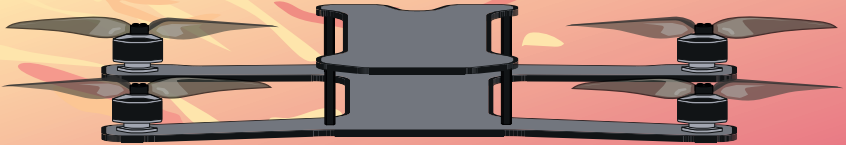


Carbon aeronautics



quadcopter build and programming manual



learn



100 %
hackable



< 250 g
weight



10 min
flight time

aeronautics

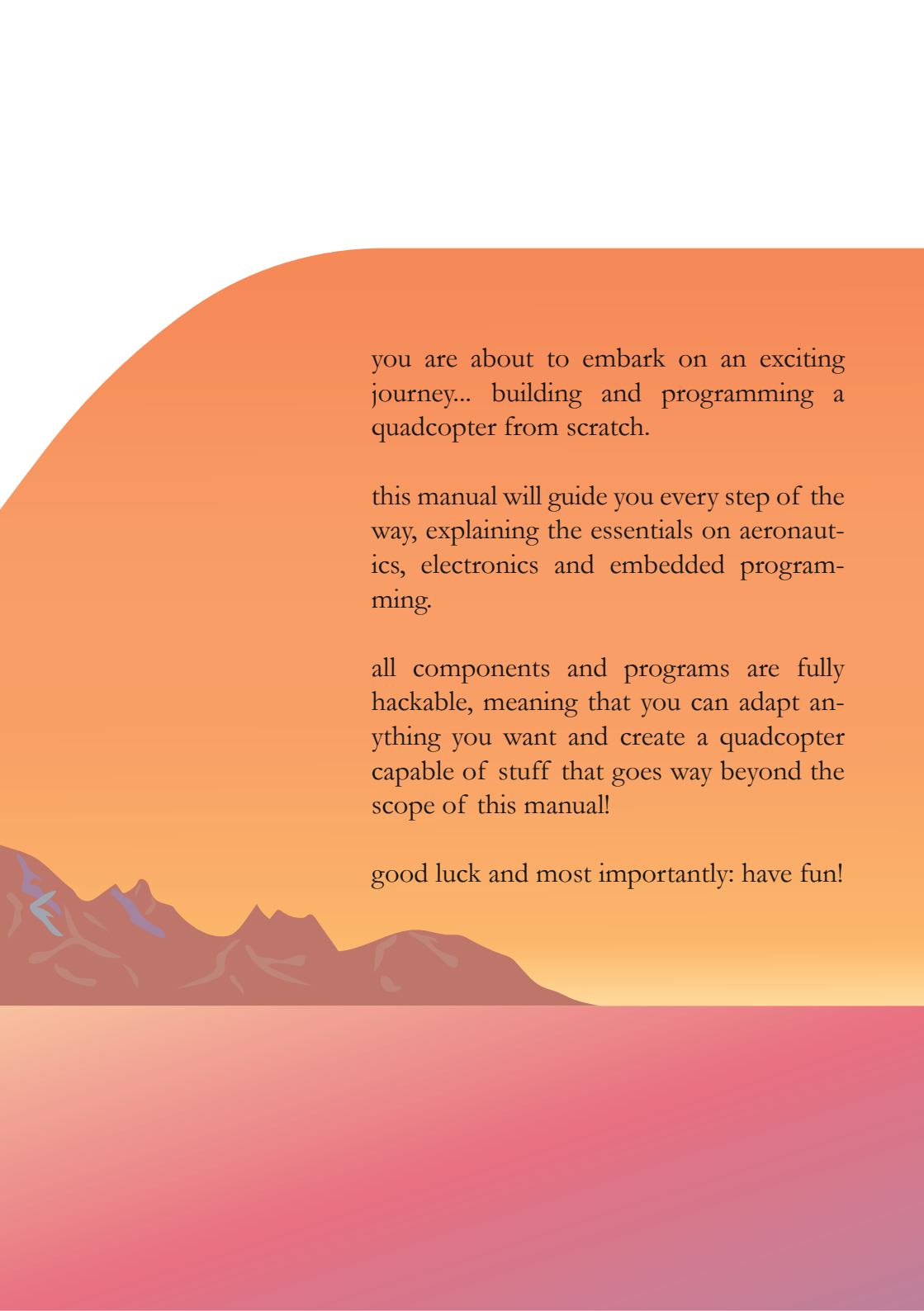
programming



electronics

Carbon aeronautics





you are about to embark on an exciting journey... building and programming a quadcopter from scratch.

this manual will guide you every step of the way, explaining the essentials on aeronautics, electronics and embedded programming.

all components and programs are fully hackable, meaning that you can adapt anything you want and create a quadcopter capable of stuff that goes way beyond the scope of this manual!

good luck and most importantly: have fun!

Carbon Aeronautics quadcopter build and programming manual

Project, text and figures by Laurens Raes

The contents of this manual are the intellectual property of the company *Carbon Aeronautics*. The text and figures in this manual are licensed under a Creative Commons Attribution - Noncommercial - ShareAlike 4.0 International Public Licence. This license lets you remix, adapt, and build upon your work non-commercially, as long as you credit *Carbon Aeronautics* (but not in any way that suggests that we endorse you or your use of the work) and license your new creations under the identical terms.

The information in this manual is provided "As Is" without any further warranty. Neither *Carbon Aeronautics* or the author has any liability to any person or entity with respect to any loss or damage caused or declared to be caused directly or indirectly by the instructions contained in this manual or by the software and hardware described in it. As *Carbon Aeronautics* has no control over the use, setup, assembly, modification or misuse of the hardware, software and information described in this manual, no liability shall be assumed nor accepted for any resulting damage or injury. By the act of use, setup or assembly, the user accepts all resulting liability.

This is not a toy but an educational product and not intended for persons below the age of 18 years old. The user is responsible for complying with the local regulations concerning unmanned aircraft when flying outdoors, and to fly in a responsible manner. This is a sophisticated product for advanced craftsman with previous experience in the field of electronics and programming. The purpose of the safety instructions and warnings in this manual is to attract your attention to possible dangers. They do not by themselves eliminate any danger, nor are they fully exhaustive. They are no substitutes for proper accident prevention measures or for the knowledge of the electric safety rules that are expected to be known by experienced craftsmen.

First edition, August 2022.

Contents

Project 1

Concept, parts and programming.....8

PART I: rate mode

Project 2

LED control.....24

Project 3

Reading your battery level.....28

Project 4

Sensing the rotation rate.....34

Project 5

Gyroscope calibration.....46

Project 6

Take your motors for a spin.....52

Project 7

Receiving commands.....58

Project 8

Controlling your motors.....66

Project 9

Battery management.....72

Project 10

Assembling your quadcopter.....80

Project 11

Quadcopter dynamics.....86

Project 12

Quadcopter rate control.....90

Project 13

The flight controller: rate mode.....96

Part II: stabilization mode

Project 14

Measuring angles110

Project 15

The Kalman filter - one dimension120

Project 16

The flight controller: stabilize mode130

Part III: velocity mode

Project 17

Measuring altitude142

Project 18

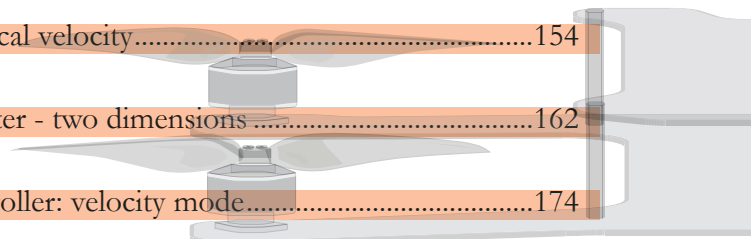
Measuring vertical velocity154

Project 19

The Kalman filter - two dimensions162

Project 20

The flight controller: velocity mode174



Part IV: quadcopter design and simulation

Project 21

Motor and sensor simulation190

Project 22

Quadcopter dynamics simulation200

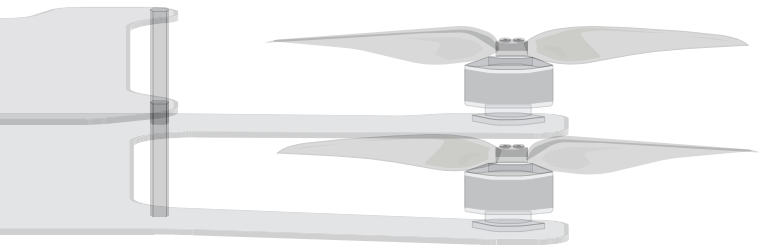
Project 23

Quadcopter PID controller210

Project 24

Estimate the PID values214

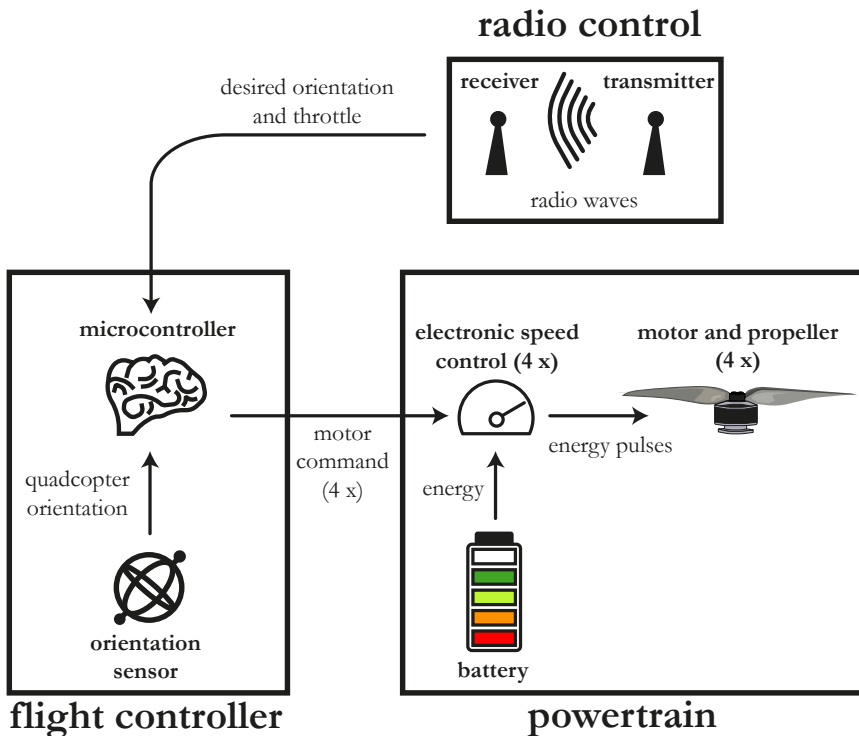
Part V: expanding your horizon





Project 1

Concept, parts and programming



Explore the basics of your quadcopter

Let's start your exciting journey in the world of aeronautics, electronics and programming with the concept behind the flying machine that you will build and the parts that you need. This manual will help you tackle the basics and enable you to build your own quadcopter; a drone with four motors.

The creation of flying machines is a true engineering challenge and involves solving several problems, from aerodynamics to power systems. In the case of a quadcopter, you rely on four motors and propellers to provide enough thrust to start flying. Obviously, these are not the only necessary components. The figure to the left displays the basic overview of a quadcopter with three major active building blocks:

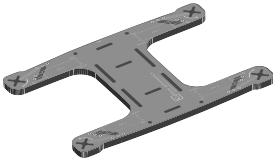
- The **radio control system**, which consists of a radiotransmitter and a receiver. The position of the sticks on the radiotransmitter are transformed into commands and subsequently sent to the receiver that is situated on your quadcopter.
- The **flight control system**, which consist of a microcontroller and some sensors. The bare minimum you need to stabilize the quadcopter is an orientation sensor, but you can add various other sensors (barometer, GPS, ultrasonic,...) to make your flight easier. The information of your sensor and the commands from your radiotransmitter are then processed in the microcontroller, which is the brain of your quadcopter. The microcontroller calculates the optimal speed of each of the four motors to keep the quadcopter in the air.
- The third building block is the **powertrain**, which is the high current part of the quadcopter. The battery is the power source of the whole system and sends energy in the form of electrical current to four electronic speed controllers (ESCs); an ESCs converts the provided current into current pulses, with a pulse length proportional to the motor command sent from the microcontroller. This gives a motor speed proportional to the motor command and in turn, a certain thrust allowing you to take off!

And basically, that's all there is to it! With the general idea behind your quadcopter clearly understood, let's have a look at all different physical parts that you will use. Your quadcopter consists of three active building blocks; a radio control system, flight controller and powertrain. Moreover, you also need a frame on which you can mount all these active components. Some auxiliary parts are also necessary, to charge the battery and test your microcontroller, sensors and powertrain before fixing them to the frame.

All necessary components are listed below and divided into parts for the frame, flight controller, powertrain, battery and radio control. Each part is available on the consumer market, so if you break a part during flight or you want to change parts, you can easily buy it yourself. This manual is designed to guide you building your own quadcopter while enabling you to change any aspect of it as well.

1 frame

1a lower quadcopter frame
CarbonAeronautics



1x

1b upper quadcopter frame
CarbonAeronautics



1x

1c frame spacers
M3 x 30 mm



4x

1d spacer fastening screws
M3 x 6 mm



12x

1e battery strap
210 mm



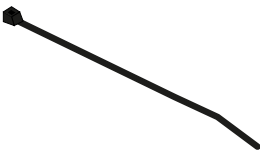
1x

1f landing pad



4x + 1 reserve

1g cable ties
16 mm



6x + 4 reserve

1h standoff spacer
M3 x 20 mm




4x + 2 reserve

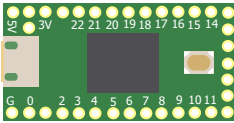
1i cable protector
500 mm



1x

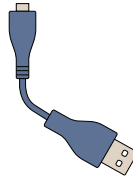
2 flight controller

2a microcontroller
Teensy 4.0 



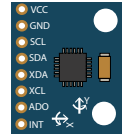
1x

2b microcontroller connector
USB A to micro B



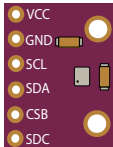
1x

2c orientation sensor
GY-521 MPU-6050



1x

2d barometer
GY-BMP280



1x

2e sensor fastening screws
M3 x 20mm



4x

2f sensor locknuts
M3



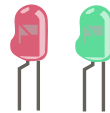
4x

2g sensor full nuts
M3



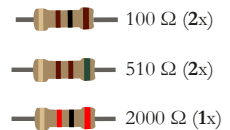
12x

2h green and red LED

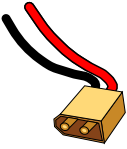


1x + 1x

2i resistors

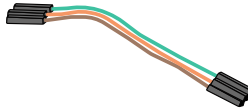


2j battery connector
XT60



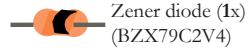
1x

2k jumper wires
female to female 10 cm

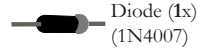


3x + 3 spares

2l diodes



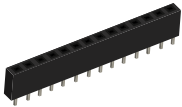
Zener diode (1x)
(BZX79C2V4)



Diode (1x)
(1N4007)

1x

2m female headers
40 pins - 2,54 mm



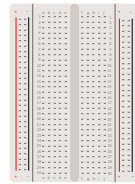
2x

2n male headers
40 pins - 2,54 mm - right angle



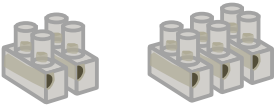
2x

2o breadboard
400 points



1x

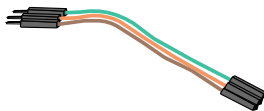
2p wire terminal strip



1x

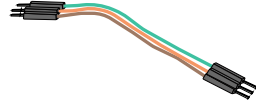
1x

2q jumper wires
male to female 10 cm



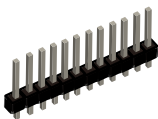
3x + 3 spares

2r jumper wires
male to male 10 cm



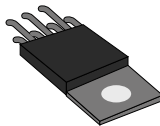
40x

2s male headers
40 pins - 2,54 mm - straight



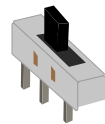
2x

2t power switch
BTS50080-1TMB



1x

2u slide switch
OS102011MS2QN1C



1x

3 powertrain

3a **motors**
GEPRC GR1105
5000 kV



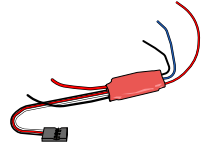
4x

3b **motor fastening screws**
M2 x 4 mm



16x

3c **Electronic speed controllers**
HobbyKing 6A
ESC with BEC



4x

3d **clockwise propellers**
Gemfan 3018R



2x + 2 reserve

3e **counter-clockwise propellers**
Gemfan 3018



2x + 2 reserve

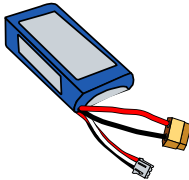
3f **propeller fastening screws**
M2 x 8 mm



4x + 8 reserve

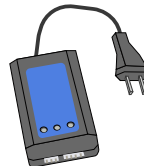
4 Battery

4a **Batteries**
Turnigy 2S 1300 mAh



2x

4b **battery charger**
Hobbyking B3AC



1x



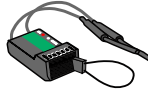
5 Radio control

5a Radiotransmitter
Flysky FS-i6



1x

4b receiver and bind plug
Flysky FS-iA6B



1x

Alternative parts

You are not limited to the parts that are described in this paragraph and chances are you want to choose different components for various reasons such as higher thrust, longer flight time or lower weight. The parts that can easily be swapped are the propellers, ESC (Electronic Speed Controller), motor and battery. To give you an idea of the possibilities, this paragraph describes some successfully tested variations on the basic quadcopter. The PID values derived later on in this manual are a good match for all variations, but remember that the weight of your quadcopter will be affected: the basic quadcopter weighs 247 gram while the combination of all the heaviest components described in this paragraph weighs 278 gram.

The **battery** is perhaps the easiest interchangeable component. The base part is a 2S battery with a capacity of 1300 mAh and a weight of 70 gram. Other tested possible batteries include a 2S battery with 1000 mAh (weight: 60 gram) or a 2S battery with a 1500 mAh capacity (weight: 80 gram). Additional capacity comes at a cost in the form of extra weight and thus a less flexible quadcopter.

The choice of your **ESC** and **motor** combination needs some more care. You should make sure that the maximal load current of both your ESC and motor are similar: the part with the smallest load current limits the load current of both components. Since a motor or ESC with a higher load current generally weighs more, the optimal combination consists of motors and ESC with similar load current. The base motor and ESC combination (GEPRC GR1105 5000 kV and Hobbyking 6A ESC with BEC) both have a load current of around 6 A. Another tested possibility is the combination of the GEPRC GR1206 4500 kV and Hobbyking 12A ESC, both having a load current of around 12 A. The 6 A combination of four motors and ESCs weighs around 50 gram, while the 12 A combination weighs 66 gram.

Another important value is the motor kV rating: this determines how fast the propeller can turn at full throttle: a 5000 kV motor turns at 5000 rpm/V. Since a 2S battery has a nominal voltage of 7.4 V, this equals to a nominal rpm of $5000 \text{ rpm/V} \times 7.4 \text{ V} = 37\,000 \text{ rpm}$. To lift a quadcopter with a weight between 200 and 300 gram, you need a motor with a kV rating between 4000 and 6000, depending on the propeller.

Once you have chosen your ESC and motor combination, the **propellers** are next. A larger propeller generates more thrust: this is great because it means your quadcopter can weigh more and can successfully combat stronger wind gusts. However, the necessary load current increases and your motor and ESC have to withstand this higher current, otherwise they will overheat and possibly start burning. Larger propellers obviously weigh more as well.

Gemfan propeller (weight in gram for 4 props)	3018 (2 blade) 3.4 g	3035 (3 blade) 5.6 g	4024 (2 blade) 6.4 g	4019 (3 blade) 8.4 g
GEPRC GR1105 5000 kV + Hobbyking 6A ESC with BEC	5 A	7 A	9 A	12 A
GEPRC GR1206 4500 kV + Hobbyking 12A ESC with BEC	5 A	7 A	9 A	12 A

Four different propellers are tested with the two motor/ESC combinations and tabulated above together with their current at full throttle with a full 2S battery. The colour code can be explained as:

- **Green:** the motor and ESC can withstand full throttle with this propeller for longer periods of time - this combination is suitable for beginners.
- **Orange:** the motor and ESC can only withstand short bursts of full throttle with this propeller - this combination is only suited for experienced flyers given the risk of motor or ESC overheating.
- **Red:** this combination is not recommended given the high risk of motor and ESC overheating.

For reference, the **maximal dimensions** for the battery and the propellers given the frame width are included here as well:

- 10.2 cm is the maximal diameter of the propeller (corresponds with a 4 inch propeller).
- The battery bay has a 12 cm x 4 cm x 3 cm dimension, but you need to leave some space for the receiver, electronic cables, protectors and screws. This limits the practical space to 8.5 cm x 3.3 cm x 1.5 cm.



Additional required tools and material

To complete your build, you also require some additional tools and material. Except for a computer, these are only necessary when starting the actual build, not when testing the components in the first projects (except if you still need to solder headers to your Teensy and sensors in order to test them on the breadboard).

- a **soldering iron** or station, to solder the motors wires, ESCs, resistors, LEDs and male/ female headers to each other / the printed circuit frame on you quad-copter frame.
- sufficient **solder material**.
- a **soldering helping hand** to clamp the parts you are soldering together.
- a **wire stripper** to strip the electrical insulation from the ESC and motor wires.
- a **wire cutter** to cut the ESC and motor wires.
- a **computer** capable of running Arduino (see arduino.cc/en/software)
- two **hex keys** (1.5 mm and 2 mm)
- a **multimeter** to check for short-circuits or bad connections.

General safety instructions

Battery

- Read the battery and battery charger manuals carefully before use.
- Never charge the batteries unattended.
- Before connecting the battery and your motor(s) or quadcopter for the first time, make sure there are no short-circuits between your soldered components using your multimeter.

Electronics

- Before connecting the electronics to a power source (such as your computer), make sure that there are no short-circuits between your soldered components using your multimeter.
- Remove the propellers and do not touch the motors unless you are sure that your program is working properly to avoid losing control over your quadcopter.
- Never run your motors without propellers.

Before flying

- Make sure the failsafe and safety-related code lines are implemented and working correctly.
- Check the regulations that are applicable in your country (with regard to maximal altitude, speed, weight,...) when flying your unmanned quadcopter outdoors.

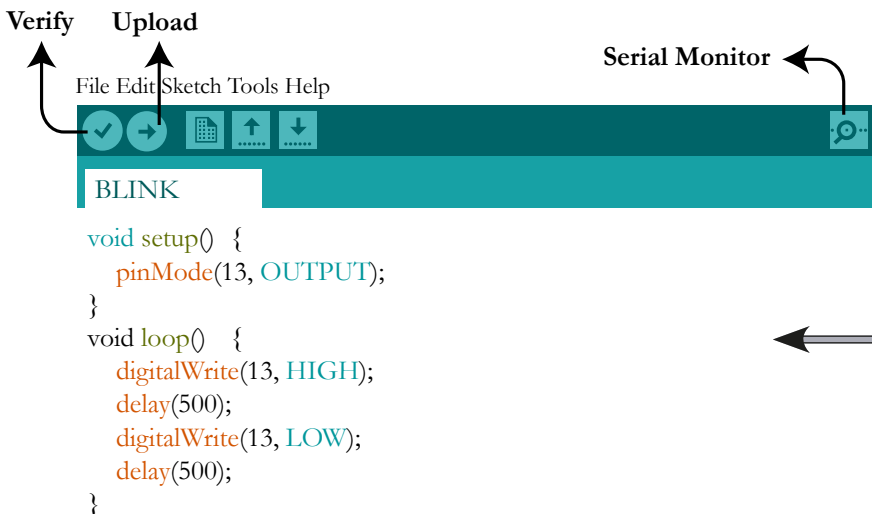


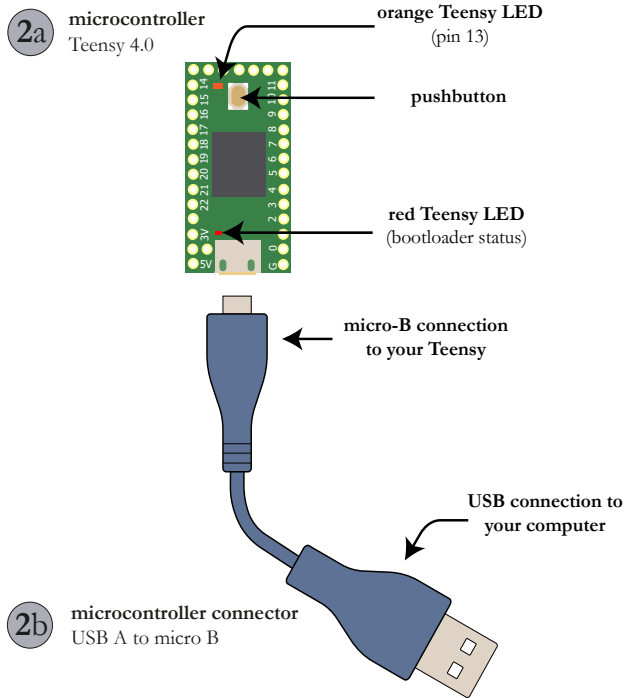
Setup your microcontroller for programming

The core of your quadcopter project is the Teensy microcontroller that you will program in such a way that it becomes the flight controller and thus brains of your project. The Arduino software will be used to program the microcontroller, together with Teensyduino.

You can find all information with regard to the installation of the necessary software on the website of the Teensy manufacturer: www.pjrc.com/teensy. The installation steps will be described here as well, but please refer to the pjrc and arduino websites if you need additional troubleshooting.

1. Connect your new Teensy to your computer using the USB cable (see figure to the right).
2. Your Teensy should come with the LED blink program pre-loaded; this means that the orange LED on your Teensy should blink slowly after connection with your computer.
3. Press and release the tiny pushbutton on the Teensy. The orange blinking LED should stop and the red Teensy LED should be visible. This means your Teensy works correctly.
4. Disconnect your Teensy from your computer by disconnecting the USB cable.





5. Download and install the Teensy Loader program, which communicates with your Teensy board. Guidance on the installation process can be found at pjrc.com/teensy/loader.html. Click on the operating system of your computer, read the information and click on the Teensy Loader link to start downloading.

6. If you do not have the Arduino software (IDE) yet, download the latest version from arduino.cc/download and install it on your computer. Guidance on the installation process can be found at arduino.cc/en/Guide/Windows or arduino.cc/en/Guide/MacOSX or arduino.cc/en/Guide/Linux.

7. The final piece of software to install is Teensyduino, the software add-on for Arduino. Download it by going to pjrc.com/teensy/td_download.html and follow the instructions on this webpage.

8. Open the Arduino IDE; a new empty sketch should load automatically. Copy the code in the figure to the left of this page and save the file under the name BLINK. Now click on 'Verify'. You will first have to save your sketch. After verification, you should view the message 'Done Compiling' below on your screen. If you get an error, verify whether you copied the code correctly.



9. Before you can upload your verified code to your Teensy, you need to setup your Teensy in the Arduino IDE. Go to tools and:

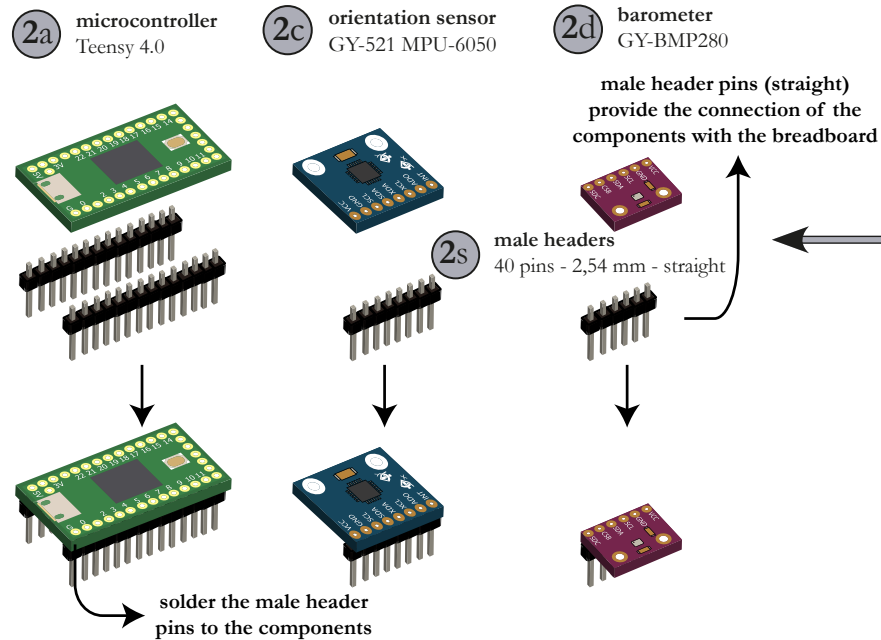
- Click on 'Boards' and 'Teensyduino' and select the Teensy 4.0 board.
- Verify that the USB type is 'Serial'.
- Verify that the CPU speed is 600 MHz.
- Connect your Teensy again with your computer using the USB cable. Under Port, a USB port should be displayed. Click on it.

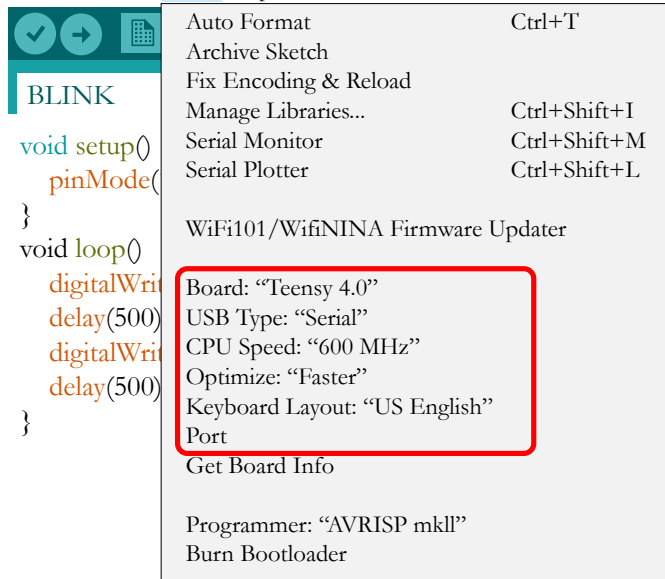
10. Press the upload button on the screen. The internal Teensy LED should start blinking again. Change the blinking speed by changing the delay time of 500 (milliseconds) in the code to for example 100 (milliseconds) to blink faster, or 1000 (milliseconds) to blink slower. Adapt and upload the code to verify that you are truly in control of the Teensy. When this test is successful, you are ready for the next project!

