Kemp & Mihai Fetecau

To test if the characteristic eigenmodes of an airplane also come from the simulation the eigenvalues of the A matrix in the state space model are plotted in Figure 4.6. To determine the eigenvalues a mass of 4500 kg, V_0 of 150 m/s, ρ of 0.8, and for asymmetric a C_L of 0.8. Furthermore stability derivatives and control derivatives given in [2].



(a) *Symmetric eigenvalues*



(b) *Asymmetric eigenvalues*

Figure 4.6: *Eigenvalues for the A matrix in state space model for symmetric and asymmetric motion*

In Figure 4.6a the eigenvalues for the symmetric modes can be seen, 2 conjugate pairs can be identified, one very close to the imaginary axis, the motion related to these eigenvalues has low damping and is known as the phugoid. Furthermore there is another conjugate pair that has high damping and short period(large imaginary part). This motion is thus known as the short period.

In Figure 4.6b one conjugate pair can be seen, this oscillation with short period and reasonable damping is known as the dutch roll. Furthermore two real eigenvalues can be seen, one to the right of the imaginary axis and therefore unstable and one far left of the imaginary axis and therefore very stable. The unstable eigenmotion, corresponds to the spiral and the stable one is called a-periodic roll.

Further testing of the correctness of the model regarding eigenvalues can be performed by comparing the eigenvalues calculated by the numerical model with the ones derived from the EOM, the so called analytical model. By assuming the form of the state vector following the exponential expression $A_x e^{\lambda t}$, it is possible to use substitution within the EOMs in order to obtain a symbolically defined characteristic equation, which can be further reduced into a fourth order polynomial of the form presented in Equation (4.1). Note that the subscript from λ_n stands for non-dimensional, as each eigenvalue is corrected for the use of non-dimensional time, done accordingly to the type of EOM, either symmetric or asymmetric.

$$A\lambda_n^4 + B\lambda_n^3 + C\lambda_n^2 + D\lambda_n + E = 0 \tag{4.1}$$

Solving the expression from Equation (4.1) with the appropriate polynomial coefficients as described in Equation (4.2), Equation (4.3), Equation (4.4), Equation (4.5), and Equation (4.6), respectively.

$$\begin{gathered} A_{sym} = 4\mu_c^2 K_Y^2 (C_{Z_{\dot{\alpha}}} - 2\mu_c) \\ A_{asym} = 16\mu_b^3 (K_X^2 K_Z^2 - K_{XZ}^2) \end{gathered} \tag{4.2}$$

$$\begin{gathered} B_{sym} = C_{m_{\dot{\alpha}}} 2\mu_c (C_{Z_q} + 2\mu_c) - C_{m_q} 2\mu_c (C_{Z_{\dot{\alpha}}} - 2\mu_c) - 2\mu_c K_Y^2 [C_{X_u} (C_{Z_{\dot{\alpha}}} - 2\mu_c) - 2\mu_c C_{Z_\alpha}] \\ B_{asym} = -4\mu_b^2 2C_{Y_\beta} (K_X^2 K_Z^2 - K_{XZ}^2) + C_{n_r} K_X^2 + C_{l_p} K_Z^2 + (C_{l_r} + C_{n_p}) K_{XZ} \end{gathered} \tag{4.3}$$

$$\begin{gathered} C_{sym} = C_{m_\alpha} 2\mu_c (C_{z_q} + 2\mu_c) - C_{m_{\dot{\alpha}}} 2\mu_c C_{X_0} + C_{X_u} (C_{Z_q} + 2\mu_c) + C_{m_q} C_{X_u} (C_{Z_{\dot{\alpha}}} - 2\mu_c) - \\ 2\mu_c C_{Z_\alpha} + 2\mu_c K_Y^2 (C_{X_\alpha} C_{Z_u} - C_{Z_\alpha} C_{X_u}) \\ C_{asym} = 2\mu_b (C_{Y_\beta} C_{n_r} - C_{Y_r} C_{n_\beta}) K_x^2 + (C_{Y_\beta} C_{l_p} - C_{l_\beta} C_{Y_p}) K_Z^2 + [(C_{Y_\beta} C_{n_p} - C_{n_\beta} C_{Y_p}) + \\ (C_{Y_\beta} C_{l_r} - C_{l_\beta} C_{Y_r})] K_{XZ}^2 + 4\mu_b C_{n_\beta} K_X^2 + 4\mu_b C_{l_\beta} K_{XZ} + \frac{1}{2} C_{l_p} C_{n_r} - C_{n_p} C_{l_r} \end{gathered} \tag{4.4}$$

$$\begin{gathered} D_{sym} = C_{m_u} C_{X_\alpha} (C_{Z_q} + 2\mu_c) - C_{Z_0} (C_{Z_{\dot{\alpha}}} - 2\mu_c) - C_{m_\alpha} 2\mu_c C_{X_0} + C_{X_u} (C_{Z_q} + 2\mu_c) + \\ C_{m_{\dot{\alpha}}} (C_{X_0} C_{X_u} - C_{Z_0} C_{Z_u}) + C_{m_q} (C_{X_u} C_{Z_\alpha} - C_{Z_u} C_{X_\alpha}) \\ D_{asym} = -4\mu_b C_L (C_{l_\beta} K_Z^2 + C_{n_\beta} K_{XZ}) + 2\mu_b (C_{l_\beta} C_{nP} - C_{n_\beta} C_{l_p}) + \frac{1}{2} C_{Y_\beta} (C_{l_r} C_{nP} - C_{n_r} C_{l_p}) + \\ \frac{1}{2} C_{Y_\beta} (C_{l_\beta} C_{n_r} - C_{n_\beta} C_{l_r}) + \frac{1}{2} C_{Y_r} (C_{l_p} C_{n_\beta} - C_{n_p} C_{l_\beta}) \end{gathered} \tag{4.5}$$

$$\begin{gathered} E_{sym} = -C_{m_u} (C_{X_0} C_{X_\alpha} + C_{Z_0} C_{Z_\alpha}) + C_{m_\alpha} (C_{X_0} C_{X_u} + C_{Z_0} C_{Z_u}) \\ E_{asym} = C_L (C_{l_\beta} C_{n_r} - C_{n_\beta} C_{l_r}) \end{gathered} \tag{4.6}$$

After successfully implementing this analytical model, the unit tests managed to be passed with a relative tolerance of 10^{-8}, when using the inputs presented in the [1]. With this test successfully passed, it is possible to confirm that the implementation of the state space model is completely verified, as any discrepancies in the A, B, C, D matrices would have raised different eigenvalues than the analytical method. For further verification purposes, it is possible to use Routh's determinant, as described in Equation (4.7), in order to check the overall stability of the analyzed aircraft.

$$R = BCD - AD^2 - B^2 E \tag{4.7}$$

This verification step will prove useful once the tuning of parameters takes place, as it is a quick test to assess the stability of the aircraft without running the simulation.

5 Validation

This final chapter of the report presents the validation process carried out by the group. This is an essential step in the design process, along with verification. The purpose of validation is to compare the results obtained from the computational model with those obtained from experiments. To facilitate this, an additional set of data was obtained by the group by means of a flight test. The experimental setup is briefly described in this chapter,

5.1. Experimental Setup

by Lorenzo Gonzalez

Experiments play a critical role in validating a model because they provide a means of comparing the model's predictions with actual observations of the system being modelled. The process of validation involves assessing the model's ability to predict the behaviour of the system under different conditions accurately, and experiments provide a way to generate data that can be used to evaluate the model's performance. The main point of validation is, in fact, to compare the results of the numerical model with the dynamic measurements obtained from an experiment with the aim of finding discrepancies and potential sources of errors in the model. For this project, the experiment conducted was a flight test performed near Rotterdam Airport. The experiment was performed on the 14th of March 2023 in overall good weather conditions, and the aircraft used was an 8 seats (not considering pilot and copilot) Cessna Ce-500 jet aircraft. The experiment was performed in a military flying area in the south of the Netherlands, around the city of Rotterdam. The group received a briefing before takeoff, and the experiment was divided into two main parts. The first part involved performing different static flight tests at various velocities while using a tablet to record specific static aircraft parameters and their live variations. After the pilot confirmed the aircraft's stability, each group member recorded the live values of these parameters for seven different airspeeds. Additionally, a center of gravity shift test was conducted where the heaviest group member moved from the back to the front of the aircraft, and the same measurements were taken.

The second part of the experiment involved obtaining dynamic measurements, and the group did not have to record any measurements as the aircraft collected them automatically. The only thing that was manually recorded was the time when every motion was performed. Five different maneuvers were performed by the pilot both in symmetric and asymmetric conditions of the aircraft. For the symmetric case, a short period and a phugoid maneuvers were performed. For the asymmetric case, an aperiodic roll, dutch roll, and a spiral were performed.

5.2. Simulated response based on control inputs

By Joachim Bron

Using the in-flight measured control inputs, the responses using our model were simulated and compared to the measured responses. This is done in this section for each eigenmode, having symmetric and asymmetric modes decoupled. The comparison between the measured aircraft response and the simulated response will pose a great starting point for the tuning of the stability derivatives within the validation process, i.e. the proof of match done in Section 5.3. Note that the yaw damped dutch roll is not given as the numerical model is unable to simulate this as there is no yaw damper implemented in the model and thus a comparison cannot be made.

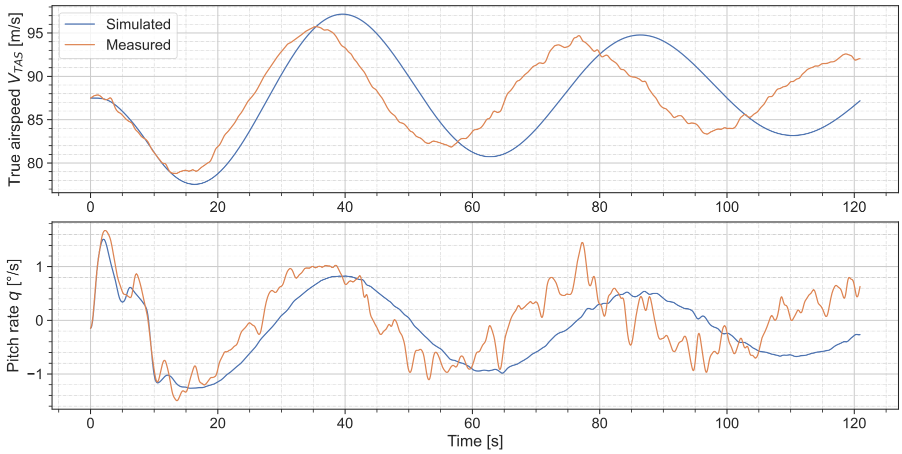


Figure 5.1: *Comparison of the measured and simulated phugoid motion*

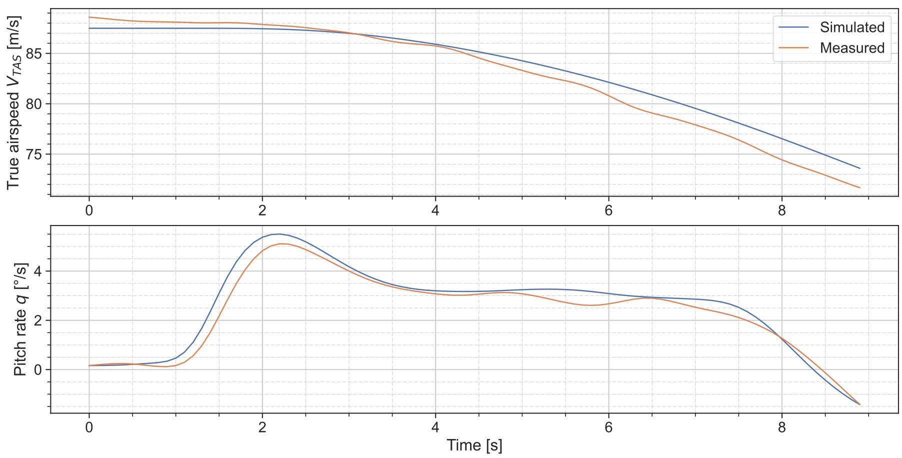


Figure 5.2: *Comparison of the measured and simulated short period*

5.2.1. Symmetric eigenmodes

The simulated and measured phugoid motions and short period are shown in Figure 5.1 and Figure 5.2, respectively. For the phugoid, only the true airspeed and pitch rate are plotted since these are the main symmetric characteristics varying during the phugoid which are of interest. The same is done for the short period.

Clearly, the initial state of the phugoid, which corresponds to the short period, is accurately predicted. Both the true airspeed and pitch rate match quite closely, and it would not be unreasonable to assume the difference to be caused mostly by noise in the measurements. However, for larger times of the phugoid, the simulated response's true airspeed and pitch rate drifts away from the measured response, which is expected as errors in the simulation accumulate over time. It can be seen that the simulated response for the phugoid's true airspeed appears to underestimate the damping (its amplitude is larger than the measured one), and overestimate the period (the simulated response has a shorter period than the measured response). For the pitch rate, the same can be said for the period, but in this case, there might be a slight overdamping for the simulated response of the true airspeed, with the difference possibly explained by the noise in the measured pitch rate. Both effects are seen in the phugoid for times after approx. 15 seconds. Although these differences are present, both curves seem to approx. converge to the same asymptote. Both eigenmotions that may occur during symmetric flight have complex non-zero eigenvalues, causing the oscillatory part, and negative real parts, causing the damping. The reason for the variations in these slight responses could potentially be explained by the assumption of linearity, which causes less damping. Furthermore, it is possible that some of the coefficients are too large or too low. For example, a higher C_{Z_u} could help reduce the period since, from a physical point of view, the forces are higher for a certain change in u, and thus the response is faster.

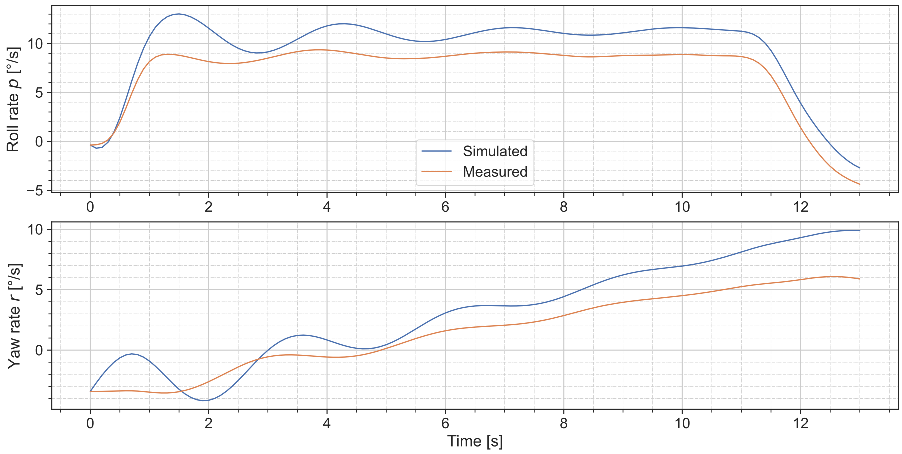


Figure 5.3: *Comparison of the measured and simulated aperiodic roll*

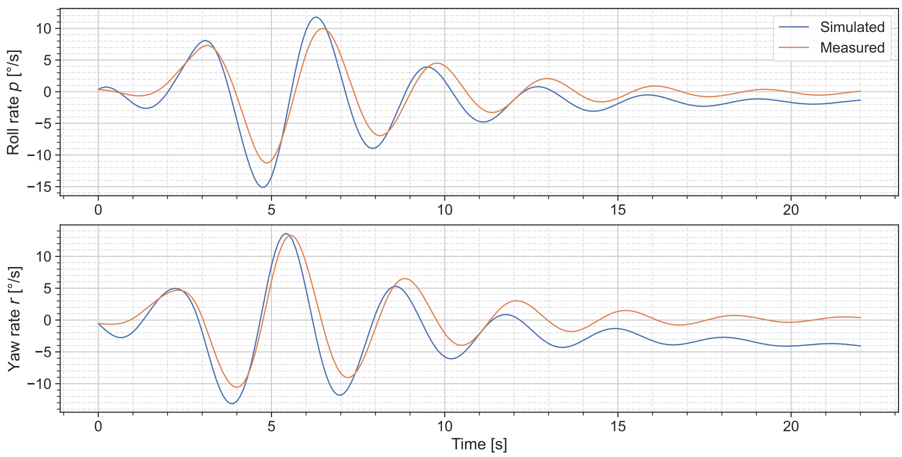


Figure 5.4: *Comparison of the measured and simulated dutch roll*

5.2.2. Asymmetric eigenmotions

The simulated and measured aperiodic roll, dutch roll, and spiral are shown in Figure 5.3, Figure 5.4, and Figure 5.5 respectively. For all three motions, only the roll and yaw rates are plotted since these are the states of interest. Note that the roll angles and yaw angles could also have been shown, but it was chosen not to look at these as they are derived from the roll and yaw rates through integration. Since they are derived and not directly measured, it was decided to focus on roll and yaw rates.

Analyzing first the aperiodic roll, multiple observations can be made. First, the global trend is correctly simulated. However, there are differences in the amplitude of the roll rate, which is overestimated. It appears that the damping is also underestimated since more oscillations are visible. The same can be said for the yaw rate, which is increasing faster than the measured one for times larger than 6 seconds.

Then, the model is able to predict the general characteristics of the dutch roll accurately. However, the period seems to be underestimated by the simulation for both the roll and yaw rates. Also, the damping seems to be underestimated. Furthermore, a slight discrepancy in the steady state value asymptote can be seen for both roll and yaw rates. For both, the simulated long-term yaw rate is different from 0, which should be the case as time becomes larger. The reason for this discrepancy could be explained by noise in the inputs, which do not go perfectly to zero at these larger times, and since the simulation uses these inputs, it also doesn't perfectly go to 0. Looking at the values for time equals 0, it can be seen that the roll and yaw rates vary too "aggressively", i.e. the slopes are too high. Again, this could be due to the inputs not being perfectly smooth at the beginning and having some noise in them. Furthermore, the linearization assumption could also be responsible for some mismatches in the measured against the simulated responses, and one of the reasons why the measured response is less oscillatory.

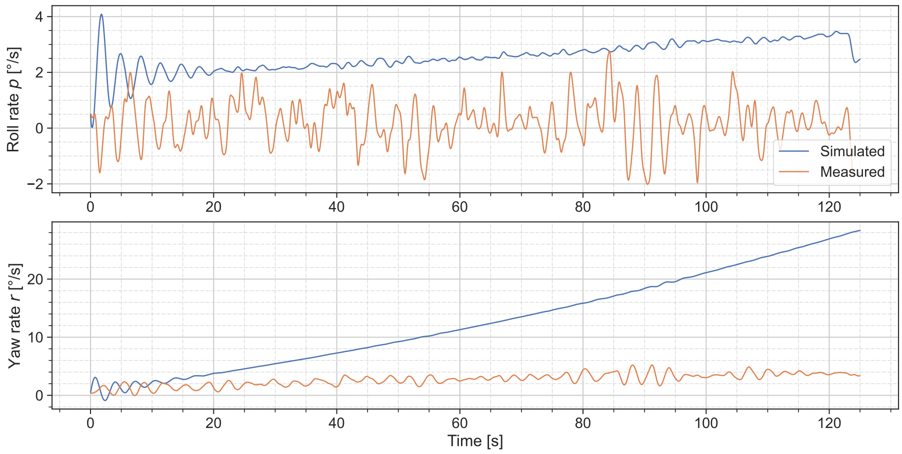


Figure 5.5: *Comparison of the measured and simulated spiral*

Finally, for the spiral, a clear mismatch in simulated and measured responses is observed. For the roll rate, the measured response seems to be approximately constant due to the noise, but in reality, it is slightly increasing (since the spiral mode is unstable). However, although the simulated response gives a slight diverging trend, the upward trend of the measured response is less pronounced. This could, again, be due to the noise, which makes it practically impossible to identify the upward trend's slope. Also, there seems to be a constant difference between both curves. The simulated response appears to have intense oscillations at the start, which seems to be incorrect. This could be due to the intense noise in the input, giving the simulation unrealistic intense inputs at the start. This is reasonable to assume since the noise is quite pronounced in the inputs (not shown for simplicity). Without this initial jump, the simulated response would be a much better match to the measured one. For the yaw rate, the difference is much more pronounced. The initial simulation for the yaw rate is quite accurate (within noise bounds), but as time increases, the simulated yaw rate diverges considerably. This could be due to the fact that eigenvalues are more positive than the real one for this eigenmode, and even a slightly more positive eigenvalue would cause significant differences for large times, but more investigation into the causes is required.

5.3. Measured and simulated response Proof-of-Match

By Mihai Fetecau

It is apparent that the simulated response does not entirely match the in-flight measured data. Thus, in order to achieve better fidelity in the simulation's output, further tuning of the control and stability derivatives is needed. This tuning, together with the reasoning behind the change in values, is provided in this section for both symmetric and asymmetric eigenmotions.

5.3.1. Symmetric eigenmotions

The phugoid is a periodic eigenmotion, which means that tuning can be performed by analyzing the oscillatory and dampening behaviour of the simulated response. Looking back at Figure 5.1, the simulated curves generally have lower periods and slightly stronger dampening behaviours. Thus, it is possible to match these two set of curves by finding and tuning only the most critical stability parameters which affect the curve's period and dampening. This can be done by applying a couple of simplifying assumptions that would reduce the Equation (2.1) to a simpler system from which the eigenvalues can be extracted symbolically. Firstly, it is possible to assume that the AOA can be completely neglected, as it does not greatly deviate from its initial state. Secondly, the derivative of the pitch rate can be assumed to equal zero due to the small changes in pitch rate. By applying these assumptions, the original system of four equations reduces to only two equations, from which it is possible to derive the approximate period and dampening of the phugoid, as shown in Equation (5.1).

$$P = 2\pi \frac{c}{V}\sqrt{\frac{4\mu_c^2}{C_{Z_u}C_{Z_0}}} \qquad \zeta = \frac{-C_{X_u}}{2\sqrt{(C_{Z_u}C_{Z_0})}} \tag{5.1}$$

Thus, in order to increase the period of the eigenmotion, it is imperative to decrease C_{Z_u}. Yet, if C_{Z_u} is decreased to much, the eigenmotion will become too damped. To counteract this effect, C_{X_u} must also be decreased until the two curves match. The result of this process can be seen in Section 5.3.1

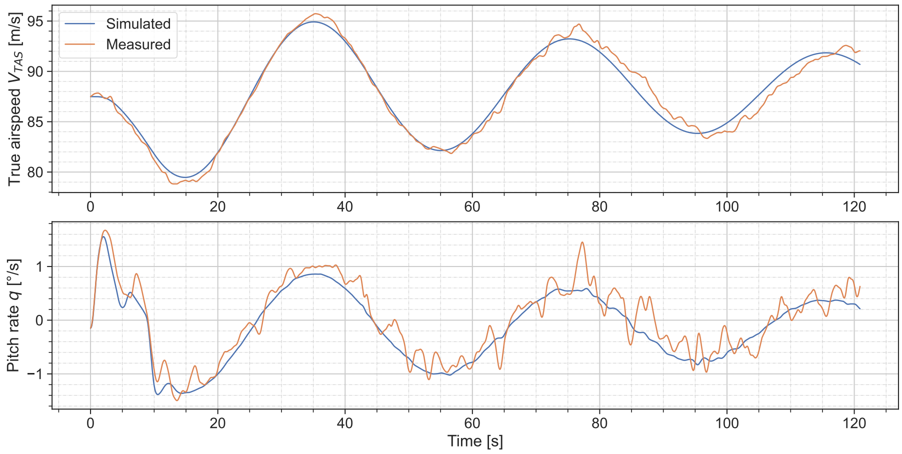


Figure 5.6: *Phugoid motion with tuned parameters*

Note that the short period eigenmotion, which is naturally contained in the response of the aircraft to a step elevator input, also scales with the newly given parameters. Although, in order to render the model accurate enough for its intended applications, it is necessary to accurately model at least the initial rise in all states without underestimating. In order to do that, the pitch stability should be slightly decreased so that an elevator input gives rise to a steeper output. An easy and intuitive method to do that is to increase the C_{m_α}. Considering that C_{m_α} was calculated through simple interpolation using only 5 points, it is easy to consider that its value is slightly different than the optimal one. Another way to increase the simulated plateau in the AOA state is to slightly decrease C_{Z_α}, also known as the lift slope of the aircraft. The results after tuning the model can be seen in Figure 5.7

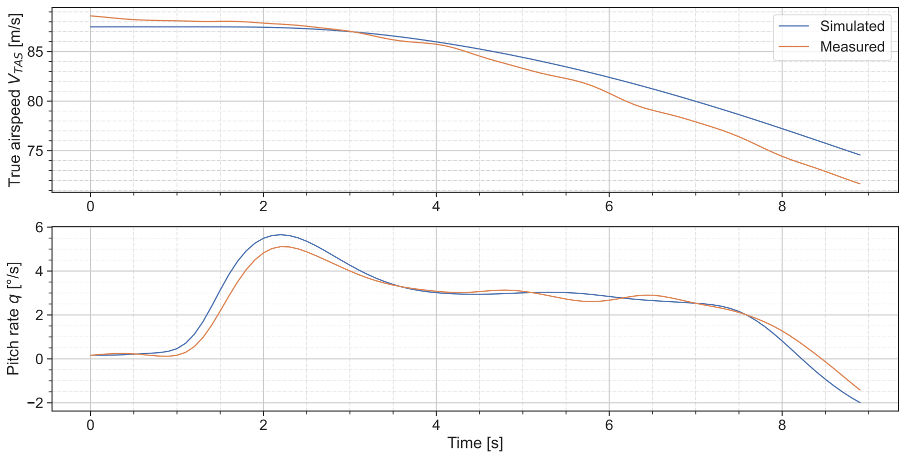


Figure 5.7: *Short period motion with tuned parameters*

All the parameters that were tuned for the symmetric eigenmotions are presented in Table 5.1, together with their initial and improved values.

