VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Control Unit For a Model Rocket

Řídící jednotka modelu rakety

Jakub Výmola

Bachelor Thesis

Supervisor: Ing. Petr Olivka, Ph.D.

Ostrava, 2021

Abstrakt

Při stále se zmenšující velikosti výpočetní techniky a při současném snižování ceny integrovaných obvodů a senzorů, se objevují další a další oblasti, kde se aktivní řízení postupně stává možným. Jednou z takových oblastí je raketové modelářství. Přesto, že jako hobby sahá svými počátky až do 50. let 20. století, aktivně stabilizované modely raket se začaly objevovat teprve nedávno.

Dostupných je několik publikací, zabývajících se modelováním řídícího systému raketových modelů. Na druhou stranu zde je málo prací, které by se orientovaly na návrh řídící jednotky, na výběr hardwaru a na implementaci softwaru.

Tato práce se zaměřuje na výběr vhodného mikrokontroléru a periferií, které budou základem řídící jednotky, a na tvorbu potřebného softwarového řešení. Vybraný mikrokontrolér nahradí mikrokontrolér Atmega328, který tvůrci často používají v řídících jednotkách na některé desce z řady Arduino. Vybrané senzory budou testovány za letu na modelu rakety a vizualizace a analýza letových dat bude součástí textu.

Klíčová slova

řídící jednotka letu, model rakety, aktivní stabilizace, mikrokontrolér

Abstract

With the ever increasing density of computational power and with simultaneous decrease of cost of integrated circuits and sensors, there are fields, where active control is just becoming possible. One such field is model rocketry. As a hobby, it has been around for decades, dating back all the way to the 1950s. Though actively controlled model rockets have just recently started appearing.

While there are publications on the topic of modeling the control system of model rockets, there is little work published about a selection of the hardware for control units, their design and software implementaion.

This work focuses on selecting the most appropriate microcontroller and peripherals for this control unit and on implementing the necessary software. The selected microcontroller will replace the Atmega328 microcontroller, which is commonly used on the control unit as a part of an Arduino board. Moreover, the selected sensors will be subjected to an actual flight of a model rocket and data visualization and analysis will be a part of the text.

Keywords

flight control unit, model rocket, active stabilization, microcontroller

Acknowledgement

I would like to thank all, who supported this work. To my supervisor, Ing. Petr Olivka, Ph.D., who was a great help with hardware selection and with consultations; to my dad, with whom I have been consulting the construction of the rockets; and to everyone else, who has helped me in any way.

Contents

List of symbols and abbreviations

AHRS	–	Attitude and Heading Reference System
API	–	Application Programming Interface
ARM	–	Advanced RISC Machine
CAA	–	Civil Aviation Authority
CFD	–	Computational Fluid Dynamics
CG	–	Center of Gravity/Center of Mass
CLI	–	Command Line Interface
CP	–	Center of (Aerodynamic) Pressure
EEPROM	–	Electrically Erasable Programmable Read-only Memory
ENU	–	East, North, Up
EU	–	European Union
ESA	–	European Space Agency
FPU	–	Floating-Point Unit
GND	–	Ground
GPS	–	Global Positioning System
GUI	–	Graphical User Interface
HPR	–	High Power Rocketry
PID	–	Proportional Integral Derivate
I^2C	–	Inter-Integrated Circuit
IDE	–	Integrated Development Environment
IMU	–	Inertial Measurement Unit
I/O	–	Input/Output
IoT	–	Internet of Things
LED	–	Light-Emmiting Diode
LiPo	–	Lithium Polymer
MARG	–	Magnetic, Angular Rate and Gravity

MCU	–	Microcontroller Unit
MEMS	–	Microelectromechanical System
NED	–	North, East, Down
OS	–	Operating System
PCB	–	Printed Circuit Board
PWM	–	Pulse Width Modulation
RAM	–	Random Access Memory
RC	–	Radio-Controlled
RGB	–	Red, Green, Blue
RISC	–	Reduced Instruction Set Computer
RTOS	–	Real-Time Operating System
SD	–	Secure Digital
SMD	–	Surface-Mount Device
SPI	–	Serial Peripheral Interface
SRAM	–	Static Random Acess Memory
THT	–	Through-Hole Technology
UART	–	Universal Asynchronous Receiver-Transmitter
UAV	–	Unmanned Aerial Vehicle
US	–	United States
USB	–	Universal Serial Bus
USART	–	Universal Synchronous/Asynchronous Receiver-Transmitter

List of Figures

List of Tables

Chapter 1

Introduction

The following text deals thoroughly with the design process of a control unit for an actively controlled model rocket. It starts with model rocket theory and a quick overview of control theory applicable to these models. Then the text continues with necessary information on how to correctly use the most important sensor, the Inertial Measurement Unit, or IMU for short, to provide correct input to a control algorithm. A big part of this text is a selection of hardware. First is a selection of a microcontroller board and a comparison of several candidates. Next is a selection of necessary sensors and peripherals. After an overview of developed software and used tools, the text ends with presenting data from powered flights and with analysis of this data.

Most attention is given to the the selection and usage of the electronics and to the data visualization. There is a comparison of several microcontrollers in terms of their speed, specifications and their capabilities. Following the comparison, there is a list of important peripherals ant their alternatives. In terms of software implementation, there is not that much information directly in the text, but all the source codes are available in the appendix of this work.

During the course of this project, two control units were designed and build. The first one will be referred to as a data-logging unit and the second one will be called a flight control unit.

The data-logging unit is based on Arduino Nano and it has served as a pathfinder solution for the more complex flight control unit, which is built around a Teensy 4.0 board. Both units are based on easily available Through-hole technology components, which makes them relatively cheap and easy to build.

In addition to the control units, two rockets were designed and built. The designs of these rockets is described in appendix B, as this is not the primary concern of this work. The first rocket will be referred to as a data-logging rocket and the second will be called an actively controlled rocket. The data-logging rocket does not have any way of actively controlling itself. It has been flown several times with the data-logging unit onboard and it has collected lots of data. The actively controlled rocket with its flight control unit is yet to fly with power from a rocket motor, but the control unit was tested to correctly control the servomotors of the rocket.

Chapter 2

Model rocketry regulations

In the US, National Association of Rocketry defines a safety code for model rockets on a national level [12]. Despite my very best effort, I have not been able to find a legislation that would regulate model rocket flights on the territory of the Czech Republic.

There is currently an EU regulation, which defines conditions for flying with drones, RC planes and helicopters – UAVs. This regulation is effective in Czechia as well [13, 14, 15] and it defines the maximum altitude for flying with UAVs in the relevant category as 120 m above ground.

However, in an e-mail correspondence with the Czech Civil Aviation Authority, it has been confirmed to me, that the aforementioned regulation does not apply to model rockets. I have sent additional e-mails to the Czech Space Office [16], Czech Model Rocketry Club [17] and to ESA. Out of these, only a member of the Model Rocketry Club replied to me with some relevant information, but even this person was unable to direct me to a specific policy, since the Club is not concerned with the work of individuals.

The e-mail correspondence with the member of the Model rocketry club is published as a part of the digital appendix of this text. I am only publishing the parts, which I have been granted a permission to share – the permission was given to me through an e-mail. I have also removed any personal information from the published correspondence.

In the end, I feel like the best I can do is to reason and justify the safety of the performed flights myself. The baseline for this reasoning was the minimum legal altitude of general aviation aircrafts, which must be at least 150 m above the highest ground object [18], and rescue helicopters, which turns out to be 30 m above ground level [19]. However, this 30 m altitude limit of rescue helicopters is in collision even with legally operated drones.

Even though some of my flights were to an altitude of 210 m above ground, and these flight will be presented in the text, the rocket will stay above the critical 150 m only for about 10–15 s. Therefore I conclude, that as long as all other safety measures are met, the safety code is complied with, there is no air traffic in sight and the flight does not take place in a protected zone around an airport, the flight is safe and does not violate any regulations.

Chapter 3

Basic model rocket theory

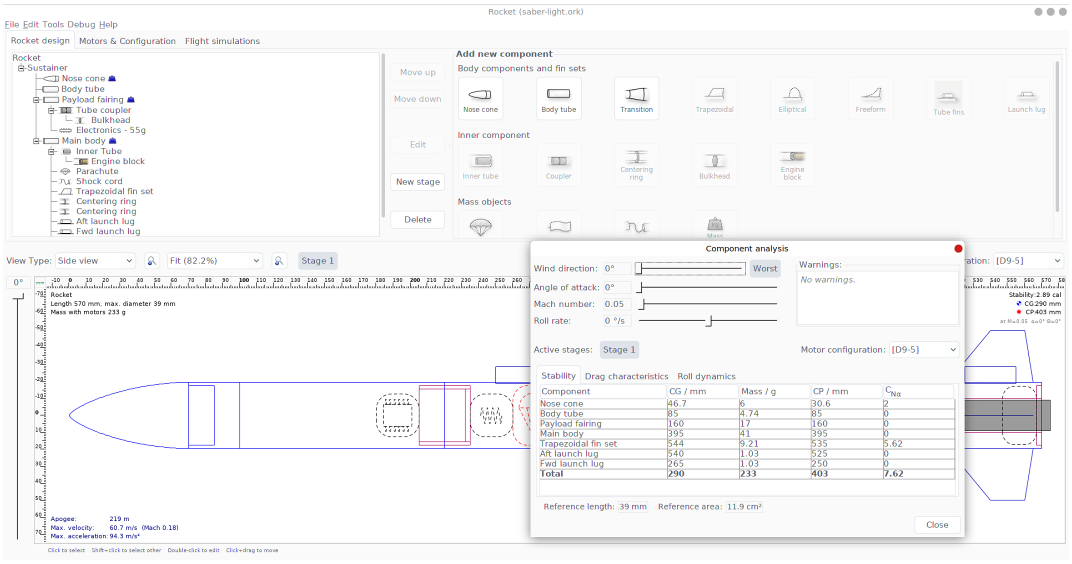
In this chapter, I will describe only the theory that is truly necessary for the rest of this work. Most of the terms mentioned in this section will be needed throughout the rest of the text. I want to highlight certain aspects of model rockets, which guided the design decisions and I hope, that with this knowledge, the justification of the final choices will be more clear.

3.1 Stability

Conventional rocket models, especially low-power ones, usually do not have any form of active control. There is no logic to keep the rocket on an intended flight path. These rockets fly straight, because they are passively stable.

There are two important physical points on the rocket, whose relative position defines the passive stability. These points are the center of mass (center of gravity, CG) and the center of pressure (CP). Loosely speaking, CP is the center of cross-sectional area of the rocket. In reality, the analysis is much more complicated and either a rocket model simulator (section 3.1.1) or a CFD software can be used to solve this problem. CG is a hypothetical point around which the force of gravity appears to act.

Passive stability is the primary concern when designing a model rocket. For the rocket to be stable, CP must be located behind CG [20]. The distance between CG and CP divided by the diameter of the rocket's body is called static margin, and its value is positive when the center of pressure is behind the center of mass, i.e. when the rocket is passively stable. Since the static margin is a unitless number, its value is given in calibers (cal.). A common value of static margin is between 1 and 2 cal. [6]. When static margin is negative, the rocket is passively unstable and it wants to fly engine first, when 0–1 caliber, the rocket is not stable enough and it can veer heavily off course. Value above 2 cal. is generally not bad, but the rocket tends to curve more into the wind and hence it can drift further from the launch pad.



Rocket
- Sustainer
  - Nose cone
  - Body tube
  - Payload fairing
    - Tube coupler
      - Bulkhead
    - Electronics - 55g
  - Main body
    - Inner Tube
      - Engine block
    - Parachute
    - Shock cord
    - Trapezoidal fin set
    - Centering ring
    - Centering ring
    - Aft launch lug
    - Fwd launch lug

Rocket
Length 570 mm, max. diameter 39 mm
Mass with motors 233 g

Stability: 2.89 cal
CG:290 mm
CP:403 mm
at M=0.05 α=0° θ=0°

Apogee: 219 m
Max. velocity: 60.7 m/s (Mach 0.18)
Max. acceleration: 94.3 m/s²

Component analysis

Wind direction: 0° Worst
Angle of attack: 0°
Mach number: 0.05
Roll rate: 0 °/s
Warnings: *No warnings.*
Active stages: Stage 1
Motor configuration: [D9-5]

Stability | Drag characteristics | Roll dynamics

Component	CG / mm	Mass / g	CP / mm	$C_{N\alpha}$
Nose cone	46.7	6	30.6	2
Body tube	85	4.74	85	0
Payload fairing	160	17	160	0
Main body	395	41	395	0
Trapezoidal fin set	544	9.21	535	5.62
Aft launch lug	540	1.03	525	0
Fwd launch lug	265	1.03	250	0
Total	**290**	**233**	**403**	**7.62**

Reference length: 39 mm Reference area: 11.9 cm²

Close

Figure 3.1: OpenRocket simulation software [21]

3.1.1 Rocket model simulators

There are two very good and very well-known programs for designing and simulating model rockets. These are OpenRocket [21] and RockSim [22], with the former one being free and open source and the latter one being paid. I have been using OpenRocket, depicted in figure 3.1, for all the rocket designs and flight simulations associated with this project. This software has consistently produced very accurate estimates of the center of mass and the center of pressure as well as the maximum flight altitude (apogee).

3.2 Rocket motors

Second most important area that drives the design process is the rocket motor selection. The selection itself is constrained by the intention of the rocket, its projected mass and the intended altitude.

The rocket motor performance is described by their thrust curve, which is a plot of the thrust force vs. time, visible in figure 3.2. The total impulse of the motor is the area under the thrust curve and the unit is N·s. Based on the total impulse, the rocket motor is assigned into a class, which is labeled by a letter. Motors with the total impulse in the range of 1.26–2.50 N·s belong to the class A. The following classes are labeled with a consecutive letters of the alphabet and the total impulse is always double, compared to the previous class. The letter designation is usually written on the casing of rocket motors for a visible indication of the motor's power.

On the casing, there are two additional numbers written. The first number specifies the average thrust of the motor. The second one is the ejection charge delay.

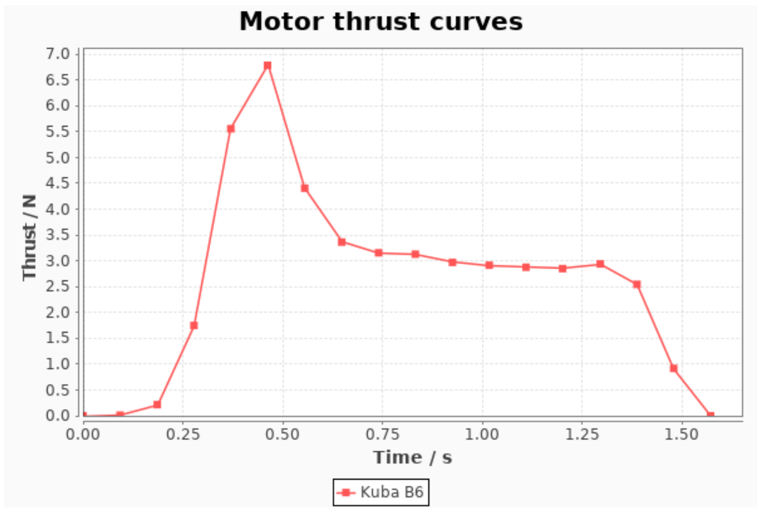


Figure 3.2: Thrust curve of B6-2 motor, which I have measured using Arduino and a load cell

An example of a motor label is B6-2, which is the motor used in my very first flight tests. This motor burns for about 1.5 s and has peak thrust of about 7 N, as seen in figure 3.2. The ejection charge ignites after 2 s after burnout in this motor.

The strongest motor one can use with model rocket is the G class motor with a maximum total impulse of 160 N·s. Above that, the rocket is no longer considered to be a model rocket, but rather a mid- or a high-power rocket and a special license is needed to fly such rocket [23, 12].

I must mention that rocket motors are dangerous pyrotechnic devices and should be handled with maximum care and caution and always by an adult. (Class C and below can be handed under a supervision of an adult.)

3.3 Rocket parts

Now it is important to take a look onto the rocket as a whole. Usually, the rocket is composed of a removable nosecone, body tube with fins, motor mount, rocket motor, shock cord and a parachute.

3.4 Flight sequence

If everything goes well, the flight looks like the following:

1. Rocket motor is inserted into the motor mount and secured.
2. Parachute is packed, pushed into the body tube from the front and the body tube is closed by partially sliding the nosecone into it – again, from the front.
3. The rocket is slid onto a guide rail, which will support the rocket before liftoff.

4. A loud countdown is given. The rocket motor is ignited and the rocket starts moving up.
5. During the initial phase of the flight, the rocket is sliding along a guide rail. This rail must be sufficiently long to support the rocket while it is picking up speed.
6. When the rocket leaves the guide rail, it is moving at a velocity at which the fins can provide enough aerodynamic force to stabilize the rocket.
7. After few seconds, the motor burns out and the delay charge starts burning.
8. After a defined period, the delay charge ignites the ejection change, which pressurizes the inside of the body tube. This pressure shoots out the nosecone and pushes out the parachute. This event hopefully occurs when the rocket is at the highest altitude.
9. The body tube and the nosecone are tied together using the shock cord, which is just a long piece of string. The parachute is attached to the shock cord as well.
10. The body tube and nosecone slowly descend together under the parachute and they safely touch down, ready to be flown again.

Chapter 4

Control theory for model rockets

4.1 Coordinate frame, transformations and axes definition

There is a convention for how the following coordinate frames and transformations are defined, when used with aircrafts [24]. Since rockets operate at a orientation which is different by 90°, relative to aircrafts, I will swap some axes to make it easier to work with.

4.1.1 Inertial frame

The coordinate frame, where the axes are Earth-fixed is called inertial frame. The z axis is aligned with magnetic North, y axis is aligned with East and x axis is aligned with the direction of gravitational force. These axes are used as a set of unmoving reference.

4.1.2 Body frame

The coordinate system where the axes are aligned with the sensor axes is called body frame.

4.1.3 Transformations between frames

To get the body frame from the inertial frame, a set of three transformations is needed.

Rotation around inertial frame z axis by an angle ϕ is called yaw. This creates an intermediate frame called vehicle-1 frame. Rotation around y axis of the vehicle-1 frame by an angle θ is called pitch, resulting in an intermediate vehicle-2 frame. Finally, rotation around vehicle-2 x axis by an angle ψ is called roll and this results in the body coordinate frame [24].

The ψ, θ and ϕ are called Euler angles and will be described in section 4.7.

The body frame x axis will interchangeably be called long axis or roll axis. Body frame y axis and z axis will be called pitch and yaw axis respectively or either of them as a short axis. This is to make the naming scheme more intuitive. The rocket motor will accelerate the rocket along the body frame x axis.