Code compatibility

The code throughout this book is compatible with the following Arduino (library) versions:

- Arduino IDE: 1.8.16
- Teensyduino: 1.55
- BasicLinearAlgebra library: 3.2.0 (only necessary for part III)





The screenshot shows the Arduino IDE interface with the 'Tools' menu open. The menu items are as follows:

- Auto Format (Ctrl+T)
- Archive Sketch
- Fix Encoding & Reload
- Manage Libraries... (Ctrl+Shift+I)
- Serial Monitor (Ctrl+Shift+M)
- Serial Plotter (Ctrl+Shift+L)
- WiFi101/WifiNINA Firmware Updater
- Board: "Teensy 4.0"**
- USB Type: "Serial"**
- CPU Speed: "600 MHz"**
- Optimize: "Faster"**
- Keyboard Layout: "US English"**
- Port
- Get Board Info
- Programmer: "AVRISP mkII"
- Burn Bootloader

The code in the background is:

```

BLINK
void setup()
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop()
  digitalWrite(LED_BUILTIN, HIGH);
  delay(500);
  digitalWrite(LED_BUILTIN, LOW);
  delay(500);
}

```

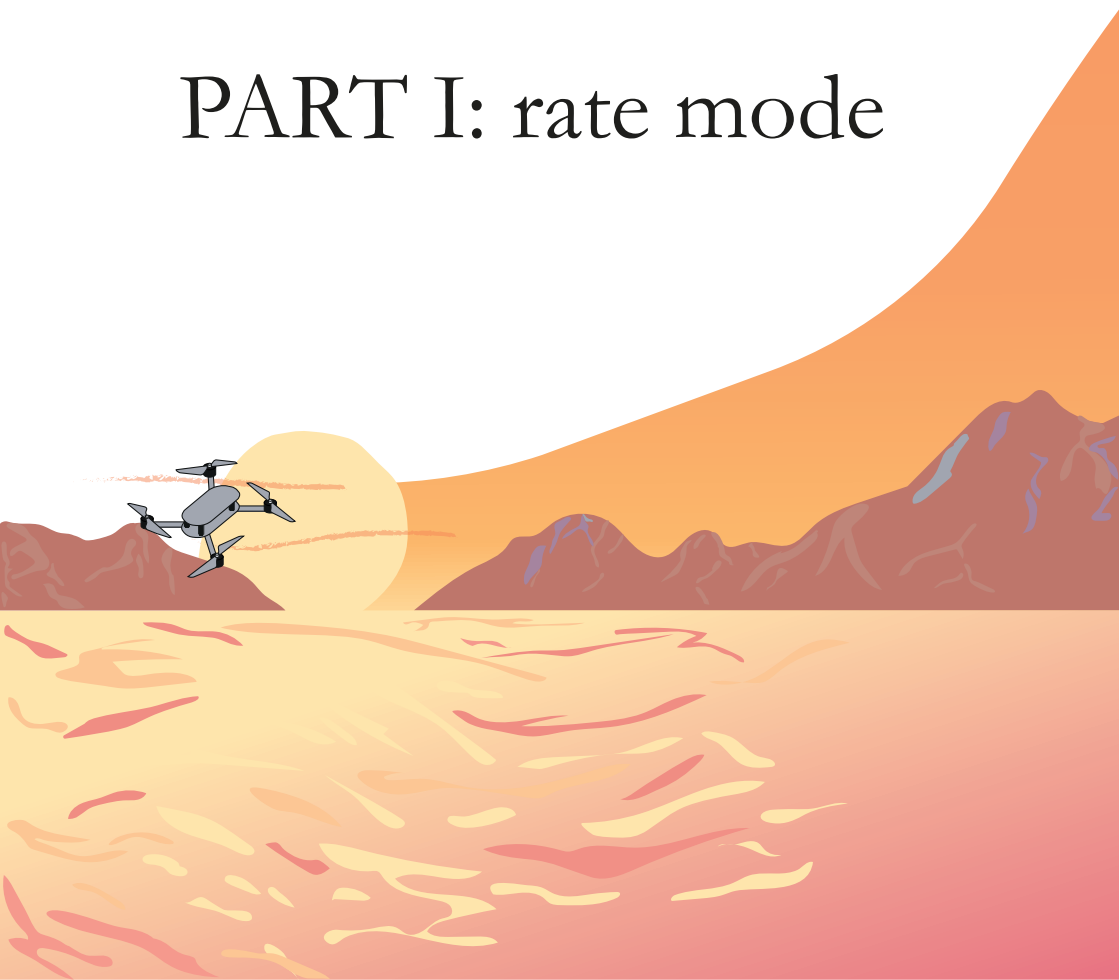
Solder pins to your microcontroller and sensors


You will use a breadboard to separately test the electronic components of your flight controller. To be able to electrically connect the components with the breadboard, you need to use straight male header pins that are soldered to your Teensy microcontroller, the MPU-6050 gyroscope and the BMP-280 pressure sensor. If these parts do not come pre-soldered with header pins, you will need to solder them yourself.

For easy soldering, you can insert the pins in your breadboard and put the component on top such that the pins are soldered straight to the microcontroller and sensors. If you have never soldered before, you can consult the internet for some tutorials.



PART I: rate mode





in the first part, you will build your quadcopter and program a flight controller that enables you to fly in rate mode; this is the easiest-to-implement controller that gives you full control over the performance of your quadcopter.

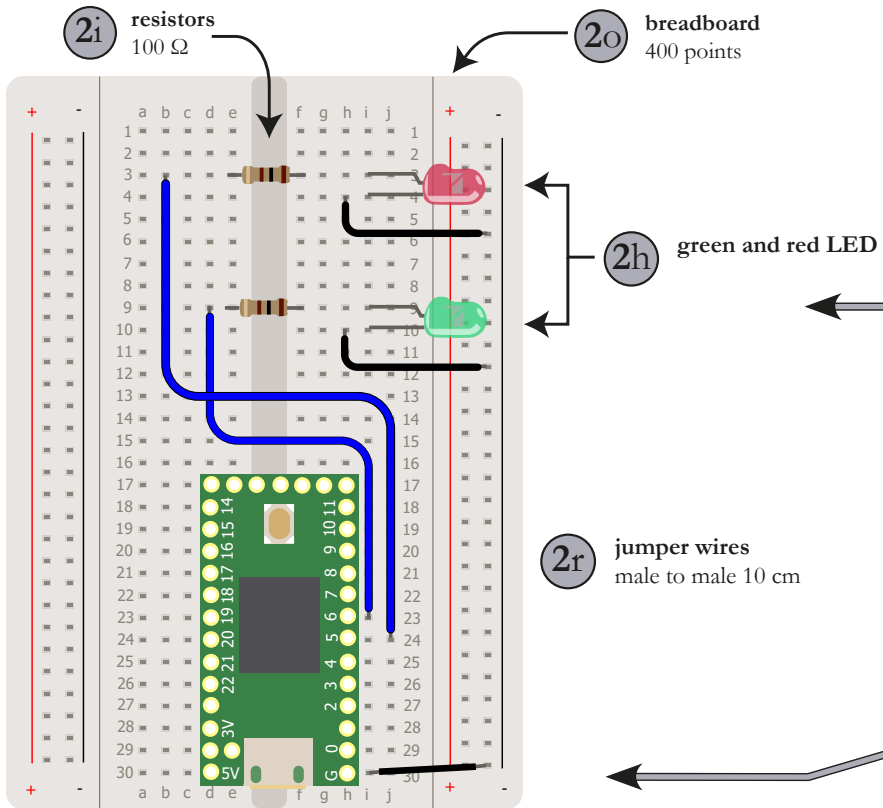
complex projects such as this one are often cut in smaller, independent pieces that are tested separately, before all components are put together.

you will follow this approach and start with simple building blocks and code, to eventually arrive at the full build and flight code.



Project 2

LED control



Use LEDs to receive feedback

Throughout this manual, you will learn how to communicate with your quadcopter by giving it commands. However, this communication goes only one-way from your radiotransmitter to the quadcopter. Sometimes it is useful to receive some feedback from the quadcopter, for example when the setup and calibration process is finished or when the battery voltage becomes low. To do this in an easy way without telemetry, you will use three signal LEDs.

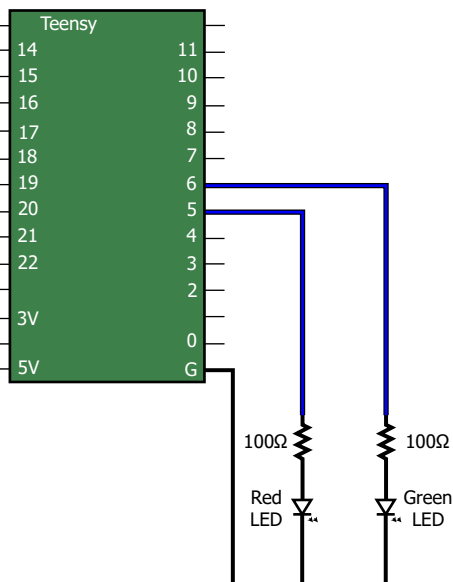
The first led you will use is the internal led of the Teensy, which you already experimented with. This orange led is controllable through pin 13 and requires no additional circuit building. Lighting this led can be useful to show that the microcontroller receives power and is working correctly.

You will also use two additional external LEDs to signal the start and end of the setup program. Before you are able to fly, the microcontroller will have to start the auxiliary sensors and calibrate them. This takes about four seconds during which the quadcopter is not yet able to start. During this time, you turn on the red LED to signal that the quadcopter is still in the setup process. When the setup process is successfully finished, you turn off the red LED and turn on the green LED. Let's start to build the electronic circuit necessary to light these external LEDs.

Connect two 100Ω resistors to pins 5 and 6 of your Teensy using jumper wires. Pin 6 gives signals to the red LED while pin 5 gives signals to the green LED. Connect the long leg (+ side or anode) of each LED with the resistor and the short leg (- side or cathode) with the negative bus line.

Configure your breadboard such that the ground G of the Teensy is connected to the negative bus line as well.

The schematic view of this circuit is shown to the right. You are now ready to program your Teensy and to give signals to each LED.



Coding

All arduino sketches consist of both a setup and a loop part. The code in the setup part of the sketch only runs once, during startup of the microcontroller. The code in the loop part of the sketch runs continuously when the setup part is finished.

As seen in the previous project, you can control the internal orange Teensy LED with pin 13. Configure the pin as an output using the command `pinMode()` and use the command `digitalWrite()` to give it the command `HIGH`, which will light the orange LED and show that the microcontroller is powered and working.

To control the external LEDs, you will use the same commands. You already connected the red LED to pin 5 and the green LED to pin 6. To show that the setup process is ongoing, you light up the red LED by giving it the `HIGH` command.

Now wait four seconds (=4000 milliseconds) using the `delay()` command in order to simulate the setup process, which will take around four seconds to be completed in your final quadcopter code.

To indicate that the setup process is finished, turn off the red LED using the command `LOW` and subsequently turn on the green LED.

The code in the loop part runs continuously. Because you do not write any commands in this part, the green LED will be continuously illuminated as demanded in the last line of the setup part.

Testing

Upload your new code to your Teensy using the USB cable and verify that all LEDs light up in the correct order. Only the green LED should remain on after four seconds.

```
1 void setup() {
```

Initialize the setup part

```
2     pinMode(13, OUTPUT);  
3     digitalWrite(13, HIGH);
```

Turn on the internal LED

```
4     pinMode(5, OUTPUT);  
5     digitalWrite(5, HIGH);
```

Turn on the red LED

```
6     delay(4000);
```

Wait 4 seconds

```
7     digitalWrite(5, LOW);  
8     pinMode(6, OUTPUT);  
9     digitalWrite(6, HIGH);  
10 }
```

Turn off the red LED and turn on the green LED

```
11 void loop() {  
12 }
```

Start the loop part

Understanding digital output pins

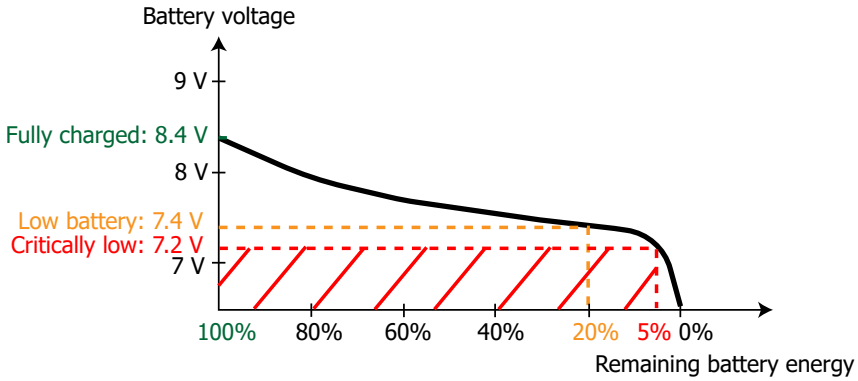
You used the pins 5, 6 and 13 as digital pins, meaning that their output voltage is binary: either HIGH (3V) or LOW (0V). This very simple command is sufficient to turn on a LED (when the 3V voltage is applied) or turn off the LED (when the 0V voltage is applied). The current necessary to light the LEDs is provided with the resistors you placed in series; through Ohm's law, you can calculate that a voltage of 3V and a resistance of $100\ \Omega$ gives a current of $3V/100\ \Omega=0.03$ Ampere or 30 mA.





Project 3

Reading your battery level



The evolution of the battery level in function of the battery voltage is displayed by the figure above. It is important to notice that discharging your battery to a too low voltage can degrade the battery and lead to a reduced capacity over time. Therefore, a good guideline for prolonged battery lifetime is to not discharge your 2S battery below its nominal voltage of 7.4V. Because the battery voltage fluctuates during flight and can drop temporarily when you suddenly increase the throttle, your flight controller will check if the voltage is above 7.5V before starting the motors.

Now how can you measure the voltage of the battery? Easy: the voltage applied to any pin of your microcontroller can be read digitally. Unfortunately, there is one catch: the pins of the Teensy are only 3.3V-tolerant, meaning that applying a voltage higher than 3.3V can damage the microprocessor. Therefore, you need to use a **voltage divider**: this electronic circuit divides the voltage of the battery to a value low enough to be used by your Teensy. Consider the first circuit displayed on the right: through Ohm's law, the current I is equal to the battery voltage V_{battery} divided by the resistance R_1 .

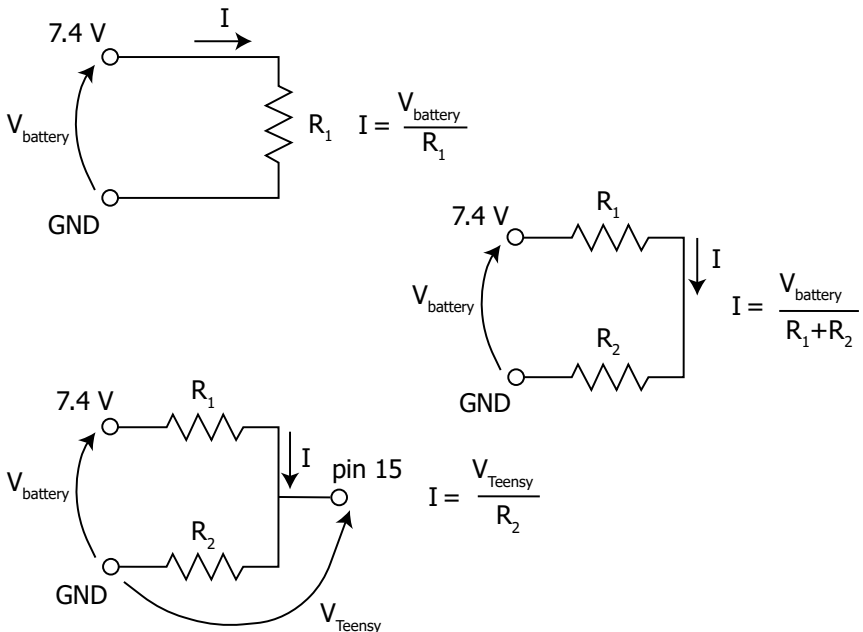
In the second circuit, a second resistance R_2 is used. The battery voltage is now equal to the current divided by the sum of two resistances. With the third circuit, you connect a pin of the Teensy between both resistances.

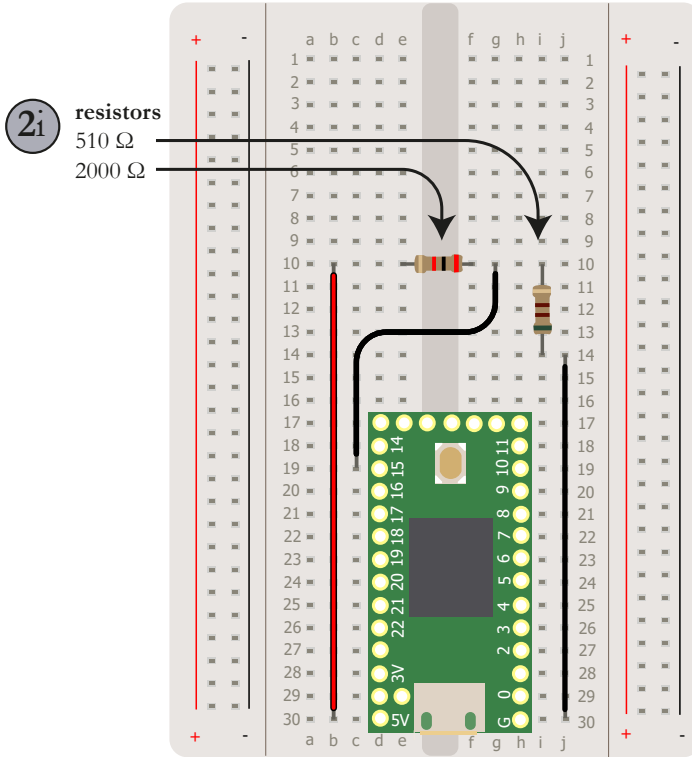
Learn to measure voltage and battery lifetime

A critical part of your quadcopter is the battery; it stores enough energy to let you fly for quite a while. But how do you know when the battery is almost empty? In this project, you will learn how the battery voltage drops during the flight and measure it in order to estimate the remaining battery lifetime.

The battery you use in this project is a 2 cell lithium-polymer battery, where the cells are placed in Series (=2S). Each cell has a nominal voltage of 3.7V and since the cells are placed in series, the total nominal voltage is equal to 7.4V. A 3S battery would give you $3 \times 3.7 = 11.1\text{V}$. The nominal voltage is the reference voltage of the battery, but you will always charge the battery up to the charge voltage, which is equal to 8.4V for a 2S battery.

When using a fully charged battery to fly your quadcopter, the battery voltage will drop from the charge voltage of 8.4V to the nominal voltage of 7.4V and even lower when you use more energy. This is inevitable and results in a lower thrust over time, because the speed of the motors is proportional to the provided voltage. Fortunately you can use this property also to your advantage, because by measuring the battery voltage you are able to estimate the remaining battery energy.





Coding

Lets first declare the voltage as a floating point number. To be able to measure the voltage multiple times without rewriting the same lines of code over and over, you will create the function `battery_voltage`. This function can be called as often as you want.

The analog voltage over pin 15 can be measured using the function `analogRead()`. Since the default resolution for `analogRead` is equal to **10** bit, a voltage of 0V gives you the digital number 0 and the maximal input voltage of 3.3V gives the digital number $2^{10}-1=1023$. Moreover you have built a 1:5 voltage divider. This means that the battery voltage is equal to the measured 3voltage divided by $1023 / (3.3 \times 5) = 62$.

You will visualize the voltage at pin 15 in real-time on your computer with the serial monitor. Set the speed at which the Teensy communicates with your laptop to 57600 bits per second.

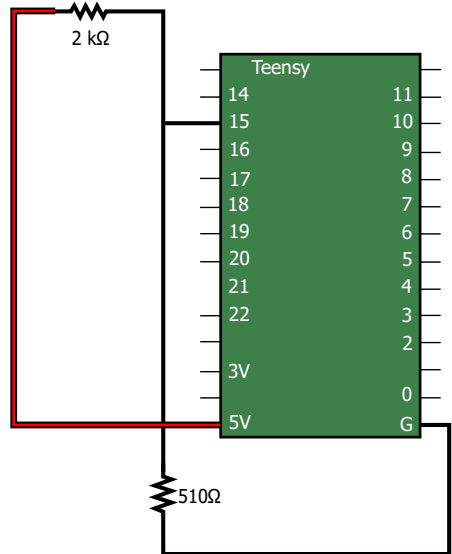
The voltage applied to the pin of the Teensy (which will be pin 15) is equal to the current divided by the second resistance R_2 . Since the current I will be the same for the second and third circuit, the following equation holds:

$$\frac{V_{battery}}{R_1 + R_2} = I = \frac{V_{Teensy}}{R_2}$$

$$V_{Teensy} = V_{battery} \cdot \frac{R_2}{R_1 + R_2}$$

By choosing the value of R_1 to be equal to 2000Ω and the value of R_2 to be equal to 510Ω , V_{Teensy} becomes equal to $V_{battery}$ divided by 5. You have now designed a 1:5 voltage divider! With a battery voltage of $8.4V$, the voltage measured by your Teensy equals $1.7V$, low enough to respect the $3.3V$ tolerance of the Teensy pins.

To test your circuit, you will not yet connect your battery but use the $5V$ output pin of the Teensy as voltage source, and measure this value with pin 15 and your new voltage divider. Connect the $5V$ pin with a 2000Ω resistor to pin 15 and the ground pin with a 510Ω resistor to pin 15 as shown on the figure to the left. You are now ready to code.



```

1 float Voltage;
2 void battery_voltage(void) {
3     Voltage=(float)analogRead(15)/62;
4 }

```

Read the battery voltage

```

5 void setup() {
6     Serial.begin(57600);

```

Setup the serial monitor



Measure the voltage each 50 milliseconds and print it to the serial monitor, with each time the unit V behind.

Testing

Upload the code and open the serial monitor (Ctrl+Shift+M or click on the serial monitor icon) with the USB adapter still connected to your Teensy. To see values that make sense, you should set the baud rate such that it corresponds with the baud rate that you have chosen in the code, namely 57600 baud. Now you should see the measured values, who will be more or less equal to 5V. When you connect the battery in a later stage, the measured voltage will vary between 8.4V and 7V.


```
7     pinMode(13, OUTPUT);
8     digitalWrite(13, HIGH);
9 }
10 void loop() {
11     battery_voltage();
12     Serial.print(Voltage);
13     Serial.println("V");
14     delay(50);
15 }
```

Print the battery voltage to the serial monitor

1. upload code

2. open serial monitor

File Edit Sketch Tools Help

Teensymonitor

15:22:15.581	->	4.88V
15:22:15.628	->	4.88V
15:22:15.674	->	4.89V
15:22:15.721	->	4.87V
15:22:15.768	->	4.88V
15:22:15.815	->	4.88V
15:22:15.861	->	4.89V
15:22:15.908	->	4.87V

Newline 57600 baud clear output

4. check measured values

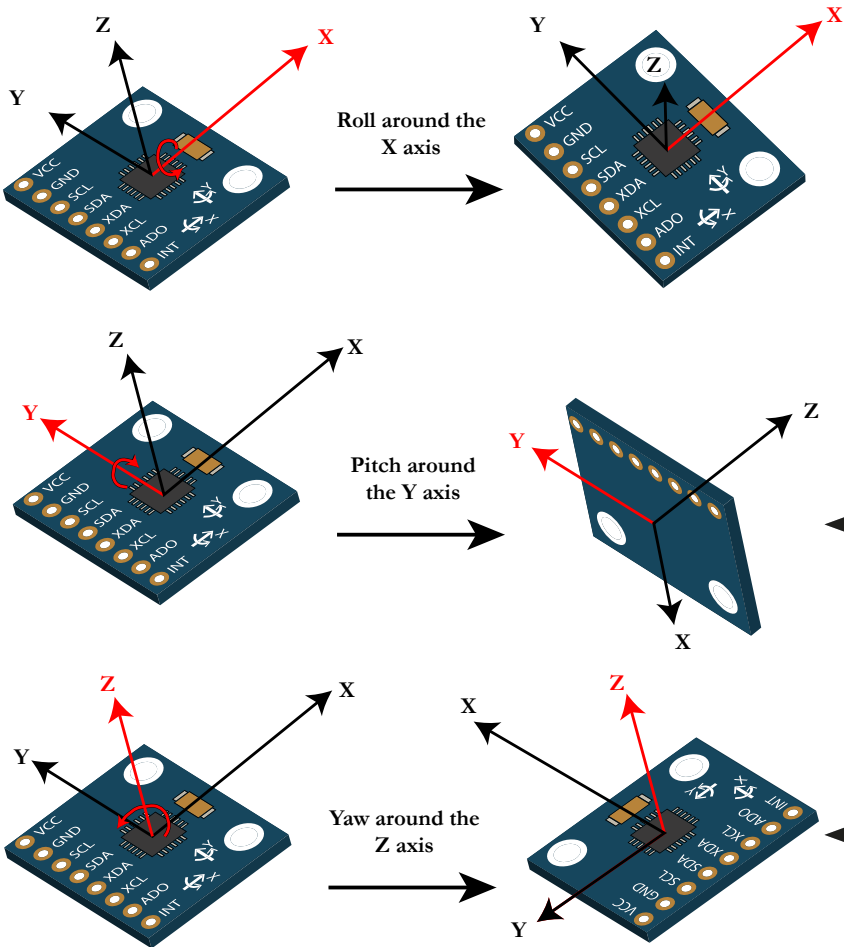
3. check baud rate





Project 4

Sensing the rotation rate



Measure the rotation of your quadcopter

To stabilize your quadcopter, you need measurements of its three-dimensional orientation. In rate control mode, it is sufficient to know the rotation rates when rolling, pitching and yawing. A sensor able to record these rotation rates is called a gyroscope. During this project, you will learn how to read the data sent from your gyroscope.

The gyroscope that you will use is included in the MPU-6050, a low-cost off-the-shelf orientation sensor. While the MPU is not a very precise sensor, its accuracy is sufficient to get great results balancing your quadcopter. You will use the gyroscope to measure the roll rate, pitch rate and yaw rate: this means that you do not measure absolute angles in degrees ($^{\circ}$), but rather angular rates in degrees per second ($^{\circ}/s$). An angular rate of $30^{\circ}/s$ for example, means that you rotate 30° each second and will perform a full 360° rotation in 12 seconds ($360^{\circ} / (30^{\circ}/s)$). You will learn later on that you can use these rotational rates to keep the quadcopter balanced; for example when you want the drone to stay at its current orientation, its angular rate needs to be $0^{\circ}/s$.

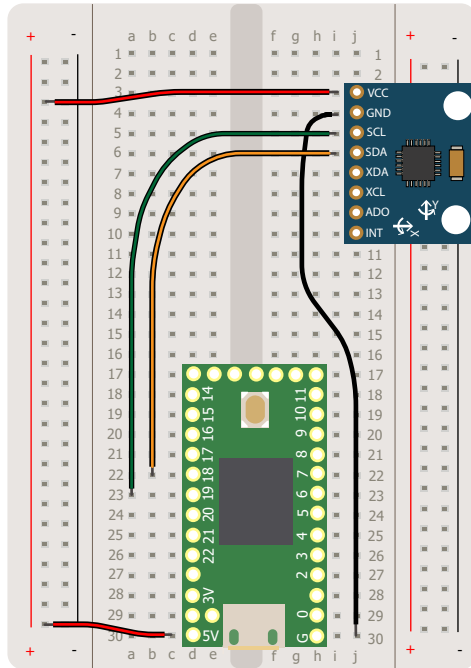
Before you continue, you need to be fully aware of the direction of the roll, pitch and yaw rotational rates. These three rotations are visualized on the figure to the left:

- A roll rotation means that you rotate clockwise around the X axis of the gyroscope.
- A pitch rotation means that you rotate clockwise around the Y-axis of the gyroscope.
- A yaw rotation means that you rotate counter clockwise around the Z axis of the gyroscope.

The respective axis around which you turn, is the only axis that keeps pointing to the same direction during the turn: on the figure, this is each time the red axis.

Mounting instructions of the gyroscope on your quadcopter

Notice that the X and Y axes and their respective rotation directions are also written physically on the MPU-6050 sensor itself. When building the quadcopter and soldering the MPU-6050 to it, always make sure that the axes written on the sensor are aligned with the roll, pitch and yaw axes of the quadcopter itself.



Coding

The code for the I2C protocol is rather complex, so you use a predefined library called `Wire.h`. This was normally installed automatically when you installed Teensyduino, but you can still install it at this point if necessary: in the Arduino IDE, go to sketch → include library → manage libraries and type `Wire` in the search bar. Click on install and you are ready to use it.

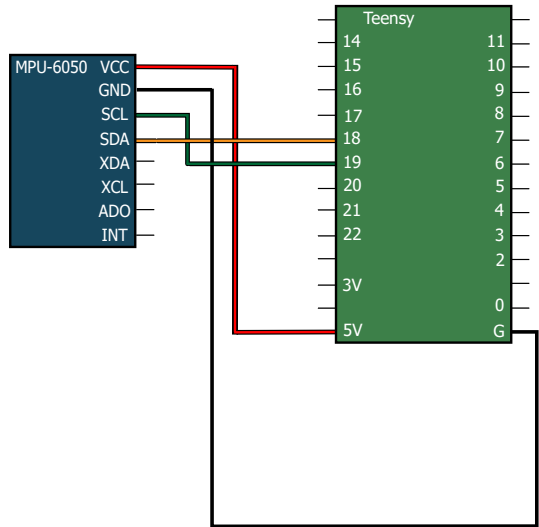
Define the roll, pitch and yaw rates in degrees/s ($^{\circ}/s$) as global variables. You will write the output of the measurements from the sensor to these variables.

Use once again a function to get the data from the gyro. With the I2C protocol, each device (sensor) has its unique address. For our MPU-6050, this address can be found in the register documentation and has a default value of `0x68`. Routing function `Wire.beginTransmission` to this address starts the communication with the sensor.