Table 5.1: *Initial and improved parameters for symmetric eigenmotions*

	Initial value	Improved value
C_Z_u	-0.37616	-0.45
C_X_u	-0.095	-0.15
C_m_alpha	-0.542	-0.5
C_Z_alpha	-5.7434	-5.2

5.3.2. Asymmetric eigenmotions

As observed in the symmetric eigenmotions, the modification of certain stability and control derivatives influences multiple eigenmodes at once. This can be seen especially in the asymmetric flight case. Depending on the E polynomial coefficient presented in Equation (4.4) and Routh's discriminant, presented in Equation (4.7), the stability of the dutch roll and the spiral can be assessed. Thus, it is imperative to obtain a combination of parameters that would guarantee stability by obtaining both positive R and E.

It is extremely difficult to completely match all the simulated eigenmotions to the measured data by increasing and decreasing parameters by hand. Thus, for the optimization process, it is important to establish a primary optimization goal focused on a selected number of improvements to certain maneuvres. For the purposes of this report, the main focus will be on improving the dutch roll angular rates, as this manoeuvre can pose a greater danger to the average flight, especially in take-off and landing conditions. The second priority of the optimization process was to better match the spiral eigenmotion across the initial part of the manoeuvre, as the pilot can easily notice in time the initiation of this eigenotion and voluntarily stop it. As seen in Section 5.2.2, the simulated aperiodic roll does not deviate heavily from the measured data. Thus, the final optimization goal will be to obtain at least a better fit for the aperiodic roll.

Proceeding as in the symmetric flight case, it is possible to obtain valuable information on the parameters to be tuned by looking at the simplified EOM and assessing periodic eigenmotions by their frequency and dampening. In the asymmetric case, there is only one periodic eigenmotion, namely dutch roll, our first optimization priority. Given the slight oscillation around the steady flight condition of the roll angle, it is possible to assume that the roll angle, and implicitly the roll rate, can be assumed to be negligible. Additionally, given that the aircraft's roll and yaw angles oscillate slightly around the steady flight condition, it can be assumed that the overall trajectory of the aircraft follows a straight line, making the Y-equation in the EOM negligible. By applying these simplifying assumptions, we obtain the period and the dampening of the dutch roll as presented in Equation (5.2)

$$P = 2\pi\frac{b}{V}\sqrt{\frac{2\mu_b K_Z^2}{C_{n_\beta}}} \qquad\qquad \zeta = -\frac{C_{n_r}}{\sqrt{2\mu_b K_Z^2 C_{n_\beta}}} \tag{5.2}$$

Looking back at the initial simulated dutch roll from Figure 5.4, It is easily observable that there is a slight mismatch in the period and dampening coefficient compared to the measured data. In order to improve the match, the simulated dutch roll needs to have a slightly higher period and a slightly higher dampening coefficient. An easy way to increase the period is to decrease C_{n_β}, also known as the weathervane stability. Furthermore, in order to obtain a more pronounced dampening, the C_{n_r} coefficient must be further decreased so that it counteracts the change of the weathervane stability. After modifying these two oscillatory indicators to a satisfactory degree, the period is matched and the first pair of peaks matches the measured data. Yet, decreasing the C_{n_r} too much will make the yaw rate of the spiral to deviate even more from the measured data, which also leads to higher discrepancies for the roll rate too. Thus, another way to dampen the simulated Dutch roll is to increase C_{l_p}, further dampening the roll rate and, inadvertently, the yaw rate. Another discrepancy between the simulated data and the measured states is that the simulated curve is consistently underestimates all parameters, from the initial response to the steady state. This can be solved by adjusting certain control derivatives, in our case, making the system more sensible to control surfaces deflections, as the whole system is linear. Through trial and error, it was found out that further decreasing $C_{l_{\delta_a}}$ and $C_{n_{\delta_a}}$ does adjust the curve correctly.

The changing of these parameters would greatly affect the spiral eigenmotion, so for the purposes of this report, the intermediary plots of the eigenmotions, such a the one for the dutch roll before optimizing the spiral, are omitted. Instead, the final iteration of the dutch roll is presented in Figure 5.8.

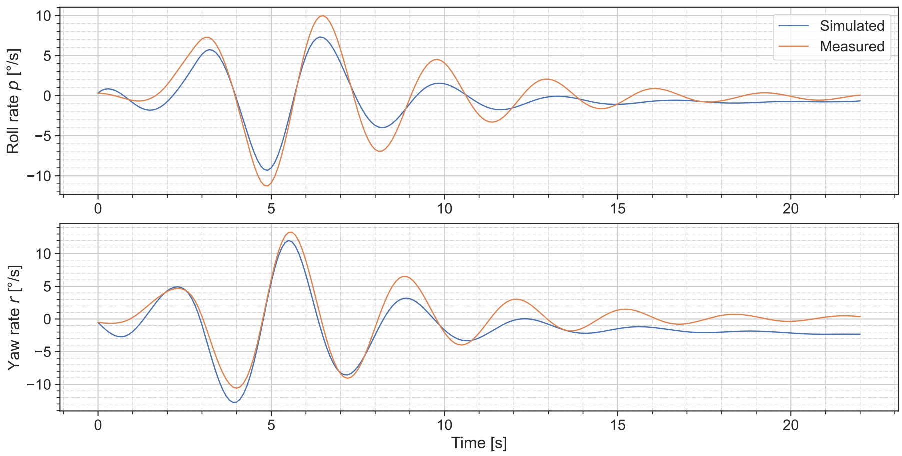


Figure 5.8: *Dutch roll eigenmotion with improved parameters*

After adjusting the aforementioned parameters to match the dutch roll, it was discovered that the spiral exhibited increased instability. In order to combat that effect, it is easy to look at the spiral stability criterion, namely having E as presented in Equation (4.6) to be positive and obtain the most critical stability derivatives from there. As we previously tuned C_{n_r} and C_{n_β}, it is possible to tune C_{l_r} and C_{l_β}. After assessing the signs of these four stability derivatives, it was concluded that lowering C_{l_r} and increasing C_{l_β} would further stabilize the spiral.

After a number of iterations so that the optimized values of the aforementioned parameters, the following spiral motion is obtained, as presented Figure 5.9.

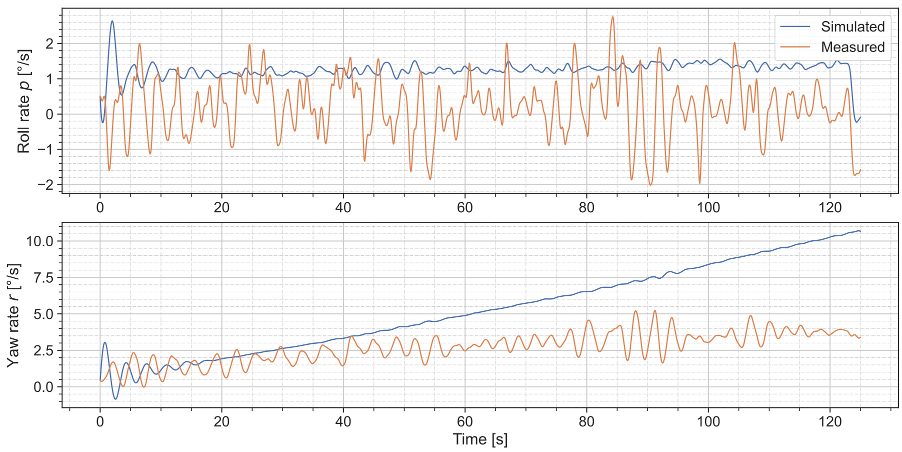


Figure 5.9: *Spiral eigenmotion with improved parameters*

Given that all asymmetric eigenmotions are coupled, the improvements on the parameters will influence the aperiodic roll too. After visual inspection and considering the aforementioned optimization goals, it was decided that no further optimization is needed for the aperiodic roll. The final version of the eigenmotion is presented in Figure 5.10.

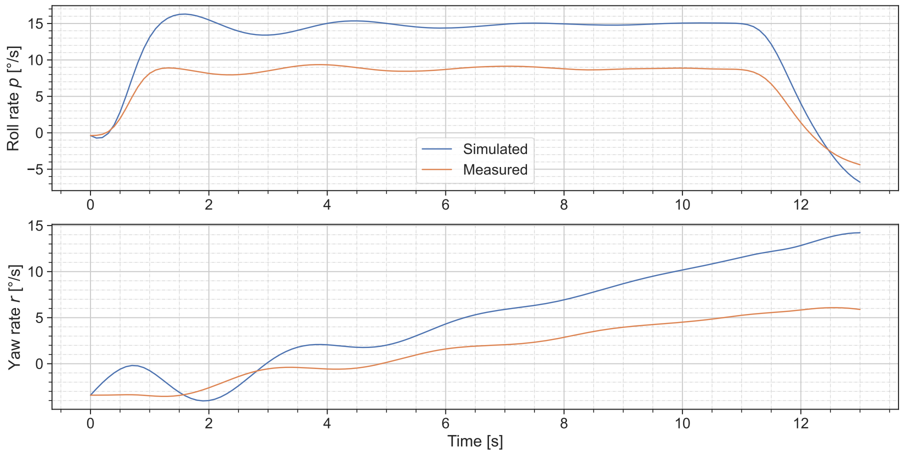


Figure 5.10: *Aperiodic roll eigenmotion with improved parameters*

All improved asymmetric parameters, together with their initial value, are presented in Table 5.2

Table 5.2: *Initial and improved parameters for the asymmetric eigenmotions*

	Initial value	Improved value
C_n_beta	0.1348	0.12
C_n_r	-0.2061	-0.28
C_n_p	-0.0602	-0.1
C_l_p	-0.71085	-0.6
C_l_r	0.2376	0.19
C_l_delta_A	-0.23088	-0.3
C_n_delta_A	-0.012	-0.03

5.4. Comparison with other test data

By Dominik Stiller

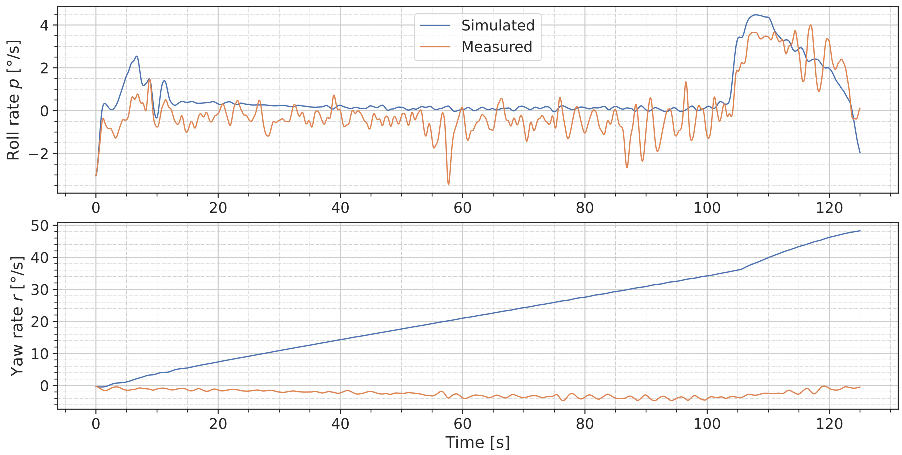


Figure 5.11: *Spiral eigenmotion with improved parameters and input from reference data.*

Figure 5.11 shows the spiral maneuver again, but with input from the reference data. We choose this maneuver because we had the largest mismatch for this eigenmotion. Comparing with the same plot from our flight's input (Figure 5.9), we see that the roll rate matches well but has, again, a slight positive bias. However, the yaw rate diverges even more than before. This indicates a problem with our model instead of the data. Our guess for this discrepancy is that the improved model prioritizes the stability of the Dutch roll. Thus, it is possible that the spiral stability criterion is overlooked for this configuration. Our suggestion would be to further decrease C_{n_r} so that we exhibit a more dampened yaw rate.

6 Conclusion

by Lorenzo Gonzalez, Dominik Stiller, Timo de Kemp

The aim of this report was to improve the group's comprehension of aircraft performance, stability, and control, as well as to practice the process of verification and validation for an aerospace engineering case study. The initial task was to develop and analyze a numerical model able to solve the equations of motion and simulate various input's responses. After that, a verification process was performed to assess the correctness and accuracy of the model results. The final step was to validate the model by comparing the results obtained from the flight test with the reference data.

We obtained confident estimates of the aerodynamic parameters, which were agreed between the reference dataset and ours. Their signs agree with the expected ones for a conventional aircraft. Similarly, we measured the eigenmotion characteristics agreed between the two datasets, particularly for phugoid and Dutch roll. Determining the same characteristics from the eigenvalues of the state space model, we find discrepancies in the damping, but the periods agree well. As expected, we find four stable eigenmodes and an unstable spiral.

The verification process confirmed individual functions worked using unit tests for functions made. After this the whole simulation was verified using pulse and step inputs on the control surfaces and checking if the response of the model is as expected. Finally the eigenvalues were checked, the expected numerical values were determined analytically and fell within an accuracy of 10^{-8}.

The validation process assessed the discrepancies between the flight data measured during a test flight and the simulated response of the model given the same input as the one measured from the flight. It was discovered that the symmetric eigenmotions matched the flight data much better than the asymmetric ones. Furthermore, the spiral eigenmode presented the highest deviation from the measured data, exhibiting clear signs of instability. An optimization process was attempted by hand in order to match the symmetric eigenmodes better and to increase the stability of the spiral while also matching the dutch roll. In the end, a trade-off was performed based on the priorities of the optimization process. The final results showed better spiral stability but was unable to make it match the flight data, unfortunately.

This being said, there are multiple things the team recommends that could improve the model to have better results. The following recommendations are ordered based on the importance of the effects they have on the final model results:

- To have a large improvement in the accuracy of the model, the team could have loaded a larger quantity of data to validate the model. For validation, the group used a reference data set that was given by the aircraft manufacturer, but this could be improved for example using the flight test data from other groups. It is essential to use data from all relevant groups to validate the model. This will help to ensure that the model is accurate and valid for a wide range of conditions. By using data from all groups, we can test the model's ability to accurately simulate a variety of scenarios, thanks to the larger amount of data to be relied on. Additionally, having more data points for the stationary measurements so that more accurate aerodynamic parameters could be obtained.
- Another improvement that could be implemented in the model is related to the use of optimisation tools. An optimization tool could help optimise the values of the several derivatives derived, which can improve the accuracy and validity of the model. By optimizing these values, we can ensure that the model accurately reflects more real-world conditions, making it more useful for future research and applications.

- A problem encountered with the model was that it was not possible for the model to simulate stick free controls without adding calculations that include the hinge moment. This was mainly due to the fact that the hinge moment being a fundamental aspect of how the aircraft control surfaces behave. Stick-free flight is a critical aspect of flight dynamics, and a model that can simulate it accurately would be highly valuable. By improving the model to simulate stick-free flight directly, we can better understand the dynamics of flight and improve the safety and efficiency of flight systems.
- Another recommendation the team gives to improve the model further is about the value of the sideslip angle. The model numerically solves for the sideslip angle resulting in an important limitation. It is crucial to directly measure the sideslip angle to validate the model for asymmetric flight better. Asymmetric flight is a challenging condition that can lead to significant safety issues. By directly measuring the sideslip angle, we can better understand how the model performs under these conditions and make necessary improvements to ensure the safety and efficiency of flight systems.

Bibliography

[1] W. van der Wal. E. Mooij, Z. P., *Lecture Notes – Flight Dynamics*, February 2023. Academic year 2022/2023.

[2] in t' Veld, A., and Mulder, T., "Flight Dynamics Assignment AE3212-II," , March 2023. Accessed 08-03-2023.

A

Appendix A

A.1. Mass balance sheet

Payload computations			
Crew & pax	xcg(datum) [inches]	Mass [pounds]	Moment [inch-pounds]
seat 1	131	209.4391	27436.52853
seat 2	131	165.3467	21660.41726
seat 3	214	121.2542	25948.40826
seat 4	214	125.6635	26891.98674
seat 5	251	143.3005	35968.41808
seat 6	251	132.2774	33201.61669
seat 7	288	211.6438	60953.40625
seat 8	288	227.0761	65397.92545
seat 10	170	152.119	25860.22335
Baggage			
Nose	74	0	0
Aft cabin	321	0	0
	338	0	0
Payload		1488.12	323318.9306

Mass and balance computations		
Item	Mass [pounds]	Moment [inch-pounds]
Basic empty mass	9172.9	2676101.846
xcg(datum) at BEM = 291.74		
Payload	1488.12	323318.9306
Zero fuel mass	10661	2999420.777
at ZFM = 281.344629		
Fuel load	2909	829630.6267
Ramp mass	13570	3829051.403
xcg(datum) at RM = 282.17		

Figure A.1: *Filled in mass balance sheet for flight taken.*

A.2. Final code structure

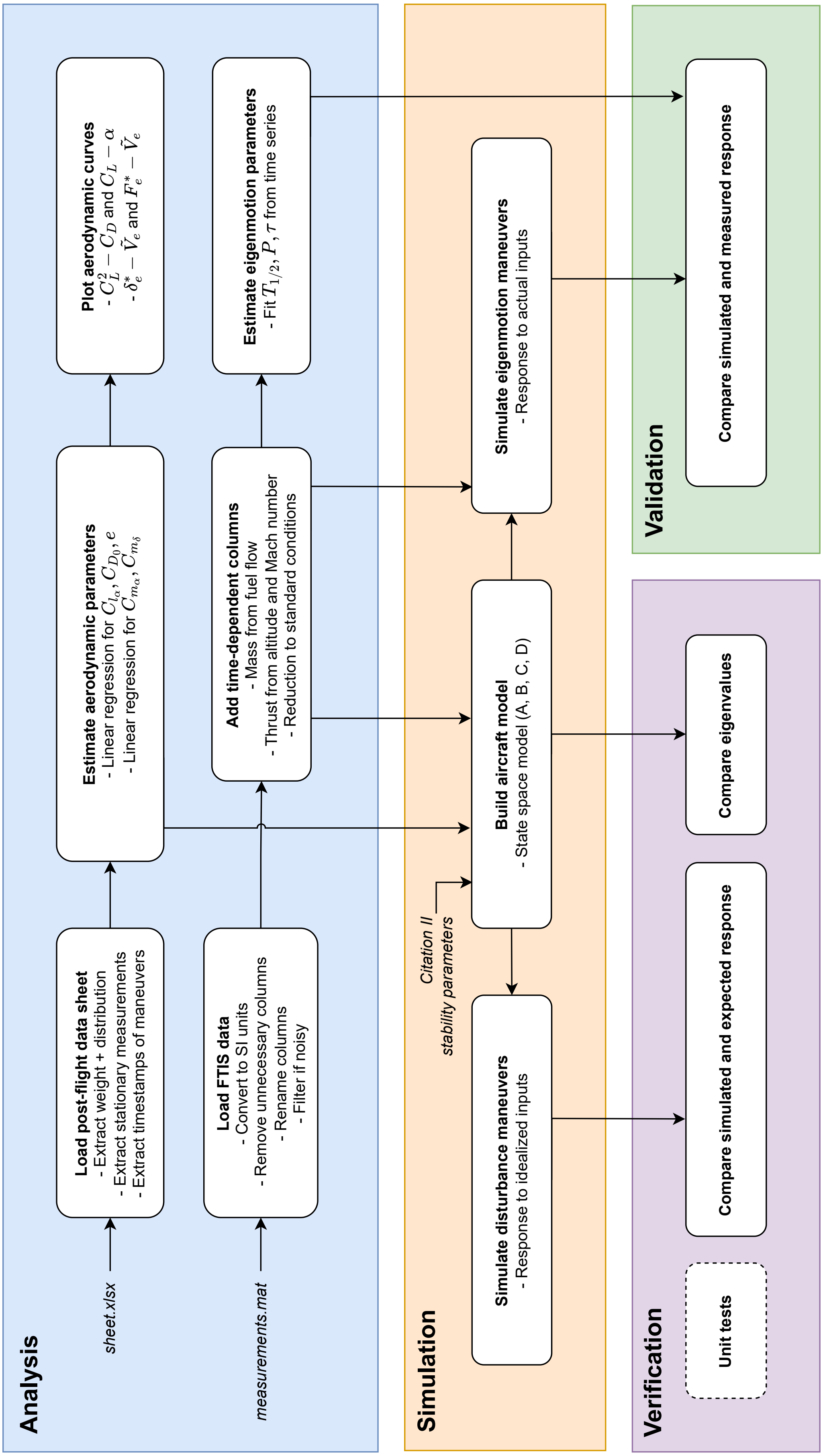


Figure A.2: *Block diagram showing how flight test data are processed to estimate parameters and compared with a simulated model for validation.*

B

Appendix B

Table B.1: *Task distribution*

	5259754	**5107245**	**5253969**	**5242231**	**5236789**	**5310474**
1. Introduction						
Problem description: relevance, report overview, etc.				x		
Block diagram			x			
2. Model						
Axis system transformation				x		
Assumptions of analytical model		x		x	x	
Assumptions of numerical model		x		x	x	
State space from analytical				x	x	
Center of gravity range		x				x
Limitations of the numerical model				x		
3. Analysis						
Measurement description						x
Loading of measurements			x			
Loading of post-flight data sheet			x			
Unit conversion functions	x					x
Reduction of measurements to standard conditions	x					x
Plot Cl-alpha, Cd-alpha and Cl2-Cd	x	x				x
Estimation of Cl_alpha, Cd0 and e	x					x
Plot trim curves: Plot elevator trim and control force curves	x	x				x
Estimation of C_m_delta	x					x
Estimation of C_m_alpha	x					x
Estimation of tau, P and T_1/2		x				
Summary of results			x			
4. Verification						
Unit tests for each previously defined function	x			x		x
Integrated testing					x	x
Eigenvalues				x	x	x
5. Validation						
Experimental set up				x		
Simulate response based on control inputs	x	x			x	
Measured and simulated response Proof-of-Match					x	
Comparison other test data			x			
6. Conclusion						
Discussion on results				x		
Validity of the model				x		
Further improvements and suggestions of the model				x		
Recomendations				x		x
Appendix						
Task distribution		x				
Code			x			
Total hours	50	47	63	46	45	47

C
Code

fd/__main__.py

```python
import sys

from fd.analysis.flight_test import FlightTest
from fd.simulation.aircraft_model import AircraftModel
from fd.simulation.simulation import Simulation
from fd.validation.comparison_eigenvalues import EigenvalueComparison

if __name__ == "__main__":
    flight_test = FlightTest(sys.argv[1])
    # print(flight_test.aerodynamic_parameters)
    # flight_test.make_aerodynamic_plots()

    aircraft_model = AircraftModel(flight_test.aerodynamic_parameters)
    simulation = Simulation(aircraft_model)

    # comparison = SimulatedMeasuredComparison(flight_test, simulation)
    # comparison.run_simulations()
    # comparison.plot_responses()

    comparison_eigenvalues = EigenvalueComparison(flight_test, aircraft_model)
    comparison_eigenvalues.compare()
```

fd/plotting.py

```python
import os
from pathlib import Path
from typing import Union

import matplotlib
import matplotlib.pyplot as plt
import seaborn as sb

sb.set(
    context="paper",
    style="ticks",
    font_scale=1.6,
    font="sans-serif",
    rc={
        "lines.linewidth": 1.2,
        "axes.titleweight": "bold",
    },
)


def save_plot(results_folder: Union[Path, str], name: str, fig=None, type="pdf"):
    if isinstance(results_folder, str):
        results_folder = Path(results_folder)

```

```python
    plots_folder = results_folder / "plots"
    plots_folder.mkdir(parents=True, exist_ok=True)

    if fig is None:
        fig = plt.gcf()
    fig.savefig(
        os.path.join(plots_folder, f"{name}.{type}"),
        dpi=450,
        bbox_inches="tight",
        pad_inches=0.01,
    )


def format_plot(
    xlocator=None,
    ylocator=None,
    tight_layout=True,
    zeroline=False,
):
    fig = plt.gcf()
    for ax in fig.axes:
        if zeroline:
            ax.axhline(0, linewidth=1.5, c="black")

        xlocator_ax = xlocator
        if not xlocator_ax:
            if ax.get_xscale() == "log":
                xlocator_ax = matplotlib.ticker.LogLocator(base=10, subs="auto", numticks=100)
            else:
                xlocator_ax = matplotlib.ticker.AutoMinorLocator()

        ylocator_ax = ylocator
        if not ylocator_ax:
            if ax.get_yscale() == "log":
                ylocator_ax = matplotlib.ticker.LogLocator(base=10, subs="auto", numticks=100)
            else:
                ylocator_ax = matplotlib.ticker.AutoMinorLocator()

        ax.get_xaxis().set_minor_locator(xlocator_ax)
        ax.get_yaxis().set_minor_locator(ylocator_ax)
        ax.grid(visible=True, which="major", linewidth=1.0)
        ax.grid(visible=True, which="minor", linewidth=0.5, linestyle="-.")

    if tight_layout:
        fig.tight_layout(pad=0.1, h_pad=0.4, w_pad=0.4)
```

fd/structs.py

```python
from dataclasses import dataclass


@dataclass()
class SimulationOutput:
    pass


@dataclass
class AerodynamicParameters:
    C_L_alpha: float
    alpha_0: float
    C_D_0: float
    C_m_alpha: float
```

```
    C_m_delta: float
    e: float
```

fd/conversion.py

```python
import datetime
import re
from typing import Union

import numpy as np


def lbshr_to_kgs(lbshr):
    """Convert value from lbs/hr to kg/s"""
    return 0.45359237 / 3600 * lbshr


def psi_to_Pa(psi):
    """Convert value from psi to Pa"""
    return 6894.757 * psi


def ftmin_to_ms(ftmin):
    """Convert value from ft/min to m/s"""
    return 0.3048 / 60 * ftmin


def lbs_to_kg(lbs):
    """Convert mass in pounds to kilograms"""
    return 0.45359237 * lbs


def kts_to_ms(kts):
    """Convert speed in knots to meters per second"""
    return 1852 / 3600 * kts


def ft_to_m(ft):
    """Convert distance in feet to meters"""
    return 0.3048 * ft


def in_to_m(ft):
    """Convert distance in inch to meters"""
    return 0.0254 * ft


def C_to_K(C):
    """Convert temperature in Celcius to Kelvin"""
    return 273.15 + C


def deg_to_rad(deg):
    """Convert angle in degrees to radians"""
    return np.deg2rad(deg)


def degs_to_rads(degs):
    """Convert rotation in degrees/s to radians/s"""
    return np.deg2rad(degs)