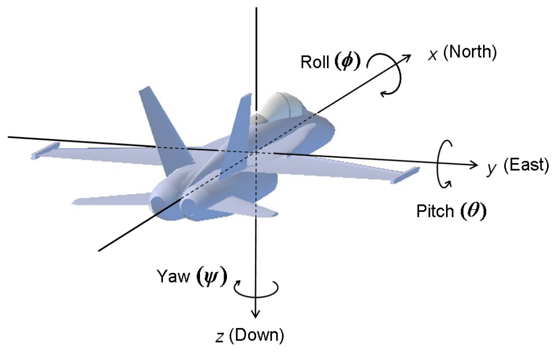


Figure 4.1: Inertial frame axes used with aircrafts [24]. By the definitions from this section, down would be x axis, North would be z axis and East would be y axis. Yaw would be roll, roll would be yaw. The angles stay the same.

4.2 Control unit requirements

The main requirement of this project is that the model rocket must be able to fly straight up. This is usually achievable, to some extent, even by uncontrolled model rockets. The flight path of these uncontrolled rockets is however highly susceptible to winds and they usually pick up some rotation along the long axis as there is very little stabilizing force acting on the rocket in this axis and the fins are usually not aligned perfectly.

To fulfill the project's requirement, the control unit must be able to do two things in real-time. First, it must be able to tell the rocket's orientation relative to the inertial coordinate system. Secondly, it must be able to provide control inputs to change the rocket's orientation. By implementing these two capabilities, a closed loop control system is created. In other words, the control unit can estimate the difference between an actual and an intended orientation and it can issue commands to minimize this difference.

Additional requirement is that the control unit must be able to tell in which phase of the flight it is. It must be able to detect certain critical events of the flight, such as lift off, burnout, reaching maximum altitude and landing.

4.3 Control mechanisms for model rockets

There are multiple alternatives on how to provide active control for the model rocket. Probably the most common one among model rocket hobbyists is to have actuated canards [1]. Additional approaches include an engine gimbal, actuated fins and a reaction wheel.

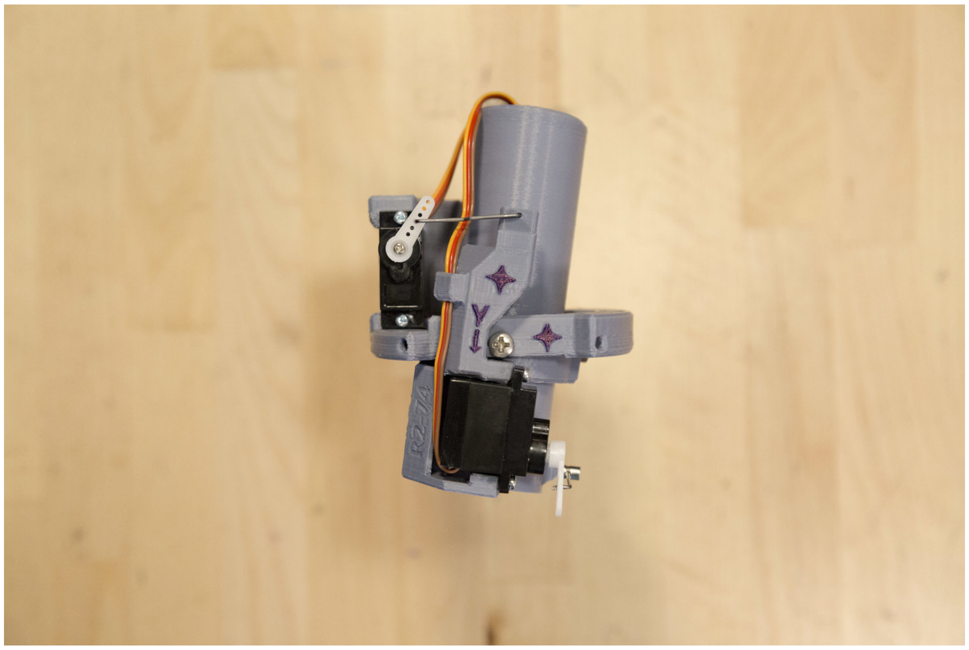


Figure 4.2: 3d-printed gimbal-capable motor mount [25]

4.3.1 Engine gimbal

Engine gimbal is perhaps the most common control mechanism for orbital rockets. In short, the rocket controls itself by angling (gimballing) the entire engine. This allows large rockets to fly straight and remain under control even if they are passively unstable. Thanks to this, large rockets don't need fins and they remain controllable in a vacuum of space.

This is less practical for model rockets, as the motor only fires for a short period of time during ascend. When the motor burns out, the model rocket looses the ability to control itself.

4.3.2 Reaction wheel

Although used mainly on satellites and spacecrafts outside of atmosphere, reaction wheel could theoretically be used on a model rocket as well. The control systems produces action by speeding up or slowing down a rotation of a disk, which, as a reaction, applies force to the entire spacecraft in an opposite direction.

This has little application in model rocketry though, as the only reasonable control input this mechanism can provide is in the roll axis. Yawing and pitching would probably require too much torque for this to be viable solution. However, this might be used in a High-power rocket in a combination with an engine gimbal [26].

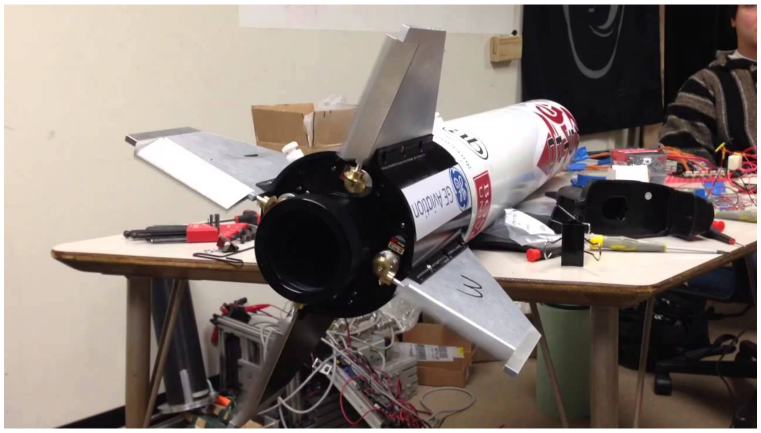


Figure 4.3: A variant of actuated fins. In this case only the trailing edge of the fin is moving. This reduces the issue with high sensitivity of the system [27].

4.3.3 Actuated fins

Actuated fins are located in the back of the model rocket. They provide both passive stability and active control at once. On the other hand, the electronics and servomotors usually sit at the front of the rocket, so there must be long cables from the front of the rocket to transfer the control inputs to the fins. They also make the analysis of the passive stability much harder, because the CP moves around as the fins are being angled. Perhaps the highest disadvantage is however, that in order to provide high enough static margins, the fins must be quite large. This makes the control mechanism more sensitive and harder to use. A workaround of this problem is visible in figure 4.3, where only the trailing edge of the fins is actuated.

4.3.4 Actuated canards

Canards are small wings located in front of the CG. They are sometimes used on airplanes, but they can be used with a controlled rocket as well. They are easy to use with model rockets, as they are physically located close to the electronics, which usually sits at the front of the rocket.

Since they are mounted at at the front of the rocket, they inherently push the CP forward, hence larger passive fins in the back of the rocket are required to compensate for the canards.

I have chosen this mechanism and the rocket design can be seen in appendix B.2.

4.4 PID controller

PID controller is a control loop mechanism that uses feedback from sensors to calculate an error between the measured process value and a setpoint. PID stands for proportional-integral-derivative – those are the three terms which define the output of this controller. Such a controller can be used in a cruise control in a car for example.

The proportional term is the error value multiplied by proportional gain Kp. Integral term is the integration of the error values over time, multiplied by integral gain Ki. In pseudo-code, integral term can be defined as: `integral = integral + error * delta_t * Ki`. Derivative term is the rate of change of the error (or of the measurement), multiplied by derivative gain Kd. Derivative term in pseudo-code can be written as: `derivative = (error - previous_error) / delta_t * Kp`. Such controller is enough to control model rocket in flight [10].

The most difficult aspect of using the PID controller is selection of the individual gains, so called tuning. When gains are selected wrong, the controller may make the system even more unstable. The tuning can be either done experimentally or analytically. This is however far beyond the scope of this project, since the gains heavily depend on the speed of the rocket, which changes quickly during flight, making the analysis very complex.

In listing 4.1, the derivative term also contains a low-pass filter to filter out high frequency noise. Furthermore, there is an anti-windup check to prevent the integral term from becoming saturated. The saturation would prevent the controller from working properly [28].

```cpp
float PIDControler::Update(float setPoint, float measurement, float deltaT) {
   float error = measurement - setPoint;
   float proportional = this->Kp * error;
   float limMaxInt = max(0, this->limMax - proportional);
   float limMinInt = min(0, this->limMin - proportional);

   this->integrator = this->integrator + 0.5 * this->Ki * deltaT * (error + this->
       prevError);
   this->integrator = min(limMaxInt, max(limMinInt, this->integrator)); //Integral
        anti-windup
   this->differentiator = (2 * this->Kd * (measurement - this->prevMeasurement)
   + (2 * this->tau - deltaT) * this->differentiator) / (2 * this->tau + deltaT);

   float output = proportional + this->integrator + this->differentiator;
   output = max(this->limMin, min(this->limMax, output)); //Output clamping
   this->prevError = error;
   this->prevMeasurement = measurement;
   return output;
}
```

Listing 4.1: Implementation of the update method of a PID controller in C++

4.5 Attitude estimation

To get the input to the control system, the control unit must be able to estimate rocket's current orientation. The inertial measurement unit (IMU) is used for this task. It is a combination of sensors which provide enough information to the control unit to calculate the body orientation.

The IMU consists of multiple MEMS devices. These include a set of three accelerometers (one for each axis) and a set of thee gyroscopes. In this configuration, the control unit is able to estimate the body orientation relative to the ground plane. However, the magnetic reference is missing, i.e. the IMU cannot provide an absolute information about a compass direction. Hence, the heading is estimated only in relative terms from the gyroscope measurements. The absence of magnetic reference causes the estimated orientation to drift around the inertial frame x axis over a period of time and while experiencing vibrations and/or rapid rotations, as will be later seen in figure 5.3.

4.5.1 Magnetometer

To provide a heading reference, the IMU can be equipped with a set of three magnetometers, which measure the strength of Earth's magnetic fields to provide a fixed point in the horizontal plane. This stops the measured orientation to drift around the inertial frame x axis.

In some literature, for example in the Magdwick's study [2], a chip which consists only of accelerometers and gyroscopes is called an IMU, while a chip that also has a set of magnetometers is called MARG (Magnetic, Angular Rate and Gravity) sensor. In the following text, I will refer to both of these configurations exclusively as IMU, and I mention the presence or absence of magnetometers only where it is necessary.

4.5.2 AHRS

Another commonly used term in this context is AHRS (attitude and heading reference system), which is very similar to an IMU or MARG, but on top of the raw values, the chip can directly output the estimated orientation.

4.6 Sensor fusion

To get an absolute orientation of the body frame, relative to the inertial frame, raw data from different sensors of the IMU must be combined together. This is called sensor fusion. There are many algorithms that do this with a varying degree of precision and speed.

When the sensor is standing still on the ground, the accelerometers essentially output the components of a gravity vector. This is enough to estimate the yaw and pitch angles, but roll angle is impossible to obtain just by reading the accelerometer measurements.

To get the roll angle, a set of three gyroscopes must be added. These are sensors that measure rotation speeds. This is still not enough to get an absolute heading, i.e. a compass direction, but if one does not need use a GPS in their control system, the compass direction is generally not necessary. Gyroscopes also greatly improve the convergence times of the yaw and pitch angle estimates [8].

Each type of sensor has its inherit defects which the other sensor(s) try to compensate [8]. Common algorithms used to fuse sensor data together with a decent speed and precision are Madgwick's [2] and Mahony's [3] sensor fusion algorithms. Both output the estimated orientation in a form of a quaternion, which can be either used directly or converted to Euler angles. Many other algorithms are available, such as modified Kalman filters [29, 30].

Both, Madgwick's and Mahony's algorithms, support magnetometer input. Having the magnetic reference is not really needed for the rocket model as long as it does not need to use GPS. If the GPS is used, it is necessary to know the position of the true North, which can be obtained by subtracting local magnetic declination [31] from the estimated magnetic heading.

I will be using the magnetometer in the rest of this work.

4.6.1 Comparison of Madgwick's and Mahony's algorithm

There are important differences between Madgwick's and Mahony's sensor fusion algorithms. Generally, Mahony's is reported to be little faster, while Madgwick's is more accurate [9].

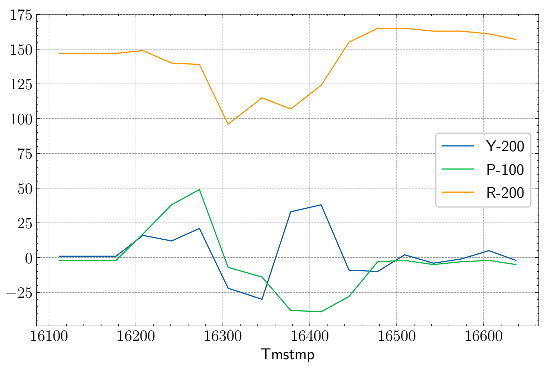
In have confirmed this in my tests as well. Figures 4.4 and 4.5 show the estimated orientation along with the measured acceleration. In these tests, I have rapidly moved the sensor in one direction without changing its tilt. It is clearly visible how Madgwick's filter estimates the Euler angles correctly, i.e. they all remain constant for the duration of the test, while Mahony's generates a phantom pitch and yaw of about 45°. Due to this behavior, I will be using Madgwick's filter in the rest of this project.

Mahony's filter has the advantage of slightly higher speed and a lower memory footprint. A control unit based on Arduino Nano, and outputting data to an SD card, was running at 49.8 Hz with Madgwick's sensor fusion algorithm and at 51.9 Hz with Mahony's.

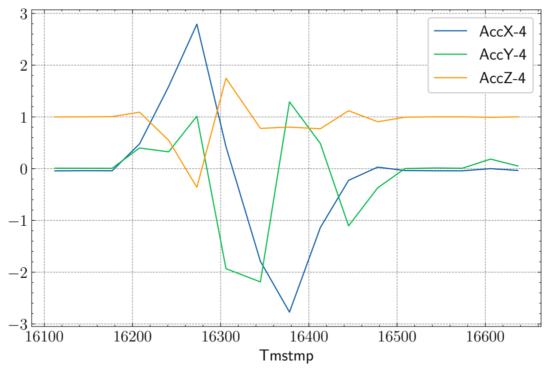
4.7 Euler angles

Orbital rockets need to fly in an arc over the horizon to reach orbit. Model rockets usually fly straight up and down. This greatly simplifies their navigation and control, because the Earth can be substituted by a flat surface. Furthermore, the range of usual orientations of the model rocket is fairly narrow.

To represent any orientation in space, a set of three angles is enough. Those angles are called yaw angle, pitch angle and roll angle, the Euler angles [24]. Yaw, pitch and roll are a set of transfor-

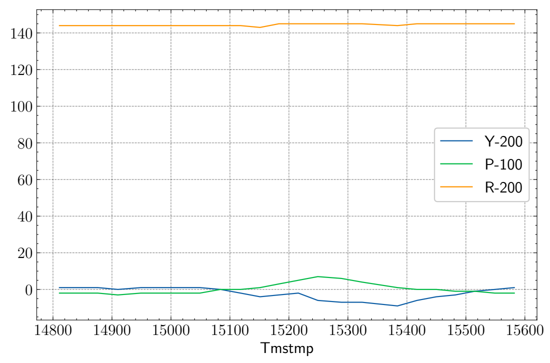


(a) Yaw, pitch, roll

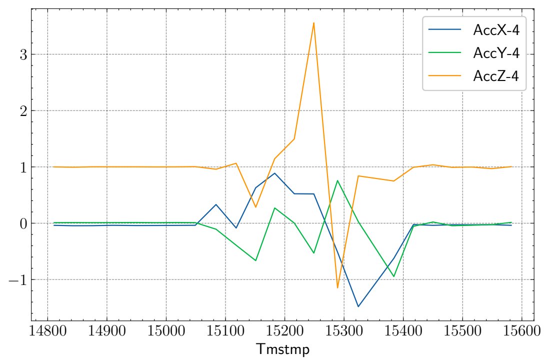


(b) Acceleration

Figure 4.4: Estimated orientation by Mahony's filter vs. measured acceleration



(a) Yaw, pitch, roll



(b) Acceleration

Figure 4.5: Estimated orientation by Madgwick's filter vs. measured acceleration

mations that are required to obtain the body frame from the inertial frame. These transformations are represented by rotational matrices. By convention, the range of angles for these rotations go from $-180°$ to $180°$ for yaw and roll and from $-90°$ to $90°$ for pitch.

The Euler angles will further simplify the whole problem of controlling the model rocket in flight, as the rocket can correct the yaw, pitch and roll angles individually and independently of each other. However, the moment of inertia of the rocket when rotating around its long axis (rolling) is naturally much smaller then for the rotations around the other two axes, hence the controller must be much less sensitive in the roll axis. There will be an independent PID controller with independent gains for each axis. Each controller will be given one Euler angle as its input and the deflection of individual fins will be the sum of the outputs from relevant controllers.

The Magdwick's and Mahony's algorithms both output the orientation estimate in a form of quaternion. Sample implementation for transforming the quaternion into the Euler angles written in C++ is the following:

```
float qw = q[0], qx = q[1], qy = q[2], qz = q[3];

yaw = atan2(2.0 * (qw * qx + qy * qz), 1.0 - 2.0 * (qx * qx + qy * qy));
pitch = asin(2.0 * (qw * qy - qx * qz));
roll = atan2(2.0 * (qx * qy + qw * qz), 1.0 - 2.0 * (qy * qy + qz * qz));
```

Listing 4.2: Conversion of quaternion to Euler angles

In this code snippet, `q` is an array with 4 float elements which hold the initial quaternion.

Chapter 5

Using the IMU

In this chapter, I want to focus on the actual IMU, as it is pretty much the heart of the control system. Without it, the rocket has no hope of successfully controlling itself. There are some crucial requirements, that must be met: the IMU must support the range of operating conditions (this will be more elaborated in section 7.1) and it must be fast and accurate.

5.1 IMU calibration

The sensors in the IMU are not perfect and they need to be calibrated. As stated in section 4.6, I will be using the magnetometer both in the data-logging unit as well as in the flight control unit. Magnetometer is possibly the most extreme example of the necessity to calibrate, since the raw measurements are usually unusable. But even the gyroscope and accelerometer both need the be calibrated since they all come with built-in biases – factory errors.

The calibration consists of acquiring needed calibration values and using those values in the control unit's code to correctly transform the measured values to a calibrated output.

5.1.1 Gyroscope calibration

Easiest to calibrate are the gyroscopes. What is needed is a series of measurements with the sensor at rest. The bias vector is simply the average of measured values in each axis. This vector then needs to be subtracted in software from every measurement, to remove the factory error.