Sensor documentation

All information about sensors and their setup can easily be accessed online; try and look up the MPU-6050 register map and product specification documentation.

How can you connect the MPU-6050 with your microcontroller? The communication protocol that you will use is I2C. This protocol needs two wires: a serial communication line (=SDA) through which the data can be transferred bit by bit, and a line that carries the clock signal (=SCL). The exact design of this protocol is beyond the scope of your project, but one of its advantages is the transfer of information from multiple sensors using the same SDA and SCL lines to the microcontroller. This will prove useful when you will connect a barometric sensor later on.



The wiring of the MPU-6050 to the Teensy is rather straightforward: connect 5V to Vcc and G to GND to feed the sensor. Subsequently, you connect the serial communication output SDA on the sensor to pin 18 of the Teensy and the clock signal output SCL to pin 19. You are now ready to start programming.

```
1 #include <Wire.h>
```

Include the Wire library

```
2 float RateRoll, RatePitch, RateYaw;
```

Declare the global variables

```
3 void gyro_signals(void) {
4     Wire.beginTransmission(0x68);
```

Start I2C communication with the gyro



Some registers can be accessed to select some predefined options of the MPU-6050. One of these options is a low pass filter, which will be necessary to filter out high frequency vibrations and hence sharp increases and decreases in rotation rates that are caused by running motors. The configuration register, where you can activate this option, has the hexadecimal address 0x1A according to the documentation (this is equal to a decimal address of 26). The options for the low-pass filter correspond to bits 0, 1 and 2 in this address (the Digital Low Pass Filter DLPF setting). You choose a low pass filter with a cut-off frequency of 10 Hz, which corresponds to a value for the DLPF setting of 5. This corresponds in turn to the following binary representation: 00000101. Converting this to a hexadecimal value gives an address of 0x05.

The table from the register documentation that explains the configuration register of the MPU-6050 is displayed on the right. The binary representation for setting the value for the DLPF is given in the third column.

In addition to the low pass filter, you also need to set the sensitivity scale factor of the sensor. The measurements of the MPU-6050 are recorded in LSB (Least Significant Bit). Choose a sensitivity setting of FS_SEL=1 to set the scale factor to 65.5 LSB/(°/s). This means that 1°/s corresponds to 65.5 LSB. You will take into account this scale factor later on in the code. The gyroscope configuration register to activate this option has the hexadecimal address 0x1B (or a decimal address of 27). The FS_SEL setting of 1 corresponds to a 2-bit binary representation of 01. The other settings in the register can be set to zero, giving a binary representation of 00001000. Converting this to a hexadecimal value gives an address equal to 0x08.

The table from the register documentation that explains the gyroscope configuration register of the MPU-6050 is displayed on the right. The binary representation for the setting of the FS_SEL value is given in the third column.

Now you are ready to start importing the measurement values of the gyro. These are located in the registers that hold the gyroscope measurements, which have the hexadecimal numbers 43 to 48. You start writing to address 0x43 to indicate the first register you will use.

Request 6 bytes from the MPU-6050 such that you can pull the information of the 6 registers 43 to 48 from the sensor.

```

5   Wire.write(0x1A);
6   Wire.write(0x05);
7   Wire.endTransmission();

```

Switch on the low-pass filter

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		
		0	0	0	0	0	1	0	1

```

8   Wire.beginTransaction(0x68);
9   Wire.write(0x1B);
10  Wire.write(0x8);
11  Wire.endTransmission();

```

Set the sensitivity scale factor

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]		-	-	-
		0	0	0	0	1	0	0	0

```

12  Wire.beginTransaction(0x68);
13  Wire.write(0x43);
14  Wire.endTransmission();

```

Access registers storing gyro measurements

```

15  Wire.requestFrom(0x68,6);

```



Registers 43 and 44 contain the gyro measurements of the rotational rate around the X axis, in LSB (Least Significant Bit). According to the documentation, they are the result of a unsigned 16-bit measurement. This means you will declare GyroX as an unsigned 16-bit integer `int16_t`. Because the measurement of the rotational rate around the X axis is spread out over two registers with each 8 bits, you will have to merge this information by calling `Wire.read()` twice.

You repeat the same code for registers 45 and 46 (rotational rate around the Y axis) and registers 47 and 48 (rotational rate around the Z axis).

The measurements are expressed in LSB but you want this information in $^{\circ}/s$, not LSB. You have set the LSB sensitivity scale factor of the MPU-6050 equal to 65.5 LSB/ $^{\circ}/s$. Therefore you just divide the values in LSB by 65.6 LSB/ $^{\circ}/s$ to get the measurement values in $^{\circ}/s$. Take care of converting the 16-bit integer values of the measurements in LSB to a floating point representation. As discussed earlier this project, the roll rate corresponds to the rotation around the X axis, the pitch rate to the rotation around the Y axis and the yaw rate to the rotation around the Z axis.

Set the clock speed of the I2C protocol to 400 kHz. This value comes from the product specifications of the MPU-6050 which states that communication with all registers of the device must be performed using I2C at 400 kHz. Use a delay of 250 milliseconds to give the MPU-6050 time to start.

To activate the MPU-6050, write to the power management register, which has the hexadecimal number 6B. All bits in this register have to be set to zero in order for the device to start and continue in power mode. Hence the hexadecimal address becomes 0x00.

Terminate the connection with the gyroscope and end the setup section.

In the loop part of the code, call your function and print the roll, pitch and yaw rates on the serial monitor. Wait 50 milliseconds after each loop to be able to read the values on the serial monitor and close the loop function.


```
16     int16_t GyroX=Wire.read()<<8 | Wire.read();
```

Read the gyro measurements around the X axis

```
17     int16_t GyroY=Wire.read()<<8 | Wire.read();
```

```
18     int16_t GyroZ=Wire.read()<<8 | Wire.read();
```

```
19     RateRoll=(float)GyroX/65.5;
```

```
20     RatePitch=(float)GyroY/65.5;
```

```
21     RateYaw=(float)GyroZ/65.5;
```

```
22 }
```

Convert the measurement units to °/s

```
23 void setup() {
```

```
24     Serial.begin(57600);
```

```
25     pinMode(13, OUTPUT);
```

```
26     digitalWrite(13, HIGH);
```

```
27     Wire.setClock(400000);
```

```
28     Wire.begin();
```

```
29     delay(250);
```

Set the clock speed of I2C

```
30     Wire.beginTransmission(0x68);
```

```
31     Wire.write(0x6B);
```

```
32     Wire.write(0x00);
```

Start the gyro in power mode

```
33     Wire.endTransmission();
```

```
34 }
```

```
35 void loop() {
```

```
36     gyro_signals();
```

```
37     Serial.print("Roll rate [°/s]= ");
```

```
38     Serial.print(RateRoll);
```

```
39     Serial.print(" Pitch Rate [°/s]= ");
```

```
40     Serial.print(RatePitch);
```

Call the predefined function to read the gyro measurements



Testing

Upload the code to your microcontroller and open the serial monitor. You will notice that not all values are equal to zero even though you do not move the MPU-6050:

Roll rate [°/s]= -8.70 Pitch Rate [°/s]= 0.89 Yaw Rate [°/s]= 1.95

Roll rate [°/s]= -8.69 Pitch Rate [°/s]= 0.92 Yaw Rate [°/s]= 1.97

Roll rate [°/s]= -8.66 Pitch Rate [°/s]= 0.87 Yaw Rate [°/s]= 1.94

It is normal when you do not have the same values as mentioned above. You will learn more on how to solve this phenomenon through calibration in the next project.

What are registers?

Registers are places on a microcontroller (the Teensy but also the MPU-6050, since this sensor has a microcontroller as well) that are used as:

- Fast storage locations to store data temporary.
- Locations where you can set predefined options.

You select a register by using its unique address, which is given in the documentation of the microcontroller or sensor. With the I2C arduino library, you use the function `Wire.write(address)` to select the register of choice.

If you select a register to set some predefined option, you once again use the `Wire.write` function. You find the predefined options once again in the documentation. Usually each register has a number of bits: you can set each bit to 0 or 1, which corresponds to different options. Converting the resulting binary representation to a hexadecimal representation gives you the argument for the `Wire.write` function.

If you select a register to read data, use the function `Wire.read()` after selecting the address and reserving the necessary bytes. Let's go back to the example of the low pass filter. Assume you want to set a low pass filter of 20 Hz instead of 10 Hz. You already know that the register address is 0x1A. The documentation of the MPU-6050 says that you need to set the value of `DLPF_CFG` to 4 for the 20 Hz filter. Moreover, `DLPF_CFG` occupies the first three bits of the 0x1A register:

```

41 Serial.print(" Yaw Rate [°/s]= ");
42 Serial.println(RateYaw);
43 delay(50);
44 }

```

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1A	26	-	-	EXT_SYNC_SET[2:0]			DLPF_CFG[2:0]		

The number 4 in a three bit binary representation is equal to 1 0 0: you just multiply each binary number with 2^n , where n is the bit number:

	Bit2	Bit1	Bit0
Binary representation	1	0	0
2^n	$2^2=4$	$2^1=2$	$2^0=1$
Decimal representation	$1 \times 4 + 0 \times 2 + 0 \times 1 = 4$		

If you decide to not set an option for bit3 to bit7 and stick with the default value, the full 8 bit binary and decimal representation becomes:

	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
Binary representation	0	0	0	0	0	1	0	0
2^n	$2^7=128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
Decimal representation	$0 \times 128 + 0 \times 64 + 0 \times 32 + 0 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1 = 4$							

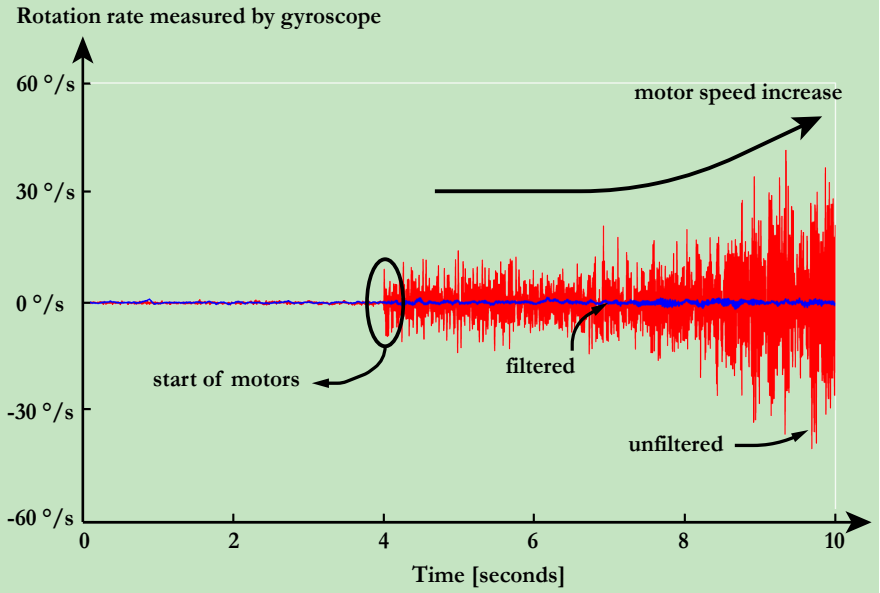
The Wire function use hexadecimal representation: conversion from decimal to hexadecimal is a little bit more complex, but you can use an online converter. 4 in hexadecimal form is equal to 0x04.



A low pass filter?

In line six of the code, you choose the option to send the measurement data through a low-pass filter with a cut-off frequency of 10 Hz. This is a crucial line of code, as the flight controller would not be able to stabilize the drone without it. Why? Your gyroscope is a very sensitive sensor and its readings will be dramatically affected by the vibrations caused by the brushless motors. The sample rate of the gyroscope is equal to 8 kHz, which corresponds to a measurement each $1/8000 = 0.000125$ seconds. The high frequency vibrations from the motors will cause small but very fast accelerations of the quadcopter frame, which are recorded by the gyroscope. The faster the motors spin, the larger the vibrations become; the figure to the right shows the readings of the gyroscope with the motors switched off, switched on and with increasing throttle. During all three cases, the quadcopter stays stationary on the ground so the rotation rate should be equal to $0^\circ/\text{s}$. From the figure you observe that once the motors are started, the unfiltered gyroscope values start to fluctuate a lot. It is clear that it becomes impossible to stabilize your quadcopter with measurement values that vary between 40 and $-40^\circ/\text{s}$ while the quadcopter itself remains stationary and the real rotation rate is equal to $0^\circ/\text{s}$.

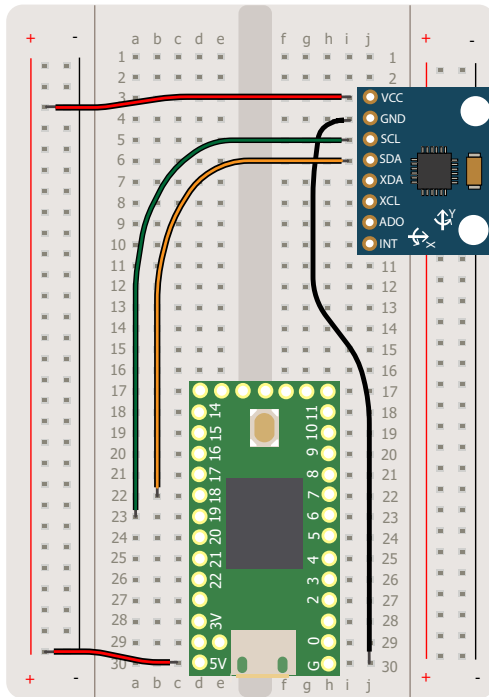
To solve this issue, you use a low pass filter with a cut-off frequency of 10 Hz. The filter attenuates the measurements of the sensor with a frequency of 10 Hz and higher. This means that sensor variations that happen faster than $1/10=0.1$ seconds will only have a limited impact on the final measurement values. The value of 10 Hz is chosen through trial-and-error during testing of the quadcopter; motor vibration frequencies change with each brushless motor and damping of the vibration depends on the whole frame. The blue line on the figure to the right shows the filtered values; they are not affected by the vibrations caused by the brushless motors and are hence suited for your flight controller.





Project 5

Gyroscope calibration



Coding

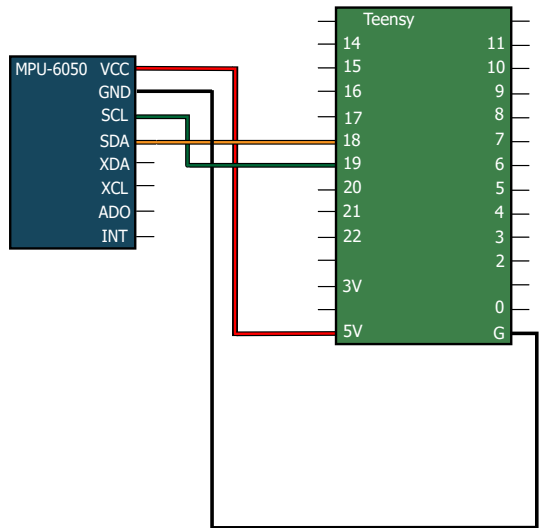
You need four additional variables for calibration: the calibration values for the roll, pitch and yaw rotation rate and a variable to keep track of the number of values you have already recorded to use for the calibration.

Teach your gyroscope the correct rotation rates

With this short project, you will teach your gyroscope the correct rotation rates using a technique called calibration. You will use known rotation rates to correct the values given by your sensor.

At the end of the previous project, you saw that the rotation measurements given by your gyroscope were not correct; even though you did not move the MPU-6050, it still gave non-zero values. You still need to tell the instrument what its physical reference point is. Adjusting the measurements of a sensor such that they correspond with real physical values is called calibration.

In the case of a gyroscope, the easiest reference value that you can use is the rotation rate when the sensor is not moving; this rotation rate should obviously be zero. Because the gyro measurements always tend to fluctuate due to small vibrations in the environment, you take the average of a large number of uncorrected measurement values when the sensor is not moving, calculate their average value and subtract this average value from all future measurement values. You can easily integrate these additional calibration steps in the code of the previous project. The electronic circuit stays the same.



```
1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float RateCalibrationRoll, RateCalibrationPitch,
  RateCalibrationYaw;
4 int RateCalibrationNumber;
5 void gyro_signals(void) {
6     Wire.beginTransmission(0x68);
```

Declare the calibration variables

```

7   Wire.write(0x1A);
8   Wire.write(0x05);
9   Wire.endTransmission();
10  Wire.beginTransmission(0x68);
11  Wire.write(0x1B);
12  Wire.write(0x08);
13  Wire.endTransmission();
14  Wire.beginTransmission(0x68);
15  Wire.write(0x43);
16  Wire.endTransmission();
17  Wire.requestFrom(0x68,6);
18  int16_t GyroX=Wire.read()<<8 | Wire.read();
19  int16_t GyroY=Wire.read()<<8 | Wire.read();
20  int16_t GyroZ=Wire.read()<<8 | Wire.read();

```

In the setup part of the program, create a loop in which you take 2000 measurement values from the gyroscope. Each value is taken 1 millisecond after the other (hence the delay(1)) which means this step takes $2000 \times 1 \text{ ms} = 2 \text{ seconds}$. You add all measured values in the Roll/Pitch/YawRateCalibration variables. During this measurement step, it is important to not move your gyroscope as the goal is to determine the measured values at a rotation rate of zero.

Take the average calibration value by dividing the sum of the 2000 measurement values by 2000. Now you have the measurement values at which the rotation rates are zero.

Once the setup is finished and you have determined the calibration values, subtract them from the measured values in order to get the correct physical values. Print the corrected values to the serial monitor.


```

21     RateRoll=(float)GyroX/65.5;
22     RatePitch=(float)GyroY/65.5;
23     RateYaw=(float)GyroZ/65.5;
24 }
25 void setup() {
26     Serial.begin(57600);
27     pinMode(13, OUTPUT);
28     digitalWrite(13, HIGH);
29     Wire.setClock(400000);
30     Wire.begin();
31     delay(250);
32     Wire.beginTransmission(0x68);
33     Wire.write(0x6B);
34     Wire.write(0x00);
35     Wire.endTransmission();
36     for (RateCalibrationNumber=0;
37         RateCalibrationNumber<2000;
38         RateCalibrationNumber++) {
39         gyro_signals();
40         RateCalibrationRoll+=RateRoll;
41         RateCalibrationPitch+=RatePitch;
42         RateCalibrationYaw+=RateYaw;
43         delay(1);
44     }
45     RateCalibrationRoll/=2000;
46     RateCalibrationPitch/=2000;
47     RateCalibrationYaw/=2000;
48 }
49 void loop() {
50     gyro_signals();
51     RateRoll-=RateCalibrationRoll;
52     RatePitch-=RateCalibrationPitch;
53     RateYaw-=RateCalibrationYaw;
54     Serial.print("Roll rate [°/s]= ");
55     Serial.print(RateRoll);
56     Serial.print(" Pitch Rate [°/s]= ");
57     Serial.print(RatePitch);
58     Serial.print(" Yaw Rate [°/s]= ");
59     Serial.println(RateYaw);
60     delay(50);
61 }

```

Perform the calibration measurements

Calculate the calibration values

Correct the measured values



Testing

When you run the code and open the serial monitor, the roll, pitch and yaw rates should be almost zero when you do not move the gyroscope. Remember that during the setup phase, when no values are yet displayed on the serial monitor, you should not move the gyroscope in order to ensure a correct calibration.

- Roll rate [°/s]= 0.09 Pitch Rate [°/s]= -0.10 Yaw Rate [°/s]= -0.03
- Roll rate [°/s]= 0.09 Pitch Rate [°/s]= -0.09 Yaw Rate [°/s]= -0.03
- Roll rate [°/s]= 0.04 Pitch Rate [°/s]= -0.04 Yaw Rate [°/s]= -0.03

Now try to experiment by moving the gyroscope in the directions displayed in the figure to the right. When you for example pitch around the Y axis from 0 to 45°, wait and go back to 0°, the pitch rate should first increase with a value proportional on how fast you rotate, subsequently fall to around 0°/s and then go negative with a value proportional on how fast you rotate back to 0°.

Pitch from 0 to 45° and hold at 45°:

- Roll rate [°/s]= 0.01 Pitch Rate [°/s]= 0.02 Yaw Rate [°/s]= 0.00
- Roll rate [°/s]= -0.06 Pitch Rate [°/s]= 185.71 Yaw Rate [°/s]= -0.10
- Roll rate [°/s]= -0.05 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.02

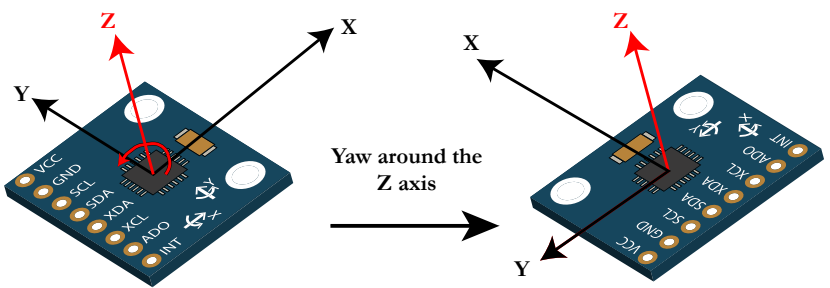
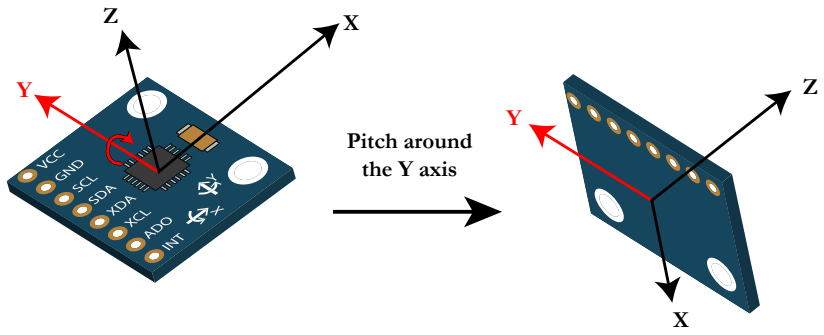
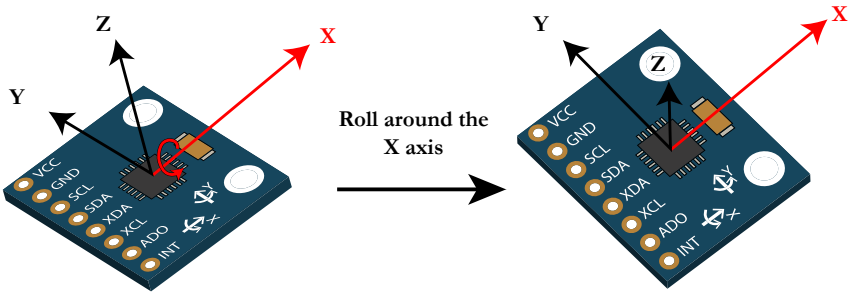
Pitch from 45° back to 0° and hold at 0°:

- Roll rate [°/s]= -0.05 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.02
- Roll rate [°/s]= 0.65 Pitch Rate [°/s]= -177.02 Yaw Rate [°/s]= 0.39
- Roll rate [°/s]= -0.03 Pitch Rate [°/s]= 0.06 Yaw Rate [°/s]= 0.00

Try the same test for the other directions to verify that your code is working properly.

Time to fly?

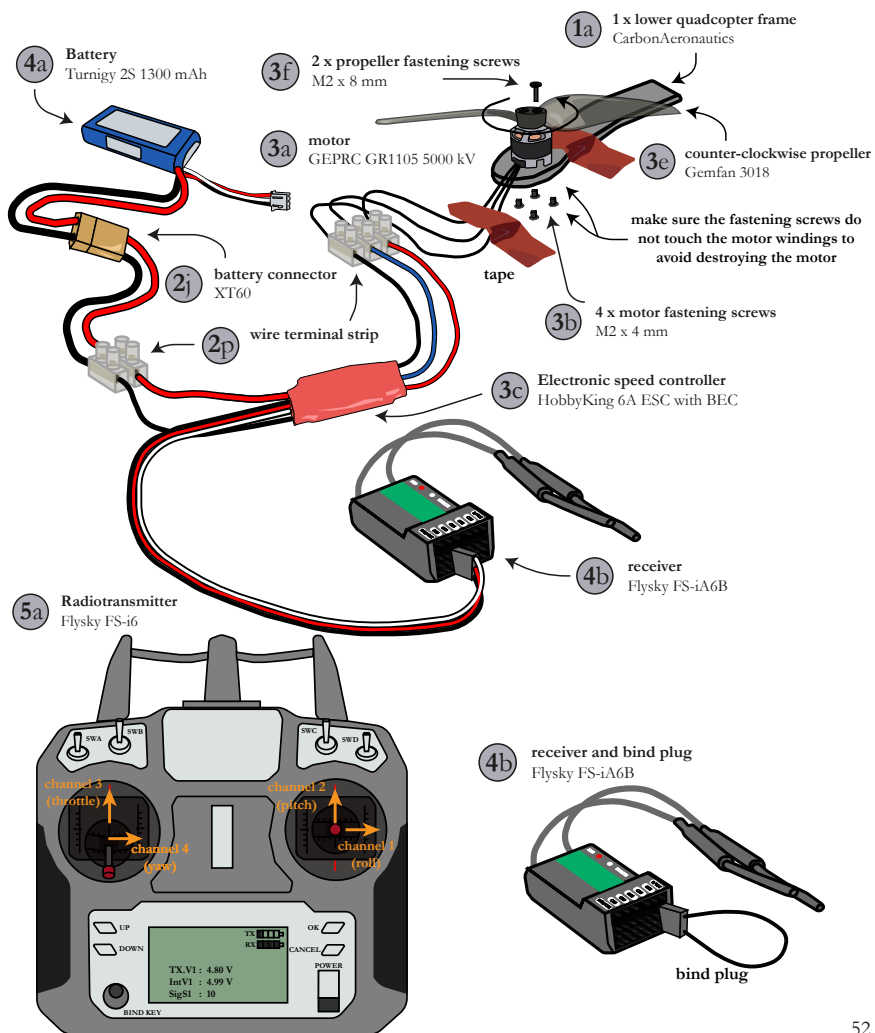
The calibration needs to be performed during each start-up procedure, because the gyroscope measurement values tend to drift over time. You cannot start the motors yet during calibration, because their vibrations will impact the quality of the calibration. This means that the setup procedure takes some seconds before you can actually start the motors and begin your flight. That is why some projects ago, you learned to signal the status of your quadcopter with the red and green LEDs. Be mindful also to not move your quadcopter during this startup phase, as this will affect the calibration quality and hence the smoothness of your subsequent flight.





Project 6

Take your motors for a spin



Test your radio, motors and ESCs

You will now test your radiotransmitter, receiver and motors. Use this opportunity to calibrate each ESC and verify that all motors spins in the correct direction.

Each radio, ESC (Electronic Speed Controller) and motor combination needs to be tested, calibrated and verified before you start soldering all parts together. Let's start with the first motor:

a. Set up your test stand

1. Slide a propeller down the motor shaft and push it firmly on the top of the motor. For motor 1, you need a counter-clockwise propeller: the leading edge of this propeller must be the first to turn counter-clockwise as shown on the picture to the left.
2. Fasten the propeller with two long M2 screws but be sure that the screw does not touch the inner motor windings.
3. Attach the motor to the drone frame with four short M2 screws.
4. Attach the drone frame firmly to your desk using tape.
5. Connect the three black motor wires using a wire terminal strip with the black, blue and red cables coming out of the ESC. It does not matter yet which motor wire is connected with which ESC wire.
6. Connect the red and black cable of the XT60 plug using another wire strip with the red and black power wires of the ESC. Do not connect the battery yet.
7. Connect the white, red and black cables with channel 3 of the receiver as visualised on the picture.

b. Bind the receiver to the radiotransmitter through the bind plug

1. Make sure the throttle stick on the radiotransmitter is in the lowest possible downward position.
2. Turn on the POWER button of your radiotransmitter while simultaneously holding the BIND KEY button. The text *RX Binding...* should be displayed.
3. Connect the bind plug with the B/VCC pins on the receiver as shown on the picture. Keep the connection between the receiver and ESC in place.
4. Connect your battery using the XT60 plug. The red LED on the receiver should illuminate and your radiotransmitter should beep one time, indicating that binding is successful. The *text SigS1* on the radiotransmitter confirms binding of the receiver. Disconnect the battery again.
5. Remove the bind plug. When connecting the battery again, your radiotransmitter should automatically connect to your receiver.

c. Test your motor and verify its turning direction

1. Turn on the radio transmitter through the POWER button.
2. Connect your battery using the XT60 plug. You should hear one beep from your radio transmitter indicating that it is connected with the receiver, and subsequently **two** beeps from the motor to indicate that you are connected to a **2S** battery followed by two more beeps from the motor indicating that the ESC preparation is completed.
3. Now slowly increase the throttle stick on your radio transmitter to turn on the motor and spin it at increasing speeds.
4. Verify that the propellers are spinning in the correct direction (motors 1+3: counter-clockwise, motors 2+4: clockwise). If they do not spin in the correct direction, remove the battery and switch two of the ESC wires going to the motor with each other to reverse the spinning direction. Note: if you connect the black, blue and red ESC wires in the correct order with the black motor wires as shown in the figure, the spinning direction will always be correct.