```

```python
def timestamp_to_s(timestamp: Union[str, datetime.time]):
    """
    Convert datetime.time or timestamp string (both from Excel sheet) to seconds.
    There is no consistent format:
     - datetime.time: mm:ss timestamp is parsed but incorrectly as hh:mm
     - h.mm:ss
     - h.mm
     - h:mm:ss
     - mm:ss
     - mm
    """
    if not timestamp:
        return None

    if isinstance(timestamp, str):
        hour = 0
        minute = 0
        second = 0

        timestamp = timestamp.strip()

        if match := re.fullmatch(r"(\d+)\.(\d+):(\d+)", timestamp):
            # h.mm:ss
            hour, minute, second = match.groups()
        elif match := re.fullmatch(r"(\d+)\.(\d+)", timestamp):
            # h.mm
            hour, minute = match.groups()
        elif match := re.fullmatch(r"(\d+):(\d+):(\d+)", timestamp):
            # h:mm:ss
            hour, minute, second = match.groups()
        elif match := re.fullmatch(r"(\d+):(\d+)", timestamp):
            # mm:ss
            minute, second = match.groups()
        elif match := re.fullmatch(r"(\d+)", timestamp):
            # mm
            minute = match.group(0)
        else:
            raise "Invalid time format"
    elif isinstance(timestamp, datetime.time):
        hour = 0
        # This is not a mistake, the datetime is interpreted incorrectly
        minute = timestamp.hour
        second = timestamp.minute
    else:
        raise "Unsupported time type"

    hour = float(hour)
    minute = float(minute)
    second = float(second)

    assert 0 <= hour
    assert 0 <= minute < 60
    assert 0 <= second < 60

    return hour * 3600 + minute * 60 + second


def inchpound_to_kgm(inchpound):
    """

    Args:
```

```
        inchpound (float): Massmoment expressed in inchpounds

    Returns (float): Massmoment expressed in kilogram meters

    """
    return inchpound * 0.45359237 * 0.0254
```

fd/util.py

```
from statistics import mean

import pandas as pd


def get_closest(df: pd.DataFrame, time) -> pd.DataFrame:
    """
    Gets the rows in df that is closest after the time given. If time is past the
    last timestamp, return the last row.

    df's index should be float timestamps.

    Args:
        time: single or multiple timestamps
        df: DataFrame

    Returns:
        Rows corresponding to the closest next time
    """
    closest_idx = df.index.searchsorted(time)
    closest_idx = closest_idx.clip(0, len(df.index) - 1)
    return df.iloc[closest_idx]


def mean_not_none(l: list[float]) -> float:
    """
    Calculate the mean of all non-None values in x.

    Args:
        l: List of elements

    Returns:
        Mean
    """
    return mean(filter(lambda e: e is not None, l))


def mean_not_nan_df(dfs: list[pd.DataFrame]) -> pd.DataFrame:
    """
    Calculate the cell-wise mean of all non-NAN values in dfs.

    Args:
        dfs: List of DataFrames

    Returns:
        Means as DataFrame
    """
    df_mean = pd.concat(dfs).groupby(level=0).mean()
    return df_mean.astype(dfs[0].dtypes)
```

fd/io.py

```python
import warnings
from typing import Any

import pandas as pd
import scipy
from openpyxl.reader.excel import load_workbook


def load_ftis_measurements(path: str) -> pd.DataFrame:
    raw = scipy.io.loadmat(f"{path}/measurements.mat", simplify_cells=True)["flightdata"]
    data = {}
    for column_name, values in raw.items():
        data[column_name] = values["data"]
    data = pd.DataFrame(data).set_index("time")
    return data


def extract_ftis_column_descriptions(path: str):
    raw = scipy.io.loadmat(f"{path}/measurements.mat", simplify_cells=True)["flightdata"]
    metadata = []
    for column_name, values in raw.items():
        metadata.append(
            {
                "Column": column_name,
                "Description": values["description"],
                "Units": values["units"],
            }
        )
    metadata = pd.DataFrame(metadata)
    metadata.to_excel("data/column_descriptions.xlsx", index=False)


def load_data_sheet(path: str) -> list[list[Any]]:
    with warnings.catch_warnings():
        warnings.filterwarnings("ignore", category=UserWarning, module="openpyxl")
        wb = load_workbook(filename=path)
    ws = wb.worksheets[0]
    return [[cell.value for cell in row] for row in ws.rows]


if __name__ == "__main__":
    import sys

    extract_ftis_column_descriptions(sys.argv[1])
```

fd/simulation/aircraft_model.py

```python
from math import sin, cos

import control.matlab as ml
import matplotlib.pyplot as plt
import numpy as np
import numpy.linalg as alg
import pandas as pd
from numpy.typing import ArrayLike

from fd.analysis.aerodynamics import calc_CL
from fd.simulation.constants import *
from fd.structs import AerodynamicParameters


class AircraftModel:
```

```
    def __init__(self, aero_params: AerodynamicParameters):
        self.aero_params = aero_params

    def get_non_dim_masses(self, m: float, rho: float):
        """

        Args:
            m: Aircraft mass
            rho: Air density for the initial steady conditions

        Returns:
            muc: Non-dimensional aircraft mass wrt MAC
            mub: Non-dimensional aircraft mass wrt wingspan

        """
        muc = m / (rho * S * c)
        mub = m / (rho * S * b)
        return muc, mub

    def get_gravity_term_coeff(self, m: float, V0: float, rho: float, th0: float):
        """

        Args:
            m: Aircraft mass
            V0: Airspeed for the initial steady flight condition
            rho: Air density for the initial steady conditions
            th0: Pitch angle for the initial steady flight condition

        Returns:
            CX0: Gravity term coefficient in X-direction
            CZ0: Gravity term coefficient in Z-direction

        """
        W = m * g
        CX0 = W * sin(th0) / (0.5 * rho * V0**2 * S)
        CZ0 = -W * cos(th0) / (0.5 * rho * V0**2 * S)
        return CX0, CZ0

    def get_state_space_matrices_symmetric_from_df(self, data: pd.DataFrame):
        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        theta0 = data["theta"].iloc[0]

        ABCD = self.get_state_space_matrices_symmetric(m, V0, rho0, theta0)

        return ABCD

    def get_state_space_matrices_symmetric(
        self, m: float, V0: float, rho: float, th0: float
    ) -> tuple[ArrayLike, ArrayLike, ArrayLike, ArrayLike]:
        """

        Args:
            m: Aircraft mass
            V0: Airspeed for the initial steady flight condition
            rho: Air density for the initial steady conditions
            th0: Pitch angle for the initial steady flight condition

        Returns:
            A: State matrix
```

```
            B: Control matrix
            C: Output matrix
            D: Feedthrough matrix

        """
        Cma = self.aero_params.C_m_alpha
        Cmde = self.aero_params.C_m_delta
        muc = self.get_non_dim_masses(m, rho)[0]
        CX0, CZ0 = self.get_gravity_term_coeff(m, V0, rho, th0)

        # C_1*x_dot + C_2*x +C_3*u = 0
        # x = [u_hat, alpha, theta, q]T
        C_1 = np.array(
            [
                [-2 * muc * c / V0, 0, 0, 0],
                [0, (CZadot - 2 * muc) * c / V0, 0, 0],
                [0, 0, -c / V0, 0],
                [0, Cmadot * c / V0, 0, -2 * muc * (KY2) * ((c / V0) ** 2)],
            ]
        )
        C_2 = np.array(
            [
                [CXu, CXa, CZ0, 0],
                [CZu, CZa, -CX0, (CZq + 2 * muc) * c / V0],
                [0, 0, 0, c / V0],
                [Cmu, Cma, 0, Cmq * c / V0],
            ]
        )
        C_3 = np.array([[CXde], [CZde], [0], [Cmde]])

        A = np.matmul(-alg.inv(C_1), C_2)
        B = np.matmul(-alg.inv(C_1), C_3)
        # In order to get the state variables as output:
        C = np.eye(4)
        D = np.zeros((4, 1))
        return A, B, C, D

    def get_state_space_matrices_asymmetric_from_df(self, data: pd.DataFrame):
        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        theta0 = data["theta"].iloc[0]
        CL = calc_CL(data["W"].iloc[0] * np.cos(theta0), V0, rho0)

        ABCD = self.get_state_space_matrices_asymmetric(m, V0, rho0, theta0, CL)

        return ABCD

    def get_state_space_matrices_asymmetric(
        self, m: float, V0: float, rho: float, th0: float, CL: float
    ) -> tuple[ArrayLike, ArrayLike, ArrayLike, ArrayLike]:
        """

        Args:
            m: Aircraft mass
            V0: Airspeed for the initial steady flight condition
            rho: Air density for the initial steady conditions
            th0: Pitch angle for the initial steady flight condition
            CL: Lift coefficient for steady flight

        Returns:
```

```
            A: State matrix
            B: Control matrix
            C: Output matrix
            D: Feedthrough matrix
        """

        mub = self.get_non_dim_masses(m, rho)[-1]
        # x = [beta, phi, p, r]T
        # C_1*x_dot + C_2*x +C_3*u = 0

        C_1 = np.array(
            [
                [(CYbdot - 2 * mub) * b / V0, 0, 0, 0],
                [0, -b / (2 * V0), 0, 0],
                [
                    0,
                    0,
                    -4 * mub * KX2 * (b / V0) * (b / (2 * V0)),
                    4 * mub * KXZ * (b / V0) * (b / (2 * V0)),
                ],
                [
                    Cnbdot * b / V0,
                    0,
                    4 * mub * KXZ * (b / V0) * (b / (2 * V0)),
                    -4 * mub * KZ2 * (b / V0) * (b / (2 * V0)),
                ],
            ]
        )
        C_2 = np.array(
            [
                [CYb, CL, CYp * (b / (2 * V0)), (CYr - 4 * mub) * (b / (2 * V0))],
                [0, 0, 1 * (b / (2 * V0)), 0],
                [Clb, 0, Clp * (b / (2 * V0)), Clr * (b / (2 * V0))],
                [Cnb, 0, Cnp * (b / (2 * V0)), Cnr * (b / (2 * V0))],
            ]
        )
        C_3 = np.array([[CYda, CYdr], [0, 0], [Clda, Cldr], [Cnda, Cndr]])

        A = -alg.inv(C_1) @ C_2
        B = -alg.inv(C_1) @ C_3

        A = np.array(
            [
                [
                    V0 / b * CYb / 2 / mub,
                    V0 / b * CL / 2 / mub,
                    V0 / b * CYp / 2 / mub * (b / 2 / V0),
                    V0 / b * (CYr - 4 * mub) / 2 / mub * (b / 2 / V0),
                ],
                [0, 0, 2 * V0 / b * (b / 2 / V0), 0],
                [
                    V0
                    / b
                    * (Clb * KZ2 + Cnb * KXZ)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                    0,
                    V0 / b * (Clp * KZ2 + Cnp * KXZ) / (4 * mub * (KX2 * KZ2 - KXZ**2)),
                    V0 / b * (Clr * KZ2 + Cnr * KXZ) / (4 * mub * (KX2 * KZ2 - KXZ**2)),
                ],
                [
```

```
                    V0
                    / b
                    * (Clb * KXZ + Cnb * KX2)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                    0,
                    V0 / b * (Clp * KXZ + Cnp * KX2) / (4 * mub * (KX2 * KZ2 - KXZ**2)),
                    V0 / b * (Clr * KXZ + Cnr * KX2) / (4 * mub * (KX2 * KZ2 - KXZ**2)),
                ],
            ]
        )
        print(np.linalg.eig(A)[0])
        B = np.array(
            [
                [V0 / b * CYda / 2 / mub, V0 / b * CYdr / 2 / mub],
                [0, 0],
                [
                    V0
                    / b
                    * (Clda * KZ2 + Cnda * KXZ)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                    V0
                    / b
                    * (Cldr * KZ2 + Cndr * KXZ)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                ],
                [
                    V0
                    / b
                    * (Clda * KXZ + Cnda * KX2)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                    V0
                    / b
                    * (Cldr * KXZ + Cndr * KX2)
                    / (4 * mub * (KX2 * KZ2 - KXZ**2))
                    / (b / (2 * V0)),
                ],
            ]
        )

        # In order to get the state variables as output:
        C = np.eye(4)
        D = np.zeros((4, 2))
        E_prim = CL * (Clb * Cnr - Cnb * Clr)
        print("E = ", E_prim)
        return A, B, C, D

    def get_eigenvalues_and_eigenvectors(self, A: ArrayLike):
        """

        Args:
            A: State matrix
            B: Control matrix
            C: Output matrix
            D: Feedthrough matrix

        Returns:
            Eigenvalues and Eigenvectors
```

```
        """
        eigenvalues, eigenvectors = alg.eig(A)
        return eigenvalues, eigenvectors

    def get_step_input(self, maneuvre_duration, dt, input_duration, input_value, plot=False):
        t = np.arange(0, maneuvre_duration + dt, dt)
        u = np.zeros(t.shape)
        u[: int(input_duration / dt)] = input_value * np.ones(u[: int(input_duration / dt)].size)
        if plot:
            fig = plt.figure()
            ax = fig.add_subplot(1, 1, 1)
            ax.plot(t, u)
            ax.set_xlabel("Time [s]")
            ax.set_ylabel("$delta_e$")
        return t, u

    def get_response_plots_symmetric(self, sys, x0, t, u, V0):
        yout, t, xout = ml.lsim(sys, u, t, x0)
        fig, axs = plt.subplots(2, 2, sharex=True)

        axs[0, 0].plot(t, xout[:, 0] + V0 * np.ones(t.size))
        axs[0, 0].set_title("V [m/sec")
        axs[0, 0].grid()

        axs[1, 0].plot(t, xout[:, 1])
        axs[1, 0].set_title("$alpha$ [rad]")
        axs[1, 0].grid()

        axs[0, 1].plot(t, xout[:, 2])
        axs[0, 1].set_title("$theta$ [rad]")
        axs[0, 1].grid()

        axs[1, 1].plot(t, xout[:, 3])
        axs[1, 1].set_title("q [rad/sec]")
        axs[1, 1].grid()

        plt.show()

    def get_response_plots_asymmetric(self, sys, x0, t, u, V0):
        yout, t, xout = ml.lsim(sys, u, t, x0)
        fig, axs = plt.subplots(2, 2, sharex=True)

        axs[0, 0].plot(t, xout[:, 0])
        axs[0, 0].set_title("$beta$ [rad]")

        axs[1, 0].plot(t, xout[:, 1])
        axs[1, 0].set_title("$phi$ [rad]")

        axs[0, 1].plot(t, xout[:, 2])
        axs[0, 1].set_title("p [rad/sec]")

        axs[1, 1].plot(t, xout[:, 3])
        axs[1, 1].set_title("r [rad/sec]")

        plt.show()

    def get_idealized_shortperiod_eigenvalues(self, m, rho, V0):
        Cma = self.aero_params.C_m_alpha
        muc = self.get_non_dim_masses(m, rho)[0]
        A = 2 * muc * (KY2) * (2 * muc - CZa)
        B = -2 * muc * KY2 * CZa - (2 * muc + CZq) * Cma - (2 * muc + CZa) * Cmq
```

```
        C = CZa * Cmq - (2 * muc + CZq) * Cma
        eigenvalue_shortperiod1 = (
            complex(-B / (2 * A), +np.sqrt(4 * A * C - B**2) / (2 * A)) * V0 / c
        )
        eigenvalue_shortperiod2 = (
            complex(-B / (2 * A), -np.sqrt(4 * A * C - B**2) / (2 * A)) * V0 / c
        )
        return eigenvalue_shortperiod1, eigenvalue_shortperiod2

    def get_idealized_phugoid_eigenvalues(self, m, rho, V0, th0):
        Cma = self.aero_params.C_m_alpha
        muc = self.get_non_dim_masses(m, rho)[0]
        _, CZ0 = self.get_gravity_term_coeff(m, V0, rho, th0)
        A = 2 * muc * (CZa * Cmq - 2 * muc * Cma)
        B = 2 * muc * (CXu * Cma - Cmu * CXa) + Cmq * (CZu * CXa - CXu * CZa)
        C = CZ0 * (Cmu * CZa - CZu * Cma)
        eigenvalue_phugoid1 = complex(-B / (2 * A), +np.sqrt(4 * A * C - B**2) / (2 * A)) * V0 / c
        eigenvalue_phugoid2 = complex(-B / (2 * A), -np.sqrt(4 * A * C - B**2) / (2 * A)) * V0 / c
        return eigenvalue_phugoid1, eigenvalue_phugoid2

    def get_idealized_aperiodicroll_eigenvalues(self, m, rho, V0):
        mub = self.get_non_dim_masses(m, rho)[1]
        eigenvalue_aperiodicroll = Clp / (4 * mub * KX2) * V0 / c
        return eigenvalue_aperiodicroll

    def get_idealized_dutchroll_eigenvalues(self, m, rho, V0):
        mub = self.get_non_dim_masses(m, rho)[1]
        A = 2 * (Cnr + 2 * KZ2 * CYb)
        B = np.sqrt(64 * KZ2 * (4 * mub * Cnb + CYb * Cnr) - 4 * (Cnr + 2 * KZ2 * CYb) ** 2)
        C = 16 * mub * KZ2
        eigenvalue_dutchroll1 = (A + B) / C * V0 / c
        eigenvalue_dutchroll2 = (A - B) / C * V0 / c
        return eigenvalue_dutchroll1, eigenvalue_dutchroll2

    def get_idealized_spiral_eigenvalues(self, m, rho, V0, CL):
        mub = self.get_non_dim_masses(m, rho)[1]
        A = 2 * CL * (Clb * Cnr - Cnb * Clr)
        B = Clp * (CYb * Cnr + 4 * mub * Cnb)
        C = Cnp * (CYb * Clr + 4 * mub * Clb)
        eigenvalue_spiral = A / (B - C) * V0 / c
        return eigenvalue_spiral

    def get_shortperiod_eigenvalues(self, m, rho, V0, A):
        (
            eigenvalue_shortperiod1,
            eigenvalue_shortperiod2,
        ) = self.get_idealized_shortperiod_eigenvalues(m, rho, V0)
        eigenvalues, _ = self.get_eigenvalues_and_eigenvectors(A)

        eig1 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_shortperiod1))]
        eig2 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_shortperiod2))]

        return eig1, eig2

    def get_phugoid_eigenvalues(self, m, rho, V0, th0, A):
        eigenvalue_phugoid1, eigenvalue_phugoid2 = self.get_idealized_phugoid_eigenvalues(
            m, rho, V0, th0
        )
        eigenvalues, _ = self.get_eigenvalues_and_eigenvectors(A)

        eig1 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_phugoid1))]
```

```python
        eig2 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_phugoid2))]

        return eig1, eig2

    def get_aperiodicroll_eigenvalues(self, m, rho, V0, A):
        eigenvalue_aperiodicroll = self.get_idealized_aperiodicroll_eigenvalues(m, rho, V0)
        eigenvalues, _ = self.get_eigenvalues_and_eigenvectors(A)

        eig1 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_aperiodicroll))]

        return eig1

    def get_dutchroll_eigenvalues(self, m, rho, V0, A):
        eigenvalue_dutchroll1, eigenvalue_dutchroll2 = self.get_idealized_dutchroll_eigenvalues(
            m, rho, V0
        )
        eigenvalues, _ = self.get_eigenvalues_and_eigenvectors(A)

        eig1 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_dutchroll1))]
        eig2 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_dutchroll2))]

        return eig1, eig2

    def get_spiral_eigenvalues(self, m, rho, V0, CL, A):
        eigenvalue_spiral = self.get_idealized_spiral_eigenvalues(m, rho, V0, CL)
        eigenvalues, _ = self.get_eigenvalues_and_eigenvectors(A)

        eig1 = eigenvalues[np.argmin(np.abs(eigenvalues - eigenvalue_spiral))]

        return eig1

    def match_eigenvalues_asymmetric(self, A, m, rho, CL):
        eigenvalues = self.get_eigenvalues_and_eigenvectors(A)
        motions = ["Aperiodic Roll", "Dutch Roll", "Spiral"]
        matched_eigenvalues = []

        for i, motion in enumerate(motions):
            if motion == "Aperiodic Roll":
                real_eigenvalues = np.real(eigenvalues)
                abs_diff = np.abs(
                    real_eigenvalues - self.get_idealized_aperiodicroll_eigenvalues(m, rho)
                )
                index = np.argmin(abs_diff)
                matched_eigenvalues.append((real_eigenvalues[index], motion))

            elif motion == "Dutch Roll":
                abs_diff = np.abs(eigenvalues - self.get_idealized_dutchroll_eigenvalues(m, rho))
                index = np.argmin(abs_diff)
                matched_eigenvalues.append((eigenvalues[index], motion))
                matched_eigenvalues.append((eigenvalues[index].conjugate(), motion))

            elif motion == "Spiral":
                real_eigenvalues = np.real(eigenvalues)
                abs_diff = np.abs(
                    real_eigenvalues - self.get_idealized_spiral_eigenvalues(m, rho, CL)
                )
                index = np.argmin(abs_diff)
                matched_eigenvalues.append((real_eigenvalues[index], motion))

        return matched_eigenvalues
```

fd/simulation/constants.py

```python
# Change comment to switch between initial and improved coefficients

# from fd.simulation.constants_init import *

from fd.simulation.constants_improved import *

# from tests.test_simulation.constants_Cessna_Ce500 import *
```

fd/simulation/constants_improved.py

```python
# Citation 550 - Linear simulation

from math import pi

from fd.conversion import lbs_to_kg, in_to_m, kts_to_ms

g = 9.81  # [m/s^2] (gravity constant)

# Aircraft mass
mass_basic_empty = lbs_to_kg(9172.9)  # basic empty weight [kg]

# CG positions of components
xcgOEW = in_to_m(291.74)
xcgP = in_to_m(131)
xcgcoor = in_to_m(170)
xcg1 = in_to_m(214)
xcg2 = in_to_m(251)
xcg3 = in_to_m(288)

# Aircraft geometry
S = 30.00  # wing area [m^2]
Sh = 0.2 * S  # stabilizer area [m^2]
Sh_S = Sh / S  # [-]
lh = 0.71 * 5.968  # tail length [m]
c = 2.0569  # mean aerodynamic cord [m]
lh_c = lh / c  # [-]
b = 15.911  # wing span [m]
bh = 5.791  # stabilizer span [m]
A = b**2 / S  # wing aspect ratio [-]
Ah = bh**2 / Sh  # stabilizer aspect ratio [-]
Vh_V = 1  # [-]
ih = -2 * pi / 180  # stabilizer angle of incidence [rad]

# Constant values concerning atmosphere and gravity
rho0 = 1.2250  # air density at sea level [kg/m^3]
p0 = 101325  # air pressure at sea level [Pa]
Tempgrad = -0.0065  # temperature gradient in ISA [K/m]
Temp0 = 288.15  # temperature at sea level in ISA [K]
R = 287.05  # specific gas constant [m^2/s^2K]
gamma = 1.4  #
cas_stall = kts_to_ms(106)  # equivalent stall speed [m/s]

# Constant values concerning aircraft inertia
KX2 = 0.019
KZ2 = 0.042
KXZ = 0.002
KY2 = 1.25 * 1.114

# Aerodynamic constants
Cmac = 0  # Moment coefficient about the aerodynamic centre [-]
```

```
CNha = 2 * pi * Ah / (Ah + 2)  # Stabilizer normal force slope [-]
depsda = 4 / (A + 2)  # Downwash gradient [-]

# standard values
Ws = 60500  # standard weight from the assignment
fuel_flow_standard = 0.048  # [kg/s]

# Stability derivatives
# CX0 = W * sin(th0) / (0.5 * rho * V0**2 * S)
# CXu = -0.09500
CXu = -0.15
CXa = +0.47966  # Positive, see FD lecture notes
CXadot = +0.08330
CXq = -0.28170
CXde = -0.03728

# CZu = -0.37616
CZu = -0.45
# CZa = -5.74340
# CZa = -5.5
CZa = -5.2
# CZadot = -0.00350
CZadot = -0.005
CZq = -5.66290
CZde = -0.69612

Cm0 = +0.0297
# Cmu = +0.06990
Cmu = 0.1
Cmadot = +0.17800
Cmq = -8.79415
CmTc = -0.0064

CYb = -0.7500
CYbdot = 0
CYp = -0.0304
CYr = +0.8495
CYda = -0.0400
CYdr = +0.2300

# Clb = -0.10260
Clb = -0.09
# Clp = -0.71085
Clp = -0.6
# Clr = +0.23760
Clr = 0.19
# Clda = -0.23088
Clda = -0.3
# Cldr = +0.03440
Cldr = 0.0344

# Cnb = +0.1348
Cnb = 0.12
Cnbdot = 0
# Cnp = -0.0602
Cnp = -0.1
# Cnr = -0.2061
Cnr = -0.28
# Cnda = -0.0120
Cnda = -0.03
Cndr = -0.0939
```

```python

# Durations of the eigenmotions
# Used for data extraction and simulation
duration_phugoid = 120  # [s]
duration_short_period = 8  # [s]
duration_dutch_roll = 20  # [s]
duration_dutch_roll_yd = 10  # [s]
duration_aperiodic_roll = 12  # [s]
duration_spiral = 120  # [s]

# Lead times for eigenmotions w.r.t. timestamp
lead_phugoid = 1  # [s]
lead_short_period = 1  # [s]
lead_dutch_roll = 2  # [s]
lead_dutch_roll_yd = 3  # [s]
lead_aperiodic_roll = 1  # [s]
lead_spiral = 5  # [s]
```