Table 5.1 shows an example of factory errors. The raw biases were measured using one of my IMUs and they would be different with another one. The biases in terms of the rotation speed are obtained by multiplying the raw values by a factor of 500/32 768. This factor is based on the currently configured sensitivity, which was $500^\circ\,\mathrm{s}^{-1}$ in this case. The gyroscopes, as all the other sensors, should be recalibrated after changing their sensitivity.

Table 5.1: Example factory errors of a particular IMU

Axis	Raw bias	Bias in terms of rotation speed
x	56.7	$0.86^\circ\,\mathrm{s}^{-1}$
y	-204.2	$-3.12^\circ\,\mathrm{s}^{-1}$
z	38.4	$0.59^\circ\,\mathrm{s}^{-1}$

5.1.2 Accelerometer calibration

The set of accelerometers requires more complex calibration. In principle, the raw measurements of these sensors are not only biased, but also distorted. These inaccuracies are different in each axis.

To achieve a basic calibration, we need to find the maximum and minimum measured value in each axis. The bias vector $\vec{b}$ is then determined as

$$b_i = \frac{v_{imax} + v_{imin}}{2} \tag{5.1}$$

where i stands for x, y and z axes, v_{imin} and v_{imax} are the minimum and maximum measured values in the corresponding axis.

Equally, the correction factor vector $\vec{f}$ for the distortion error is determined as

$$f_i = \frac{n}{v_{max} - v_{min}} \tag{5.2}$$

where n is the desired vector norm – this value depends on the filtering algorithm used, but for Madgwick's and Mahony's algorithms, n can be an arbitrary number. It is important, that the value of n is the same in each axis.

To get the calibrated vector $\vec{u}$ from the raw measurement $\vec{v}$, following formula is used:

$$u_i = (v_i - b_i)f_i \tag{5.3}$$

5.1.3 Magnetometer calibration

The magnetometer calibration is very similar to the calibration of the accelerometers. Although one can probably get away with using the IMU without calibrating the accelerometers, it is absolutely crucial to calibrate the magnetometers. The factory error is much greater with these sensors and they are also much more sensitive to the outer environment. Radio signals or even home power lines greatly affect the measurements, hence they should ideally be re-calibrated before every use.

In the magnetometer, there are two mechanisms that contribute to the errors. One mechanism is a so-called hard iron distortion. This is produced by an object which is creating magnetic fields, such as a live circuit or a magnet. If this object is physically attached to the rocket, it will cause a constant and permanent bias of the measurements which needs to be compensated for.

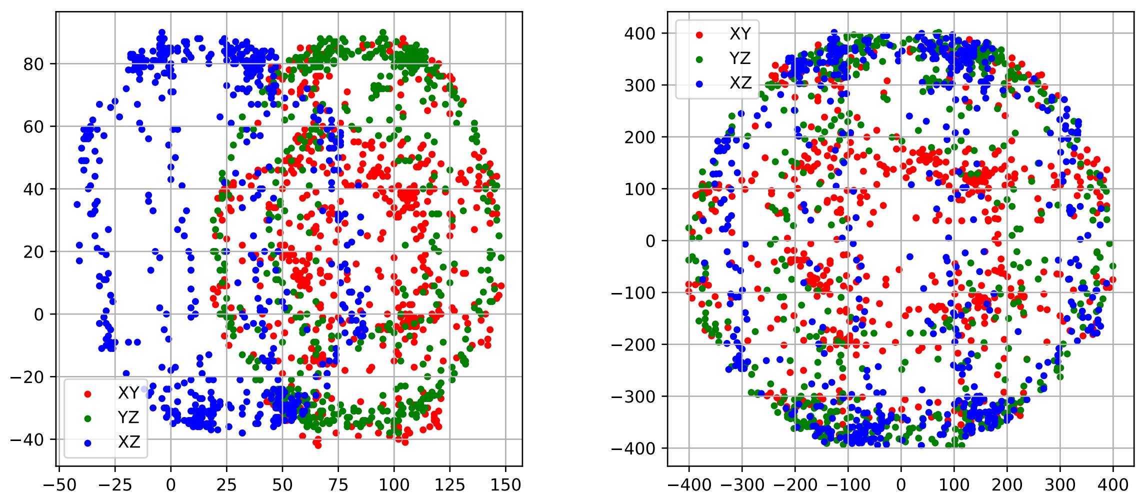


Figure 5.1: Comparison of raw (left) and calibrated (right) magnetometer measurements

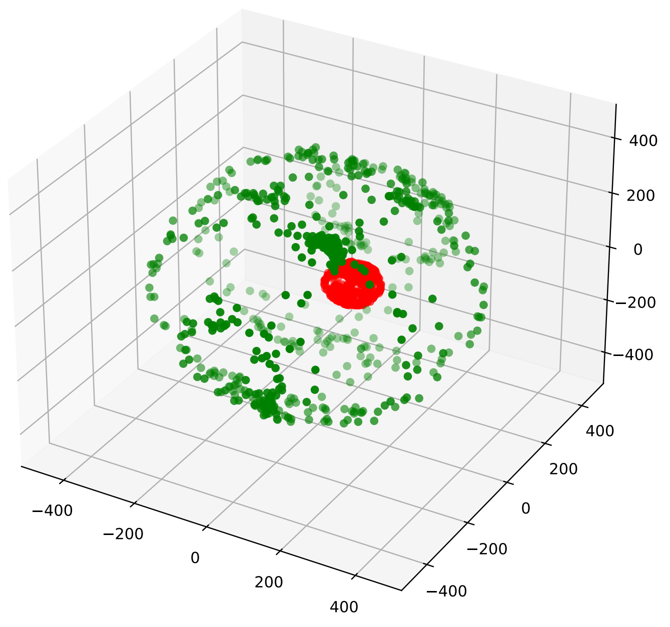


Figure 5.2: 3D visualization of plots from figure 5.1 for a better illustration. Red plot represents the uncalibrated data set and green plot is the calibrated one.

The other mechanism is a so called soft iron distortion and it is caused by materials that distort existing magnetic fields, such as nickel or iron. This kind of error is however much less significant compared to the hard iron distortion [32]. Along with these distortions, it is necessary to compensate for the imperfections of the sensors' sensitivity and alignment.

Even though the error sources are more complex, than in the case of accelerometers, the same basic correction method can be used even with magnetometers. Again, it is necessary to determine the the offset and the correction factor using the equations (5.1) and (5.2) and to apply this correction using equation (5.3).

In figures 5.1 and 5.2, it is evident, that the uncalibrated plot is not centered and it is not spherical. The calibrated plot is almost a perfect sphere with a predefined radius and its center in the origin.

5.2 More accurate calibration of accelerometer and magnetometer

To calibrate the set of accelerometers or magnetometers more accurately, it is important to notice, that there is a factory error even in the alignment of the individual sensors.

This means, that all three accelerometer will, to some extent, contribute to a measured acceleration in any given body axis. Similarly all the magnetometers will contribute to the measured value in any axis. Contribution of one sensor will be dominant for a given axis, the contribution of other two sensors will be very small.

To get the vector of calibrated values $\vec{u}$ from raw measurement vector $\vec{v}$, a calibration matrix M and a bias vector $\vec{b}$ must be used in the following form:

$$\vec{u} = (\vec{v} - \vec{b})M \tag{5.4}$$

The calibration values for the accelerometer can look like this:

$$M = \begin{bmatrix} 0.998694 & 0.005377 & -0.002012 \\ 0.005377 & 0.998725 & -0.000803 \\ -0.002012 & -0.000803 & 0.976216 \end{bmatrix}$$

$$\vec{b} = \begin{pmatrix} 21.22 & 55.93 & -278.88 \end{pmatrix}$$

Notice the significance of the values in the main diagonal of matrix M. If the simple calibration method from sections 5.1.2 and 5.1.3 was used, the values outside of the main diagonal would be 0.

To obtain the calibration matrix and the bias vector, a program called Magneto v1.2 was used (section 10.3.2).

To go through the calibration properly, I used the following guides and threads: [4, 5].

Another method for advanced calibration is to use AI [32].

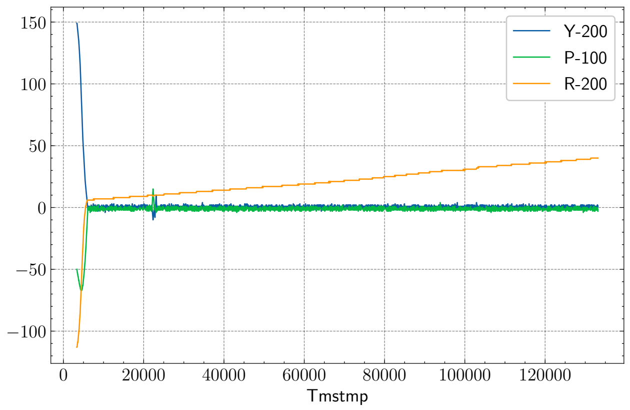
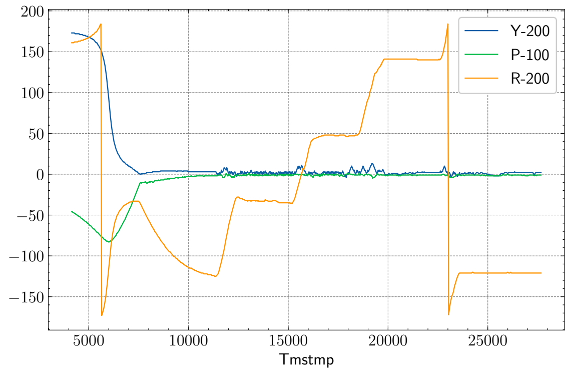
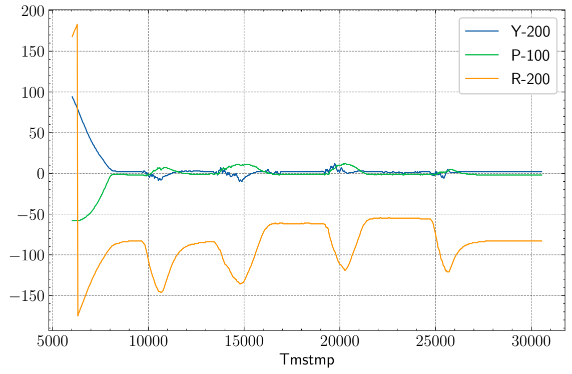


Figure 5.3: Roll drift when not using magnetometer nor any compensation



(a) Madgwick's sensor fusion algorithm with calibrated magnetometer



(b) Madgwick's sensor fusion algorithm with uncalibrated magnetometer

Figure 5.4: Comparison of using calibrated and uncalibrated magnetometer

5.3 Effects of magnetometer and poor calibration

The drift around the inertial frme x axis when using Madgwick's algorithm without magnetometer can be seen in figure 5.3. Madgwick's algorithm provides a compensation mechanism for the drift, by supplying the approximate drift rate. This method is still susceptible to noise and to inaccurate measurements. In my test the roll angle estimate (R) had drifted by about 45° in 2 min when not using any compensation. Had the magnetometer been used, the roll estimate would have stayed constant, just like the yaw (Y) and pitch (P) angles.

Figure 5.4 shows the difference between using calibrated and uncalibrated magnetometer. In both cases the IMU was rotated 4 times by 90° around the inertial frame x axis. In figure 5.4b, it is visible, that the sensor fusion algorithm does not converge to a correct orientation.

It is evident, that for a model rocket, which needs to know its orientation for a period of several seconds, avoiding the use of magnetometer is far better than using an uncalibrated or

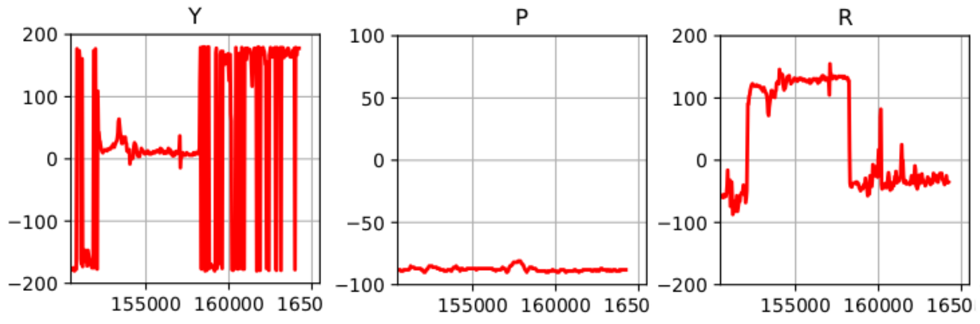


Figure 5.5: Gimbal lock illustration

poorly calibrated one. This can result in an option of buying a cheaper chip.

It is also clearly visible, that the algorithm initially needs about 3 s to converge. Figure 5.4 shows that the convergence time of roll angle estimate, when using magnetometer, can be as high as 7 s.

5.4 Gimbal lock

When the pitch angle approaches ±90°, the transformations achieved by yaw and roll become ambiguous and the result is an unreliable information about yaw and roll. This phenomena is called a gimbal lock. For applications, where the range of possible pitch angles is small, such as passenger airplanes, this does not pose a thread. Since model rockets ideally fly within a very narrow band of pitch angles, the gimbal lock should not be a problem for them either. It will be noticeable only when the rocket reaches apogee and the nose starts dropping below the horizon. In this phase of the flight, the parachute will hopefully be about to open, so there is no need for the control system to still be controlling the rocket at this point.

An example behavior of the Euler angles, while experiencing gimbal lock, is seen in figure 5.5. The pitch angle is about −90°. Very little movements of the sensor cause the yaw and roll estimate to jump around. These angles still represent the real orientation, but this orientation can now be achieved in two different ways with the Euler angles, instead of just one [24]. Euler angles near gimbal lock cannot be used as an input to a control system.

5.4.1 Avoiding the gimbal lock during ascend

In order to avoid gimbal lock, the pitch angle must never approach ±90°. But it is possible, that the IMU will be mounted at a 90° angle in the rocket from the very beginning and therefore the pitch angle would always be close to 90°. There are at least two ways to mitigate this problem.

First way is to never physically mount the IMU in this orientation. This is less ideal, because it constraints the physical layout of the control unit. And if the IMU itself is on the same PCB as the microcontroller, it will very likely need to be mounted at this problematic angle.

Second option is to swap axes of the IMU in code. This means that the sensor fusion algorithm will be given the measured values in a different order and/or with different signs and it will output a pitch angle of 0° when the chip is mounted vertically in a standing rocket. This trick is much more flexible and will be used throughout the rest of this work. It means that the IMU can be mounted in any orientation and still output the desired angles.

```
// The order of arguments passed to the sensor fusion algorithm for the MPU-9250.
    This produces 0 deg pitch when the chip is horizontal
MadgwickQuaternionUpdate(Axyz[0], Axyz[1], Axyz[2], Gxyz[0], Gxyz[1], Gxyz[2],
    Mxyz[1], Mxyz[0], -Mxyz[2], deltat);

// The order of arguments to produce 0 deg pitch when the MPU-9250 mounted
    vertically in a rocket
MadgwickQuaternionUpdate(-Axyz[2], Axyz[1], Axyz[0], -Gxyz[2], Gxyz[1], Gxyz[0], -
    Mxyz[2], -Mxyz[0], Mxyz[1], deltat);
```

Listing 5.1: Swapping axes in code to avoid gimbal lock in the expected orientations

Either of these methods does not eliminate the possibility of the gimbal lock, they just push this phenomena to an orientation where it is no longer problematic. Quaternions, as directly outputted from the Madgwick's sensor fusion algorithm, do not suffer from gimbal lock and these can be used for control as well. They would be the next logical step to improve the control system.

Chapter 6

Selection of microcontrollers

This chapter will document the requirements, comparison, testing and a final selection of a microcontroller, which will be used to build the flight control unit for an actively controlled rocket.

In the following sections, I will be comparing microcontroller development boards, instead of microcontrollers alone, since the control unit will be built with the MCU development board soldered to a prototype board. Ideally, the control unit would be assembled using a custom-made PCB with most of the components used in their SMD variants, but that is outside of the scope of this project. The development board will be connected to the selected sensors using wires and traces on the prototype board. For the sensors, available breakout boards will be used and those will be soldered to the same prototype board as the microcontroller development board. Hence the physical dimensions of all the boards will be really important.

Initially, I have developed and built a very simple data-logging unit based on Arduino Nano, which is small and cheap. The data-logging unit was used to test the selected sensors, write the initial code, pinpoint the shortcomings of Arduino for a better selection of the microcontroller for the flight control unit and to gather data from flights of the data-logging model rocket (appendix B.1). Moreover, I will be able to use the flight data to test algorithms for detecting different stages of the flight.

I will talk about the design and assembly of the data-logging unit in section 8.1 and about the flight control unit in section 8.2. The selection of sensors will be described in chapter 7.

6.1 Hardware requirements for the control units

From the development and usage of the data-logging control unit, it was evident, that a fast microcontroller with a rather large Flash memory and RAM will be needed to build the flight control unit. The theory listed in the previous chapters now becomes really important, because it defines a set of attributes and capabilities that the selected microcontroller must possess.

The selected development board must be rather small, as the inside diameter of the actively controlled model rocket is 5 cm (appendix B.2). The microcontroller must have at least one I^{2}C interface, an SPI interface, ideally an UART interface and it must have enough I/O pins for at least one LED, a button and five servomotors. The required connectivity is however fulfilled by pretty much any MCU. Additionally, the microcontroller must have at least 64 kB of Flash memory. The size of RAM wasn't an issue in the data-logging unit with the Atmega328 and its 2 kB, but I expect the RAM requirements to grow beyond just 2 kB. Ideally, the microcontroller should not be very expensive, because there is a rather high change of it being destroyed.

Debugging interface, wireless connectivity, rich documentation and good support are very beneficial and will be considered as well.

I will include the Arduino Nano in this comparison, to serve as a baseline for all the other microcontrollers.

6.2 Comparison of considered development boards

The development boards compared in the following sections are:

- Arduino Nano
- ESP-WROVER-KIT
- FRDM-K64F
- Teensy 4.0
- Nucleo-L432KC

The comparison of the features of all the MCUs is available in a table C.2 in the appendix.

6.2.1 Arduino Nano

Arduino Nano [33] is the smallest official Arduino development board, it is very common, cheap and easy to work with. I have build the data-logging unit around this board.

It features the Atmega328 microcontroller with an 8-bit AVR architecture [34]. It operates at 5 V. The clock speed is 16 MHz, it has 32 kB of Flash memory, out of which 2 kB is used for bootloader. The size of SRAM si 2 kB and it also has 1024 B of EEPROM. There is a single SPI, an I^{2}C and an UART interface.

The layout of the development board is open-source which allows for very cheap clones to be sold. Original Arduino Nano costs \$20, but it can be bought from other sources for as low as \$2.50.

The amount of peripheral connectivity with this board is more than sufficient to control model rocket. With the data-logging unit, I have used 8 digital pins to connect the sensors, button, LED and an SD card reader. This leaves 14 more pins to connect the servomotors and any additional chips, buzzers, LEDs or buttons.

During the programming of the data-logging unit, it became apparent, that 32 kB of Flash memory would not be enough for the flight control unit. I have had problems fitting just the data-logging program to the Arduino, let alone a program with some control algorithm and additional libraries for controlling the servomotors. I have also found it rather slow for a real-time control of a model rocket.