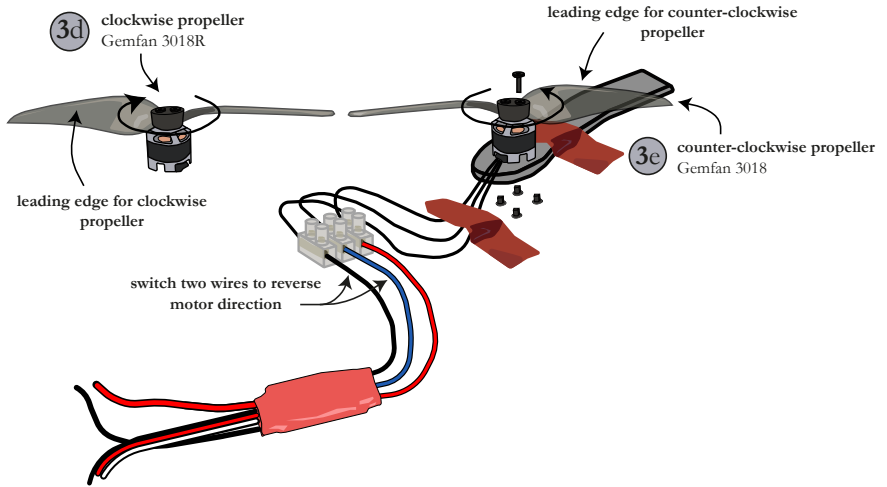
d. Calibrate the ESC

ESC calibration always necessary to be able to control your quadcopter. Through calibration, you tell the ESC what the upper and lower positions of the radio transmitter sticks are. Carry out the calibration with the help of the following steps:

1. Make sure your battery is not plugged in and the radio transmitter is turned off.
2. Turn on the radio transmitter and put the throttle stick to its uppermost position. When connecting the battery, this will make sure the ESC goes in programming mode.
3. Do **not connect the battery yet but first read these instructions**: after connecting the battery, you will first hear one beep from the radio transmitter, next you hear subsequent beeps from the motor. You will need to move the throttle stick to its lowest position **between the first and the fourth beep of the motor!** If you are too late, do not attempt anymore to lower the throttle stick but just disconnect the battery and try again starting from step 1.
4. When you understand the instructions from step 3, connect the battery, wait until the beep from the radio transmitter has passed and lower the throttle stick to its lowest position between the first and the fourth beep of the motor.
5. After two seconds, the motor should give once again two times two beeps indicating that the calibration is finished. Congratulations, you finished your motor setup! Go once again to step c to test the throttle response.

c. Testing motors 2 to 4

Steps a, c and d need to be carried out for the other motors and ESC too. **Try to reverse the spinning directions of some motors by switching the ESC wires and changing the propellers.**



ESC programming

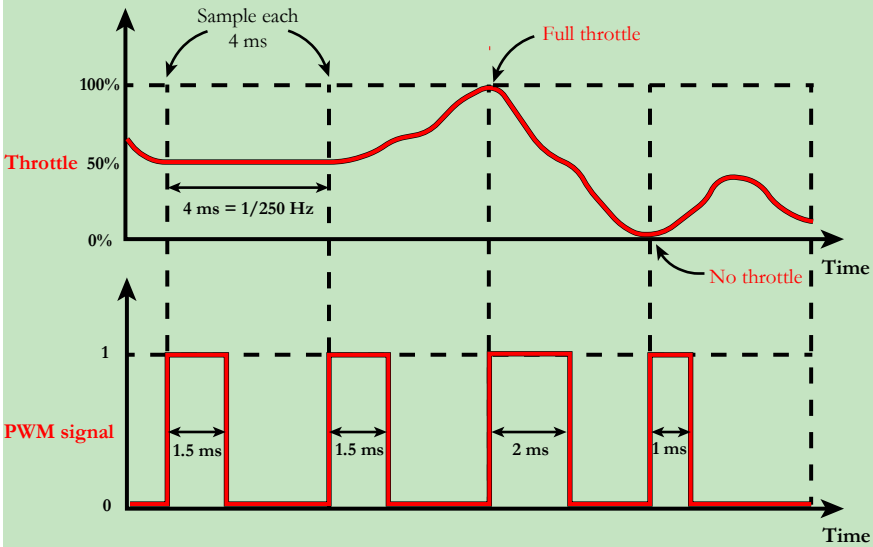
Not only ESC calibration can be carried out by putting the ESC into programming mode, but other settings can be adjusted as well. The first four beeps are followed by four beeps with a slightly different noise, indicating a different setting. By moving your throttle stick down during these beeps, you can choose to activate the corresponding setting. Through this method, you can choose from multiple settings; more information can be found in the datasheet of your ESC.



ESC control through PWM

You control the speed of your motor through the ESC, which in turn gets its commands from channel 3 (the throttle channel) of your receiver. The receiver sends Pulse Width Modulated (PWM) signals to the ESC to this channel, indicating a desired throttle value between 0 and 100%. But what is a PWM signal, really?

In essence, a PWM signal is just a signal that varies between a HIGH voltage (for example 5V) and a LOW voltage (for example 0V), or between 1 and 0 to keep the concept simple. It is the time length during which the signal stays HIGH that is used to transfer information. Suppose for example that a time length of 1 ms HIGH corresponds with no throttle (0%) and a time length of 2 ms HIGH corresponds with full throttle (100%). The PWM frequency of most receivers and ESCs is 4 ms (or $1/0.004=250$ Hz), meaning that your chosen signal repeats itself every 4 ms with a length that corresponds with your throttle command. This concept is illustrated in the figure below and will be used later on to send commands to your ESCs using your Teensy instead of directly from the receiver.

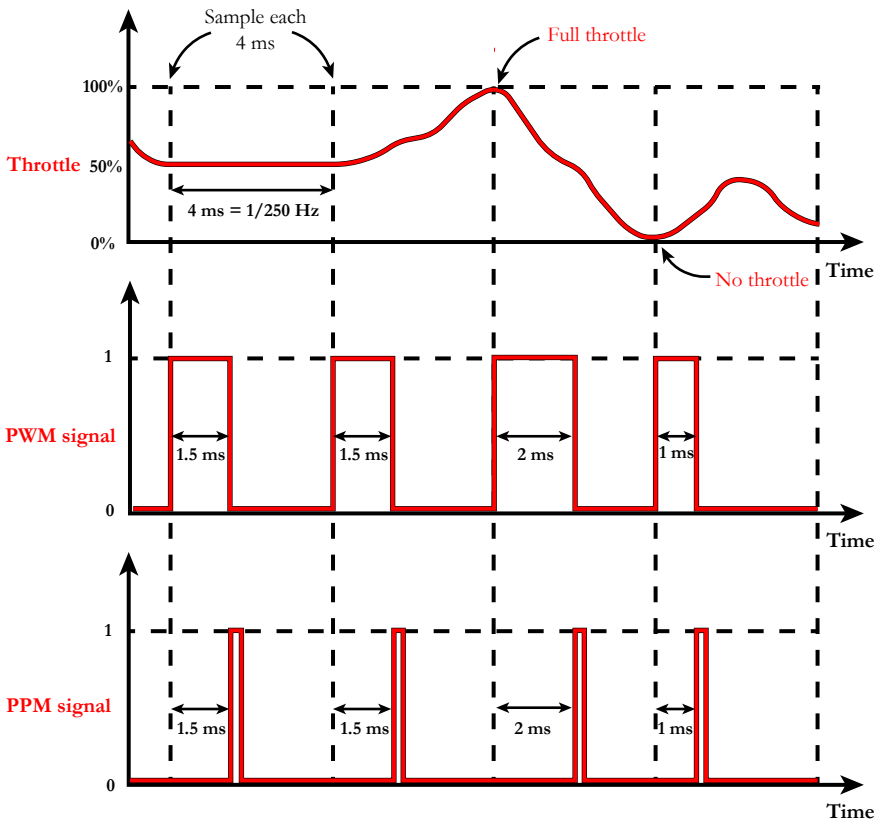






Project 7

Receiving commands



Process commands sent to the receiver

The commands that you give through your radio controller are transmitted via radio waves and picked up by your receiver. The receiver then converts the radio waves to a signal which can be read by your microcontroller. You will now learn how to convert these signals from the receiver to variables that can be used further in the flight code.

There are multiple methods to transfer information through digital signals, one of which you already learned: Pulse Width Modulation (PWM). PWM is an easy method to send information of one radiochannel from the receiver to the microcontroller. However, receiving information from multiple radiochannels would require one signal cable to the microcontroller for each channel. This is cumbersome when you need a lot of channels, meaning you will have to master another technique: Pulse Position Modulation (PPM).

You already saw that you can use the width of a PWM signal to transfer information: a signal width of 1 ms (=1000 μ s) corresponds for example with no throttle (0%) and a width of 2 ms (=2000 μ s) corresponds with full throttle (100%). Using the technique of Pulse Position Modulation, you are able to transfer the same information using the position of the signal in time, instead of the width. With PPM, the width of each signal stays the same but its position changes each time depending on the value of the radiochannel.

An example is displayed to the left: the first throttle value sampled from the radiochannel is equal to 50%. Using PWM, this corresponds to a signal width of 1.5 ms (=1500 μ s). With PPM, the signal starts after 1.5 ms and has the same width each time. When 4 ms have passed, another throttle value is sampled and the cycle begins again.

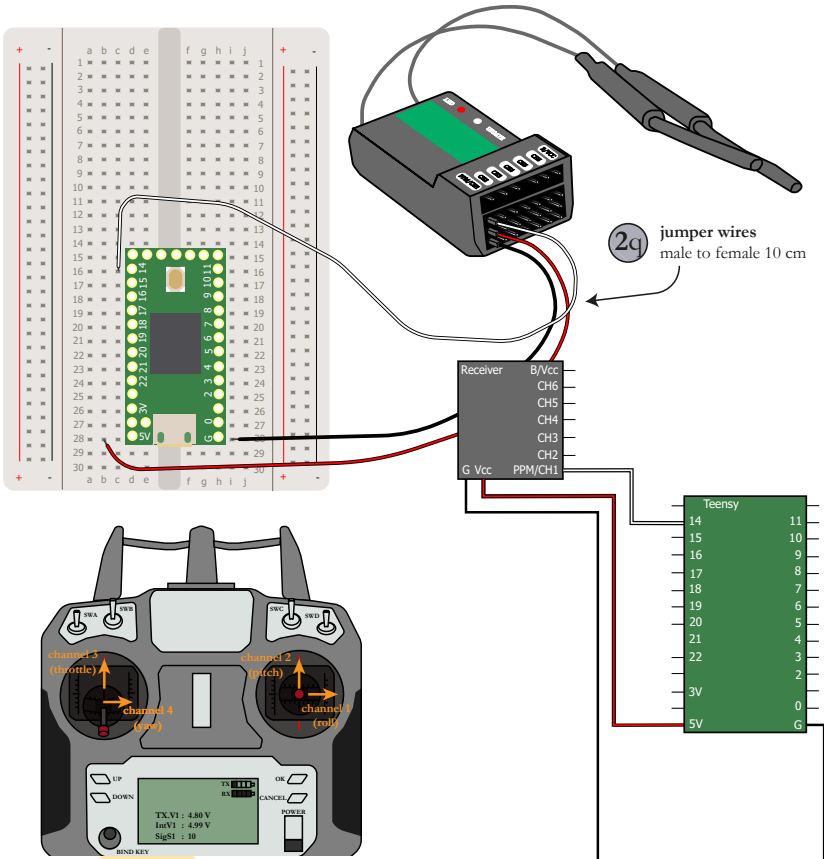
Setting up your radiotransmitter in PPM mode

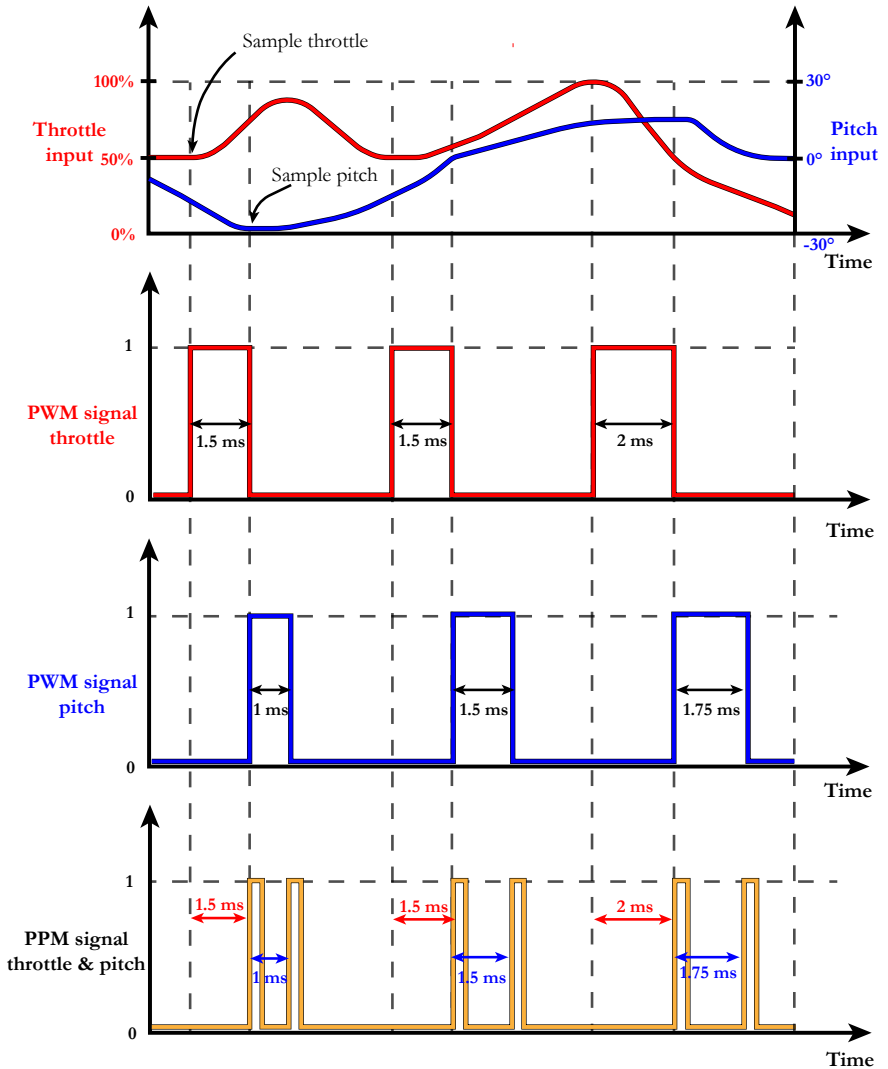
The standard operating mode of your transmitter is PWM, not PPM! This means you have to configure the transmitter to use PPM for this project with the following steps:

Switch on the transmitter → hold the OK button for two seconds → choose System → go down and choose "RX setup" → go down and choose "Output mode" → choose "PPM" and hold the CANCEL button for two seconds to save your choice.

Now what about receiving information from multiple radiochannels? Consider a two-channel example: you want to receive information about the throttle and the pitch input. These inputs are independent of each other and require two separate signal cables if you would use PWM. When receiving the two PWM signals in the microcontroller, you will need two different `analogRead()` commands. Because they cannot be processed simultaneously in the microcontroller, either the throttle or the pitch values will be sampled (=read) first, after which the other signal follows directly. Besides the need for two signal cables, this also requires more calculation time from the microcontroller.

You are now ready to take advantage of the interesting property of PPM: because only the position of the signals changes and not their width, sampling both the throttle and pitch signals directly after each other allows you to keep track of their original values by measuring the time from each rising signal value. This means that with one signal cable, information from multiple signals can be ‘transported’.





Using this knowledge, you can now easily build the connection from the receiver to the Teensy. Channel 1 of the receiver has also "PPM" written on it, which means that you can use these connectors to send PPM signals to your microcontroller. Use a cable to connect the second connector of the receiver starting from above to pin 14 of the Teensy as displayed on the figure to the left. Connect the third and fourth connector of the receiver to 5V and ground on the Teensy respectively. This will ensure that the receiver is fed from the microprocessor. You are now ready to start coding.



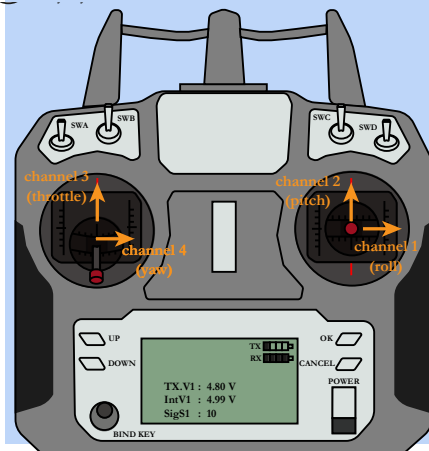
Coding

The code for handling Pulse Position Modulation is rather complex, so you again use a predefined library called `PulsePosition.h`. This library is normally already installed when you installed Teensyduino, but you can still install it at this point with the "manage libraries tool" like you already did with `Wire.h`. Next you create a PPM input object, which is in this case the receiver input. You track each pulse starting from their rising edge.

Use two global variables for this project: one array which can store up to eight channel values (initialized as eight zeroes) and one integer that stores the number of channels transmitter by the receiver.

To be able to read the receiver data multiple times in the code, you create a function. This function will first check how many channels are available by writing `.available` behind the PPM input object; if there are channels available, it reads the value of each channel and stores it in the array of receiver values.

Read the values sent from the receiver by calling the function defined in line 5. Print the available number of channels followed by the values for each channel.



Channels 1, 2, 3 and 4 correspond respectively with the roll, pitch, yaw and throttle inputs. Because the array numbering in the Arduino IDE starts with 0 instead of 1, `ReceiverValue[0]` actually corresponds with channel 1 or the value for roll.

Finally, you have to use a delay of 50 milliseconds to be able to read the values that will be displayed on the serial monitor.

```
1 #include <PulsePosition.h>
2 PulsePositionInput ReceiverInput(RISING);
```

Use the PulsePosition library

```
3 float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
4 int ChannelNumber=0;
```

Declare the variables to store channel info

```
5 void read_receiver(void) {
6     ChannelNumber = ReceiverInput.available();
7     if (ChannelNumber > 0) {
8         for (int i=1; i<=ChannelNumber;i++){
9             ReceiverValue[i-1]=ReceiverInput.read(i);
10            }
11    }
12 }
```

Define a function to read the receiver data

```
13 void setup() {
14     Serial.begin(57600);
15     pinMode(13, OUTPUT);
16     digitalWrite(13, HIGH);
17     ReceiverInput.begin(14);
18 }
```

```
19 void loop() {
20     read_receiver();
21     Serial.print("Number of channels: ");
22     Serial.print(ChannelNumber);
23     Serial.print(" Roll [μs]: ");
24     Serial.print(ReceiverValue[0]);
25     Serial.print(" Pitch [μs]: ");
26     Serial.print(ReceiverValue[1]);
27     Serial.print(" Throttle [μs]: ");
28     Serial.print(ReceiverValue[2]);
29     Serial.print(" Yaw [μs]: ");
30     Serial.println(ReceiverValue[3]);
31     delay(50);
32 }
```

Read and display the PPM information on the serial monitor



Testing

Now you are ready to read the data send from the transmitter:

- Connect the Teensy to your computer through the USB cable;
- Upload to code to your Teensy and open the serial monitor.

If the radio transmitter is not yet switched on, the LED on the receiver should blink. The values displayed on the serial monitor will all be zero, except for the number of channels, which should be equal to -1. This is the default value when no channels are discovered.

When you switch on your radio transmitter, the LED on the receiver should stop blinking and the transmitter commands should be displayed on the serial monitor in microseconds [μs]. Since the roll, pitch and yaw sticks are always centred, the values you record will be around 1500 μs for the corresponding channels.

Now change the positions of the roll, pitch, throttle and yaw sticks and verify that they vary between 1000 and 2000 μs . You have now successfully made a radio-connection between your transmitter, the receiver and microcontroller!

Switching your radiotransmitter back to PWM mode

For the next projects, the receiver needs to stay in PPM mode. However, if you want to redo the previous project in which you controlled one motor and one ESC through PWM without a microcontroller in between, do not forget to switch your radiotransmitter back to its default PWM setting. This can be carried out using the same procedure as switching to PPM mode, which you learned at the start of this project.

11:26:52.994 -> Number of channels: -1 Roll [μ s]: 0.00 Pitch [μ s]: 0.00 ...

11:26:53.041 -> Number of channels: -1 Roll [μ s]: 0.00 Pitch [μ s]: 0.00 ...

11:26:53.089 -> Number of channels: 8 Roll [μ s]: 1499.97 Pitch [μ s]: 1498.99 ...

11:26:53.135 -> Number of channels: 8 Roll [μ s]: 1498.96 Pitch [μ s]: 1500.00 ...

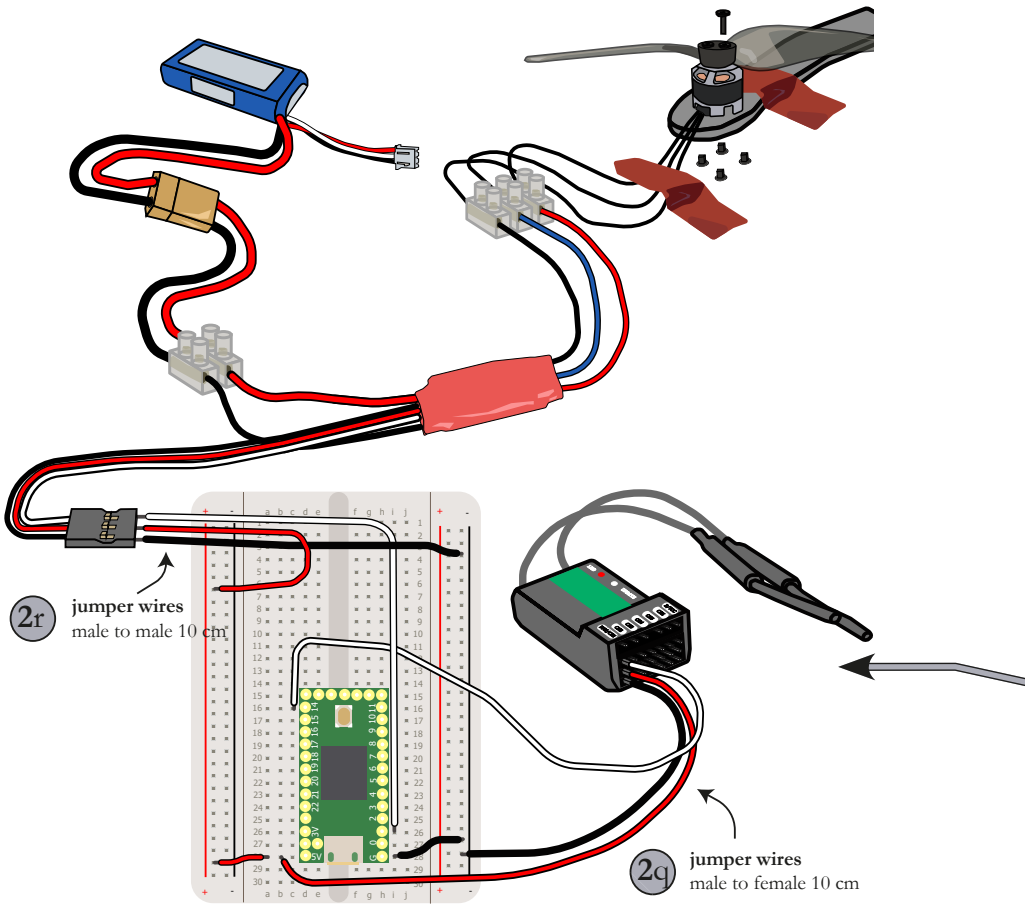
11:26:53.181 -> Number of channels: 8 Roll [μ s]: 1498.96 Pitch [μ s]: 1496.99 ...





Project 8

Controlling your motors



Use PPM to control motor speed

As the name implies, an electronic speed controller is able to control the speed of your brushless motor. In this project, you will learn how to send commands to this controller and thus the motor through your microcontroller.

The electronic speed controller (ESC) is like your Teensy a microcontroller, but it has only one purpose: adapting the voltage going to the motor in such a way that the motor changes its speed. The speed at which a brushless motor turns depends on the supplied voltage, according to the kV constant. A motor rated at 4000 kV turns at 4000 rpm/V. A supplied voltage of 3V gives a turning rate of $4000 \text{ rpm/V} \times 3\text{V} = 12\,000 \text{ rpm}$. If you would directly connect your battery to the brushless motor, it would turn at a constant speed, as the voltage supplied by the battery does not change.

So how does the ESC adapt the voltage supplied to the brushless motor? By closing and opening the connection between the battery and the brushless motor, you can change the average voltage supplied to the motor. The longer the connection stays closed, the higher the supplied average voltage will be (with the maximum being the voltage supplied by the battery if the connection remains closed the whole time). You can control this average voltage through pulse-width modulation. This means that you need to send a PWM signal from the Teensy to the electronic speed controller in order to control the brushless motor.

You will now control the first brushless motor through the physical circuit displayed on the left;

- Use some tape to make sure that the motors do not lift off.
- Re-use the same circuit with whom you already connected the receiver when testing the motors.
- Connect the three wires coming out of the other side of the ESC to the wires of the brushless motors using the wire terminal strip.
- Connect the power output of your ESC with the XT60 cables using another wire terminal strip. Be careful not to switch the polarity of the cables!
- Connect the white signal cable from the ESC to pin 1 of the Teensy.
- Connect the 5V and 0V of the ESC to 5V and GND of the Teensy respectively. Besides the power circuit going to the motor, your ESC has a second circuit that stays at 5V all the time and which enables you to power both the motors and your electronics with the same battery.

Only connect the battery once you uploaded the code to the Teensy and disconnected your USB cable from your computer (as a precautionary measure if something is wrong with the wiring).

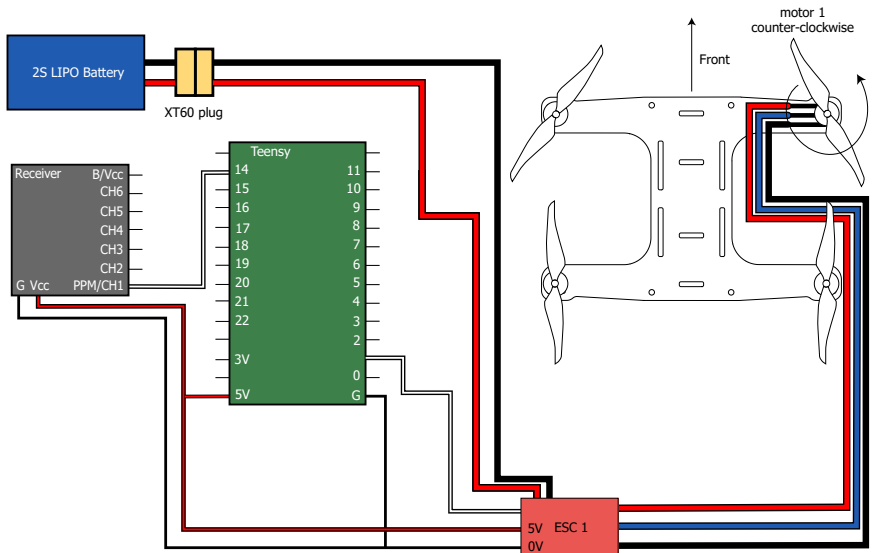
Make sure you use motor 1 (turning counter-clockwise) for this project. If you notice during testing that it spins clockwise instead of counter-clockwise, just switch two wires coming out of the motor with each other using the wire terminal strips.

In the code, you will control the brushless motor using a PWM signal sent from the Teensy to the ESC but you will use a PPM signal coming from the receiver to the Teensy. The throttle (so channel 3 or `ReceiverValue[2]` in Arduino language) will be used to set the speed of the brushless motor. The PWM values that you send to the ESC are the same as the PWM values that you get from the receiver: 1000 μs gives no throttle (motor does not turn), while 2000 μs gives full throttle.

Coding

Define the value for the throttle as a floating point number. The future value for this variable lies between 1000 and 2000 μs .

You will send the PWM signals from pin 1 of your Teensy to motor 1. Configuring pin 1 to send PWM signals can be done with the function `analogWriteFrequency`(pin number, PWM frequency). The PWM frequency used in most ESCs is equal to 250 Hz ($=1/250=0.004\text{ s}=4000\mu\text{s}$). This value can be found in the manual of your ESC.



```

1 #include <PulsePosition.h>
2 PulsePositionInput ReceiverInput(RISING);
3 float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
4 int ChannelNumber=0;

5 float InputThrottle;

6 void read_receiver(void) {
7     ChannelNumber = ReceiverInput.available();
8     if (ChannelNumber > 0) {
9         for (int i=1; i<=ChannelNumber;i++){
10             ReceiverValue[i-1]=ReceiverInput.read(i);
11         }
12     }
13 }

14 void setup() {
15     Serial.begin(57600);
16     pinMode(13, OUTPUT);
17     digitalWrite(13, HIGH);
18     ReceiverInput.begin(14);

```

Define the throttle variable

```

19     analogWriteFrequency(1, 250);

```

Send PWM signals from your Teensy



By default, the resolution of PWM signals sent by the Teensy is 8 bit. This means that the signal ranges between 0 and $2^8-1=255$. For our application, this would give a too coarse control, so you will change from an 8 bit to a 12 bit resolution; the PWM signal going from the Teensy to the ESC ranges between 0 and $2^{12}-1=4095$. For a frequency of $250\text{ Hz}=4000\text{ }\mu\text{s}$, 0 then corresponds with $0\text{ }\mu\text{s}$ and 4095 corresponds with $4000\text{ }\mu\text{s}$. When you want to send a PWM command in μs to the ESC, do not forget to multiply that value with $4095/4000=1.024$.

SAFETY RELATED LINES: in order to avoid a motor start when you did not yet touch the radiotransmitter (for example when the throttle stick was not in the lowest position after your last flight), you add some additional lines before finishing the setup process. If the values sent from channel 3 (`ReceiverValue[2]` = the throttle channel) are bigger than $1050\text{ }\mu\text{s}$ or lower than $1020\text{ }\mu\text{s}$, the code will not continue. When you move the throttle stick around its lowest value range (between 0.2 and 0.5%), the code continues and you can start the motor.

You already know that the throttle corresponds to channel 3 or `ReceiverValue[2]` from the radiotransmitter. The value of the throttle ranges from 1000 (no throttle) to $2000\text{ }\mu\text{s}$ (full throttle). You send this value to pin 1 and subsequently also the ESC and motor 1 through the `analogWrite` function. Remember to convert the throttle values in μs to their 12 bit equivalent by multiplying them with 1.024.

Testing

Once you uploaded the code successfully, disconnect the USB from the Teensy and connect the battery. Normally the LEDs on both the Teensy and the receiver should be illuminated as they receive power from the battery. Next, switch on your radiotransmitter (remember it should still be in PPM mode, not PWM); you should hear two times two beeps from your motor. Now slowly increase the throttle on your radiotransmitter to start the motor. Verify that the motor spins in the correct (counter-clockwise) direction.

