Development of a numerical simulation tool to explore thrust vector control on sounding rockets

Marcus Steen Nitschke

Thesis submitted for the degree of
Master in Electrical Engineering, Informatics and Technology
60 credits

Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2023

Development of a numerical simulation tool to explore thrust vector control on sounding rockets

Marcus Steen Nitschke

Development of a numerical simulation tool to explore thrust vector control on sounding rockets

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This Master's thesis develops a numerical simulation tool in Python to simulate rocket trajectory with Thrust Vector Control (TVC). The model implements a Proportional-Integral-Derivative (PID) controller adjusting the thrust vector to achieve an accuracy of $1.55 \cdot 10^{-5}$ rad in the vertical orientation during launch in ideal conditions. To study the potential of TVC on sounding rockets, rocket trajectories are simulated in different wind conditions, with and without TVC, to determine how TVC affects the accuracy and stability during the launch in any conditions. The simulations show an increase in accuracy and stability, but it is impossible to conclude the exact increase due to limited verification and inaccuracies in the model during launch.

Acknowledgments

I want to express my deepest gratitude to my supervisor, Professor Ketil Røed, for his guidance during my work. I am also grateful to my mother, Cecilie, for her feedback on my writing, Ole for nurturing my interest in space, and Randi for keeping me sane while finishing my thesis. I want to thank my father and all my friends and family for their support and patience.

Lastly, thanks to my fellow Master's students at 333 and everyone at SEF for making my time at the university more fun and exciting.

Contents

Chapter 1

Introduction

1.1 Motivation

At the University of Oslo (UiO), there is a particular interest in the ionosphere to research plasma due to solar weather. The ionosphere is an ionized region of the atmosphere ranging from around 60 km to 1000 km (Schunk and Nagy, 2000, p. 6). In periods of substantial solar activity, disturbances in the ionosphere from the solar wind interacting with Earth's magnetic field can cause disturbance for navigational satellites. To study this effect, the electron density, which is at its greatest between 100 km and 400 km, is measured with a multi-Needle Langmuir Probe instrument developed at UiO (Bekkeng et al., 2010; Jacobsen et al., 2010). Due to the proximity to the polar cap region, where Earth's magnetic field funnels ionized plasma, sounding rockets carrying the multi-Needle Langmuir Probe are launched from Andøya in Norway or from the eastern part of Canada.

When preparing a launch, the mission is simulated to increase safety and ensure mission success. Software such as (ASTOS, 2023), (OpenRocket, 2023) or RockSim (Apogee, 2023) are all able to simulate rocket trajectories, but there is no readily available software able to simulate rocket trajectories with Thrust Vector Control (TVC). The primary focus of this thesis is to implement this functionality.

TVC is the ability of a vehicle to change the direction of thrust generated from its engine to control its attitude and angular velocity (Sutton and Biblarz, 2016, p. 671). It is only effective while the thruster operates and produces an exhaust jet. After burnout, the rocket requires separate mechanisms to control attitude or flight path. TVC could be implemented on a second-stage booster to optimize the trajectory of the sounding rocket to spend more time in the region of interest and thus potentially increase the number of samples from each mission. TVC could also be used on a first- or single-stage booster during launch. One downside of using complex control mechanisms is an increased risk during launch, but the TVC also has the potential to reduce risks by improving accuracy.

Implementing TVC in existing open-sourced software, such as OpenRocket, would require extensive knowledge about the inner workings of the software, and the amount of work needed to study the software is comparable to building a simplified model. A new model also shows the implementation of the topics in this thesis. A

simplified model is sufficient when exploring the effects of TVC, but eventual future use might require more precise simulations. By implementing a simplified model in Python, a widely used and powerful programming language, it can later be improved if needed.

The simulation tool is tested using TVC on a single-stage sounding rocket and comparing the results to simulations without TVC in the same software and to simulations in OpenRocket. The resulting simulations can also be used to study the effect TVC has on sounding rockets during launch and whether TVC affects the accuracy and stability of the rocket during its burn time.

1.2 Objectives

The primary objective of this thesis is to develop a numerical simulation tool in Python that can simulate the trajectory of sounding rockets with TVC. The model used in this tool aims to be advanced enough to provide a reliable baseline for comparing simulations with and without TVC in an environment that is easy to develop further.

A secondary objective is to explore how sounding rockets with TVC behave during launch. Rocket trajectories are simulated in different wind conditions, with and without TVC, using the simulation tool developed in this thesis. The goal is to determine how TVC affects the accuracy during the launch of sounding rockets in any conditions by examining the rocket's orientation.

1.3 Thesis outline

Below is an overview of this thesis' contents:

Chapter 2: Theoretical Background summarizes research and basic principles providing a theoretical background for the rest of the thesis. This chapter includes information on rocketry, rocket propulsion, rocket trajectory, and numerical methods used when developing the model for simulating rocket trajectories with TVC.

Chapter 3: Methods presents a numerical model derived from first principles used to simulate rocket trajectory. This chapter explains the parameters used and all assumptions and simplifications made.

Chapter 4: Results contains results from the simulations. This chapter compares results from the tool developed in this thesis to simulations in OpenRocket. In addition, this chapter compares the results from simulations with and without TVC in the developed model.

Chapter 5: Discussion reviews the simulations from chapter 4 and discusses important considerations when implementing TVC on a sounding rocket learned when developing the model in this thesis.

Chapter 6: Conclusion summarizes the work done in this thesis and concludes the discussion in chapter 5. In addition, it presents speculations around the potential of TVC on future sounding rocket missions and possible future studies related to the work in this thesis.

Chapter 2

Theoretical Background

This chapter presents a basic introduction to theoretical material and research relevant to this thesis and introduces the theory used to develop the simulation software. It summarizes background information on rocketry, rocket propulsion, rocket trajectory, and numerical methods used when developing the model for simulating rocket trajectories with TVC.

2.1 Rocket types

A rocket is a chamber containing propellant and a small payload launched into the air. Most rockets have a pointy cylindrical shape due to their aerodynamic properties. A perfect example of this in its simplest form is a firework rocket. This rocket contains a payload consisting of firework stars and gunpowder, and the propellant is more gunpowder that sends the rocket flying when ignited. In a scientific rocket, the firework stars in the payload are replaced with sensors and systems for storing, processing, and transmitting data. Larger rockets can also contain landing vehicles and even humans. The following sections will look at different kinds of rockets, their purpose, and their method of trajectory control.

Model Rocket

Model rockets are small amateur rockets that reach low altitudes, typically between 50 m and 500 m (Estes, 2023). These rockets use engines with solid propellant, generally with a thrust between 5 N and 30 N. Most model rockets are spin-stabilized, but some rely on thrust vector controlling to stabilize the rocket mid-flight. As all model rockets are for amateur use, they are great for educational purposes or as a hobby but not used scientifically or commercially.

Weather balloons

Although weather balloons do not fit the definition of a rocket, they still carry a payload into the atmosphere and are immensely important to any rocket mission. The balloon is made of either latex or neoprene, with a size determined by the desired ascent rate, the type of gas used, often hydrogen or helium, and the desired maximum altitude (Dabberdt and Turtiainen, 2014, p. 273-284). Below the balloon

is a parachute and a separation line connecting to the payload. The payload consists of meteorological sensors, electronics for signal processing, and a radio transmitter to relay the measurements back to a receiver at the launch station. The sensors collect data about air pressure, temperature, wind speed and direction, humidity, location, and altitude before it bursts at around 30 km altitude. The data is then transmitted and analyzed for weather reports, which is vital for rocket missions.

Sounding Rocket

Sounding rockets carry scientific instruments into the upper atmosphere along a parabolic trajectory before parachutes deploy and the rocket falls back to Earth (NASA, 2021b). This is illustrated in Figure 2.1.

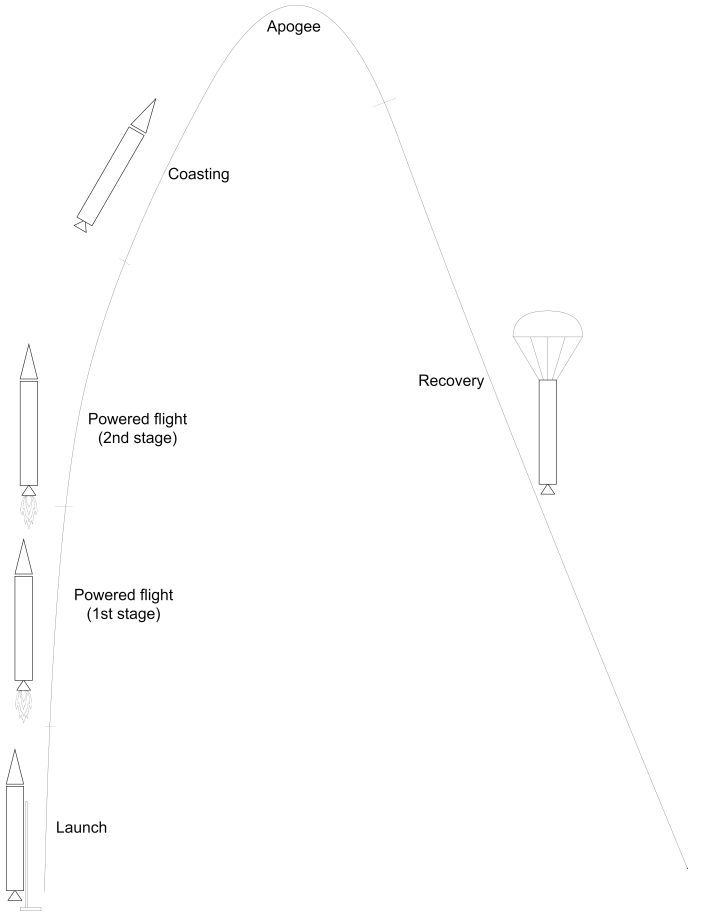

Figure 2.1: Stages of flight for sounding rockets: Launch, powered flight (one or more stages), coasting, and recovery.

Sounding rockets are used to reach altitudes that weather balloons cannot reach, and they typically spend between 5-20 minutes in space. Sounding rockets use one or more stages of engines using chemical propellants and are spin-stabilized. Depending on the research application, the sounding rockets' payload primarily consists of sensors and scientific equipment. As the rocket follows its trajectory, it uses telemetry to transmit data to the ground station. Some rockets can carry a payload of up to 200 kg, but the payload mass and the maximum altitude vary significantly between rockets.

The overall cost of sounding rocket missions is meager compared to orbiter missions (NASA, 2021c). The rockets are pre-fabricated, and the payloads are often recoverable, so the costs are spread out over multiple missions. They do not need expensive boosters or extended telemetry and tracking coverage since they never go into orbit.

Launch vehicles

A launch vehicle is a rocket able to carry a payload to outer space and into orbit (McConnaughey et al., 2012). Most of these rockets have multistage engines that use chemical propellants and thrust vector control to stabilize the vehicle during flight and to perform orbital maneuvers. NASA classifies launch vehicles according to the weight of their maximum payload:

- Small: 0–2 t payloads
- Medium: 2–20 t payloads
- Heavy: 20–50 t payloads
- Super Heavy: more than 50 t payloads

In the early days of space exploration, all launch vehicles were one-time use, with boosters that separated from their payload and disintegrated during atmospheric re-entry or on contact with the ground. In recent years, however, private spacecraft manufacturers have developed the ability to land and reuse the launch vehicles' booster stages, further reducing the costs of placing payloads into orbit.

TVC is used on most launch vehicles today because it can guide the rocket precisely to a particular flight destination (Sutton and Biblarz, 2016, p. 671-677). Soviet launch vehicles typically use multiple thrusters and hinges, and many U.S. vehicles use gimbals.

2.2 Propellant

Rocket propellants can be in a solid, liquid, or gaseous state, depending on the fuel used (Wertz et al., 2011, p. 533-541). The efficiency of the propellant used is measured in specific impulse and defined as,

$$I_{sp} = \frac{F}{\dot{m}g} = \frac{v_e}{g} \ , \tag{2.1}$$

where F is the thrust in N, $\dot{m}$ is the mass flow rate in kg/s, g is the acceleration due to gravity in m/s^2 and v_e is the exhaust velocity in m/s. Another key measure of propulsion system performance is the total change in velocity it can produce. Tsiolkovsky's rocket equation defines this as,

$$\Delta V = gI_{sp}\ ln\Big(\frac{m_0}{m_f}\Big) = v_e\ ln\Big(\frac{m_0}{m_f}\Big) \ , \tag{2.2}$$

where g is the acceleration due to gravity in m/s^2, I_{sp} is the specific impulse in s, m_0 is the wet mass of the rocket in kg, m_f is the dry mass of the rocket in kg and v_e is the

exhaust velocity in m/s. Typically the ΔV is defined by the mission requirement, and the dry mass and the specific impulse are known variables, meaning one can derive the required amount of fuel needed for the mission. Since different fuels each have a distinct specific impulse, the required ΔV, practical concerns such as cost and complexity, and safety concerns determine the fuel best suited for the mission.

Rocket propulsion systems are often classified by their energy source, including chemical, electric and nuclear, which all have advantages and disadvantages. Chemical propulsion is the only system used on sounding rockets today. It is also the only known system to produce enough thrust to propel through Earth's atmosphere without the risk of nuclear contamination. Chemical rockets use the energy from a high-pressure combustion reaction from the propellant and an oxidizer to heat, expand and accelerate the product gasses to high velocities. The physical state of the fuel can further categorize chemical propulsion systems.

Liquid Propellant

In a liquid rocket, separate tanks keep the fuel and oxidizer in a liquid state (Sutton and Biblarz, 2016, p. 189-192). A turbo pump draws the liquids through piping into the injection plate that sprays them into the combustion chamber to react, with the byproducts expanding through the nozzle. Engines using liquid propellant require complex mechanical parts and a pump to keep the fuel and oxidizer flowing. Depending on the flow pressure of fuel and oxidizer, the thrust can be varied, interrupted, or re-ignited. However, most liquid propellants are corrosive, and the propellant must be drained if a launch is postponed or canceled.

Solid Propellant

In solid propellant rockets, fuel and oxidizers are mixed and cast into a solid material that surrounds an open center (Wertz et al., 2011, p. 533-541). The flame travels through the center when ignited, consuming the propellant radially outward, and reacting gases expand through the nozzle. The fuel and oxidizer will only combust once ignited, but once it has, it is impossible to stop the reaction until the engine burns out. In addition to not requiring pumps or piping, solid propellants have a higher density, resulting in less volume for the same fuel mass. The downside, however, is a lower specific impulse than traditional liquid propellant.

Hybrid Propellant

A hybrid-propellant rocket usually has a solid fuel and a liquid oxidizer (Sutton and Biblarz, 2016, p. 593-595). The fluid oxidizer can make it possible to throttle and restart the engine just like a liquid-fueled rocket. Hybrid propellant typically offers a higher specific impulse than solid propellant and a higher density than liquid propellant. Because just one constituent is a fluid, hybrids can be simpler than liquid rockets depending on a motive force to transport the fluid into the combustion chamber. Fewer fluids typically mean fewer and smaller piping systems, valves, and pumps.

2.3 Rocket trajectory

Four forces act on the rocket body during flight: Weight, thrust, drag, and lift (Sutton and Biblarz, 2016, p. 104-105). These forces are illustrated in Figure 2.2. They all vary in magnitude and direction depending on different factors and determine the rocket trajectory. When the rocket is exposed to these forces, it is accelerated, as stated by Newton's second law:

$$F = ma \ , \tag{2.3}$$

where m is the mass in kg and a is the acceleration in $\mathrm{m/s^2}$. Unlike airplanes that use lift to counteract the forces of gravity, a rocket uses its thrust for this purpose. As the rocket uses its propellant to generate thrust, its mass gradually reduces, causing the acceleration from the same amount of force to increase. Smaller rockets typically use the lift to balance the rocket, while larger rockets use additional mechanisms to control their trajectory. Thrust is the largest force acting on the rocket while the engine is firing, and the drag is typically much greater than the lift. The acceleration due to gravity stays more or less constant during the flight.

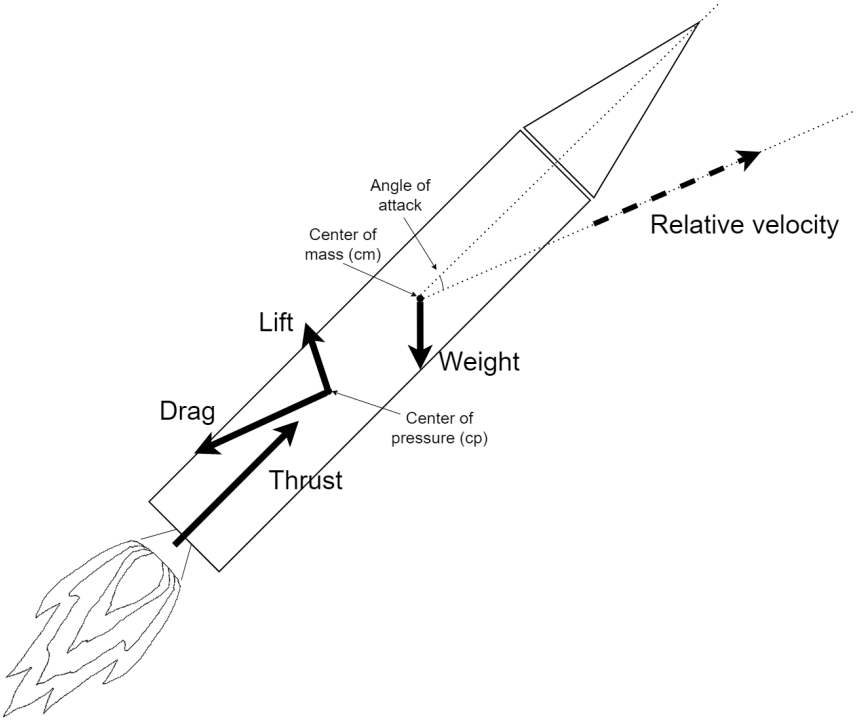

Figure 2.2: Shows the angle of attack and the forces acting on a rocket during flight: Thrust, weight, drag, and lift.

Weight

Weight is the force acting on the rocket due to gravity. This force always acts vertically downwards from the center of mass on the rocket. The magnitude of the weight force is given by the acceleration due to gravity and the rocket's mass. As the

rocket uses fuel to generate thrust, the mass gradually reduces, causing the weight to decrease as well. However, this weight reduction does not affect the rocket's acceleration. As the rocket experience less impact from gravity at higher altitudes, the acceleration due to gravity still varies during flight.

Thrust

Rockets generate thrust by ejecting matter in one direction and thus creating a force acting on the rocket engine in the opposite direction due to Newton's third law of motion. This is called rocket propulsion. In addition to the component from Equation 2.3, the relation between the surrounding air pressure and the exit pressure, and the area of the nozzle affects the total thrust,

$$F_T = \dot{m} v_e (p_0 - p) A_e \ , \tag{2.4}$$

where $\dot{m}$ is the mass flow rate in kg/s, v_e is the exhaust velocity in m/s, p_0 is the exit pressure of the nozzle in Pa, p is the atmospheric pressure in Pa and A is the area of the exit of the nozzle in m^2. To maximize the pressure thrust from the latter component in Equation 2.4, the nozzle is usually designed so that its exit pressure is equal to the ambient air pressure, where the nozzle is operating at its optimum expansion ratio, somewhere at or above sea level (Sutton and Biblarz, 2016, p. 31-33). Rockets using multiple stages often have different nozzles to maintain their efficiency. Because the air pressure decrease with altitude, the thrust, and specific impulse also change as the rocket reaches higher altitudes. Most solid rocket engines used in sounding rockets output their maximum thrust at takeoff, as seen in the example in Figure 2.3. This is when the surface area of the propellant is at its highest, causing more propellant to react and thus increasing the mass flow rate. After some time, the surface area is reduced, and the thrust generated by the engine decreases.

Drag

Drag is the force acting from the center of pressure opposite the rocket's direction of motion in Figure 2.2. This force is caused by a combination of skin friction drag caused by friction between the rocket and the surrounding air and form drag caused by a low-pressure zone behind the rocket (Hoerner, 1965, ch. 1, p. 7). If there are any irregularities in the shape of the rocket, such as fins, pockets of turbulent air arise and increase the skin drag. The drag equation (Glenn research center, 2021a) combines these contributions and defines the force from drag as,

$$F_D = \frac{1}{2} \rho v^2 C_D A \ , \tag{2.5}$$

where ρ is the air density in kg/m^3, v is the velocity of the rocket relative to the air in m/s, A is the cross-sectional area of the nose cone in m^2 and C_D is the dimensionless drag coefficient of the rocket. For most of the flight, the rocket will

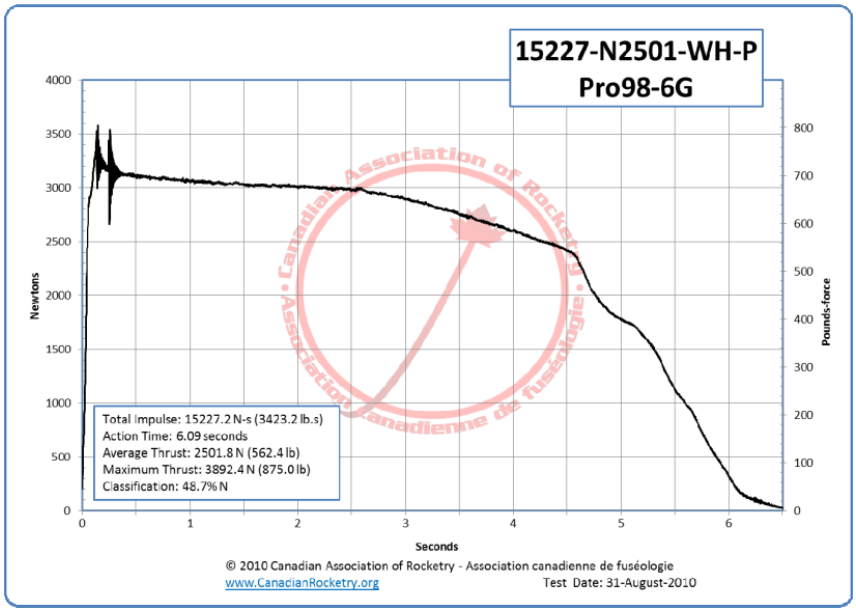

Figure 2.3: Canadian Association of Rocketry, 2010, Thrust curve for a Pro98-6G rocket engine. (http://pro38.com/products/pro98/motor/images/15227N2501-P.pdf)

experience drag approximately proportional to the velocity squared. At low speeds, however, or close to the speed of sound, the drag coefficient changes. The latter can be explained by wave drag. When the rocket passes through the air, it creates waves of pressure around its body, which compresses as the velocity increases. The ratio of the rocket's velocity to the speed of sound is called the Mach number. At Mach 1, the pressure waves combine, causing a massive increase in skin friction drag and form drag. At even higher velocities, this effect diminishes.

In addition, the air density in itself changes with atmospheric pressure and temperature as derived from the ideal gas law,

$$PV = n\overline{R}T \ , \tag{2.6}$$

where P is the atmospheric pressure in Pa, V is the volume of the air in m^3, n is the number of mol, $\overline{R}$ is the universal gas constant in J/Kmol and T is the temperature in K. By replacing,

$$n = \frac{m}{M} \ ,$$
$$\rho = \frac{m}{V} \text{ and}$$
$$R = \frac{\overline{R}}{M} \ ,$$

where m is the total mass in kg, M is the molar mass in kg/mol and R is the specific gas constant in J/kgK, the air density can be derived as,

$$\rho = \frac{P}{RT} \ . \tag{2.7}$$

The air density is halved by doubling the temperature or halving the atmospheric pressure. In a real rocket mission, these parameters are impossible to control. However, as the rocket's altitude increases, the atmospheric pressure, and temperature decrease at a different rate, causing the air density to drop (International Orginization for Standardization, 1975). This result in a reduced effect from both drag and lift as the altitude increases.

Lift

Lift is the aerodynamic force acting on the center of pressure perpendicular to the drag in Figure 2.2. Lift is produced when a solid object changes the direction of a flow of fluid (McLean, 2012, p. 266-280). When the rocket body is angled relative to the direction of motion, the surrounding air is pushed away from its natural flow. This angle is called the angle of attack. According to Newton's third law, the air has to push back with an equal and opposite magnitude to balance the forces. The angled rocket increases the pressure on the side angled towards the direction of motion, called the "leading edge," and decreases the pressure on the side facing away, called the "trailing edge." This pressure difference causes a net force from the leading edge towards the trailing edge with a magnitude defined by the lift equation (Glenn research center, 2021b) as,

$$F_L = \frac{1}{2}\rho v^2 A C_L \ , \tag{2.8}$$

where ρ is the air density in kg/m^3, v is the velocity of the rocket relative to the air in m/s, A is the wing area in m^2 and C_L is the lift coefficient. The lift coefficient depends on the rocket's shape and the attack angle. In addition, the lift coefficient includes the Mach number and the Reynolds number. Reynolds number is the ratio of inertia forces to viscous forces in liquid flow (Anderson, 2016, p. 267-268). It is a dimensionless parameter relating to the transition from laminar to turbulent flow. The lift coefficient is often found experimentally.

2.4 Attitude determination and control

All rocket missions require some trajectory control to maintain or alter their course. Orbital missions need high precision to perform orbital maneuvers, and even sounding rockets need systems to counteract the effects of wind and turbulence.

For a rocket to control its trajectory, it has to be able to control one or more of its principal axis. On a rocket, this is the pitch-, yaw- and roll-axis, and they both move and rotate along with the rocket relative to Earth (NASA, 2021a). Pitch is a controlled rotation about the vehicle's pitch axis, angling the nose of the rocket up or down relative to the direction of motion. Yaw is a rotation about

the yaw-axis, turning the nose sideways relative to the direction of motion, and roll is a rotation about the roll-axis going from the tail through the center line of the rocket to the nose. Figure 2.4 shows the pitch- and roll axis. The yaw axis would, in this figure, be pointing toward the reader, perpendicular to the pitch axis.

Most sounding rockets today utilize spin stabilization using fins to maintain control of their principal axis. This is due to its simplicity and effectiveness compared to its added mass and actuating power consumption (Sutton and Biblarz, 2016, p 671). In contrast, orbital missions typically use some form of TVC as orbital maneuvers require more precise trajectory control. TVC is also used on guided missiles for military applications.