Despite being a very poor choice, the Arduino Nano in the data-logging unit gave me a great idea about what the final microcontroller will need to be like. Additionally, it has one great advantage, I am familiar with its programming model (section 9.1.1). Therefore, I want to aim for a board which is similarly easy to program and use.

6.2.2 ESP-WROVER-KIT

This rather large board is one of the development boards available for the ESP32 microcontroller. This board in particular features the ESP32-WROVER-B module, an ESP32 based MCU. There is also an SD-card reader, LCD display, RGB LED and an FTDI FT2232 chip directly on the board [35].

The ESP32 is a very popular system on a chip microcontroller developed by Espressif Systems. It has a powerful Tensilca Xtensa LX6 32-bit dual-core microprocessor and it operates at 240 MHz. The microcontroller has 4 MB of Flash memory and 320 kB of SRAM. The MCU operates at 3.3 V [36].

The FT2232 chip provides USB-to-serial and USB-to-JTAG interfaces which allow for communication and debugging of the software. However, I was not able to make the debugging work with this development board, although I suspect that it was a problem caused by my of OS.

ESP32 supports the Arduino programming model, along with Espressif's own ESP-IDF (IoT Development Framework, section 9.1.3). The ESP-IDF delivers a multitude of libraries for an IoT development [37] as well as tools necessary to compile code, upload it and debug. These tools are cross-platform and they are used through a CLI.

However, using the Arduino programming model in PlatformIO (section 9.2.1), I have had a hard time using the board. Especially when I wanted to use the same libraries which worked on the Arduino Nano. Since this board is very large and it would not fit into the rocket frame anyway, I didn't want to spend too much time with it. But there are other ESP32 boards available, which are way smaller. The average price of ESP-WROOM-32-based development board is about $10 in local e-shops and it can be bought for as low as $4 from international sources.

Overall, the connectivity and the performance is really good on paper. Had I figured out how to make the needed libraries work, a smaller ESP32 development board would have been be a really good choice for the flight control unit.

6.2.3 FRDM-K64F

FRDM-K64F is a development board for the K64F microcontroller [38]. K64F is an MCU based on the ARM Cortex-M4 32-bit core with an FPU. It operates at 3.3 V, has a clock speed of 120 MHz, 1 MB of Flash memory and 256 kB of SRAM [39].

The microcontroller supports many communication interfaces, including 10/100 Mbit/s Ethernet, 3x I^2C interface, 3x SPI interface and some more. The development board also features FXOS8700CQ – a chip with a set of accelerometers and magnetometers; a micro SD card socket and an RJ45 connector. The board has two rows of pins with the outer row being compatible with the Arduino R3 standard. There are 40 I/O pins available in total, out of which 24 can be used as analog input. To board supports CMSIS-DAP [40] or J-LINK debugging through a USB cable.

This board is very feature rich, however it is a little bit too heavy and large for the 5 cm diameter of the controlled rocket. I have not been able to find any other development board featuring this microcontroller and hence the only option to use this MCU in this project would be to build a custom board with it.

The cost of the development board is around $40, which is more on the expensive side.

As for the programming models, it supports ARM Mbed (section 9.1.2). There is no support for the Arduino programming model, which makes this board a little harder to use for me. But ironically, I have found it easier to use than the ESP-WROVER-KIT. However, must of the libraries that are needed for this project would need to be rewritten for Mbed.

6.2.4 Teensy 4.0

Teensy is a series of very popular, Arduino-compatible boards. Developed and sold by PJRC, Teensy 4.0 features an IMXRT1062 microcontroller based on a very powerful Cortex-M7 processor core [41]. The clock speed of this chip is 600 MHz, it contains a 32-bit and 64-bit FPU, has 1984 kB of Flash memory, 1024 kB of RAM and 1 kB of emulated EEPROM. Cortex-M7 is also the first ARM microcontroller core to feature branch prediction, which should further improve the performance of this board.

The board has 40 I/O pins, out of which 31 can be used as PWM and 14 as analog input. It also supports as much as 7 serial ports, 3 SPI and 3 I^2C ports, cryptographic acceleration and few other functions. Physically the board is very small, at around 35 mm$\times$18 mm.

For the programming models, Teensy supports the Arduino programming model and, to a limited extent, CircuitPython. Teensy code can also be compiled with a GNU Make a in terminal. It does not support ARM Mbed or any other development platform and it is also not equipped with on-board debugging, therefore external J-LINK probe would need to be used to allow for debugging the software [42].

The cost of this board is \$20 from the official PJRC shop, which is not that expensive, but together with the shipping cost, it is considerably more expensive than the Arduino Nano or the ESP32.

Except for the missing on-board debugging support, this board seems like a very good choice. It is very small and it is compatible with the Arduino. I have found that all the needed libraries for this project worked well practically without any modification. I have observed the MCU to be heating up significantly more than other microcontrollers, but this should not pose a risk.

6.2.5 Nucleo-L432KC

Nucleo-L432KC is a rather cheap alternative to Arduino Nano, with very low power consumption, designed and manufactured by STMicroelectronics. The microcontroller on this board is an STM32L432KCU6, based on the ARM Cortex-M4 core with FPU [43]. This 32-bit processor operates at 80 MHz, has 256 kB of Flash memory and 64 kB.

Great advantage of this board is the great compatibility with the Arduino Nano. The pin layout of this board is practically identical to the Arduino board, although the functions of some pins slightly differ. In total there are 14 digital I/O pins and 8 analog pins, same as with the Arduino Nano. In contrast to Atmega328, the STM32L432KCU6 features 2 I^2C, 2 SPI and 2 USART interfaces.

The microcontroller is ARM Mbed enabled, but it supports the Arduino programming model as well. The support for Arduino programming model and the same pin layout should theoretically mean very few differences between the data-logging unit and the flight control unit, if I were to choose this board.

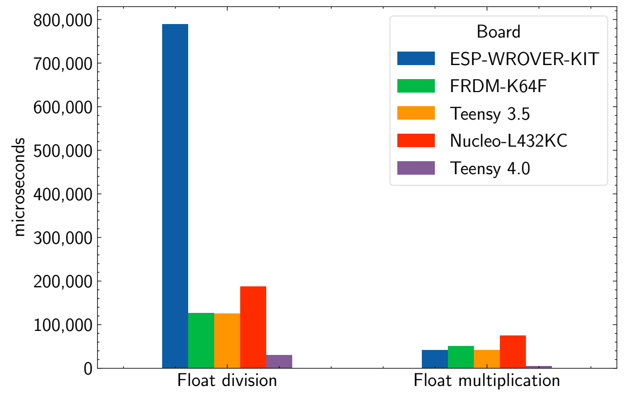
This board also supports debugging using ST-LINK/V2-1 debugger, which transfers data over USB cable, no additional debugging probe is needed. This has worked really well for me with the PlatformIO IDE.

The dimensions of this board are about the same as the Arduino Nano at around 50 mm×15 mm. It costs about \$10.

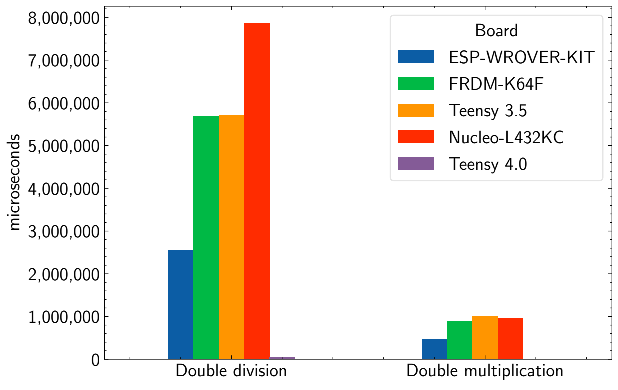
Overall, this board seems like a really good choice for the flight control unit. The price is reasonable and the supported models and the ease of use certainly help. The downsides of this board are the relatively small Flash memory and slower clock speed compared to the other boards.

6.3 Benchmarking of MCUs

Speeds of the considered MCUs will be shown here. I have focused my benchmarks on operations and functions which will be used heavily in the control unit software. The plots in figures 6.1, 6.2 and 6.3 show times per 1 000 000 iterations of critical operation on given development boards in microseconds.



(a) 32-bit floating point



(b) 64-bit floating point

Figure 6.1: Microseconds to 1 million floating point operations on selected microcontrollers

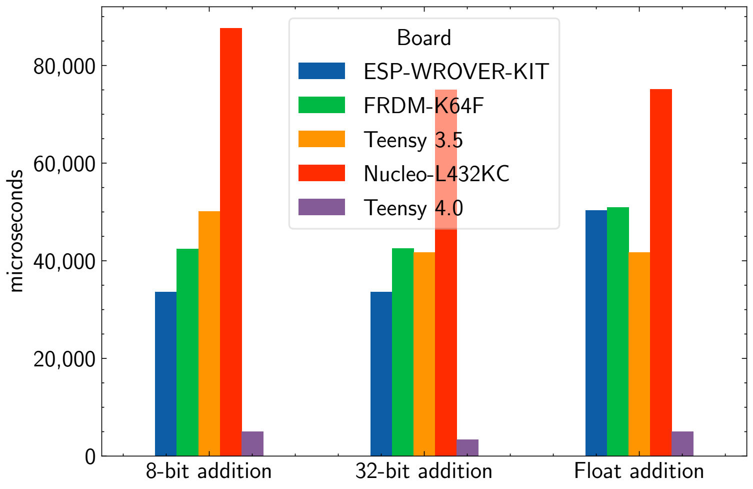


Figure 6.2: Microseconds to 1 million 8-bit integer, 32-bit integer and 32-bit float additions on selected microcontrollers

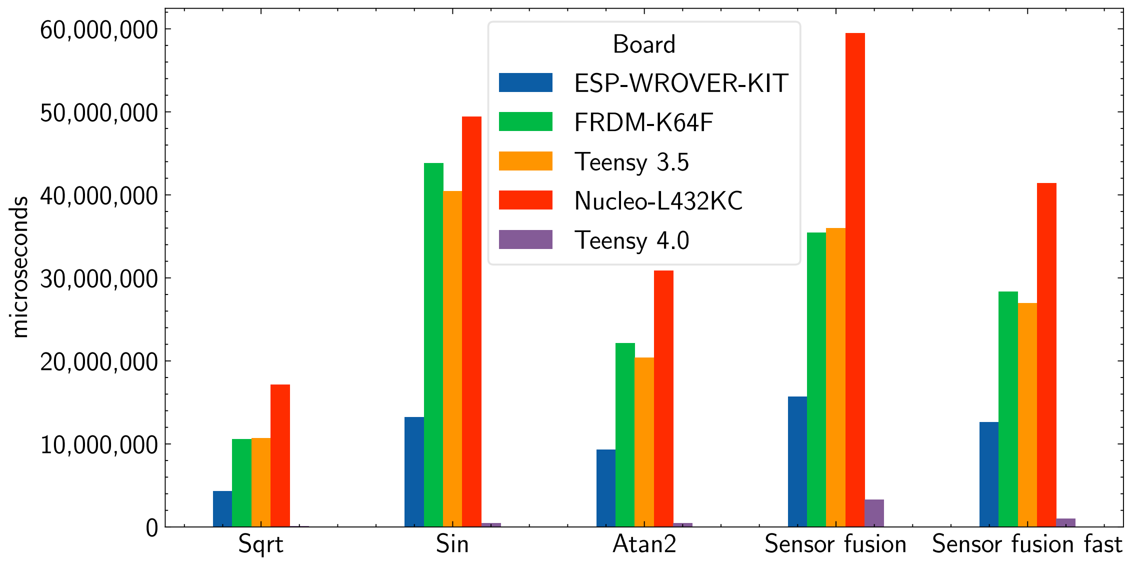


Figure 6.3: Microseconds to 1 million iterations of of important functions as well as Madgwick's sensor fusion algorithm

```
void floatMultiplication() {
  float number = 1;
  unsigned long endTime;
  unsigned long startTime = micros();

  while(1) {
    for(unsigned long counter = 0; counter < iterCount; counter++) {
      number *= 1.00001f;
    }
    endTime = micros();

    Serial.printf("%d -- %d\n", number, endTime - startTime);

    delay(1000);
    startTime = micros();
    number = 1;
  }
}
```

Listing 6.1: The general structure of the benchmark. This specific case is for float multiplication in the Arduino programming model.

I have omitted Arduino Nano from the plots, since it was on average 80× worse than all the other boards and it would make it much harder to see the differences between those other boards. Square root and trigonometric functions were very decent on the Atmega328, only about 4× slower than the rest of the MCUs on average. However, floating point operations were more than 100× slower than the rest. This is partly due to the fact, that Atmega328 does not feature FPU.

I have also included Teensy 3.5 in the plots, because I have had one available. It is evident, that the performance of this board is pretty much the same as the performance of FRDM-K64F. They both feature very similar MCU, however the Teensy board is much smaller and it supports the Arduino programming model.

The times of all the tests, including the Arduino Nano, are shown in a table C.1.

Table 6.1: Sizes of basic data types of the selected MCUs in bits

Board	Arduino Nano	ESP-WROVER-KIT	FRDM-K64F	Teensy 3.5	Nucleo-L432KC	Teensy 4.0
Integer	16	32	32	32	32	32
Long	32	32	32	32	32	32
Float	32	32	32	32	32	32
Double	32	64	64	64	64	64

6.3.1 Teensy 4.0 speed

What immediately becomes evident is the shear speed of the Teensy 4.0 with its Cortex-M7-based MCU. Even tough the purple bar is sometimes so small, that it is barely visible, it is always there.

A great advantage of Teensy 4.0 is the fact, that the clock speed is about 5× faster than the rest of the boards. The other factor that helps Teensy 4.0 a lot is the fact, that Cortex-M7 core has branch prediction. My benchmark was a simple loop with a million iterations of a single operation. The branch prediction should help a lot in such case. The real-world performance of this MCU should therefore be slower.

That being said, figure 6.3 shows a speed of the Madgwick's sensor fusion algorithm plotted in the two rightmost groups. This is much more complicated than a single instruction and there are no conditions in the algorithm itself. Still the Teensy dominates over the rest of the boards. The `Sensor fusion fast` benchmark uses an implementation of a fast inverse square root [44].

6.4 MCU selection

Taking all the information from this chapter into account, I have decided to use the Teensy 4.0 board as the base for the flight control unit.

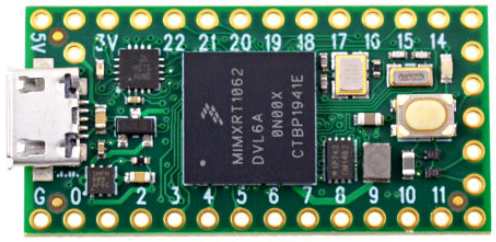


Figure 6.4: Teensy 4.0 development board

The main decision factors were the speed, the ease of use, large Flash size, enough RAM and a great compatibility with the data-logging unit, which had already been developed during making this selection.

I have taken into account the lack of support for the on-board debugging. I have also considered the increased heating and larger power consumption, neither of which will be an issue, since the control unit will need to operate only for several minutes at a time and the largest compute load will fit into just few seconds.

Chapter 7

Peripherals for the control units

7.1 MPU-9250

As the IMU for both of the control units, I have chosen the MPU-9250 [45]. It features 3-axis accelerometer, 3-axis gyroscope and 3-axis magnetometer.

It uses either I^2C or SPI interface for communication. The I^2C address of this sensor is `b110100X`, where X is determined by the logic level on an AD0 pin. If the voltage is high, X is 1 and if it's low, X is 0. This allows for 2 different MPU-9250s to be connected to the same bus when using I^2C. The I^2C communication is done at 400 kHz.

The MPU-9250 is a single package which consists of two chips. First chip is an MPU-6050 – an IMU without magnetometers; and the second is an AK8963 – a chip with a set of magnetometers. These chips are connected via an internal I^2C line and the MPU-9250 package contains a register, which, after setting the right value, allows for direct communication to the AK8963 over the external I^2C. This is used to configure the AK8963 and to read data from the magnetometers.

The accelerometers support ranges of ±2 g's, ±4 g's, ±8 g's and ±16 g's[1], the gyroscopes support rotation rates of $\pm250^\circ\,\mathrm{s}^{-1}$, $\pm500^\circ\,\mathrm{s}^{-1}$, $\pm1000^\circ\,\mathrm{s}^{-1}$ and $\pm2000^\circ\,\mathrm{s}^{-1}$ and the magnetometers have a range of $\pm4800\,\mu\mathrm{T}$. All of these values are well within what is needed for the model rocket control unit.

Apart from normal measurements, the MPU-9250 can provide an external interrupt via its `INT` pin. This interrupt can be configured to trigger during different events. Possibly the most useful one being the "wake up on motion" function, that would signalize the lift-off of the model rocket. However, I have decided not to use this functionality.

The alignment of axes between accelerometers/gyroscopes and magnetometers is not consistent. Accelerometers and gyroscopes use an NED convention (North, east, down), whereas magnetometers uses ENU (East, north, up). This must be taken into account when passing the measured values

[1] 1 g is an acceleration equivalent of $9.8\,\mathrm{m{\cdot}s^{-2}}$

into the sensor fusion algorithm. See section 5.4.1 for the correct order of arguments passed into the sensor fusion algorithm.

The reason for selecting this board is very simple. It is relatively cheap, easy to use and very easy to obtain. It might currently be the most common IMU among hobbyists, which results in plenty of online resources which help greatly with using this board. It also operates fine both on 3.3 V and on 5 V.

7.1.1 MPU-9250 alternatives

7.1.1.1 BNO055

BNO055 is an IMU designed by Bosch [46]. It is more precise than MPU-9250, can communicate over I^2C and, perhaps most notably, it can perform sensor fusion on an integrated MCU and output the orientation quaternion directly, if desired. Therefore, this chip should be called AHRS, rather than IMU.

The ability to perform sensor fusion can offload a lot of work from the main microcontroller. The individual sensors also feature configurable low-pass filters and they can operate in several different modes. The maximum ranges of measurements are comparable to the MPU-9250 [47].

However, this device is about twice as expensive than the MPU-9250, and I was not able to find a breakout board for this sensor in local online stores.

7.1.1.2 ICM-20948

This device is very low power and, functionally, it is comparable to the MPU-9250. It does not support any of the on-board logic, like the BNO055 does. It communicates over I^2C and SPI and has equal measurement ranges like the previous sensors [48]. However, it is primarily designed for smartphones and wearables and I would probably not trust this sensor on a model rocket. Due to its low power consumption, this can be used as a backup IMU though. I was not able to find a breakout board for this device.

7.2 BMP280

BMP280 is the sensor of choice to measure pressure and hence to determine the barometric altitude of the rocket. It is a tiny sensor which can measure a pressure range of 1100–300 hPa, which is an equivalent of -500 to 9000 m below/above sea level. The relative accuracy of this sensor is ± 0.12 hPa, which, at sea level, is equivalent to a resolution of about ± 1 m of altitude [49].