Troubleshooting when the motor does not start:

- Verify that your receiver and Teensy get power by looking at their LEDs.
- Verify that your radiotransmitter is connected with your receiver (the LED on the receiver should not blink but should stay illuminated).
- Verify that the setting of your radiotransmitter is correct (in PPM mode).
- Verify the connections to the motor.

```
20 analogWriteResolution(12);
21 delay(250);
```

Set the PWM frequency

```
22 while (ReceiverValue[2] < 1020 ||
23         ReceiverValue[2] > 1050) {
24         read_receiver();
25         delay(4);
26 }
```

Avoid uncontrolled motor start

```
27 void loop() {
28     read_receiver();
29     InputThrottle=ReceiverValue[2];
30     analogWrite(1,1.024*InputThrottle);
31 }
```

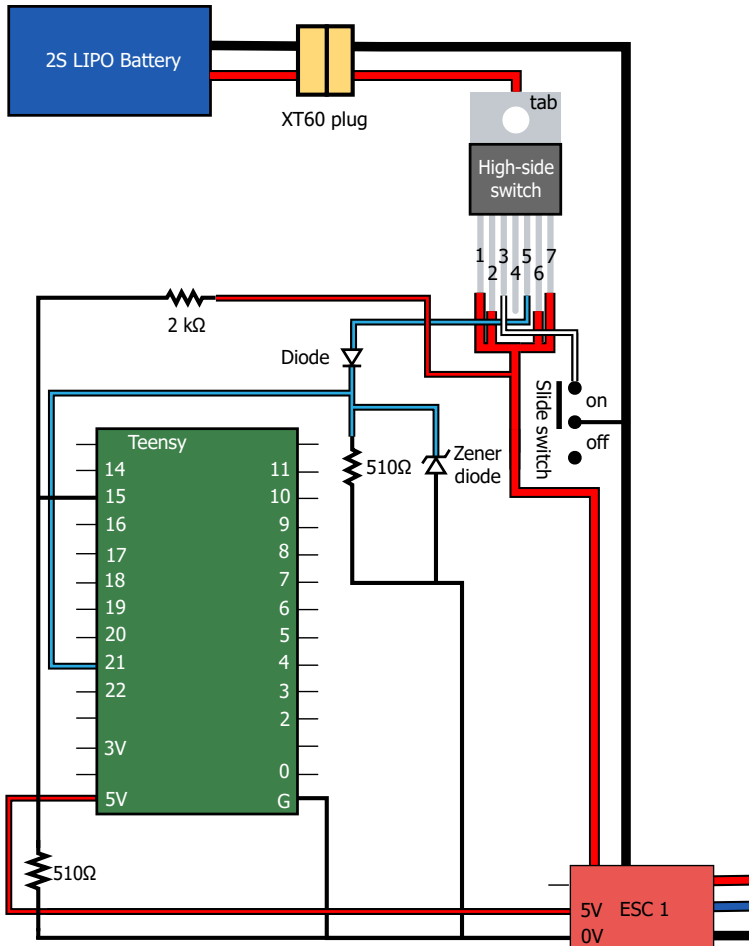
Send the throttle input to the motor.





Project 9

Battery management



Monitor and protect your battery

Batteries store energy - a lot of energy. When this amount of energy is accidentally released in a short period of time, due to for example a short circuit, it can cause significant damage. Monitoring and closely protecting the battery is therefore very important.

A first obvious feature for each battery management system is a **switch**: you want to be able to turn off the power from the battery towards your motors at all time, even during full throttle. A standard (breadboard) slide switch can only switch off currents less than 1 A at 12 V; these type of switches are insufficient for our application: at full throttle, the current drawn by all four motors can easily surpass 20 A.

Additionally, you want some form of **battery protection** as well. When the battery is accidentally short-circuited, all the energy in the battery will be released nearly instantaneous. This causes the point with the highest resistance in the short-circuit to heat up and start burning; this can be a cable, a trace on the printed circuit board or the battery itself. The burning of the cable or the trace will cause the short circuit to disappear because the conductive band is broken, at the cost of destroying the printed circuit board. But when the battery itself has the highest resistance in the short circuit, it can burn and explode causing further damage.

A third feature is some form of **current measurement** to be able to monitor the battery level more closely - more about this will be explained further in this project. For your quadcopter application, you use a special transistor that integrates all of the above features: a high side power switch. This type of transistor is placed between the positive side of the battery and the load (hence the name high side). An Infineon BTS50055-1TMB or BTS50080-1TMB transistor will be used for this application because these switches are capable of conducting rather high load currents of respectively 70 and 37.5 A.

To be able to conduct these high loads, you need to use the tab of the high side switch, then connect output pins 1, 2, 6 and 7 with each other before making the connection with the load (which will be the ESCs in this case). You will connect pin 3 with a slide switch that is in turn connected with the high-current GND line; the transistor is switched on when a current flows between pin 3 and the GND. When the transistor is switched off, the voltage between pin 3 and the GND almost equals the battery voltage (maximally 8.4V for a 2S battery). The transistor has the capability to switch off the load current at full throttle, so you can instantaneously switch off your quadcopter at full power using the slide switch.

A downside of this high load current is the very high short circuit current limit: up to 180 A. This means that a short circuit in your printed circuit board can nonetheless cause serious damage!

Monitoring the current

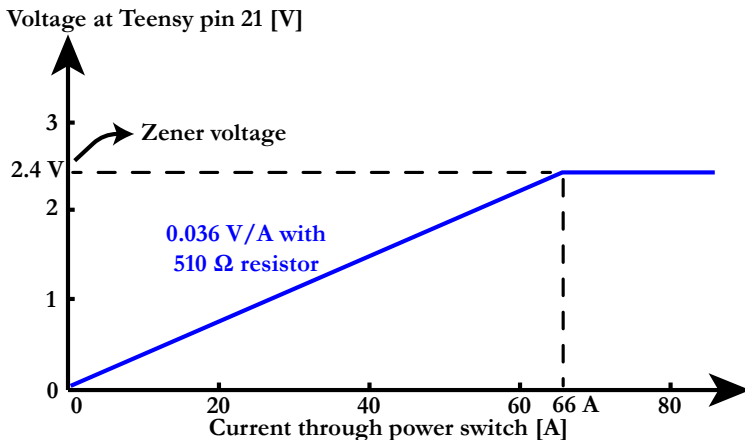
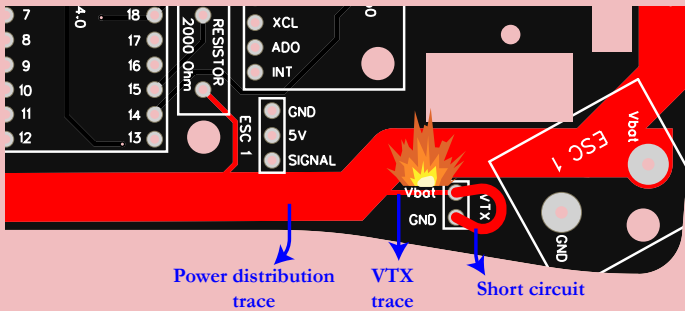
You already learned that measuring the voltage of your battery enables you to predict its remaining lifetime. Whenever you put the battery under heavy load, which will be the case when you throttle up the motors, the voltage will drop significantly without a real decrease in battery level, reducing your capability to accurately monitor the remaining battery level during flight. The only way to overcome this problem is to directly measure the current drawn from the battery.

A special feature of your Infineon transistor is its current sensing capability: a current that is smaller but proportional to the load current flows from output pin 5 of the transistor. According to the datasheet of the transistor, the current sense ratio is typically equal to 14 000. This means that a load current of 14 A through the transistor corresponds with a current of 1 mA from pin 5. With your Teensy, you cannot measure a current, only voltages. That is why you use a resistor to convert the current to a measurable voltage. Using Ohm's law, you know that a current of 1 mA through a resistor of $510\ \Omega$ will cause a voltage drop over the resistor of $1\ \text{mA} \times 510\ \Omega = 510\ \text{mV}$ or 0.51 V. In other words, a current of 14 A through the transistor corresponds to a voltage drop over the resistor of 0.1 V. Assuming this remains proportional over the full measuring range gives a voltage to current ratio of $0.51\ \text{V}/14\ \text{A}$ or $0.036\ \text{V/A}$.

You will connect the $510\ \Omega$ resistor to pin 21 of your Teensy for voltage measurement. Before wiring everything up, you also need to consider the effect of a short circuit in the high current part of your current sensing circuit. In the event of a short circuit, the load current can easily exceed 180 A for a couple of milliseconds before the transistor switches off. Such a high current will lead to a high voltage over our resistor: $180\ \text{A} \times 0.036\ \text{V/A} = 6.5\ \text{V}$. Knowing that the input pins of our Teensy are only 3.3V-tolerant, a voltage this high will probably destroy the microcontroller. To solve this issue, you will use a Zener diode. A Zener is a special diode that does not conduct any current below a certain fixed voltage, the "Zener voltage". For this project, you use a diode that has a Zener voltage of 2.4 V and needs a minimum current of 5 mA to start conducting at the right time. Adding the Zener diode in parallel to the resistor will cause the voltage over your Teensy to never exceed 2.4 V, protecting the microcontroller in the event of a short circuit on the load side. Of course the opposite can still happen: if the battery is connected in reverse and you have a short circuit, the Zener diode will not protect the Teensy because there is no "negative" Zener voltage limit. That's why you add a normal diode as well to prevent any negative voltage occurring over the Teensy.

Beware - you are not protected from all type of short circuits!

The installation of high side load switch does not prevent all types of short circuits and hence damage to your battery and/or quadcopter can still occur. Therefore, always make sure that you do not have any shorts in your circuit using your multimeter before connecting the battery. One example of a short circuit that the switch does not protect against is visualized in the figure below. Your quadcopter comes with a connection for a video transmitter (VTX), in case you want to add an FPV camera. This VTX connection is directly connected to the positive and negative power lines but it has a much smaller trace width than the power lines. The VTX traces are therefore not suited to accommodate high currents, and certainly not a short circuit current that can exceed 180 A before it is shut off by the high side load switch. This means that if you would try to create a short circuit by connecting the VTX Vbat and GND lines to each other and subsequently connect the battery, the traces on your power distribution board will start to burn and be destroyed.



Combining current and voltage for battery monitoring

It is not possible to test the current measurement on a solderless breadboard, since it requires currents too high for the traces on the breadboard. Instead, this part will explain the necessary coding and reasoning for accurate battery monitoring. All lines will be directly implemented in the flight controller and can be tested once the quadcopter is fully constructed.

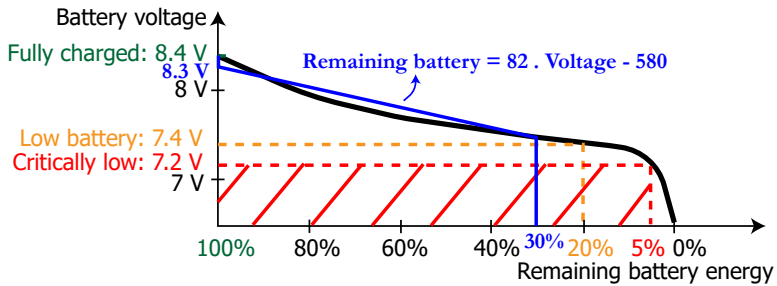
The variables necessary for accurately monitoring the battery capacity - besides the voltage and current - are the remaining battery capacity, the battery capacity at start, the current that you have consumed during flight and the default battery capacity. All battery capacity variables have the unit mAh; a battery of 1300 mAh can sustain a current of 1 A or 1000 mA during a period of $1300 \text{ mAh} / 1000 \text{ mA} = 1.3$ hours. The default battery used in this project has a capacity of 1300 mAh; initialize the `BatteryDefault` variable with this number.

The current can be read from the voltage of pin 21 using the function `analogRead`. Remember that the default resolution for `analogRead` is equal to 10 bit, so a voltage of 0 V gives you the digital number 0 and the maximal input voltage of 3.3 V gives the digital number $2^{10}-1=1023$. Moreover you have built a system for which you have a voltage of 0.036 V for each Ampere flowing through the power switch. This means that the current flowing through the power switch is equal to the measured digital number at pin 21 divided by $((1023 / 3.3 \text{ V}) \times 0.036 \text{ V/A})$. Or equivalently, multiplying the measured digital number with 0.089.

When connecting the battery, you first need an indication of the actual battery capacity before you can calculate its evolution during flight. Luckily, the measurement of the battery voltage using our voltage divider and pin 15 is highly accurate if the motors are not started yet (and thus no voltage drop is present). In the setup phase, you hence determine the battery level using pin 15. There is only a (quasi) linear relation of the battery voltage to its capacity between 8.3 V and 7.5 V. If the voltage is higher than 8.3 V, you considered it to be at 100 % capacity (=1300 mAh) and you turn off the red LED. If the voltage lies below 7.5 V, you consider the battery to be at a capacity of 30% or less and the red LED stays illuminated. The following linear approximation between voltage and capacity is valid for the 1300 mAh - 2S battery:

$$\text{Remaining capacity [\%]} = 82 \cdot \text{Voltage} - 580$$

This approximation can be extracted experimentally with a sophisticated battery charger that indicates both the charged current and the actual voltage and subsequently plot it in a graph, like the one on top of the next page.



```

1 float Voltage, Current, BatteryRemaining, BatteryAtStart;
2 float CurrentConsumed=0;
3 float BatteryDefault=1300;

```

Define the battery monitoring variables

```

4 void battery_voltage(void) {
5     Voltage=(float)analogRead(15)/62;
6     Current=(float)analogRead(21)*0.089;
7 }

```

Measure the voltage and current of the circuit

```

8 void setup() {
9     digitalWrite(5, HIGH);
10    battery_voltage();
11    if (Voltage > 8.3) { digitalWrite(5, LOW);
12        BatteryAtStart=BatteryDefault; }
13    else if (Voltage < 7.5) {
14        BatteryAtStart=30/100*BatteryDefault ;}
15    else { digitalWrite(5, LOW);
16        BatteryAtStart=(82*Voltage-580)/100*
17        BatteryDefault; }
18 }

```

Determine the battery capacity prior to flight



During flight, you use the measured current to follow the evolution of your battery capacity. Since each iteration k in our main loop takes 0.004 seconds, the consumed current can be calculated with the formula:

$$\text{Current}_{\text{Consumed}}(k)[\text{mAh}] = \text{Current}_{\text{Measured}}(k)[\text{A}] \cdot \frac{1000 \frac{\text{mA}}{\text{A}}}{3600 \frac{\text{s}}{\text{h}}} \cdot 0.004 \text{ s} + \text{Current}_{\text{Consumed}}(k-1)[\text{mAh}]$$

The remaining capacity is subsequently calculated by subtracting the consumed current from the battery capacity at startup. When the battery capacity falls below 30%, illuminate the red LED.

Operating the quadcopter without power switch

The high-side power switch is one of the more exotic components in this project and can be hard to come by. Most quadcopters made by hobbyist do not contain such a feature. Although it is not recommended, you can decide to exclude the power switch by shorting the battery tab on the printed circuit board, as illustrated on the figure to the right. Be aware that this removes any short-circuit protection and any control over switching on and off your quadcopter, other than physically connecting or removing the battery via the XT60 plug. The slide switch, (Zener) diodes and 510 Ω resistor are not necessary anymore because you will not have any current sensing capabilities. This means that a voltage measurement is the only way of monitoring the remaining battery energy. The code for this basic method is displayed below; once the voltage drops below 7.5 V the red LED is illuminated. Remember that when voltage drops during for example a sudden throttle increase, the measured voltage does not give an accurate reflection of the remaining battery energy.

```

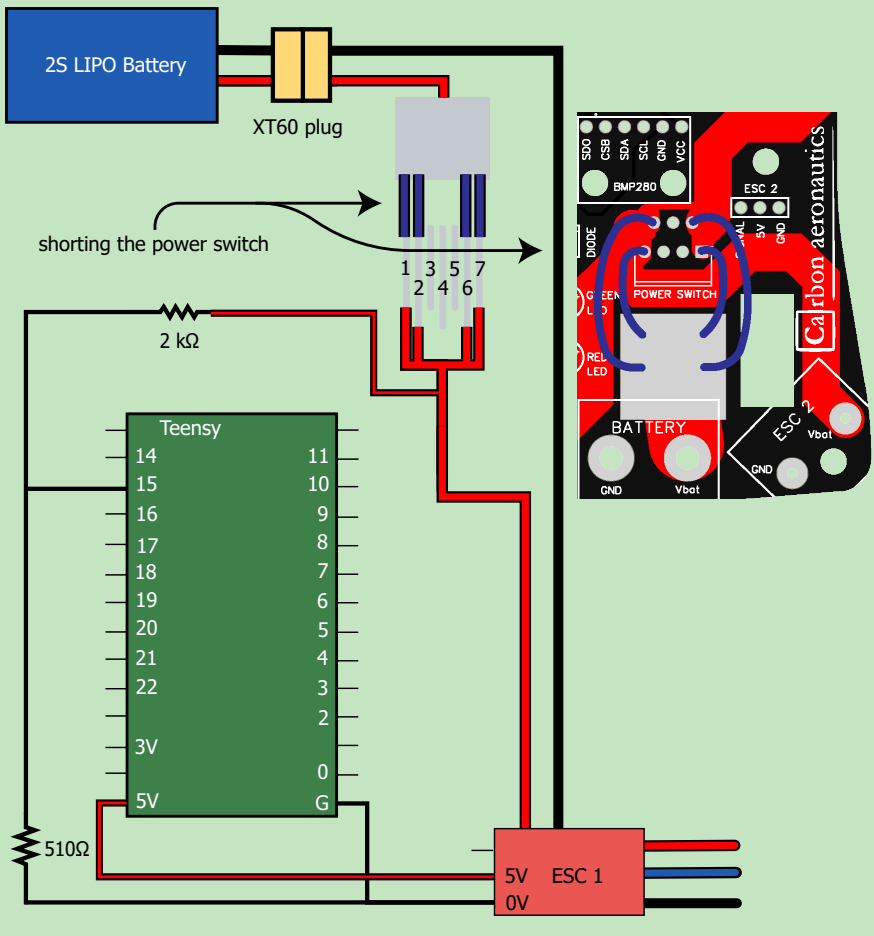
1 float Voltage;
2 void battery_voltage(void) {
3     Voltage=(float)analogRead(15)/62;
4 }
5 void setup() {
6     if (Voltage > 7.5) digitalWrite(5, LOW);
7 }
8 void loop() {
9     battery_voltage();
10    if (Voltage < 7.5) digitalWrite(5, HIGH);
11    else digitalWrite(5, LOW);
12 }
```

```

19 void loop() {
20     battery_voltage();
21     CurrentConsumed=Current*1000*0.004/3600+
22         CurrentConsumed;
23     BatteryRemaining=(BatteryAtStart-
24         CurrentConsumed)/BatteryDefault*100;
25     if (BatteryRemaining<=30) digitalWrite(5, HIGH);
26     else digitalWrite(5, LOW);
27 }

```

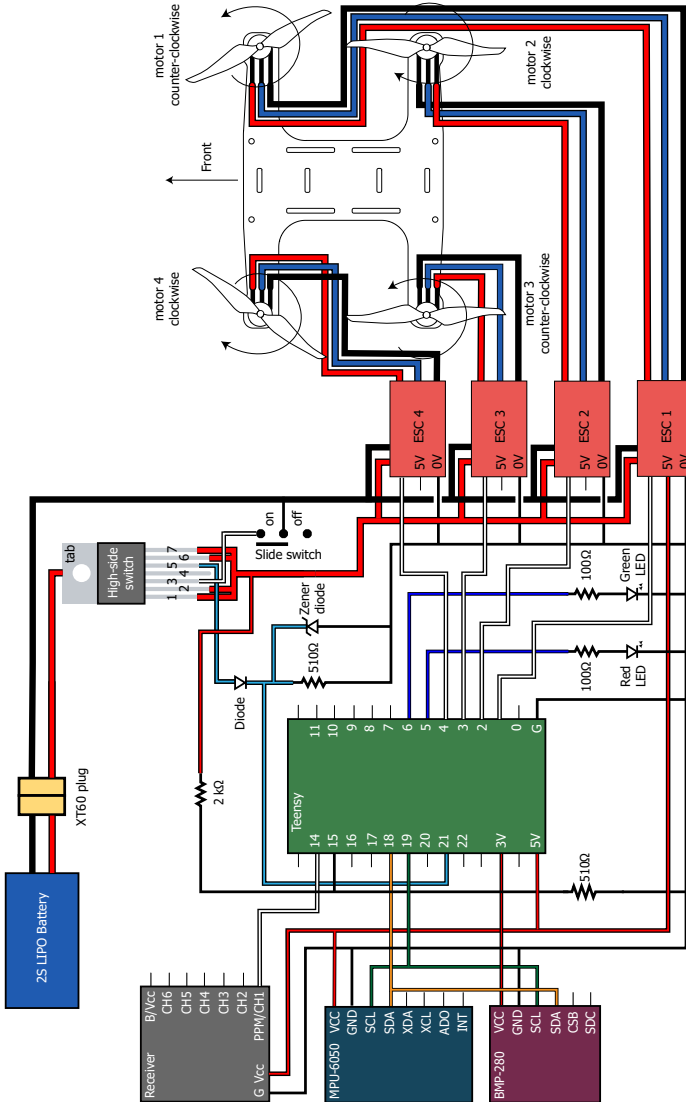
Calculate the battery capacity during flight





Project 10

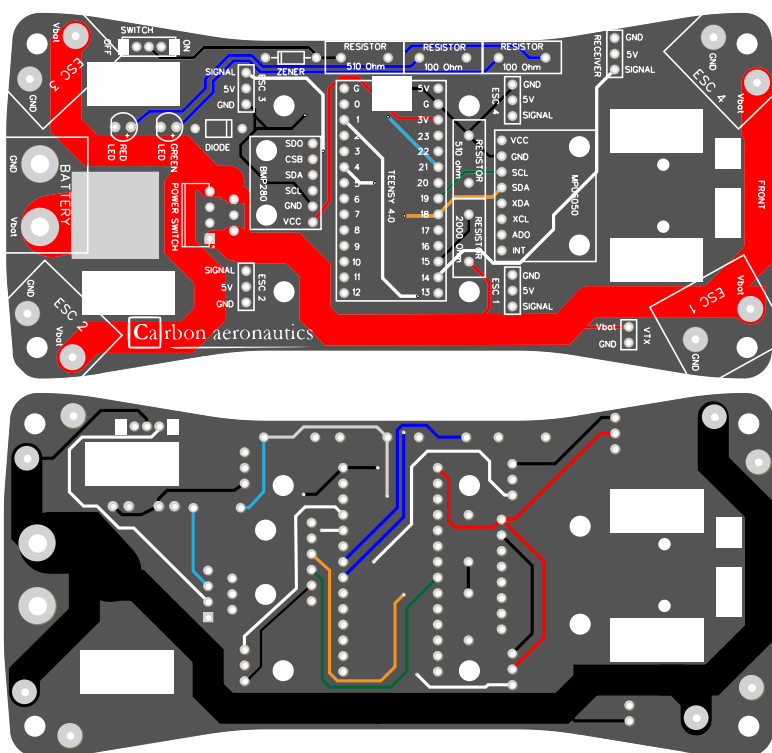
Assembling your quadcopter



Build your quadcopter

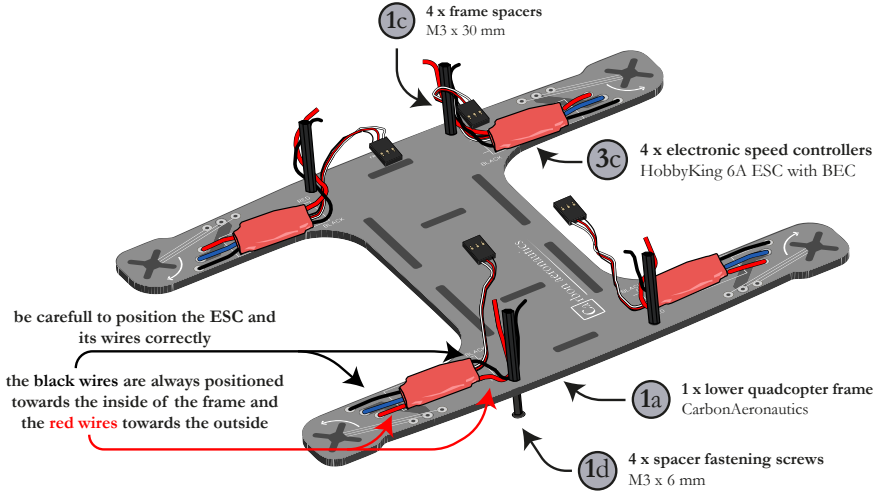
Now that you understand and tested all electronic components of your quadcopter, it is finally time to put it all together! The lower quadcopter frame will house all power electronics and the motors, while the upper frame mainly holds the control electronics and the power distribution.

The upper part of the quadcopter frame houses all the connections necessary for powering and controlling your quadcopter. In essence, it is a so called Printed Circuit Board or PCB: this is nothing more than alternating layers of insulating material and conductive copper. Inside your PCB, all necessary connections between the components are provided in the form of conductive traces. This means that most wires in the schematic of your quadcopter electronics on the left are already integrated in the upper quadcopter frame. The upper and bottom layer of your upper frame PCB are visualized below, together with the traces which are coloured similarly to the wires in the schematic. Notice that the power traces are much wider than the signal traces.

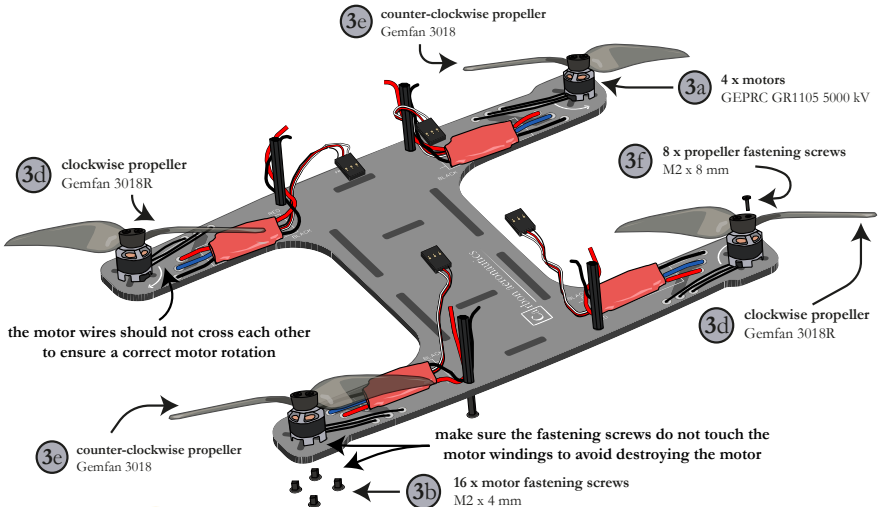


You are now ready to start the quadcopter assembly. During the first two steps, you position the four ESC and motors on the lower quadcopter frame and solder each of the three wires coming out of the ESC and motors to the frame. To ensure a correct rotation direction of the motors, make sure that the cables do not cross each other. The black and red wires coming out of the ESC should match the colours indicated on the frame itself.

ESC and spacer assembly

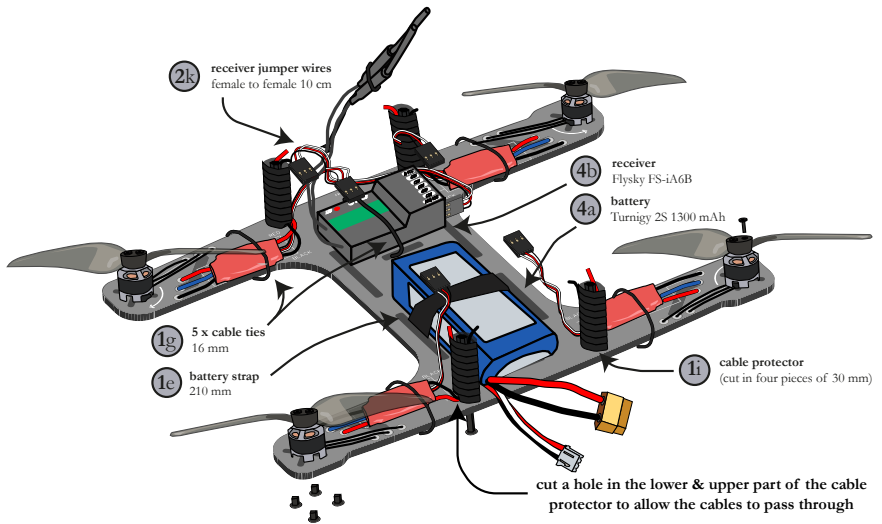


Motor and propeller assembly

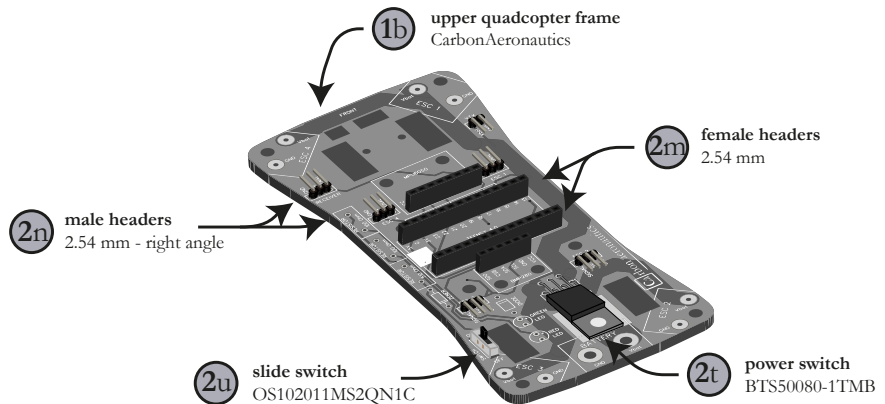


Use the lower frame to hold the receiver and the actual battery. Attach the receiver with a cable tie to the frame and provide a future connection for its signal and power to the microcontroller using a jumper wire. Attach the battery to the frame with a battery strap, making future battery replacement easy. Protect the cables coming from the ESCs using cable protectors, which you cut in pieces such that they cover the full length of the frame spacers. Cut some additional holes to allow the cables to go to the ESC itself and to the upper frame. Once finished, start mounting the necessary headers and switches on the upper quadcopter frame.