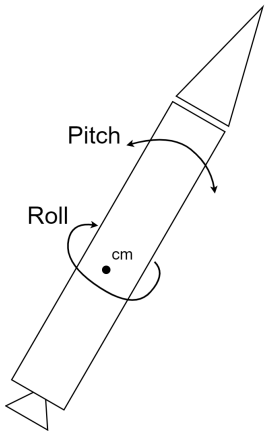

Figure 2.4: Pitch- and roll-axis for a sounding rocket.

Spin stabilization

Spin stabilization is a passive way to stabilize the rocket mid-flight. By angling the tail fins at a slight angle relative to the vehicle's direction of motion, a roll is induced to reduce the dispersion potential due to vehicle misalignment (Sounding Rockets Program Office, 2015). The roll cancels out slight variations in aerodynamics or thrust. In addition, the gyroscopic effect reduces the impact of external forces on the rocket. Just as a spinning top maintains its balance while turning, so does the rocket resist lateral movement by conserving total angular momentum. Spin stabilization works well for relatively small external forces, but this is not enough to keep the rocket stable during strong winds.

Thrust Vector Control

TVC is the ability of a vehicle to change the direction of thrust generated from its engine to generate a torque about its principal axis (Sutton and Biblarz, 2016, p. 671-686). TVC is only possible when the propulsion system generates thrust. If the rocket requires control of its attitude or flight path during other stages of the flight, separate mechanisms are needed.

In a static rocket engine without thrust vector control, the trust vector always points toward the tip of the nosecone through the center of mass, as illustrated on the left in Figure 2.5. If the rocket changes direction during the flight, the thrust vector follows. TVC enables this vector to change direction mid-flight and can be done in several ways.

Gimbaled thrust involves moving the nozzle, combustion chamber, or the entire engine assembly, including fuel and oxidizer pumps, using a universal joint. The whole

engine is pivoted on a bearing such that the thrust vector is rotated, as shown in the right side of Figure 2.5. This allows the rocket to perform pitch and yaw maneuvers. Because the line of action of the thrust vector from a single gimbaled source of thrust is always parallel to the roll axis, this maneuver requires an additional source of thrust or a separate system. This method of TVC is often used on rockets using a liquid propellant and is an effective way of TVC with only minor losses in forward thrust.

Several small auxiliary thrusters that can gimbal in one axis can perform attitude control in all axis without sacrificing engine efficiency. The downside, however, is increased complexity and weight.

The rocket can maneuver without moving its engine by using external jet vanes to deflect exhaust from the nozzle. This form of TVC enables roll control with a single engine but has the disadvantage of needing active cooling or complex materials to prevent melting. The exhaust vanes also reduce the efficiency of the rocket.

Reactive fuel injection is another type of thrust vectoring that utilizes liquid fuel in a solid booster with a fixed nozzle. The exhaust plume is modified by injecting fuel on one side of the booster, resulting in a different thrust on that side of the rocket. With three injectors, the rocket can perform pitch and yaw maneuvers.

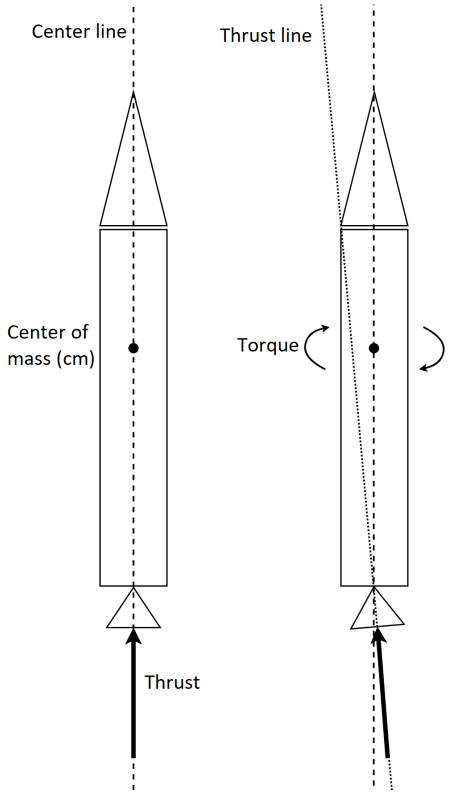

Figure 2.5: Gimbaled thrust vector control. The left rocket shows a system without TVC, and the right shows how the thrust line changes with TVC.

Rotational information

To enable TVC, the rocket use sensors to determine its rotation during flight. What sensors are used depends on the mission specifications. However, multiple different sensors are often combined to achieve the required accuracy.

A magnetometer is a sensor that measures the direction and strength of a magnetic field and is widely used in most areas of technology (You, 2017, ch. 9). On rocket missions, a magnetometer is used to measure the geomagnetic field vector information of the rocket's position, which is used to calculate the angle relative to the local magnetic field. There are numerous methods to measure the magnetic field, and each method offers a different sensitive range, resolution, and ability to measure the vector information. For a rocket near the Earth's surface, the magnetometer has to be a vector sensor that works in a medium intense magnetic field (between 1 mGs and 1 Gs) with a small size and lightweight. Flux gate magnetometers, search coil magnetometers, and MEMS magnetometers are used today.

Sun sensors utilize the bright light from the sun to measure the relative angle of the vehicle compared to the sun, and the output is a vector pointing from the sensor toward the sun (Canuto et al., 2018, p. 438). These sensors are used on all satellites and spacecraft during the sun acquisition phase for power supply, battery recharging, and initial attitude determination.

A gyroscope is a device that measures rotation. This can be done in several ways, including mechanical, laser, or MEMS technology (Fraden, 2015, p. 385-392). All gyroscopes require an additional sensor to correct for drift over the long term, but they provide high-accuracy measurements when calibrated.

Positional information

In addition to the rotation, the rocket must have sensors to determine its position during flight. Although these sensors do not directly impact the TVC, they still play a vital role in rocket missions either to make sense of the payload data, to place a satellite at the correct position in orbit, or for mission control to monitor the location of the rocket during flight.

Accelerometers are used for measuring acceleration. The acceleration is calculated from Newton's third law of motion using a force sensor to measure the force exerted by a known mass (Fraden, 2015, p. 392-397). The position is then found by integrating the acceleration twice. Three orthogonally placed accelerometers are needed to detect acceleration in all three axes. The sensors need to be lightweight and small to work on a rocket. Both capacitive and piezoresistive accelerometers fit this application and are suited for MEMS technology.

Satellite navigation systems such as the Global Positioning System (GPS) are constellations of navigational satellites used to determine the geolocation of the receiver. GPS receivers are often included on sounding rockets to monitor their position, often in combination with other systems. The specified accuracy of the publicly available GPS service is 13 m for horizontal positioning and 22 m for vertical positioning, but the actual performance is typically better than the specification (Hegarty & Chatre, 2008).

2.5 Launch conditions

When launching a rocket, the launch angles are defined as elevation and azimuth. Elevation is the angle from the horizon toward the sky. Azimuth is the angle on the horizon from North to East. These angles are found by simulating the trajectory with the latest information about atmospheric conditions.

The atmospheric conditions are complex, with varying air pressure and density and wind with variable speed and direction. A common simplification regarding the wind is using the "ballistic wind" as a metric. The ballistic wind is an idealized vector with a direction and magnitude that stays constant throughout the flight, leading to the rocket landing at the exact location as with the actual wind that changes over the flight's course. The ballistic wind behaves much like an average of the different winds encountered during the flight and is determined by using a weather balloon before launching the rocket.

The atmospheric pressure and density depend on multiple factors but are often simplified to only vary with altitude, as in the International Standard Atmosphere (International Orginization for Standardization, 1975). The International Standard Atmosphere divides the atmosphere into sections with a linear ratio between absolute temperature and altitude, as shown in Figure 2.6. This ratio, defined as the temperature lapse rate, stays constant throughout each part of the atmosphere. The temperature lapse rate and measurements from the weather balloon provide a reasonable estimate of the atmospheric pressure and density along the rocket's trajectory.

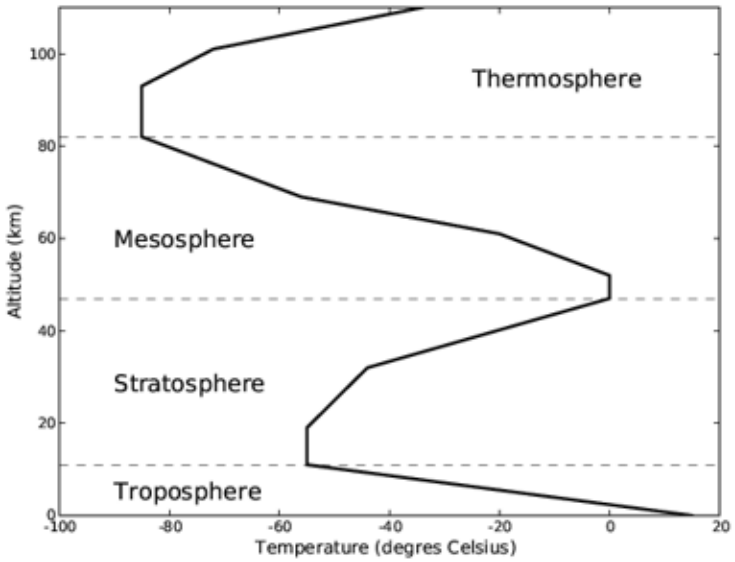

Figure 2.6: Study.com, 2010, Temperature variations in the atmosphere. (https://study.com/learn/lesson/lapse-rates-atomsphere-types-formulas-overview.html)

When launching a rocket, it is often mounted to a support structure to keep it in place before launch and to guide it during liftoff. Model rockets use simple launch rods, and sounding rockets use larger launch rails. When all systems are ready, the rocket is ignited as dictated by the launch protocol of the launch site. The support structure guides the rocket until it has high enough velocity for the aerodynamics to stabilize it for the rest of the flight. The exact length of the launch rail varies. When the rocket is attached to the launch rail, only the force component parallel to the launch rail affects the rocket (Peng et al., 2017). This eliminates any external torques that can cause instabilities. When the rocket leaves the launch rail, it has enough velocity for the drag to induce a spin due to angled tail fins, and thus spin stabilizes the rocket for the remainder of the flight. For rockets with TVC, the launch rail is not as crucial as for spin-stabilized rockets, but it still improves the settling time of the Proportional-Integral-Derivative controller (PID) and helps increase safety during the launch.

2.6 Numerical methods

A numerical method is a technique used to approximate a mathematical problem. It is a vital tool in computer programming as computers cannot solve equations analytically. In this thesis, rocket trajectory is modeled as a set of ordinary differential equations, each consisting of one unknown variable and its derivative, which are solved numerically. There are too many numerical methods to cover them all, but typically, more advanced numerical methods improve the accuracy of the approximation, reduce the simulation time and add complexity. The following section will explore the numerical methods most relevant to this thesis.

Forward Euler

The Forward Euler method is the basis of most other numerical methods and is based on a simple principle: If the velocity of a particle is known, its change in position over a small time step is approximately equal to the time step times the velocity. This motion can be described as:

$$p_1 = p_0 + dt \cdot v \ ,$$

where p_1 is the position after the time step dt, p_0 is the initial position and v is the velocity. This calculation can then be repeated for any number of time steps. If the velocity changes over time, the accuracy increases by reducing the time step. This might be accurate "enough" for a few time steps, but each step introduces a local truncation error proportional to the step size squared (Butcher, 2008, p. 66-84). The total truncation error is proportional to the step size. Although the error from the Euler method is reduced with a smaller step size, the rounding error increases. The expected total rounding error roughly equals $\epsilon/\sqrt{h}$, where ϵ is the machine precision. In Python, this precision is $\epsilon = 2.22 \cdot 10^{-16}$, and with a step size of 0.01, the total rounding error is about $2.22 \cdot 10^{-15}$.

Runge-Kutta

Runge-Kutta methods are a family of numerical methods generalizing the Euler method by allowing several evaluations of the derivative to take place in one time step (Butcher, 2008, p. 93-104). The most used of these methods often referred to as the Runge-Kutta method, is a fourth-order method using a weighted average of four increments of the time step to calculate the next value. For a given function, this can be expressed as:

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 &= hf(t_n, y_n) \\
k_2 &= hf(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}) \\
k_3 &= hf(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}) \\
k_4 &= hf(t_n + h, y_n + hk_3) \ ,
\end{aligned}
$$

where h is the step size. This results in a far more accurate approximation than the Forward Euler method with a local truncation error proportional to h^5 and a total truncation error proportional to h^4 (Butcher, 2008, p. 165-166), but with the downside of added complexity.

Chapter 3

Methods

This chapter presents a numerical model derived from first principles used to simulate rocket trajectory. The model aims to provide a reliable baseline for comparing simulations with and without TVC in an environment that is easy to develop further. All simplifications made are addressed in this chapter. The results from the simulations are later presented in chapter 4 and used in chapter 5 to discuss the system's ability to control the rocket in different wind conditions.

The theory presented in this chapter on PID is based on (Ang et al., 2005). In addition, the youtube playlist (B. Douglas [MATLAB], 2022) explains the same theory in more detail.

3.1 Assumptions and simplifications

The rocket is modeled as a rigid body axially symmetric about its roll axis. When simulating with the TVC, it is also assumed that the response time is infinitesimal with perfect accuracy compared to the output from the PID controller. In reality, this is not the case and is discussed in greater detail in Chapter 5. In addition, several other assumptions and simplifications are made.

Atmospheric conditions

The modeling of the atmospheric conditions is a limiting factor of any rocket simulation software. It is impossible to predict exact values for how the aerodynamic forces affect the rocket. All software today has some simplifications in its model. The goal of this simulation is not to improve the accuracy of other software but to provide a baseline for comparing the results of the TVC.

The atmospheric density is simplified to only vary with altitude, as in the International Standard Atmosphere (International Orginization for Standardization, 1975). This simplification uses temperature lapse rate, the rate of change in temperature as a function of altitude, to model the changes in atmospheric density. Since most of the flight in this simulation takes place in the troposphere, the overall temperature lapse rate is simplified to equal the temperature lapse rate in the troposphere defined by the ISO standard. The temperature at a given altitude is then defined as:

$$T = T_0 - Lh \ ,$$

where T_0 is the absolute temperature at sea level in K, L is the temperature lapse rate in K/m and h is the altitude in m. The pressure at a given altitude is,

$$p = p_0(1 - \frac{Lh}{T_0})^{\frac{gM}{RL}} \ ,$$

where p_0 is the standard atmospheric pressure at sea level in Pa, L is the temperature lapse rate in K/m, h is the altitude in m, T_0 is the absolute temperature at sea level in K, g is the gravitational constant in m/s^2, M is the molar mass of dry air in J/molK and R is the universal gas constant in kg/mol. The air density at a given altitude is then derived from Equation 2.7 as,

$$\rho = \frac{p_0 M}{RT_0}(1 - \frac{Lh}{T_0})^{\frac{gM}{RL}-1}$$

The wind is modeled as a ballistic wind in the horizontal plane with zero elevation.

Aerodynamics

To model the aerodynamics, the airflow around the rocket is assumed to be steady and without turbulence. The forces acting on the rocket are modeled as drag force and lift force. The coefficients from the drag and the lift combine the effects of parasitic drag, base drag, skin friction drag, and pressure drag.

The drag force is modeled using Equation 2.5 to find its magnitude and direction and simplified with a constant drag coefficient. Commercial rocket producers typically state the drag coefficient for the vehicle at low velocities. In reality, the drag coefficient changes with the Mach number. This relation is often determined experimentally. In this simulation, the drag coefficient is simplified to be higher than the stated value but stays constant with changing Mach number. It is also assumed there are no shock waves from increasing Mach number.

The area used to calculate drag is simplified to be constant and equal to the front area of the rocket. In reality, this area changes with an increased angle of attack as more of the rocket are exposed to air moving relative to the rocket. The same is true for the area used to calculate lift, where the area is simplified to the side area of the rocket.

The lift is modeled using Equation 2.8, but as with the drag coefficient, the lift coefficient is simplified. The lift coefficient of a rocket is determined experimentally and is not typically stated in the producer's catalog. In this simulation, the force from the lift is simplified by modeling the lift coefficient as a triangle function:

$$Cl = \begin{cases} Cl_{\max}(|1 - \frac{|\alpha-\beta|}{\beta}|) & \text{if } |\alpha| < \beta \\ 0 & \text{otherwise,} \end{cases}$$

where Cl_{max} is the maximum lift coefficient, α is the angle of attack and β is the critical angle. This function is shown in Figure 3.1. The model could be simplified further, but without increasing the coefficient with an increasing angle of attack, the rocket's angle would oscillate and complicate the readings from the TVC.

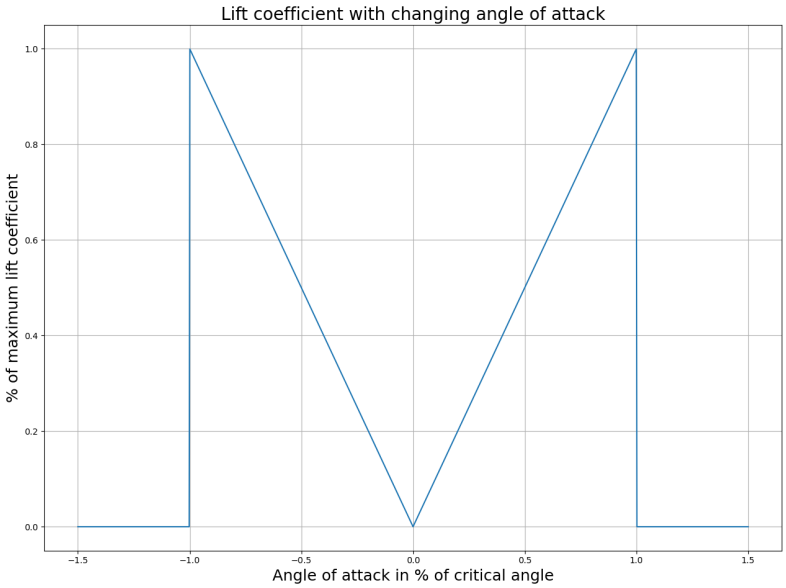

Figure 3.1: Percentage of maximum lift coefficient as a function of angle of attack.

Mass

The mass distribution of the rocket heavily influences how it behaves when exposed to external forces. All free-floating objects rotate around their center of mass, meaning the center of rotation is the same as the center of mass. When the engine burns fuel to generate thrust, the mass of the rocket decrease, and the center of mass will shift slightly depending on where the fuel is stored. As the center of mass determines the rotational center of the rocket, the moment of inertia also changes. The center of mass is assumed fixed to avoid this extra calculation. The mass moment of inertia is simplified by assuming the center of mass is placed at the geometrical center of the rocket. At this location, the mass moment of inertia can be simplified to the mass moment of inertia of a cylinder with a uniform mass density as shown in Equation 3.1:

$$I = \frac{m}{12}(3r^2 + h^2) \ , \tag{3.1}$$

where m is the mass in kg, r is the radius in m and h is the height in m. For all other calculations, the center of mass is set below the geometrical center.

In addition, the mass flow rate is assumed to be constant during the burn time. In reality, the mass flow rate varies with time and the propellant used. For solid propellants, the maximum flow rate is reached shortly after ignition when the surface area of the propellant is at its highest. This process is described in detail in Chapter 2. Assuming a constant mass flow rate makes the rocket slightly less susceptible to

forces during the burn time due to its higher mass. With a constant mass flow rate, the mass of the rocket is expressed as,

$$m(t) = \begin{cases} m_{\text{wet}} - \dot{m}t & \text{if } t < \text{burn time} \\ m_{\text{dry}} & \text{otherwise} \end{cases} \tag{3.2}$$

where m_{wet} is the wet mass of the rocket in kg, m_{dry} is the dry mass of the rocket in kg, $\dot{m}$ is the mass flow rate in kg/s and t is the time in s.

Thrust

The exhaust velocity is assumed constant, and the pressure in the combustion chamber is assumed to equal the external pressure. Combined with a constant mass flow rate, this results in a constant thrust force during the burn time. These assumptions result in a simplified expression for the thrust as defined in equation 3.3,

$$\vec{F}_{\text{thrust}}(t) = \begin{cases} F_{\text{thrust}}\vec{r} & \text{if } t < \text{burn time} \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

The simplifications make the rocket accelerate slower and lower the effect of drag and lift at the earlier part of the flight. A constant thrust also simplifies the thrust vector control. With a varying thrust, the result from the PID controller would have to be scaled with the generated thrust to keep the rocket stable. This simplification eliminates the need for such scaling.

It is assumed that the acceleration due to gravity stays constant throughout the flight. In reality, the gravitational constant is different depending on where on Earth it is measured. It also changes with altitude. For sounding rockets, this variation is slight but would result in a slightly higher apogee and longer flight time.

3.2 Coordinates

The model and the following simulation operate with two Cartesian coordinate systems as illustrated in Figure 3.2: A global coordinate system with its origin on the ground level where the rocket is launched and another local coordinate system with its origin in the rocket's center of mass and spanning the rocket's pitch, yew, and roll axis.

The global coordinates represent the rocket trajectory by defining its position in three dimensions during the flight. The local coordinate system is used to specify the distance of various forces from the rotational center to calculate torque and the orientation of the thrust vector.

To translate a vector from the local coordinate system to the global coordinate system, it is rotated using a rotational matrix. The rotation is done by combining the matrices for the pitch and yaw rotation, as all vectors in the local system are independent of the roll.

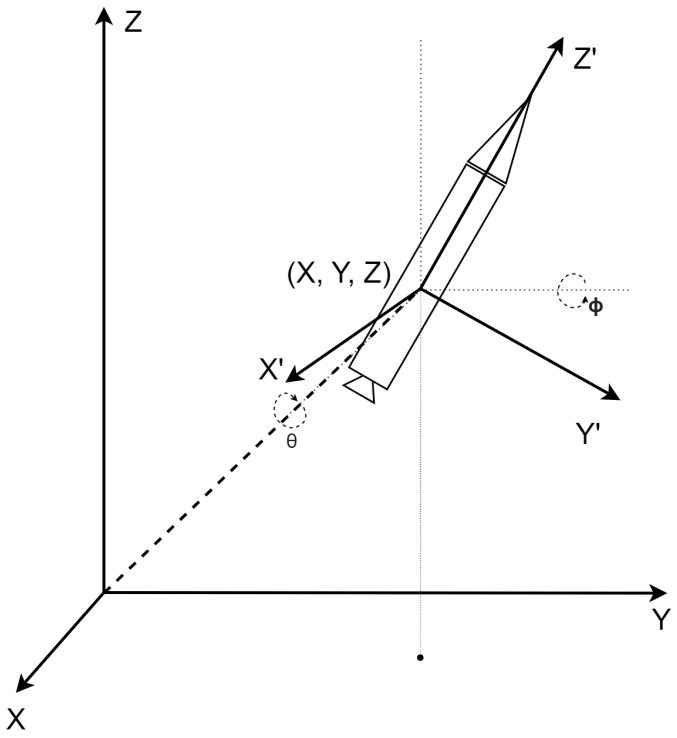

Figure 3.2: A representation of the coordinate systems used.

$$\begin{bmatrix} x_G \\ y_G \\ z_G \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & \sin(\phi) \\ 0 & 1 & 0 \\ -\sin(\phi) & 0 & \cos(\phi) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix}$$
$$\begin{bmatrix} x_G \\ y_G \\ z_G \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\theta)\sin(\phi) & \cos(\theta)\sin(\phi) \\ 0 & \cos(\theta) & -\sin(\theta) \\ -\sin(\phi) & \sin(\theta)\cos(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix}$$

where θ is the rotation about the x-axis and ϕ is the rotation about the y-axis. To translate a vector from the global coordinate system to the local coordinates, the inverse operation is needed:

$$\begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = \begin{bmatrix} \cos(\phi) & \sin(\theta)\sin(\phi) & \cos(\theta)\sin(\phi) \\ 0 & \cos(\theta) & -\sin(\theta) \\ -\sin(\phi) & \sin(\theta)\cos(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix}^T \begin{bmatrix} x_G \\ y_G \\ z_G \end{bmatrix}$$
$$\begin{bmatrix} x_L \\ y_L \\ z_L \end{bmatrix} = \begin{bmatrix} \cos(\phi) & 0 & -\sin(\phi) \\ \sin(\theta)\sin(\phi) & \cos(\theta) & \sin(\theta)\cos(\phi) \\ \cos(\theta)\sin(\phi) & -\sin(\theta) & \cos(\theta)\cos(\phi) \end{bmatrix} \begin{bmatrix} x_G \\ y_G \\ z_G \end{bmatrix}$$

3.3 Modeling rocket trajectory

A complete model for the rocket trajectory must be constructed to determine how the thrust vector control influences the rocket during flight. The position of the

rocket at any given time during flight is found by integrating the acceleration found from Newton's second law in Equation 2.3 twice. The total force of the rocket is the sum of its four components: Thrust, drag, lift, and weight. The double integral gives the combined expression for the rocket trajectory,

$$\vec{r}(t) = \iint \vec{a}(t) = \iint \frac{\vec{F}_{\text{thrust}}(t) + \vec{F}_{\text{drag}}(t) + \vec{F}_{\text{lift}}(t) + \vec{F}_{\text{weight}}(t)}{m(t)} \ . \tag{3.4}$$

Although Equation 3.4 could be shortened further, the Python implementation is more readable when the contributions from the thrust and the aerodynamic forces are kept separate. Equations 3.5-3.8 show the contribution from each force with the simplifications mentioned taken into account.

$$\vec{F}_{\text{weight}}(t) = m(t)\vec{g} \tag{3.5}$$
$$\vec{F}_{\text{thrust}}(t) = C\vec{r}(t) \tag{3.6}$$
$$\vec{F}_{\text{drag}}(t) = \frac{1}{2}\rho(t)C_D(t)A_D\vec{v^2}(t) \tag{3.7}$$
$$\vec{F}_{\text{lift}}(t) = \frac{1}{2}\rho(t)C_L(t)A_L\vec{v^2}_{\perp}(t) \tag{3.8}$$

While the force from weight always points toward negative z in the global coordinate system, the force from thrust, drag, and lift all change in direction depending on the rocket's orientation. When the center of pressure is located below the center of mass, the drag and lift force pushes the nose of the rocket toward the direction of relative motion. This creates a torque equal to the fraction of the force perpendicular to its central axis times the distance between the center of mass and the center of pressure. The torque created from thrust depends on the angle of the thrust vector control and has its moment arm equal to the distance between the engine and the center of mass. The angular acceleration created by this torque depends on its mass moment of inertia as described in Equation 3.9:

$$\alpha = \frac{\tau}{I} \ , \tag{3.9}$$

where α is the angular acceleration, τ is the torque and I is the mass moment of inertia. The angular acceleration is integrated once to find the angular velocity and twice to find the angular position.