fd/simulation/simulation.py

```python
import control.matlab as ml
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from pandas import DataFrame

from fd.analysis.aerodynamics import calc_CL
from fd.analysis.flight_test import FlightTest
from fd.plotting import format_plot
from fd.simulation import constants
from fd.simulation.aircraft_model import AircraftModel
from fd.structs import AerodynamicParameters


class Simulation:
    def __init__(self, model: AircraftModel):
        self.model = model

    def simulate_asymmetric(self, data, flip_input=False) -> DataFrame:
        t = data.index

        delta_a = data["delta_a"] - data["delta_a"].iloc[0]
        delta_r = data["delta_r"] - data["delta_a"].iloc[0]
        input = np.column_stack((delta_a, delta_r))
        if flip_input:
            input *= -1

        phi0 = data["phi"].iloc[0]
        p0 = data["p"].iloc[0]
        r0 = data["r"].iloc[0]
        state0_absolute = np.array([0, phi0, p0, r0])

        ABCD = self.model.get_state_space_matrices_asymmetric_from_df(data)

        sys = ml.ss(*ABCD)
        yout, t, xout = ml.lsim(sys, input, t)
        yout += state0_absolute
        result = np.hstack((np.transpose(t).reshape((len(t), 1)), yout))
        df_result = pd.DataFrame(result, columns=["t", "beta", "phi", "p", "r"])
        df_result = df_result.set_index("t", drop=True)

        return df_result
```

```python

    def simulate_symmetric(self, data):
        t = data.index

        delta_e = data["delta_e"] - data["delta_e"].iloc[0]
        input = delta_e

        theta0 = data["theta"].iloc[0]
        u_hat0 = 0
        alpha0 = data["alpha"].iloc[0]
        q0 = data["q"].iloc[0]
        state0_absolute = np.array([u_hat0, alpha0, theta0, q0])

        ABCD = self.model.get_state_space_matrices_symmetric_from_df(data)

        sys = ml.ss(*ABCD)
        yout, t, xout = ml.lsim(sys, input, t)
        yout += state0_absolute
        result = np.hstack((np.transpose(t).reshape((len(t), 1)), yout))
        df_result = pd.DataFrame(result, columns=["t", "u_hat", "alpha", "theta", "q"])
        df_result = df_result.set_index("t", drop=True)

        return df_result


if __name__ == "__main__":
    # sim = Simulation(
    #     AircraftModel(
    #         AerodynamicParameters(
    #             C_L_alpha=4.758556374647304,
    #             alpha_0=-0.023124783070063493,
    #             C_D_0=0.023439123324849084,
    #             # C_m_alpha=-0.5554065208385275,
    #             C_m_alpha=-0.5,
    #             C_m_delta=-1.3380975545274032,
    #             e=1.0713238368125688,
    #         )
    #     )
    # )
    ft = FlightTest("data/B24")
    df = ft.df_spiral
    aircraft_model = AircraftModel(ft.aerodynamic_parameters)
    sim = Simulation(aircraft_model)
    df_out = sim.simulate_asymmetric(df, flip_input=False)
    fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1)
    """
    y1 = "tas"
    y2 = "alpha"
    y3 = "theta"
    y4 = "q"
    """
    y1 = "beta"
    y2 = "phi"
    y3 = "p"
    y4 = "r"

    # ax1.plot(df_out.index, df_out["u_hat"] * df["tas"].iloc[0] + df["tas"].iloc[0])
    ax1.plot(df_out.index, df_out[y1])
    # ax1.plot(df_out.index, df[y1], color="black")
    ax1.set_ylabel(y1)
    ax2.plot(df_out.index, df_out[y2])
```

```
    ax2.plot(df_out.index, df[y2], color="black")
    # ax2.set_ylim(-0.2, 2.7)
    ax2.set_ylabel(y2)
    ax3.plot(df_out.index, df_out[y3])
    ax3.plot(df_out.index, df[y3], color="black")
    # ax3.set_ylim(-0.3, 0.25)
    ax3.set_ylabel(y3)
    ax4.plot(df_out.index, df_out[y4])
    ax4.plot(df_out.index, df[y4], color="black")
    ax4.set_ylim(-0.25, 0.3)
    ax4.set_ylabel(y4)
    # ax5.plot(df_out.index, df['delta_'])
    ax4.set_xlabel("t")

    format_plot()
    plt.show()
```

fd/simulation/constants_init.py

```
# Citation 550 - Linear simulation

from math import pi

from fd.conversion import lbs_to_kg, in_to_m, kts_to_ms

g = 9.81  # [m/s^2] (gravity constant)

# Aircraft mass
mass_basic_empty = lbs_to_kg(9172.9)  # basic empty weight [kg]

# CG positions of components
xcgOEW = in_to_m(291.74)
xcgP = in_to_m(131)
xcgcoor = in_to_m(170)
xcg1 = in_to_m(214)
xcg2 = in_to_m(251)
xcg3 = in_to_m(288)

# Aircraft geometry
S = 30.00  # wing area [m^2]
Sh = 0.2 * S  # stabilizer area [m^2]
Sh_S = Sh / S  # [-]
lh = 0.71 * 5.968  # tail length [m]
c = 2.0569  # mean aerodynamic cord [m]
lh_c = lh / c  # [-]
b = 15.911  # wing span [m]
bh = 5.791  # stabilizer span [m]
A = b**2 / S  # wing aspect ratio [-]
Ah = bh**2 / Sh  # stabilizer aspect ratio [-]
Vh_V = 1  # [-]
ih = -2 * pi / 180  # stabilizer angle of incidence [rad]

# Constant values concerning atmosphere and gravity
rho0 = 1.2250  # air density at sea level [kg/m^3]
p0 = 101325  # air pressure at sea level [Pa]
Tempgrad = -0.0065  # temperature gradient in ISA [K/m]
Temp0 = 288.15  # temperature at sea level in ISA [K]
R = 287.05  # specific gas constant [m^2/s^2K]
gamma = 1.4  #
cas_stall = kts_to_ms(106)  # equivalent stall speed [m/s]

# Constant values concerning aircraft inertia
```

```
KX2 = 0.019
KZ2 = 0.042
KXZ = 0.002
KY2 = 1.25 * 1.114

# Aerodynamic constants
Cmac = 0  # Moment coefficient about the aerodynamic centre [-]
CNha = 2 * pi * Ah / (Ah + 2)  # Stabilizer normal force slope [-]
depsda = 4 / (A + 2)  # Downwash gradient [-]

# standard values
Ws = 60500  # standard weight from the assignment
fuel_flow_standard = 0.048  # [kg/s]

# Stability derivatives
# CX0 = W * sin(th0) / (0.5 * rho * V0**2 * S)
CXu = -0.09500
CXa = +0.47966  # Positive, see FD lecture notes
CXadot = +0.08330
CXq = -0.28170
CXde = -0.03728

CZu = -0.37616
CZa = -5.74340
CZadot = -0.00350
CZq = -5.66290
CZde = -0.69612

Cm0 = +0.0297
Cmu = +0.06990
Cmadot = +0.17800
Cmq = -8.79415
CmTc = -0.0064

CYb = -0.7500
CYbdot = 0
CYp = -0.0304
CYr = +0.8495
CYda = -0.0400
CYdr = +0.2300

Clb = -0.10260
Clb = -0.13
Clp = -0.71085
Clr = +0.23760
Clda = -0.23088
Cldr = +0.03440

Cnb = +0.1348
Cnbdot = 0
Cnp = -0.0602
Cnr = -0.2061
Cnda = -0.0120
Cndr = -0.0939

# Durations of the eigenmotions
# Used for data extraction and simulation
duration_phugoid = 120  # [s]
duration_short_period = 8  # [s]
duration_dutch_roll = 20  # [s]
duration_dutch_roll_yd = 10  # [s]
```

```
duration_aperiodic_roll = 12  # [s]
duration_spiral = 120  # [s]

# Lead times for eigenmotions w.r.t. timestamp
lead_phugoid = 1  # [s]
lead_short_period = 1  # [s]
lead_dutch_roll = 2  # [s]
lead_dutch_roll_yd = 3  # [s]
lead_aperiodic_roll = 1  # [s]
lead_spiral = 5  # [s]
```

fd/analysis/flight_test.py

```
from pathlib import Path

import numpy as np

from fd.analysis.aerodynamic_plots import (
    plot_elevator_control_force,
    plot_elevator_trim_curve,
    plot_cl_alpha,
    plot_cl_cd,
)
from fd.analysis.aerodynamics import (
    estimate_CL_alpha,
    estimate_CD0_e,
    estimate_Cmalpha,
    calc_Cmdelta,
)
from fd.analysis.data_sheet import DataSheet, AveragedDataSheet
from fd.analysis.ftis_measurements import FTISMeasurements
from fd.simulation import constants
from fd.simulation.constants import (
    duration_phugoid,
    duration_dutch_roll,
    duration_short_period,
    duration_dutch_roll_yd,
    duration_aperiodic_roll,
    duration_spiral,
    lead_spiral,
    lead_aperiodic_roll,
    lead_dutch_roll_yd,
    lead_dutch_roll,
    lead_short_period,
    lead_phugoid,
)
from fd.structs import AerodynamicParameters


class FlightTest:
    """Stores raw measurements, data sheet and estimated parameters"""

    ftis_measurements: FTISMeasurements
    data_sheet: AveragedDataSheet
    aerodynamic_parameters: AerodynamicParameters

    def __init__(self, data_path: str):
        self.data_sheet = AveragedDataSheet(
            {p.name: DataSheet(str(p)) for p in Path(data_path).glob("**/*.xlsx")}
        )
        self.ftis_measurements = FTISMeasurements(data_path, self.data_sheet.mass_initial)
        self._estimate_aerodynamic_parameters()
```

```
        self.data_sheet.add_reduced_elevator_deflection_timeseries(
            self.aerodynamic_parameters.C_m_delta
        )

    def _estimate_aerodynamic_parameters(self):
        C_L_alpha, _, alpha_0 = estimate_CL_alpha(self.df_clcd["C_L"], self.df_clcd["alpha"])
        C_D_0, e = estimate_CD0_e(self.df_clcd["C_D"], self.df_clcd["C_L"])

        cg_aft = self.df_cg_shift.iloc[0]
        cg_front = self.df_cg_shift.iloc[1]
        C_m_delta = calc_Cmdelta(
            cg_aft["x_cg"],
            cg_front["x_cg"],
            cg_aft["delta_e"],
            cg_front["delta_e"],
            self.df_cg_shift["W"].mean(),
            self.df_cg_shift["tas"].mean(),
            self.df_cg_shift["rho"].mean(),
        )

        C_m_alpha = estimate_Cmalpha(
            self.df_elevator_trim["alpha"], self.df_elevator_trim["delta_e"], C_m_delta
        )

        self.aerodynamic_parameters = AerodynamicParameters(
            C_L_alpha, alpha_0, C_D_0, C_m_alpha, C_m_delta, e
        )

    def make_aerodynamic_plots(self):
        plot_cl_alpha(
            self.df_clcd["C_L"],
            self.df_clcd["alpha"],
            self.aerodynamic_parameters.C_L_alpha,
            self.aerodynamic_parameters.alpha_0,
        )

        plot_cl_cd(
            self.df_clcd["C_L"],
            self.df_clcd["C_D"],
            self.aerodynamic_parameters.C_D_0,
            self.aerodynamic_parameters.e,
        )

        plot_elevator_trim_curve(
            self.df_elevator_trim["delta_e_reduced"],
            self.df_elevator_trim["alpha"],
            self.df_elevator_trim["cas_reduced"],
            self.df_cg_shift["delta_e_reduced"].iloc[1],
            self.df_cg_shift["alpha"].iloc[1],
            self.df_cg_shift["cas_reduced"].iloc[1],
            constants.Cm0,
            self.aerodynamic_parameters.C_m_delta,
            constants.cas_stall,
        )

        plot_elevator_control_force(
            self.df_elevator_trim["F_e_reduced"],
            self.df_elevator_trim["cas_reduced"],
            constants.cas_stall,
        )

```

```
    @property
    def df_ftis(self):
        return self.ftis_measurements.df

    @property
    def df_clcd(self):
        return self.data_sheet.df_clcd

    @property
    def df_elevator_trim(self):
        return self.data_sheet.df_elevator_trim

    @property
    def df_cg_shift(self):
        return self.data_sheet.df_cg_shift

    @property
    def df_phugoid(self):
        return self._get_maneuver_df(
            self.data_sheet.timestamp_phugoid, duration_phugoid, lead_phugoid
        )

    @property
    def df_short_period(self):
        return self._get_maneuver_df(
            self.data_sheet.timestamp_short_period, duration_short_period, lead_short_period
        )

    @property
    def df_dutch_roll(self):
        return self._get_maneuver_df(
            self.data_sheet.timestamp_dutch_roll, duration_dutch_roll, lead_dutch_roll
        )

    @property
    def df_dutch_roll_yd(self):
        return self._get_maneuver_df(
            self.data_sheet.timestamp_dutch_roll_yd, duration_dutch_roll_yd, lead_dutch_roll_yd
        )

    @property
    def df_aperiodic_roll(self):
        return self._get_maneuver_df(
            self.data_sheet.timestamp_aperiodic_roll, duration_aperiodic_roll, lead_aperiodic_roll
        )

    @property
    def df_spiral(self):
        return self._get_maneuver_df(self.data_sheet.timestamp_spiral, duration_spiral, lead_spiral)

    def _get_maneuver_df(self, timestamp: float, duration: float, lead: float):
        """Get rows from FTIS data corresponding to a certain maneuver, identified by start timestamp and durat
        df = self.df_ftis.loc[timestamp - lead : timestamp + duration].copy()
        df.index = np.round(df.index - df.index[0], 2)
        df["time_min"] -= df["time_min"][0]
        return df
```

fd/analysis/reduced_values.py

```
import numpy as np

from fd.simulation import constants
```

```python


def calc_reduced_equivalent_V(Ve, W):
    """

    Args:
        Ve (float): Equivalent velocity[m/s]
        W (float): Weight of the aircraft[N]

    Returns (float): Reduced equivalent airspeed[m/s]

    """
    return Ve * np.sqrt(constants.Ws / W)


def calc_reduced_elevator_deflection(delta_e_meas, Cmdelta, Tcs, Tc):
    """

    Args:
        delta_e_meas (float): The measured elevator deflection[deg]
        Cmdelta (float): Change in moment coefficient due to elevator defection[-]
        Tcs (float): Thrust coefficient in standard conditions[-]
        Tc (float): Thrust coefficient for conditions used[-]

    Returns (float): The reduced elevator deflection angle[deg]

    """

    return delta_e_meas - constants.CmTc / Cmdelta * (Tcs - Tc)


def calc_reduced_stick_force(Fe_aer, W):
    """

    Args:
        Fe_aer (float): The measured stick force[N]
        W (float): The actual weight of the aircraft[N]

    Returns (float): The reduced stick force[N]

    """

    return Fe_aer * constants.Ws / W
```

fd/analysis/ftis_measurements.py

```python
from fd.analysis.util import add_common_derived_timeseries
from fd.conversion import (
    lbshr_to_kgs,
    lbs_to_kg,
    ft_to_m,
    kts_to_ms,
    C_to_K,
    deg_to_rad,
    degs_to_rads,
)
from fd.io import load_ftis_measurements
from fd.simulation.constants import g

COLUMNS = {
    "vane_AOA": "alpha",
    "elevator_dte": "delta_t_e",
```

```
    "lh_engine_FMF": "fuel_flow_left",
    "rh_engine_FMF": "fuel_flow_right",
    "lh_engine_FU": "fuel_used_left",
    "rh_engine_FU": "fuel_used_right",
    "column_Se": "s_e",
    "column_fe": "F_e",
    "delta_a": "delta_a",
    "delta_e": "delta_e",
    "delta_r": "delta_r",
    "Dadc1_bcAlt": "h",
    "Dadc1_mach": "M",
    "Dadc1_cas": "cas",
    "Dadc1_tas": "tas",
    "Dadc1_sat": "T_static",
    "Dadc1_tat": "T_total",
    "Ahrs1_Roll": "phi",
    "Ahrs1_Pitch": "theta",
    "Ahrs1_bRollRate": "p",
    "Ahrs1_bPitchRate": "q",
    "Ahrs1_bYawRate": "r",
    "Fms1_trueHeading": "chi",
    "Ahrs1_bLongAcc": "acc_x",
    "Ahrs1_bLatAcc": "acc_y",
}


class FTISMeasurements:
    def __init__(self, data_path: str, mass_initial: float):
        self.df = load_ftis_measurements(data_path)
        self._process_ftis_measurements()
        self._add_derived_timeseries(mass_initial)

    def _process_ftis_measurements(self):
        self.df = self.df[COLUMNS.keys()].rename(columns=COLUMNS)

        # Convert to SI units
        self.df["alpha"] = deg_to_rad(self.df["alpha"])
        self.df["delta_e"] = deg_to_rad(self.df["delta_e"])
        self.df["delta_t_e"] = deg_to_rad(self.df["delta_t_e"])
        self.df["delta_a"] = deg_to_rad(self.df["delta_a"])
        self.df["delta_r"] = deg_to_rad(self.df["delta_r"])
        self.df["s_e"] = deg_to_rad(self.df["s_e"])
        self.df["phi"] = deg_to_rad(self.df["phi"])
        self.df["theta"] = deg_to_rad(self.df["theta"])
        self.df["p"] = degs_to_rads(self.df["p"])
        self.df["q"] = degs_to_rads(self.df["q"])
        self.df["r"] = degs_to_rads(self.df["r"])
        self.df["fuel_flow_left"] = lbshr_to_kgs(self.df["fuel_flow_left"])
        self.df["fuel_flow_right"] = lbshr_to_kgs(self.df["fuel_flow_right"])
        self.df["fuel_used_left"] = lbs_to_kg(self.df["fuel_used_left"])
        self.df["fuel_used_right"] = lbs_to_kg(self.df["fuel_used_right"])
        self.df["h"] = ft_to_m(self.df["h"])
        self.df["cas"] = kts_to_ms(self.df["cas"])
        self.df["tas"] = kts_to_ms(self.df["tas"])
        self.df["T_static"] = C_to_K(self.df["T_static"])
        self.df["T_total"] = C_to_K(self.df["T_total"])
        self.df["acc_x"] = self.df["acc_x"] * g
        self.df["acc_y"] = self.df["acc_y"] * g

        return self.df

```

```
    def _add_derived_timeseries(self, mass_initial: float):
        self.df["time_min"] = self.df.index / 60
        self.df["m"] = mass_initial - self.df["fuel_used_left"] - self.df["fuel_used_right"]
        self.df = add_common_derived_timeseries(self.df)
```

fd/analysis/util.py

```
import pandas as pd

from fd.analysis.thermodynamics import (
    calc_mach,
    calc_static_temperature,
    calc_static_pressure,
    calc_density,
)
from fd.simulation import constants


def add_common_derived_timeseries(df: pd.DataFrame) -> pd.DataFrame:
    df["W"] = df["m"] * constants.g

    df["M"] = df.apply(lambda row: calc_mach(row["h"], row["cas"]), axis=1)
    df["T_static"] = df.apply(lambda row: calc_static_temperature(row["T_total"], row["M"]), axis=1)
    df["p_static"] = df.apply(lambda row: calc_static_pressure(row["h"]), axis=1)
    df["rho"] = df.apply(lambda row: calc_density(row["p_static"], row["T_static"]), axis=1)

    return df
```

fd/analysis/center_of_gravity.py

```
import numpy as np
import scipy.stats as stats

from fd import conversion as con
from fd.simulation import constants


def lin_moment_mass():
    """

    Returns (float, float): Slope of the linear moment-vs-mass function and the intercept.

    """

    mass = np.linspace(100, 4900, 49)
    mass = np.append(mass, 5008.0)
    moment = np.array(
        [
            29816,
            59118,
            87908,
            116542,
            144840,
            173253,
            201480,
            229884,
            258192,
            286630,
            315018,
            343452,
            371852,
            400323,
```

```
            428776,
            457224,
            485656,
            514116,
            542564,
            570990,
            599404,
            627847,
            656282,
            684696,
            713100,
            741533,
            769960,
            798434,
            826906,
            855405,
            883904,
            912480,
            941062,
            969697,
            998340,
            1027008,
            1055684,
            1084387,
            1113100,
            1141820,
            1170550,
            1199331,
            1228118,
            1256904,
            1285686,
            1314473,
            1343248,
            1372056,
            1400846,
            1432034,
        ]
    )

    massSI = con.lbs_to_kg(mass)
    momentSI = con.inchpound_to_kgm(moment)

    result = stats.theilslopes(momentSI, massSI, alpha=0.99)
    # plt.scatter(massSI, momentSI)
    # plt.plot(massSI, result[0] * massSI + result[1])
    # plt.show()

    return result[0], result[1]


def calc_cg_position(
    mfuel, massP1, massP2, masscoor, mass1L, mass1R, mass2L, mass2R, mass3L, mass3R, shift=False
):
    mtot = (
        constants.mass_basic_empty
        + mfuel
        + massP1
        + massP2
        + masscoor
        + mass1L
        + mass1R
```

```
            + mass2L
            + mass2R
            + mass3L
            + mass3R
        )
        slope, intersect = lin_moment_mass()
        momentfuel = slope * mfuel + intersect
        if shift:
            xcg = (
                momentfuel
                + (massP1 + massP2) * constants.xcgP
                + (constants.xcgP + (constants.xcgcoor - constants.xcgP) * 2 / 3) * mass3R
                + masscoor * constants.xcgcoor
                + (mass1R + mass1L) * constants.xcg1
                + (mass2R + mass2L) * constants.xcg2
                + (mass3L) * constants.xcg3
                + constants.mass_basic_empty * constants.xcgOEW
            ) / mtot
        else:
            xcg = (
                momentfuel
                + (massP1 + massP2) * constants.xcgP
                + masscoor * constants.xcgcoor
                + (mass1R + mass1L) * constants.xcg1
                + (mass2R + mass2L) * constants.xcg2
                + (mass3R + mass3L) * constants.xcg3
                + constants.mass_basic_empty * constants.xcgOEW
            ) / mtot
        return xcg
```

fd/analysis/aerodynamics.py

```
import math

import numpy as np
import scipy.stats as stats

from fd.simulation import constants


def calc_true_V(T, M):
    """

    Args:
        T (float): Static temperature[K]
        M (float): Mach number[-]

    Returns (float): True airspeed[m/s]

    """
    return M * np.sqrt(constants.gamma * constants.R * T)


def calc_dynamic_pressure(V: float, rho: float) -> float:
    """
    Calculate dynamic pressure

    Args:
        V: velocity (CAS or TAS) [m/s]
        rho: air density (sea level or actual) [kg/m^3]

    Returns:
```

```
        Dynamic pressure [Pa]
    """
    return rho * V**2 / 2


def calc_equivalent_V(Vt, rho):
    """

    Args:
        Vt (float): True airspeed[m/s]
        rho (float): Density[kg/m^3]

    Returns (float): Equivalent airspeed[m/s]

    """
    return Vt * np.sqrt(rho / constants.rho0)


def calc_CL(W: float, V: float, rho: float, S=constants.S) -> float:
    """
    Calculate CL for a given combination of W, rho, V and S.
    Args:
        W (array_like): Weight [N]
        rho (float): Air density [kg/m3]
        V (array_like): True airspeed [m/s]
        S (float): Surface area [m2]

    Returns:
        (array_like): CL [-]
    """

    return W / (calc_dynamic_pressure(V, rho) * S)


def estimate_CL_alpha(CL: float, alpha: float) -> tuple[float, float, float]:
    """
    Calculate the slope, CL-intercept and alpha intercept of the CL-alpha plot using a Theil-Sen robust linear
    Args:
        CL (array_like): CL [-]
        alpha (array_like): angle of attack [deg]

    Returns:
        CLalpha (float): slope of CL-alpha plot [1/deg]
        CL_alpha_equals0 (float): CL at alpha = 0
        alpha_0 (float): alpha at CL = 0
    """
    # CLalpha, CL_alpha_equals0, _, _ = stats.theilslopes(CL, alpha, alpha=0.99)
    CLalpha, CL_alpha_equals0, _, _, _ = stats.linregress(alpha, CL)
    alpha_0 = -CL_alpha_equals0 / CLalpha

    return CLalpha, CL_alpha_equals0, alpha_0


def calc_CD(T: float, V: float, rho: float, S: float = constants.S) -> float:
    """
    This function calculates the drag coefficient CD[-] based on the thrust.

    Args:
        T (array_like): Thrust[N].
        V (array_like): True airspeed[m/s].