Battery and receiver assembly

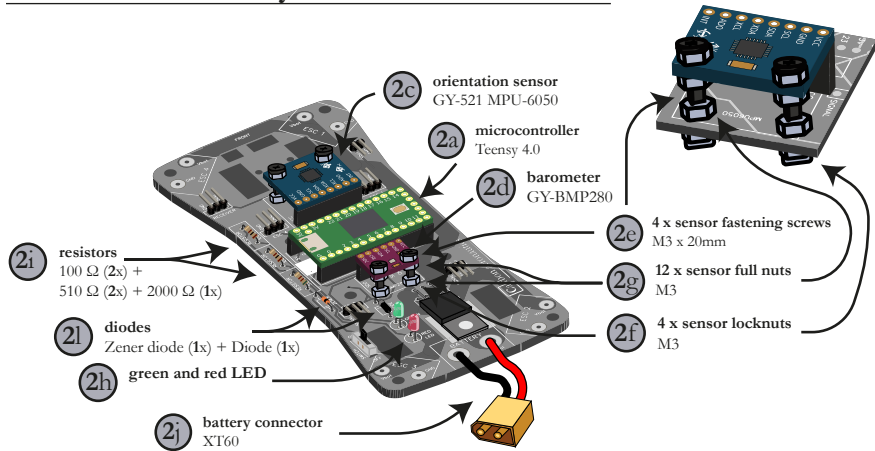


Headers and switches assembly

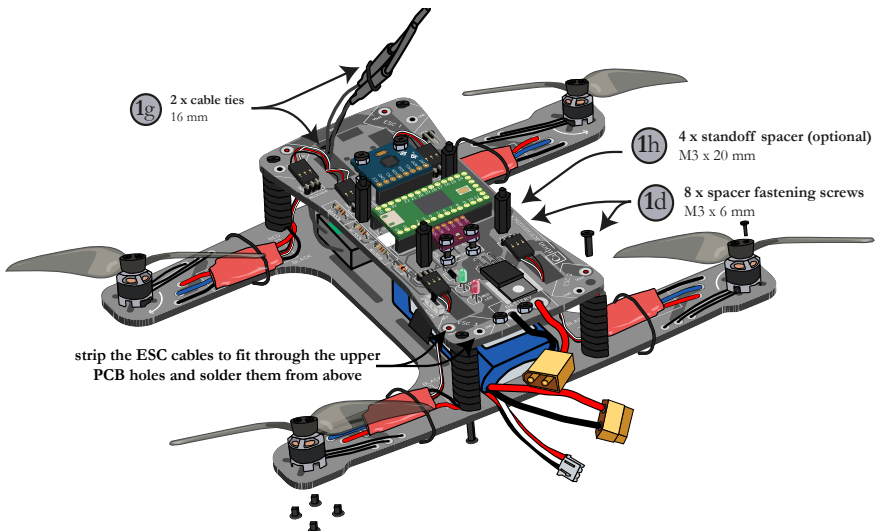


Now solder the additional electronics to the upper quadcopter frame: the resistors, diodes and battery connector. Assemble the orientation sensor and the barometer to the frame using fastening screws, full nuts and locknuts. The use of the barometer will be explained further in the project. Slide the microcontroller in the headers to finalize the upper frame. To connect the upper to the lower frame, first solder the ESC power cables to the upper frame then connect both frames using fastening screws to each other with the spacers.

Electronics assembly



Upper and lower frame assembly



Testing

Now that construction is finished, it is time to check the correct wiring and soldering of all components. **Do this before connecting the battery.** While this may sound boring, it is an essential step after assembling any complex product and will make troubleshooting in the next phases easier. A good test procedure would be to:

- Verify all connections using a multimeter and the schematic given at the beginning of this project.
- Verify that there are no short circuits between wires/pins that should not be connected with each other, also through the use of your multimeter. Pay extra attention to the absence of shorts between the red and black wire of the XT60 battery connector.
- Next, apply power to the prototype board using the USB port of the Teensy. Install the previous Arduino programs that you developed to illuminate the LEDs, read the gyro data and read the receiver data. Verify that they function correctly; this ensures a thorough check of your correct soldering and wiring.

When you are sure that there are no short circuits, connect the battery to continue testing:

- Measure the battery voltage by installing the correct Arduino program.

In the last step, you will test the motors and their correct rotation direction. Connect one ESC with channel 3 of the receiver, just like you did previously. Make sure you configure your radiotransmitter back to PWM instead of PPM. Turn on the radiotransmitter through the POWER button. Connect your battery with the XT60 plug. You should once again hear one beep from your radiotransmitter which indicates that it is connected with the receiver, and subsequently four beeps from the motor. Now slowly increase the throttle stick and verify that the motor turns in the required direction. Carry out the same test for all motors.

When (one of) the motors do not work, verify that:

- The battery is connected;
- The red LED on the receiver is illuminated continuously (a blinking led indicates that the transmitter is not connected, no led means no power to the ESC);
- The ESC is connected to channel 3 of the receiver;
- The transmitter setting is PWM instead of PPM.

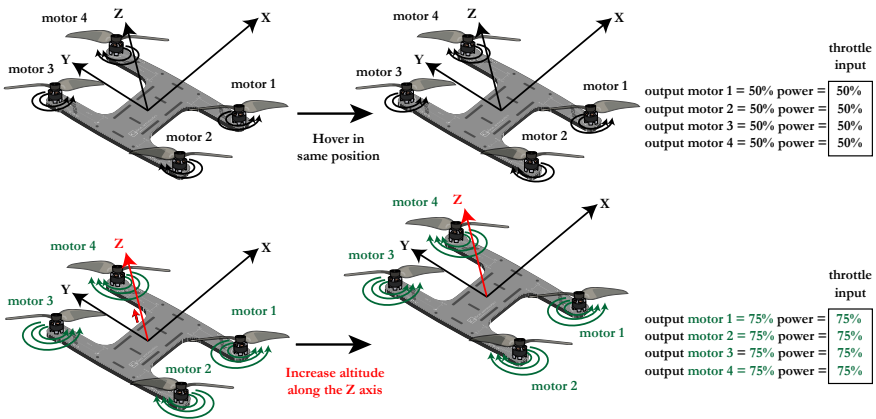
If none of the above verifications solves the problem, verify that you soldered all respective wires correctly and resolder where necessary.





Project 11

Quadcopter dynamics



When you want to change the direction of your drone things become more tricky; assume you want the drone to stay at the same altitude but move sideways to the right (= roll around the X axis). The throttle input will be equal to 50% for all motors as you do not change altitude, but in order to initiate this sideways movement the power output of motors 3 and 4 (=the left motors) should be higher than the power output of motors 1 and 2 (=the right motors). This means that you need an additional **roll input**, which will lower the power of motors 1 and 2 with for example 25% and at the same time increase the power of motors 3 and 4 with 25%. The same reasoning holds for the **pitch input** and the **yaw input**, but with other motor combinations as displayed in the figure on the right.

A nice property of this definition of throttle, roll, pitch and yaw input is that you can write all movements as a linear combination of each other, for all motor outputs:

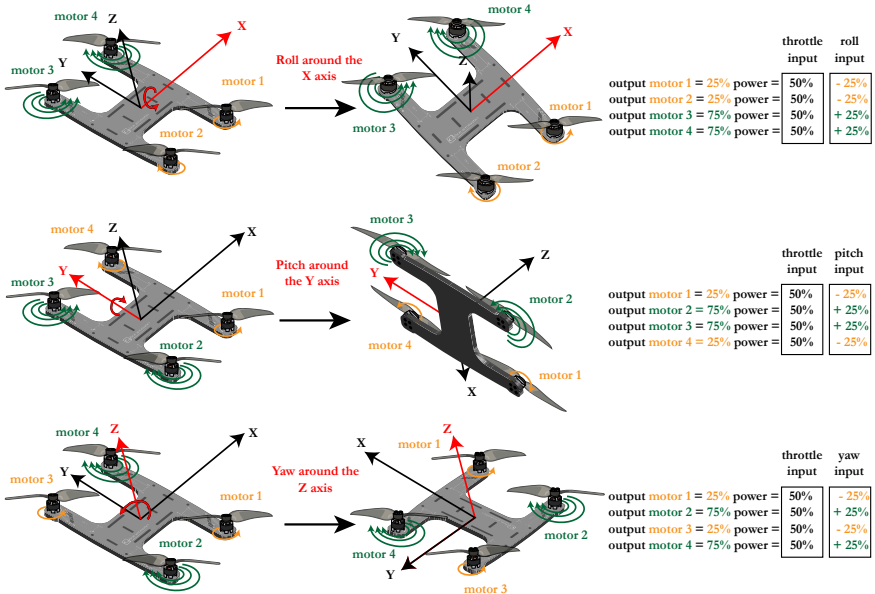
$$\begin{aligned} \text{output motor 1} &= \text{throttle input} - \text{roll input} - \text{pitch input} - \text{yaw input} \\ \text{output motor 2} &= \text{throttle input} - \text{roll input} + \text{pitch input} + \text{yaw input} \\ \text{output motor 3} &= \text{throttle input} + \text{roll input} + \text{pitch input} - \text{yaw input} \\ \text{output motor 4} &= \text{throttle input} + \text{roll input} - \text{pitch input} + \text{yaw input} \end{aligned}$$

Learn how your quadcopter travels in space

With all ingredients for a control system available and tested, it is time to learn how a quadcopter moves through space with your inputs given to the radio-transmitter.

You already know how to control the motors with your Teensy, how to measure the rotation rate with the gyroscope and how to receive and read commands with your radiotransmitter and receiver. These are all essential ingredients, but you still need to learn how they have to work together to be able to fly.

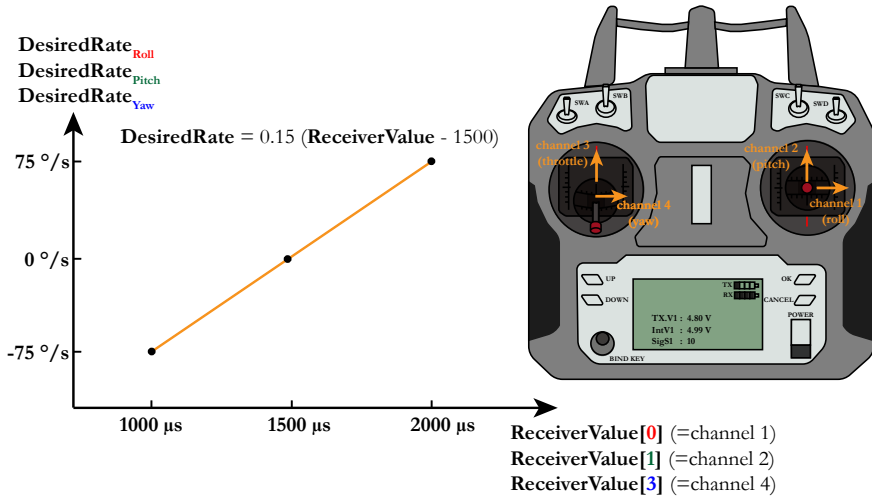
The first thing you need to understand is how you can use the four motors of your quadcopter to steer it in the directions you want. You do this by changing the power and thus rotation speed of the motors. Let's assume a perfect world for this project: no wind disturbances, instantaneous motor reaction and a uniform weight distribution. To let your quadcopter hover in the same position, each motor will have to work at around 50% of its power as shown on the left figure. To increase the altitude at which your drone is flying, you can simply increase the power of all four motors to for example 75%. To keep the quadcopter level, it is important that all motors should increase their power at the same time in order to keep the quadcopter level. The command to keep all motors at the same power level will be called the **throttle input**.



In reality you do not send a power percentage to the motor but rather a PWM value between 1000 and 2000 μs , where 1000 μs corresponds with 0% motor output and 2000 μs with 100% motor output. In your code, the throttle input will vary between 1000 and 1800 μs to leave 20% motor output available at all times for rolling, pitching and yawing.

From receiver commands to desired rotation rates

The receiver sends commands to the microcontroller that vary between 1000 and 2000 μs according to the position of the radiotransmitter stick. For the throttle stick, this corresponds nicely to 0 and 100% power. For the roll, pitch and yaw sticks, whose default position is physically in the middle of the radiotransmitter at 1500 μs , you need to transform the PWM values to physical rotation rates. You can choose your maximal and minimal desired rotation rate; the higher the values the more agile your drone will be, but also the harder to control. For now, take the limit values of 75 $^\circ/\text{s}$ and -75 $^\circ/\text{s}$. The transformation from the PWM values to the rotational rates is then visualised in the figure below together with the corresponding linear correlation.



Transforming desired rotation rates to motor input?

You might think a second transformation is necessary: from the desired roll/pitch/yaw rotational rates to the motor roll/pitch/yaw input. This is true, but it is very difficult to measure which rotational rates correspond to a certain motor power level. Moreover, even if you obtain accurate data for this transformation, a huge problem remains; the reasoning in this project holds only for a quadcopter in a perfect world. In the real world however, even small disturbances will destabilize your quadcopter if you would introduce a fixed transformation.

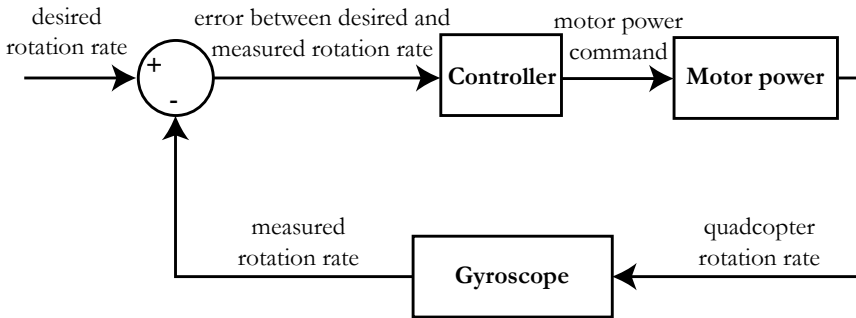
For example, assume that the weight of your quadcopter is slightly higher in the back than in the front. This means that in order to hover, the back motors will need a slightly higher power output than the front motors. Any wind disturbance during the flight will have to be corrected immediately by varying the output of each motor accordingly. It is impossible to adjust for both imperfections using manual corrections within normal human reaction times; these phenomena need to be corrected by your fast microcontroller and a technique called PID feedback control.





Project 12

Quadcopter rate control



Now suppose that the controller just consists of the difference between the desired and the measured rotation rate, multiplied with a constant P:

$$Input_{motor} = P \cdot (DesiredRate - Rate)$$

By defining the error during each iteration k of the control loop:

$$Error(k) = DesiredRate(k) - Rate(k)$$

you can simplify the equation rewriting the motor input during iteration k as:

$$Input_{motor}(k) = P_{term}(k) = P \cdot Error(k)$$

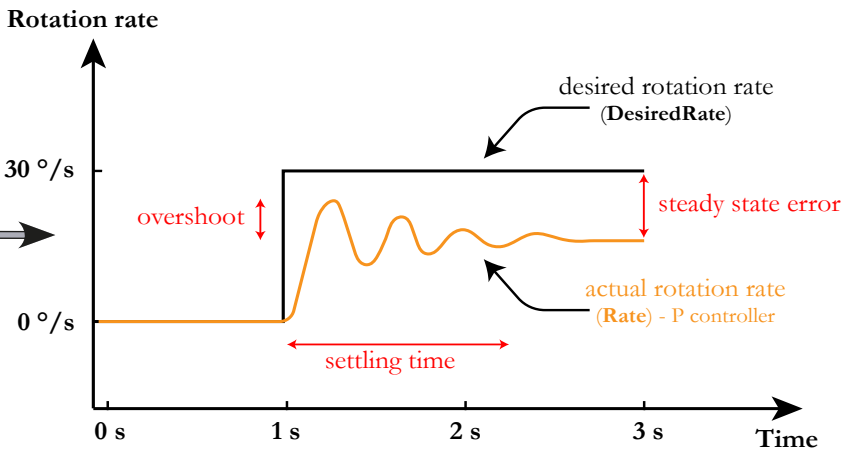
where P_{term} is the **proportional term** of the controller. The response of such a controller to a change in the desired rotation rate is visualised on the graph to the right: the higher you choose the value for P to be, the faster the actual rotation rate will approach the desired rotation rate and the smaller the settling time, which is a good thing. However, a larger P will also give a larger overshoot, meaning that the quadcopter might bounce violently when changing the desired rotation rate. Whatever the value of P there might also be a steady state error: the actual rotation rate never reaches the desired rotation rate. You overcome this issue by adding an **integral term**: this term will sum the past errors hence eliminating the steady state error.

Learn how to stabilize your quadcopter

With normal human reaction times, it is not possible to keep a quadcopter stable in the air. In this project you learn how to use a fast control loop to stabilize the quadcopter automatically while also taking into account the commands you give it with the radiotransmitter.

To stabilize your quadcopter, you need to use a very fast automated control loop that sends new commands to each of the four motors, multiple times per second. The control system that you will use for your quadcopter will be a 250 Hz system; this means that every $1/250 = 0.004$ seconds, all four motors will receive new commands. These commands are generated depending on the commands you give yourself through the radiotransmitter, but are also generated automatically based on the actual rotation rate of the quadcopter in space, which is measured by your gyroscope.

The closed control loop that you will use for the roll, pitch and yaw rotation rates is displayed on the left. You use the gyroscope sensor to measure the actual rotation rate of the quadcopter and compare it to the desired rotation rate, which you have sent from the radiotransmitter. The error between both is transformed by the controller to a motor power command that is sent to each of the four motors. The resulting change in motor power changes the rotation rate of the quadcopter to a value that should be closer to the desired rotation rate than before. The actual rotation rate is measured once again and the process restarts. Each loop occurs every 0.004 seconds during flight.



The addition of the integral term can be implemented in the control equation through:

$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) = P \cdot Error(k) + I \cdot \int_0^{k \cdot T_s} Error(t) \cdot dt$$

where the T_s is the length of one iteration, 0.004 s for our 250 Hz control loop. Discretization of the integral can easily be done through:

$$Input_{motor}(k) = P \cdot Error(k) + I_{term}(k-1) + I \cdot \frac{(Error(k) + Error(k-1)) \cdot T_s}{2}$$

The figure on the right shows the response of the Proportional-Integral (PI) controller; the steady state error disappeared but the system still has a large overshoot and a long settling time. A final improvement can be realized by adding a **derivative term**. Since a derivative along a function predicts its future value, this term will help to reduce the overshoot and hence the settling time:

$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) + D_{term}(k)$$

$$Input_{motor}(k) = P_{term}(k) + I_{term}(k) + D \cdot \frac{d}{dt} Error(t)$$

the derivative will be discretized as well, giving the final discrete equation for a PID controller:

$$Input_{motor}(k) = P \cdot Error(k) + I_{term}(k-1) + I \cdot \frac{(Error(k) + Error(k-1)) \cdot T_s}{2} + D \cdot \frac{(Error(k) - Error(k-1))}{T_s}$$

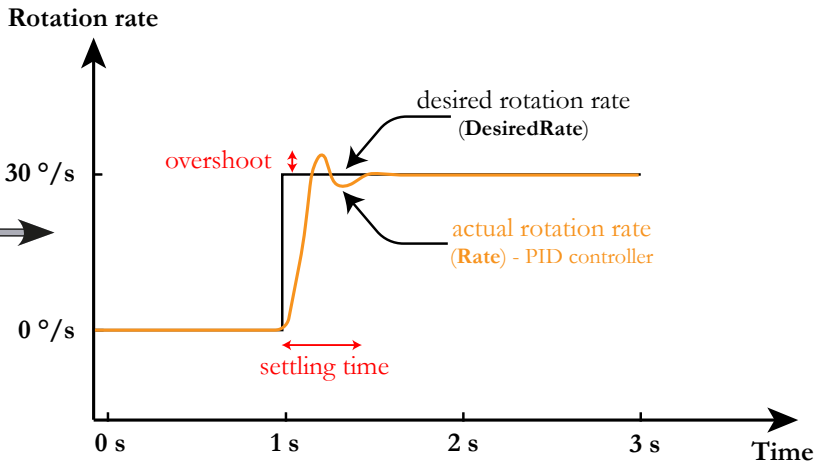
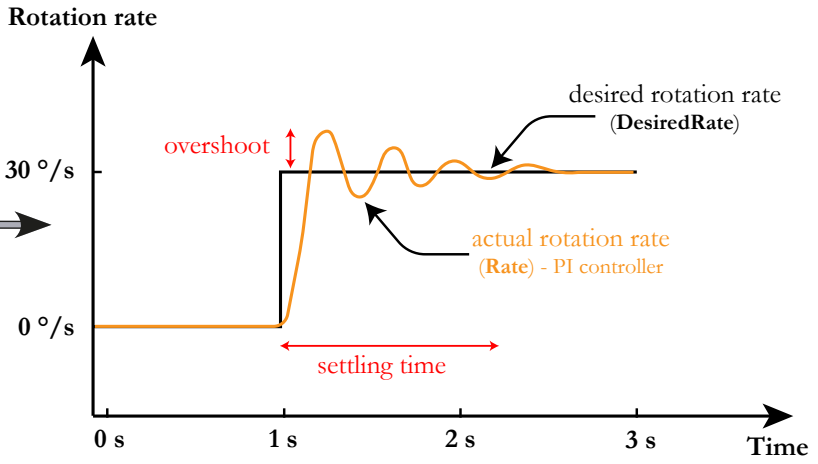
As the figure shows, the PID controller allows the quadcopter to approach the desired rotation rate quite fast with a small overshoot and settling time.

Obviously, the PID controller needs to be implemented for all three rotation rates; roll, pitch and yaw. For the roll rotation rate for example, the PID equation becomes:

$$Input_{Roll}(k) = P_{Roll} \cdot Error_{Roll}(k) + I_{term, Roll}(k-1) + I_{Roll} \cdot \frac{(Error_{Roll}(k) + Error_{Roll}(k-1)) \cdot T_s}{2} + D_{Roll} \cdot \frac{(Error_{Roll}(k) - Error_{Roll}(k-1))}{T_s}$$

which can be further simplified by saving the Iterm and Error during each iteration for the next iteration through the equations $PrevError_{Roll} = Error_{Roll}(k-1)$ and $PrevIterm_{Roll} = Iterm_{Roll}(k-1)$. This way, the iteration indexes k can be removed from the above equation:

$$Input_{Roll} = P_{Roll} \cdot Error_{Roll} + PrevIterm_{Roll} + I_{Roll} \cdot \frac{(Error_{Roll} + PrevError_{Roll}) \cdot T_s}{2} + D_{Roll} \cdot \frac{(Error_{Roll} - PrevError_{Roll})}{T_s}$$



The error for the roll rate is given by the difference between the desired roll rate and the measured roll rate by the gyroscope:

$$Error_{Roll} = DesiredRate_{Roll} - Rate_{Roll}$$

To make the notation easier, simplify the PID equation for the roll rate by saying that the motor input for the roll rate is function of the different parameters:

$$Input_{Roll} = f(Error_{Roll}, P_{Roll}, I_{Roll}, D_{Roll}, PrevError_{Roll}, PrevIterm_{Roll})$$

The above equations can be copied for the pitch and yaw rates in order to get the full PID controller.

In the right figure, the full control loop with equations is visualized. Start each loop with obtaining the commands from receiver, corresponding to the position of the sticks on your radiotransmitter. Transform these receiver values to the desired roll, pitch and yaw rates and the value for the throttle. Next, obtain the actual roll, pitch and yaw rates of the quadcopter from your gyroscope. These are subtracted from the desired rotation rates to obtain the error between both. Now you have the necessary information for the PID equations, which gives you the motor input values for roll, pitch and yaw. Introduce these input values in the motor power commands that were derived in project 11. With the first iteration now complete, wait until 0.004 seconds have passed to start the next iteration.

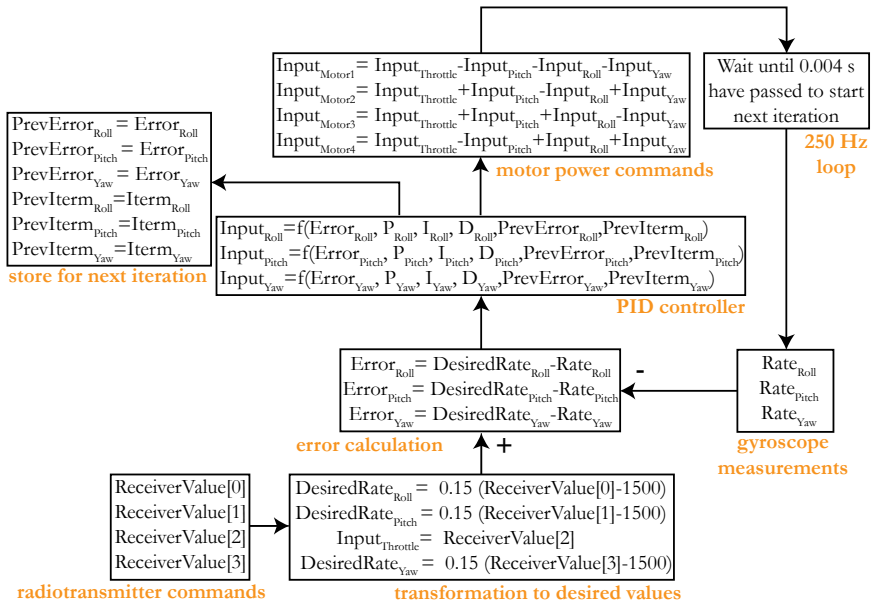
PID tuning

An important unknown that you did not yet determine are the values for P, I and D. These constants need to be chosen in such a way that their combination stabilizes the flight of your quadcopter. The following values are a good compromise between agility and stability for your quadcopter for all tested motor/ESC/propeller/battery combinations (see project 1):

- $P_{RateRoll} = P_{RatePitch} = 0.6$
- $I_{RateRoll} = I_{RatePitch} = 3.5$
- $D_{RateRoll} = D_{RatePitch} = 0.03$

Notice that the values for the roll and pitch rates are equal; this is evident since the quadcopter is (almost) symmetrical in both directions. For the yaw rates, the PID values are:

- $P_{RateYaw} = 2$
- $I_{RateYaw} = 12$
- $D_{RateYaw} = 0$



Finding these optimal values is not easy; there exist some basic methods to obtain them through calculations, but in the end you will always have to test and retest to find the values that work for your quadcopter. Usually the trail and error method is done by first choosing and testing a P value, then a value for I and finally also a value for D. The values can be chosen with the following guidelines:

- A high P value increases the responsiveness of your quadcopter, but a too high P value will cause your quadcopter to overcorrect and experience high frequency oscillations.
- A high I value stops unwanted drifting of your quadcopter, but a too high I value will cause your quadcopter to feel unresponsive.
- Finally the D value reduces the oscillations caused by the P value. Setting the D value too high causes motor vibrations.

It can be quite cumbersome to test different PID values, fortunately it only needs to be done once for each design.





Project 13

The flight controller: rate mode

```
1 #include <Wire.h>
2 float RatePitch, RateRoll, RateYaw;
3 float RateCalibrationPitch, RateCalibrationRoll,
4     RateCalibrationYaw;
5 int RateCalibrationNumber;

6 #include <PulsePosition.h>
7 PulsePositionInput ReceiverInput(RISING);
8 float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
9 int ChannelNumber=0;

10 float Voltage, Current, BatteryRemaining, BatteryAtStart;
11 float CurrentConsumed=0;
12 float BatteryDefault=1300;

13 uint32_t LoopTimer;

14 float DesiredRateRoll, DesiredRatePitch,
15     DesiredRateYaw;
16 float ErrorRateRoll, ErrorRatePitch, ErrorRateYaw;
17 float InputRoll, InputThrottle, InputPitch, InputYaw;
18 float PrevErrorRateRoll, PrevErrorRatePitch,
19     PrevErrorRateYaw;
20 float PrevItermRateRoll, PrevItermRatePitch,
21     PrevItermRateYaw;
22 float PIDReturn[]={0, 0, 0};
23 float PRateRoll=0.6 ; float PRatePitch=PRateRoll;
24     float PRateYaw=2;
25 float IRateRoll=3.5 ; float IRatePitch=IRateRoll;
26     float IRateYaw=12;
27 float DRateRoll=0.03 ; float DRatePitch=DRateRoll;
28     float DRateYaw=0;
```

Define the gyro variables (projects 4 and 5)

Define the receiver variables (project 7)

Define the battery variables (project 9)

Define the parameter containing the length of each control loop

All variables necessary for the PID control loop are declared in this part, including the values for the P, I and D parameters

Create your first flight controller

After a lot of hard work, you now have all the ingredients to create your first flight controller and test it on your quadcopter. Let's put all pieces together!

```
23 float MotorInput1, MotorInput2, MotorInput3,  
    MotorInput4;
```

Declare the input variables to the motors

```
24 void battery_voltage(void) {  
25     Voltage=(float)analogRead(15)/62;  
26     Current=(float)analogRead(21)*0.089;  
27 }
```

Battery function (projects 3 and 9)

```
28 void read_receiver(void){  
29     ChannelNumber = ReceiverInput.available();  
30     if (ChannelNumber > 0) {  
31         for (int i=1; i<=ChannelNumber;i++){  
32             ReceiverValue[i-1]=ReceiverInput.read(i);  
33         }  
34     }  
35 }
```

Receiver function (project 7)

```
36 void gyro_signals(void) {  
37     Wire.beginTransmission(0x68);  
38     Wire.write(0x1A);  
39     Wire.write(0x05);  
40     Wire.endTransmission();  
41     Wire.beginTransmission(0x68);  
42     Wire.write(0x1B);  
43     Wire.write(0x08);  
44     Wire.endTransmission();  
45     Wire.beginTransmission(0x68);  
46     Wire.write(0x43);  
47     Wire.endTransmission();  
48     Wire.requestFrom(0x68,6);  
49     int16_t GyroX=Wire.read()<<8 | Wire.read();  
50     int16_t GyroY=Wire.read()<<8 | Wire.read();  
51     int16_t GyroZ=Wire.read()<<8 | Wire.read();
```

Gyro function (project 4)

Define a function that is called for each PID calculation of the roll, pitch and yaw rotation rates. You saw the equations for each term already in project 12. An important addition here is the limit of $400 \mu\text{s}$ on the I term; this term is used to avoid integral windup. Integral windup is a phenomena in which the integral term accumulates a significant error due saturation and causes a large overshoot. For example when the quadcopter cannot achieve the desired setpoint because it did not yet lift off the ground. Another limit is placed on the full output, to avoid a significant imbalance between to roll, pitch and yaw commands to the motor.

Return the values for the motor command, the error and the integral term from the PID equations to the main program.