This sensor communicates both over I^2C and SPI. The I^2C slave address in defined by connecting pin SD0 either to GND (address 0x76) or to V_{DDIO} (address 0x77). The sensor operates at 3.3 V.

The reason for choosing this device is the same as with the MPU-9250. It is cheap and easily available in a form of a breakout board. It is very common, so there is a lot of resources online.

It works well both with the Arduino Nano and with the 3.3 V MCUs. The pressure range and the precision is well within of what is required for low-power model rockets.

7.2.1 BMP280 alternatives

7.2.1.1 MS5607-02BA03

This sensor is far more accurate than the BMP280 with the resolution as high as 20 cm. The minimum pressure is also significantly lower, at 10 hPa. Neither of these benefits would probably be enough to justify the use of this sensor over BMP280 in my intended use case, especially since I was not able to find a breakout board for it. The chip can use either an I^2C or an SPI interface for communication and it operates at 3.3 V [50].

7.2.2 MicroSD slot

There are a lot of available breakout boards with a MicroSD slot to log the flight data to an external storage. Pretty much any of those boards can be used.

My board of choice operates at 3.3 V and it uses SPI for communication with the SD card [51]. However, it is not ideal to rely on a physical connection to the SD card during high vibrations in flight, as the card could momentarily disconnect. Ideally, one should store the flight data temporarily on an external Flash memory and then, after a safe landing, transfer the data from the Flash onto the SD card.

This was not done in this project though, and I was writing data directly to the SD card during flight, risking the potential data loss. However, I have not encountered a problem in any of my flights.

7.3 Not used peripherals

7.3.1 GPS

The GPS module was not used as a part of this project, but it was selected. I wanted to use the NEO-6M GPS. It sends data at 5 Hz with a claimed resolution of 2.5 m [52]. In my tests, I saw the module to be able to distinguish distances of about 7 m. After considering the benefits, the added complexity, the module size and weight and its precision, I have decided against the use of this module as a part of the flight control unit.

7.3.2 Radio antenna, telemetry

To receive a real-time information about the current state of the rocket, especially during flight, some model rocket makers include chips and antennas for wireless communication as a part of the

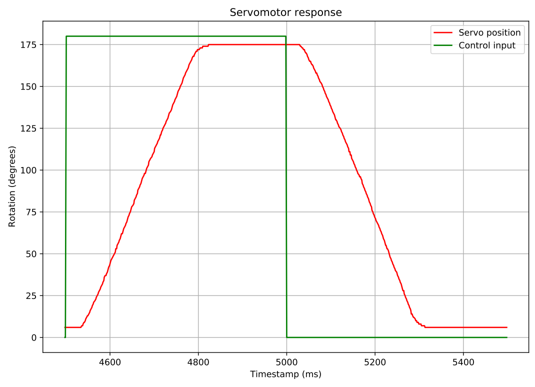


Figure 7.1: Response to the control input of the SG-90 servomotor

control unit. I have decided not to include telemetry in this project, since the added complexity and weight far exceed the advantages of such system when flying in low altitudes.

7.3.3 External Flash memory

This probably should have been included, but I was not able to find a breakout board for the Flash memory module in time and therefore this peripheral was not used in neither the data-logging unit or the flight control unit. As stated in section 7.2.2, writing data directly onto an SD card during flight is a bad idea, since the card can get disconnected due to vibrations and acceleration. Safer approach is to log data to a Flash memory and transfer this data to the SD card after landing.

7.4 Servomotor

For the actively controlled model rocket, having reliable, fast and precise servomotors is crucial. However, I decided to start with cheap servos in the beginning, since there is a large chance of crash. The servomotor that I have selected is the TOWERPRO SG-90 Micro Servo [53].

It weighs 9 g, has a torque of 1.8 kg·cm, speed of 0.1 s/60°, operates at a voltage of 5 V and it costs about $3.

None of the pins on any MCU development board can deliver sufficient current to supply power to even one servo, experiencing the expected loads, and therefore the servomotors must be powered separately – in parallel to the other boards. The LiPo battery that I have decided to use has a nominal voltage of 7.4 V, which is just at the edge of tolerable voltages for the servo. I don't want to risk long-term damage to the servo due to high voltage, so I decided to step the battery voltage down using several diodes, that can deliver high enough current, in series. A 5 V DC-DC converter would be a much better choice, but due to size constraints it wasn't really possible in my case.

7.4.1 Servomotor behavior and data - speed, precision, delay

I wanted to see how the servo reacts to a control input, so I have built a small rig to connect the servo to a potentiometer and I have logged the response using an Arduino.

In figure 7.1, you can see the green plot of the control input and the red plot of the response. The servomotor has about 32 ms delay before it starts moving. This delay can be caused by margins in the servomotor and in the measuring device. The maximum speed at 6 V is about 0.1 s/67°, which is consistent with the documentation. The angles range from 6° to 175°, which is 11° less than the servomotor should have been able to achieve. This was also confirmed by visually observing these tests, but it is not an issue.

Chapter 8

Design and assembly of the control units

8.1 Design and assembly of the data-logging unit

The data-logging unit is based around Arduino Nano. The IMU is an MPU-9250 (section 7.1), there is a BMP280 as a pressure sensor (section 7.2) and a slot for a MicroSD card.

Both the MPU-9250 and the BMP280 are connected to the same I^2C bus and data is written to the SD card over SPI. The logic level of the I^2C bus is held at 3.3 V by 10 kΩ pull-up resistors in the breakout boards of both the MPU-9250 and the BMP280. The schematic design and pictures of this unit are shown in appendix D.

The Arduino Nano is connected directly to a 7.4 V LiPo battery (pins VIN and GND). The 7.4 V is well within the range of tolerable supply voltages for the Arduino. All the mentioned peripherals are powered from the Arduino 3.3 V output, which can deliver a maximum of 50 mA. According to the datasheets, the BMP280 and MPU-9250 together should draw a maximum of around 4 mA, which is fine. The Micro SD card should draw around 30 mA while writing and reading. However, from my measurements, the total current consumption through the 3V3 pin on the Arduino Nano peaked at 25 mA during initialization and at around 5 mA during continuous operation. The average current consumption through the 3V3 pin was 1.5 mA.

Measured current draw from the battery was a constant 40 mA during initialization and 34 mA during continuous operation. Single 450 mAh 7.4 V LiPo battery should last for more than 12 h when fully charged.

8.1.1 Basic peripherals

Basic peripherals of this unit consist of one LED and a button.

The button is connected to pin A0, which is held high by Arduino's internal pull-up resistor. The other side of the button is connected to GND. This button is only used to start the data-logging process once the rocket is in the vertical position and ready to fly. After the button is pressed, the data-logging loop starts and the button no longer has any purpose.

The LED signalizes the readiness of the control unit, different error states and finally the data-logging procedure. When the unit is ready, the LED shines continuously. Upon error, the LED blinks certain amounts of time, which encodes the error message, and then stays off for 1 s before repeating the cycle. When the data-logging process is happening, the LED changes state every iteration, resulting in rapid blinking.

8.2 Design and assembly of the flight control unit

The schematic design of this unit is show in appendix E. The selected MCU board for this control unit is the Teensy 4.0. The sensors have remained the same as with the data-logging unit – MPU-9250 as the IMU and BMP280 as the pressure sensor. There is the same MicroSD slot and the same basic peripherals – a button and an LED.

On top of that, the circuitry contains an L7805 – a 5 V voltage regulator; and a connector for five servomotors. The regulator must be there, because the battery voltage is outside of the tolerable range for the VIN pin of the Teensy.

Initially, there were two L7805s. One was powering the Teensy and two servomotors and the other was powering the remaining three servomotors. However, the regulators could not handle the current needed by the servomotors and once a servo started moving, the Teensy would restart. Hence, I have decided to keep a single L7805 just for the Teensy, since it needs more steady voltage, and step down the voltage for the servomotors using diodes (D2 – D7 in the schematic). The output voltage to the servos is now at around 6.4 V with no load, when the battery is fully charged. The servomotors should be able to handle these levels without long-term damage. An input capacitor for the servomotors would probably protect them from voltage fluctuations, but I have ran out of space on the board.

The entire control unit has been measured to draw around 200 mA without the servomotors. Each servo can draw 100–350 mA. Single 450 mAh 7.4 V LiPo battery should last for more than 30 min with this unit. This is still more than enough, but it is significantly less than the data-logging unit.

A PCB version of this control unit would use a 5 V DC-DC converters for the MCU and for the servos, as these are much more efficient and suitable for this application.

Chapter 9

Programming models and tools

In this chapter, I will first list the most relevant programming models for the control units. I will then compare available IDEs and finally, I will mention command line tools, which can be used with the mentioned programming models.

9.1 Programming models

9.1.1 Arduino

The Arduino programming model is a set of open-source libraries and functions for C++. Initially intended for the AVR architecture, those libraries have been ported to support a variety of boards and MCUs [54].

In my case, this programming model is supported by the Nucleo-L432KC, Teensy 4.0 and ESP-WROVER-KIT and, of course, Arduino Nano. The only board which does not support this model is the FRDM-K64F board.

This model, is very popular, it is very simple to use and there are lots of resources available online. There is a lot of libraries for many different peripherals, including libraries for all the peripherals which I have been using during the development of the control units.

Since I was using the Arduino Nano and Teensy 4.0 boards for the control units, I was practically forced to use this programming model. However, I have found the development to be easy and I didn't really miss the features of more advanced programming models, e.g. the Mbed OS. That being said, the program running on the control units is currently rather simple and the Arduino programming model might eventually start showing its limitations.

9.1.2 Mbed OS

Mbed OS is intended as a programming platform for IoT devices based on the Cortex-M boards. It provides an abstraction layer for the microcontroller, which allows the same code to run on any

Mbed-enabled board [55].

It provides two profiles. First profile is called full profile, which contains RTOS functionality and a full set of supported features. This profile allows to run fully deterministic, multithreaded, real-time application on the Mbed-enabled boards. The other profile is called bare metal profile. This is designed for applications without the need for complex thread management. Compared to the full profile, the bare metal profile contains only the minimum subset of the available APIs and it does not support any of the RTOS features.

From my selection of boards, the Mbed OS is supported on the Nucleo-K432KC board and on the FRMD-K64F. It is not supported on the Arduino Nano and Teensy boards or on the ESP-WROVER-KIT.

Had the Teensy 4.0 in the flight control unit supported this programming model I would have seriously considered it. For a larger project I would tend to prioritize this Mbed OS over the Arduino programming model.

9.1.3 ESP-IDF

Espressif IoT Development framework is the official development framework for the ESP32-based MCUs [56]. It supports only boards by Espressif, so it is not really interesting to this project.

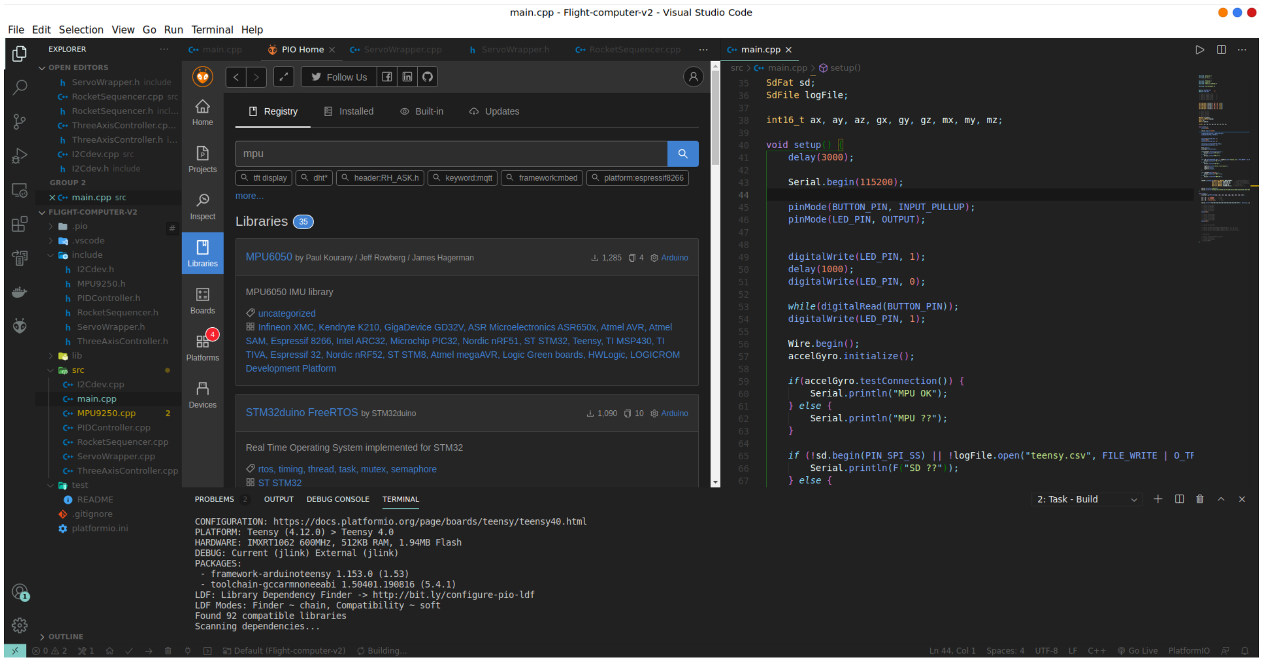
9.2 Development environments

9.2.1 PlatformIO

PlatformIO is a great open-source tool, which I have been using to program all the MCU code for this project. It consists of a cross-platform IDE and a set of CLI tools, so called PlatformIO Core. The IDE is delivered as a plugin to VSCode [57].

It supports all the common programming models, such as Arduino programming model, Mbed OS or ESP-IDF and it has native support for over 1000 boards. It hosts a repository with almost 11 000 libraries, most of which have been submitted by the community. This repository includes all of the popular Arduino libraries, libraries for the ESP32 boards, etc. A library can be added to the project either manually or via a graphical tool available in VSCode.

When creating new project, one selects the target board and programming model and a pre-configured project is created. Connecting a board and uploading a program to the board by clicking an upload button added into VSCode by PlatformIO is usually all that is needed, regardless of the programming model used. If the board comes with on-board debugging, there is usually a default configuration in PlatformIO that works out of the box. I have been able to debug both the Nucleo-K432KC and FRDM-K64F boards without any additional configuration. However, I have not been able to debug the ESP-WROVER-KIT at all.


main.cpp - Flight-computer-v2 - Visual Studio Code
File Edit Selection View Go Run Terminal Help
Registry
Installed
Built-in
Updates
mpu
Libraries 35

MPU6050 by Paul Kourany / Jeff Rowberg / James Hagerman — 1,285 — 4 — Arduino

MPU6050 IMU library

uncategorized

Infineon XMC, Kendryte K210, GigaDevice GD32V, ASR Microelectronics ASR650x, Atmel AVR, Atmel SAM, Espressif 8266, Intel ARC32, Microchip PIC32, Nordic nRF51, ST STM32, Teensy, TI MSP430, TI TIVA, Espressif 32, Nordic nRF52, ST STM8, Atmel megaAVR, Logic Green boards, HWLogic, LOGICROM Development Platform

STM32duino FreeRTOS by STM32duino — 1,090 — 10 — Arduino

Real Time Operating System implemented for STM32

rtos, timing, thread, task, mutex, semaphore

ST STM32

```cpp
SdFat sd;
SdFile logFile;

int16_t ax, ay, az, gx, gy, gz, mx, my, mz;

void setup() {
    delay(3000);

    Serial.begin(115200);

    pinMode(BUTTON_PIN, INPUT_PULLUP);
    pinMode(LED_PIN, OUTPUT);


    digitalWrite(LED_PIN, 1);
    delay(1000);
    digitalWrite(LED_PIN, 0);

    while(digitalRead(BUTTON_PIN));
    digitalWrite(LED_PIN, 1);

    Wire.begin();
    accelGyro.initialize();

    if(accelGyro.testConnection()) {
        Serial.println("MPU OK");
    } else {
        Serial.println("MPU ??");
    }

    if (!sd.begin(PIN_SPI_SS) || !logFile.open("teensy.csv", FILE_WRITE | O_TF
        Serial.println(F("SD ??"));
    } else {
```

```
CONFIGURATION: https://docs.platformio.org/page/boards/teensy/teensy40.html
PLATFORM: Teensy (4.12.0) > Teensy 4.0
HARDWARE: IMXRT1062 600MHz, 512KB RAM, 1.94MB Flash
DEBUG: Current (jlink) External (jlink)
PACKAGES:
 - framework-arduinoteensy 1.153.0 (1.53)
 - toolchain-gccarmnoneeabi 1.50401.190816 (5.4.1)
LDF: Library Dependency Finder -> http://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 92 compatible libraries
Scanning dependencies...
```

Figure 9.1: VSCode with PlatformIO. The screenshot shows PlatformIO's library manger.

9.2.2 Arduino IDE

Arduino IDE is the official development environment for the Arduino boards. The list of supported boards can be expanded to Teensy or STM32 boards, but the feature set is inferior compared to PlatformIO and any non-Arduino programming model cannot be used. Navigation in projects is much harder and there is no support for on-board debugging [58].

9.2.3 MCUXpresso

MCUXpresso is an IDE developed by NXP. It is intended for Cortex-M-based microcontrollers by NXP. Based on Eclipse IDE, it provides features such as code compilation, advanced debugging, profiling and more [59].

I have not found it very useful in my case, since I needed to work also with boards, which are not based on a Cortex-M core, during this project and having one tool (PlatformIO) for any platform was much more convenient for me.

9.3 Software tools

9.3.1 Mbed CLI

To compile code written using the Mbed OS, the Mbed website provides an online Mbed compiler [60]. However, there is another tool, which I would personally prefer much more – the Mbed

CLI [61]. It is a Python-based package that allows compilation, uploading, debugging, etc. using a command line interface.

9.3.2 Arduino CLI

Official Arduino website provides a download for its own CLI tool for compilation and uploading the programs onto the Arduino MCUs. It is called Arduino CLI [62] and I have found it very convenient. It is simple to use and it is independent of the text editor, which is what I always prefer.

Chapter 10

Implementation of the MCU software

All the code for the MCUs was written in VSCode, using the PlatformIO extension. All the source codes are listed in the appendix A.

10.1 Software for the data-logging unit

The software for the data-logging unit is dependent on libraries for the MPU-9250, BMP280 and the SD card reader. The MPU-9250 library has been provided in a thread at the Arduino forum [5]. The library for BMP280 has been developed by Adafruit and it is available through the PlatformIO's library manager.

The SD card interface is provided by a library called SdFat, which is also available through the library manager in PlatformIO. I have used the SdFat library, rather than the default Arduino SD library, because of its reduced Flash memory usage. This library also contains a class called `MinimumSerial`, which is a minimal implementation of the serial communication protocol used to talk to a computer over a USB cable. This has further reduced the program's memory usage.

The main loop of the program runs at around 50 Hz with the data being logged only onto the SD card – this is reasonable configuration for flight. With both the serial output and the SD card output enabled, the loop runs at around 32 Hz.