```

```python
    Returns:
        CD (array_like): Drag coefficient CD[-].

    """
    return T / (calc_dynamic_pressure(V, rho) * S)


def estimate_CD0_e(CD: list, CL: list) -> tuple[float, float]:
    """
    This function uses the parabolic drag formula to calculate the zero lift drag, CD0[-], and the oswald
    efficiency factor, e[-].

    Args:
        CD (list): The drag coefficient CD[-].
        CL (list): The lift coefficient CL[-].

    Returns:
        CD0, e (float, float): Zero lift drag coefficient, CD0[-], oswald efficiency factor, e[-].

    """

    # slope, CD0, _, _ = stats.theilslopes(CD, CL**2, alpha=0.99)
    slope, CD0, _, _, _ = stats.linregress(CL**2, CD)
    e = 1 / (math.pi * constants.A * slope)

    return CD0, e


def estimate_Cmalpha(alpha, delta_e, Cmdelta):
    """

    Args:
        alpha (array_like): Angle of attack[deg]
        delta_e (array_like): Elevator deflection[deg]
        Cmdelta (float): Change in moment coefficient due to elevator deflection[-]

    Returns (float): Change in moment coefficient due to angle of attack[-]

    """

    # slope, _, _, _ = stats.theilslopes(delta_e, alpha, alpha=0.99)
    slope, _, _, _, _ = stats.linregress(alpha, delta_e)
    return -slope * Cmdelta


def calc_Cmdelta(
    xcg1: float,
    xcg2: float,
    deltae1: float,
    deltae2: float,
    W: float,
    V: float,
    rho: float,
):
    """

    Args:
        xcg1 (float): X-position of the center of gravity during the first test.(aft cg)[m]
        xcg2 (float): X-position of the center of gravity during the second test.(front cg)[m]
        deltae1 (float): Deflection of the elevator during the first test.[deg]
        deltae2 (float): Deflection of the elevator during the second test.[deg]
```

```
        W (float): Weight of the aircraft during the tests.[N]
        V (float): Velocity of the aircraft during the tests.[m/s]
        rho (float): Air density.[kg/m^3]

    Returns: Cmdelta (float): The moment coefficient change due to the elevator deflection.[-]

    """
    Delta_cg = xcg2 - xcg1
    Delta_delta_e = deltae2 - deltae1
    C_N = W / (calc_dynamic_pressure(V, rho) * constants.S)
    return -1 / Delta_delta_e * C_N * Delta_cg / constants.c
```

fd/analysis/aerodynamic_plots.py

```
import math
from functools import partial

import numpy as np
import scipy.stats as stats

from fd.analysis.aerodynamics import calc_dynamic_pressure
from fd.plotting import *
from fd.simulation import constants
from fd.simulation.constants import rho0


def plot_cl_alpha(CL, alpha, Clalpha, alpha0):
    """
    Plotting og the lift slope
    Args:
        CL (array_like): Lift coefficient CL[-]
        alpha (array_like): Angle of attack alpha[rad]
        Clalpha (float): slope of the linear part of the CL-alpha curve
        alpha0 (float): crossing of the CL-alpha curve with the alpha axis

    Returns:

    """
    fig, ax = plt.subplots(figsize=(12, 6))

    aa = np.linspace(alpha0, max(alpha), 20)

    ax.plot(
        aa,
        Clalpha * (aa - alpha0),
        "r",
        label="Best fit ($C_{L_\\alpha}$ = "
        + f"{Clalpha:.3} 1/rad"
        + ", $\\alpha_0$ = "
        + f"{alpha0:.3} rad)",
    )
    ax.scatter(alpha, CL, marker="x", color="black", s=50, label="Data")

    ax.set_xlabel(r"$\alpha$")
    ax.set_ylabel("$C_L$")

    ax.legend()

    format_plot()
    save_plot("data/", "cl_alpha")
    plt.show()

```

```python

def plot_cl_cd(CL, CD, CD0, e):
    """
    Plotting of the drag polar
    Args:
        CL (array_like): The lift coefficient[-]
        CD (array_like): The drag coefficient[-]
        CD0 (float): Zero lift drag coeffcient[-]
        e (float): The oswald efficiency factor[-]

    Returns:

    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

    yy = np.linspace(0, max(CL), 20)

    ax1.plot(CD0 + yy / (math.pi * constants.A * e), yy, "r")
    ax1.scatter(CD, CL**2, marker="x", color="black", s=50)
    ax1.set_xlabel("$C_D$")
    ax1.set_ylabel("$C_L^2$")

    ax2.plot(
        CD0 + yy**2 / (math.pi * constants.A * e),
        yy,
        "r",
        label="Best fit ($C_{D_0}$ = " + f"{CD0:.3}" + ", $e$ = " + f"{e:.3})",
    )
    ax2.scatter(CD, CL, marker="x", color="black", s=50, label="Data")
    ax2.set_xlabel("$C_D$")
    ax2.set_ylabel("$C_L$")
    ax2.legend()

    format_plot()
    save_plot("data/", "cl_cd")
    plt.show()


# Elevator curves
def plot_delta_e_V_inv_squared(
    delta_e, V_inv_squared, xlabel_input="$1/V^2$ [1/(m/s)$^2$]", ylabel_input="$\delta_e$ [rad]"
):
    """
    Plotting of the elevator trim curve for delta_e vs 1/v^2. Should be proportional to 1/v^2, so linear.

    Args:
        delta_e (array_like): elevator deflection [rad]
        V_inv_squared (array_like): reciprocal of airspeed squared [1/(m/s)^2]

    Returns:
        delta vs 1/V^2 elevator trim curve plot
    """
    fig, ax = plt.subplots(figsize=(12, 6))

    # slope, y_intercept, _, _ = stats.theilslopes(delta_e, V_inv_squared, alpha=0.99)
    slope, y_intercept, _, _, _ = stats.linregress(V_inv_squared, delta_e)
    xx = np.linspace(0, max(V_inv_squared) * 1.05, 2)
    plt.plot(xx, slope * xx + y_intercept, "r")
    plt.scatter(V_inv_squared, delta_e, marker="x", color="black", s=50)
    plt.xlabel(xlabel_input)
    plt.ylabel(ylabel_input)
```

```python

    plt.gca().invert_yaxis()
    format_plot()
    plt.show()

    return slope, y_intercept


# [TODO: add fully controlled parameters to plots once these are know and its known how we want to do this]
def plot_elevator_trim_curve(
    delta_e,
    alpha,
    cas,
    delta_e_front,
    alpha_front,
    cas_front,
    C_m_0,
    C_m_delta_e,
    cas_stall,
):
    """
    Plotting of the elevator trim curve. Depending on if V or V_e is used and the deflection or reduced deflect
    the non-reduced or reduced elevator trim curve plots can be obtained. Should be proportional to 1/v^2

    Args:
        delta_e (array_like): elevator deflection [rad]
        alpha (array_like):  angle of attack [rad]
        cas (array_like): CAS [m/s]
        cas_stall (float): stall CAS [m/s]
    """
    # slope_V_inv_sq, y_intercept_V_inv_sq, _, _ = stats.theilslopes(
    #     delta_e, 1 / cas**2, alpha=0.99
    # )
    slope_V_inv_sq, y_intercept_V_inv_sq, _, _, _ = stats.linregress(1 / cas**2, delta_e)
    delta_e_asymptote = -C_m_0 / C_m_delta_e

    fig, (ax_alpha, ax_V) = plt.subplots(1, 2, figsize=(12, 6), sharey="all")

    # Delta_e vs alpha
    # slope_alpha, y_intercept_alpha, _, _ = stats.theilslopes(delta_e, alpha, alpha=0.99)
    slope_alpha, y_intercept_alpha, _, _, _ = stats.linregress(alpha, delta_e)
    xx_alpha = np.linspace(0, max(alpha) * 1.05, 2)
    ax_alpha.plot(xx_alpha, slope_alpha * xx_alpha + y_intercept_alpha, "r", label=r"$\alpha$")

    # Delta_e vs alpha - alpha0
    alpha0 = (y_intercept_V_inv_sq - y_intercept_alpha) / slope_alpha
    # slope_alpha0, y_intercept_alpha0, _, _ = stats.theilslopes(delta_e, alpha - alpha0, alpha=0.99)
    slope_alpha0, y_intercept_alpha0, _, _, _ = stats.linregress(alpha - alpha0, delta_e)
    xx_alpha0 = np.linspace(0, max(alpha - alpha0) * 1.05, 2)
    ax_alpha.plot(
        xx_alpha0, slope_alpha0 * xx_alpha0 + y_intercept_alpha0, "b", label=r"$\alpha-\alpha_0$"
    )
    ax_alpha.scatter(alpha - alpha0, delta_e, marker="x", color="black", s=50)
    ax_alpha.set_xlabel(r"$\alpha$ [rad]")
    ax_alpha.set_ylabel(r"$\delta_e^*$ [rad]")

    ax_alpha.scatter(alpha, delta_e, marker="x", color="black", s=50, label="Regular CG")
    ax_alpha.scatter(alpha_front, delta_e_front, color="r", marker="x", s=50, label="Forward CG")

    ax_alpha.legend()

```

```
    # Delta_e vs V
    xx_V = np.linspace(0.95 * cas_stall, 200, 100)
    ax_V.plot(xx_V, y_intercept_V_inv_sq + slope_V_inv_sq * 1 / xx_V**2, "r")
    ax_V.scatter(cas, delta_e, marker="x", color="black", s=50)
    ax_V.scatter(cas_front, delta_e_front, color="r", marker="x", s=50)
    ax_V.axvline(x=cas_stall, color="b", linestyle="--", label="$V_{stall}$")
    ax_V.axhline(
        y=delta_e_asymptote,
        linestyle=":",
        color="black",
        label=r"$\delta_{e,V \to \infty}}$ = " + f"{delta_e_asymptote:.3} rad",
    )
    ax_V.set_xlabel(r"$\tilde V_e$ [m/s]")
    ax_V.legend()
    ax_V.invert_yaxis()

    format_plot()
    save_plot("data/", "elevator_trim_curve")
    plt.show()


def plot_elevator_control_force(F_e, cas, cas_stall):
    """
    Plotting of the elevator control force curve. Depending on if V or V_e is used and the F_e or reduced F_e,
    the non-reduced or reduced elevator control force curve plots can be obtained.

    The elevator control force is plotted against dynamic pressure and airspeed.

    Args:
        F_e (array_like): Control stick force [N]
        cas (array_like): CAS [m/s]
        cas_stall (float): stall CAS [m/s]
    """
    fig, (ax_q, ax_V) = plt.subplots(1, 2, figsize=(12, 6), sharey="all")

    dynamic_pressure_stall = calc_dynamic_pressure(cas_stall, rho0)
    dynamic_pressure = partial(calc_dynamic_pressure, rho=rho0)(cas)
    # slope, y_intercept, _, _ = stats.theilslopes(F_e, dynamic_pressure, alpha=0.99)
    slope, y_intercept, _, _, _ = stats.linregress(dynamic_pressure, F_e)

    # F_e vs q
    xx_q = np.linspace(dynamic_pressure_stall, max(dynamic_pressure) * 1.05, 2)
    ax_q.plot(xx_q, y_intercept + slope * xx_q, "r")
    ax_q.scatter(dynamic_pressure, F_e, marker="x", color="black", s=50)
    ax_q.axvline(
        x=dynamic_pressure_stall,
        color="black",
        linestyle="--",
        label=r"$\frac{1}{2}\rho_0 V_{stall}^2$",
    )

    xx2 = np.linspace(0, dynamic_pressure_stall, 2)
    ax_q.plot(xx2, y_intercept + slope * xx2, "r", linestyle=":")
    ax_q.set_xlabel(r"$\frac{1}{2}\rho_0 \tilde V_e^2$ [Pa]")
    ax_q.set_ylabel(r"$F_e^*$ [N]")
    ax_q.legend()

    # F_e vs V
    xx_V = np.linspace(cas_stall, 95, 100)
    ax_V.plot(
        xx_V,
```

```
        y_intercept + slope * partial(calc_dynamic_pressure, rho=rho0)(xx_V),
        "r",
        label="Best fit",
    )
    ax_V.scatter(cas, F_e, marker="x", color="black", s=50, label="Data")
    ax_V.axvline(x=cas_stall, color="black", linestyle="--", label="$V_{stall}$")

    xx_V_stall = np.linspace(0, cas_stall, 100)
    ax_V.plot(
        xx_V_stall,
        y_intercept + slope * partial(calc_dynamic_pressure, rho=rho0)(xx_V_stall),
        "r",
        linestyle=":",
    )
    ax_V.set_xlabel(r"$\tilde V_e$ [m/s]")
    ax_V.legend()
    ax_V.invert_yaxis()

    format_plot(zeroline=True)
    save_plot("data/", "elevator_force_curve")
    plt.show()
```

fd/analysis/thrust.py

```
import subprocess
import tempfile
from math import exp, sqrt, pow
from pathlib import Path
from typing import Optional

import pandas as pd

from fd.analysis.aerodynamics import calc_dynamic_pressure
from fd.simulation import constants

ittmax = 730
Ne = 2
NL = 104
tto = 0
pto = 0
po = 0
t_o = 0
T_isa = 0
rho = 0
nu = 0
mu = 0
a_s = 0
elpcc = 0
r = 0
tel = 0
mh = 0
mc = 0
wh = 0
wc = 0
tt5 = 0
mf = 0
tet = 0
tt1 = 0
tt2 = 0
tt3 = 0
tt4 = 0
tt6 = 0
```

```
tt7 = 0
ttb = 0
t8 = 0
t9 = 0
pa = 0
pt1 = 0
pt2 = 0
itt = 0
s = 0
pt3 = 0
pt4 = 0
pt5 = 0
pt6 = 0
pt7 = 0
bpr = 0
dnct = 0
nr = 0
nf = 0
nc = 0
nb = 0
dpt = 0
nt = 0
nm = 0
nnc = 0
nnh = 0
mht3p3 = 0
ehpc = 0
mhd = 0
tt3d = 0
b = 0
thetat = 0
c = 0
d = 0
Ah = 0
Ac = 0
pce = 0
phe = 0
p3krit = 0
p7krit = 0
dncn = 0
dhnr = 0
dtnr = 0
dhmnr = 0
vo = 0
tt4max = 0
itrel = 0
dmhnr = 0
theta = 0
NLcor = 0
NLcort1 = 0
NLcort = 0
dncn1 = 0
elpc = 0
elpc1 = 0
nf1 = 0
fi = 0
deltem = 0
delmh = 0
Tn = 0
mfi = 0
```

```
def atmos(h, M, T_static):
    global T_isa, po, t_o, tto, pto, a_s
    T_isa = 288.15 - 0.0065 * h
    po = 101325 * pow(T_isa / 288.15, 5.256)
    if h >= 11000:
        T_isa = 216.65
        po = 22631.23 * exp(-9.80665 / 216.65 / 287.05 * (h - 11000))
    t_o = T_static
    # rho=po/287.05/temp
    tto = t_o * (1 + 0.2 * M**2)
    pto = po * (1 + 0.2 * M**2) ** 3.5
    a_s = sqrt(1.4 * 287.05 * t_o)
    # mu=0.000017894*pow((t_o/288.15),1.50)*(288.15+110.4)/(to+110.4)
    # nu=mu/rho
    return T_isa


def stuw(h, M, dtemp):
    global ittmax, Ne, NL, tto, pto, po, t_o, T_isa, rho, nu, mu, a_s, elpcc, r, tel, mh, mc, wh, wc, tt5, mf,

    theta = t_o / 288.15
    thetat = tto / 288.15
    NLcor = NL / sqrt(theta)
    NLcort = NL / sqrt(thetat)
    #    Rendement van de verschillende componenten
    #    van de motor
    #   dnc = 0#
    dncn = -0.1209875 * (NLcort - 104) + 0.0005625 * (pow(NLcort, 2) - pow(104, 2))
    dnct = (4.5 * pow(10, -4) * dtemp + 1.5 * pow(10, -5) * pow(dtemp, 2)) * (
        1 - 0.69263 * h / 2438.4
    )
    dtnr = 0.000032 * pow(dtemp, 2) + 0.000233 * dtemp
    dhnr = -0.02 * (h / 2438.4) * (dtemp / 30)
    dhmnr = (
        -0.001 * pow((h / 3048), 4)
        + 0.0081667 * pow((h / 3048), 3)
        - 0.0085 * pow((h / 3048), 2)
        + 0.0023333 * (h / 3048)
        - 0.001
    )
    dmhnr = 0
    if M >= 0.2:
        dmhnr = (-0.12 * pow(M, 2) + 0.024 * M) * (-0.0625 * pow((h / 3048), 2) + 0.5 * h / 3048)
    nr = 0.989
    nf = 0.7
    nc = 0.73
    nb = 0.972
    dpt = 0.95
    nt = 0.86
    nm = 0.985
    nnc = 0.925
    nnh = 0.96
    Ac = 0.0779
    Ah = 0.05244
    tt4max = 635
    vo = M * a_s
    pt2 = nr * pto
    tt2 = tto
    tel = 0
    while not mfi == 0:
```

```
        #       start iteratieloop voor motorgegevens
        #       bij gegeven brandstofstroom mfi (Tt5=0)
        r = 0
        s = 0
        b = 0
        mhd = 10
        tt3d = 337.466
        c = 0
        while b < 50:
            c = c + 1
            d = 0
            while d < 30:
                d = d + 1
                ttb = nb * 41.865 * pow(10, 6) * mfi / 1147 / mhd
                fi = 0.3452334 * (1 + ttb / tt3d)
                ehpc = pow(1 + fi * (pow(6.5625, (0.4 / 1.4 / nc)) - 1), (nc * 1.4 / 0.4))
                mht3p3 = 0.0011217 * ehpc / sqrt(fi) / 6.5625
                pt3 = mhd * sqrt(tt3d) / mht3p3
                elpc = pt3 / pt2
                tt3 = tt2 * pow(elpc, (0.4 / 1.4 / nf))
                deltem = tt3d / tt3
                if deltem > 1.0005 or deltem < 0.9995:
                    tt3d = tt3d + (tt3 - tt3d) * 1.3
                    if tt3d < 100:
                        break  # inner loop

            if W_end0():
                break

        if W_end1():
            break

    while itt >= ittmax:
        mfi = 1450 * 0.4536 / 3600
        #       start iteratieloop voor motorgegevens bij overschrijding
        #       van ITTmax
        r = 0
        s = 0
        while r < 50:
            b = 0
            mhd = 10
            tt3d = 337.466
            c = 0
            while b < 50:
                c = c + 1
                d = 0
                while d < 30:
                    d = d + 1
                    ttb = nb * 41.865 * pow(10, 6) * mfi / 1147 / mhd
                    fi = 0.3452334 * (1 + ttb / tt3d)
                    ehpc = pow(1 + fi * (pow(6.5625, (0.4 / 1.4 / nc)) - 1), (nc * 1.4 / 0.4))
                    mht3p3 = 0.0011217 * ehpc / sqrt(fi) / 6.5625
                    pt3 = mhd * sqrt(tt3d) / mht3p3
                    elpc = pt3 / pt2
                    tt3 = tt2 * pow(elpc, (0.4 / 1.4 / nf))
                    deltem = tt3d / tt3
                    if deltem > 1.0005 and deltem < 0.9995:
                        tt3d = tt3d + (tt3 - tt3d) * 1.3
                        if tt3d < 100:
                            break
```

```
                if W_end4():
                    break

        if tel < 20:
            if 12231.4 * elpc - 13041.14 < 0:
                NLcort = 30
            else:
                NLcort = 23.198 + sqrt(12231.4 * elpc - 13041.14)
            dncn = (
                46.449
                - 2.681348 * NLcort
                + 0.067482533 * pow(NLcort, 2)
                - 0.00072636 * pow(NLcort, 3)
                + 0.0000027318 * pow(NLcort, 4)
            ) / 100
            dncn = -0.1209875 * (NLcort - 104) + 0.0005625 * (pow(NLcort, 2) - pow(104, 2))
            nf1 = (
                -14803
                - 0.2407085
                + 96754 * elpc
                + 0.939903 * elpc
                - 279446 * pow(elpc, 2)
                - 0.26169 * pow(elpc, 2)
                + 468125 * pow(elpc, 3)
                + 0.06834 * pow(elpc, 3)
                - 501269 * pow(elpc, 4)
                - 0.62651 * pow(elpc, 4)
                + 355822 * pow(elpc, 5)
                + 0.2019 * pow(elpc, 5)
                - 167440 * pow(elpc, 6)
                - 0.79086 * pow(elpc, 6)
                + 50370 * pow(elpc, 7)
                + 0.802172 * pow(elpc, 7)
                - 8790 * pow(elpc, 8)
                - 0.376607 * pow(elpc, 8)
                + 678 * pow(elpc, 9)
                + 0.0601726 * pow(elpc, 9)
            )
            nc = nf1 - 0.015
            if abs(elpc1 / elpc - 1) < 0.001:
                break
            nf = nf1
            elpc1 = elpc
            tel = tel + 1
        else:
            break

    return Tn


def W_end0():
    global ittmax, Ne, NL, tto, pto, po, t_o, T_isa, rho, nu, mu, a_s, elpcc, r, tel, mh, mc, wh, wc, tt5, mf,

    if pt3 / po < 1:
        if b > 15:
            return True
        b = b + 1
        mhd = mhd - 0.5
        c = 0
    else:
        p3krit = pow((1 - 0.4 / 2.4 / nnc), (-1.4 / 0.4))
```

```
        if pt3 / po < p3krit:
            wc = sqrt(2 * nnc * 1005 * tt3 * (1 - pow((po / pt3), (0.4 / 1.4))))
            pce = po
        else:
            pce = pt3 / p3krit
            t9 = tt3 * (1 - nnc * (1 - pow((pce / pt3), (0.4 / 1.4))))
            wc = sqrt(1.4 * 287.05 * t9)
        mc = Ac * pce * wc / 287.05 / t9
        tt4 = tt3 * pow(ehpc, (0.4 / 1.4 / nc))
        tt5 = tt4 + ttb
        tt6 = tt5 - 1005 / 1147 / nm * (tt4 - tt3)
        bpr = mc / mhd
        tt7 = tt6 - 1005 * (1 + bpr) * (tt3 - tt2) / 1147 / nm
        if tt7 < 10:
            if b > 20:
                return True
            mhd = mhd - 0.5
            b = b + 1
            c = 0
        else:
            pt5 = pto * nr * elpc * ehpc * dpt
            pt6 = pt5 * pow((tt6 / tt5), (1.33 / 0.33 / nt))
            pt7 = pt6 * pow((tt7 / tt6), (1.33 / 0.33 / nt))
            if pt7 / po < 1:
                if b > 35:
                    return True
                mhd = mhd - 0.25
                b = b + 1
                c = 0
            else:
                p7krit = pow((1 - 0.33 / 2.33 / nnh), (-1.33 / 0.33))
                if pt7 / po < p7krit:
                    wh = sqrt(2 * nnh * 1147 * tt7 * (1 - pow((po / pt7), (0.33 / 1.33))))
                    t8 = tt7 * (1 - nnh * (1 - pow((po / pt7), (0.33 / 1.33))))
                    phe = po
                else:
                    phe = pt7 / p7krit
                    t8 = tt7 * (1 - nnh * (1 - pow((phe / pt7), (0.33 / 1.33))))
                    if t8 < 20:
                        return True
                    wh = sqrt(1.33 * 287.05 * t8)
                mh = Ah * wh * phe / 287.05 / t8
                delmh = mh / mhd
                if c >= 40:
                    return True
                if delmh < 0.999:
                    mhd = mhd + 0.1 * (mh - mhd)
                elif delmh > 1.001:
                    mhd = mhd + 0.05 * (mh - mhd)
                else:
                    Tn = mc * (wc - vo) + mh * (wh - vo) + Ac * (pce - po) + Ah * (phe - po)
                    # echo "Tn (1): ", Tn, "<br>"
                    mf = ttb / nb / 41.875 * pow(10, 6) * 1147 * mh
                    itt = tt7 + 3 * (tt3 - tt2) - 273.15
                    return True
    return False


def W_end1():
    global ittmax, Ne, NL, tto, pto, po, t_o, T_isa, rho, nu, mu, a_s, elpcc, r, tel, mh, mc, wh, wc, tt5, mf,