To ensure a bumpless restart after landing your quadcopter, the PID error and integral values that are passed to the next iterations need to be reset once you land the quadcopter. This is also necessary to avoid any windup as well.

At the start of the setup phase, the red LED connected with pin 5 is illuminated to show that the microcontroller is still in the setup phase. As usual, the LED on the Teensy itself is illuminated as well to show that it receives power.

```

52     RateRoll=(float)GyroX/65.5;
53     RatePitch=(float)GyroY/65.5;
54     RateYaw=(float)GyroZ/65.5;
55 }

```

```

56 void pid_equation(float Error, float P , float I, float D,
    float PrevError, float PrevIterm) {
57     float Pterm=P*Error;
58     float Iterm=PrevIterm+I*(Error+
        PrevError)*0.004/2;
59     if (Iterm > 400) Iterm=400;
60     else if (Iterm <-400) Iterm=-400;
61     float Dterm=D*(Error-PrevError)/0.004;
62     float PIDOutput= Pterm+Iterm+Dterm;
63     if (PIDOutput>400) PIDOutput=400;
64     else if (PIDOutput <-400) PIDOutput=-400;

```

PID function

```

65     PIDReturn[0]=PIDOutput;
66     PIDReturn[1]=Error;
67     PIDReturn[2]=Iterm;
68 }

```

Return the output from the PID function

```

69 void reset_pid(void) {
70     PrevErrorRateRoll=0; PrevErrorRatePitch=0;
    PrevErrorRateYaw=0;
71     PrevItermRateRoll=0; PrevItermRatePitch=0;
    PrevItermRateYaw=0;
72 }

```

PID reset function

```

73 void setup() {
74     pinMode(5, OUTPUT);
75     digitalWrite(5, HIGH);
76     pinMode(13, OUTPUT);
77     digitalWrite(13, HIGH);

```

Visualize the setup phase using the red LED

```

78     Wire.setClock(400000);
79     Wire.begin();
80     delay(250);
81     Wire.beginTransmission(0x68);
82     Wire.write(0x6B);
83     Wire.write(0x00);
84     Wire.endTransmission();

```

Communication with the gyroscope and calibration (project 4 and 5)



```
85     for (RateCalibrationNumber=0;
        RateCalibrationNumber<2000;
        RateCalibrationNumber++) {
86         gyro_signals();
87         RateCalibrationRoll+=RateRoll;
88         RateCalibrationPitch+=RatePitch;
89         RateCalibrationYaw+=RateYaw;
90         delay(1);
91     }
```

When the time consuming part of the setup process is finished, illuminate the green LED to show that the quadcopter is able to start. However, only dim the red LED when the battery voltage is higher than 7.5 V.

SAFETY RELATED LINES: just before finishing the setup process, you need to check that the throttle stick is in its lowest position. Otherwise, if you accidentally left the throttle stick in a higher position and the radiotransmitter is not nearby, the motors could suddenly start after the setup process without you controlling it. With these lines, you stay in an infinite `while` loop until you move the throttle stick between 1020 and 1050 μs (so moving it from the lowest position to a slightly higher position).

In the last line of the setup process, start a timer that will count the time in the loop process and go to the next iteration after exactly 4000 μs or 0.004 s, to create a $1/0.004 \text{ s} = 250 \text{ Hz}$ control loop.

```
92 RateCalibrationRoll/=2000;
93 RateCalibrationPitch/=2000;
94 RateCalibrationYaw/=2000;
```

```
95 analogWriteFrequency(1, 250);
96 analogWriteFrequency(2, 250);
97 analogWriteFrequency(3, 250);
98 analogWriteFrequency(4, 250);
99 analogWriteResolution(12);
```

Set the PWM frequency to 250 Hz and the resolution to 12 bit for all motors (project 8)

```
100 pinMode(6, OUTPUT);
101 digitalWrite(6, HIGH);
102 battery_voltage();
103 if (Voltage > 8.3) { digitalWrite(5, LOW);
104     BatteryAtStart=BatteryDefault; }
105 else if (Voltage < 7.5) {
106     BatteryAtStart=30/100*BatteryDefault ;}
107 else { digitalWrite(5, LOW);
108     BatteryAtStart=(82*Voltage-580)/100*
109     BatteryDefault; }
```

Show the end of the setup process and determine the initial battery voltage percentage (project 9)

```
109 ReceiverInput.begin(14);
110 while (ReceiverValue[2] < 1020 ||
111     ReceiverValue[2] > 1050) {
112     read_receiver();
113     delay(4);
114 }
```

Avoid accidental lift off after the setup process

```
114 LoopTimer=micros();
115 }
```

Start the timer

```
116 void loop() {
117     gyro_signals();
118     RateRoll-=RateCalibrationRoll;
119     RatePitch-=RateCalibrationPitch;
120     RateYaw-=RateCalibrationYaw;
```

Measure the rotation rates and subtract the calibration values (project 5)

```
121 read_receiver();
```

Read the receiver commands (project 7)



Transform the commands from the receiver in μs to the desired roll, pitch and yaw rates in $^{\circ}/\text{s}$ as explained in project 11. The throttle command remains in μs .

Calculate the difference between the desired rotation rates and the measured rotation rates.

Start the PID calculations for each of the three rotation rates. The outputs of these calculations are stored in the array `PIDReturn`; the roll/pitch/yaw input for the motors is stored in position 0, the error value and value for the Iterm that needs to be used for the next iteration is stored in positions 1 and 2. Retrieve these values each time for the corresponding rotation rate to be able to use them in the next iteration.

With the throttle stick, you are able to go to 2000 μs , which would give maximal power to all four motors. However, this would give no room to stabilize the roll, pitch and yaw rates. That is why you limit the throttle value to 1800 μs or 80%.

Now calculate the motor inputs with the quadcopter dynamics equations you saw during project 11. Remember to convert the throttle values in μs to their 12 bit equivalent by multiplying them with 1.024.

```

122 DesiredRateRoll=0.15*(ReceiverValue[0]-1500);
123 DesiredRatePitch=0.15*(ReceiverValue[1]-1500);
124 InputThrottle=ReceiverValue[2];
125 DesiredRateYaw=0.15*(ReceiverValue[3]-1500);

```

Calculate the desired roll, pitch and yaw rates

```

126 ErrorRateRoll=DesiredRateRoll-RateRoll;
127 ErrorRatePitch=DesiredRatePitch-RatePitch;
128 ErrorRateYaw=DesiredRateYaw-RateYaw;

```

Calculate the errors for the PID calculations

```

129 pid_equation(ErrorRateRoll, PRateRoll, IRateRoll,
    DRateRoll, PrevErrorRateRoll,
    PrevItermRateRoll);
130 InputRoll=PIDReturn[0];
    PrevErrorRateRoll=PIDReturn[1];
    PrevItermRateRoll=PIDReturn[2];
131 pid_equation(ErrorRatePitch, PRatePitch,
    IRatePitch, DRatePitch, PrevErrorRatePitch,
    PrevItermRatePitch);
132 InputPitch=PIDReturn[0];
    PrevErrorRatePitch=PIDReturn[1];
    PrevItermRatePitch=PIDReturn[2];
133 pid_equation(ErrorRateYaw, PRateYaw,
    IRateYaw, DRateYaw, PrevErrorRateYaw,
    PrevItermRateYaw);
134 InputYaw=PIDReturn[0];
    PrevErrorRateYaw=PIDReturn[1];
    PrevItermRateYaw=PIDReturn[2];

```

Execute the PID calculations

```

135 if (InputThrottle > 1800) InputThrottle = 1800;

```

Limit the throttle output

```

136 MotorInput1= 1.024*(InputThrottle-InputRoll
    -InputPitch-InputYaw);
137 MotorInput2= 1.024*(InputThrottle-InputRoll
    +InputPitch+InputYaw);
138 MotorInput3= 1.024*(InputThrottle+InputRoll
    +InputPitch-InputYaw);
139 MotorInput4= 1.024*(InputThrottle+InputRoll
    -InputPitch+InputYaw);

```

Use the quadcopter dynamics equations (project 11)



Make sure that the inputs to the motors do not exceed $2000 \mu\text{s}$ after the dynamic equations to avoid overloading them.

To avoid stopping the motors in mid-flight, keep them turning at 18% when the motor input decreases below $1180 \mu\text{s}$ (= the `ThrottleIdle` value).

SAFETY RELATED LINES: the previous lines would mean you can never switch off the motors, as they keep turning at minimally 18%. Just before sending the commands to the motors, you add the condition that if the throttle stick is brought to its lowest position (below $1050 \mu\text{s}$), all four motors stop (e.g. the value of `ThrottleCut-Off` is $1000 \mu\text{s}$ or 0% power). Usually you would do this after landing the quadcopter. The PID parameters need to be reset in case you want to have a bumpless restart.

Now you are finally ready to sent the commands to each of the four motors.

The last step in the iteration is to wait until the $4000 \mu\text{s}$ or 0.004 s have passed using a while loop. When this condition is met, reset the timer to the actual time and the program can proceed to the next iteration. Congratulations, you created a 250 Hz control loop!


```

140   if (MotorInput1 > 2000)MotorInput1 = 1999;
141   if (MotorInput2 > 2000)MotorInput2 = 1999;
142   if (MotorInput3 > 2000)MotorInput3 = 1999;
143   if (MotorInput4 > 2000)MotorInput4 = 1999;

```

Limit the maximal power commands sent to the motors

```

144   int ThrottleIdle=1180;
145   if (MotorInput1 < ThrottleIdle) MotorInput1 =
146       ThrottleIdle;
147   if (MotorInput2 < ThrottleIdle) MotorInput2 =
148       ThrottleIdle;
149   if (MotorInput3 < ThrottleIdle) MotorInput3 =
150       ThrottleIdle;
151   if (MotorInput4 < ThrottleIdle) MotorInput4 =
152       ThrottleIdle;

```

Keep the quadcopter motors running at minimally 18% power during flight

```

153   int ThrottleCutOff=1000;
154   if (ReceiverValue[2]<1050) {
155       MotorInput1=ThrottleCutOff;
156       MotorInput2=ThrottleCutOff;
157       MotorInput3=ThrottleCutOff;
158       MotorInput4=ThrottleCutOff;
159       reset_pid();
160   }
161   analogWrite(1,MotorInput1);
162   analogWrite(2,MotorInput2);
163   analogWrite(3,MotorInput3);
164   analogWrite(4,MotorInput4);

```

Make sure you are able to turn off the motors

Sent the commands to the motors

```

165   battery_voltage();
166   CurrentConsumed=Current*1000*0.004/3600+
        CurrentConsumed;
167   BatteryRemaining=(BatteryAtStart-
        CurrentConsumed)/BatteryDefault*100;
168   if (BatteryRemaining<=30) digitalWrite(5, HIGH);
169   else digitalWrite(5, LOW);

```

Keep track of battery level (project 9)

```

170   while (micros() - LoopTimer < 4000);
171   LoopTimer=micros();
172 }

```

Finish the 250 Hz control loop



Before you fly... radiotransmitter failsafe

Imagine you are flying your quadcopter and suddenly, your radiotransmitter loses power or signal. What will happen to your quadcopter? Well, you did not program any return to home function, so it will just keep flying until it runs out of battery or crashes. To avoid this, your radiotransmitter can tell the receiver to give a throttle command of 1000 μ s when it loses connection. This means that the motors of your quadcopter will stop turning and it will fall to the ground - not the ideal solution but better to have some damage than a quadcopter on the loose.

Power on the transmitter \rightarrow hold the OK button for two seconds \rightarrow choose System \rightarrow go down and choose "RX setup" \rightarrow go down and choose "Failsafe". Choose channel 3 (the throttle channel) then click UP or DOWN to activate the failsafe of channel 3 (ON). Now lower the throttle stick to its lowest position and press and hold the CANCEL function to tell the receiver it should send a throttle command of 1000 μ s or 0% when it loses contact with the radiotransmitter. Return to the previous screen on the transmitter, which should show -100% at channel 3. The failsafe is now set.

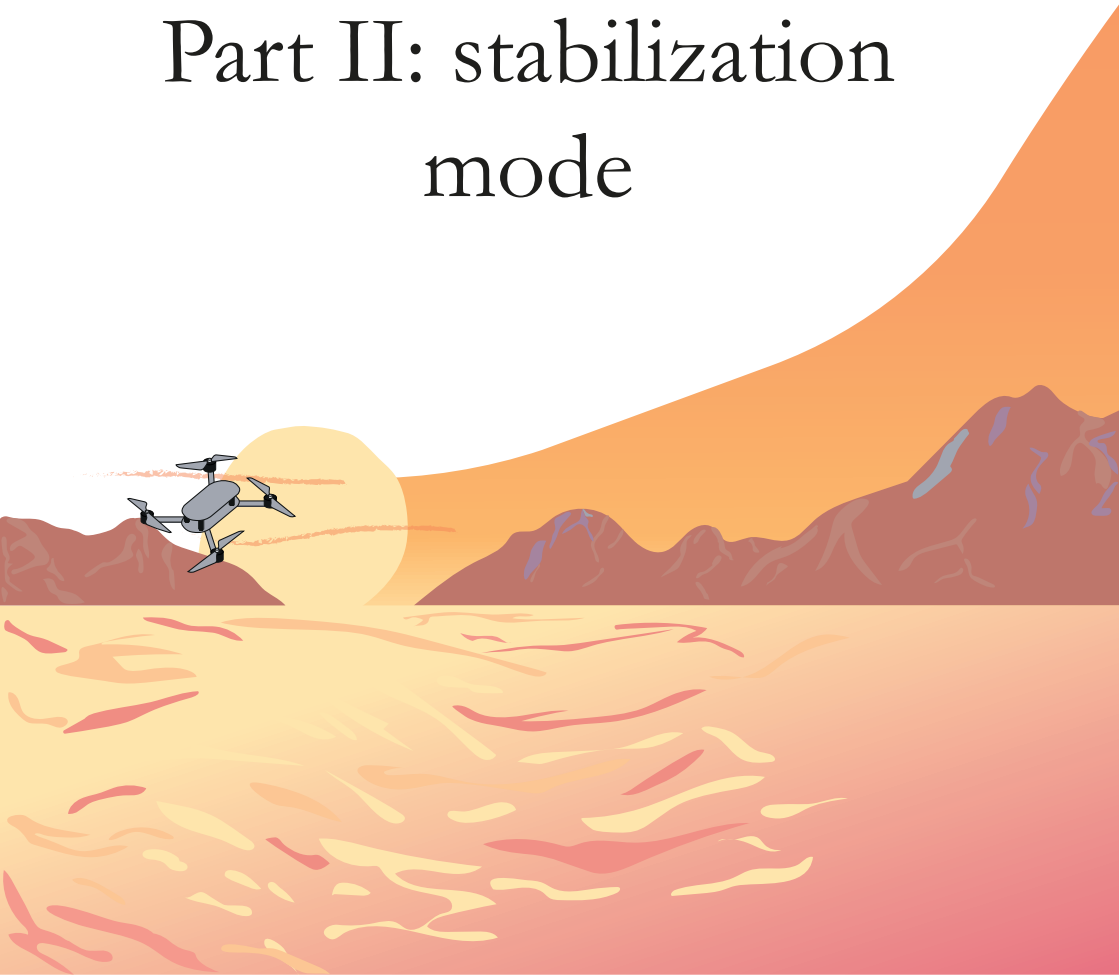
Start-up and flying your quadcopter

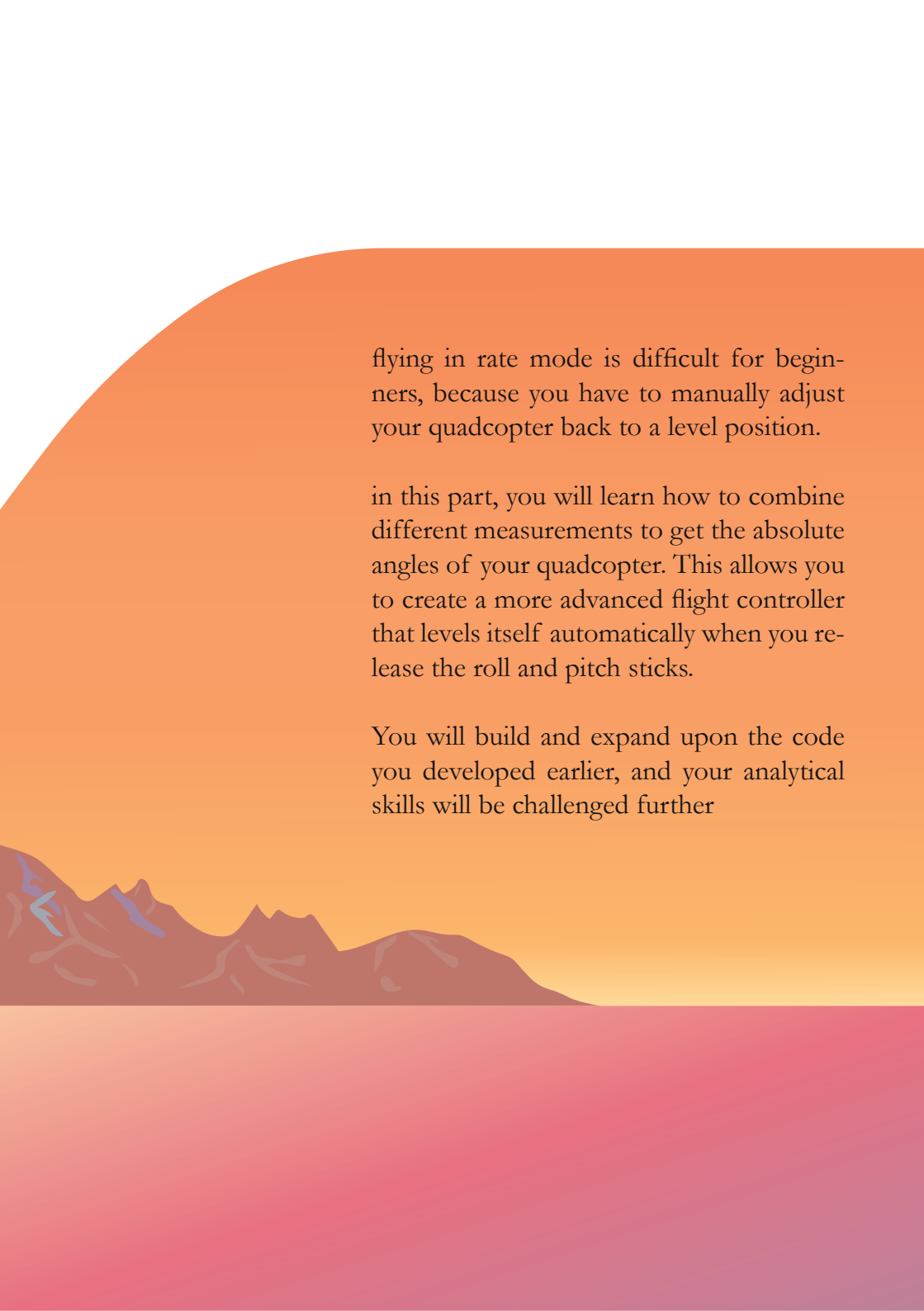
When you have set the radiotransmitter failsafe, it is time to start flying. After you connect the battery and turn on the slide switch, the red LED should be lighted indicating the ongoing startup process. Wait a couple of seconds without touching your quad (to avoid calibration errors). When the green LED illuminates, you can move the throttle stick slightly upward and each motor will beep four times indicating that you are ready to go. Increase the throttle stick to 30% power such that the motors are turning but the quad is not yet taking off, **then turn off the radiotransmitter to test the failsafe**. If after a second all motors turn off, the test is successful.

You can now start your first flight. Flying in rate control mode is rather difficult, so be sure to fly outside at a large grass field without any people nearby to minimize any damage to the quadcopter or others in the event of a crash. You can play with the PID values to optimize the quadcopter's response to your liking.



Part II: stabilization mode





flying in rate mode is difficult for beginners, because you have to manually adjust your quadcopter back to a level position.

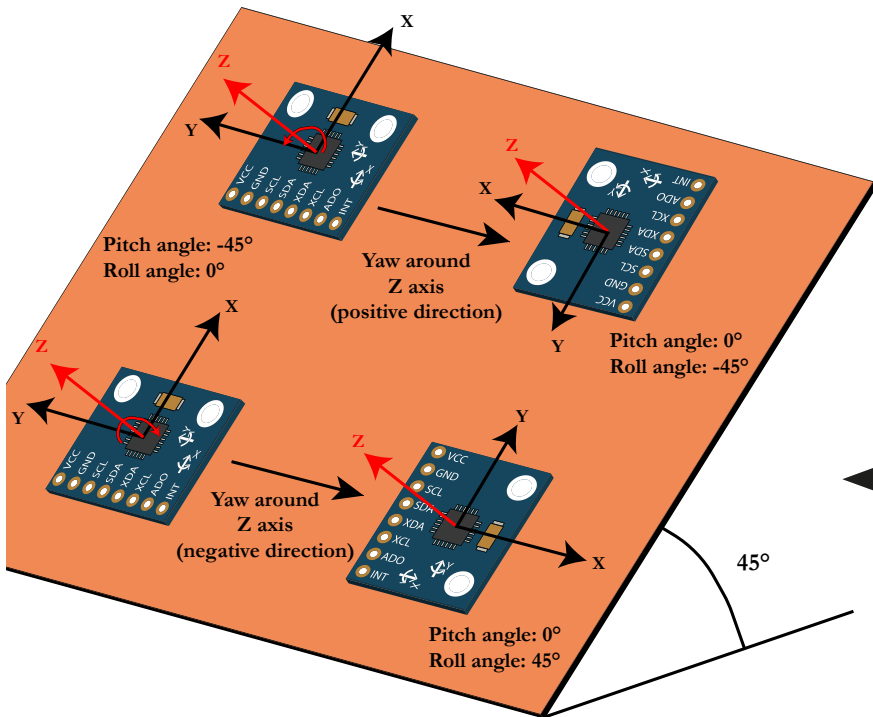
in this part, you will learn how to combine different measurements to get the absolute angles of your quadcopter. This allows you to create a more advanced flight controller that levels itself automatically when you release the roll and pitch sticks.

You will build and expand upon the code you developed earlier, and your analytical skills will be challenged further



Project 14

Measuring angles



Learn to measure angles - twice

At this point the rotation rates of your quadcopter have no more secrets for you. Now it is time to explore a more difficult topic; how can you measure the absolute roll and pitch angles of your quadcopter?

The absolute roll and pitch angles of your quadcopter are a key component for a flight controller that works in stabilize mode; knowing the angles allows you to level the quadcopter exactly and make flying a lot easier. But how can you measure the absolute roll and pitch angles? In this project, you will explore two methods, both with their own (dis)advantages.

1. Integrating the gyro rotation rates

A first and very easy solution to obtain the absolute angles consists of integration of the rotation rates that are measured by the gyroscope. For the pitch, this can be represented with the equation;

$$Angle_{pitch} = \int_0^{k \cdot T_s} Rate_{pitch} \cdot dt$$

With $Rate_{pitch}$ in degrees per second ($^{\circ}/s$), $Angle_{pitch}$ in degrees ($^{\circ}$), T_s the duration of one (0.004 s) iteration and k the number of iterations. Discretization of this integral to use in your code gives the equation:

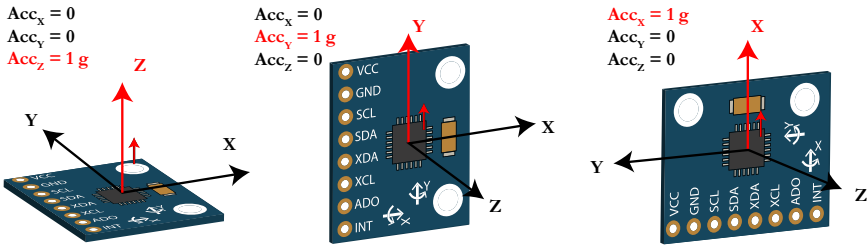
$$Angle_{pitch}(k) = Angle_{pitch}(k - 1) + Rate_{pitch}(k) \cdot T_s$$

If this looks too easy... well that's because unfortunately it is. When the quadcopter is yawing left (or right) around the Z axis without any pitch rotation rate around the Y axis, the pitch angle will nonetheless decrease (or increase) as the direction of the Y axis changes. During this yaw movement, the roll angle will increase (or decrease) as well because the direction of the X axis also changes. Hence even with a zero roll or pitch rotation rate, the roll and pitch angles can change. This phenomenon is visualized in the figure to the left for a pure yaw movement using an inclined plate with a fixed angle of 45° .

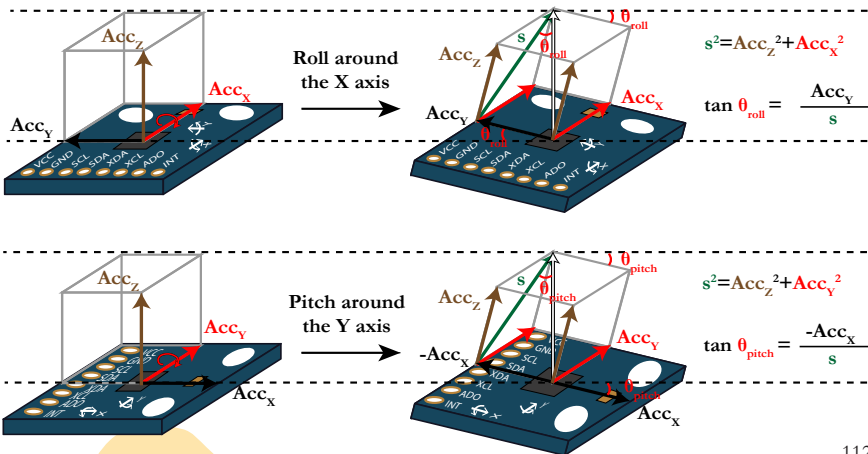
You can integrate the change of the roll and pitch angle with the yaw movements in the equations, but you will fortunately not need to do this for this application; as you will see later, this error is not the only issue with integrating the gyro measurements. For now, let's explore the second method to obtain angles; using an accelerometer.

2. The accelerometer

Your MPU-6050 sensor is not only a gyroscope but also contains an accelerometer. As the name implies, the accelerometer measures the acceleration of the sensor along the X, Y and Z directions. Remember that the gyroscope measures the rotation rates around the X, Y and Z directions, not along. From basic physics, you remember that we experience a gravitational acceleration anywhere on earth and that this gravitational acceleration is equal to the gravitational constant; 1 g or 9,81 m/s². This means that when you let your MPU-6050 sensor lie flat on a table without moving it, the measurement of the acceleration along the Z direction (or Acc_z) is equal to 1 g. The acceleration along the X and Y axes will be zero in this case. Similarly, when you position the sensor such that one of the other axes lies perpendicular to the surface of the table, the corresponding acceleration is equal to 1 g.



Of course, any other direction not along one of the three main axes will result in a nonzero acceleration value for all three directions Acc_x, Acc_y and Acc_z. Through some clever mathematical equations, this accelerometer property will enable you to calculate the exact roll and pitch angles of your quadcopter. Let's assume you roll around the X axis until you reach the angle θ_{roll} . To visualize this transformation, a box bounded by the X, Y and Z directions is sketched on the figure below.



From your basic trigonometry knowledge, you know that the tangent of the angle of a triangle is equal to the length of the opposite side of the triangle divided by the length of the adjacent side of the triangle. In the case of the angle θ_{roll} , the opposite side is equal to Acc_y while the adjacent side is equal to a certain length s :

$$\tan(\theta_{roll}) = \frac{Acc_y}{s}$$

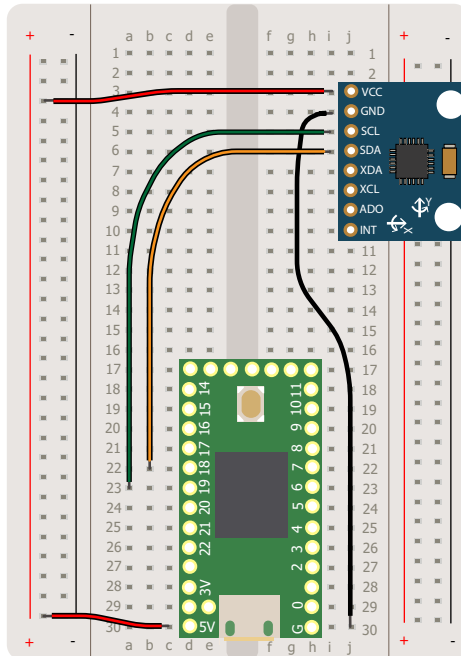
Using the Pythagoras rule on the triangle formed by s , Acc_x and Acc_z , you are able to derive that $s^2 = Acc_x^2 + Acc_z^2$, thus the angle θ_{roll} can be expressed by the equation:

$$\theta_{roll} = \text{atan} \left(\frac{Acc_y}{\sqrt{Acc_x^2 + Acc_z^2}} \right)$$

Through similar reasoning and with the help of the next figure you can express the pitch angle by:

$$\theta_{pitch} = \text{atan} \left(\frac{-Acc_x}{\sqrt{Acc_y^2 + Acc_z^2}} \right)$$

And that's it, you are now able to calculate the roll and pitch angles from the values of your accelerometer! Now transform this into a working code. For this part, you only need your Teensy and MPU-6050; you can choose to test on a breadboard, or directly on your assembled quadcopter.



Coding

Modify the code that you have already developed to read the gyroscope in order to extract the accelerometer measurements of the MPU-6050 as well.

Start with defining the variables that will hold the acceleration values in the X, Y and Z direction, together with the roll and pitch angles.

First configure the accelerometer output. This can be done by choosing the AFS_SEL setting using register 0x1C (see the MPU-6050 documentation). The options for the accelerometer correspond to bits 3 and 4. You will choose a full scale range of ± 8 g, which corresponds to an LSB sensitivity of 4096 LSB/g and a value for the AFS_SEL setting of 2, or a 0 for bit 3 and a 1 for bit 4. This corresponds in turn to the following 8 bit binary representation: 00010000. Converting this to a hexadecimal value gives 0x10.