This results in a set of ordinary differential equations (ODEs) which can be solved numerically using a modified Forward Euler step function:

$$\vec{r}(n+1) = \vec{r}(n) + h\vec{v}(n+1) \tag{3.10}$$
$$\vec{v}(n+1) = \vec{v}(n) + h\vec{a}(n) \ , \tag{3.11}$$

where n is the step and h is the step size. Higher-order numerical methods, such as the Runge-Kutta method, would result in a lower truncation and rounding error and a faster simulation but would require a slightly more complex implementation. Since the model's accuracy is not that important, and the error from the simplifications above potentially represents a much larger error, the readability of the forward Euler makes it more suited for this simulation.

3.4 A model for the thrust vector control

Regardless of which method is used to achieve Thrust Vector Control, a control algorithm is needed for the system to function. A PID controller is a control loop mechanism using feedback to correct for an offset in a desired set point value, called the error. PID is a simple and efficient control loop mechanism using both the past error, the present error, and a prediction of the future error to modify the system's gain through proportional, integral, and derivative terms. Each of these terms affects the output variable differently, and the appropriate weight of each term differs between applications. These weights are called the proportional, integral, and derivative coefficients and represent the gain for each term. In a general PID controller, the gain is then used to regulate the future process before the new output is measured and used as feedback to calculate the next error value, as shown in Figure 3.3.

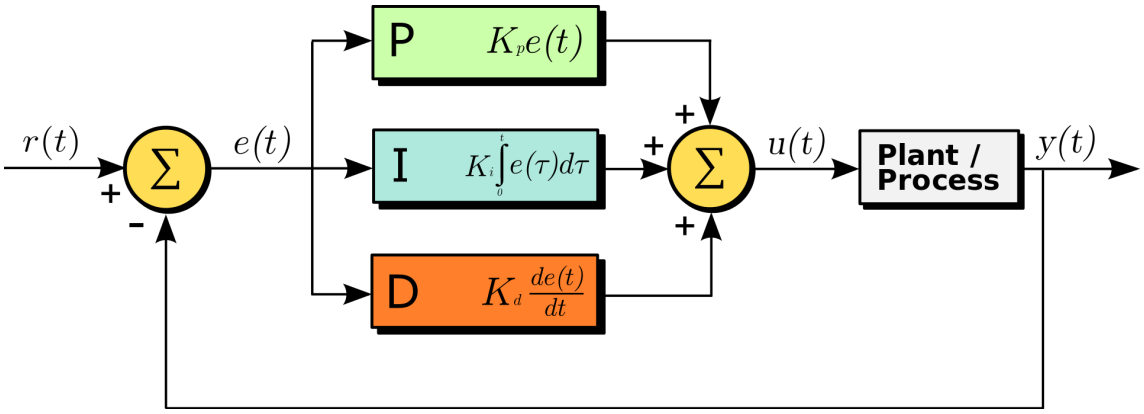

Figure 3.3: Urquizo, 2008, Block diagram of a PID-controller in a feedback loop. (https://commons.wikimedia.org/wiki/File:PID.svg)

PID is widely used to control a variety of physical applications due to its flexibility and how easy it is to implement. The application heavily influences how the PID controller behaves and how each term affects the overall system. Simple systems typically only need one or two terms, and the controller is called by the letter of these terms.

To implement a PID controller, one must first identify the input and output variables of the system and decide on the desired output value. The difference between the desired output value, called the set point, and the measured output value, called the process variable, is defined as the error value:

$$e(t) = SP - PV, \tag{3.12}$$

where SP is the set point value and PV is the measured process variable. The controller then adjusts the system's gain by using proportional, integral, and derivative terms, correcting the error over time. The gain of the PID is defined as,

$$u(t) = K_P e(t) + K_I \int_0^t e(t)dt + K_D \frac{de(t)}{dt} \ , \tag{3.13}$$

where K_P is the proportional coefficient, K_I is the coefficient of the integral response, and K_D is the coefficient of the derivative response. On a rocket, the process using the gain is the engine controllers that depending on the method of TVC, use this gain to find where to angle the engine, external boosters, or exhaust. In this simulation, there is no need for engine controllers as the gain can be used directly. The set value is the desired direction of the rocket. The output from the PID is the angle of the thrust vector, which is used to calculate the rocket's angular acceleration. The angular acceleration, in turn, affects the angular velocity and, finally, the rocket's orientation. The orientation is used as the process variable to calculate the error for the next time step. This way, the PID will always try to point the rocket toward the desired angle while the engine is firing.

Each term must be given an appropriate weight to minimize the error quickly with minimal oscillations. This is done by tuning the coefficients and is valid for any system requiring a control algorithm. To successfully tune the coefficients, it is crucial to understand how each term affects the final gain.

P-term

The proportional term relies only on the error value and applies a proportional gain to determine the ratio of output response to the error value. When the error is significant, the gain from the proportional term is high, and when the error is small, the gain is low. This term generally has the most significant impact on the control system's settling time but can cause oscillations with higher values. In a simple system, where the only thing affecting the process variable is the controller itself, a simple P-controller is enough to get to the set point quickly. In more complex systems, however, the P term alone would likely settle somewhere close to, but not exactly, at the set point. The distance between where the process variable settles and the set point is defined as the steady-state error.

Figure 3.4 illustrates how the controller would behave with different proportional coefficients in a first-order system where a P-controller is responsible for keeping the temperature at 3 degrees Celsius in a refrigerator where some heat leaks from the surroundings. With a small coefficient, the steady-state error is significant, and the settling time is extended. By increasing the coefficient, both factors are reduced, but with too large a coefficient, the system oscillates.

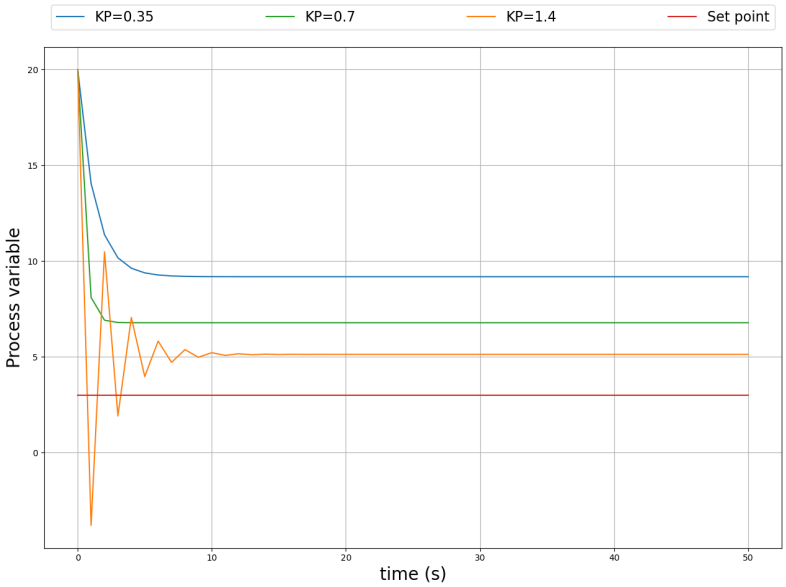

Figure 3.4: How different proportional coefficients affect a first order system.

I-term

The integral term sums the error over time and applies a gain based on the previous error. When the error is significant over some time, the gain from the I-term is significant, but sudden changes in error have little effect on the gain. Because of this, the I-term ignores the steady state error but has a longer settling time than the P-term. In addition, the I-term will either converge to but never reach the exact set value or have some overshoot depending on the coefficient.

Figure 3.5 shows how a simple I-controller would behave with different coefficients in the same first-order system as above. Here, an increasing coefficient reduces the settling time, but with a too large coefficient, the system overshoots.

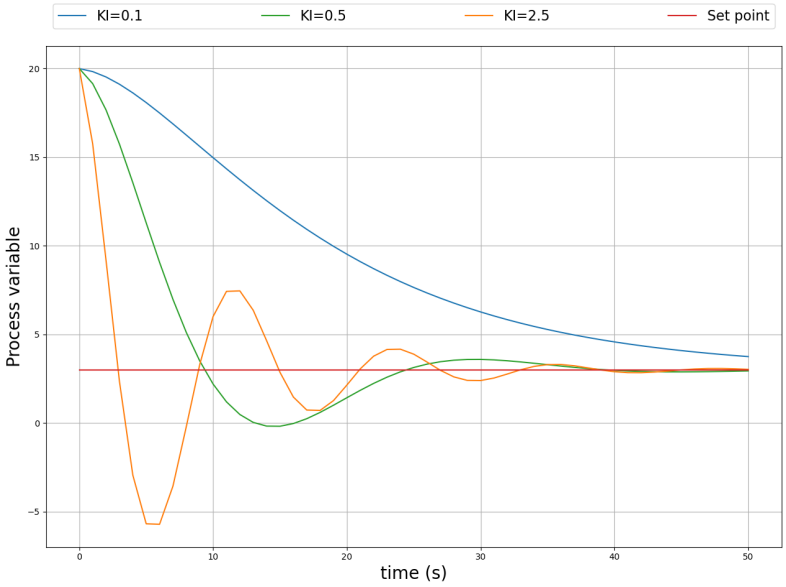

Figure 3.5: How different integral coefficients affect a first order system.

By combining the P- and I-terms, the system takes advantage of the strength of both terms to shorten the settling time and reduce the steady-state error. This is illustrated in Figure 3.6

Integration windup is a build-up of the integrated error term because the controller requests a larger gain than the system can provide. The control loop can encounter integrator windup in a system with physical constraints limiting the output, such as a saturation voltage or mechanical limits. This windup causes the I-term to increase further, resulting in a large overshoot. To avoid integrator windup in a system with physical constraints limiting the output, the I-term must be unable to increase the gain above the limit of the actuator. This can be done by clamping the error value when the I-term would increase the gain above a specific limit or by digitally limiting the number of bits storing the error term. In some applications, the clamping limit should be below the actual limit of the actuator, as the limit can change with both temperature and wear.

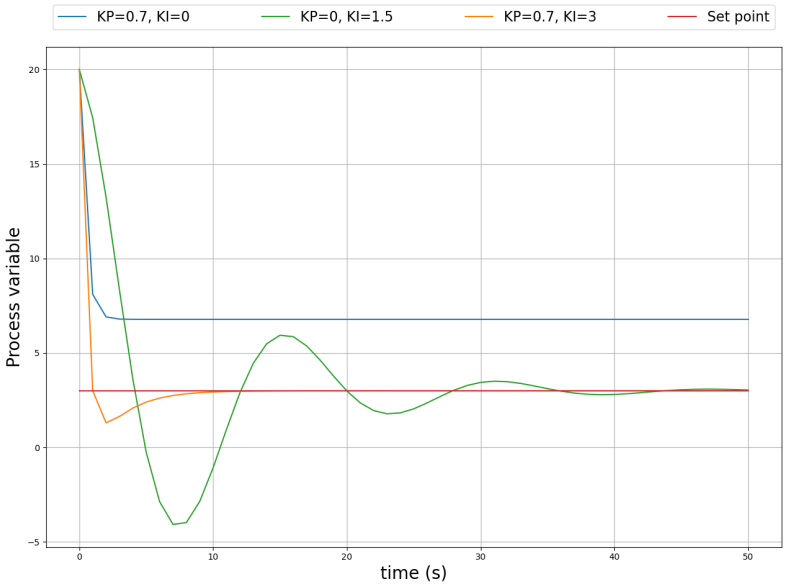

Figure 3.6: P-, I- and PI-controller in a first order system.

On a rocket, there are often physical constraints limiting the range of the PID controller. These vary between rockets, depending on the design of the engine casing, method of TVC, available power, and the saturation voltage of the actuators. In some cases, there are also constraints required by the safety protocol on the launch site. Although the burn time of most sounding rockets is relatively short, and thus the time the TVC stays in effect is limited, any overshoot is critical due to the large forces involved. A method for handling integrator windup is therefore required.

D-term

The derivative term helps stabilize the system by decreasing the gain when the error is reducing rapidly and reducing overshoot by increasing the gain when the error increases. This help speeds up the system when correctly tuned but can also lead to over-dampening when the D-term is too large. A controller with a D-term is only

used in combination with one or both of the P- and I-terms. The effect of introducing a D-term to the P- and I-controllers above is illustrated in Figures 3.7-3.8.

Another problem with a large D-term is an increased sensitivity to noise in the input signal. There will always be some noise in the input signal, but filtering can limit the sensitivity. In Figures 3.7-3.8, this is due to numerical errors and low sample size, but outside a simulation, this can be the noise from several sources. As the D-term responds more to rapid changes, it is more sensitive to high-frequency noise and should always be paired with a low-pass filter.

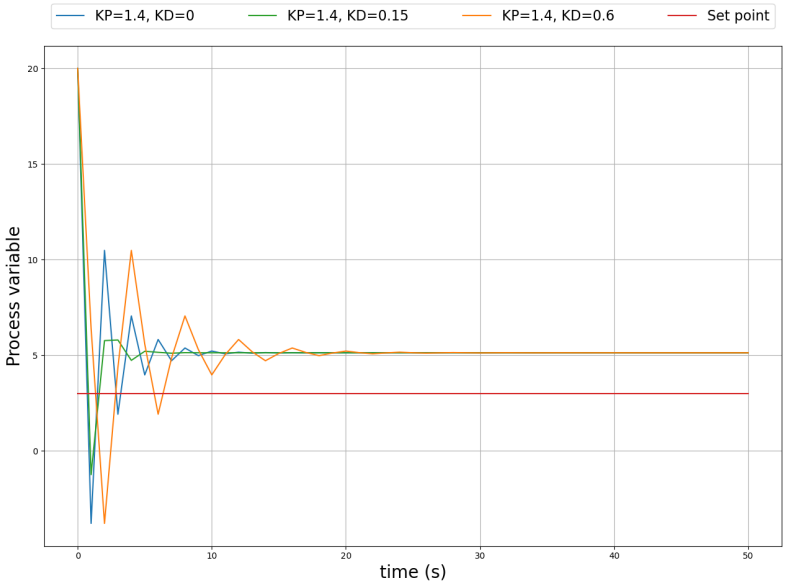

Figure 3.7: Pd-controller with different derivative coefficients in a first-order system.

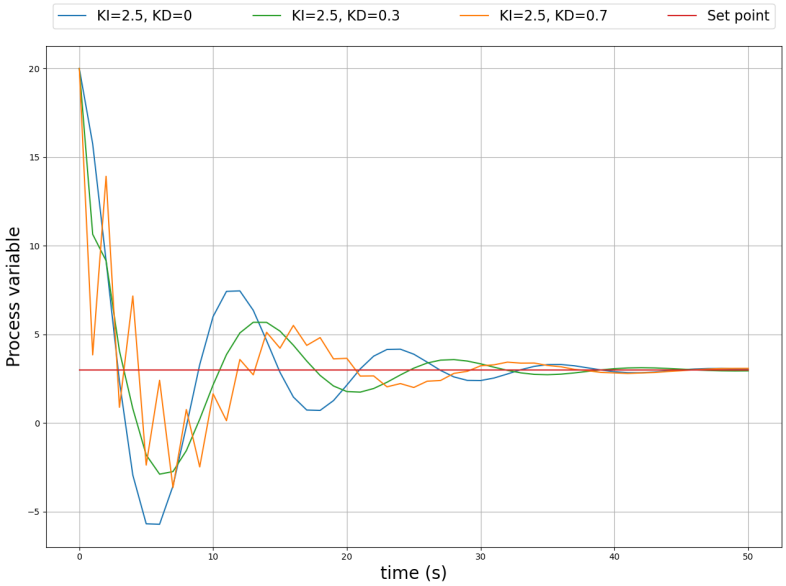

Figure 3.8: Id-controller with different derivative coefficients in a first-order system.

3.5 Implementation

The model is implemented as an object in Python. The code is made using NumPy and native Python for all calculations and matplotlib for plotting figures and is included in Appendix A.

There are several benefits to using object-oriented programming for this problem. By creating a general tool, multiple rockets can be simulated by making multiple instances with different input variables. Object-oriented programming also makes for an easily readable code and a simple hierarchy by designing the model to be modular, as shown in Figure 3.9.

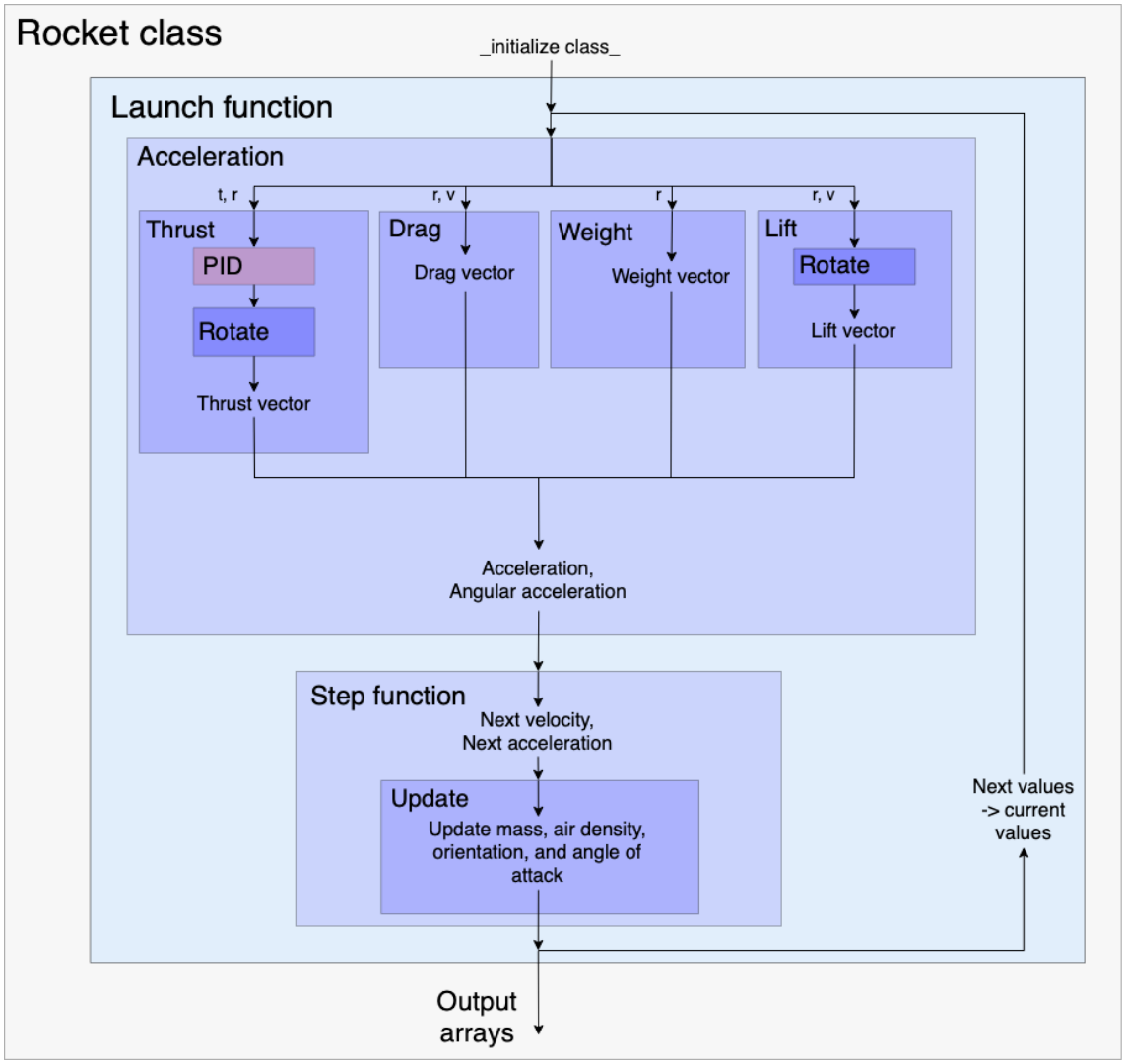

Figure 3.9: Code hierarchy. The red box is with the inclusion of TVC.

When called, the "launch" function simulates the rocket trajectory by calling the acceleration and step functions a set number of times determined by the length of the simulation and the time step, which are both defined by the user. This part of the simulation is the initialization of the numerical integrator that solves the ODEs in Equations 3.10 - 3.11.

The simulation advances by calling on four functions that implement the forces from Equations 3.5-3.8 and add their contributions in the "acceleration" function, which calculate the acceleration and angular acceleration for the next time step. These values, in turn, are used as input in the "step" function that solves the ODEs to find the velocity and positional vectors and finally calls the "update" function, which updates both the mass and atmospheric pressure and calculates the angle of attack that are all used in the next iteration. During the first few seconds, the rocket is still attached to the launch rail, and only the forces with a component parallel to the launch angle are in effect. The forces affect the rocket as usual for the remainder of the flight.

The TVC is implemented as a subclass changing the angle of the thrust vector in the "thrust" function according to the results from the PID. Instead of being represented as a vector toward positive z in the local reference frame, the force from thrust also has x-and y components in the local coordinates found using the trigonometric functions on the pitch and yaw angles from the PID. The PID is implemented as a function with the current position as input, returning the thrust angle. The PID coefficients, along with the maximum TVC angle, are defined by the user. The "PID" function calculates the error as the difference between the pitch and yaw angle of the rocket and the desired pitch and yaw angles, which defaults to the launch angle. The function then calculates the two angles' proportional, integral, and derivative terms in parallel and adds the three terms to find the required gain in each direction.

The function for each force utilizes both the global and the local frame of reference. All torques are calculated in a local frame of reference, but the force is typically easier to represent in a global frame of reference. The "rotate" function handles the conversion between these frames of reference and is used when calculating all the forces.

The force from weight points toward negative z in a global frame of reference during free flight, but while the rocket is mounted to the launch rail, it behaves differently. The vector in a global frame of reference is converted to the local frame of reference to find the component parallel to the launch rail, which is then rotated back. This results in a vector in the global frame of reference pointing downwards along the launch rail.

As the TVC calculates the optimal direction of thrust from the rocket's perspective, it is best to define it as a force toward positive z even without the TVC and then rotate it to the global frame of reference. With TVC, the vector is first rotated in the local frame of reference before converting it. However, the torque generated from angling the thrust vector is calculated directly from the local frame of reference's x- and y components.

As the torque generated from the drag force is modeled only as the lift in this simulation, this force only contributes to the movement modeled in the global frame of reference. Its direction is found by inverting the velocity vector in the global frame of reference.

The direction of the lift force is found by the cross-product of the direction of the relative velocity and the result of the cross-product of the rocket's orientation, and the direction of the relative velocity in a global frame of reference. In contrast, other software typically utilizes "quaternions," a 3D representation of complex numbers, to rotate vectors. The double cross-product improves the readability by being a more intuitive approach to the rotation. The vector is then scaled to a unit vector and multiplied by the magnitude of the lift force. The torque from the lift is calculated from the x- and y components in the local frame of reference. When the rocket is mounted to the launch rail, the lift force is translated to a local frame of reference to find the component parallel to the launch rail. While the rocket stays mounted, the angle of attack is determined only by the direction of the wind.

Tuning the PID

When tuning the PID, it is crucial to understand how the system behaves and how each term affects the final gain. A stable sounding rocket will always point toward the direction of relative motion. In windy conditions, the lift force constantly tries to point the rocket toward the direction of the wind and to be able to control the trajectory. The controller must be able to compensate for this force. As a safety precaution, oscillations must be avoided. Some overshoot can be allowed, but the settling time should be as short as possible to avoid unnecessary fuel consumption. The TVC has some limitations on the output response, which require a way to handle integrator windup. In this simulation, the user defines the system's constraints by setting a maximum angle of the thrust vector.

The actual tuning of the PID is done by adjusting the P-, I-, and D-coefficients. There are numerous methods of approaching this. The representation of the rocket trajectory made in this thesis is a second-order system: requiring double integration of the acceleration. In the S-domain, the double integral corresponds to having two poles, and the PID controller introduces another pole in origin and two zeros defined by the coefficients. The coefficients can then be adjusted to move the zeros to get the required response from the system. Another approach is to analytically determine the coefficients using defined methods such as the Cohen-Coon or Ziegler-Nichols tuning methods. These methods all work for most continuous models, but with a discrete-time numerical integration, it is easier to adjust the coefficients manually and compare the results.

The first step is to determine the sign of the gain. The error is defined as positive when the rocket's elevation from the horizon is lower than the desired elevation and negative when the elevation is larger than desired. This means the gain should be positive, as shown in Figure 3.10.

The next step is to find the order of magnitude in the proportional term and then for the integral term, shown in Figures 3.11-3.12. Some oscillations with the integral term are expected as long as the general trend of the output improves when increasing the order of magnitude of the coefficient.

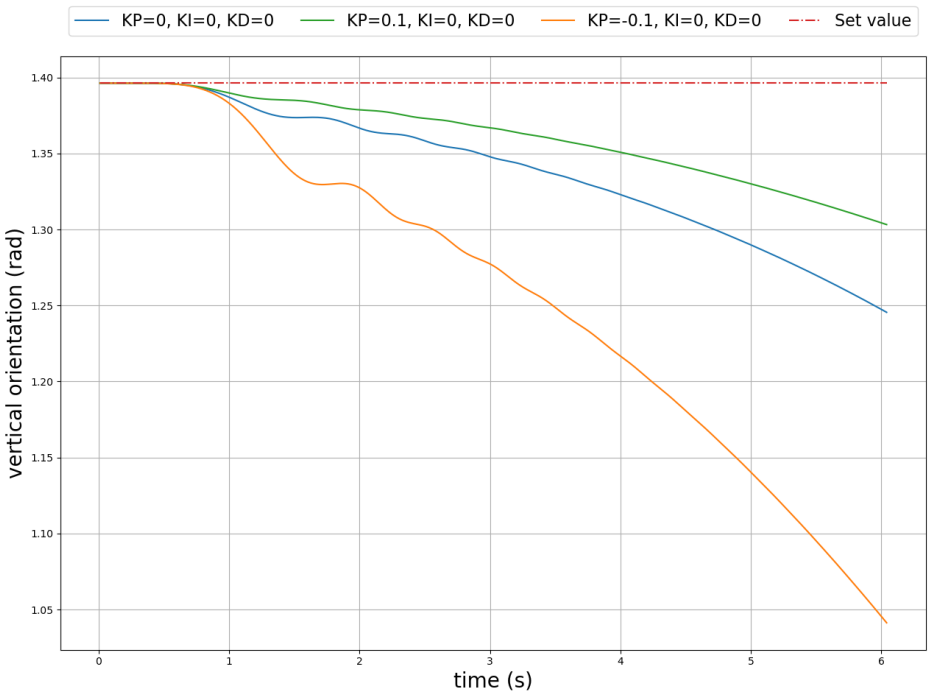

Figure 3.10: Determining sign of the gain.