The program logs raw data from all the sensors and it runs the Madgwick's sensor fusion algorithm during each iteration. I have tried to remove the sensor fusion algorithm, since it does not provide any additional value, but that only increased the output speed by about 10 Hz. Thus, I have decided to keep the sensor fusion as a part of the code.

10.2 Software for the flight control unit

The software running inside of the flight control unit is based on the same libraries as the software for the data-logging unit. Additionally, it uses an Arduino library for controlling the servomotors.

The `MinimumSerial` class is not used, because it does not work with the Teensy 4.0 board and the lack of Flash memory is no longer an issue on this board.

From an architectural point of view, the most dominant feature is the use of the State pattern [7] to easily toggle between different states of the control unit. The `AbstractSequencer` interface defines Init and Update methods. The software currently has three classes which implement this interface – `MenuSequencer`, `CalibrationSequencer` and `RocketSequencer`. The use of this pattern was guided by the convenience of having the calibration routine as a part of the main software. This means that there is no need to separately upload the calibration routine as a standalone program.

In every iteration of the main loop, the Update method of the current state is called. The Update method returns the name of the state which should be used in the next iteration of the main loop. This determines, whether the state will get changed or not. When the state is changed, the Init method of the newly selected state is called.

The entire rocket's state, including raw sensor data, sensor fusion output, references to sensor objects, etc. are located in the `RocketState` object. This class could have probably been implemented as a Singleton [7], but since the instance of this class is created statically in the beginning of the program, there should not be a risk of having multiple `RocketState` objects.

This software also uses an implementation of a PID controller from section 4.4. The logic is encapsulated in a `PIDController` class with the gains and limits of the controller passed to the constructor of this class. There are three instances of the `PIDController` in the code, one for each axis and with potentially different gains. An instance of the `RocketSequencer` class handles the output to the servomotors. As stated before, this output is the sum of the outputs from the relevant PID controllers.

10.3 Additional software

During the course of this project, it was needed to use and/or create some extra software. This included a data collection routine to collect raw data from the IMU, a calibration software, real-time plotter, which was of great help, and scripts to visualize calibration and flight data.

10.3.1 Raw IMU data collection routine

Both control units occasionally need to run a routine to collect the raw measurements from the IMU. This is used to determine the maximum and minimum measured values of magnetometers and accelerometers in all axes. These ranges are used with the Magneto v1.2 calibration software (section 10.3.2) to obtain the bias vectors and calibration matrices.

In the case of the data-logging unit, this routine always has to be uploaded to the MCU, instead of the data-logging program, because both of these programs would not fit into the Flash memory at once. This is very inconvenient, especially before a rocket launch, since uploading these programs back and forth takes a lot of time. But it has to be done, because the magnetometer must always be calibrated.

This routine depends on the same MPU-9250 library as the primary software for control units.

When this routine is started, it first collects an average bias of the gyroscope. During this phase, the control unit must be at rest. After that, the routine collects specified number of samples from accelerometers and magnetometers. While collecting, the unit must be rotated slowly, to correctly capture the maximum range of values for all the sensors. The collected accelerometer and magnetometer samples are both saved into a single csv file, where each line has six elements, three for accelerometers and three for magnetometers. The routine also writes all the data to a serial port.

10.3.2 Magneto v1.2

Magneto v1.2 is used to obtain the calibration matrices and a bias vectors, which are used to transform the raw measurements into usable values for a sensor fusion algorithm.

It was originally written as a GUI application for Windows, but its source code is now available[1]. In order to compile, it must be provided with several algebraic functions[2]. In addition to the original implementation, I have added an option for a command line argument to pass the name of a file with the raw data. I was then able to run this program from an Ubuntu 20.04 terminal.

Furthermore, I have created a Python wrapper script on top of Magneto v1.2. This script first separates the input file, obtained by the aforementioned IMU raw data collection routine, into two separate files, one for magnetometer measurements and the other for accelerometer data. The scripts then calls Magneto v1.2 for both of these files individually.

Listing 10.1 shows an example call of the Magneto v1.2 wrapper script. The script reads the `MAGACCEL.CSV` file directly from an SD card. The path to this file is given as the first argument. The second argument to the wrapper is the norm of the gravity vector, third is the norm of magnetic field vector and the last argument is a path to a file with the average offsets of the gyroscopes.

The wrapper first outputs the data in a format which is convenient for use with a Jupyter notebook. This helps with visualizing the calibration, if desired. Below the `"=== CONTROL UNIT CALIBRATION ==="` line is the output with the correct calibration values in C syntax and the text can be directly copied into the source code of the control unit program. Notice the accelerometer bias vector in array `A` at indexes $0-2$ and the accelerometer calibration matrix in array `A` at indexes $3-11$. These are values are similar to what has been seen in section 5.2.

The output has been trimmed, because the accelerometer and magnetometer calibration is done in the same way.

[1] 11, Available from: `https://sites.google.com/site/sailboatinstruments1/c-language-implementation`.

[2] 63, Available from: `http://www.mymathlib.com/matrices/`.

```
jakub@kuba-linux:~/School/Bakalarka/Magneto-C$ ./magneto-wrapper.py /media/jakub/
    E887-A82A/MAGACCEL.CSV 2048 400 /media/jakub/E887-A82A/GYRO.CSV

=== JUPYTER literals ===
ACCEL OFFSET
[0.734263, 13.610052, -153.658549]
ACCEL MATRIX
[0.994287, -0.014832, -0.011972, -0.014832, 0.994467, 0.009378, -0.011972,
    0.009378, 0.978073]

MAG OFFSET
...
=== CONTROL UNIT CALIBRATION ===
ax = ax - 0.734263;   // A[0]
ay = ay - 13.610052;   // A[1]
az = az - -153.658549;   // A[2]

Axyz[0] = (ax * 0.994287 + ay * -0.014832 + az * -0.011972) * ascale; // A[3] A[6]
     A[9]
Axyz[1] = (ax * -0.014832 + ay * 0.994467 + az * 0.009378) * ascale; // A[4] A[7]
    A[10]
Axyz[2] = (ax * -0.011972 + ay * 0.009378 + az * 0.978073) * ascale; // A[5] A[8]
    A[11]

Gxyz[0] = ((float)gx - 4.6) * gscale;
Gxyz[1] = ((float)gy - -26.4) * gscale;
Gxyz[2] = ((float)gz - -0.7) * gscale;

mx = mx - 83.135251; // M[0]
...
```

Listing 10.1: Calling the Magneto v1.2 wrapper with a real input

10.3.3 Real-time plotter

I wanted to see the data from the control units in real-time. For this purpose, I have created a simple Python script, which reads data from a serial port the control unit and it plots this data.

This gave me a great insight into the performance of the control unit and especially the into the sensor performance and the sensor fusion algorithm estimation correctness.

This script is based on Python libraries `serial` and `matplotlib`, also available through `pip`. It expects certain format of the data from the control unit. The last message before the data starts flowing must be "`Logging`". This string is detected by the real-time plotter. Immediately after this line, names of the columns must be printed and these must be separated by `\t`, just like a `csv` file would have. The column names must also contain the expected range of values – this is required to be an integer after a hyphen. This information is used to construct the charts. Finally, the data is read from the serial port line-by-line and is used to update the plots.

```
Logging
Tmstmp Y-200 P-100 R-200 AccX-4 AccY-4 AccZ-4 GrX-6 GrY-6 GrZ-6 MgX-500 MgY-500
    MgZ-500 hPa-1500 Btn-2 Frq-100
3133 136 54 172 0.061662 -0.122814 0.983335 0.005273 0.031825 -0.019548 0.723665
    -0.047183 0.818920 989.34 1 0.32
...
```

Listing 10.2: Required output of a control unit which is used to plot data in real-time

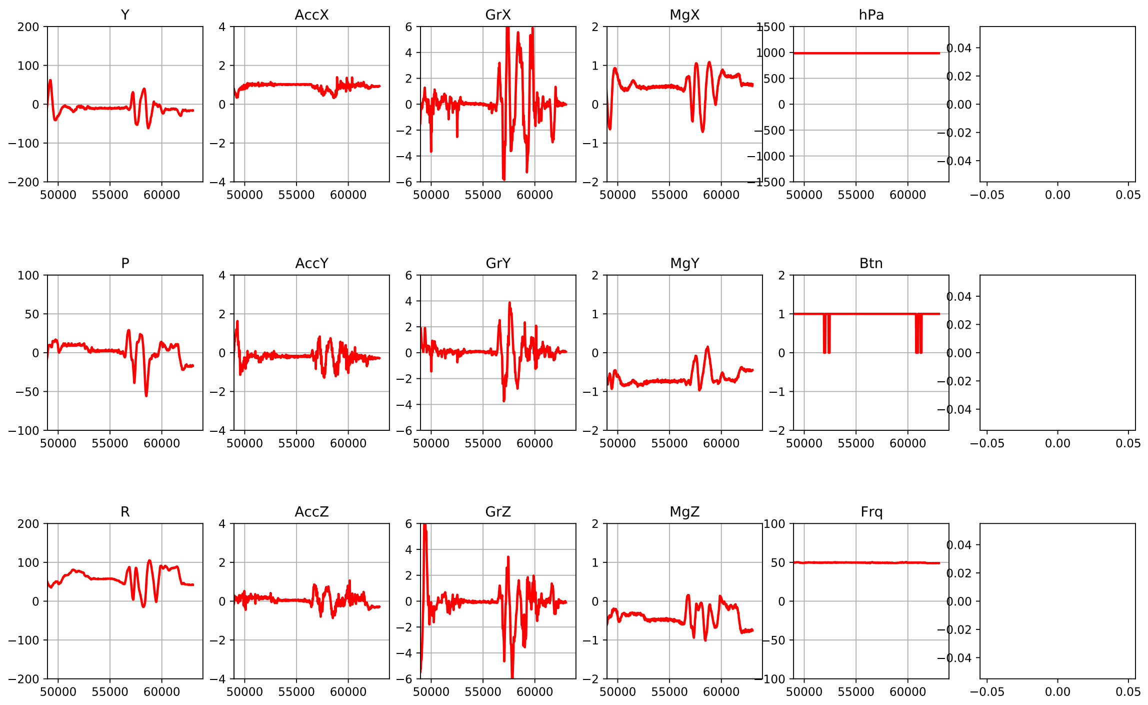


Figure 10.1: Real-time plotter window. Y, P, R means Yaw, Pitch, Roll, Btn is a button state.

Chapter 11

Testing and visualized data

In this chapter, I will present the results of several tests of the developed control units. Shortcuts or abbreviations are often used in the plots. This was needed to save Flash memory in the data-logging unit. Hopefully the meaning of the labels will be clear.

In the plots, `Tmstmp` means timestamp and its value is in ms. The time starts from 0 ms when the unit is powered up. Similarly, `Y` means yaw angle, `P` means pitch angle and `R` means roll angle.

All of the plots have been trimmed to show only the significant events, e.g. the entire flight.

The visualization has been done in `jupyter notebook` [64] using `pandas` [65] and `matplotlib` [66] libraries. The visual style of the plots is modified by `SciencePlots` library [67].

11.1 Data-logging rocket flights

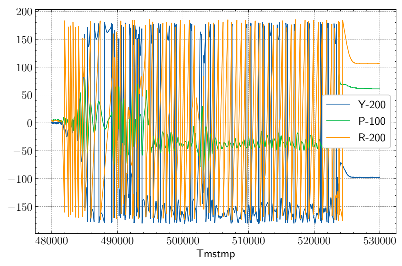
This section shows the best data that I have obtain from 3 flights with the data-logging rocket and with the data-logging unit onboard. The plots in figure 11.1 are taken from two different flights. This is the reason why the timestamps don't match.

Figure 11.1a depicts the estimated orientation of the rocket, as represented by Euler angles.

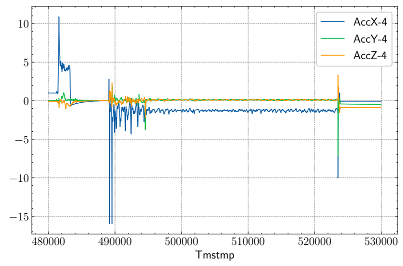
The roll estimate seems reasonable, as it shows the rocket spinning along the vertical axis with a period of about 0.75 s – this is normal and an actively controlled rocket with similar control unit onboard should therefore have correct information to control its roll.

However, the pitch, and especially the yaw angle estimates are both terrible. The plots in figure 11.1a are unfortunately not very clear, but it should hopefully be visible. These plots show that the sensor fusion algorithm started estimating an upside-down orientation very shortly after burnout. This is, of course, incorrect. Further more, the pitch angle estimates are oscillating heavily at a frequency of about 1 Hz long before the parachute has opened.

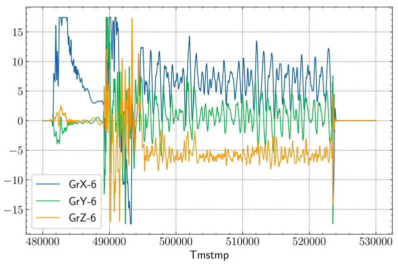
This is probably caused by a combination of factors. A minor factor is the fact, that the gyroscope measurements, visible in figure 11.1c, are saturated and are clipping. This is a configuration issue and the gyroscopes can be set to a higher range. More significant factor is the negative accel-



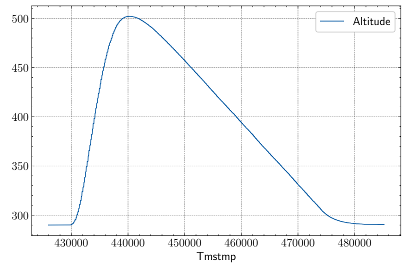
(a) Yaw, pitch, roll



(b) Acceleration profile



(c) Rotation speed



(d) Altitude

Figure 11.1: Yaw, pitch and roll, acceleration, rotation speed and altitude plots of data-logging rocket flights

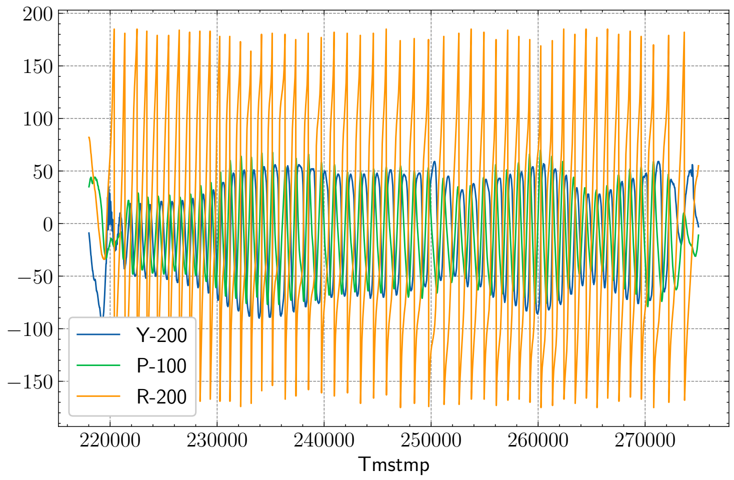
eration, caused by atmospheric drag, after the motor burnout, visible in figure 11.1b. This probably confuses the sensor fusion algorithm into thinking that the rocket is upside down. This issue has to be solved before any attempt of actively controlling the yaw and pitch of the rocket.

I am very happy with the figure 11.1b, which shows the measured acceleration. The plot looks exactly as expected. There is a period of no activity before launch. Then, high acceleration begins, which peaks at about 11 g's, followed by sustained acceleration of about 4.5 g's, which lasts for approximately 2 s. This exactly matches the general thrust profile of model rocket motor visible in figure 3.2. After burnout, there is a very clear deceleration, which exponentially dies away. The deceleration is caused by atmospheric drag acting on the rocket. The drag is a function of *velocity squared*, so as the rocket slows down, this negative acceleration decreases. After about 5 s after burnout, parachute deployment is visible as the large spikes. From this point on, the nosecone with the data-logging unit are tumbling upside down under a parachute, before the rocket touches down, again visible as the large spikes.

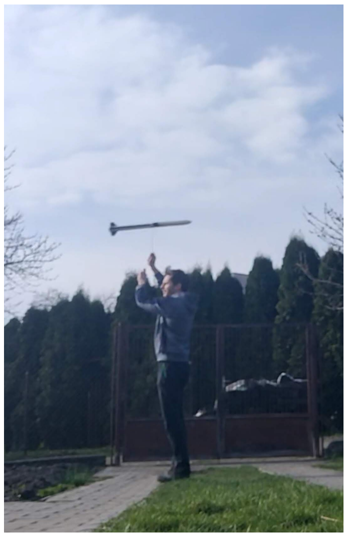
The altitude data is not directly measured by the rocket, but it is derived from the measured pressure and known elevation of the launch site. The rocket has reached maximum altitude of 212 m above ground.

Both, the maximum altitude and the acceleration, very closely match a simulation done in OpenRocket.

The data from these flight can be used to modify the software and these changes could then be



(a) Estimated Euler angles from the tangled test



(b) Flying with the rocket on a string around head

Figure 11.2: Estimated orientation vs. reality during the tangled test

verified by future flights. These flight have greatly helped me to understand the behavior of the sensors and the sensor fusion algorithm and it allowed me to find what changes need to be made. I have also verified that the connection to the SD card is secure and that it did not disconnect during flight. I have been monitoring state of the button, and it too was not being accidentally pressed.

11.2 Tangled tests of the data-logging rocket

To test the control system of the actively controlled rocket, I wanted to use a technique, where the model rocket connected to a 2 to 3 m long string in its CG. The rocket is then manually span above head in a circle while holding the string in hands.

I have tested this technique with the passively stabilized rocket and figure 11.2a shows the estimated orientation. Note that in this figure, yaw and roll axes are swapped, relative to the rocket's body frame, therefore roll still shows magnetic heading.

Although not as bad is the estimate of powered flights, the yaw and pitch angles are still quite different from reality. The actual pitch and yaw angles were oscillating between $\pm 10°$ at most, which is clearly visible in figure 11.2b. The string is connected to the rocket in a specific point, so the IMU should be almost perfectly horizontal and therefore, the estimate of the yaw angle should be very close to $0°$, regardless of the rocket's pitch angle.

The discrepancy between the estimate and reality is probably due to the centrifugal force caused by the rotation. This result is too bad for an attempt of active control during such test and hence I cannot use this technique for the actively controlled rocket.

11.3 Actively controlled rocket tests

Although I have been able to verify, that the flight control unit works and that it can reliably control the servomotors and thus the canards, I couldn't find a way to test the control system without risking damage to the rocket. My hopes were high for the tangled test described in section 11.2. However, the estimated angles from the Madgwick's sensor fusion algorithm are not good enough in such a test. This has stopped me from developing the control unit software, until I can find more suitable and reliable method for estimating the rocket's orientation.

Chapter 12

Conclusion

This project's primary goal was to design, build and test a control unit for a model rocket. The text thoroughly describes the design process and comments on the required software solutions, which include the software for the control units, data visualization scripts and some auxiliary programs. The work goes into greater detail of the control unit design from a hardware perspective.