```

```
    if c >= 40:
        return True
    if tel < 20:
        if 12231.4 * elpc - 13041.14 < 0:
            NLcort = 30
        else:
            NLcort = 23.198 + sqrt(12231.4 * elpc - 13041.14)
        dncn = (
            46.449
            - 2.681348 * NLcort
            + 0.067482533 * pow(NLcort, 2)
            - 0.00072636 * pow(NLcort, 3)
            + 0.0000027318 * pow(NLcort, 4)
        ) / 100
        dncn = -0.1209875 * (NLcort - 104) + 0.0005625 * (pow(NLcort, 2) - pow(104, 2))
        nf1 = (
            -14803
            - 0.2407085
            + 96754 * elpc
            + 0.939903 * elpc
            - 279446 * pow(elpc, 2)
            - 0.26169 * pow(elpc, 2)
            + 468125 * pow(elpc, 3)
            + 0.06834 * pow(elpc, 3)
            - 501269 * pow(elpc, 4)
            - 0.62651 * pow(elpc, 4)
            + 355822 * pow(elpc, 5)
            + 0.2019 * pow(elpc, 5)
            - 167440 * pow(elpc, 6)
            - 0.79086 * pow(elpc, 6)
            + 50370 * pow(elpc, 7)
            + 0.802172 * pow(elpc, 7)
            - 8790 * pow(elpc, 8)
            - 0.376607 * pow(elpc, 8)
            + 678 * pow(elpc, 9)
            + 0.0601726 * pow(elpc, 9)
        )
        nc = nf1 - 0.015
        elpcc = elpc1 / elpc - 1
        if elpcc < 0:
            elpcc = -elpcc
        if elpcc < 0.001:
            return True
        nf = nf1
        elpc1 = elpc
        tel = tel + 1
    else:
        return True
    return False


def W_end4():
    global ittmax, Ne, NL, tto, pto, po, t_o, T_isa, rho, nu, mu, a_s, elpcc, r, tel, mh, mc, wh, wc, tt5, mf,

    if pt3 / po < 1:
        if b > 15:
            return True
        b = b + 1
        mhd = mhd - 0.5
        c = 0
    else:
```

```
        p3krit = pow((1 - 0.4 / 2.4 / nnc), (-1.4 / 0.4))
        if pt3 / po < p3krit:
            wc = sqrt(2 * nnc * 1005 * tt3 * (1 - pow((po / pt3), (0.4 / 1.4))))
            t9 = tt3 * (1 - nnc * (1 - pow((po / pt3), (0.4 / 1.4))))
            pce = po
        else:
            pce = pt3 / p3krit
            t9 = tt3 * (1 - nnc * (1 - pow((pce / pt3), (0.4 / 1.4))))
            wc = sqrt(1.4 * 287.05 * t9)
        mc = Ac * pce * wc / 287.05 / t9
        tt4 = tt3 * pow(ehpc, (0.4 / 1.4 / nc))
        tt5 = tt4 + ttb
        tt6 = tt5 - 1005 / 1147 / nm * (tt4 - tt3)
        bpr = mc / mhd
        tt7 = tt6 - 1005 * (1 + bpr) * (tt3 - tt2) / 1147 / nm
        if tt7 < 10:
            if b > 20:
                return True
            mhd = mhd - 0.5
            b = b + 1
            c = 0
        else:
            pt5 = pto * nr * elpc * ehpc * dpt
            pt6 = pt5 * pow((tt6 / tt5), (1.33 / 0.33 / nt))
            pt7 = pt6 * pow((tt7 / tt6), (1.33 / 0.33 / nt))
            if pt7 / po < 1:
                if b > 35:
                    return True
                mhd = mhd - 0.25
                b = b + 1
                c = 0
            else:
                p7krit = pow((1 - 0.33 / 2.33 / nnh), (-1.33 / 0.33))
                if pt7 / po < p7krit:
                    wh = sqrt(2 * nnh * 1147 * tt7 * (1 - pow((po / pt7), (0.33 / 1.33))))
                    t8 = tt7 * (1 - nnh * (1 - pow((po / pt7), (0.33 / 1.33))))
                    phe = po
                else:
                    phe = pt7 / p7krit
                    t8 = tt7 * (1 - nnh * (1 - pow((phe / pt7), (0.33 / 1.33))))
                    if t8 < 20:
                        return True
                    wh = sqrt(1.33 * 287.05 * t8)
                mh = Ah * wh * phe / 287.05 / t8
                delmh = mh / mhd
                if c < 100:
                    if delmh < 0.999:
                        mhd = mhd + 0.1 * (mh - mhd)
                    elif delmh > 1.001:
                        mhd = mhd + 0.05 * (mh - mhd)
                else:
                    Tn = mc * (wc - vo) + mh * (wh - vo) + Ac * (pce - po) + Ah * (phe - po)
                    # echo "Tn (2): ", Tn, "<br>"
                    mf = ttb / nb / 41.875 * pow(10, 6) * 1147 * mh
                    itt = tt7 + 3 * (tt3 - tt2) - 273.15
                    print(itt, ittmax)
                    itrel = itt / ittmax
                    if itrel - 1 < 0:
                        itrel = -(itrel - 1)
                    if itrel > 0.0001:
                        if itrel > 1.5:
```

```
                        itrel = 1.5
                    if itrel < 0.5:
                        itrel = 0.5
                    mfi = mfi - (itrel - 1) * mfi
                    r = r + 1
                    if r > 50:
                        return True
                    else:
                        b = 0
                        mhd = 10
                        tt3d = 337.466
                        c = 0
                else:
                    return True
    return False


def calculate_thrust(h: float, M: float, T_static: float, fuelflow: float) -> float:
    """
    Calculates thrust based on operating conditions.

    Notes:
        - Must be used for each engine individually, using the sum
        of left and right fuel flows as input gives the wrong results.
        - T_static is the static temperature. The relation with dT (used in the
        Excel sheet) is T_static = T_isa + dT (T_isa = 288.15 - 0.0065 * h).

    Args:
        h: Altitude [m]
        M: Mach number [-]
        T_static: Static temperature [K]
        fuelflow: Fuel mass flow [kg/s]

    Returns:
        Thrust [N]
    """
    global mfi
    T_isa = atmos(h, M, T_static)
    delta_T = T_static - T_isa
    mfi = fuelflow
    try:
        return stuw(h, M, delta_T)
    except (ValueError, OverflowError):
        # Some values cause math domain or overflow errors
        return None


def calculate_thrust_from_row(
    row: pd.Series, fuel_flow: Optional[float] = None
) -> tuple[float, float]:
    if fuel_flow is None:
        fuel_flow_left = row["fuel_flow_left"]
        fuel_flow_right = row["fuel_flow_right"]
    else:
        fuel_flow_left = fuel_flow
        fuel_flow_right = fuel_flow
    return (
        calculate_thrust(row["h"], row["M"], row["T_static"], fuel_flow_left),
        calculate_thrust(row["h"], row["M"], row["T_static"], fuel_flow_right),
    )
```

```python

def calculate_thrust_from_df(df: pd.DataFrame, fuel_flow: Optional[float] = None) -> pd.DataFrame:
    return df.apply(calculate_thrust_from_row, args=(fuel_flow,), axis=1, result_type="expand")


def calculate_thrust_exe(
    h: float, M: float, T_static: float, fuel_flow_left: float, fuel_flow_right: float
):
    thrust_exe = Path(".") / "bin/thrust.exe"
    cwd = Path(tempfile.gettempdir())
    input_file = cwd / "matlab.dat"
    output_file = cwd / "thrust.dat"

    with input_file.open("w") as f:
        T_isa = 288.15 - 0.0065 * h
        dT = T_static - T_isa
        f.write(f"{h:f} {M:f} {dT:f} {fuel_flow_left:f} {fuel_flow_right:f}")

    try:
        subprocess.run(thrust_exe.absolute(), cwd=cwd, stdout=subprocess.DEVNULL, timeout=5)
    except subprocess.TimeoutExpired:
        return None

    with output_file.open("r") as f:
        thrusts = f.readline().split()

    # Delete temporary files
    input_file.unlink()
    output_file.unlink()

    return float(thrusts[0]), float(thrusts[1])


def calculate_thrust_from_row_exe(
    row: pd.Series, fuel_flow: Optional[float] = None
) -> tuple[float, float]:
    if fuel_flow is None:
        fuel_flow_left = row["fuel_flow_left"]
        fuel_flow_right = row["fuel_flow_right"]
    else:
        fuel_flow_left = fuel_flow
        fuel_flow_right = fuel_flow
    return (
        calculate_thrust_exe(row["h"], row["M"], row["T_static"], fuel_flow_left),
        calculate_thrust_exe(row["h"], row["M"], row["T_static"], fuel_flow_right),
    )


def calculate_thrust_from_df_exe(
    df: pd.DataFrame, fuel_flow: Optional[float] = None
) -> pd.DataFrame:
    thrust_exe = Path(".") / "bin/thrust.exe"
    cwd = Path(tempfile.gettempdir())
    input_file = cwd / "matlab.dat"
    output_file = cwd / "thrust.dat"

    with input_file.open("w") as f:
        for row in df.itertuples():
            T_isa = 288.15 - 0.0065 * row.h
            dT = row.T_static - T_isa

```

```
            if fuel_flow is None:
                fuel_flow_left = row.fuel_flow_left
                fuel_flow_right = row.fuel_flow_right
            else:
                fuel_flow_left = fuel_flow
                fuel_flow_right = fuel_flow

            f.write(f"{row.h:f} {row.M:f} {dT:f} {fuel_flow_left:f}  {fuel_flow_right:f}\n")

    try:
        subprocess.run(
            thrust_exe.absolute(), cwd=cwd, stdout=subprocess.DEVNULL, timeout=5 * len(df.index)
        )
    except subprocess.TimeoutExpired:
        return None

    with output_file.open("r") as f:
        thrusts = [[float(t) for t in line.split()] for line in f.readlines()]

    # Delete temporary files
    input_file.unlink()
    output_file.unlink()

    return pd.DataFrame(thrusts, index=df.index)


def calc_Tc(T: float, V: float, rho: float, S: float = constants.S) -> float:
    """
    Calculate Tc for a given combination of T, rho, V and S.

    Args:
        T (array_like): Thrust [N]
        rho (float): Air density [kg/m3]
        V (array_like): True airspeed [m/s]
        S (float): Surface area [m2]

    Returns:
        (array_like): Tc [-]
    """

    return T / (calc_dynamic_pressure(V, rho) * S)
```

fd/analysis/characteristic_motion_parameters.py

```
import numpy as np
from math import *
from fd.simulation.constants import *
from fd.simulation.aircraft_model import AircraftModel
from fd.structs import AerodynamicParameters


def time_constant_aperiodic_roll(eig, Ve):
    """ "
    Calculating time constant for the aperiodic roll
    Args:
        eig: eigenvalue for aperiodic roll
        Ve: equivalent velocity during aperiodic roll

    """
    tau = -(1 / eig) * (c / Ve)

    return tau
```

```python


def time_constant_spiral(eig, Ve):
    """ "
    Calculating time constant for the spiral
    Args:
        eig: eigenvalue spiral
        Ve: equivalent velocity during spiral

    """
    tau = -(1 / eig) * (c / Ve)
    return tau


def characteristics_dutch_roll(imag_eig, real_eig, Ve):
    """
    Calculating the period and time to damp to half amplitude for the Dutch roll
    Args:
        imag_eig: imaginairy part of eigenvalue for dutch roll
        real_eig: real part of eigenvalue of dutch roll
        Ve: equivalent velocity during dutch roll

    Returns:

    """
    P = ((2 * pi) / (imag_eig)) * (b / Ve)
    T_half = (np.log(0.5) / (real_eig)) * (b / Ve)
    return P, T_half


def characteristics_phugoid(imag_eig, real_eig, Ve):
    """
    Calculating the period and time to damp to half amplitude for the Dutch roll
    Args:
        imag_eig: imaginairy part of eigenvalue for phugoid
        real_eig: real part of eigenvalue of phugoid
        Ve: equivalent velocity during phugoid

    Returns:

    """
    P = ((2 * pi) / (imag_eig)) * (b / Ve)
    T_half = (np.log(0.5) / (real_eig)) * (b / Ve)
    return P, T_half


def characteristics_short_period(imag_eig, real_eig, Ve):
    """
    Calculating the period and time to damp to half amplitude for the short period
    Args:
        imag_eig: imaginairy part of eigenvalue for the short period
        real_eig: real part of eigenvalue of the short period
        Ve: equivalent velocity during the short period

    Returns:

    """
    P = ((2 * pi) / (imag_eig)) * (b / Ve)
    T_half = (np.log(0.5) / (real_eig)) * (b / Ve)
    return P, T_half
```

fd/analysis/data_sheet.py

```python
from typing import Any

import numpy as np
import pandas as pd

from fd.analysis.aerodynamics import (
    calc_true_V,
    calc_CL,
    calc_CD,
)
from fd.analysis.center_of_gravity import calc_cg_position
from fd.analysis.reduced_values import (
    calc_reduced_equivalent_V,
    calc_reduced_elevator_deflection,
    calc_reduced_stick_force,
)
from fd.analysis.thrust import calculate_thrust_from_df, calc_Tc
from fd.analysis.util import add_common_derived_timeseries
from fd.conversion import (
    lbs_to_kg,
    timestamp_to_s,
    ft_to_m,
    kts_to_ms,
    lbshr_to_kgs,
    C_to_K,
    deg_to_rad,
)
from fd.io import load_data_sheet
from fd.simulation.constants import mass_basic_empty, fuel_flow_standard
from fd.util import mean_not_none, mean_not_nan_df

COLUMNS = {
    "hp": "h",
    "IAS": "cas",
    "a": "alpha",
    "de": "delta_e",
    "detr": "delta_t_e",
    "Fe": "F_e",
    "FFl": "fuel_flow_left",
    "FFr": "fuel_flow_right",
    "F. used": "fuel_used",
    "TAT": "T_total",
}


class DataSheet:
    def __init__(self, data_path: str):
        self._extract(load_data_sheet(data_path))

    def _extract(self, ws: list[list[Any]]):
        """
        Extract parameters from PFDS into variables.

        Args:
            ws: worksheet from Excel file as list of lists
        """
        self._extract_mass(ws)

        self.df_clcd = DataSheet._extract_data_sheet_series(ws, 27, 33)
        self.df_elevator_trim = DataSheet._extract_data_sheet_series(ws, 58, 64)
```

```
        self.df_cg_shift = DataSheet._extract_data_sheet_series(ws, 74, 75)

        self.timestamp_phugoid = timestamp_to_s(ws[82][3])
        self.timestamp_short_period = timestamp_to_s(ws[83][3])
        self.timestamp_dutch_roll = timestamp_to_s(ws[82][6])
        self.timestamp_dutch_roll_yd = timestamp_to_s(ws[83][6])
        self.timestamp_aperiodic_roll = timestamp_to_s(ws[82][9])
        self.timestamp_spiral = timestamp_to_s(ws[83][9])

    def _extract_mass(self, ws: list[list[Any]]):
        self.mass_pilot_1 = ws[7][7]
        self.mass_pilot_2 = ws[8][7]
        self.mass_coordinator = ws[9][7]
        self.mass_observer_1l = ws[10][7]
        self.mass_observer_1r = ws[11][7]
        self.mass_observer_2l = ws[12][7]
        self.mass_observer_2r = ws[13][7]
        self.mass_observer_3l = ws[14][7]
        self.mass_observer_3r = ws[15][7]
        self.mass_block_fuel = lbs_to_kg(ws[17][3])

        self.mass_initial = (
            mass_basic_empty
            + self.mass_block_fuel
            + self.mass_pilot_1
            + self.mass_pilot_2
            + self.mass_coordinator
            + self.mass_observer_1l
            + self.mass_observer_1r
            + self.mass_observer_2l
            + self.mass_observer_2r
            + self.mass_observer_3l
            + self.mass_observer_3r
        )

    @staticmethod
    def _extract_data_sheet_series(
        ws: list[list[Any]], row_start: int, row_end: int
    ) -> pd.DataFrame:
        # Extract column names
        if ws[row_start - 1][1] is None:
            column_names = ws[row_start - 3]
        else:
            # Empty row between header and data missing for cg shift data
            column_names = ws[row_start - 2]

        # Build DataFrame from rows
        rows = ws[row_start : row_end + 1]
        df = pd.DataFrame(rows, columns=column_names).drop(
            columns=["nr.", "ET*", None], errors="ignore"
        )
        df = df.dropna(subset="time").reset_index(drop=True)
        df["time"] = df["time"].apply(timestamp_to_s)
        # Force as float since Excel sheet may store numbers as strings
        df = df.astype("float64")
        df = DataSheet._process_data_sheet_series(df)

        return df

    @staticmethod
    def _process_data_sheet_series(df: pd.DataFrame) -> pd.DataFrame:
```

```
        df = df.rename(columns=COLUMNS)

        df["h"] = ft_to_m(df["h"])
        df["cas"] = kts_to_ms(df["cas"])
        df["alpha"] = deg_to_rad(df["alpha"])
        df["fuel_flow_left"] = lbshr_to_kgs(df["fuel_flow_left"])
        df["fuel_flow_right"] = lbshr_to_kgs(df["fuel_flow_right"])
        df["fuel_used"] = lbs_to_kg(df["fuel_used"])
        df["T_total"] = C_to_K(df["T_total"])

        if "delta_e" in df.columns:
            df["delta_e"] = deg_to_rad(df["delta_e"])
        if "delta_t_e" in df.columns:
            df["delta_t_e"] = deg_to_rad(df["delta_t_e"])

        return df


class AveragedDataSheet:
    def __init__(self, data_sheets: dict[str, DataSheet]):
        assert len(data_sheets) > 0, "No data sheets found, check your working directory"

        self.data_sheet_names = list(data_sheets.keys())
        self.data_sheets = list(data_sheets.values())
        self._calculate_averages()
        self._add_derived_timeseries()

    def _calculate_averages(self):
        self.df_clcd = self._calculate_dataframe_average_and_check_deviations(
            [ds.df_clcd for ds in self.data_sheets]
        )
        self.df_elevator_trim = self._calculate_dataframe_average_and_check_deviations(
            [ds.df_elevator_trim for ds in self.data_sheets]
        )
        self.df_cg_shift = self._calculate_dataframe_average_and_check_deviations(
            [ds.df_cg_shift for ds in self.data_sheets]
        )

        self.timestamp_phugoid = mean_not_none([ds.timestamp_phugoid for ds in self.data_sheets])
        self.timestamp_short_period = mean_not_none(
            [ds.timestamp_short_period for ds in self.data_sheets]
        )
        self.timestamp_dutch_roll = mean_not_none(
            [ds.timestamp_dutch_roll for ds in self.data_sheets]
        )
        self.timestamp_dutch_roll_yd = mean_not_none(
            [ds.timestamp_dutch_roll_yd for ds in self.data_sheets]
        )
        self.timestamp_aperiodic_roll = mean_not_none(
            [ds.timestamp_aperiodic_roll for ds in self.data_sheets]
        )
        self.timestamp_spiral = mean_not_none([ds.timestamp_spiral for ds in self.data_sheets])

        self.mass_pilot_1 = mean_not_none([ds.mass_pilot_1 for ds in self.data_sheets])
        self.mass_pilot_2 = mean_not_none([ds.mass_pilot_2 for ds in self.data_sheets])
        self.mass_coordinator = mean_not_none([ds.mass_coordinator for ds in self.data_sheets])
        self.mass_observer_1l = mean_not_none([ds.mass_observer_1l for ds in self.data_sheets])
        self.mass_observer_1r = mean_not_none([ds.mass_observer_1r for ds in self.data_sheets])
        self.mass_observer_2l = mean_not_none([ds.mass_observer_2l for ds in self.data_sheets])
        self.mass_observer_2r = mean_not_none([ds.mass_observer_2r for ds in self.data_sheets])
        self.mass_observer_3l = mean_not_none([ds.mass_observer_3l for ds in self.data_sheets])
```

```python
        self.mass_observer_3r = mean_not_none([ds.mass_observer_3r for ds in self.data_sheets])
        self.mass_block_fuel = mean_not_none([ds.mass_block_fuel for ds in self.data_sheets])
        self.mass_initial = mean_not_none([ds.mass_initial for ds in self.data_sheets])

    def _calculate_dataframe_average_and_check_deviations(
        self, dfs: list[pd.DataFrame], threshold_pct=5, check_deviations=False
    ) -> pd.DataFrame:
        """
        Average data from multiple data sheets and check if any values deviate more than threshold_pct % from p
        """
        # Calculate mean of non-NA values
        df_mean = mean_not_nan_df(dfs)

        # Check deviations
        if check_deviations:
            for df_idx, df in enumerate(dfs):
                # Calculate mean absolute percentage error
                error: pd.DataFrame = ((df - df_mean) / df_mean).abs() * 100
                exceeds_error_threshold = np.argwhere(error.to_numpy() > threshold_pct)
                if len(exceeds_error_threshold) > 0:
                    data_sheet_name = self.data_sheet_names[df_idx]
                    print(
                        f"Some values in {data_sheet_name} exceed the {threshold_pct} % threshold"
                    )
                    for i in range(exceeds_error_threshold.shape[0]):
                        row, col = exceeds_error_threshold[i]
                        column_name = df.columns[col]
                        print(
                            f"Row {row}, column {column_name} ({error.iloc[row, col]:.3} %): "
                            f"{data_sheet_name} = {df.iloc[row, col]:.3}, mean = {df_mean.iloc[row, col]:.3}"
                        )
                    print()

        return df_mean

    def _add_derived_timeseries(self):
        for df in [self.df_clcd, self.df_elevator_trim, self.df_cg_shift]:
            df["time_min"] = df["time"] / 60
            df["m"] = self.mass_initial - df["fuel_used"]
            df["m_fuel"] = self.mass_block_fuel - df["fuel_used"]
            df = add_common_derived_timeseries(df)

            df["tas"] = df.apply(lambda row: calc_true_V(row["T_static"], row["M"]), axis=1)
            df["cas_reduced"] = df.apply(
                lambda row: calc_reduced_equivalent_V(row["cas"], row["W"]), axis=1
            )
            # No need to calculate equivalent airspeed, is already given in data as IAS

            # Calculate thrust (actual + standardized)
            df[["T_left", "T_right"]] = calculate_thrust_from_df(df)
            # df[["T_left", "T_right"]] = calculate_thrust_from_df_exe(df)
            df["T"] = df["T_left"] + df["T_right"]

            df[["T_s_left", "T_s_right"]] = calculate_thrust_from_df(
                df, fuel_flow=fuel_flow_standard
            )
            # df[["T_s_left", "T_s_right"]] = calculate_thrust_from_df_exe(df, fuel_flow=fuel_flow_standard)
            df["T_s"] = df["T_s_left"] + df["T_s_right"]

            # Calculate coefficients
            # Using CAS + rho0 gives the same results as TAS + rho
```

```python
            df["C_L"] = df.apply(lambda row: calc_CL(row["W"], row["tas"], row["rho"]), axis=1)
            df["C_D"] = df.apply(lambda row: calc_CD(row["T"], row["tas"], row["rho"]), axis=1)
            df["T_c"] = df.apply(lambda row: calc_Tc(row["T"], row["tas"], row["rho"]), axis=1)
            df["T_c_s"] = df.apply(lambda row: calc_Tc(row["T_s"], row["tas"], row["rho"]), axis=1)

        for df in [self.df_elevator_trim, self.df_cg_shift]:
            df["F_e_reduced"] = df.apply(
                lambda row: calc_reduced_stick_force(row["F_e"], row["W"]), axis=1
            )
            # Reduced elevator deflection cannot be calculated here because C_m_delta is not known yet

        self.df_cg_shift["shift"] = [False, True]
        df["x_cg"] = self.df_cg_shift.apply(
            lambda row: calc_cg_position(
                row["m_fuel"],
                self.mass_pilot_1,
                self.mass_pilot_2,
                self.mass_coordinator,
                self.mass_observer_1l,
                self.mass_observer_1r,
                self.mass_observer_2l,
                self.mass_observer_2r,
                self.mass_observer_3l,
                self.mass_observer_3r,
                row["shift"],
            ),
            axis=1,
        )

    def add_reduced_elevator_deflection_timeseries(self, C_m_delta: float):
        for df in [self.df_elevator_trim, self.df_cg_shift]:
            df["delta_e_reduced"] = df.apply(
                lambda row: calc_reduced_elevator_deflection(
                    row["delta_e"], C_m_delta, row["T_c_s"], row["T_c"]
                ),
                axis=1,
            )
```

fd/analysis/thermodynamics.py

```python
import numpy as np

from fd.simulation import constants


def calc_static_pressure(hp):
    """
    Calculate the static pressure from the pressure height
    Args:
        hp (float): Pressure height[m]

    Returns (float): The static pressure for the pressure height given[Pa]

    """
    return constants.p0 * (1 + constants.Tempgrad * hp / constants.Temp0) ** (
        -constants.g / (constants.Tempgrad * constants.R)
    )


def calc_mach(hp, Vc):
    """

```

```
    Args:
        hp (float): Pressure height[m]
        Vc (float): The calibrated speed[m/s]

    Returns (float): Mach number for the conditions given[-]

    """
    return np.sqrt(
        2
        / (constants.gamma - 1)
        * (
            (
                1
                + constants.p0
                / calc_static_pressure(hp)
                * (
                    (
                        1
                        + (constants.gamma - 1)
                        / (2 * constants.gamma)
                        * constants.rho0
                        / constants.p0
                        * Vc**2
                    )
                    ** (constants.gamma / (constants.gamma - 1))
                    - 1
                )
            )
            ** ((constants.gamma - 1) / constants.gamma)
            - 1
        )
    )


def calc_static_temperature(Ttot, M):
    """

    Args:
        Ttot (float): Total temperature[K]
        M (float): Mach number[-]

    Returns (float): Static temperature[K]

    """
    return Ttot / (1 + (constants.gamma - 1) / 2 * M**2)


def calc_density(p, T):
    """

    Args:
        p (float): Static pressure[Pa]
        T (float): Static temperature[K]

    Returns (float): Density[kg/m^3]

    """
    return p / (constants.R * T)
```

fd/validation/eigenmotion_characteristics.py

```python
import numpy as np

from fd.simulation.constants import *


def time_constant_aperiodic_roll(eig: complex, V0):
    """
    Calculating time constant for the aperiodic roll
    Args:
        eig: eigenvalue for the aperiodic roll

    Returns:
        time constant
    """
    if abs(eig.imag) > 0:
        print(f"WARNING: aperiodic roll eigenvalue should be real, is {eig}")
    tau = -(1 / eig.real)
    return tau


def time_constant_spiral(eig: complex):
    """
    Calculating time constant for the spiral
    Args:
        eig: eigenvalue for the spiral

    Returns:
        Time constant
    """
    if abs(eig.imag) > 0:
        print(f"WARNING: spiral eigenvalue should be real, is {eig}")
    tau = -(1 / eig.real)
    return tau


def characteristics_dutch_roll(eig: complex):
    """
    Calculating the period and time to damp to half amplitude for the Dutch roll
    Args:
        eig: eigenvalue for the Dutch roll

    Returns:
        Period, time to half amplitude
    """
    P = (2 * pi) / abs(eig.imag)
    T_half = np.log(0.5) / eig.real
    return P, T_half


def characteristics_phugoid(eig: complex):
    """
    Calculating the period and time to damp to half amplitude for the Dutch roll
    Args:
        eig: eigenvalue for the phugoid

    Returns:
        Period, time to half amplitude
    """
    P = (2 * pi) / abs(eig.imag)
    T_half = np.log(0.5) / eig.real
    return P, T_half
```

```python


def characteristics_short_period(eig: complex):
    """
    Calculating the period and time to damp to half amplitude for the short period
    Args:
        eig: eigenvalue for the short period

    Returns:
        Period, time to half amplitude
    """
    P = (2 * pi) / abs(eig.imag)
    T_half = np.log(0.5) / eig.real
    return P, T_half
```