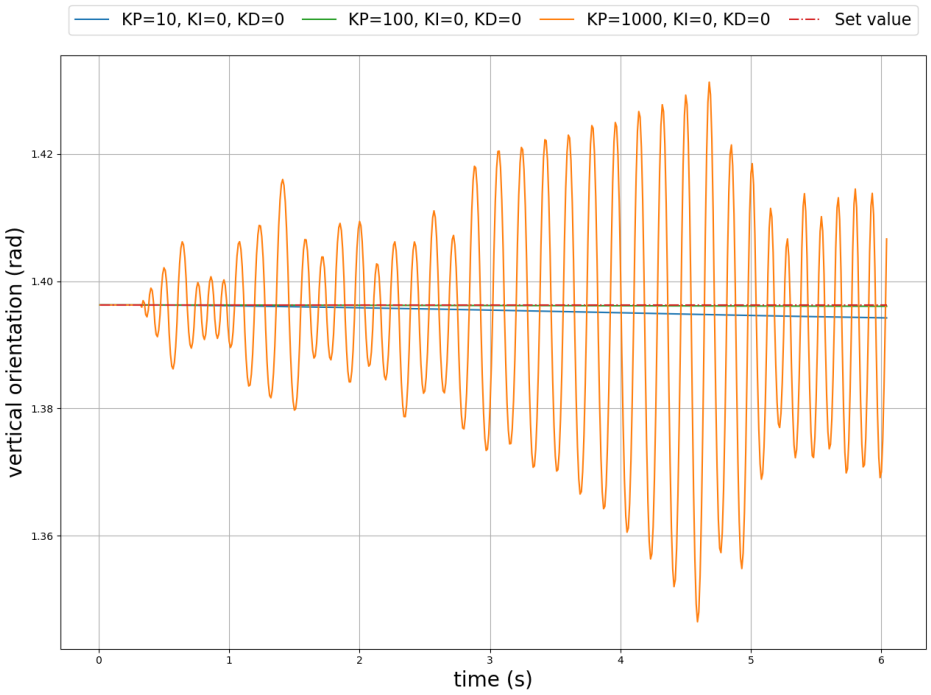

Figure 3.11: Determining the order of magnitude of the P-term.

The oscillations are reduced by finding the correct order of magnitude of the derivative term, as shown in Figure 3.13. The final values are found by tweaking the coefficients based on the knowledge of how they influence the system, as shown in Figure 3.14.

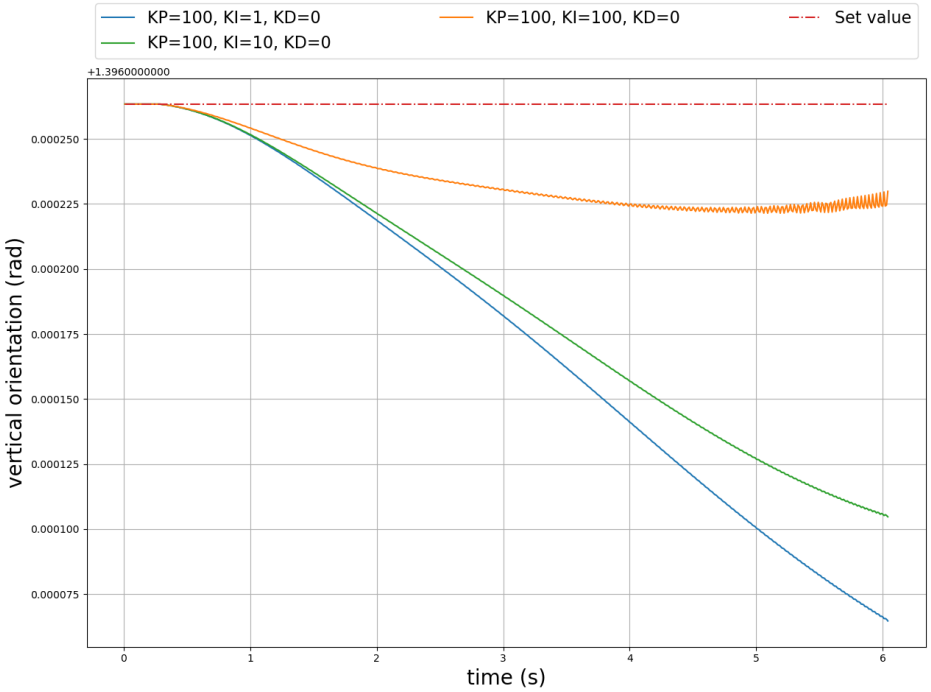

Figure 3.12: Determining the order of magnitude of the I-term.

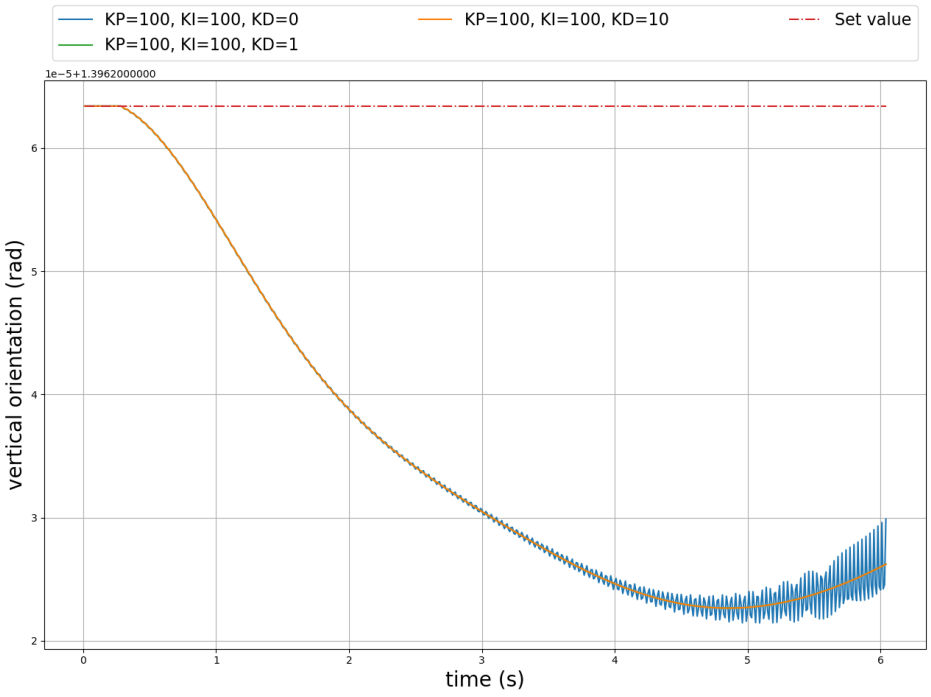

Figure 3.13: Determining the order of magnitude of the D-term.

Although there is still some steady state error, further increases in both the I-term and D-term would cause uncontrollable oscillations. The finally-tuned PID controller error is $1,55 \cdot 10^{-5}$ rad.

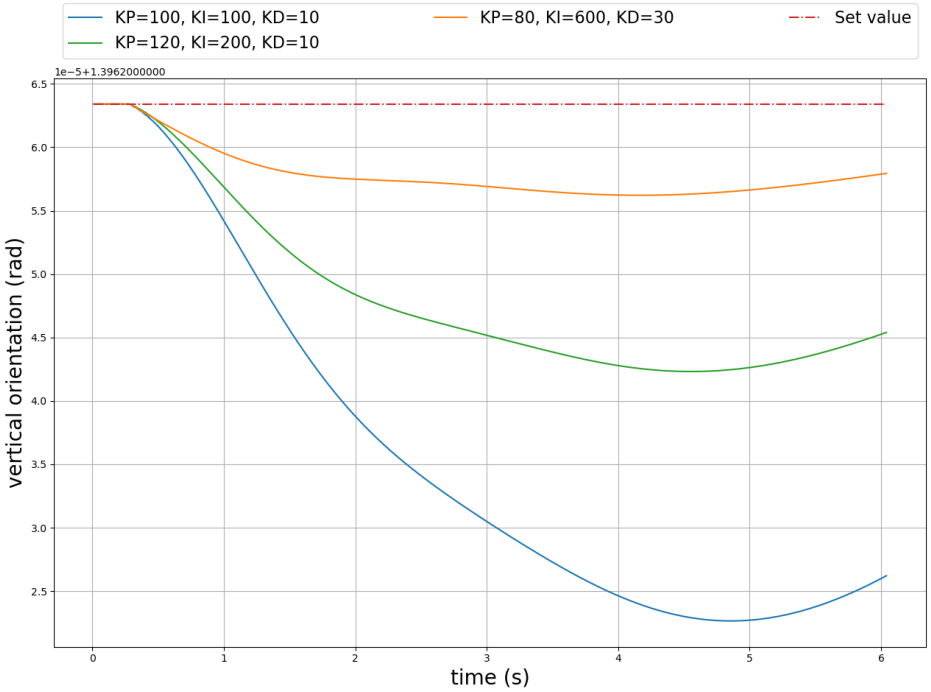

Figure 3.14: Final tuning of the system.

3.6 Parameters

All the functions in Equation 3.4 rely on one or more predefined constants. Some of these are given by the rocket or the engine used. These are used as input variables when initializing the class in Python. Table 3.1 lists all other constants used in the simulation.

Constant	**Value**
Gravitational constant	$9.806\,65\,\mathrm{m/s^2}$
Atmospheric pressure at sea level	$101\,325\,\mathrm{Pa}$
The absolute temperature at sea level	$288.15\,\mathrm{K}$
Temperature lapse rate	$0.0065\,\mathrm{K/m}$
Ideal gas constant	$8.314463\,\mathrm{J/mol\,k}$
The molar mass of dry air	$0.0289654\,\mathrm{kg/mol}$

Table 3.1: Table of constants used.

The initial simulations with the model use the same parameters as the student rocket on "CaNoRock," a student rocket program collaborating with UiO (UiO, 2015), with a Pro98 engine to provide a realistic baseline with readily available parameters. The same model can be used on any one-stage rocket or with custom parameters. To later compare the results from the simulation with OpenRocket in chapters 4 and 5, the parameters are changed to match the available parameters in that software.

OpenRocket is open-source, accessible, and easy-to-use software and is the go-to software for model rocketry. OpenRocket was initially developed as a master's thesis (Niskanen, 2009) and has since been maintained and improved by a large community (OpenRocket, 2023).

All parameters used are listed in Table 3.2. Some parameters are not stated in the manual (Andøya Space Education, 2023) or on the producers' website (Cesaroni Technology Inc., 2023) and must be estimated further. These estimations are based on similar data from OpenRocket to compare the results and can be found in Table 3.3.

Parameter	**Value from student rocket**	**Updated value**
Dry mass	10.058 kg	9.85 kg
Wet mass	18.554 kg	18.554 kg
Length	2.708 m	2.71 m
Radius	0.0513 m	0.515 m
Avg. thrust	2501.8 N	2529 N
Engine burn time	6.09 s	6.04 s

Table 3.2: Table of parameters gathered from the student rocket (Andøya Space Education, 2023; Cesaroni Technology Inc., 2023), and the updated parameters used in the simulations.

The center of pressure should be around one to two diameters behind the center of gravity to keep the rocket stable. This is done by modifying the placement of mass and the size and shape of the tail fins in OpenRocket. The drag and lift coefficients are the averages of the coefficients calculated by the software after the rocket leaves the launch rail. The critical lift angle and the launch rail length are guesstimated to what seem like reasonable values. In addition, the maximum time, launch angle, ballistic wind speed and direction, and PID coefficients are provided as inputs by the user and vary between simulations. These are stated along with the results of the simulations in chapter 4.

Parameter	**Value**
Drag coefficient (C_D)	0.75
Lift coefficient (C_L)	0.15
Center of mass	0.710 m
Center of pressure	0.510 m
Critical angle of lift	20 deg
Length of launch rail	5 m

Table 3.3: Table of estimated parameters

3.7 Testing and verification

The model itself must be verified to make conclusions from the simulation data. The verification is done through comparison with existing software. Verification using other software alone will not accurately represent the deviation between the simulation results and data gathered from an actual launch. Ideally, the model used in this thesis should be tested against a real-life launch, both with and without TVC. However, a comparison with existing software can be used to indicate realism in the model.

Although sounding rocket missions require more advanced and precise numerical models in their actual planning, OpenRocket is still more accurate than this model aims to be. The simplicity and accessibility of OpenRocket, therefore, make it the prime candidate for initial verification.

The verification is done by simulating the same rocket with the same parameters in both OpenRocket and the Python model designed in this thesis and comparing the results with different initial conditions. In addition, the simulations with OpenRocket are both with and without spin. The spin results from a 3 deg "cant" on the tail fins, the angle of the fins relative to the longitudinal axis of the rocket. The parameters used are listed in Tables 3.2 and 3.3.

The "simulator options" chosen in OpenRocket are "flat earth" geodetic calculations using Runge-Kutta with a time step of 0.01 s. Launch conditions for the various simulations are an elevation of 80 degrees from the horizon and an azimuth of 90 degrees from North to East. All from around Andøya Rocket Range. An actual launch from Andøya would be towards the ocean to the west, but in this simulation, the only difference between launching East or West is the sign of the lateral distance. Ideally, multiple launch angles should be simulated to compare where the rocket lands in different conditions. However, in this thesis, the trajectories are compared and discussed directly without varying the launch angle. The launch rail is set to 5 m in both software, and the atmospheric conditions are the International Standard Atmosphere at ground level. The comparisons vary the average wind speed and direction, with the turbulence and standard deviation set to zero. The rocket design and the thrust curve for the rocket engine are shown in Figure 3.15. Here, the red line is the actual thrust curve of the engine, and the green line is the thrust curve used in all simulations.

This setup has comparable parameters to the model constructed in this thesis but with a different numerical solver and a different model of the atmospheric conditions and the forces involved. The results should be comparable but not matching.

As the validation of the model is an abstract concept, no precise metrics can alone determine if the model is valid. The same is true when exploring how sounding rockets with TVC behave during launch. Instead, a variety of parameters must be examined and discussed.

Whether the rocket hits a predetermined target or area of interest can be determined by the location of a specific point along its trajectory. Since the rockets in these simulations do not have any specific target, the location in all three axes can be plotted against the time since launch, and the resulting trajectories can be compared. This can show how the different systems behave but cannot explain any eventual differences.

As the set target of the PID controller is the rocket's pitch and yaw angles, these parameters directly affect the TVC and should be examined. However, since the implementation of angles in Python differs from OpenRocket, the vertical orientation is used instead. The vertical orientation is the same in both models and contains the same information as the pitch and yaw angles as long as the horizontal orientation stays constant. As the horizontal orientation in this model only changes during crosswind, this is an acceptable constraint.

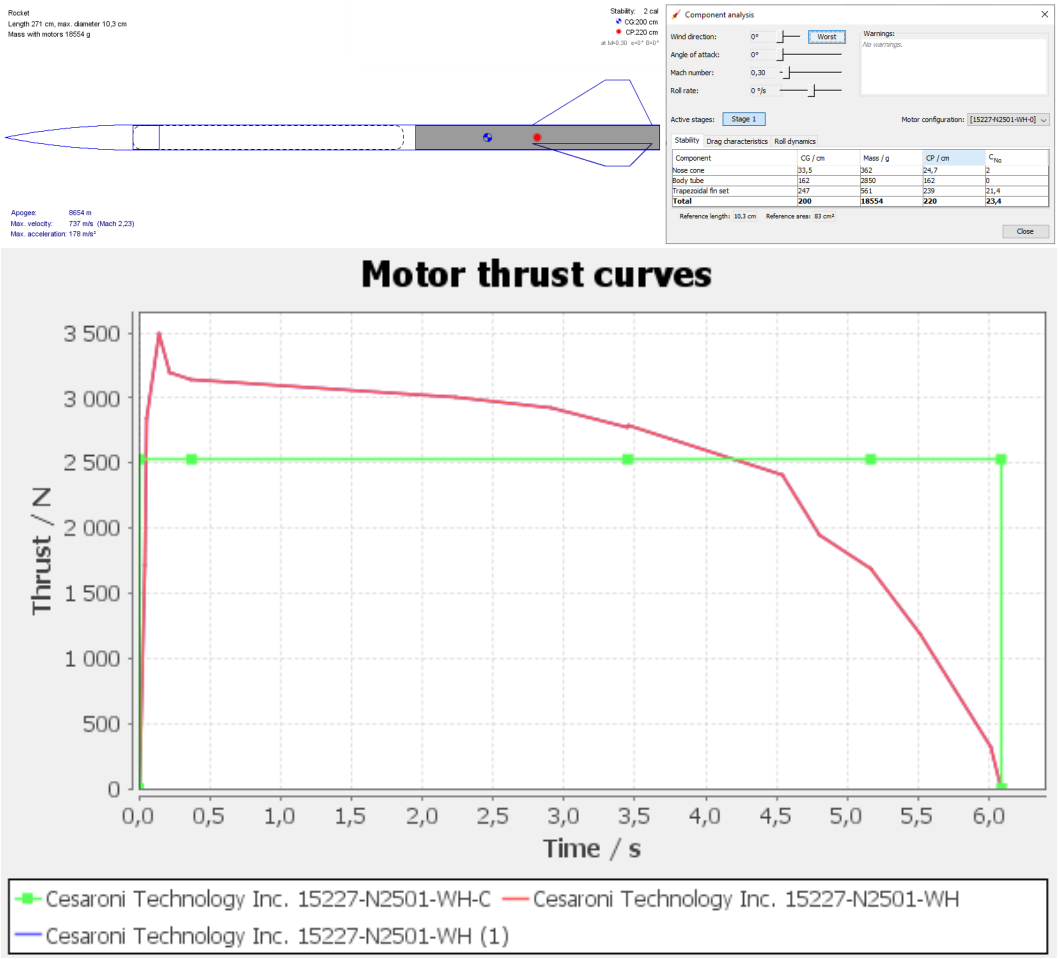

Figure 3.15: Rocket design and thrust curve in OpenRocket.

Finally, the velocity and acceleration of the rockets plotted against the time since launch can highlight when any differences appear. These parameters, combined with the knowledge of how the model is built, provide a good overview of the simulations for further discussion in chapter 5.

Chapter 4

Results

The first section of this chapter compares the simulations from the tool developed in this thesis without TVC and simulations in OpenRocket. The last section presents simulation results with and without TVC in the developed model. These results are later used in chapter 5 to discuss the model's validity and effects of TVC during the launch of sounding rockets.

4.1 Validation of the model

The model designed in this thesis is validated by comparing the result from simulations in Python and OpenRocket. Table 4.1 shows an overview of all the wind conditions simulated in this section. The exact wind speeds correlate to the upper end of numbers 2, 4, and 6 on the Beaufort scale (Met Office, 2023). This clearly shows the effect of increasing wind speed, and any stronger winds would result in all simulations experiencing a tumble during launch. Tumble is an uncontrolled rotation and indicates that the conditions are too harsh for a safe launch. Only the simulation with a wind speed of 8 m/s is done in tailwind and crosswind to limit the number of figures. At this wind speed, the effect of wind direction is visible.

Figure	**Wind speed**	**Wind direction**
Figure 4.1	0 m/s	E (headwind)
Figure 4.2	3 m/s	E (headwind)
Figure 4.3	8 m/s	E (headwind)
Figure 4.4	14 m/s	E (headwind)
Figure 4.5	8 m/s	N (crosswind)
Figure 4.6	8 m/s	W (tailwind)

Table 4.1: Table of figures and wind parameters used to validate the Python model.

The top left plot in all figures shows the rocket's trajectory, and the top right plot shows the absolute velocity and acceleration. The lower left plot of the figures shows the vertical orientation during flight, and the lower right plot shows the vertical orientation during burn time.

Most notable is the difference in the distance along the East axis between the different models, shown in blue in the top left plot in Figures 4.1-4.6. The python

simulations have an increased angular velocity during the burn time, resulting in a lower vertical orientation in the early stages of flight, as shown in the lower two plots of the figures. As shown in Figure 3.15, the thrust curve is equal between all simulations, but the direction of thrust changes with the orientation. This change in thrust significantly affects lateral movement, which also influences altitude.

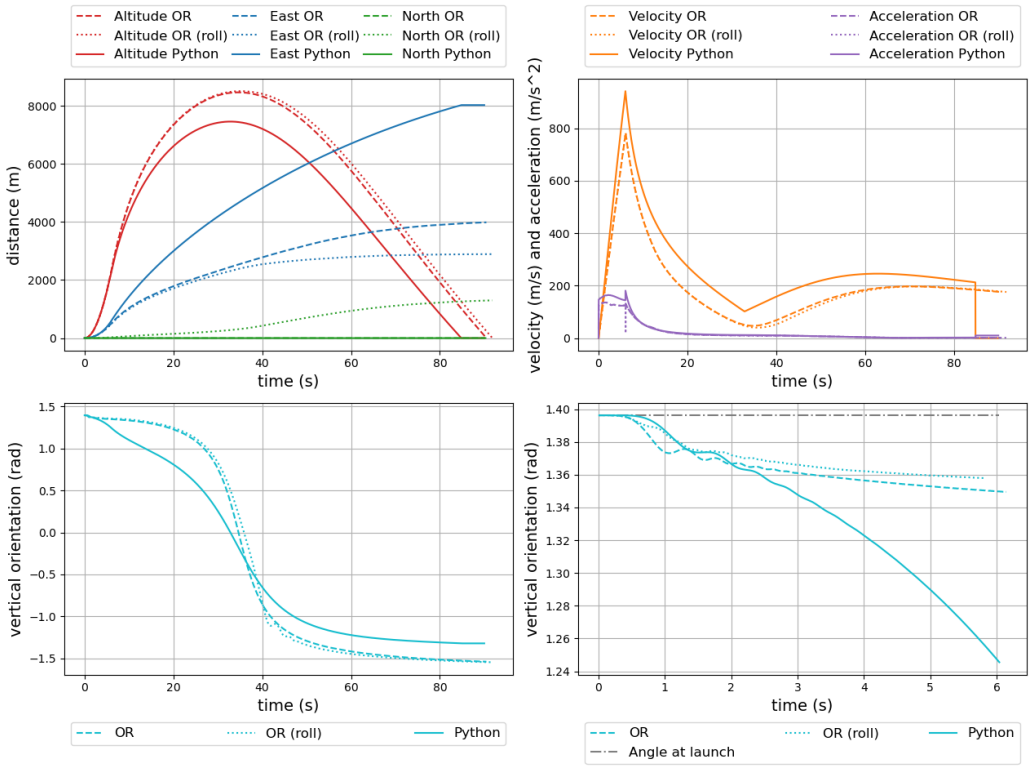

Figure 4.1: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python without wind.

The oscillations in vertical orientation during burn time are about the same as in the simulations in OpenRocket without spin, with a comparable frequency and amplitude. This frequency stays relatively constant with increasing wind speed, but the amplitude of the oscillations increases. In OpenRocket with spin, this oscillation has some destructive interference reducing its amplitude. These oscillations are likely due to the aerodynamics needing some time to stabilize. In Figure 4.4, the simulation with the spin-stabilized rocket aborts due to too large an angle of attack, and both simulations in OpenRocket experience tumble during launch.

Though the spin in OpenRocket reduces the movement along the East axis, it results in a difference along the North axis, meaning the difference in total lateral distance is less than the difference along the East axis. Without wind, the difference in the total lateral distance with and without spin is about the same. The model in OpenRocket with spin is more affected by increasing wind, although the distance East is less affected.

Although there is a substantial difference in lateral distance between the models, the overall effect of increasing wind is similar. In 8 m/s headwind, the Python simulation has a difference of 685 m in lateral distance from the initial simulation without wind. In OpenRocket without spin, the same difference is 687 m.

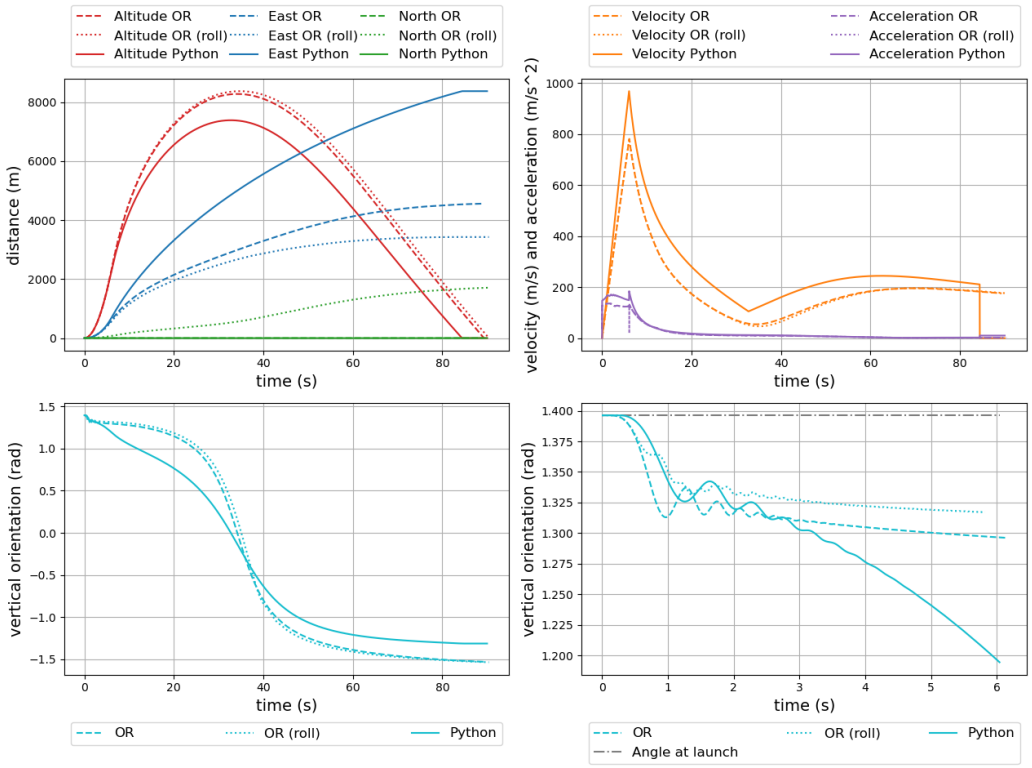

Figure 4.2: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python with 3 m/s headwind.