This project fulfills all the specified tasks and personally, I rate this project as very successful, even though the flight control unit was never tested in an actively stabilized flight. To accomplish such thing, a much deeper analysis of the entire control system, precise modeling and tuning of the control loop would be needed. However, this would be a subject of a completely different scientific field.

When doing research for this thesis, I have found several publications that go into a great detail on modeling the rocket's aerodynamics and dynamic behavior of the control mechanism, but I have not been able to find a work which would be mainly concerned with building and programming the control unit. From this point of view, I see the text as beneficial and it can serve as a reference for further research and, hopefully, even for hobbyists when trying to build a controlled model rocket. Additionally, I provide the source codes for the control units in the appendix of this work, which can also be of a great help to anyone.

The assembled flight control unit worked exactly as it was designed to. The voltage levels for the servomotors, for the MCU and for the sensors were in the expected and tolerable ranges, the power usage was reasonable and the performance of the MCU and sensors themselves was excellent.

To build and fly the data-logging unit before designing the flight control unit has proven to be an incredibly helpful decision. This allowed me to deeply analyze the flight data and see the shortcoming of this initial solution.

The one result, which is really concerning is the performance of the Madgwick's sensor fusion algorithm on a model rocket. Although it works fine on the ground and in powered portion of the flight, once a negative acceleration occurs, the algorithm quickly starts outputting unusable values.

12.1 Further work

12.1.1 More suitable sensor fusion

Perhaps the quickest and easiest improvement of the current results would be to find a more appropriate sensor fusion algorithm or to change the method of estimating orientation altogether. The desired outcome is to have the correct estimate of the rocket's orientation during both powered and unpowered parts of the ascend, all the way to the maximum altitude.

This improvement can be achieved by using the already collected raw data, since this data is not dependent on the used sensor fusion algorithm. This way, the change can be implemented without a need for another flight of the model rocket. An optional flight might however be performed to verify the correctness of this change.

12.1.2 Controlled powered flight

Once the attitude estimation algorithm outputs correct values and once the control system has been correctly tuned, the flight control unit can be flown in the actively stabilized rocket which has already been built for this purpose.

12.1.3 Single-board control unit

Great addition to this work would be to take the selected sensors, peripherals and microcontroller, or any of the alternatives, and build a board on a custom-made PCB with those components' SMD variants. This would allow to shrink the board greatly, to add a lot of useful components, and it would greatly simplify the wiring. On the other hand, the unit would be more expensive and soldering the SMD components is harder and it might require additional equipment.

12.1.4 Simulation

Another possible extension of this work would be to write a software to simulate the flight and the control unit. OpenRocket already provides a Java interface to manipulate the simulation parameters, such as the rocket's fin angle [68]. However, I have found that it is only possible to affect the roll angle of the rocket using this method. Controlling pitch and yaw seems not to be supported by OpenRocket right now. Therefore an improvement of this simulator or creating a brand new software would greatly help with building actively controlled model rockets.

Bibliography

1. GUERRERO, Valeria Avila; BARRANCO, Angel; CONDE, Daniel. Active Control Stabilization of High Power Rocket. 2018-06, pp. 87. Available also from: `https://scholarcommons.scu.edu/cgi/viewcontent.cgi?article=1080&context=mech_senior`.
2. MADGWICK, S. O. H.; HARRISON, A. J. L.; VAIDYANATHAN, R. Estimation of IMU and MARG Orientation Using a Gradient Descent Algorithm. In: *2011 IEEE International Conference on Rehabilitation Robotics* [online]. Zurich: IEEE, 2011-06, pp. 1–7 [visited on 2021-03-08]. Available from DOI: `10.1109/ICORR.2011.5975346`.
3. MAHONY, R.; HAMEL, Tarek; PFLIMLIN, Jean-Michel. Nonlinear Complementary Filters on the Special Orthogonal Group. *IEEE Transactions on Automatic Control* [online]. 2008-06, vol. 53, no. 5, pp. 1203–1217 [visited on 2021-01-28]. Available from DOI: `10.1109/TAC.2008.923738`.
4. MALLON, Edward. *Tutorial: How to Calibrate a Compass (and Accelerometer) with Arduino* [online]. 2015-05-23 [visited on 2021-03-09]. Available from: `https://thecavepearlproject.org/2015/05/22/calibrating-any-compass-or-accelerometer-for-arduino/`.
5. REMINGTON, J. *Fatal Bug in Sparkfun and Kris Winer's MPU9250 Code* [online] [visited on 2021-03-09]. Available from: `https://forum.arduino.cc/index.php?topic=645527.0`.
6. JEŘÁBĚK, Karel et al. *Raketové Modely.* 1st ed. Praha: Modela, 1985. Available also from: `http://www.mo-na-ko.net/download/rakety01.zip`.
7. GAMMA, Erich (ed.). *Design Patterns: Elements of Reusable Object-Oriented Software.* 39. printing. Boston: Addison-Wesley, 2011. Addison-Wesley Professional Computing Series. ISBN 978-0-201-63361-0.
8. *Sensor Fusion on Android Devices: A Revolution in Motion Processing* [online]. In collab. with SACHS, David. 2010-08-02 [visited on 2021-03-08]. Available from: `https://www.youtube.com/watch?v=C7JQ7Rpwn2k`.

9. LUDWIG, Simone; BURNHAM, Kaleb; JIMENEZ, Antonio; TOUMA, Pierre. Comparison of Attitude and Heading Reference Systems Using Foot Mounted MIMU Sensor Data: Basic, Madgwick, and Mahony. In: SOHN, Hoon (ed.). *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2018* [online]. Denver, United States: SPIE, 2018-03-27, p. 96 [visited on 2021-02-15]. ISBN 978-1-5106-1692-9 978-1-5106-1693-6. Available from DOI: `10.1117/12.2296568`.

10. *How To Build a Thrust Vectored Model Rocket - National Rocketry Conference 2020* [online]. In collab. with BARNARD, Joe. 2020 [visited on 2021-04-20]. Available from: `https://www.youtube.com/watch?v=4cw9K9yuIyU`.

11. *Magneto v1.2 - C-Language Implementation - Sailboatinstruments1* [online] [visited on 2021-03-09]. Available from: `https://sites.google.com/site/sailboatinstruments1/c-language-implementation`.

12. *Model Rocket Safety Code | National Association of Rocketry* [online] [visited on 2021-04-21]. Available from: `https://www.nar.org/safety-information/model-rocket-safety-code/`.

13. *Provoz v rámci „Otevřené“ (Open) kategorie* [online] [visited on 2021-04-21]. Available from: `https://www.caa.cz/provoz/bezpilotni-letadla/otevrena-kategorie-open/provoz-v-ramci-otevrene-open-kategorie/`.

14. *eRules pro Bezpilotní Systémy (UAS) (Nařízení (EU) 2019/947 a (EU) 2019/945)* [online] [visited on 2021-04-21]. Available from: `https://www.caa.cz/wp-content/uploads/2021/04/eRules_UAS_CS_08-04-2021_v2-1.pdf`.

15. *BEZPILOTNÍ SYSTÉMY Školicí Materiál ÚCL ve Formě FAQ - Nejčastěji Kladených Dotazů k Problematice* [online] [visited on 2021-04-21]. Available from: `https://www.caa.cz/wp-content/uploads/2020/11/FAQ-DRONES_CS.pdf`.

16. *CZECH SPACE OFFICE* [online] [visited on 2021-04-21]. Available from: `https://www.czechspace.cz/en`.

17. *KRaM – Raketoví modeláři* [online] [visited on 2021-04-21]. Available from: `http://svazmodelaru.cz/kram/`.

18. *VFR Příručka - Česká Republika* [online] [visited on 2021-04-21]. Available from: `https://aim.rlp.cz/vfrmanual/actual/enr_2_cz.html`.

19. *Předpis L2 - Doplněk O* [online] [visited on 2021-04-21]. Available from: `https://aim.rlp.cz/predpisy/predpisy/dokumenty/L/L-2/data/effective/doplO.pdf`.

20. TOM BENSON. *Rocket Stability* [online]. 2014-06-12 [visited on 2021-03-05]. Available from: `https://www.grc.nasa.gov/www/k-12/rocket/rktstab.html`.

21. *OpenRocket Simulator* [online] [visited on 2021-03-04]. Available from: `http://openrocket.info/`.

22. *RockSim V10 - Single User* [online] [visited on 2021-03-04]. Available from: https://www.apogeerockets.com/index.php?main_page=product_software_info&cPath=13_206&products_id=2636.

23. *Bezpecnost* [online] [visited on 2021-03-05]. Available from: http://raketove.modely.sweb.cz/bezpecnost.htm.

24. *Understanding Euler Angles | CH Robotics* [online] [visited on 2021-03-10]. Available from: http://www.chrobotics.com/library/understanding-euler-angles.

25. *TVC - BPS.Space* [online] [visited on 2021-04-18]. Available from: https://bps.space/tvc.

26. *Lumineer Design - 10km/Mach 1.7 L3 Rocket* [online]. In collab. with BARNARD, Joe. 2021-03-03 [visited on 2021-03-08]. Available from: https://www.youtube.com/watch?v=PuK4khlWKqg&t=388s.

27. *ASTRo Fin Actuation Test* [online]. In collab. with BOSTON UNIVERSITY ROCKET PROPULSION GROUP. 2014 [visited on 2021-04-18]. Available from: https://www.youtube.com/watch?v=lb3gqGsuEHI.

28. *PID Controller Implementation in Software* [online]. In collab. with SALMONY, Philip. 2020 [visited on 2021-04-20]. Available from: https://www.youtube.com/watch?v=zOByx3Izf5U.

29. *Extended Kalman Filter - AHRS 0.3.0-Rc1 Documentation* [online] [visited on 2021-03-08]. Available from: https://ahrs.readthedocs.io/en/latest/filters/ekf.html.

30. WANG, Li; ZHANG, Zheng; SUN, Ping. Quaternion-Based Kalman Filter for AHRS Using an Adaptive-Step Gradient Descent Algorithm. *International Journal of Advanced Robotic Systems*. 2015-09-24, vol. 12, pp. 1. Available from DOI: 10.5772/61313.

31. *NCEI Geomagnetic Calculators* [online]. U.S. Department of Commerce [visited on 2021-03-08]. Available from: https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml?#declination.

32. *Tutorial - Web Configure : MPU9250 - Calibrate Magnetometer on ESPrtk Using Magneto 1.2 - ESPrtk -ESP32 RTK* [online] [visited on 2021-01-28]. Available from: http://esprtk.wap.sh/tt/t3/tt_w_mpu9250_calibrate_magnetometer_esprtk.html.

33. *Arduino Nano | Arduino Official Store* [online] [visited on 2021-03-12]. Available from: https://store.arduino.cc/arduino-nano.

34. *Atmega328 Datasheet* [online] [visited on 2021-03-12]. Available from: https://atmega32-avr.com/Download/atmega328_datasheet.pdf.

35. *ESP-WROVER-KIT V4.1 Getting Started Guide - ESP32 - ESP-IDF Programming Guide Latest Documentation* [online] [visited on 2021-03-13]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-wrover-kit.html#.

36. *ESP32-WROVER-B Datasheet* [online] [visited on 2021-03-13]. Available from: `https://www.espressif.com/sites/default/files/documentation/esp32-wrover-b_datasheet_en.pdf`.

37. *API Reference - ESP32 - ESP-IDF Programming Guide Latest Documentation* [online] [visited on 2021-03-13]. Available from: `https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html`.

38. *Kinetis K64F Sub-Family Datasheet* [online] [visited on 2021-03-13]. Available from: `https://www.nxp.com/docs/en/data-sheet/K64P144M120SF5.pdf`.

39. *FRDM-K64F | Mbed* [online] [visited on 2021-03-13]. Available from: `https://os.mbed.com/platforms/frdm-k64f/`.

40. *CMSIS DAP - Handbook | Mbed* [online] [visited on 2021-04-21]. Available from: `https://os.mbed.com/handbook/CMSIS-DAP?utm_source=platformio&utm_medium=docs`.

41. *Cortex-M7* [online] [visited on 2021-03-15]. Available from: `https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7`.

42. *Teensy 4.0 PlatformIO 5.1.1b1 Documentation* [online] [visited on 2021-03-15]. Available from: `https://docs.platformio.org/en/latest/boards/teensy/teensy40.html`.

43. *NUCLEO-L432KC - STM32 Nucleo-32 Development Board with STM32L432KC MCU, Supports Arduino Nano Connectivity - STMicroelectronics* [online] [visited on 2021-03-18]. Available from: `https://www.st.com/en/evaluation-tools/nucleo-l432kc.html`.

44. *Understanding Quake's Fast Inverse Square Root – BetterExplained* [online] [visited on 2021-04-27]. Available from: `https://betterexplained.com/articles/understanding-quakes-fast-inverse-square-root/`.

45. *MPU-9250 Product Specification Revision 1.1* [online] [visited on 2021-03-10]. Available from: `https://invensense.tdk.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf`.

46. *BNO055* [online] [visited on 2021-03-21]. Available from: `https://www.bosch-sensortec.com/products/smart-sensors/bno055/`.

47. *BNO0055 - Datasheet* [online]. 2014-11 [visited on 2021-03-21]. Available from: `https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf`.

48. *ICM-20948 Datasheet* [online] [visited on 2021-03-21]. Available from: `https://invensense.tdk.com/wp-content/uploads/2016/06/DS-000189-ICM-20948-v1.3.pdf`.

49. *BMP280 Digital Pressure Sensor Datasheet* [online] [visited on 2021-03-21]. Available from: `https://cdn-shop.adafruit.com/datasheets/BST-BMP280-DS001-11.pdf`.

50. *MS5607-02BA03 Datasheet* [online] [visited on 2021-03-21]. Available from: https://www.te.com/commerce/DocumentDelivery/DDEController?Action=showdoc&DocId=Data+Sheet%7FMS5607-02BA03%7FB2%7Fpdf%7FEnglish%7FENG_DS_MS5607-02BA03_B2.pdf%7FCAT-BLPS0035.
51. *microSD Card modul SPI 3.3V - laskarduino.cz* [online] [visited on 2021-03-21]. Available from: https://www.laskarduino.cz/microsd-card-modul-spi-3-3v/.
52. *NEO-6 u-Blox 6 GPS Datasheet* [online] [visited on 2021-03-21]. Available from: https://drive.google.com/file/d/0B4B30jzMyzG8S0VtcWM5dFBvM1U/view?usp=drive_open&usp=embed_facebook.
53. *SG90 Servo Datasheet* [online] [visited on 2021-03-21]. Available from: https://datasheetspdf.com/pdf/791970/TowerPro/SG90/1.
54. *Arduino - PlatformIO 5.2.0a3 Documentation* [online] [visited on 2021-03-23]. Available from: https://docs.platformio.org/en/latest/frameworks/arduino.html.
55. *Introduction - Introduction to Mbed OS 6 | Mbed OS 6 Documentation* [online] [visited on 2021-03-23]. Available from: https://os.mbed.com/docs/mbed-os/v6.9/introduction/index.html.
56. *ESP-IDF Programming Guide - ESP32 - ESP-IDF Programming Guide Latest Documentation* [online] [visited on 2021-03-23]. Available from: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/.
57. *PlatformIO* [online] [visited on 2021-03-23]. Available from: https://platformio.org.
58. *Arduino - Software* [online] [visited on 2021-03-23]. Available from: https://www.arduino.cc/en/software.
59. *MCUXpresso IDE* [online] [visited on 2021-03-26]. Available from: https://www.nxp.com/design/software/development-software/mcuxpresso-software-and-tools-/mcuxpresso-integrated-development-environment-ide:MCUXpresso-IDE.
60. *Getting Started - Build Tools | Mbed OS 6 Documentation* [online] [visited on 2021-03-26]. Available from: https://os.mbed.com/docs/mbed-os/v6.9/build-tools/mbed-online-compiler.html.
61. *Mbed CLI 2 - Build Tools | Mbed OS 6 Documentation* [online] [visited on 2021-03-26]. Available from: https://os.mbed.com/docs/mbed-os/v6.9/build-tools/mbed-cli-2.html.
62. *Arduino Pro* [online] [visited on 2021-03-26]. Available from: https://www.arduino.cc/pro/cli.
63. *Matrices and Linear Algebra* [online] [visited on 2021-03-09]. Available from: http://www.mymathlib.com/matrices/.

64. *Project Jupyter* [online] [visited on 2021-04-21]. Available from: `https://www.jupyter.org`.

65. *Pandas - Python Data Analysis Library* [online] [visited on 2021-04-21]. Available from: `https://pandas.pydata.org/`.

66. *Matplotlib: Python Plotting - Matplotlib 3.4.1 Documentation* [online] [visited on 2021-04-21]. Available from: `https://matplotlib.org/`.

67. GARRETT, John. *Garrettj403/SciencePlots* [online]. 2021-03-26 [visited on 2021-03-26]. Available from: `https://github.com/garrettj403/SciencePlots`.

68. *Simulation Listeners - OpenRocket Wiki* [online] [visited on 2021-04-21]. Available from: `http://wiki.openrocket.info/Simulation_Listeners`.

Appendix A

Digital appendix

All the digital appendices are available in the IS Edison. What follows is an overview of the directory names and a short description.

A.1 Datasheets

The most important datasheets are available in the `Datasheets` directory.

A.2 Websites

Several websites, from which critical information was taken, are located in the `Websites` directory.

A.3 E-mail correspondence

An e-mail correspondence with a member of the Czech Model Rocketry Club is in the `E-mails` directory as a text file. Any personal information has been removed from the file and only the allowed sections are shared. The permission was given to me via an a-mail.

A.4 Software

A.4.1 Arduino benchmark

The code of the benchmark, which was used with MCUs which support the Arduino programming mode, is located in the `Software/Arduino-benchmark` directory.

A.4.2 Mbed benchmark

The source code of the benchmark for the FRDM-K64F board is available in the `Software/Mbed-benchmark` directory.

A.4.3 Magneto

`Software/Magneto_and_MagnetoWrapper` contains the Magneto v1.2 and my custom wrapper, described in section 10.3.2.

The Magneto source files are empty, due to potential license issues, but references to the websites with the source code are listed in this text. It should be possible to recreate the program.

A.4.4 IMU calibration

Calibration routine for the data-logging control unit is available in `Software/MCU_calibration_program`.

A.4.5 Data-logging software

The main program for the data-logging control unit is located in `Software/MCU_data_logging_program`.

A.4.6 Flight control software

The program to run on the flight-control unit is located inside of the `Software/MCU_flight_control_program` directory.

A.4.7 Visualization scripts

`Software/Visualizations` directory groups the Jupyter notebooks for visualization of calibration and flights, and a Python script used for the real-time serial port visualization.

These programs require `pandas`, `numpy`, `matplotlib` and `SciencePlots` libraries to be available in the environment. All of those libraries can be installed via `pip`.

A.5 Miscellaneous

In the `MISC` directory, there are two downloaded documents, which deal with the regulations of UAVs.