fd/validation/comparison_eigenvalues.py

```python
import numpy as np

from fd.analysis.aerodynamics import calc_CL
from fd.analysis.flight_test import FlightTest
from fd.simulation.aircraft_model import AircraftModel
from fd.validation.eigenmotion_characteristics import (
    time_constant_aperiodic_roll,
    characteristics_dutch_roll,
    time_constant_spiral,
    characteristics_phugoid,
    characteristics_short_period,
)


class EigenvalueComparison:
    def __init__(self, flight_test: FlightTest, model: AircraftModel):
        self.flight_test = flight_test
        self.model = model

    def compare(self):
        self._compare_phugoid()
        self._compare_short_period()
        self._compare_dutch_roll()
        self._compare_aperiodic_roll()
        self._compare_spiral()

    def _compare_phugoid(self):
        data = self.flight_test.df_phugoid
        A, _, _, _ = self.model.get_state_space_matrices_symmetric_from_df(data)
        eigs, _ = self.model.get_eigenvalues_and_eigenvectors(A)
        eig_phugoid = eigs[2]
        P, T_half = characteristics_phugoid(eig_phugoid)
        print(f"Phugoid (simulated): P = {P:.3f} s, T_half = {T_half:.3f} s, {eig_phugoid}")

        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        theta0 = data["theta"].iloc[0]
        eig_phugoid = self.model.get_idealized_phugoid_eigenvalues(m, rho0, theta0, V0)[0]
        P, T_half = characteristics_phugoid(eig_phugoid)
        print(f"Phugoid (idealized): P = {P:.3f} s, T_half = {T_half:.3f} s")

    def _compare_short_period(self):
        data = self.flight_test.df_short_period
        A, _, _, _ = self.model.get_state_space_matrices_symmetric_from_df(data)
```

```python
        eigs, _ = self.model.get_eigenvalues_and_eigenvectors(A)
        eig_short_period = eigs[0]
        P, T_half = characteristics_short_period(eig_short_period)
        print(
            f"Short period (simulated): P = {P:.3f} s, T_half = {T_half:.3f} s, {eig_short_period}"
        )

        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        eig_short_period = self.model.get_idealized_shortperiod_eigenvalues(m, rho0, V0)[0]
        P, T_half = characteristics_short_period(eig_short_period)
        print(f"Short period (idealized): P = {P:.3f} s, T_half = {T_half:.3f} s")

    def _compare_dutch_roll(self):
        data = self.flight_test.df_dutch_roll
        A, _, _, _ = self.model.get_state_space_matrices_asymmetric_from_df(data)
        eigs, _ = self.model.get_eigenvalues_and_eigenvectors(A)
        eig_dutch_roll = eigs[1]
        P, T_half = characteristics_dutch_roll(eig_dutch_roll)
        print(f"Dutch roll (simulated): P = {P:.3f} s, T_half = {T_half:.3f} s, {eig_dutch_roll}")

        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        eig_dutch_roll = self.model.get_idealized_dutchroll_eigenvalues(m, rho0, V0)[0]
        P, T_half = characteristics_dutch_roll(eig_dutch_roll)
        print(f"Dutch roll (idealized): P = {P:.3f} s, T_half = {T_half:.3f} s")

    def _compare_aperiodic_roll(self):
        data = self.flight_test.df_aperiodic_roll
        A, _, _, _ = self.model.get_state_space_matrices_asymmetric_from_df(data)
        eigs, _ = self.model.get_eigenvalues_and_eigenvectors(A)
        eig_aperiodic_roll = eigs[0]
        V0 = data["tas"].iloc[0]
        tau = time_constant_aperiodic_roll(eig_aperiodic_roll, V0)
        print(f"Aperiodic roll (simulated): tau = {tau:.3f}, {eig_aperiodic_roll}")

        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        rho0 = data["rho"].iloc[0]
        eig_aperiodic_roll = self.model.get_idealized_aperiodicroll_eigenvalues(m, rho0, V0)
        tau = time_constant_aperiodic_roll(eig_aperiodic_roll, V0)
        print(f"Aperiodic roll (idealized): tau = {tau:.3f}")

    def _compare_spiral(self):
        data = self.flight_test.df_spiral
        A, _, _, _ = self.model.get_state_space_matrices_asymmetric_from_df(data)
        eigs, _ = self.model.get_eigenvalues_and_eigenvectors(A)
        eig_spiral = eigs[3]
        tau = time_constant_spiral(eig_spiral)
        print(f"Spiral (simulated): tau = {tau:.3f}, {eig_spiral}")

        m = (data["m"].iloc[0] + data["m"].iloc[-1]) / 2
        V0 = data["tas"].iloc[0]
        rho0 = data["rho"].iloc[0]
        theta0 = data["theta"].iloc[0]
        CL = calc_CL(data["W"].iloc[0] * np.cos(theta0), V0, rho0)
        eig_aperiodic_roll = self.model.get_idealized_spiral_eigenvalues(m, rho0, V0, CL)
        tau = time_constant_spiral(eig_aperiodic_roll)
        print(f"Spiral (idealized): tau = {tau:.3f}")
```

fd/validation/comparison.py

```python
import pandas as pd
from matplotlib import pyplot as plt
from math import pi

from fd.analysis.flight_test import FlightTest
from fd.plotting import format_plot
from fd.simulation.simulation import Simulation
from fd.plotting import save_plot


class SimulatedMeasuredComparison:
    simulated_dutch_roll: pd.DataFrame

    def __init__(self, flight_test: FlightTest, simulation: Simulation):
        self.flight_test = flight_test
        self.simulation = simulation

    def run_simulations(self):
        self.simulated_dutch_roll = self.simulation.simulate_asymmetric(
            self.flight_test.df_dutch_roll, flip_input=True
        )
        self.simulated_phugoid = self.simulation.simulate_symmetric(self.flight_test.df_phugoid)
        self.simulated_aperiodic_roll = self.simulation.simulate_asymmetric(
            self.flight_test.df_aperiodic_roll
        )
        self.simulated_dutch_roll_yd = self.simulation.simulate_asymmetric(
            self.flight_test.df_dutch_roll_yd, flip_input=True
        )
        self.simulated_spiral = self.simulation.simulate_asymmetric(self.flight_test.df_spiral)
        self.simulated_short_period = self.simulation.simulate_symmetric(
            self.flight_test.df_short_period
        )

    def plot_responses(self):
        self.plot_phugoid_full()
        self.plot_phugoid()
        self.plot_short_period_full()
        self.plot_short_period()
        self.plot_spiral_full()
        self.plot_spiral()
        self.plot_dutch_roll_full()
        self.plot_dutch_roll()
        self.plot_dutch_roll_yd_full()
        self.plot_dutch_roll_yd()
        self.plot_aperiodic_roll_full()
        self.plot_aperiodic_roll()
        print("Done")

    def plot_dutch_roll(self):
        fig, (ax_p, ax_r) = plt.subplots(2, 1, figsize=(12, 6))

        ax_p.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_dutch_roll.index,
            self.flight_test.df_dutch_roll["p"] * 180 / pi,
            label="Measured",
```

```
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")
        ax_p.legend()

        ax_r.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_dutch_roll.index,
            self.flight_test.df_dutch_roll["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "dutch_roll")
        plt.show()

    def plot_dutch_roll_full(self):
        fig, (ax_b, ax_phi, ax_p, ax_r) = plt.subplots(4, 1, figsize=(12, 12))

        ax_b.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["beta"] * 180 / pi,
            label="Simulated",
        )
        ax_b.set_ylabel("Sideslip angle $beta$ [°]")

        ax_phi.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["phi"] * 180 / pi,
            label="Simulated",
        )
        ax_phi.plot(
            self.flight_test.df_dutch_roll.index,
            self.flight_test.df_dutch_roll["phi"] * 180 / pi,
            label="Measured",
        )
        ax_phi.set_ylabel("Roll angle $phi$ [°]")
        ax_phi.legend()

        ax_p.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_dutch_roll.index,
            self.flight_test.df_dutch_roll["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")

        ax_r.plot(
            self.simulated_dutch_roll.index,
            self.simulated_dutch_roll["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_dutch_roll.index,
            self.flight_test.df_dutch_roll["r"] * 180 / pi,
```

```
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "dutch_roll_full")
        plt.show()

    def plot_dutch_roll_yd(self):
        fig, (ax_p, ax_r) = plt.subplots(2, 1, figsize=(12, 6))

        ax_p.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_dutch_roll_yd.index,
            self.flight_test.df_dutch_roll_yd["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")
        ax_p.legend()

        ax_r.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_dutch_roll_yd.index,
            self.flight_test.df_dutch_roll_yd["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "dutch_roll_yd")
        plt.show()

    def plot_dutch_roll_yd_full(self):
        fig, (ax_b, ax_phi, ax_p, ax_r) = plt.subplots(4, 1, figsize=(12, 12))

        ax_b.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["beta"] * 180 / pi,
            label="Simulated",
        )
        ax_b.set_ylabel("Sideslip angle $beta$ [°]")

        ax_phi.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["phi"] * 180 / pi,
            label="Simulated",
        )
        ax_phi.plot(
            self.flight_test.df_dutch_roll_yd.index,
            self.flight_test.df_dutch_roll_yd["phi"] * 180 / pi,
            label="Measured",
        )
        ax_phi.set_ylabel("Roll angle $phi$ [°]")
        ax_phi.legend()
```

```

        ax_p.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_dutch_roll_yd.index,
            self.flight_test.df_dutch_roll_yd["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")

        ax_r.plot(
            self.simulated_dutch_roll_yd.index,
            self.simulated_dutch_roll_yd["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_dutch_roll_yd.index,
            self.flight_test.df_dutch_roll_yd["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "dutch_roll_yd_full")
        plt.show()

    def plot_aperiodic_roll(self):
        fig, (ax_p, ax_r) = plt.subplots(2, 1, figsize=(12, 6))

        ax_p.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_aperiodic_roll.index,
            self.flight_test.df_aperiodic_roll["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")
        ax_p.legend()

        ax_r.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_aperiodic_roll.index,
            self.flight_test.df_aperiodic_roll["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "aperiodic_roll")
        plt.show()

    def plot_aperiodic_roll_full(self):
        fig, (ax_b, ax_phi, ax_p, ax_r) = plt.subplots(4, 1, figsize=(12, 12))
```

```
        ax_b.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["beta"] * 180 / pi,
            label="Simulated",
        )
        ax_b.set_ylabel("Sideslip angle $beta$ [°]")

        ax_phi.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["phi"] * 180 / pi,
            label="Simulated",
        )
        ax_phi.plot(
            self.flight_test.df_aperiodic_roll.index,
            self.flight_test.df_aperiodic_roll["phi"] * 180 / pi,
            label="Measured",
        )
        ax_phi.set_ylabel("Roll angle $phi$ [°]")
        ax_phi.legend()

        ax_p.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_aperiodic_roll.index,
            self.flight_test.df_aperiodic_roll["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")

        ax_r.plot(
            self.simulated_aperiodic_roll.index,
            self.simulated_aperiodic_roll["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_aperiodic_roll.index,
            self.flight_test.df_aperiodic_roll["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "aperiodic_roll_full")
        plt.show()

    def plot_spiral(self):
        fig, (ax_p, ax_r) = plt.subplots(2, 1, figsize=(12, 6))

        ax_p.plot(
            self.simulated_spiral.index,
            self.simulated_spiral["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_spiral.index,
            self.flight_test.df_spiral["p"] * 180 / pi,
            label="Measured",
        )
```

```
        ax_p.set_ylabel("Roll rate $p$ [°/s]")
        ax_p.legend()

        ax_r.plot(
            self.simulated_spiral.index,
            self.simulated_spiral["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_spiral.index,
            self.flight_test.df_spiral["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "spiral")
        plt.show()

    def plot_spiral_full(self):
        fig, (ax_b, ax_phi, ax_p, ax_r) = plt.subplots(4, 1, figsize=(12, 12))

        ax_b.plot(
            self.simulated_spiral.index, self.simulated_spiral["beta"] * 180 / pi, label="Simulated"
        )
        ax_b.set_ylabel("Sideslip angle $beta$ [°]")

        ax_phi.plot(
            self.simulated_spiral.index, self.simulated_spiral["phi"] * 180 / pi, label="Simulated"
        )
        ax_phi.plot(
            self.flight_test.df_spiral.index,
            self.flight_test.df_spiral["phi"] * 180 / pi,
            label="Measured",
        )
        ax_phi.set_ylabel("Roll angle $phi$ [°]")
        ax_phi.legend()

        ax_p.plot(
            self.simulated_spiral.index,
            self.simulated_spiral["p"] * 180 / pi,
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_spiral.index,
            self.flight_test.df_spiral["p"] * 180 / pi,
            label="Measured",
        )
        ax_p.set_ylabel("Roll rate $p$ [°/s]")

        ax_r.plot(
            self.simulated_spiral.index,
            self.simulated_spiral["r"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_spiral.index,
            self.flight_test.df_spiral["r"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Yaw rate $r$ [°/s]")

        format_plot()
```

```
        save_plot("C:\SVV\Results_init", "spiral_full")
        plt.show()

    def plot_phugoid(self):
        fig, (ax_p, ax_r) = plt.subplots(2, 1, figsize=(12, 6))
        ax_p.plot(
            self.simulated_phugoid.index,
            self.simulated_phugoid["u_hat"] * self.flight_test.df_phugoid["tas"].iloc[0]
            + self.flight_test.df_phugoid["tas"].iloc[0],
            label="Simulated",
        )
        ax_p.plot(
            self.flight_test.df_phugoid.index,
            self.flight_test.df_phugoid["tas"],
            label="Measured",
        )
        ax_p.set_ylabel("True airspeed $V_{TAS}$ [m/s]")
        ax_p.legend()

        ax_r.plot(
            self.simulated_phugoid.index,
            self.simulated_phugoid["q"] * 180 / pi,
        )
        ax_r.plot(
            self.flight_test.df_phugoid.index,
            self.flight_test.df_phugoid["q"] * 180 / pi,
        )
        ax_r.set_xlabel("Time [s]")
        ax_r.set_ylabel("Pitch rate $q$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "phugoid")
        plt.show()

    def plot_phugoid_full(self):
        fig, (ax_u, ax_alpha, ax_theta, ax_q) = plt.subplots(4, 1, figsize=(12, 12))
        ax_u.plot(
            self.simulated_phugoid.index,
            self.simulated_phugoid["u_hat"] * self.flight_test.df_phugoid["tas"].iloc[0]
            + self.flight_test.df_phugoid["tas"].iloc[0],
            label="Simulated",
        )
        ax_u.plot(
            self.flight_test.df_phugoid.index,
            self.flight_test.df_phugoid["tas"],
            label="Measured",
        )
        ax_u.set_ylabel("True airspeed $V_{TAS}$ [m/s]")
        ax_u.legend()

        ax_alpha.plot(
            self.simulated_phugoid.index,
            self.simulated_phugoid["alpha"] * 180 / pi,
        )
        ax_alpha.plot(
            self.flight_test.df_phugoid.index,
            self.flight_test.df_phugoid["alpha"] * 180 / pi,
        )
        ax_alpha.set_ylabel("Angle of attack $alpha$ [°]")

        ax_theta.plot(
```

```
                self.simulated_phugoid.index,
                self.simulated_phugoid["theta"] * 180 / pi,
            )
            ax_theta.plot(
                self.flight_test.df_phugoid.index,
                self.flight_test.df_phugoid["theta"] * 180 / pi,
            )
            ax_theta.set_ylabel("Pitch angle $theta$ [°]")

            ax_q.plot(
                self.simulated_phugoid.index,
                self.simulated_phugoid["q"] * 180 / pi,
            )
            ax_q.plot(
                self.flight_test.df_phugoid.index,
                self.flight_test.df_phugoid["q"] * 180 / pi,
            )
            ax_q.set_xlabel("Time [s]")
            ax_q.set_ylabel("Pitch rate $q$ [°/s]")

            format_plot()
            save_plot("C:\SVV\Results_init", "phugoid_full")
            plt.show()

        def plot_short_period(self):
            fig, (ax_u, ax_q) = plt.subplots(2, 1, figsize=(12, 6))
            ax_u.plot(
                self.simulated_short_period.index,
                self.simulated_short_period["u_hat"] * self.flight_test.df_phugoid["tas"].iloc[0]
                + self.flight_test.df_phugoid["tas"].iloc[0],
                label="Simulated",
            )
            ax_u.plot(
                self.flight_test.df_short_period.index,
                self.flight_test.df_short_period["tas"],
                label="Measured",
            )
            ax_u.set_ylabel("True airspeed $V_{TAS}$ [m/s]")
            ax_u.legend()

            ax_q.plot(
                self.simulated_short_period.index,
                self.simulated_short_period["q"] * 180 / pi,
            )
            ax_q.plot(
                self.flight_test.df_short_period.index,
                self.flight_test.df_short_period["q"] * 180 / pi,
            )
            ax_q.set_xlabel("Time [s]")
            ax_q.set_ylabel("Pitch rate $q$ [°/s]")

            format_plot()
            save_plot("C:\SVV\Results_init", "short_period")
            plt.show()

        def plot_short_period_full(self):
            fig, (ax_u, ax_alpha, ax_theta, ax_q) = plt.subplots(4, 1, figsize=(12, 12))
            ax_u.plot(
                self.simulated_short_period.index,
                self.simulated_short_period["u_hat"] * self.flight_test.df_phugoid["tas"].iloc[0]
                + self.flight_test.df_phugoid["tas"].iloc[0],
```

```
            label="Simulated",
        )
        ax_u.plot(
            self.flight_test.df_short_period.index,
            self.flight_test.df_short_period["tas"],
            label="Measured",
        )
        ax_u.set_ylabel("True airspeed $V_{TAS}$ [m/s]")
        ax_u.legend()

        ax_alpha.plot(
            self.simulated_short_period.index,
            self.simulated_short_period["alpha"] * 180 / pi,
        )
        ax_alpha.plot(
            self.flight_test.df_short_period.index,
            self.flight_test.df_short_period["alpha"] * 180 / pi,
        )
        ax_alpha.set_ylabel("Angle of attack $alpha$ [°]")

        ax_theta.plot(
            self.simulated_short_period.index,
            self.simulated_short_period["theta"] * 180 / pi,
        )
        ax_theta.plot(
            self.flight_test.df_short_period.index,
            self.flight_test.df_short_period["theta"] * 180 / pi,
        )
        ax_theta.set_ylabel("Pitch angle $theta$ [°]")

        ax_q.plot(
            self.simulated_short_period.index,
            self.simulated_short_period["q"] * 180 / pi,
        )
        ax_q.plot(
            self.flight_test.df_short_period.index,
            self.flight_test.df_short_period["q"] * 180 / pi,
        )
        ax_q.set_xlabel("Time [s]")
        ax_q.set_ylabel("Pitch rate $q$ [°/s]")

        format_plot()
        save_plot("C:\SVV\Results_init", "short_period_full")
        plt.show()
```

fd/Verfication/eigenvalues.py

```
from fd.analysis.flight_test import FlightTest
from fd.simulation.aircraft_model import AircraftModel
from fd.simulation.simulation import Simulation
from fd.structs import AerodynamicParameters
from fd.validation.comparison import SimulatedMeasuredComparison
import control.matlab as ml
import numpy as np
from fd.plotting import *


flight_test = FlightTest("data/B24")
aero_params = AerodynamicParameters(
    C_L_alpha=4.758556374647304,
    alpha_0=-0.02312478307006348,
    C_D_0=0.023439123324849084,
```

```
    C_m_alpha=-0.5554065208385275,
    C_m_delta=-1.3380975545274032,
    e=1.0713238368125688,
)

aircraft_model = AircraftModel(aero_params)
A, B, C, D = aircraft_model.get_state_space_matrices_symmetric(5000, 150, 0.6, 0)
eig = np.linalg.eig(A)[0]

x_sym = eig.real
y_sym = eig.imag

A, B, C, D = aircraft_model.get_state_space_matrices_asymmetric(5000, 150, 0.6, 0, 0.8)
eigassym = np.linalg.eig(A)[0]


x_sym = eig.real
y_sym = eig.imag

x_assym = eigassym.real
y_assym = eigassym.imag


plt.scatter(x_sym, y_sym, marker="x")
plt.ylabel("Imaginary part")
plt.xlabel("Real part")
format_plot()
save_plot("data/", "eig_symmetric")
plt.show()

plt.scatter(x_assym, y_assym, marker="x")
plt.ylabel("Imaginary part")
plt.xlabel("Real part")
format_plot()
save_plot("data/", "eig_asymmetric")
plt.show()
```

fd/Verfication/integral_verification.py

```
from fd.analysis.flight_test import FlightTest
from fd.simulation.aircraft_model import AircraftModel
from fd.simulation.simulation import Simulation
from fd.structs import AerodynamicParameters
from fd.validation.comparison import SimulatedMeasuredComparison
import control.matlab as ml
import numpy as np
import matplotlib.pyplot as plt
from fd.plotting import *

test = "pulse_aileron"

if test == "pulse_elevator":
    flight_test = FlightTest("data/B24")
    aero_params = AerodynamicParameters(
        C_L_alpha=4.758556374647304,
        alpha_0=-0.023124783070063 48,
        C_D_0=0.023439123324849084,
        C_m_alpha=-0.5554065208385275,
        C_m_delta=-1.3380975545274032,
        e=1.0713238368125688,
    )

```

```
    aircraft_model = AircraftModel(aero_params)
    A, B, C, D = aircraft_model.get_state_space_matrices_symmetric(4500, 150, 0.8, 0)
    # print(np.linalg.eig(A)[0])
    sys = ml.ss(A, B, C, D)
    t = np.linspace(0, 10, 10000)
    x0 = [[0], [0], [0], [0]]
    u = np.zeros([len(t), 1])
    u[0] = 0.1
    yout, t, xout = ml.lsim(sys, u, t, x0)
    plt.figure(figsize=(12, 3))
    plt.plot(t, xout[:, 1])
    plt.ylabel("$alpha$ [rad]")
    plt.xlabel("Time [s]")
    format_plot()
    save_plot("data/", "int_test_pulse_elev")
    plt.show()
    # aircraft_model.get_response_plots_symmetric(sys, x0, t, u, 150)

elif test == "step_elevator":
    flight_test = FlightTest("data/B24")
    aero_params = AerodynamicParameters(
        C_L_alpha=4.758556374647304,
        alpha_0=-0.023124783070063480,
        C_D_0=0.023439123324849084,
        C_m_alpha=-0.5554065208385275,
        C_m_delta=-1.3380975545274032,
        e=1.0713238368125688,
    )

    aircraft_model = AircraftModel(aero_params)
    A, B, C, D = aircraft_model.get_state_space_matrices_symmetric(4500, 150, 0.8, 0)
    sys = ml.ss(A, B, C, D)
    t = np.linspace(0, 400, 10000)
    x0 = [[0], [0], [0], [0]]
    u = np.ones([len(t), 1])
    u = u * 0.01
    yout, t, xout = ml.lsim(sys, u, t, x0)
    plt.figure(figsize=(6, 3))
    plt.plot(t, xout[:, 3])
    plt.ylabel("$q$ [rad]")
    plt.xlabel("Time [s]")
    format_plot()
    save_plot("data/", "int_test_step_elev_q")
    plt.show()
    # aircraft_model.get_response_plots_symmetric(sys, x0, t, u, 150)

elif test == "pulse_rudder":
    flight_test = FlightTest("data/B24")
    aero_params = AerodynamicParameters(
        C_L_alpha=4.758556374647304,
        alpha_0=-0.023124783070063480,
        C_D_0=0.023439123324849084,
        C_m_alpha=-0.5554065208385275,
        C_m_delta=-1.3380975545274032,
        e=1.0713238368125688,
    )

    aircraft_model = AircraftModel(aero_params)
    A, B, C, D = aircraft_model.get_state_space_matrices_asymmetric(4500, 150, 0.8, 0, 0.8)
    # print(np.linalg.eig(A)[0])
    sys = ml.ss(A, B, C, D)
```

```
    t = np.linspace(0, 10, 1000)
    x0 = [[0], [0], [0], [0]]
    u = np.zeros([len(t), 2])
    inp = np.ones([10, 1])
    u[0, 1] = 0.01
    # u[1,1] = -0.01
    # u[0:10, 1:] = inp*0.1
    # u[10:20, 1:] = inp * -0.1
    yout, t, xout = ml.lsim(sys, u, t, x0)
    plt.figure(figsize=(12, 3))
    plt.plot(t, xout[:, 3])
    plt.ylabel("$r$ [rad/sec]")
    plt.xlabel("Time [s]")
    format_plot()
    save_plot("data/", "int_test_pulse_rudder")
    plt.show()
    # aircraft_model.get_response_plots_asymmetric(sys, x0, t, u, 150)

elif test == "pulse_aileron":
    flight_test = FlightTest("data/B24")
    aero_params = AerodynamicParameters(
        C_L_alpha=4.758556374647304,
        alpha_0=-0.023124783070063848,
        C_D_0=0.023439123324849084,
        C_m_alpha=-0.5554065208385275,
        C_m_delta=-1.3380975545274032,
        e=1.0713238368125688,
    )

    aircraft_model = AircraftModel(aero_params)
    A, B, C, D = aircraft_model.get_state_space_matrices_asymmetric(4500, 150, 0.8, 0, 0.8)
    # print(np.linalg.eig(A)[0])
    sys = ml.ss(A, B, C, D)
    t = np.linspace(0, 10, 1000)
    x0 = [[0], [0], [0], [0]]
    u = np.zeros([len(t), 2])
    inp = np.ones([10, 1])
    u[0, 0] = 0.01
    # u[1,1] = -0.01
    # u[0:10, 1:] = inp*0.1
    # u[10:20, 1:] = inp * -0.1
    yout, t, xout = ml.lsim(sys, u, t, x0)
    plt.figure(figsize=(12, 3))
    plt.plot(t, xout[:, 2])
    plt.ylabel("$p$ [rad/sec]")
    plt.xlabel("Time [s]")
    format_plot()
    # save_plot("data/", "int_test_pulse_aileron")
    plt.show()
    # aircraft_model.get_response_plots_asymmetric(sys, x0, t, u, 150)

elif test == "step_aileron":
    flight_test = FlightTest("data/B24")
    aero_params = AerodynamicParameters(
        C_L_alpha=4.758556374647304,
        alpha_0=-0.023124783070063848,
        C_D_0=0.023439123324849084,
        C_m_alpha=-0.5554065208385275,
        C_m_delta=-1.3380975545274032,
        e=1.0713238368125688,
    )
```

```python

    aircraft_model = AircraftModel(aero_params)
    A, B, C, D = aircraft_model.get_state_space_matrices_asymmetric(4500, 150, 0.8, 0, 0.8)
    # print(np.linalg.eig(A)[0])
    sys = ml.ss(A, B, C, D)
    t = np.linspace(0, 100, 1000)
    x0 = [[0], [0], [0], [0]]
    u = np.zeros([len(t), 2])
    inp = np.ones([len(t), 1])
    u[:, :1] = 0.01 * inp
    # print(u)
    # u[1,1] = -0.01
    # u[0:10, 1:] = inp*0.1
    # u[10:20, 1:] = inp * -0.1
    yout, t, xout = ml.lsim(sys, u, t, x0)
    plt.figure(figsize=(12, 3))
    plt.plot(t, xout[:, 2])
    plt.ylabel("$p$ [rad/sec]")
    plt.xlabel("Time [s]")
    format_plot()
    save_plot("data/", "int_test_step_aileron")
    plt.show()
    # aircraft_model.get_response_plots_asymmetric(sys, x0, t, u, 150)
```