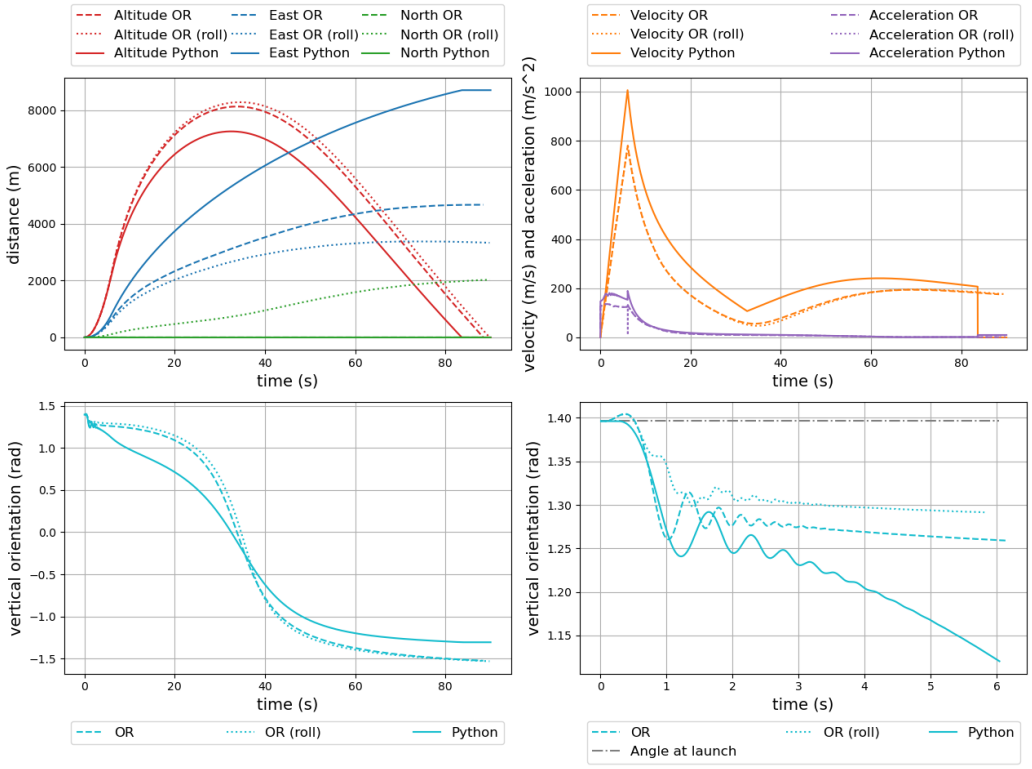

Figure 4.3: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python with 8 m/s headwind.

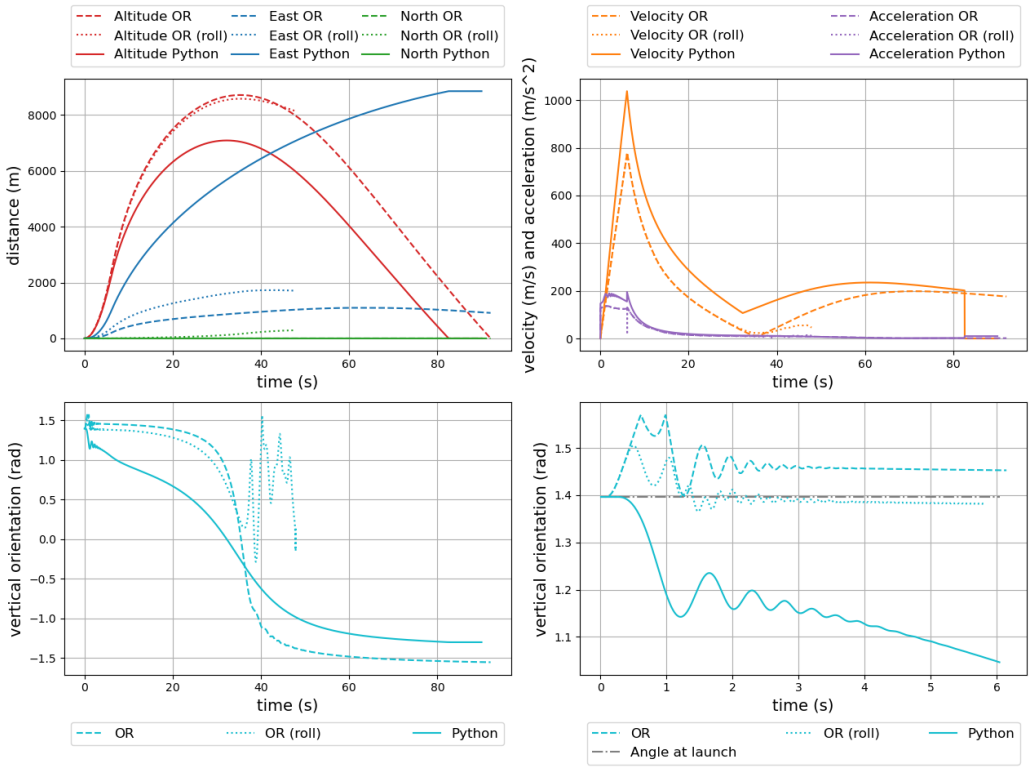

Figure 4.4: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python with 14 m/s headwind.

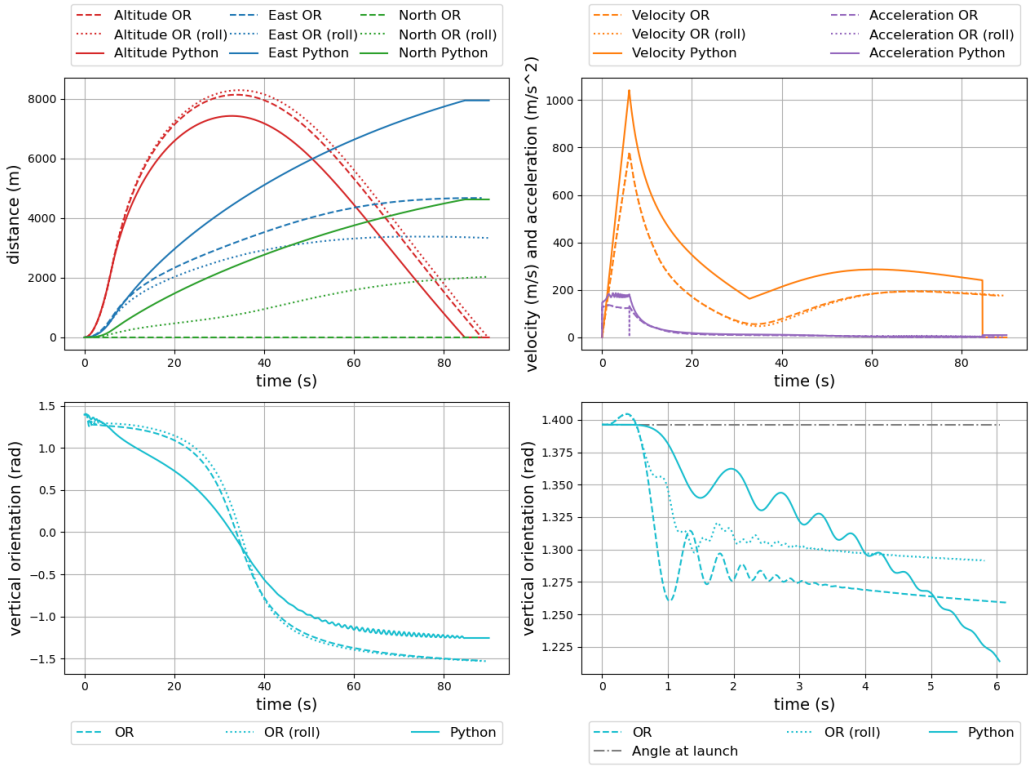

Figure 4.5: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python with 8 m/s crosswind.

The top right plot in the figures shows the difference in absolute velocity and absolute acceleration. The simulations in OpenRocket have a dip in acceleration at apogee, while the simulations in Python have a small spike around the same time. This results in a difference in velocity during the recovery stage of the flight.

OpenRocket also changes drag- and lift coefficients, the center of pressure, the center of pressure, and the gravitational constant during the flight. There are also some differences between the models' mass moment of inertia and air pressure. The impacts of these factors are discussed in chapter 5.

With tailwind and crosswind, the differences between the simulations are further pronounced. In Figure 4.5, the vertical orientation in the python simulation starts with a smaller error than in OpenRocket but does not stabilize in the same way. Although not shown in the figures, the azimuth angle seems to change the same way as the vertical orientation, which leads to a significant distance in the North-axis in the top left plot of the figure. In OpenRocket without spin, there is no movement North with this wind condition. Figure 4.6 shows both the Python simulation and the simulation in OpenRocket with spin tumble during launch.

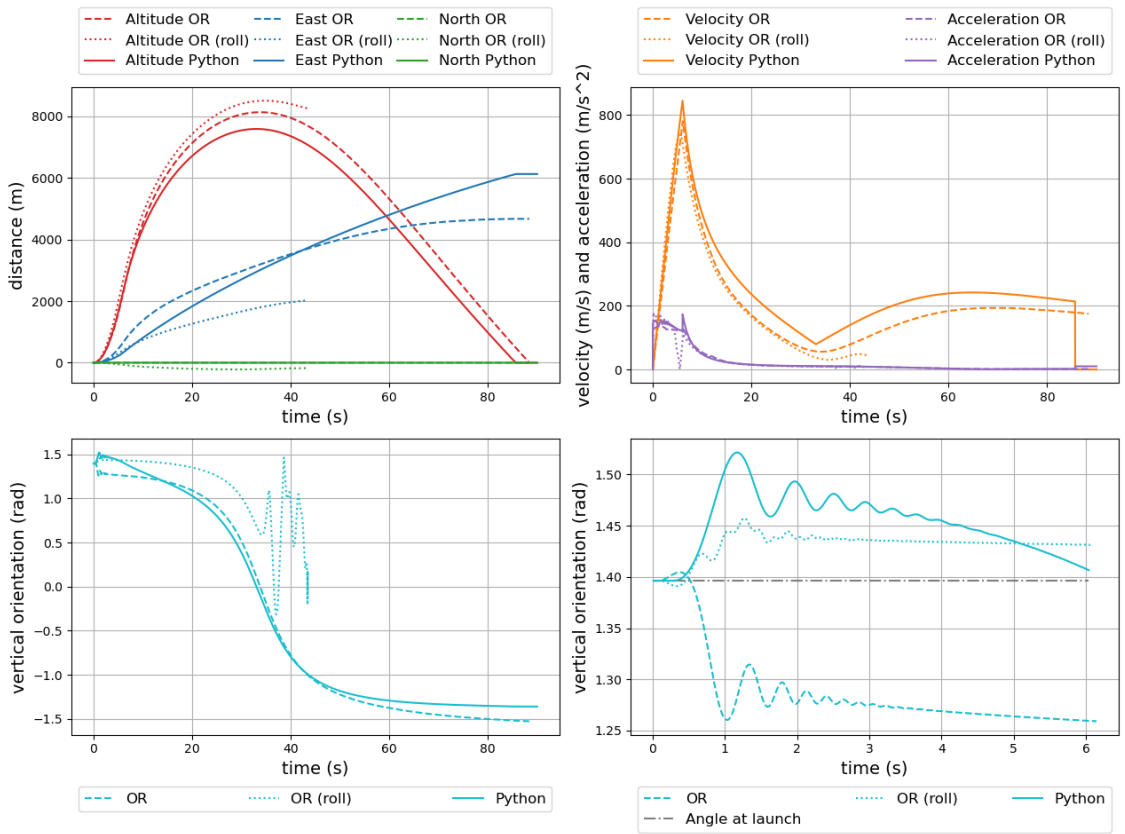

Figure 4.6: Comparison of the trajectory, velocity, acceleration, and vertical orientation between OpenRocket (OR) and Python with 8 m/s tailwind.

4.2 Results with TVC

To show the effects of TVC, the results from the simulations are compared to simulations without TVC in different wind conditions. In addition to comparing the model in the same conditions used to validate the model, the model with TVC is tested in harsher conditions. These wind speeds are chosen to match Beaufort numbers 7, 8, and 10 (Met Office, 2023). Table 4.2 shows an overview of all the wind conditions simulated in this section.

Figure	**Wind speed**	**Wind direction**
Figure 4.7	0 m/s	E (headwind)
Figure 4.8	3 m/s	E (headwind)
Figure 4.9	8 m/s	E (headwind)
Figure 4.10	14 m/s	E (headwind)
Figure 4.11	8 m/s	N (crosswind)
Figure 4.12	8 m/s	W (tailwind)
Figure 4.14	17 m/s, 21 m/s and 25 m/s	E (headwind)
Figure 4.15	17 m/s, 21 m/s and 25 m/s	W (tailwind)
Figure 4.16	17 m/s, 21 m/s and 25 m/s	N (crosswind)
Figure 4.13	3 m/s	E (headwind)

Table 4.2: Table of figures and wind parameters used to determine the effects of TVC.

The top left plot in Figures 4.7-4.12 shows the rocket's trajectory, and the top right plot shows the absolute velocity and absolute acceleration. In Figures 4.14-4.16, the trajectory is enlargened, but the velocity and acceleration are not shown. The lower left plot of all the figures shows the vertical orientation during flight, and the lower right plot shows the vertical orientation during burn time. Figure 4.13 shows the distribution of forces with and without TVC.

Without TVC, the distance along the East axis increases with an increasing headwind. This is because the aerodynamics angle the rocket towards the wind. As the wind is constant with altitude and has zero elevation, the rocket reduces its vertical orientation more rapidly with an increasing headwind, as shown in the lower right plot of Figures 4.7-4.10. With a lower vertical orientation of the rocket, the thrust vector also has a lower vertical orientation, resulting in a more significant movement along the East axis and a lower maximum altitude.

In the simulation with TVC, the opposite effect is observed. Although the aerodynamics tries to reduce the vertical orientation, the TVC pushes back during the burn time of the engine, and the thrust angle stays constant. A stronger wind increases the drag force during the rest of the flight, slowing the rocket. The lower left plot of the figures shows the total change in the vertical orientation. This difference is due to the rocket with TVC having a higher altitude and less movement along the East axis.

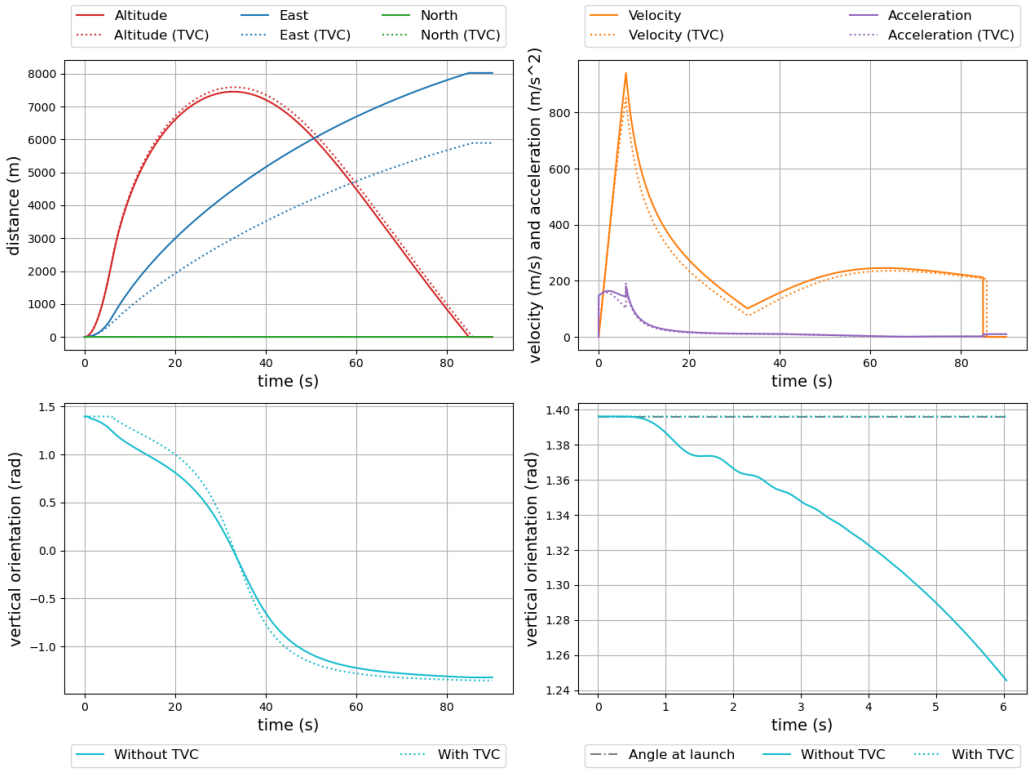

Figure 4.7: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC without wind.

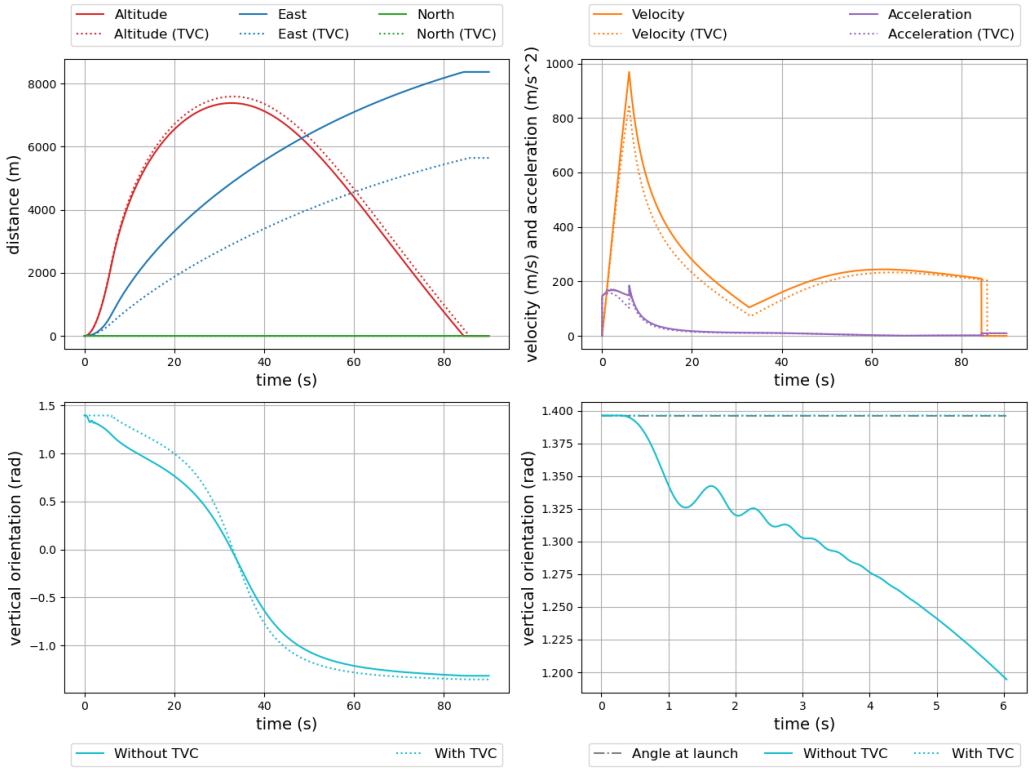

Figure 4.8: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC during 3 m/s headwind.

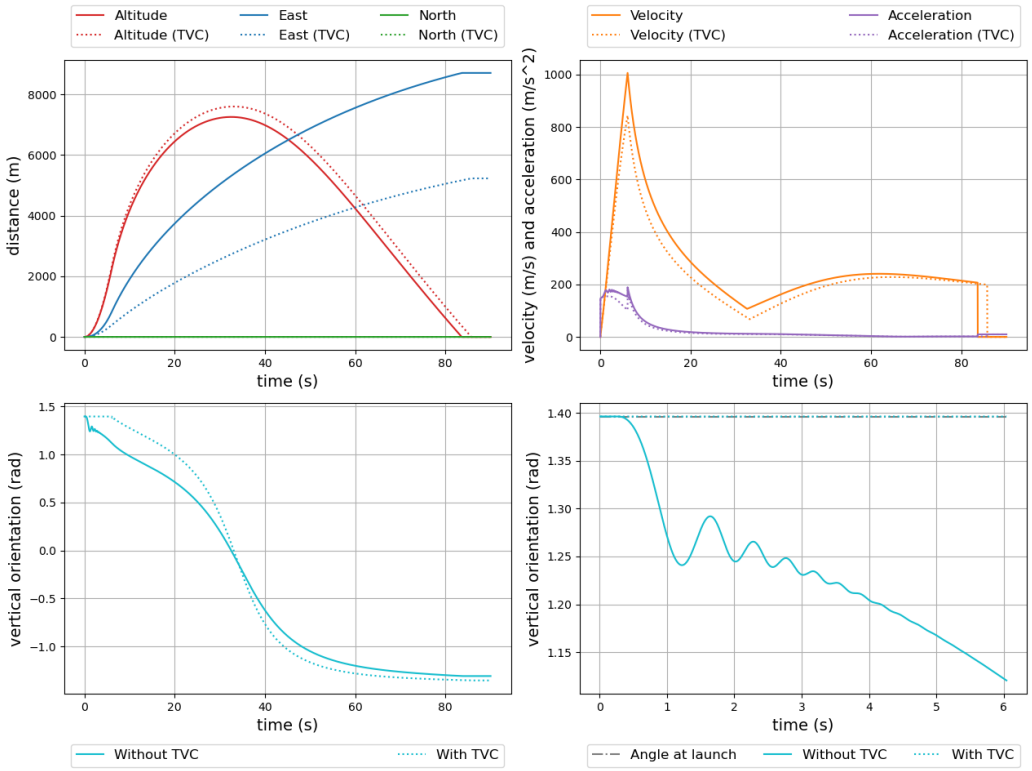

Figure 4.9: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC during 8 m/s headwind.

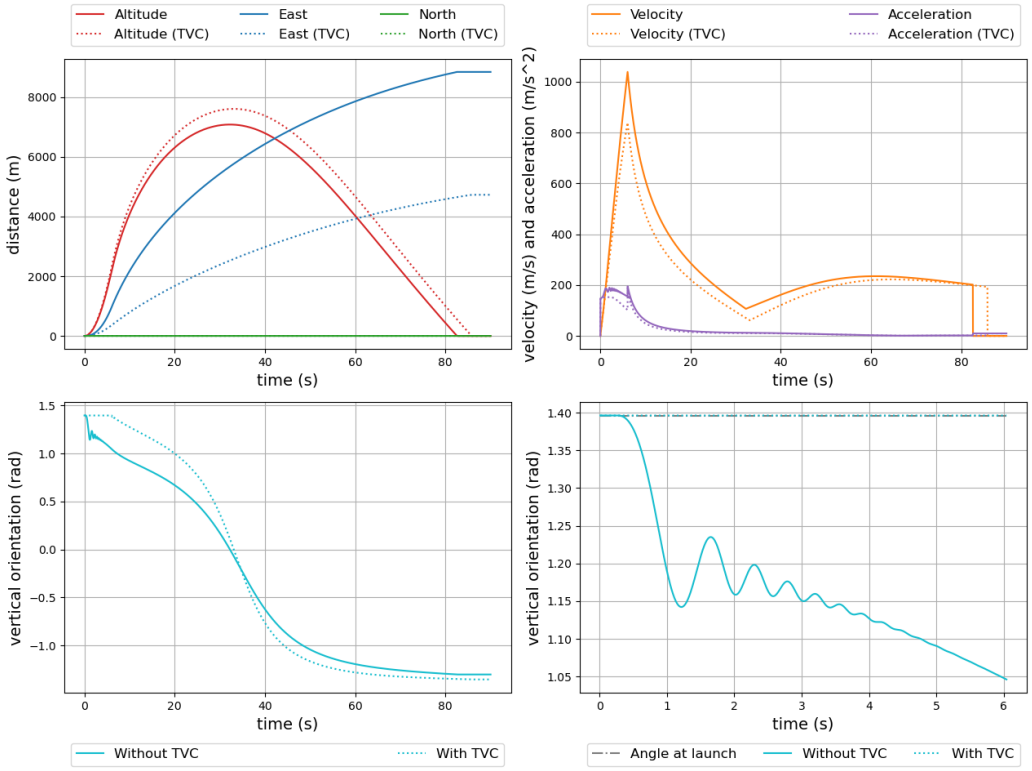

Figure 4.10: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC during 14 m/s headwind.

The top right plot in the figures shows a difference in velocity with and without TVC. Without TVC, the maximum velocity increases with an increasing headwind, while the simulation with TVC stays about the same. This is likely because the velocity is more evenly distributed along the East axis and in altitude without TVC, resulting in less overall drag. This also explains the slight difference in acceleration during the burn time.

Figure 4.11 shows the trajectory when the rocket experience crosswind. In addition to the differences mentioned, there is also some movement along the North axis due to the wind. Without TVC, the rocket turns toward the wind during the engine burn time to produce a thrust along that axis. With TVC, the rocket maintains its course during the burn time but is slowly pushed by the wind.