Appendix B

Model rocket designs

B.1 Data-logging rocket design

The data-logging rocket was designed as a pathfinder for the construction and to eventually carry a small data-logging unit, to test the sensors and the rest of the electronics. As seen in figure B.1 this rocket is designed to initially fly with B6-2 motor to an altitude of about 40 meter above ground. It has rather long payload section at the front, which is depicted as the blue tube between the nosecone and the purple cylinder. This section will house the data-logging unit and battery. The rocket will need to switch to stronger D9-5 motor for flights with the data-logging unit onboard, since the added hardware is quite heavy. The configuration with the D9-5 motor and the data-logging unit is not shown in the figure B.1, because OpenRocket does not allow to mix different motor configurations with different components in the rocket.

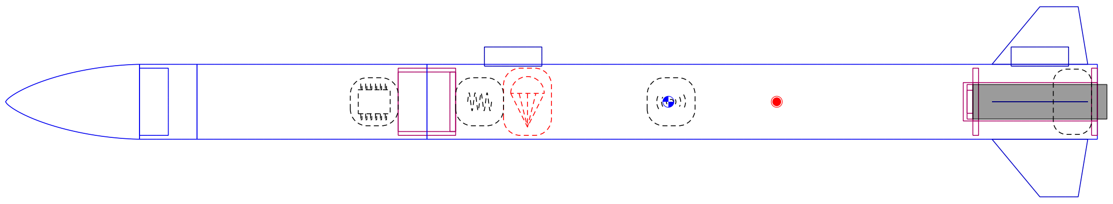
The body diameter is 36 mm and stability in the depicted configuration is around 1.5 calibers.

B.2 Actively controlled rocket design

For the actively controlled rocket, I needed to use a 5 cm body diameter, since it will need to house four servomotors and a parachute deployment mechanism with additional fifth servomotor. For actively controlled flights, motors with no ejection charge will be used and the parachute will be deployed mechanically by a servo.

The rocket weights about 420 g and is 95 cm tall. The static margin of this rocket is close to 3.

The control mechanism selected for this rocket are the actuated canards. Those can clearly be seen in figure B.3, along with the mechanical connections of the canards to the servomotors. There is a door, starting just above the canards, which is meant to house the parachute. It is held shut by another servomotor, just barely visible in figure B.3a. In the same area as the parachute door, but from the other side, there is a space for the flight control unit and battery. This is visible in figure E.3.



Rocket

Stages: 1

Mass (with motor): 122 g

Stability: 1.45 cal

CG: 346 mm

CP: 403 mm

B6-2

Altitude	39.6 m
Flight Time	14.7 s
Time to Apogee	3.62 s
Optimum Delay	2.05 s
Velocity off Pad	6.36 m/s
Max Velocity	22 m/s
Velocity at Deployment	1.39 m/s
Landing Velocity	3.94 m/s

Motor	Avg Thrust	Burn Time	Max Thrust	Total Impulse	Thrust to Wt	Propellant Wt	Size
B6	3.18 N	1.34 s	6.79 N	4.29 Ns	2.66:1	5.6 g	18/70 mm

Figure B.1: Data-logging rocket design in OpenRocket

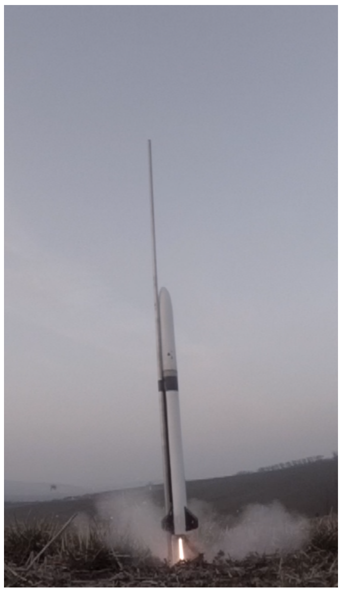
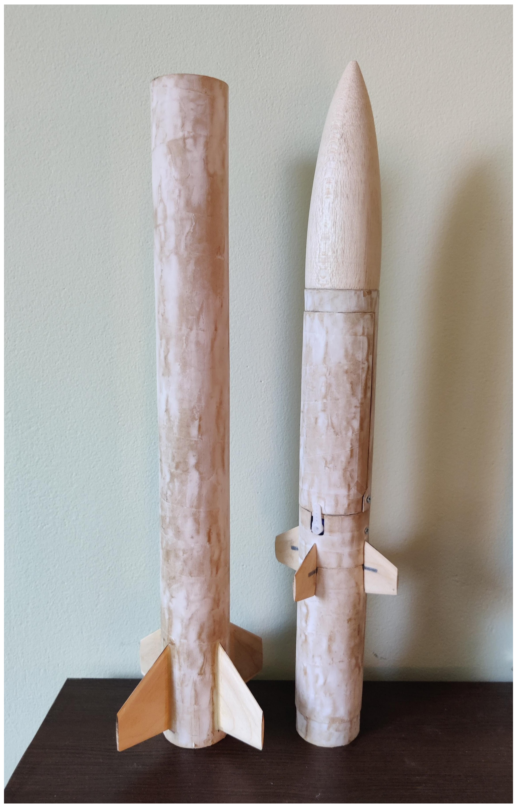


Figure B.2: Data-logging rocket on a guide rail just after ignition


(a) Actively controlled rocket in two halves, unpainted

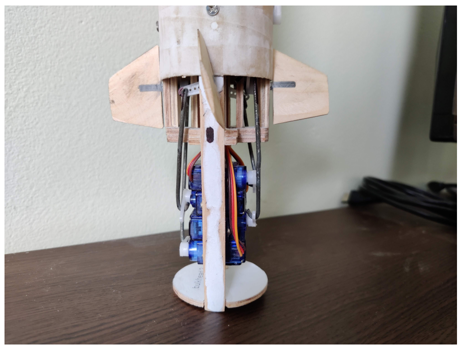

(b) Control mechanism of the actively controlled rocket

Figure B.3: Actively controlled rocket

Appendix C

MCU comparison

Table C.1: Microseconds to 1 000 000 iterations of certain operations on selected development boards

Board	Arduino Nano	ESP-WROVER-KIT	FRDM-K64F	Teensy 3.5	Nucleo-L432KC	Teensy 4.0
Float division	31692336	789958	127345	125189	187695	30003
Float multiplication	1050014	41988	50942	4174	75137	5009
Double division		2552238	5688187	5718754	7868026	53338
Double multiplication		470259	891383	1001795	963264	11669
8-bit addition	75466	33584	42443	50081	8764	5002
32-bit addition	943304	33588	42563	41738	75054	3335
Float addition	10087808	50371	50942	41738	75132	5001
Sqrt	34858944	4297146	10584643	10676101	17162523	73339
Sin	131939884	13194127	43813376	40459645	49398698	463558
Atan2	179115432	9324215	22158116	20369909	30846549	451693
Sensor fusion	17697124	15697628	35431761	36005082	59492979	3296033
Sensor fusion fast	18326444	12610983	28340745	2695529	41407385	996051

Table C.2: Feature comparison of considered MCU development boards

Board	Arduino Nano	ESP-WROVER-KIT	FRDM-K64F	Teensy 3.5	Nucleo-L432KC	Teensy 4.0
MCU	Atmega328P	ESP32-WROVER-B	MK64FN 1M0VLL12	MK64FX512	STM32 L432KCU6	IMXRT1062
Processor core	8-bit AVR	32-bit Xtensa dual-core	32-bit Cortex-M4	32-bit Cortex-M4F	32-bit Cortex-M4	32-bit Cortex-M7
FPU	No	No	32-bit	32-bit	32-bit	32-bit and 64-bit
Clock speed	16 MHz	240 MHz	120 MHz	120 MHz	80 MHz	600 MHz
FLASH memory	32 kB	4 MB	1 MB	512 kB	256 kB	1984 kB
(S)RAM	2 kB	320 kB	256 kB	256 kB	64 kB	1024 kB
EEPROM	1 kB	None	None	4 kB	None	1 kB emulated
UART	1	3	5	6	2	7
SPI	1	3	2	3	2	3
I2C	1	2	2	3	2	3
I/O pins	22	34	40	64	22	40
External input voltage	7–12 V	5 V	5–9 V	5 V	7–12 V	5 V
Logic level voltage	5 V	3.3 V	3.3 V	3.3 V	3.3 V	3.3 V
Size	45×18 mm	84×85 mm	87×53 mm	61×18 mm	50×15 mm	36×18 mm
Cost	\$2 – \$20	\$40	\$41	\$24.25	\$10	\$20
Programming models	Arduino	Arduino, ESP-IDF	Mbed OS	Arduino	Arduino, Mbed OS, CMSIS	Arduino
Built-in connectivity	No	802.11b/g/n, Bluetooth v4.2, BLE	Ethernet 10/100 Mbit/s	No	No	No
Debugging	No	FTDI chip	CMSIS-DAP	External J-LINK	ST-LINK	External J-LINK

Appendix D

Data-logging unit

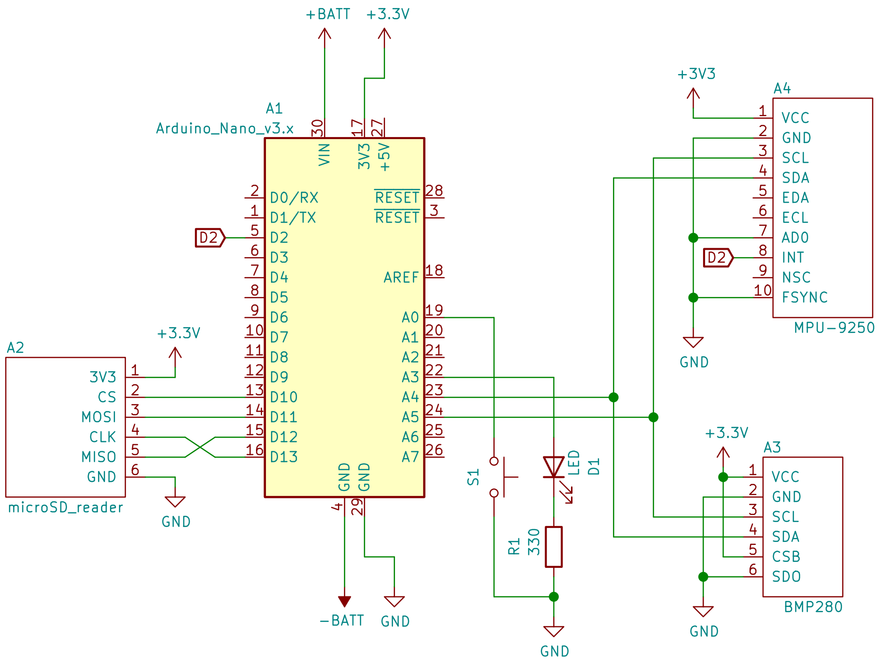


Figure D.1: Schematic design of the data-logging unit

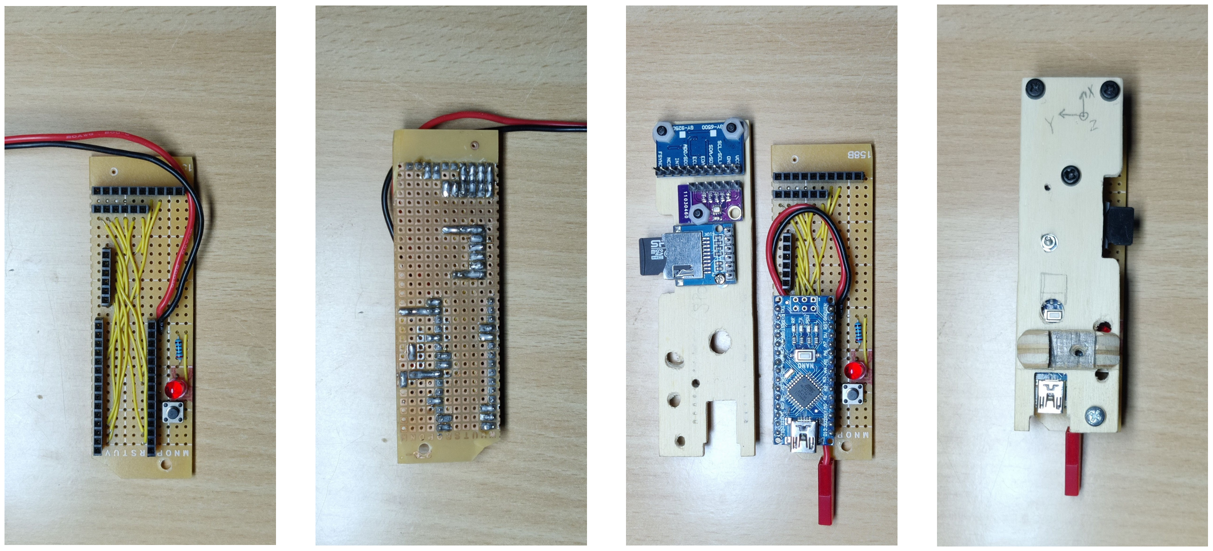

(a) Wiring (b) Back side (c) With sensors (d) Assembled

Figure D.2: Data-logging unit built on a prototype board

Appendix E

Flight control unit

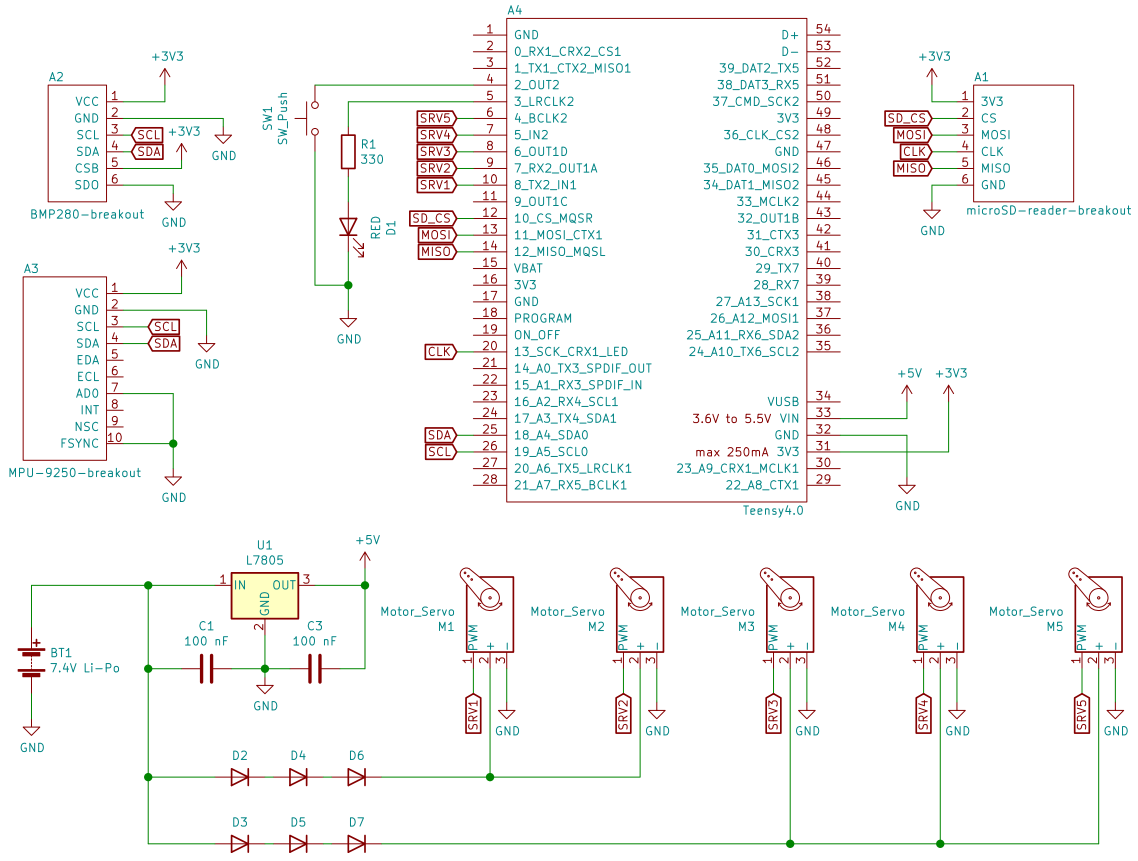
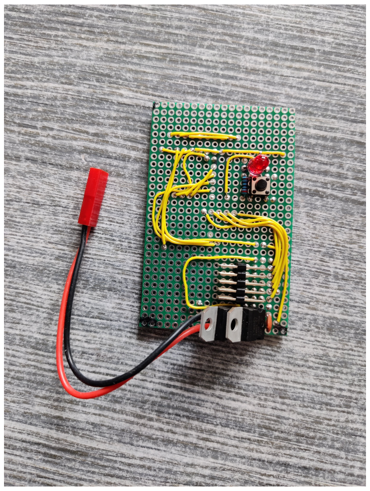
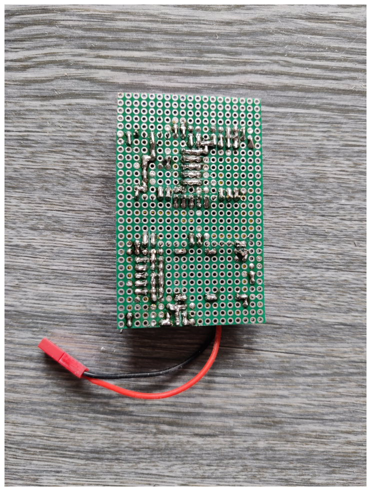


Figure E.1: Schematic design of the flight control unit


(a) Wiring (still with two L7805)


(b) Traces

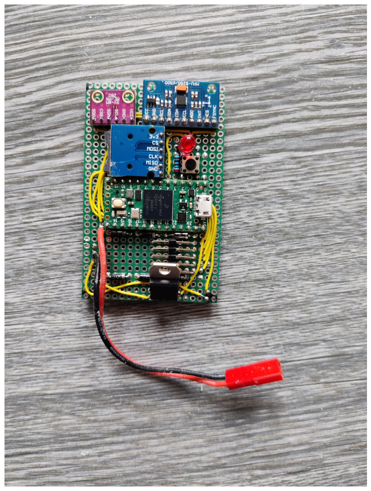

(c) Assembled unit

Figure E.2: Flight control unit assembly

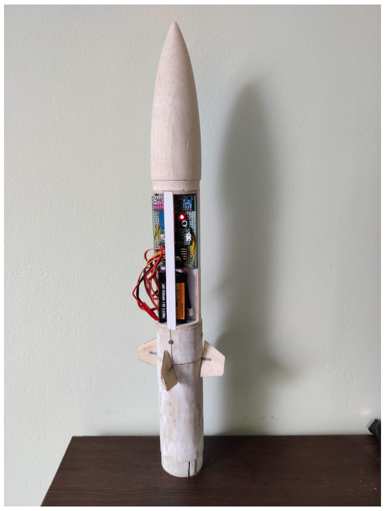


Figure E.3: Flight control unit fitted into the rocket payload section with battery and servomotors connected