tests/test_conversion.py

```python
import datetime
from unittest import TestCase

from numpy.testing import assert_allclose

from fd import conversion


class TestConversion(TestCase):
    def test_deg_to_rad(self):
        assert_allclose(conversion.deg_to_rad(0), 0)
        assert_allclose(conversion.deg_to_rad(90), 1.570796326794897)
        assert_allclose(conversion.deg_to_rad(22.0), 0.38397243543)  # randomly generated

    def test_lbshr_to_kgs(self):
        assert_allclose(conversion.lbshr_to_kgs(125), 0.015749735069444)
        assert_allclose(conversion.lbshr_to_kgs(0), 0)

    def test_psi_to_Pa(self):
        assert_allclose(conversion.psi_to_Pa(25), 172368.925)
        assert_allclose(conversion.psi_to_Pa(0), 0)

    def test_ftmin_to_ms(self):
        assert_allclose(conversion.ftmin_to_ms(0), 0)
        assert_allclose(conversion.ftmin_to_ms(16.1), 0.081788)

    def test_lbs_to_kg(self):
        assert_allclose(conversion.lbs_to_kg(2.2), 0.997903214)
        assert_allclose(conversion.lbs_to_kg(0), 0)

    def test_kts_to_ms(self):
        assert_allclose(conversion.kts_to_ms(1.2), 0.617333333333333)
        assert_allclose(conversion.kts_to_ms(0), 0)

    def test_ft_to_m(self):
        assert_allclose(conversion.ft_to_m(6.5), 1.9812)
```

```
        assert_allclose(conversion.ft_to_m(0), 0)

    def test_in_to_m(self):
        assert_allclose(conversion.in_to_m(6.5), 0.1651)
        assert_allclose(conversion.in_to_m(0), 0)

    def test_C_to_K(self):
        assert_allclose(conversion.C_to_K(26.2), 299.35)
        assert_allclose(conversion.C_to_K(-67.9), 205.25)
        assert_allclose(conversion.C_to_K(0), 273.15)

    def test_timestamp_to_s(self):
        assert_allclose(conversion.timestamp_to_s("00:00"), 0)
        assert_allclose(conversion.timestamp_to_s("2.0:00"), 7200)
        assert_allclose(conversion.timestamp_to_s("1.15:00"), 4500)
        assert_allclose(conversion.timestamp_to_s(" 1.15:15"), 4515)
        assert_allclose(conversion.timestamp_to_s("1:15:00"), 4500)
        assert_allclose(conversion.timestamp_to_s("1:15:15 "), 4515)
        assert_allclose(conversion.timestamp_to_s("1.15"), 4500)
        assert_allclose(conversion.timestamp_to_s("2"), 120)
        assert_allclose(conversion.timestamp_to_s("5"), 300)
        assert_allclose(conversion.timestamp_to_s(datetime.time(0, 0)), 0)
        assert_allclose(conversion.timestamp_to_s(datetime.time(2, 0)), 120)
        assert_allclose(conversion.timestamp_to_s(datetime.time(1, 15)), 75)
```

tests/test_util.py

```
from unittest import TestCase

import pandas as pd
from pandas._testing import assert_frame_equal

from fd.util import mean_not_none, get_closest, mean_not_nan_df


class TestUtil(TestCase):
    def test_get_closest(self):
        df = pd.DataFrame([[1], [2], [3], [4]], index=[0, 1.3, 4.5, 5.6])

        # Single rows
        self.assertEqual(get_closest(df, -3)[0], 1)
        self.assertEqual(get_closest(df, 0)[0], 1)
        self.assertEqual(get_closest(df, 0.5)[0], 2)
        self.assertEqual(get_closest(df, 1.3)[0], 2)
        self.assertEqual(get_closest(df, 4.51)[0], 4)
        self.assertEqual(get_closest(df, 100)[0], 4)

        # Multiple rows
        self.assertListEqual(list(get_closest(df, [0.5, 0.5, 0.5])[0]), [2, 2, 2])
        self.assertListEqual(list(get_closest(df, [0.5, 1.3, 4.51])[0]), [2, 2, 4])
        self.assertListEqual(list(get_closest(df, [-3, 100])[0]), [1, 4])

    def test_mean_not_none(self):
        self.assertAlmostEqual(mean_not_none([0, None, 1]), 0.5)
        self.assertAlmostEqual(mean_not_none([None, 3.4, 3.8, None]), 3.6)

    def test_mean_not_nan_df(self):
        df1 = pd.DataFrame({"a": [1, 2, pd.NA], "b": [4, 5, 6]})
        df2 = pd.DataFrame({"a": [7, pd.NA, pd.NA], "b": [pd.NA, 11, 12]})
        df3 = pd.DataFrame({"a": [13, 14, pd.NA], "b": [pd.NA, 17, pd.NA]})

        df_mean_expected = pd.DataFrame({"a": [7, 8, pd.NA], "b": [4, 11, 9]})
```

```python

        df_mean = mean_not_nan_df([df1, df2, df3])

        assert_frame_equal(df_mean, df_mean_expected)
```

tests/test_simulation/constants_Cessna_Ce500.py

```python
# Aircraft geometry:
V = 59.9
S = 24.2  # wing area [m^2]
lh = 5.5  # tail length [m]
c = 2.022  # mean aerodynamic cord [m]
KY2 = 0.980
KX2 = 0.012
KZ2 = 0.037
KXZ = 0.002
th0 = 0
muc = 102.7
mub = 15.5
V0 = 59.9
m = 4547.8
xcg = 0.3 * c


# Stability derivatives:
CX0 = 0
CXu = -0.2199
CXa = 0.4653
CXadot = 0
CXq = 0
CXde = 0

CYb = -0.9896
CYp = -0.0870
CYr = 0.4300
CYda = 0
CYdr = 0.3037
CYbdot = 0

CZ0 = -1.1360
CZu = -2.2720
CZa = -5.1600
CZadot = -1.4300
CZq = -3.8600
CZde = -0.6238

Cmu = 0
Cma = -0.4300
Cmadot = -3.7000
Cmq = -7.0400
Cmde = -1.5530

Cnb = 0.1638
Cnp = -0.0108
Cnr = -0.1930
Cnda = 0.0286
Cndr = -0.1261
Cnbdot = 0


Clb = -0.0772
Clp = -0.3444
```

```
Clr = 0.2800
Clda = -0.2349
Cldr = 0.0286

b = 13.36
g = 9.80665
CL = 1.1360
```

tests/test_simulation/test_eigenvalues.py

```
import unittest
from unittest import skip

import numpy as np
from numpy.testing import assert_allclose

from fd.simulation.aircraft_model import AircraftModel
from fd.structs import AerodynamicParameters
from tests.test_simulation.constants_Cessna_Ce500 import *


class TestEigenvalues(unittest.TestCase):
    def test_type_eigenvalues_symmetric(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_symmetric(m, V0, rho, th0)
        eigenvalues, eigenvectors = model.get_eigenvalues_and_eigenvectors(A)
        first = eigenvalues[0] == np.conj(eigenvalues[1])
        second = eigenvalues[2] == np.conj(eigenvalues[3])

        self.assertTupleEqual((first, second), (True, True))

    @skip
    def test_type_eigenvalues_asymmetric(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        CL = 1.1360
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_asymmetric(m, V0, rho, th0, CL)
        eigenvalues, eigenvectors = model.get_eigenvalues_and_eigenvectors(A)
        self.assertTrue(eigenvalues[0] < 0)
        self.assertTrue(eigenvalues[1] == np.conj(eigenvalues[2]))
        self.assertTrue(eigenvalues[3] > 0)

    @skip
    def test_shortperiod_eigenvalues(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
```

```
        rho = 0.904627056
        th0 = 0
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_symmetric(m, V0, rho, th0)
        print(model.get_eigenvalues_and_eigenvectors(A)[0])
        eig1, eig2 = model.get_idealized_shortperiod_eigenvalues(m, rho, V0)
        eigenvalues2 = complex(-0.039161, -0.037971) * V0 / c
        eigenvalues1 = complex(-0.039161, -0.037971) * V0 / c
        self.assertAlmostEqual(eig1, eigenvalues2)
        self.assertAlmostEqual(eig2, eigenvalues1)

    @skip
    def test_phugoid_eigenvalues(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_symmetric(m, V0, rho, th0)
        # print(model.get_eigenvalues_and_eigenvectors(A)[0])
        eig1, eig2 = model.get_idealized_phugoid_eigenvalues(m, rho, V0, th0)
        eigenvalues2 = complex(-0.00029107, 0.0066006) * V0 / c
        eigenvalues1 = complex(-0.00029107, -0.0066006) * V0 / c
        self.assertAlmostEqual(eig1, eigenvalues2)
        self.assertAlmostEqual(eig2, eigenvalues1)

    @skip
    def test_aperiodicroll_eigenvalues(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        CL = 1.1360
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_asymmetric(m, V0, rho, th0, CL)

        A_prim = 4 * muc**2 * KY2 * (CZadot - 2 * muc)
        B_prim = (
            Cmadot * 2 * muc * (CZq + 2 * muc)
            - Cmq * 2 * muc * (CZadot - 2 * muc)
            - 2 * muc * KY2 * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
        )
        C_prim = (
            Cma * 2 * muc * (CZq + 2 * muc)
            - Cmadot * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
            + Cmq * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
            + 2 * muc * KY2 * (CXa * CZu - CZa * CXu)
        )
        D_prim = (
            Cmu * (CXa * (CZq + 2 * muc) - CZ0 * (CZadot - 2 * muc))
            - Cma * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
            + Cmadot * (CX0 * CXu - CZ0 * CZu)
            + Cmq * (CXu * CZa - CZu * CXa)
        )
        E_prim = -Cmu * (CX0 * CXa + CZ0 * CZa) + Cma * (CX0 * CXu + CZ0 * CZu)
```

```
        p = (E_prim, D_prim, C_prim, B_prim, A_prim)

        roots = np.polynomial.polynomial.polyroots(p)
        eig1 = model.get_aperiodicroll_eigenvalues(m, rho, V0, A)
        eigenvalues1 = -0.3291 * V0 / b

        self.assertAlmostEqual(roots[2], eig1)

    @skip
    def test_dutchroll_eigenvalues(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        CL = 1.1360
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_asymmetric(m, V0, rho, th0, CL)
        # print(model.get_eigenvalues_and_eigenvectors(A)[0])
        eig1, eig2 = model.get_dutchroll_eigenvalues(m, rho, V0, A)
        eigenvalues1 = complex(-0.0313, 0.3314) * V0 / b
        eigenvalues2 = complex(-0.0313, -0.3314) * V0 / b
        self.assertAlmostEqual(eig1, eigenvalues1)
        self.assertAlmostEqual(eig2, eigenvalues2)

    @skip
    def test_spiral_eigenvalues(self):
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        CL = 1.1360
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_asymmetric(m, V0, rho, th0, CL)
        # print(model.get_eigenvalues_and_eigenvectors(A)[0])
        eig1 = model.get_spiral_eigenvalues(m, rho, V0, CL, A)
        eigenvalues1 = -0.0108 * V0 / b
        self.assertAlmostEqual(eig1, eigenvalues1)

    def test_Routh_symm(self):
        A_prim = 4 * muc**2 * KY2 * (CZadot - 2 * muc)
        B_prim = (
            Cmadot * 2 * muc * (CZq + 2 * muc)
            - Cmq * 2 * muc * (CZadot - 2 * muc)
            - 2 * muc * KY2 * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
        )
        C_prim = (
            Cma * 2 * muc * (CZq + 2 * muc)
            - Cmadot * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
            + Cmq * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
            + 2 * muc * KY2 * (CXa * CZu - CZa * CXu)
        )
        D_prim = (
            Cmu * (CXa * (CZq + 2 * muc) - CZ0 * (CZadot - 2 * muc))
            - Cma * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
            + Cmadot * (CX0 * CXu - CZ0 * CZu)
```

```python
            + Cmq * (CXu * CZa - CZu * CXa)
        )
        E_prim = -Cmu * (CX0 * CXa + CZ0 * CZa) + Cma * (CX0 * CXu + CZ0 * CZu)
        R = B_prim * C_prim * D_prim - A_prim * D_prim**2 - B_prim**2 * E_prim
        np.testing.assert_equal(R > 0, True)

    def test_Routh_asymm(self):
        A_prim = 16 * mub**3 * (KX2 * KZ2 - KXZ**2)
        B_prim = (
            -4
            * mub**2
            * (2 * CYb * (KX2 * KZ2 - KXZ**2) + Cnr * KX2 + Clp * KZ2 + (Clr + Cnp) * KXZ)
        )
        C_prim = (
            2
            * mub
            * (
                (CYb * Cnr - CYr * Cnb) * KX2
                + (CYb * Clp - Clb * CYp) * KZ2
                + ((CYb * Cnp - Cnb * CYp) + (CYb * Clr - Clb * CYr)) * KXZ
                + 4 * mub * Cnb * KX2
                + 4 * mub * Clb * KXZ
                + 0.5 * (Clp * Cnr - Cnp * Clr)
            )
        )
        D_prim = (
            -4 * mub * CL * (Clb * KZ2 + Cnb * KXZ)
            + 2 * mub * (Clb * Cnp - Cnb * Clp)
            + 0.5 * CYb * (Clr * Cnp - Cnr * Clp)
            + 0.5 * CYp * (Clb * Cnr - Cnb * Clr)
            + 0.5 * CYr * (Clp * Cnb - Cnp * Clb)
        )
        E_prim = CL * (Clb * Cnr - Cnb * Clr)
        R = B_prim * C_prim * D_prim - A_prim * D_prim**2 - B_prim**2 * E_prim
        np.testing.assert_equal(R > 0, True)

    @skip
    def test_analytic_eigenvalues_symmetric(self):
        # In order to perform this test you need to:
        # 1. Change the imported constants file in aircraft model with the ones for cessna Ce500
        # 2. Comment any mub calculation out from the aircraft model
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_symmetric(m, V0, rho, th0)
        eigenvalues = model.get_eigenvalues_and_eigenvectors(A)[0]

        A_prim = 4 * muc**2 * KY2 * (CZadot - 2 * muc)
        B_prim = (
            Cmadot * 2 * muc * (CZq + 2 * muc)
            - Cmq * 2 * muc * (CZadot - 2 * muc)
            - 2 * muc * KY2 * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
        )
        C_prim = (
            Cma * 2 * muc * (CZq + 2 * muc)
            - Cmadot * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
```

```
            + Cmq * (CXu * (CZadot - 2 * muc) - 2 * muc * CZa)
            + 2 * muc * KY2 * (CXa * CZu - CZa * CXu)
        )
        D_prim = (
            Cmu * (CXa * (CZq + 2 * muc) - CZ0 * (CZadot - 2 * muc))
            - Cma * (2 * muc * CX0 + CXu * (CZq + 2 * muc))
            + Cmadot * (CX0 * CXu - CZ0 * CZu)
            + Cmq * (CXu * CZa - CZu * CXa)
        )
        E_prim = -Cmu * (CX0 * CXa + CZ0 * CZa) + Cma * (CX0 * CXu + CZ0 * CZu)
        p = (E_prim, D_prim, C_prim, B_prim, A_prim)
        roots = np.polynomial.polynomial.polyroots(p)

        assert_allclose(roots * V0 / c, np.sort(eigenvalues), rtol=1e-8)

    @skip
    def test_analytic_eigenvalues_asymmetric(self):
        # In order to perform this test you need to:
        # 1. Change the imported constants file in aircraft model with the ones for cessna Ce500
        # 2. Comment any mub calculation out from the aircraft model
        aero_params = AerodynamicParameters
        aero_params.C_m_alpha = -0.4300
        aero_params.C_m_delta = -1.5530
        m = 4547.8
        V0 = 59.9
        rho = 0.904627056
        th0 = 0
        CL = 1.1360
        model = AircraftModel(aero_params)
        A, B, C, D = model.get_state_space_matrices_asymmetric(m, V0, rho, th0, CL)
        eigenvalues = model.get_eigenvalues_and_eigenvectors(A)[0]
        A_prim = 16 * mub**3 * (KX2 * KZ2 - KXZ**2)
        B_prim = (
            -4
            * mub**2
            * (2 * CYb * (KX2 * KZ2 - KXZ**2) + Cnr * KX2 + Clp * KZ2 + (Clr + Cnp) * KXZ)
        )
        C_prim = (
            2
            * mub
            * (
                (CYb * Cnr - CYr * Cnb) * KX2
                + (CYb * Clp - Clb * CYp) * KZ2
                + ((CYb * Cnp - Cnb * CYp) + (CYb * Clr - Clb * CYr)) * KXZ
                + 4 * mub * Cnb * KX2
                + 4 * mub * Clb * KXZ
                + 0.5 * (Clp * Cnr - Cnp * Clr)
            )
        )
        D_prim = (
            -4 * mub * CL * (Clb * KZ2 + Cnb * KXZ)
            + 2 * mub * (Clb * Cnp - Cnb * Clp)
            + 0.5 * CYb * (Clr * Cnp - Cnr * Clp)
            + 0.5 * CYp * (Clb * Cnr - Cnb * Clr)
            + 0.5 * CYr * (Clp * Cnb - Cnp * Clb)
        )
        E_prim = CL * (Clb * Cnr - Cnb * Clr)
        p = (E_prim, D_prim, C_prim, B_prim, A_prim)
        roots = np.polynomial.polynomial.polyroots(p)

        assert_allclose(roots * V0 / b, np.sort(eigenvalues), rtol=1e-8)
```

```python


if __name__ == "__main__":
    unittest.main()
```

tests/analysis/test_thrust.py

```python
from unittest import TestCase

from numpy.testing import assert_allclose

from fd.analysis.thrust import calculate_thrust, calc_Tc


class TestThrust(TestCase):
    def test_thrust(self):
        # Calculated from Excel sheet
        # Static temperatures in this test are calculated as temperature from ISA + dT
        assert_allclose(calculate_thrust(3000, 0.4, 268.65 + 0.5, 0.1), 4096.2853587604200)
        assert_allclose(calculate_thrust(3000, 0.4, 268.65 + 0.5, 0.09), 3510.0944255666300)
        assert_allclose(calculate_thrust(5000, 0.4, 255.65 + 0.5, 0.09), 4004.8876358752600)
        assert_allclose(calculate_thrust(5000, 0.8, 255.65 + 0.5, 0.09), 2732.5546243401900)
        assert_allclose(calculate_thrust(5000, 0.1, 255.65 + 0.5, 0.09), 5369.0542444565900)
        assert_allclose(calculate_thrust(100, 0.1, 287.50 + 0.7, 0.1), 4920.5995394974200)

    def test_calc_Tc(self):
        assert_allclose(calc_Tc(2000, 300, 1.225, 15), 2.41874527589e-3)
```

tests/analysis/test_aerodynamics.py

```python
from unittest import TestCase

from numpy.testing import assert_allclose

from fd.analysis.aerodynamics import *


class TestAerodynamics(TestCase):
    def test_calc_true_V(self):
        assert_allclose(calc_true_V(600, 0.9), 441.937574777)
        assert_allclose(calc_true_V(200, 0.7), 198.452160482)
        assert_allclose(calc_true_V(555, 0.21), 99.1764547914)

    def test_calc_equivalent_V(self):
        assert_allclose(calc_equivalent_V(100, 1.225), 100)
        assert_allclose(calc_equivalent_V(250, 0.82), 204.540300904)
        assert_allclose(calc_equivalent_V(75, 0.105), 21.9577516413)

    def test_calc_CL(self):
        assert_allclose(calc_CL(1000, 10, 1.225), 0.54421769)
        assert_allclose(
            calc_CL(np.array([1000, 15000]), np.array([10, 30]), 1.225),
            np.array([0.54421769, 0.90702948]),
            rtol=1e-01,
        )
        # assert_allclose(calc_CL([1000, 15000], [10, 30]), np.array[0.54421769, 0.90702948], rtol=1e-01)

    def test_estimate_CL_alpha(self):
        assert_allclose(
            estimate_CL_alpha(np.array([0.1, 0.2, 0.3]), np.array([0, 5, 10])),
            [0.02, 0.1, -5.0],
            rtol=1e-01,
```

```
        )

    def test_calc_CD(self):
        assert_allclose(calc_CD(1000, 10, 1.225), 0.54421769, rtol=1e-01)
        assert_allclose(
            calc_CD(np.array([1000, 15000]), np.array([10, 30]), 1.225),
            np.array([0.54421769, 0.90702948]),
            rtol=1e-01,
        )

    def test_calc_CD0_e(self):
        assert_allclose(
            estimate_CD0_e(
                np.array([0.0318, 0.0532, 0.024, 0.063]), np.array([0.5, 0.84, 0.29, 0.955])
            ),
            [0.02, 0.8],
            rtol=1e-01,
        )
        assert_allclose(
            estimate_CD0_e(
                np.array([0.032, 0.053, 0.025, 0.065]), np.array([0.51, 0.83, 0.28, 0.95])
            ),
            [0.02, 0.8],
            rtol=1e-01,
        )

    def test_calc_Cmdelta(self):
        assert_allclose(calc_Cmdelta(20, 19, 2, 1, 10000, 120, 0.6), -0.037513002)
        assert_allclose(calc_Cmdelta(20.01, 19.99, 1.6, 1, 10000, 110, 0.2), -0.00446435726)

    def test_estimate_Cmalpha(self):
        assert_allclose(estimate_Cmalpha([1, 2, 3], [0.5, 1, 1.5], -0.01), 0.005)
        assert_allclose(estimate_Cmalpha([1.01, 2, 3], [0.5, 1.01, 1.5], -0.01), 0.005, rtol=1e-1)
```

tests/analysis/test_center_of_gravity.py

```
from unittest import TestCase

from numpy.testing import assert_allclose

from fd.analysis.center_of_gravity import *


class TestAerodynamics(TestCase):
    def test_lin_moment_mass(self):
        assert_allclose(lin_moment_mass(), [7.238938216, 6.598248836])

    def test_get_cg(self):
        assert_allclose(calc_cg_position(1000, 80, 80, 80, 80, 80, 80, 80, 80, 80), 7.14458657)
        assert_allclose(calc_cg_position(1000, 0, 0, 0, 0, 0, 0, 0, 0, 0), 7.37828993)
        assert_allclose(
            calc_cg_position(1000, 80, 80, 80, 80, 80, 80, 80, 80, 80, True), 7.09932165
        )
```

tests/analysis/test_reduced_values.py

```
from unittest import TestCase

from numpy.testing import assert_allclose

from fd.analysis.reduced_values import *

```

```
class TestReducedValues(TestCase):
    def test_calc_reduced_equivalent_V(self):
        assert_allclose(calc_reduced_equivalent_V(300, 60500), 300)
        assert_allclose(calc_reduced_equivalent_V(95, 100000), 73.892658634)
        assert_allclose(calc_reduced_equivalent_V(245, 20000), 426.116914708)

    def test_calc_reduced_elevator_deflection(self):
        assert_allclose(calc_reduced_elevator_deflection(3.0, -0.04, 0.01, 0.011), 3.00016)
        assert_allclose(calc_reduced_elevator_deflection(3, -0.04, 0.01, 0.01), 3.0)
        assert_allclose(calc_reduced_elevator_deflection(3, -0.04, 0.5, 0.1), 2.936)

    def test_calc_reduced_stick_force(self):
        assert_allclose(calc_reduced_stick_force(20, 60500), 20)
        assert_allclose(calc_reduced_stick_force(100, 5000), 1210)
        assert_allclose(calc_reduced_stick_force(1, 100000), 0.605)
```

tests/analysis/test_thermodynamics.py

```
from unittest import TestCase

from numpy.testing import assert_allclose

from fd.analysis.thermodynamics import *


class TestThermodynamics(TestCase):
    def test_calc_stat_pres(self):
        assert_allclose(calc_static_pressure(1000), 89870.773519)
        assert_allclose(calc_static_pressure(0), 101325)
        assert_allclose(calc_static_pressure(10672), 23815.2625371)

    def test_calc_mach(self):
        assert_allclose(calc_mach(1000, 100), 0.3116119528)
        assert_allclose(calc_mach(10672, 150), 0.85210191358)
        assert_allclose(calc_mach(0, 20), 0.05877270993)

    def test_calc_static_temp(self):
        assert_allclose(calc_static_temperature(350, 0.2), 347.222222222)
        assert_allclose(calc_static_temperature(500, 0.9), 430.292598967)
        assert_allclose(calc_static_temperature(100, 0.5), 95.2380952381)

    def test_calc_density(self):
        assert_allclose(calc_density(100000, 288), 1.20962279123)
        assert_allclose(calc_density(1005000, 600), 5.83522034489)
        assert_allclose(calc_density(80000, 200), 1.3934854555)
```

bin/generate_code_for_appendix.py

```
paths = []
paths.extend(Path("fd").glob("**/*.*"))
paths.extend(Path("tests").glob("**/*.*"))
paths.append(Path("bin/generate_code_for_appendix.py"))  # so meta

check_is_file = lambda f: f.is_file()
check_has_proper_extension = lambda f: f.suffix in [".py"]
check_is_not_init = lambda f: f.name != "__init__.py"

paths = filter(
    lambda f: check_is_file(f) and check_has_proper_extension(f) and check_is_not_init(f),
    paths,
)
```

```

with Path("data/appendix_code_generated.tex").open("w") as f:
    for code_file in paths:
        f.write("\\paragraph{" + str(code_file).replace("_", "\\_") + "}\n")
        f.write("\\begin{pythoncode}\n")
        f.write(code_file.read_text())
        f.write("\\end{pythoncode}\n")
        f.write("\n")
```