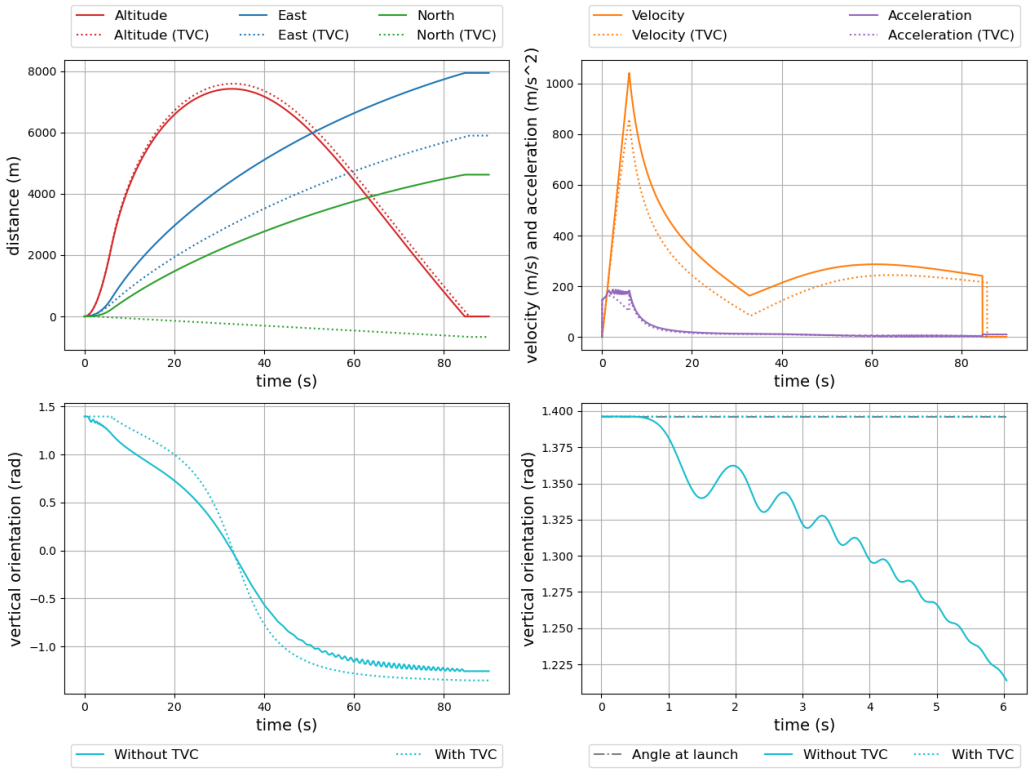

Figure 4.11: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC during 8 m/s crosswind.

The oscillations in the vertical orientation from about 50 seconds stop in the simulation with TVC. At this point, the engine has stopped firing, and the TVC is disconnected. The velocity is about 100 m/s slower than the simulation without TVC.

Figure 4.12 shows the trajectory in a tailwind. While the model without TVC tumbles during launch, the model with TVC can maintain its vertical orientation and thus avoid tumbling.

Figure 4.13 shows the distribution of forces. As the rocket does not change its vertical orientation during burn time, the horizontal fraction of the thrust affecting the rocket stays approximately constant with TVC. Instead, the "excess" force is directed vertically, resulting in a slightly higher altitude. While the rocket's orientation is

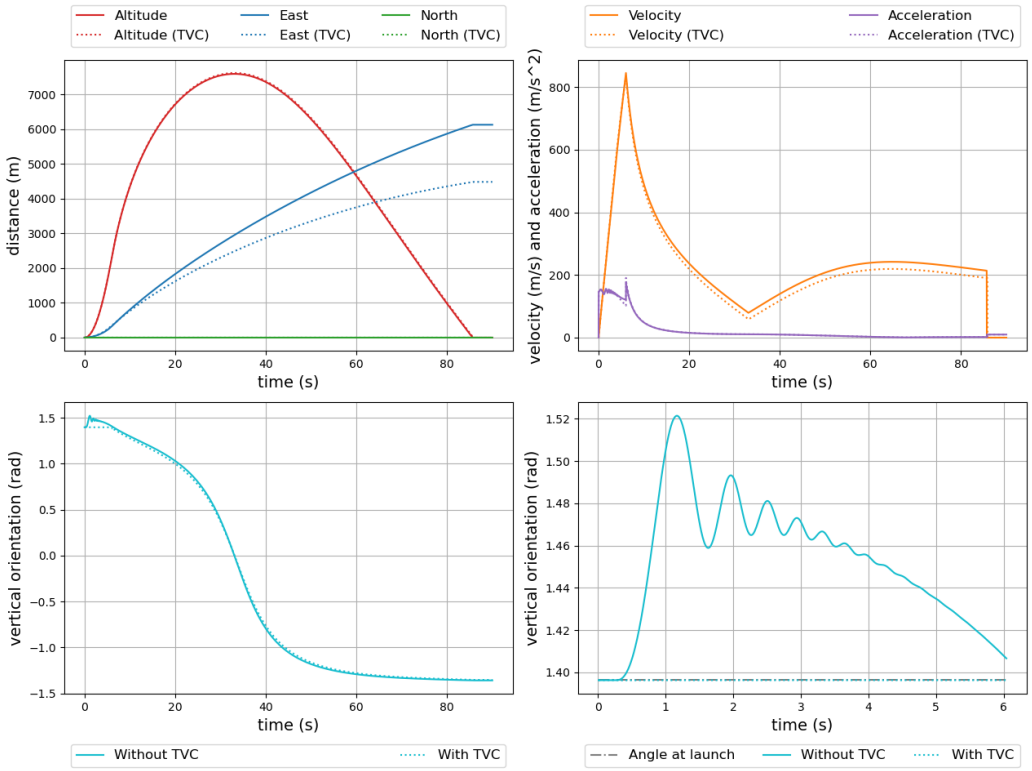

Figure 4.12: Shows the trajectory, velocity, acceleration, and vertical orientation in simulations with and without TVC during 8 m/s tailwind.

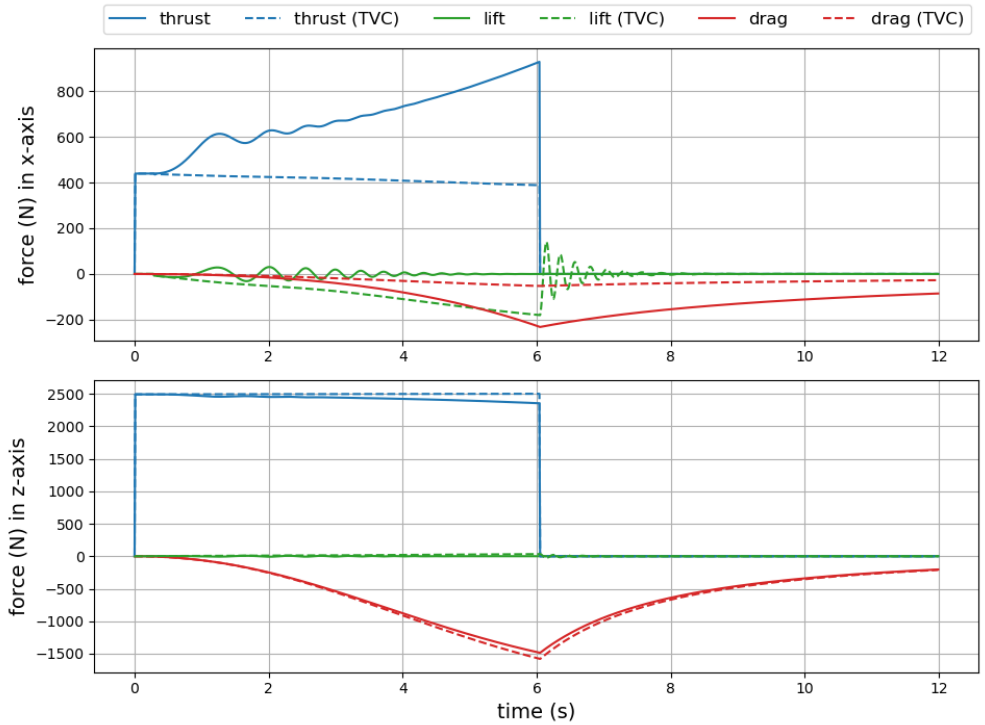

Figure 4.13: Shows the distribution of thrust-, drag- and lift force horizontally (x-axis) and vertically (z-axis) in simulations with and without TVC during 3 m/s headwind.

locked during the burn time, there is a slight build-up of lift force that releases when the engine stops firing. This causes slight oscillations after the burn time in the simulations with TVC.

Extreme cases

Figures 4.14-4.16 show simulations with TVC in conditions that would be unsafe for traditional sounding rockets. In a headwind, the rocket behaves as in calmer conditions but with less movement in the East axis due to the drag from increased relative velocity.

There are some oscillations in the vertical orientation with very high frequency during the first three seconds in strong wind, but still in the magnitude of $7.5 \cdot 10^{-5}$ rad at its peak. This is expected to decrease with on-site tuning or in simulations with less noise and smaller step size.

In a tailwind, the initial oscillations in vertical orientation have about half the amplitude as in a headwind, but an additional peak with a longer settling time, during the first second of flight. In these simulations, the lateral distance increases with increasing wind speed. In a crosswind, the movement in the East axis stays reasonably constant but with some drift along the North axis. This drift increases with increasing wind speed. There are also oscillations in vertical orientation during the burn time, but with a lower frequency than in headwind. The magnitude of this oscillation is in the order of $7.5 \cdot 10^{-6}$ rad at its peak.

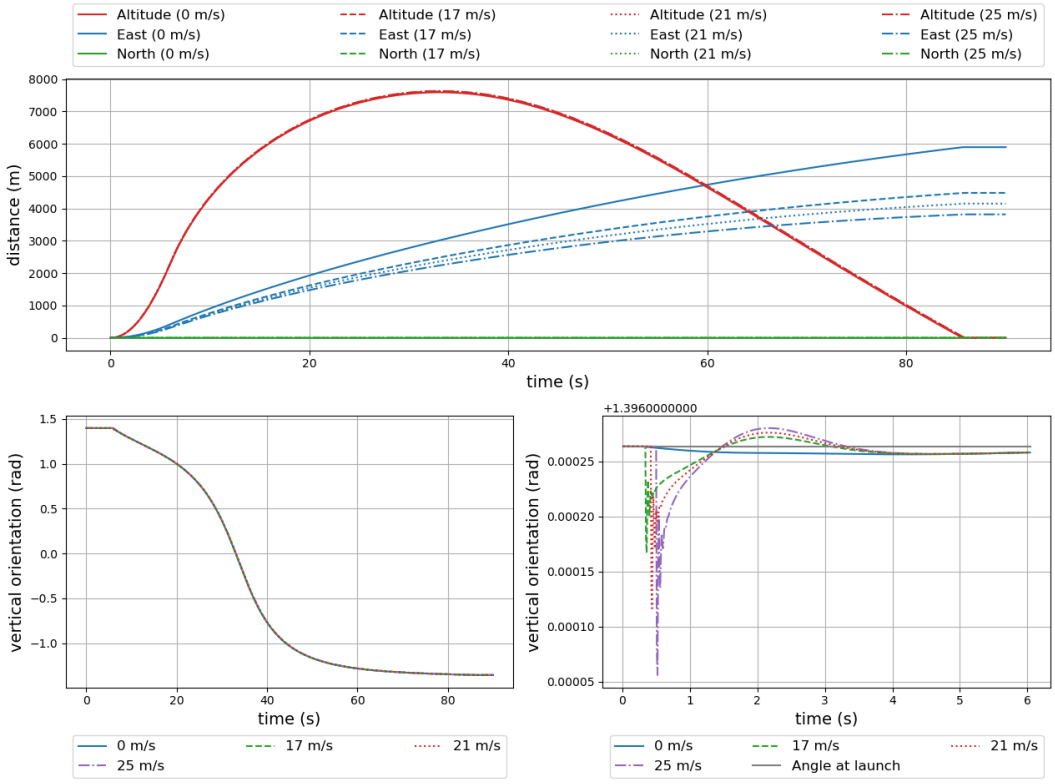

Figure 4.14: Shows the trajectory and the vertical orientation of simulations with TVC in powerful headwinds.

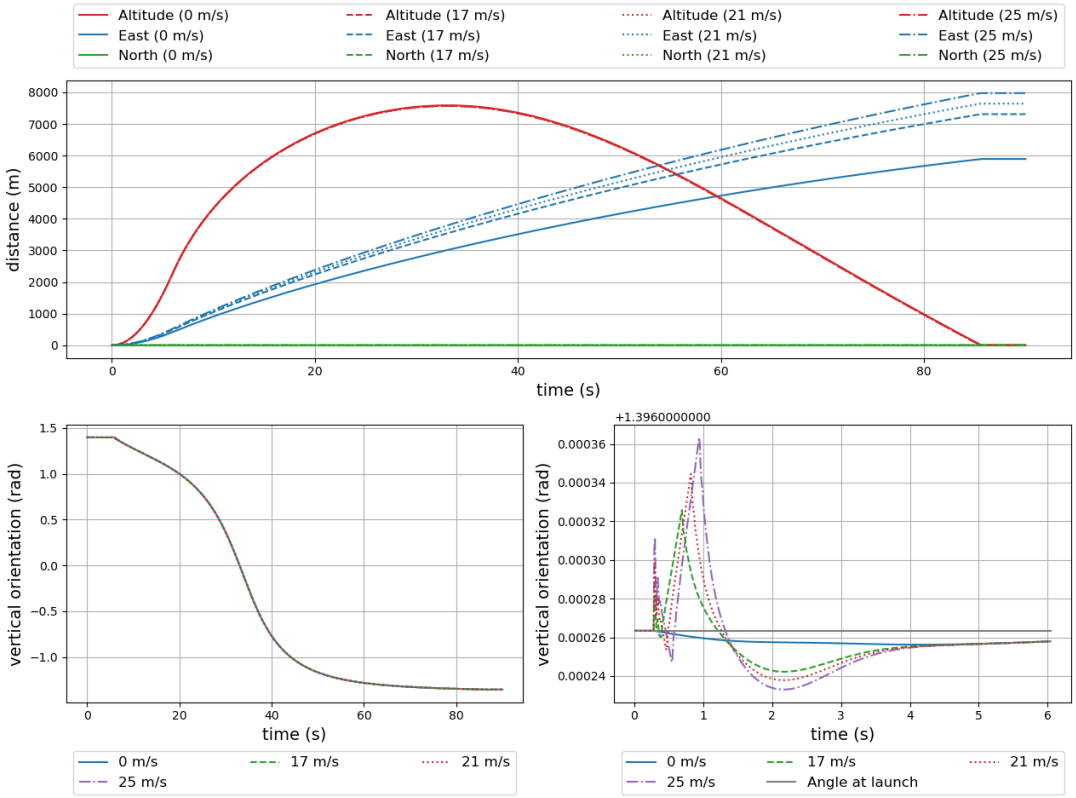

Figure 4.15: Shows the trajectory and the vertical orientation of simulations with TVC in powerful tailwinds.

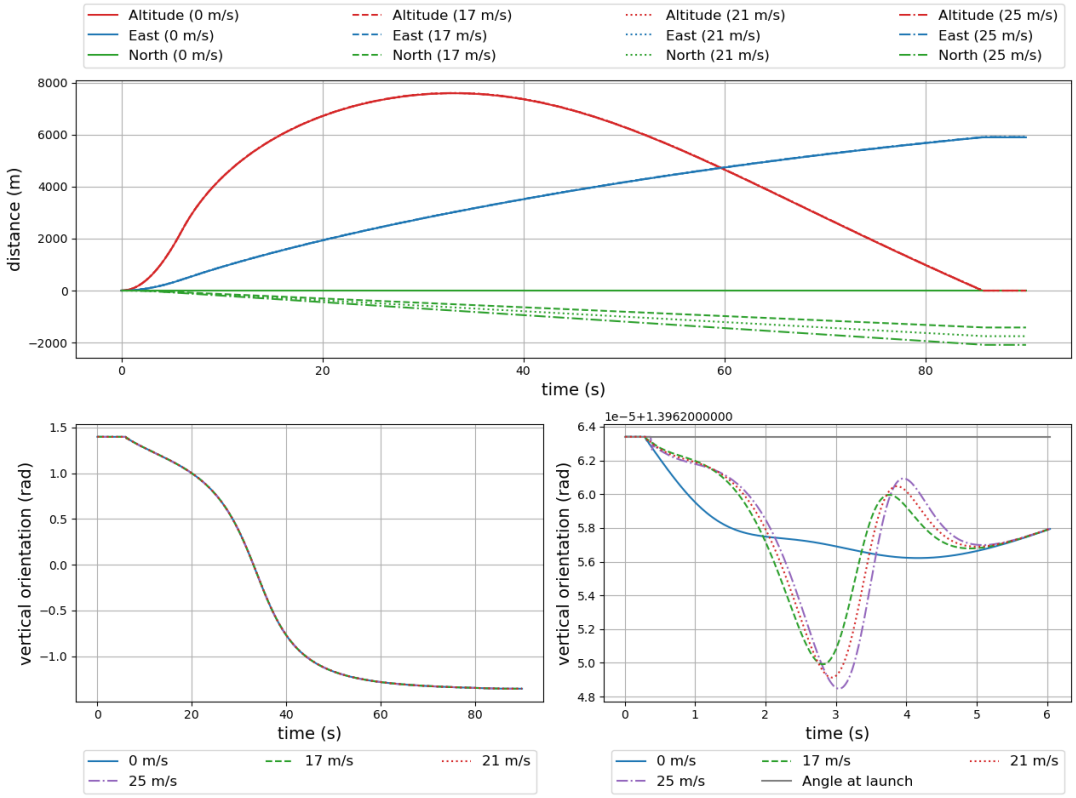

Figure 4.16: Shows the trajectory and the vertical orientation of simulations with TVC in powerful crosswinds.

Chapter 5

Discussion

This chapter reviews the simulations from chapter 4, debates the model's validity, and discusses important considerations when implementing TVC on a sounding rocket learned when developing the model in this thesis.

5.1 A review of the simulations

The model is designed to provide a reliable baseline for comparing simulations with and without TVC while making some simplifications. This section discusses how the base model without TVC compares to the simulations in OpenRocket, how TVC affects the base model, and how these results can be translated to an actual launch.

5.1.1 Accuracy of the base model

The general trend of the python simulation matches OpenRocket, but there is a substantial difference in lateral distance. The difference in lateral distance without wind indicates some error in the Python model. When launching at 80 degrees elevation, it makes little sense for the rocket to travel further in lateral distance than in altitude. That deviation can be explained, at least partially, by the rapid change in vertical orientation during the burn time. This, however, is alarming, as the primary source of error is in the area where the TVC operates. This must be taken into account when discussing the final results with TVC.

The two models respond similarly to increasing headwinds. The response to headwind can then be assumed to be correct with TVC. In conditions other than a headwind, the models diverge. Although these conditions may be considered unsafe for launching a real sounding rocket, it shows the limits of the different models.

The distance toward North in the Python simulation when in a crosswind is too large only to be the result of drift. This is likely due to a similar error in azimuth as in vertical orientation, as mentioned earlier. These angles are closely related due to how angles are implemented in the Python model. The results in OpenRocket are also strange. If the 8 m/s wind changes the distance East by 687 m in a headwind, it should have some effect on the distance North in a crosswind. This indicates some errors in the OpenRocket simulations, either in the model created for this thesis or in how the software handles conditions outside its intended use.

In tailwind, both the OpenRocket simulation with spin and the simulation in Python tumble during launch. Although the vertical orientation at launch might be too steep in a strong tailwind, the OpenRocket simulation without spin does just fine. Due to how aerodynamics is implemented in the python model, the rocket tumbles more easily with rapid changes in the vertical orientation. However, rapid changes in any orientation are considered unsafe on a real mission. This test confirms that the results in conditions other than direct headwind should be analyzed further.

Difference in parameters

Most parameters were chosen to match Python and OpenRocket, but some were more difficult to implement. Figure 5.1 shows these differences in a simulation without wind.

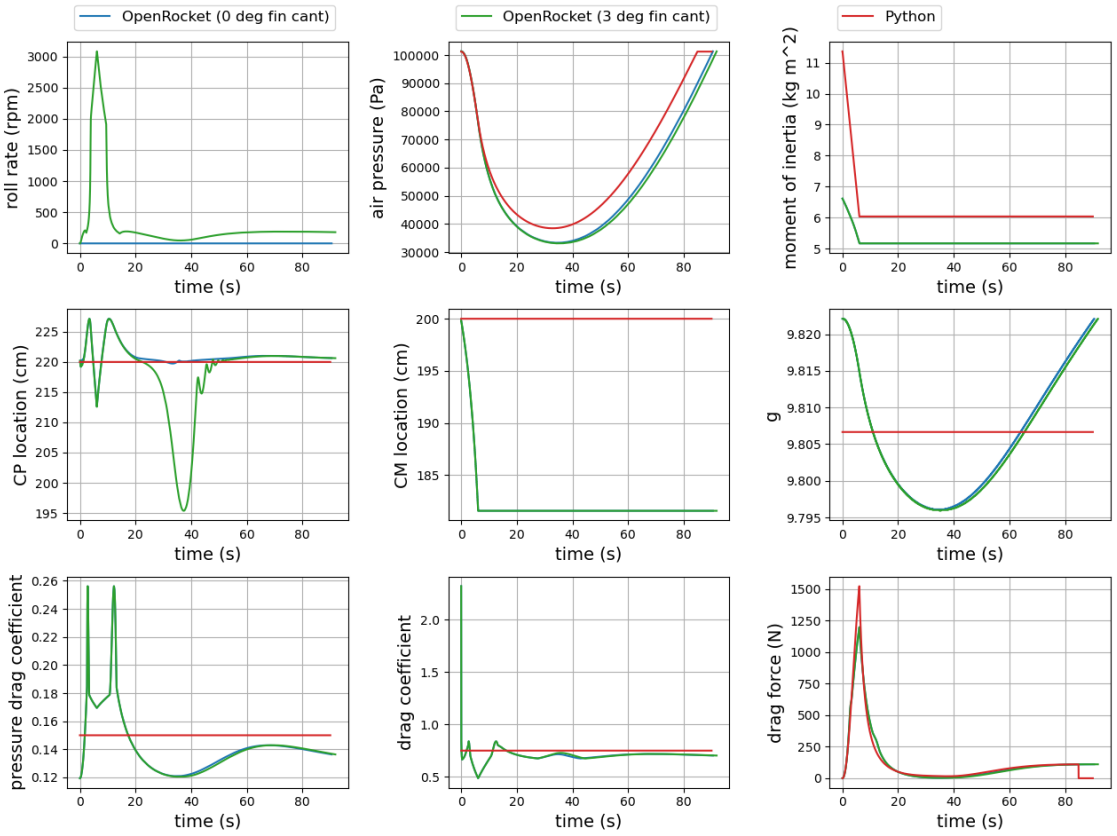

Figure 5.1: Differences in roll rate, air pressure, the moment of inertia, the center of pressure (C_P), the center of mass (C_M), gravitational constant, lift coefficient and drag coefficient between Python and OpenRocket.

The simulations in OpenRocket are both with and without a fin cant that induces a spin. While the Python model uses constant values for the gravitational constant, drag coefficient, lift coefficient, location of the center of mass, and center of pressure, OpenRocket uses multiple data points for each of them, resulting in an overall more complex simulation. However, the difference in these parameters alone would make OpenRocket more susceptible to torques during the early stages of flight. The difference in the moment of inertia has the same effect. The difference in air pressure is due to the difference in altitude, but the implementation is the same

in both models. The difference in drag force explains some of the differences in absolute velocity and acceleration. The difference in lift coefficient might also have an effect due to the corresponding reduction in lift force.

In addition to differences in parameters, the numerical method used to solve the ODEs impacts the final result. The total truncation error is irrelevant, but the change in the direction of the thrust force amplifies the errors in orientation during the burn time. Thus the impact of the truncation error during one time step is disproportionally significant. Although the error is small compared with the other sources, OpenRocket uses Runge-Kutta as its numerical solver, which is more accurate than the forward Euler used in the Python model.

In sum, the simplifications made in these parameters are expected to reduce the change in vertical orientation during the burn time but with reduced accuracy due to the numerical solver. Assuming the model designed in OpenRocket behaves as it should, this would indicate some error in the implementation of torque in the Python model. Either numerically in the integration loop, aerodynamics modeling, or both.

Conclusion

The Python model was expected to have some errors due to the number of simplifications and the error in the numerical method. OpenRocket itself is far from the most accurate baseline for comparison, and the results from varying the wind direction also indicate some errors in the OpenRocket simulations. There are likely some errors in both simulations, and therefore difficult to determine the model's accuracy in Python from this comparison. Although the results from the comparison are non-conclusive, they highlight some areas where the model is vulnerable. When evaluating the results with TVC, the difference in lateral distance is expected to be greater than it would be for an actual launch. The trajectory after burnout might have more inaccuracies than the simplifications alone due to the implementation of aerodynamics.

5.1.2 Effects of TVC

As it is impossible to determine the accuracy of the base model without further study and improvements, the accuracy of the TVC compared to OpenRocket cannot be determined either. However, the general trend of the results is enough to continue discussing how an actual launch would be affected by TVC. The difference in lateral distance in these results is expected to be greater than for an actual launch.

All simulations show a substantial decrease in lateral distance and an increase in altitude using TVC, with a larger difference in stronger winds. While the lateral distance increases with an increasing wind without TVC, the opposite is observed with TVC. The lateral distance with TVC is still larger than in the simulations with OpenRocket. With TVC, the vertical orientation matches the set point for the PID during the entire burn time for all simulations, and there are no signs of instability.

Although the wind direction greatly influences the trajectory, it does not affect the performance of the TVC. Although Figure 4.13 shows oscillations in the lift force with TVC after the burn time, the other figures show that the rocket's stability is

just as good in any wind direction. Oscillations impact stability most during the burn time when the thrust amplifies any rotations. Even though the oscillations in the lift force with TVC after the burn time have a higher amplitude than the oscillations during burn time without TVC, the overall system is more stable with TVC.

The simulation in tailwind shows the advantage of TVC during launch. Where the rocket without TVC tumbles, the rocket with TVC shows no sign of instability. In a crosswind, the rocket with TVC can withstand the pull in azimuth, but this pull is likely just an error in the model. Although the difference in the lateral distance when the rocket lands changes by as much as 2 km in storm-force winds, the rocket still seems stable in these conditions with TVC. Because of the inaccuracies in the Python model, it is difficult to conclude from the simulations in Figures 4.14-4.16.

Conclusion

As the magnitude of the thrust force is the highest among the forces affecting the rocket, the stability, and orientation during the burn time affect the final trajectory considerably. These simulations show that it is possible to successfully tune a PID controller to keep the rocket from deviating from its set direction. There is undoubtedly an increase in accuracy and stability during the launch with TVC, but it is impossible to determine the exact increase from this simulation. In reality, the increase in accuracy would depend on the rocket's design and how much it tilts during launch.

5.2 Important lessons

Choice of rocket engine

The choice of rocket engine heavily impacts the requirements of the controller. A simple PID controller is best suited when the output thrust is close to constant. Large variations in thrust would cause the controller to act non-deterministic for a short time step without other systems, as the I-term would not be able to compensate in time.

As the TVC only works while the engine is firing, it is more effective on engines with a longer burn time. This enables the TVC to maintain the rocket's set course for longer.

The full potential of TVC is the possibility of changing the trajectory mid-flight to maximize the time spent in the relevant area. This would require turning the thrust on and off during the flight or second-stage boosters.

Settling time

This model assumes that the response time of the TVC is infinitesimal with perfect accuracy compared to the output from the PID controller. In reality, there are sampling delays from the sensors, signals processing, and actuators changing the angle of the engine. Although the PID controller can compensate for these delays, the final settling time would increase and thus lower the efficiency of the TVC.

In addition, this negatively impacts fuel efficiency. As more fuel is spent to generate torques to stabilize the rocket, less force can be used to accelerate the rocket. This topic is not discussed in this thesis, but it is still worth noting.

When designing the TVC, the specifications for the sensors and actuators and the required resources for signal processing in the flight computer must be defined with the engine specifications in mind. The maximum thrust of the rocket engine limits the tolerance for delays in the PID. A higher peak thrust requires higher precision and faster response than a lower peak thrust, as the system is more likely to overshoot, reducing the stability and increasing the settling time.

On tuning the PID to actual conditions

A significant advantage of PID controllers is their ability to be tuned on the spot. Although an initial tuning is a good starting point, different conditions require some alterations to the initial tuning. Figure 5.2 shows how the initial tuning affects the rocket in different conditions during the burn time of the engine in this simulation. Although 25 m/s headwind might be too severe, even with TVC, it still shows the general trend of the system with increasing wind speed. Depending on the conditions, increasing wind speed requires additional dampening to minimize the amplitude of the oscillations. This would result in a higher D-term. The I-term would need to be increased to reduce the steady-state error, and the P-term would be reduced to compensate if the controller still lacks stability. In addition, the increased D-term would require additional precision in filtering the input signal to limit noise from affecting the output signal.

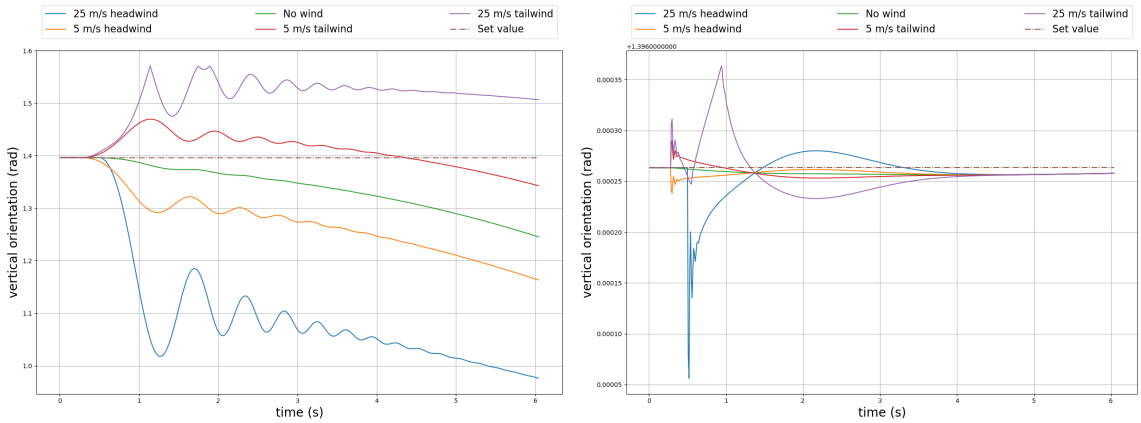

Figure 5.2: How the vertical orientation changes during the burn time in different wind conditions. The left plot shows simulations without TVC, and the right plot shows simulations with the initially tuned TVC.

Since the noise in this model would grow out of control with an increasing D-term, all simulations use the same coefficients for the PID. On an actual sounding rocket, the difference in performance in strong winds versus lower wind speeds is expected to be slightly lower than the results in this model.

Combination with other systems

TVC can easily be combined with spin stabilization, although the spin is irrelevant

to the model in this thesis due to the implementation of forces and aerodynamics. With gimballed thrust and short delays in the PID, the controller can compensate for a spin throughout the flight. This would result in a TVC-controlled flight during the burn time and spin stabilization for the rest. As spin-stabilization is not needed during the burn time, an actuator that angles the fins after the engine has stopped firing can help reduce drag and complexity in the TVC early in the flight and still enable both systems.

Chapter 6

Conclusion

The primary objective of this thesis was to develop a numerical simulation tool in Python that can simulate the trajectory of sounding rockets with TVC. The secondary objective was to explore how sounding rockets with TVC behave to determine how the TVC affects the accuracy during launch using the simulation tool developed.

The model developed in this thesis can simulate rocket trajectories with TVC during launch. The simulations showed that it is possible to successfully tune a PID controller to keep the rocket from deviating from its set vertical orientation with an accuracy of $1.55 \cdot 10^{-5}$ rad. It is, however, impossible to determine the base model's accuracy from the comparison with OpenRocket due to the error in vertical orientation during the burn time in Python. The model was expected to have some errors due to the number of simplifications, and the primary objective was an overall success. The model can be developed further to improve accuracy or add functionality. The results from the simulations highlight some areas where the model is vulnerable and should be improved if used in further studies. These are addressed in the final remarks.

Although it is impossible to conclude the goal in the second objective, there is undoubtedly an increase in accuracy and stability during the launch with TVC. In reality, the increase in accuracy would depend on the rocket's design and how much it tilts during launch. The difference in the altitude and lateral distance when using TVC is likely less in an actual launch than in the results presented in this thesis.

6.1 Final remarks

Why TVC is not used today

The discussion in existing papers and textbooks around the uses of TVC on sounding rockets is limited. However, the motives for using TVC on any vehicle using jet propulsion, as stated by (Sutton and Biblarz, 2016, p. 672), is:

- Willful changes to a flight path or trajectory.
- Vehicle rotation or changes in attitude during powered flight.
- Deviation corrections from intended trajectory or attitude during powered flight.
- Thrust corrections for misalignments of fixed nozzles in the main propulsion system during its operation.

Misalignments of a fixed nozzle can be corrected by spin stabilization, but any active corrections or changes to trajectory or attitude require an active system, such as TVC. Orbital launchers require TVC for the precision required to enter and maintain their desired orbit. This control is not needed on most sounding rocket missions. Orbital missions usually have a large budget and a long development time. Sounding rockets are usually cheap and have a short development time (NASA, 2021c). The cost and added development time for TVC might be too steep in cases where satellites in Low Earth Orbit are an alternative. This is likely why TVC is currently under-explored for sounding rockets.

The closest relatives to sounding rockets in terms of size are missiles designed for military purposes. Though TVC was used on German missiles in World War II (Sutton and Biblarz, 2016, p. 676), its use in military applications has not transferred to sounding rockets. Military technology is mainly classified or otherwise unavailable to the public, which might be another reason TVC is not used on sounding rockets.

TVCs potential on sounding rockets

Although this thesis discusses the effects of thrust vector control, the major drawbacks of this kind of system remain. One hurdle is the complexity of the system and the expertise needed to design, implement and maintain it. The other one is the cost, both a matter of priority. With more complex systems, more things can go wrong. This might result in an increased need for wider safety margins during launch, though this is difficult to quantify in a master's thesis. It is also beyond the scope of this study to discuss the economics of this decision. However, from a practical point of view, TVC is undoubtedly an enabler of other technologies.

The real potential of using TVC on sounding rockets is to perform changes to its trajectory mid-flight, not just during launch. This could enable the rocket to achieve a more optimal trajectory, allowing more time spent in the region of interest and more samples from each mission. In addition, precise control of the rocket's attitude is a prerequisite for safely landing it and making it completely reusable. Another possibility is to use TVC combined with a variable thrust to hover in the region of interest. This could further increase the number of samples by multiple orders

of magnitude. The rocket would thus function as a hybrid between a rocket and a satellite.

Future work

Some of the simplifications made in this thesis could be implemented more accurately with little work. Moment of inertia, gravitation, center of pressure, center of mass, drag coefficient, and lift coefficient all differ from the model in OpenRocket and should be investigated. If using an engine with uneven thrust, its thrust curve could also be added.

The work in this thesis could be taken further in two main directions. One is to keep focusing on sounding rockets and implement a second booster stage with TVC. The rocket could then be simulated with different launch angles and set angles for the two PIDs to study how to optimize the trajectory for measuring plasma. In addition, implementing a more efficient numerical solver, like Runge-Kutta, can improve accuracy, which might be necessary for a second booster stage. Runge-Kutta would reduce the truncation and rounding error, which might also limit the D-term in the PID.

Another direction could be shifting the focus to model rocketry and a study on rocket stability to determine how the rocket behaves in gusts of wind and turbulence. The TVC can be implemented and tested on a model rocket and compared to the simulation. If implementing the TVC on a model rocket, the sensor data should be filtered to compensate for noise amplification in the D-term. By implementing spin-stabilization after burnout, the stability could be improved. The inclusion of spin stabilization requires an overhaul of the local coordinates, but by translating the output of the PID controller to spherical coordinates, the roll can be added to the output during the booster stage to utilize both methods.

Bibliography

Anderson, J. D. (2016). *Introduction to flight* (8th ed.). McGraw Hill Education.

Andøya Space Education. (2023). Andøya space eduaction student rocket. Retrieved January 10, 2023, from https://learn.andoyaspace.no/ebook/student-rocket-pre-study/narom-student-rocket/

Ang, K. H., Chong, G., & Li, Y. (2005). PID control system analysis, design, and technology. *IEEE Transactions on Control Systems Technology*, *13*(4). https://doi.org/10.1109/TCST.2005.847331

Apogee. (2023). RS-PRO: Flight simulator for high speed, high altitude rockets. Retrieved January 10, 2023, from https://www.apogeerockets.com/index.php?main_page=product_software_info&cPath=13_205&products_id=726

ASTOS. (2023). Analysis, simulation and trajectory optimization software for space applications. Retrieved January 10, 2023, from https://www.astos.de/products/astos

B. Douglas [MATLAB]. (2022). Understanding pid control [youtube playlist]. Retrieved December 10, 2022, from https://youtube.com/playlist?list=PLn8PRpmsu08pQBgjxYFXSsODEF3Jqmm-y

Bekkeng, T. A., Jacobsen, K. S., Bekkeng, J. K., Pedersen, A., Lindem, T., Lebreton, J.-P., & Moen, J. I. (2010). Design of a multi-needle langmuir probe system. *Measurement Science and Technology*, *21*(8). https://doi.org/10.1088/0957-0233/21/8/085903

Butcher, J. C. (2008). *Numerical methods for ordinary differential equations* (2nd ed.). Wiley.

Canadian Association of Rocketry. (2010). Thrust curve for a Pro98-6G rocket engine [graph]. Cesaroni Technology Inc. http://pro38.com/products/pro98/motor/images/15227N2501-P.pdf

Canuto, E., Novara, C., Massotti, L., Carlucci, D., & Montenegro, C. P. (2018). *Spacecraft dynamics and control: The embedded model control approach*. Elsevier Science and Technology Books.

Cesaroni Technology Inc. (2023). Pro98 15227n2501-p. Retrieved January 10, 2023, from http://pro38.com/products/pro98/motor/MotorData.php?prodid=15227N2501-P

Dabberdt, W., & Turtiainen, H. (2014). *Encyclopedia of atmospheric sciences* (2nd ed., Vol. 4). Academic Press.

Estes. (2023). Get started - estes rockets. Retrieved January 10, 2023, from https://estesrockets.com/get-started/

Fraden, J. (2015). *Handbook of modern sensors physics, designs, and applications: Physics, designs, and applications* (5th ed.). Springer.

Glenn research center. (2021a). The drag equation. Retrieved January 10, 2023, from https://www.grc.nasa.gov/www/k-12/rocket/drageq.html#:~:text=The%20drag%20equation%20states%20that,times%20the%20reference%20area%20A.

Glenn research center. (2021b). The lift coefficient. Retrieved January 10, 2023, from https://www.grc.nasa.gov/www/k-12/rocket/liftco.html

Hegarty, C. J., & Chatre, E. (2008). Evolution of the global navigation SatelliteSystem (GNSS). *Proceedings of the IEEE*, *96*(12). https://doi.org/10.1109/jproc.2008.2006090

Hoerner, S. F. (1965). *Aerodynamic drag and hydrodynamic resistance*. Published by the Author.

International Orginization for Standardization. (1975). Standard atmosphere (ISO 2533:1975). https://www.iso.org/standard/7472.html

Jacobsen, K. S., Pedersen, A., Moen, J. I., & Bekkeng, T. A. (2010). A new langmuir probe concept for rapid sampling of space plasma electron density. *Measurement Science and Technology*, *21*(8). https://doi.org/10.1088/0957-0233/21/8/085902

McConnaughey, P. K., Femminineo, M. G., Koelfgen, S. J., Lepsc, R. A., Ryan, R. M., & Taylor, S. A. (2012). *NASA's Launch Propulsion Systems Technology Roadmap*. National Aeronautics and Space Administration (NASA).

McLean, D. (2012). *Understanding aerodynamics arguing from the real physics*. John Wiley and Sons.

Met Office. (2023). Beaufort wind force scale. Retrieved January 10, 2023, from https://www.metoffice.gov.uk/weather/guides/coast-and-sea/beaufort-scale

NASA. (2021a). Rocket rotations. Retrieved January 10, 2023, from https://www.grc.nasa.gov/www/k-12/rocket/rotations.html

NASA. (2021b). Sounding rockets and tracers. Retrieved July 15, 2022, from https://www.nasa.gov/mission_pages/sounding-rockets/tracers/rockets.html

NASA. (2021c). Sounding rockets overview. Retrieved July 15, 2022, from https://www.nasa.gov/mission_pages/sounding-rockets/missions/index.html

Niskanen, S. (2009). *Development of an open source model rocket simulation software* (Master's thesis). Helsinki University of Technology. https://github.com/openrocket/openrocket/releases/download/Development_of_an_Open_Source_model_rocket_simulation-thesis-v20090520/Development_of_an_Open_Source_model_rocket_simulation-thesis-v20090520.pdf

OpenRocket. (2023). Openrocket simulator. Retrieved January 10, 2023, from https://openrocket.info/

Peng, W., Zhang, Q., Yang, T., & Feng, Z. (2017). A high-precision dynamic model of a sounding rocket and rapid wind compensation method research. *Advances in Mechanical Engineering*, *9*(7), 168781401771394. https://doi.org/10.1177/1687814017713944

Schunk, R. W., & Nagy, A. F. (2000). *Ionospheres: Physics, plasma physics, and chemistry (cambridge atmospheric and space science series)*. Cambridge University Press.

Sounding Rockets Program Office. (2015). *Nasa sounding rockets user handbook*. NASA. https://sites.wff.nasa.gov/code810/files/SRHB.pdf

Study.com. (2010). Temperature variations in the atmosphere [graph]. https://study.com/learn/lesson/lapse-rates-atomsphere-types-formulas-overview.html

Sutton, G. P., & Biblarz, O. (2016). *Rocket propulsion elements*. John Wiley and Sons, Inc.

UiO. (2015). CaNoRock. Retrieved January 10, 2023, from https://www.mn.uio.no/fysikk/english/research/projects/4dspace/education/03_canorock.html

Urquizo, A. (2008). Block diagram of a PID-controller in a feedback loop [image]. https://commons.wikimedia.org/wiki/File:PID.svg

Wertz, J. R., Everett, D. F., & Puschell, J. J. (2011). *Space mission engineering: The new SMAD*. Microcosm Press.

You, Z. (2017). *Space microsystems and micro/nano satellites*. Elsevier Science and Technology Books.

Appendix A

Source Code

This appendix contains all code used to simulate trajectories in this thesis. In addition, the design file for the model in Open Rocket, along with all .py- and .csv-files, are uploaded to GitHub: https://github.uio.no/master-projects/Marcus-Nitschke-python

A.1 launchsim.py

The base model.

```
import numpy as np
import matplotlib.pyplot as plt

class Rocket():
    def __init__(self, launch_ang, tmax, wind_speed, wind_ang, dry_mass,
                 wet_mass, length, cd, cl, critical_angle, hcm, hcp, radius,
                 thrustforce, burntime, dt=0.01, rail_length=5.0):
        """Initialize class.
        Args:
            launch_ang (touple): Launch angle (deg [elevation, azimuth]).
            tmax (int): Length of simulation (s).
            wind_speed (float): Wind speed (m/s).
            wind_ang (float): Wind direction (deg).
            dry_mass (float): Dry mass (kg).
            wet_mass (float): Wet mass (kg).
            length (float): Length of the rocket (mm).
            cd (float): Drag coefficient.
            cl (float): Lift coefficient.
            hcm (float): Height of the centre of mass (mm).
            hcp (float): Height of the centre of pressure (mm).
            radius (float): Radius of nose cone (mm).
            thrustforce (float): Force of thrust (N).
            burntime (float): Burntime (s).
            dt (float): Time step (s). Default to 0.01.
            rail_length (float): Lenght of launch rail (m). Default to 5.0.
        """
        self._dt = dt
        self.t = np.linspace(0, tmax, tmax*int(1/self._dt)+1)
        self.r, self.v, self.a = np.zeros((3, tmax*int(1/self._dt)+1, 5))

```

```
31          #Set initial launch angle in Euler angles pitch and yaw.
32          self.r[0][3] = -((np.pi/2)-launch_ang[0]*np.pi/180)*np.cos(
33              launch_ang[1]*np.pi/180)
34          self.r[0][4] = ((np.pi/2)-launch_ang[0]*np.pi/180)*np.sin(
35              launch_ang[1]*np.pi/180)
36          for i in range(2):
37              if abs(self.r[0][i+3]) < 10E-14:
38                  self.r[0][i+3] = 0
39
40          self._rodlenght = rail_length
41          self._burntime = burntime
42          self._thrustforce = thrustforce
43          self._mt = wet_mass
44          self._mf = wet_mass-dry_mass
45          self._m = [wet_mass]
46          self._cd = cd
47          self._cl = cl
48          self._hcm = hcm/1000
49          self._hcp = hcp/1000
50          self._wind = -wind_speed*np.array([np.sin(wind_ang*np.pi/180),
51              np.cos(wind_ang*np.pi/180), 0])
52          self._side_area = (length/1000)*(radius/1000)*2
53          self._front_area = np.pi*((radius/1000)**2)
54          self._critical_angle = critical_angle*np.pi/180
55          self._radius = radius/1000
56          self._length = length/1000
57          self.angle_attack = [np.zeros(3)]
58          self.fthrust = [np.zeros(3)]
59          self.fdrag = [np.zeros(3)]
60          self.flift = [np.zeros(3)]
61
62          self._g = 9.80665 #Gravitational constant (m/s^2).
63          self._p0 = 101325 #standard absolute atmospheric pressure (Pa).
64          self._air_molar = 0.0289654 #Molar mass of dry air (kg/mol).
65          self._gas_constant = 8.314463 #Ideal gas constant (J/(mol*k)).
66          self._t0 = 288.15 #Absolute temperature at sea level (K).
67          self._t_laps = 0.0065 #Temperature laps rate (K/m).
68          self._rho = [1.225] #Air density (kg/m^3).
69
70          #First set of calculations for the air density.
71          self._c = (self._p0*self._air_molar/(self._gas_constant*self._t0))
72          self._exp = ((self._g*self._air_molar/(
73              self._gas_constant*self._t_laps))-1)
74          return
75
76      def rotate(self, rot, vector, inverse=False):
77          """
78          Rotate a vector counter clockwise by its x- and y-axis.
79
80          Args:
81              rot (np.array): Rotation (pitch, yaw).
82              vector (np.array): input vector  (East, North, Altitude).
83              inverse (bool): Invert the rotation if 'True'. Default 'False'.
84          Returns:
85              np.array: Rotated output vector.
86          """
87          rotation = np.array([[np.cos(rot[1]), np.sin(rot[1])*np.sin(rot[0]),
88              np.sin(rot[1])*np.cos(rot[0])], [0, np.cos(rot[0]),
```

```
89              -np.sin(rot[0])], [-np.sin(rot[1]), np.cos(rot[1])*np.sin(rot[0]),
90              np.cos(rot[1])*np.cos(rot[0])]])
91
92          if inverse:
93              output = rotation.T @ vector
94          else:
95              output = rotation @ vector
96          return output
97
98      def update(self, t, r, v):
99          """Update mass (kg) as a function of time (s), air density (kg/m^3) as a
100         function of height (m) and calculate the angle of attack (rad).
101         Args:
102             t (float): Time since initialization in seconds.
103             r (np.array): Positional vector (East, North, altitude, theta, phi).
104             v (np.array): Velocity vector (East, North, altitude, theta, phi).
105         """
106         if t >= self._burntime:
107             self._m.append(self._mt - self._mf)
108         else:
109             self._m.append(self._mt - (self._mf*t/self._burntime))
110
111         self._rho.append(self._c*(1-(self._t_laps*r[2]/self._t0))**self._exp)
112
113         rel_v = v[:3] - self._wind
114         motion_theta = 0
115         if abs(rel_v.dot(rel_v)) > 0:
116             motion_theta = np.arccos(rel_v[2]/np.sqrt(rel_v.dot(rel_v)))
117         rocket_theta = np.arccos(np.cos(r[3])*np.cos(r[4]))
118         self.angle_attack.append(np.array([motion_theta, rocket_theta,
119             motion_theta - rocket_theta]))
120         return
121
122     def weight(self, r):
123         """Weight as a vector pointing from the center of mass of the rocket
124          towards negative z in the main coordinate system.
125         Returns:
126             np.array: Force from weight as a vector
127                 (East, North, altitude, theta, phi).
128         """
129         weight = np.array([0, 0, -self._g*self._m[-1]])
130         if np.sqrt(r.dot(r)) < self._rodlenght:
131             l_weight = np.array([0, 0,
132                 self.rotate(r[3:], weight, inverse=True)[2]])
133             weight = self.rotate(r[3:], l_weight)
134         return np.array([weight[0], weight[1], weight[2], 0, 0])
135
136     def thrust(self, t, r):
137         """Thrust as a vector pointing from the back of the rocket towards
138          positive z in the local reference frame, converted into the global
139          reference frame.
140         Args:
141             t (float): Time since initialization in seconds.
142             r (np.array): Positional vector (East, North, altitude, theta, phi).
143         Returns:
144             np.array: Force from thrust as a vector
145                 (East, North, altitude, theta, phi).
146         """
```

```
147         if t < self._burntime:
148             #Thrust is now a vector in the local reference frame.
149             l_thrust = self._thrustforce*np.array([0, 0, 1])
150             #Convert the vector to the global reference frame.
151             g_thrust = self.rotate(r[3:], l_thrust)
152             return np.array((g_thrust[0], g_thrust[1], g_thrust[2], 0, 0))
153         else:
154             return np.zeros(5)
155 
156     def drag(self, r, v):
157         """Drag as a vector pointing from the center of pressure opposite the
158          direction of motion in a global reference frame.
159         Args:
160             r (np.array): Positional vector (East, North, altitude, theta, phi).
161             v (np.array): Velocity vector (East, North, altitude, theta, phi).
162         Returns:
163             np.array: Force from drag as a vector
164                 (East, North, altitude, theta, phi).
165         """
166         rel_v = v[:3] - self._wind
167         #Drag in global reference frame.
168         g_drag = -0.5*self._cd*self._front_area*self._rho[-1]*rel_v*abs(rel_v)
169 
170         return np.array([g_drag[0], g_drag[1], g_drag[2], 0, 0])
171 
172     def lift(self, r, v):
173         """Lift as a vector pointing from the center of pressure perpendicular
174          to the direction of motion in a global reference frame.
175         Args:
176             r (np.array): Positional vector (East, North, altitude, theta, phi).
177             v (np.array): Velocity vector (East, North, altitude, theta, phi).
178         Returns:
179             np.array: Force from lift as a vector
180                 (East, North, altitude, theta, phi).
181         """
182         rel_v = v[:3] - self._wind
183         orientation = self.rotate(r[3:], np.array([0, 0, 1]))
184         dir_lift = np.cross(rel_v, np.cross(orientation, rel_v))
185 
186         cl = 0
187         if abs(self.angle_attack[-1][2]) < self._critical_angle:
188             cl = self._cl*abs(1 - (abs(self.angle_attack[-1][2] -
189                  self._critical_angle)/self._critical_angle))
190 
191         lift = 0.5*cl*self._side_area*self._rho[-1]*rel_v**2
192         mag_lift = np.sqrt(lift.dot(lift))
193 
194         if dir_lift.dot(dir_lift) != 0:
195             g_lift = mag_lift*dir_lift/np.sqrt(dir_lift.dot(dir_lift))
196         else:
197             g_lift = np.array([0, 0, 0])
198 
199         #Convert the vector to the local reference frame.
200         l_lift = self.rotate(r[3:], g_lift, inverse=True)
201 
202         f_lift = np.array([g_lift[0], g_lift[1], g_lift[2],
203             -l_lift[1]*(self._hcp-self._hcm), l_lift[0]*(self._hcp-self._hcm)])
204         if np.sqrt(r.dot(r)) < self._rodlenght:
```

```
205             g_lift = self.rotate(r[3:], np.array([0, 0, l_lift[2]]))
206             f_lift = np.array([g_lift[0], g_lift[1], g_lift[2], 0, 0])
207         return f_lift
208 
209     def acceleration(self, t, r, v):
210         """Calculate the acceleration using Newtons 2. law.
211         Args:
212             t (float): Time since initialization in seconds.
213             r (np.array): Positional vector (East, North, altitude, theta, phi).
214             v (np.array): Velocity vector (East, North, altitude, theta, phi).
215         Returns:
216             np.array: New acceleration vector
217                 (East, North, altitude, theta, phi).
218         """
219         f_thrust = self.thrust(t, r)
220         f_drag = self.drag(r, v)
221         f_weight = self.weight(r)
222         f_lift = self.lift(r, v)
223         acc = np.array([(f_thrust[:3] + f_drag[:3] + f_lift[:3] +
224             f_weight[:3])/self._m[-1]])
225         ang_acc = np.array([(f_thrust[3:] + f_lift[3:])/
226             (self._m[-1]*(3*self._radius**2+self._length**2)/12)])
227         for i in range(2):
228             if abs(ang_acc[-1][i]) < 10E-14:
229                 ang_acc[-1][i] = 0
230         self.fthrust.append(f_thrust[:3])
231         self.fdrag.append(f_drag[:3])
232         self.flift.append(f_lift[:3])
233         return np.concatenate((acc, ang_acc), axis=None)
234 
235     def launch(self):
236         """Solves the differential equation using the step function."""
237         for i in range(len(self.t)-1):
238             self.a[i+1] = self.acceleration(self.t[i], self.r[i], self.v[i])
239             self.r[i+1], self.v[i+1] = self.step(self.t[i], self.r[i],
240                                                  self.v[i], self.a[i+1])
241         self.angle_attack = np.array(self.angle_attack, dtype=object)
242         self.fdrag = np.array(self.fdrag, dtype=object)
243         self.fthrust = np.array(self.fthrust, dtype=object)
244         self.flift = np.array(self.flift, dtype=object)
245         return
246 
247     def step(self, t, r, v, a):
248         """A modified Forward Euler step function.
249         Args:
250             t (float): Time since initialization in seconds.
251             r (np.array): Positional vector (East, North, altitude, theta, phi).
252             v (np.array): Velocity vector (East, North, altitude, theta, phi).
253             a (np.array): Acceleration vector
254                 (East, North, altitude, theta, phi).
255         Returns:
256             np.array: New positional vector (East, North, altitude, theta, phi).
257             np.array: New velocity vector (East, North, altitude, theta, phi).
258          """
259         v_nxt = np.zeros(5)
260         if r[2] >= 0:
261             v_nxt = v + self._dt*a
262         r_nxt = r + self._dt*v_nxt
```

```
263             self.update(t, r_nxt, v_nxt)
264             return r_nxt, v_nxt
```

A.2 TVC.py

Implementation of TVC.

```
import numpy as np
import matplotlib.pyplot as plt
from launchsim import Rocket

class Rocket_TVC(Rocket):

    def __init__(self, kp, ki, kd, launch_ang, max_angle=5, **kwargs):
        """Initialize class.
        Args:
            kp (float): proportional coefficient.
            ki (float): integral coefficient.
            kd (float): derivative coefficient.
            launch_ang (touple): Launch angle (deg [theta, phi]).
            max_angle (float): Maximum angle of the thrust vector in any
                direction (deg).
            **kwargs: input for super class.
        """
        super().__init__(launch_ang, **kwargs)
        self._kp = kp
        self._ki = ki
        self._kd = kd
        self._max_angle = max_angle
        self._prev_error = np.zeros(2)
        self._total_error = 0
        self._desired_ang = np.array([self.r[0][3], self.r[0][4]])
        return

    def thrust(self, t, r):
        """Thrust as a vector pointing from the back of the rocket with an
         angle depending on the TVC from positive z in the local frame of
         reference, converted into the global frame of reference.
        Args:
            t (float): Time since initialization (s).
            r (np.array): Positional vector
                (East, North, altitude, pitch, yaw).
        Returns:
            np.array: Force from thrust as a vector
                (East, North, altitude, pitch, yaw).
        """
        if t < self._burntime:
            #Thrust is now a vector in the local frame of reference.
            thrust = self._thrustforce*np.array([0, 0, 1])
            l_thrust = self.rotate(self.pid(r), thrust)
            #Convert the vector to the global frame of reference.
            g_thrust = self.rotate(r[3:], l_thrust)
            return np.array((g_thrust[0], g_thrust[1], g_thrust[2],
                             -l_thrust[1]*self._hcm, l_thrust[0]*self._hcm))
        else:
            return np.zeros(5)

    def pid(self, r):
        """Calculate the angle of the thrust vector using PID.
        Args:
            r (np.array): Positional vector (East, North, altitude, pitch, yaw).
        Returns:
```

```
56              np.array: Angle of thrust vector (rad [pitch, yaw]).
57          """
58          error = self._desired_ang - r[3:]
59          self._total_error += error*self._dt
60          proportional = self._kp*error
61          integral = self._ki*self._total_error
62          derivative = self._kd*(error - self._prev_error)
63          u = proportional + integral + derivative
64
65          for i in range(len(u)):
66              if u[i] > self._max_angle*(np.pi/180):
67                  u[i] = self._max_angle*(np.pi/180)
68              elif u[i] < -self._max_angle*(np.pi/180):
69                  u[i] = -self._max_angle*(np.pi/180)
70          self._prev_error = error
71          return u
```

A.3 comparison.py

Comparison with OpenRocket

```
import numpy as np
import matplotlib.pyplot as plt
from launchsim import Rocket
import pandas as pd

def comparison(df, mk, dfr):
    plt.figure(figsize=[12.8, 9.6])
    plt.subplot(221)
    plt.plot(df['Time (s)'], df['Altitude (m)'], '--', c='tab:red',
        label="Altitude OR")
    plt.plot(dfr['Time (s)'], dfr['Altitude (m)'], ':', c='tab:red',
        label="Altitude OR (roll)")
    plt.plot(mk.t, mk.r.T[2], c='tab:red', label="Altitude Python")
    plt.plot(df['Time (s)'], df['Position East of launch (m)'], '--',
        c='tab:blue', label="East OR")
    plt.plot(dfr['Time (s)'], dfr['Position East of launch (m)'], ':',
        c='tab:blue', label="East OR (roll)")
    plt.plot(mk.t, mk.r.T[0], c='tab:blue', label="East Python")
    plt.plot(df['Time (s)'], df['Position North of launch (m)'], '--',
        c='tab:green', label="North OR")
    plt.plot(dfr['Time (s)'], dfr['Position North of launch (m)'], ':',
        c='tab:green', label="North OR (roll)")
    plt.plot(mk.t, mk.r.T[1], c='tab:green', label="North Python")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("distance (m)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.subplot(222)
    plt.plot(df['Time (s)'], df['Total velocity (m/s)'], '--', c='tab:orange',
        label="Velocity OR")
    plt.plot(dfr['Time (s)'], dfr['Total velocity (m/s)'], ':', c='tab:orange',
        label="Velocity OR (roll)")
    plt.plot(mk.t, abs(mk.v.T[0])+abs(mk.v.T[1])+abs(mk.v.T[2]),
        c='tab:orange', label="Velocity Python")
    plt.plot(df['Time (s)'], df['Total acceleration (m/s)'], '--',
        c='tab:purple', label="Acceleration OR")
    plt.plot(dfr['Time (s)'], dfr['Total acceleration (m/s)'], ':',
        c='tab:purple', label="Acceleration OR (roll)")
    plt.plot(mk.t, abs(mk.a.T[0])+abs(mk.a.T[1])+abs(mk.a.T[2]),
        c='tab:purple', label="Acceleration Python")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("velocity (m/s) and acceleration (m/s^2)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left", mode="expand",
        ncol=2, fontsize=12)
    plt.grid()
    plt.subplot(223)
    plt.plot(df['Time (s)'], df['Vertical orientation (zenith) (rad)'], '--',
        c='tab:cyan', label="OR")
    plt.plot(dfr['Time (s)'], dfr['Vertical orientation (zenith) (rad)'], ':',
        c='tab:cyan', label="OR (roll)")
    plt.plot(mk.t[1:], -mk.angle_attack.T[1][1:]+np.pi/2, c='tab:cyan',
        label="Python")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
```

```
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.subplot(224)
    plt.plot(df['Time (s)'][:654],
        df['Vertical orientation (zenith) (rad)'][:654], '--', c='tab:cyan',
        label="OR")
    plt.plot(mk.t[1:605], 604*[-mk.angle_attack.T[1][2]+np.pi/2], '-.',
        c='tab:gray', label="Angle at launch")
    plt.plot(dfr['Time (s)'][:5730],
        dfr['Vertical orientation (zenith) (rad)'][:5730], ':', c='tab:cyan',
        label="OR (roll)")
    plt.plot(mk.t[1:605], -mk.angle_attack.T[1][1:605]+np.pi/2, c='tab:cyan',
        label="Python")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.tight_layout()

inputs = {"tmax":90, "wind_speed":0, "wind_ang":0, "dry_mass":9.85,
    "wet_mass":18.554, "length":2710, "cd":0.75, "cl":0.15,
    "critical_angle":20, "hcm":710, "hcp":510, "radius":51.5,
    "thrustforce":2529, "burntime":6.04}

#Results from open rocket without roll.
df1 = pd.read_csv('csv/openrocket_0mps.csv', sep=',')
df2 = pd.read_csv('csv/openrocket_3mps90deg.csv', sep=',')
df3 = pd.read_csv('csv/openrocket_8mps90deg.csv', sep=',')
df4 = pd.read_csv('csv/openrocket_14mps90deg.csv', sep=',')
df5 = pd.read_csv('csv/openrocket_8mps0deg.csv', sep=',')
df6 = pd.read_csv('csv/openrocket_8mps270deg.csv', sep=',')

#Results from open rocket with roll.
dfr1 = pd.read_csv('csv/openrocket_0mps_roll.csv', sep=',')
dfr2 = pd.read_csv('csv/openrocket_3mps90deg_roll.csv', sep=',')
dfr3 = pd.read_csv('csv/openrocket_8mps90deg_roll.csv', sep=',')
dfr4 = pd.read_csv('csv/openrocket_14mps90deg_roll.csv', sep=',')
dfr5 = pd.read_csv('csv/openrocket_8mps0deg_roll.csv', sep=',')
dfr6 = pd.read_csv('csv/openrocket_8mps270deg_roll.csv', sep=',')

#Python, no wind.
mk1 = Rocket(launch_ang=(80, 90), **inputs)
mk1.launch()

#Python, some wind.
inputs.update(wind_speed=3, wind_ang=90)
mk2 = Rocket(launch_ang=(80, 90), **inputs)
mk2.launch()

#Python, strong wind.
inputs.update(wind_speed=8, wind_ang=90)
mk3 = Rocket(launch_ang=(80, 90), **inputs)
mk3.launch()

#Python, very strong wind.
inputs.update(wind_speed=14, wind_ang=90)
```

```
mk4 = Rocket(launch_ang=(80, 90), **inputs)
mk4.launch()

#Python, crosswind.
inputs.update(wind_speed=8, wind_ang=0)
mk5 = Rocket(launch_ang=(80, 90), **inputs)
mk5.launch()

#Python, tailwind.
inputs.update(wind_speed=8, wind_ang=270)
mk6 = Rocket(launch_ang=(80, 90), **inputs)
mk6.launch()

comparison(df1, mk1, dfr1)
plt.savefig("Comparison_no_wind")
plt.clf()
comparison(df2, mk2, dfr2)
plt.savefig("Comparison_3mps_headwind")
plt.clf()
comparison(df3, mk3, dfr3)
plt.savefig("Comparison_8mps_headwind")
plt.clf()
comparison(df4, mk4, dfr4)
plt.savefig("Comparison_14mps_headwind")
plt.clf()
comparison(df5, mk5, dfr5)
plt.savefig("Comparison_8mps_crosswind")
plt.clf()
comparison(df5, mk6, dfr6)
plt.savefig("Comparison_8mps_tailwind")
plt.clf()

plt.figure(figsize=[12.8, 9.6])
plt.subplot(331)
plt.plot(df1['Time (s)'], df1['Roll rate (rpm)'], c='tab:blue',
    label="OpenRocket (0 deg fin cant)")
plt.plot(dfr1['Time (s)'], dfr1['Roll rate (rpm)'], c='tab:green')
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("roll rate (rpm)", fontsize=14)
plt.legend(fontsize=12, bbox_to_anchor=(0,1.02,1,0.2), loc="lower left")
plt.grid()
plt.subplot(332)
plt.plot(df1['Time (s)'], df1['Air pressure (Pa)'], c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['Air pressure (Pa)'], c='tab:green',
    label="OpenRocket (3 deg fin cant)")
plt.plot(mk1.t, np.array(mk1._rho)*(8.3145*(288.15-0.0065*mk1.r.T[2])/0.029),
    c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("air pressure (Pa)", fontsize=14)
plt.legend(fontsize=12, bbox_to_anchor=(0,1.02,1,0.2), loc="lower left")
plt.grid()
plt.subplot(333)
plt.plot(df1['Time (s)'], df1['Longitudinal moment of inertia (kgm)'],
    c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['Longitudinal moment of inertia (kgm)'],
    c='tab:green')
plt.plot(mk1.t, (np.array(mk1._m)*(3*mk1._radius**2+mk1._length**2)/12),
    c="tab:red", label="Python")
```

```
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("moment of inertia (kg m^2)", fontsize=14)
plt.legend(fontsize=12, bbox_to_anchor=(0,1.02,1,0.2), loc="lower left")
plt.grid()
plt.subplot(334)
plt.plot(df1['Time (s)'], df1['CP location (cm)'], c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['CP location (cm)'], c='tab:green')
plt.plot(mk1.t, len(mk1.t)*[(mk1._length-mk1._hcp)*100], c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("CP location (cm)", fontsize=14)
plt.grid()
plt.subplot(335)
plt.plot(df1['Time (s)'], df1['CG location (cm)'], c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['CG location (cm)'], c='tab:green')
plt.plot(mk1.t, len(mk1.t)*[(mk1._length-mk1._hcm)*100], c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("CM location (cm)", fontsize=14)
plt.grid()
plt.subplot(336)
plt.plot(df1['Time (s)'], df1['Gravitational acceleration (m/s)'],
    c='tab:blue', label="0 deg fin cant")
plt.plot(dfr1['Time (s)'], dfr1['Gravitational acceleration (m/s)'],
    c='tab:green')
plt.plot(mk1.t, len(mk1.t)*[mk1._g], c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("g", fontsize=14)
plt.grid()
plt.subplot(337)
plt.plot(df1['Time (s)'], df1['Pressure drag coefficient (?)'], c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['Pressure drag coefficient (?)'],
    c='tab:green')
plt.plot(mk1.t, len(mk1.t)*[mk1._cl], c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("pressure drag coefficient", fontsize=14)
plt.grid()
plt.subplot(338)
plt.plot(df1['Time (s)'], df1['Drag coefficient (?)'], c='tab:blue')
plt.plot(dfr1['Time (s)'], dfr1['Drag coefficient (?)'], c='tab:green')
plt.plot(mk1.t, len(mk1.t)*[mk1._cd], c="tab:red")
plt.xlabel("time (s)", fontsize=14)
plt.ylabel("drag coefficient", fontsize=14)
plt.grid()
plt.tight_layout()
plt.savefig("compare_parameters")
plt.clf()
```

A.4 results.py

Results of TVC

```
import numpy as np
import matplotlib.pyplot as plt
from launchsim import Rocket
from TVC import Rocket_TVC

def trajectory(mk, tvc):
    plt.figure(figsize=[12.8, 9.6])
    plt.subplot(221)
    plt.plot(mk.t, mk.r.T[2], c='tab:red', label="Altitude")
    plt.plot(mk.t, tvc.r.T[2], ':', c='tab:red', label="Altitude (TVC)")
    plt.plot(mk.t, mk.r.T[0], c='tab:blue', label="East")
    plt.plot(mk.t, tvc.r.T[0], ':', c='tab:blue', label="East (TVC)")
    plt.plot(mk.t, mk.r.T[1], c='tab:green', label="North")
    plt.plot(mk.t, tvc.r.T[1], ':', c='tab:green', label="North (TVC)")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("distance (m)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.subplot(222)
    plt.plot(mk.t, abs(mk.v.T[0])+abs(mk.v.T[1])+abs(mk.v.T[2]),
        c='tab:orange', label="Velocity")
    plt.plot(mk.t, abs(tvc.v.T[0])+abs(tvc.v.T[1])+abs(tvc.v.T[2]), ':',
        c='tab:orange', label="Velocity (TVC)")
    plt.plot(mk.t, abs(mk.a.T[0])+abs(mk.a.T[1])+abs(mk.a.T[2]),
        c='tab:purple', label="Acceleration")
    plt.plot(mk.t, abs(tvc.a.T[0])+abs(tvc.a.T[1])+abs(tvc.a.T[2]), ':',
        c='tab:purple', label="Acceleration (TVC)")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("velocity (m/s) and acceleration (m/s^2)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left", mode="expand",
        ncol=2, fontsize=12)
    plt.grid()
    plt.subplot(223)
    plt.plot(mk.t[1:], -mk.angle_attack.T[1][1:]+np.pi/2, c='tab:cyan',
        label="Without TVC")
    plt.plot(mk.t[1:], -tvc.angle_attack.T[1][1:]+np.pi/2, ':', c='tab:cyan',
        label="With TVC")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.subplot(224)
    plt.plot(mk.t[1:605], 604*[-mk.angle_attack.T[1][2]+np.pi/2], '-.',
        c='tab:gray', label="Angle at launch")
    plt.plot(mk.t[1:605], -mk.angle_attack.T[1][1:605]+np.pi/2, c='tab:cyan',
        label="Without TVC")
    plt.plot(mk.t[1:605], -tvc.angle_attack.T[1][1:605]+np.pi/2, ':',
        c='tab:cyan', label="With TVC")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
```

```
    plt.tight_layout()

def extreme(mk, tvc, tvce1, tvce2, tvce3):
    plt.figure(figsize=[12.8, 9.6])
    plt.subplot(211)
    plt.plot(mk.t, tvc.r.T[2], c='tab:red', label="Altitude (0 m/s)")
    plt.plot(mk.t, tvc.r.T[0], c='tab:blue', label="East (0 m/s)")
    plt.plot(mk.t, tvc.r.T[1], c='tab:green', label="North (0 m/s)")
    plt.plot(mk.t, tvce1.r.T[2], '--', c='tab:red', label="Altitude (17 m/s)")
    plt.plot(mk.t, tvce1.r.T[0], '--', c='tab:blue', label="East (17 m/s)")
    plt.plot(mk.t, tvce1.r.T[1], '--', c='tab:green', label="North (17 m/s)")
    plt.plot(mk.t, tvce2.r.T[2], ':', c='tab:red', label="Altitude (21 m/s)")
    plt.plot(mk.t, tvce2.r.T[0], ':', c='tab:blue', label="East (21 m/s)")
    plt.plot(mk.t, tvce2.r.T[1], ':', c='tab:green', label="North (21 m/s)")
    plt.plot(mk.t, tvce3.r.T[2], '-.', c='tab:red', label="Altitude (25 m/s)")
    plt.plot(mk.t, tvce3.r.T[0], '-.', c='tab:blue', label="East (25 m/s)")
    plt.plot(mk.t, tvce3.r.T[1], '-.', c='tab:green', label="North (25 m/s)")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("distance (m)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,1.02,1,0.2), loc="lower left", mode="expand",
        ncol=4, fontsize=12)
    plt.grid()
    plt.subplot(223)
    plt.plot(mk.t[1:], -tvc.angle_attack.T[1][1:]+np.pi/2, c='tab:blue',
        label="0 m/s")
    plt.plot(mk.t[1:], -tvce3.angle_attack.T[1][1:]+np.pi/2, '-.',
        c='tab:purple', label="25 m/s")
    plt.plot(mk.t[1:], -tvce1.angle_attack.T[1][1:]+np.pi/2, '--',
        c='tab:green', label="17 m/s")
    plt.plot(mk.t[1:], -tvce2.angle_attack.T[1][1:]+np.pi/2, ':',
        c='tab:red', label="21 m/s")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.subplot(224)
    plt.plot(mk.t[1:605], -tvc.angle_attack.T[1][1:605]+np.pi/2, c='tab:blue',
        label="0 m/s")
    plt.plot(mk.t[1:605], -tvce3.angle_attack.T[1][1:605]+np.pi/2, '-.',
        c='tab:purple', label="25 m/s")
    plt.plot(mk.t[1:605], -tvce1.angle_attack.T[1][1:605]+np.pi/2, '--',
        c='tab:green', label="17 m/s")
    plt.plot(mk.t[1:605], 604*[-mk.angle_attack.T[1][2]+np.pi/2], c='tab:gray',
        label="Angle at launch")
    plt.plot(mk.t[1:605], -tvce2.angle_attack.T[1][1:605]+np.pi/2, ':',
        c='tab:red', label="21 m/s")
    plt.xlabel("time (s)", fontsize=14)
    plt.ylabel("vertical orientation (rad)", fontsize=14)
    plt.legend(bbox_to_anchor=(0,-0.35,1,0.2), loc="upper left", mode="expand",
        ncol=3, fontsize=12)
    plt.grid()
    plt.tight_layout()

inputs = {"tmax":90, "wind_speed":0, "wind_ang":0, "dry_mass":9.85,
    "wet_mass":18.554, "length":2710, "cd":0.75, "cl":0.15,
    "critical_angle":20, "hcm":710, "hcp":510, "radius":51.5,
    "thrustforce":2529, "burntime":6.04}
```

```
114
115 #No wind.
116 mk1 = Rocket(launch_ang=(80, 90), **inputs)
117 mk1.launch()
118 tvc1 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
119 tvc1.launch()
120
121 #Some wind.
122 inputs.update(wind_speed=3, wind_ang=90)
123 mk2 = Rocket(launch_ang=(80, 90), **inputs)
124 mk2.launch()
125 tvc2 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
126 tvc2.launch()
127
128 #Strong wind.
129 inputs.update(wind_speed=8, wind_ang=90)
130 mk3 = Rocket(launch_ang=(80, 90), **inputs)
131 mk3.launch()
132 tvc3 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
133 tvc3.launch()
134
135 #Very strong wind.
136 inputs.update(wind_speed=14, wind_ang=90)
137 mk4 = Rocket(launch_ang=(80, 90), **inputs)
138 mk4.launch()
139 tvc4 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
140 tvc4.launch()
141
142 #Crosswind.
143 inputs.update(wind_speed=8, wind_ang=0)
144 mk5 = Rocket(launch_ang=(80, 90), **inputs)
145 mk5.launch()
146 tvc5 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
147 tvc5.launch()
148
149 #Tailwind.
150 inputs.update(wind_speed=8, wind_ang=270)
151 mk6 = Rocket(launch_ang=(80, 90), **inputs)
152 mk6.launch()
153 tvc6 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
154 tvc6.launch()
155
156 #Extreme conditions in headwind
157 inputs.update(wind_speed=17, wind_ang=90)
158 tvc6 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
159 tvc6.launch()
160 inputs.update(wind_speed=21, wind_ang=90)
161 tvc7 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
162 tvc7.launch()
163 inputs.update(wind_speed=25, wind_ang=90)
164 tvc8 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
165 tvc8.launch()
166
167 #Extreme conditions in crosswind
168 inputs.update(wind_speed=17, wind_ang=0)
169 tvc9 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
170 tvc9.launch()
171 inputs.update(wind_speed=21, wind_ang=0)
```

```
tvc10 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
tvc10.launch()
inputs.update(wind_speed=25, wind_ang=0)
tvc11 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
tvc11.launch()

#Extreme conditions in tailwind
inputs.update(wind_speed=17, wind_ang=270)
tvc12 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
tvc12.launch()
inputs.update(wind_speed=21, wind_ang=270)
tvc13 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
tvc13.launch()
inputs.update(wind_speed=25, wind_ang=270)
tvc14 = Rocket_TVC(80, 600, 30, launch_ang=(80, 90), **inputs)
tvc14.launch()

trajectory(mk1, tvc1)
plt.savefig("tvc_no_wind")
plt.clf()

trajectory(mk2, tvc2)
plt.savefig("tvc_3mps_headwind")
plt.clf()
trajectory(mk3, tvc3)
plt.savefig("tvc_8mps_headwind")
plt.clf()
trajectory(mk4, tvc4)
plt.savefig("tvc_14mps_headwind")
plt.clf()
trajectory(mk5, tvc5)
plt.savefig("tvc_8mps_crosswind")
plt.clf()
trajectory(mk6, tvc6)
plt.savefig("tvc_8mps_tailwind")
plt.clf()

extreme(mk1, tvc1, tvc6, tvc7, tvc8)
plt.savefig("tvc_extreme_headwind")
plt.clf()
extreme(mk1, tvc1, tvc9, tvc10, tvc11)
plt.savefig("tvc_extreme_crosswind")
plt.clf()
extreme(mk1, tvc1, tvc12, tvc13, tvc14)
plt.savefig("tvc_extreme_tailwind")
plt.clf()

plt.figure(figsize=[9.6, 7.2])
plt.subplot(211)
plt.plot(mk2.t[:1200], mk2.fthrust[:1200, 0], '-', c='tab:blue', label="thrust")
plt.plot(tvc2.t[:1200], tvc2.fthrust[:1200, 0], '--', c='tab:blue', label="thrust
    (TVC)")
plt.plot(mk2.t[:1200], mk2.flift[:1200, 0], '-', c='tab:green', label="lift")
plt.plot(tvc2.t[:1200], tvc2.flift[:1200, 0], '--', c='tab:green', label="lift (
    TVC)")
plt.plot(mk2.t[:1200], mk2.fdrag[:1200, 0], '-', c='tab:red', label="drag")
plt.plot(tvc2.t[:1200], tvc2.fdrag[:1200, 0], '--', c='tab:red', label="drag (TVC
    )")
```

```
227 plt.ylabel("force (N) in x-axis", fontsize=14)
228 plt.legend(fontsize=12, bbox_to_anchor=(0,1.02,1,0.2), ncol=6, loc="lower left")
229 plt.grid()
230 plt.subplot(212)
231 plt.plot(mk2.t[:1200], mk2.fthrust[:1200, 2], '-', c='tab:blue')
232 plt.plot(mk2.t[:1200], mk2.fdrag[:1200, 2], '-', c='tab:red')
233 plt.plot(mk2.t[:1200], mk2.flift[:1200, 2], '-', c='tab:green')
234 plt.plot(tvc2.t[:1200], tvc2.fthrust[:1200, 2], '--', c='tab:blue')
235 plt.plot(tvc2.t[:1200], tvc2.fdrag[:1200, 2], '--', c='tab:red')
236 plt.plot(tvc2.t[:1200], tvc2.flift[:1200, 2], '--', c='tab:green')
237 plt.xlabel("time (s)", fontsize=14)
238 plt.ylabel("force (N) in z-axis", fontsize=14)
239 plt.grid()
240 plt.tight_layout()
241 plt.savefig("forces_3mps")
242 plt.clf()
```