The values of the accelerometer are located in the registers with hexadecimal numbers 3B to 40. Start writing to address 0x3B to indicate the first register and request 6 bytes from the address of the sensor, 0x68. The accelerometer measurements in LSB are once again spread out over two registers with each 8 bits.

```
1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
```

```
3 float AccX, AccY, AccZ;
4 float AngleRoll, AnglePitch;
```

Define the accelerometer variables

```
5 float LoopTimer;
6 void gyro_signals(void) {
7     Wire.beginTransmission(0x68);
8     Wire.write(0x1A);
9     Wire.write(0x05);
10    Wire.endTransmission();
```

Switch on the low-pass filter (project 4)

```
11    Wire.beginTransmission(0x68);
12    Wire.write(0x1C);
13    Wire.write(0x10);
14    Wire.endTransmission();
```

Configure the accelerometer output

```
15    Wire.beginTransmission(0x68);
16    Wire.write(0x3B);
17    Wire.endTransmission();
18    Wire.requestFrom(0x68,6);
19    int16_t AccXLSB = Wire.read() << 8 |
20    Wire.read();
21    int16_t AccYLSB = Wire.read() << 8 |
22    Wire.read();
23    int16_t AccZLSB = Wire.read() << 8 |
24    Wire.read();
```

Pull the accelerometer measurements from the sensor

```
25    Wire.beginTransmission(0x68);
26    Wire.write(0x1B);
27    Wire.write(0x8);
28    Wire.endTransmission();
29    Wire.beginTransmission(0x68);
30    Wire.write(0x43);
31    Wire.endTransmission();
```

Configure the gyroscope output and pull rotation rate measurements from the sensor (project 4)



To convert the accelerometer measurements from LSB to g, remember that you configured the AFS_SEL setting to an LSB sensitivity of 4096 LSB/g. To get the measurements in g, just divide the measurements in LSB by 4096 LSB/g.

At the start of this project, you learned how to calculate the roll and pitch angles from the accelerometer values. You can use these equations at this point, as long as you take into account that the arctangens calculated by Arduino returns a result in radians, not in degrees. To convert the angles from radians to degrees, just divide the results by $\pi/180$.

Testing and calibration

When you run the code with the MPU-6050 flat on a table without moving and open the serial monitor, you will notice that the acceleration values in the X, Y and Z directions are not exactly 0, 0 and 1 g as they should be but rather:

```
Acceleration X [g]= 0.04 Acceleration Y [g]= -0.02 Acceleration Z [g]= 1.11
Acceleration X [g]= 0.04 Acceleration Y [g]= -0.03 Acceleration Z [g]= 1.11
Acceleration X [g]= 0.03 Acceleration Y [g]= -0.03 Acceleration Z [g]= 1.10
```

It is normal when you do not have the same values as mentioned above, because each sensor is slightly different. Calibration is once again necessary to correct these values. Because the MPU-6050 has to be exactly level when doing the accelerometer calibration, it is recommended to do this beforehand;

- Normally your MPU-6050 lies already flat on the table. The acceleration in the Z direction should be 1 in this case; for the above values for example, this would give a correction value of 0.11.
- For the calibration of the acceleration in the X direction, you have to tilt the MPU-6050 vertically along the X axis. You should now get a value close to 1. Note once again the difference between the value you get and 1 g.
- Now do the same for the acceleration in the Y direction.

```

29     Wire.requestFrom(0x68,6);
30     int16_t GyroX=Wire.read()<<8 | Wire.read();
31     int16_t GyroY=Wire.read()<<8 | Wire.read();
32     int16_t GyroZ=Wire.read()<<8 | Wire.read();
33     RateRoll=(float)GyroX/65.5;
34     RatePitch=(float)GyroY/65.5;
35     RateYaw=(float)GyroZ/65.5;

```

```

36     AccX=(float)AccXLSB/4096;
37     AccY=(float)AccYLSB/4096;
38     AccZ=(float)AccZLSB/4096;

```

Convert the measurements to physical values

```

39     AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
        AccZ))*1/(3.142/180);
40     AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
        AccZ))*1/(3.142/180);
41 }

```

Calculate the absolute angles

```

42 void setup() {
43     Serial.begin(57600);
44     pinMode(13, OUTPUT);
45     digitalWrite(13, HIGH);
46     Wire.setClock(400000);
47     Wire.begin();
48     delay(250);
49     Wire.beginTransmission(0x68);
50     Wire.write(0x6B);
51     Wire.write(0x00);
52     Wire.endTransmission();
53 }

```

Communication with the gyroscope and calibration (project 4 and 5)

```

54 void loop() {
55     gyro_signals();
56     Serial.print("Acceleration X [g]= ");
57     Serial.print(AccX);
58     Serial.print(" Acceleration Y [g]= ");
59     Serial.print(AccY);
60     Serial.print(" Acceleration Z [g]= ");
61     Serial.println(AccZ);
62     delay(50);
63 }

```

Print the accelerometer values



The three calibration values you get should be subtracted from lines 36 to 38 in the code, so insert **your own values** in the **yellow space**. Run the program again and verify that the acceleration in all directions is correct.

Next, verify that the calculated roll and pitch angles are correct. You can easily check this by replacing lines 56 to 61 the code with the lines written on the right. When the MPU-6050 lies level on a table, the value for both angles should be very close to zero if you have done a correct calibration.

Accelerometer trigonometry or gyro integration?

You explored two different methods to calculate the roll and pitch angles; one method integrated the rotation rates coming from the gyroscope, and the second method used trigonometry on accelerometer measurements. It is now time to evaluate the advantages and disadvantages of each method. Most disadvantages come with the gyro integration method;

- Pure integration of the roll and pitch rotation rates does not take into account roll and pitch angle changes when yawing, as you already saw before. This means that the calculated angles will not be fully correct during flight.
- With integration, you add the change in angle to the previous angle for each iteration. Because each measurement has an error, this also means that you will add the errors of each measurements. This causes an ever increasing error as shown on the picture to the right; after three minutes, the angle deviation is already equal to 1°.
- Your integration always starts from an angle equal to zero. If the surface on which the quadcopter sits is not level, the angles will be wrong.

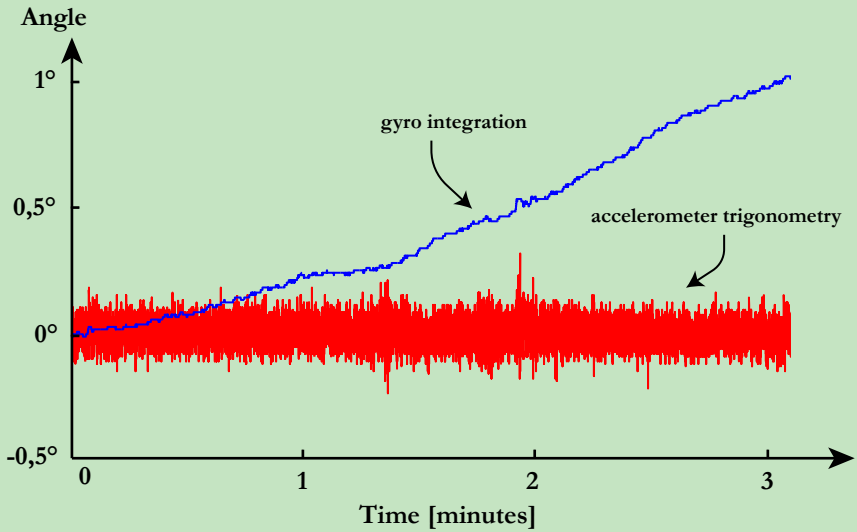
By testing the accelerometer and the resulting angles calculated through trigonometry, you will see that none of the above disadvantages occur for this type of measurement. So why all this trouble, why can't you just use the accelerometer? Well, unfortunately the accelerometer is extremely sensitive to vibrations; it measures acceleration after all. So sensitive, that even with the low pass filter you already configured in the MPU-6050, the resulting angles cannot be handled by your PID controller. This is visible in the picture to the right; the gyro integration over time stays quite continuous, while the accelerometer angles are not continuous at all. This effect will be magnified when the motors are running. The disadvantages of both methods necessitate a different solution, which you will explore with the next project.

```
36 AccX=(float)AccXLSB/4096-0.05;  
37 AccY=(float)AccYLSB/4096+0.01;  
38 AccZ=(float)AccZLSB/4096-0.11;
```

Correct the accelerometer values after calibration

```
54 void loop() {  
55     gyro_signals();  
56     Serial.print("Roll angle [°]= ");  
57     Serial.print(AngleRoll);  
58     Serial.print(" Pitch angle [°]= ");  
59     Serial.println(AnglePitch);  
60     delay(50);  
61 }
```

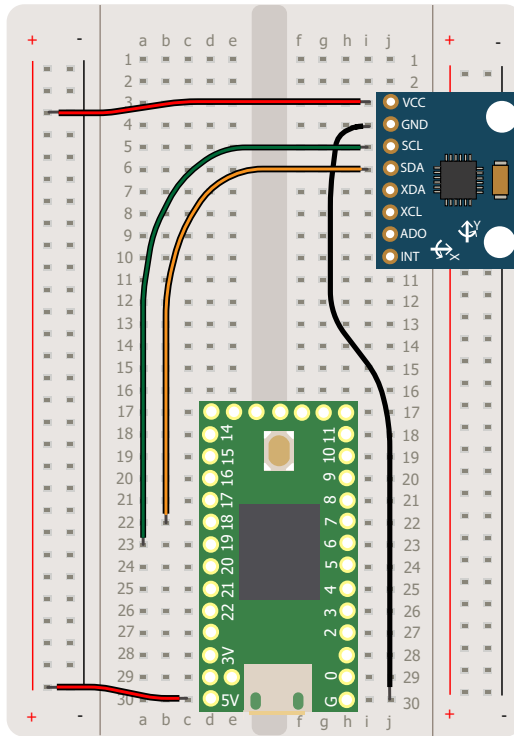
Check the measured roll and pitch angles





Project 15

The Kalman filter - one dimension



A final equation is necessary to update the uncertainty on the new angle prediction, once again by using the Kalman Gain:

$$Uncertainty_{angle}(k) = (1 - Gain_{kalman}) \cdot Uncertainty_{angle}(k)$$

Congratulations, you now learned how to combine the two methods, each with their own uncertainties and errors, to predict the most accurate value for the angle. This approach can be followed for both the roll and pitch angles.

Combine two imperfect measurements

During the previous project you explored two methods of measuring and calculating angles and found out that the disadvantages of both make them unsuitable for a flight controller. However, what if you can combine both measurements to get rid of their individual disadvantages? This is exactly what you are going to do during this project, through an iterative mathematical concept called the **Kalman filter**.

Let's begin by rewriting the equation that you used to transform the rotation rate of the gyroscope to the angle, with T_s the time of one iteration k (0.004 s in our case):

$$Angle_{kalman}(k) = Angle_{kalman}(k - 1) + T_s \cdot Rate(k)$$

Assume now that this calculation gives you a prediction for the angle, but not its final value, because it is prone to a number of errors. Calculate the uncertainty on the prediction of the angle as the sum of the uncertainty on the previous angle prediction (iteration $k-1$) and the uncertainty on the evolution of the angle:

$$Uncertainty_{angle}(k) = Uncertainty_{angle}(k - 1) + T_s^2 \cdot 4^2$$

The uncertainty on the evolution of the angle is estimated as $T_s^2 \cdot 4^2$ because:

- The standard deviation σ of the rotation rate measurement error is $4^\circ/s$, giving a variance σ^2 of $4^2=16$. The rotation rate measurement error is an estimation; it includes the actual imperfection of the sensor itself, but also the fact that you do not take into account the yaw rotation rate in the angle calculation.
- Because the rotation rate is multiplied with T_s ($=0.004$ s) in the equation, this has to be taken into account in the variance calculation as well, using the factor T_s^2 .

In the next step you determine the so-called Kalman gain. This gain weighs your prediction of the angle ($Angle_{kalman}(k)$) as calculated above with the measured angle ($Angle$) using our accelerometer to obtain a new prediction for the angle:

$$Angle_{kalman}(k) = Angle_{kalman}(k) + Gain_{kalman} \cdot (Angle(k) - Angle_{kalman})$$

Now how do you calculate this Kalman gain? The gain is defined as the relative ratio of the uncertainty on the predicted angle to the uncertainty on the measured angle with the accelerometer:

$$Gain_{kalman} = \frac{Uncertainty_{angle}(k)}{Uncertainty_{angle}(k) + 3^2}$$

In the equation, you assume that the standard deviation σ of the accelerometer measurement error is equal to 3° .

General form of the Kalman filter

The Kalman filter that you derived in this project is specifically adapted to predict the roll or pitch angle. This is a one dimensional Kalman filter, as the so-called ‘state’ of the system consists of only one value: the roll (or pitch) angle. This approach can be expanded to multi-dimensional states using vectors and matrices. The ‘general’ form of the Kalman filter is written below and will be used when you are estimating the altitude of the quadcopter further on. For comparison, the values for all vectors and matrices in our current example are also written.

1. Predict the current state of the system:

$$S(k) = F \cdot S(k-1) + G \cdot U(k)$$

S=state vector (Angle_{kalman})
 F=state transition matrix (1)
 G=control matrix (0.004)
 U=input variable (Rate)

2. Calculate the uncertainty of the prediction:

$$P(k) = F \cdot P(k-1) \cdot F^T + Q$$

P=prediction uncertainty vector (Uncertainty_{angle})
 Q=process uncertainty ($T_s^2 \cdot 4^2$)

Coding

Connect your MPU-6050 to your Teensy to test the Kalman filter.

Define the roll and pitch angles coming from our Kalman filter. Your initial guess for the angle values is zero, because the quadcopter will generally take off from a rather level surface. Off course, the surface will never be exactly level, so you take the uncertainty (=variance σ^2) on the initial guess for the angles to be $(2^\circ)^2$. If you take off from a surface that is not level at all, the Kalman filter will use the accelerometer values to quickly correct this initial wrong guess.

Define the output from the Kalman filter; this are two variables: the Kalman prediction for the state (the angle in our case) and the uncertainty on this prediction. Both variables are updated during each iteration.

3. Calculate the Kalman gain from the uncertainties on the predictions and measurements:

$$L(k) = H \cdot P(k) \cdot H^T + R$$
$$K = P(k) \cdot \frac{H^T}{L(k)} = P(k) \cdot H^T \cdot L(k)^{-1}$$

L= Intermediate matrix
K=Kalman gain
H=Observation matrix (=1)
R=Measurement uncertainty ($\Gamma_s^2 \cdot 3^2$)

4. Update the predicted state of the system with the measurement of the state through the Kalman gain:

$$S(k) = S(k) + K \cdot (M(k) - H \cdot S(k))$$

M=measurement vector (Angle)

5. Update the uncertainty of the predicted state:

$$P(k) = (I - K \cdot F) \cdot P(k)$$

I=unity matrix (=1)

```
1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float RateCalibrationRoll, RateCalibrationPitch,
  RateCalibrationYaw;
4 int RateCalibrationNumber;
5 float AccX, AccY, AccZ;
6 float AngleRoll, AnglePitch;
7 uint32_t LoopTimer;

8 float KalmanAngleRoll=0,
  KalmanUncertaintyAngleRoll=2*2;
9 float KalmanAnglePitch=0,
  KalmanUncertaintyAnglePitch=2*2;

10 float Kalman1DOutput[]={0,0};
```

Define the gyroscope and accelerometer variables

Define the predicted angles and the uncertainties

Initialize the output of the filter



Create the function that calculates the predicted angle and uncertainty using the Kalman equations

```

11 void kalman_1d(float KalmanState,
    float KalmanUncertainty, float KalmanInput,
    float KalmanMeasurement) {
12     KalmanState=KalmanState+0.004*KalmanInput;
13     KalmanUncertainty=KalmanUncertainty + 0.004
        * 0.004 * 4 * 4;
14     float KalmanGain=KalmanUncertainty * 1/
        (1*KalmanUncertainty + 3 * 3);
15     KalmanState=KalmanState+KalmanGain * (
        KalmanMeasurement-KalmanState);
16     KalmanUncertainty=(1-KalmanGain) *
        KalmanUncertainty;

```

Kalman filter output

```

17     Kalman1DOutput[0]=KalmanState;
    Kalman1DOutput[1]=KalmanUncertainty;
18 }

```

Read the rotation rates, acceleration and angles from the MPU-6050 (project 14)

```

19 void gyro_signals(void) {
20     Wire.beginTransmission(0x68);
21     Wire.write(0x1A);
22     Wire.write(0x05);
23     Wire.endTransmission();
24     Wire.beginTransmission(0x68);
25     Wire.write(0x1C);
26     Wire.write(0x10);
27     Wire.endTransmission();
28     Wire.beginTransmission(0x68);
29     Wire.write(0x3B);
30     Wire.endTransmission();
31     Wire.requestFrom(0x68,6);
32     int16_t AccXLSB = Wire.read() << 8 |
        Wire.read();
33     int16_t AccYLSB = Wire.read() << 8 |
        Wire.read();
34     int16_t AccZLSB = Wire.read() << 8 |
        Wire.read();
35     Wire.beginTransmission(0x68);
36     Wire.write(0x1B);
37     Wire.write(0x8);
38     Wire.endTransmission();

```

The next step is to create the function for the Kalman filter. Four variables are necessary to initiate this function;

- the Kalman prediction for the previous state (the angle in this case);
- the uncertainty on the Kalman prediction for the previous state;
- the input for the new Kalman prediction of the state (the rotation rate from the gyroscope in this case);
- the measurement that will be compared with the new Kalman prediction of the state (the angles measured by the accelerometer in this case).

You use these four variables to solve the five equations that were explained on the previous pages.

The output of the Kalman filter function consists of a prediction for the state (the angle) and the corresponding uncertainty.

```
39   Wire.beginTransmission(0x68);
40   Wire.write(0x43);
41   Wire.endTransmission();
42   Wire.requestFrom(0x68,6);
43   int16_t GyroX=Wire.read()<<8 | Wire.read();
44   int16_t GyroY=Wire.read()<<8 | Wire.read();
45   int16_t GyroZ=Wire.read()<<8 | Wire.read();
46   RateRoll=(float)GyroX/65.5;
47   RatePitch=(float)GyroY/65.5;
48   RateYaw=(float)GyroZ/65.5;

49   AccX=(float)AccXLSB/4096-0.05;
50   AccY=(float)AccYLSB/4096+0.01;
51   AccZ=(float)AccZLSB/4096-0.11;

52   AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
53   AccZ))*1/(3.142/180);
54   AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
55   AccZ))*1/(3.142/180);
56 }
57 void setup() {
58   Serial.begin(57600);
59   pinMode(13, OUTPUT);
60   digitalWrite(13, HIGH);
61   Wire.setClock(400000);
```

Do not forget to put your own accelerometer calibration values **here** (project 14)



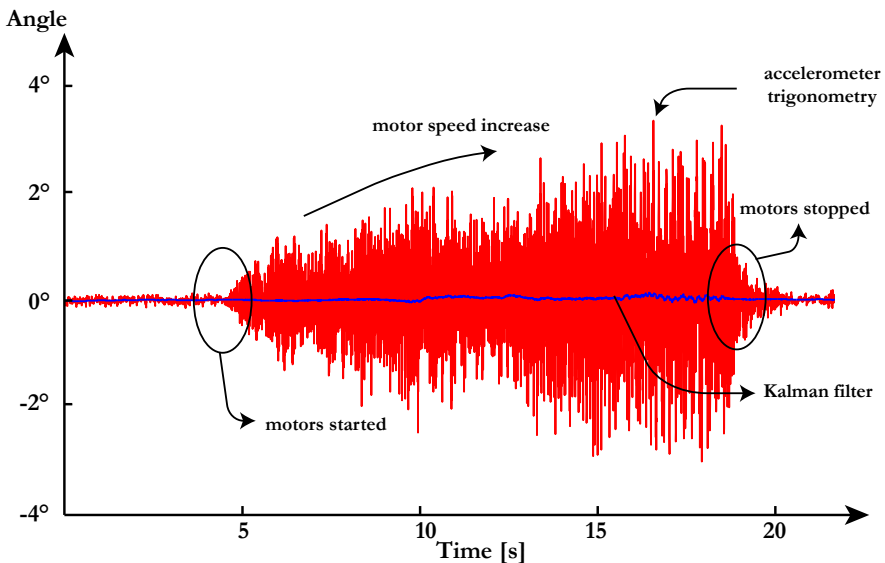
```

60  Wire.begin();
61  delay(250);
62  Wire.beginTransmission(0x68);
63  Wire.write(0x6B);
64  Wire.write(0x00);
65  Wire.endTransmission();
66  for (RateCalibrationNumber=0;
      RateCalibrationNumber<2000;
      RateCalibrationNumber++) {
67      gyro_signals();
68      RateCalibrationRoll+=RateRoll;
69      RateCalibrationPitch+=RatePitch;

```

Communication with the gyroscope and calibration (project 4 and 5)

When the rotation rates from the gyro and the angles from the accelerometer are measured, start the iteration for the Kalman filter. As already seen, the output of the filter will be the Kalman prediction for the roll and pitch angles together with their uncertainties.



```

70         RateCalibrationYaw+=RateYaw;
71         delay(1);
72     }
73     RateCalibrationRoll/=2000;
74     RateCalibrationPitch/=2000;
75     RateCalibrationYaw/=2000;
76     LoopTimer=micros();
77 }
78 void loop() {
79     gyro_signals();
80     RateRoll-=RateCalibrationRoll;           Calculate the rotation
81     RatePitch-=RateCalibrationPitch;        rates
82     RateYaw-=RateCalibrationYaw;
83     kalman_1d(KalmanAngleRoll,              Start the iteration
      KalmanUncertaintyAngleRoll, RateRoll, AngleRoll);    for the Kalman fil-
84     KalmanAngleRoll=Kalman1DOutput[0];      ter with the roll and
85     KalmanUncertaintyAngleRoll=Kalman1DOutput[1];  pitch angles
86     kalman_1d(KalmanAnglePitch,
      KalmanUncertaintyAnglePitch, RatePitch, AnglePitch);
87     KalmanAnglePitch=Kalman1DOutput[0];
88     KalmanUncertaintyAnglePitch=Kalman1DOutput[1];
89
90     Serial.print("Roll Angle [°] ");
91     Serial.print(KalmanAngleRoll);
92     Serial.print(" Pitch Angle [°] ");
93     Serial.println(KalmanAnglePitch);
94     while (micros() - LoopTimer < 4000);
95     LoopTimer=micros();
96 }

```

Testing

The measurement results for the angles coming directly from the accelerometer and the angles as predicted by the Kalman filter are shown in the figure to the left, without any vibrations from the motors and when the motors are started. In both cases the quadcopter stayed stationary on the ground; you can observe the noisiness of the accelerometer values, with angles that vary between plus and minus 3° around the real quadcopter angle. The angle calculation from the Kalman filter on the other hand stays very stable and is therefore more suited as input for your new flight controller.

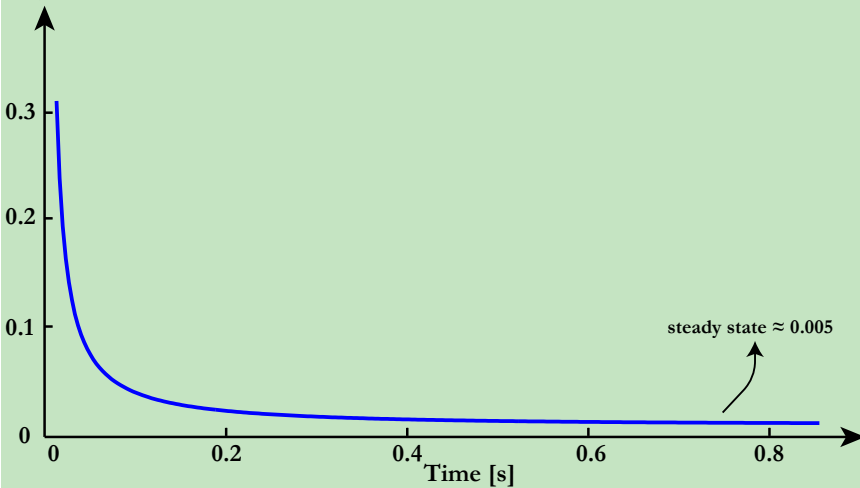


What is the Kalman gain, physically?

A key element of the Kalman filter is the Kalman gain. This gain weighs the importance of the angle prediction, through the gyro integration, with the measured angle using the accelerometer. As the gain is a weighting factor, its value always lies between zero and one. A high Kalman gain gives a large importance to the measurement (e.g. the accelerometer), while a low Kalman gain gives a larger importance to the prediction (e.g. the integration of the rotation rate).

For the angle Kalman filter, the evolution of the gain in time is given in the figure below. As you can see, the Kalman gain is high initially, because of the initial importance of the absolute accelerometer values. But rather quickly, the angle prediction using the integration of the rotation rate becomes more important. Essentially the Kalman filter uses the gyroscope integration prediction most of the time during the flight. The accelerometer pitch angles are used to make sure that the gyroscope integration does not diverge too much from the accelerometer pitch angles, for example due to drift. You now truly have a method to combine the best of both measurements!

Kalman gain
(angle calculation)

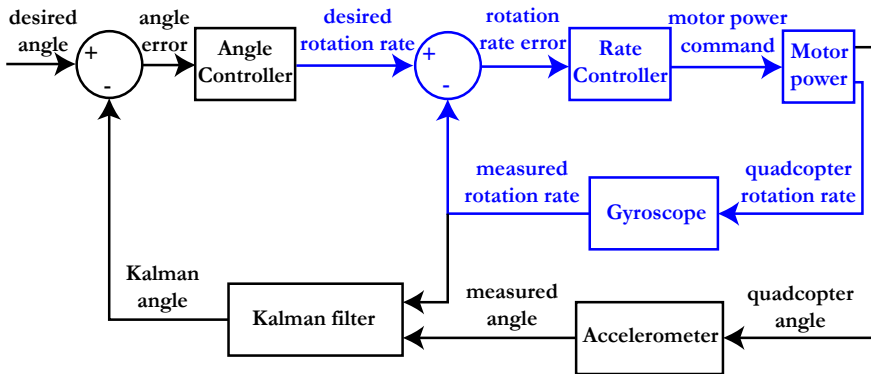






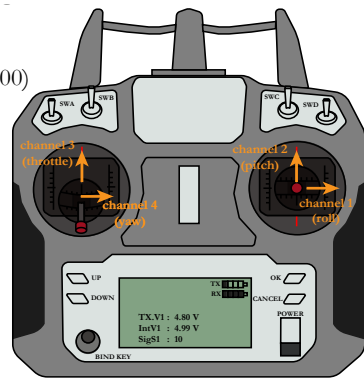
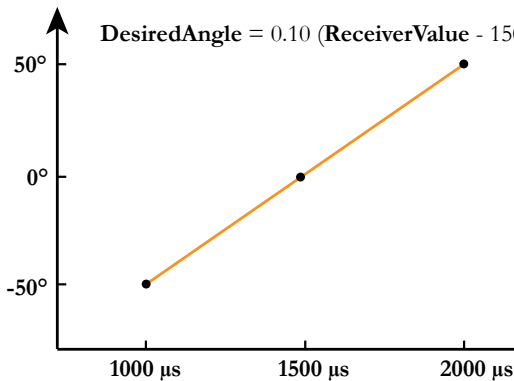
Project 16

The flight controller: stabilize mode



DesiredAngle^{Roll}
DesiredAngle^{Pitch}

$$\text{DesiredAngle} = 0.10 (\text{ReceiverValue} - 1500)$$



ReceiverValue[0] (=channel 1)
ReceiverValue[1] (=channel 2)

Stabilize your quadcopter based on its angles

With your previous flight controller, you stabilized your quadcopter based on its rotation rates measured by the gyroscope. You experienced that this made your quadcopter rather difficult to fly. An easier-to-fly flight controller stabilizes your drone based on its angles. To achieve this stabilization, you will use a so-called cascaded controller.

Flying a quadcopter with a flight controller based on rotation rates is rather difficult, because each time you have to manually adjust the quadcopter back to a level position; releasing the roll and pitch sticks stops the rotation rate because the command sent from the radiotransmitter becomes $0^\circ/\text{s}$, but the flight angle remains the same and does not return to 0° . So if you were flying with a roll angle of 30° at the moment you release the stick, the angle would remain equal to 30° .

You will now implement a controller that stabilizes the quadcopter based on its angles; this will be easier to fly because when you release the roll and pitch stick of the radiotransmitter, the quadcopter will self-level itself to a roll and pitch angle of 0° . Angle control is not necessary for the yaw direction, as you usually do not want the yaw angle to go back to a reference point during flight. To implement roll and pitch angle control, you will keep the PID controller that you used for the rate mode as the so-called inner control loop, and add a second PID controller in front that uses the angles instead of the rotation rates as outer loop. This is a cascaded controller.

The idea for this cascaded controller is illustrated with the figure on the left. You have already programmed the inner loop with the rate PID controller in your first flight controller; the desired rotation rate will not be given by the radiotransmitter values in this case, but by the angle controller through the outer loop. The angle is calculated using the Kalman filter from the gyroscope and accelerometer measurements as learned in the previous project. For this controller, only a P term is necessary; the P values for roll and pitch can be set equal to 2 for good stability & performance.

The last thing that you need to do, is to transform the values sent from the receiver to physical roll and pitch angles, as you did before with the roll, pitch and yaw rotation rates. In this case, you will choose the minimal and maximal values for the desired angles to be -50° and $+50^\circ$; this will be sufficient to achieve high speeds with your quadcopter. Since you already programmed an angle Kalman filter and a PID controller has no more secrets to you, you are now ready to develop a stabilize-mode flight controller.

Coding

```

1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float RateCalibrationRoll, RateCalibrationPitch,
   RateCalibrationYaw;
4 int RateCalibrationNumber;
5 #include <PulsePosition.h>
6 PulsePositionInput ReceiverInput(RISING);
7 float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0};
8 int ChannelNumber=0;
9 float Voltage, Current, BatteryRemaining, BatteryAtStart;
10 float CurrentConsumed=0;
11 float BatteryDefault=1300;
12 uint32_t LoopTimer;
13 float DesiredRateRoll, DesiredRatePitch,
   DesiredRateYaw;
14 float ErrorRateRoll, ErrorRatePitch, ErrorRateYaw;
15 float InputRoll, InputThrottle, InputPitch, InputYaw;
16 float PrevErrorRateRoll, PrevErrorRatePitch,
   PrevErrorRateYaw;
17 float PrevItermRateRoll, PrevItermRatePitch,
   PrevItermRateYaw;
18 float PIDReturn[]={0, 0, 0};
19 float PRateRoll=0.6; float PRatePitch=PRateRoll;
   float PRateYaw=2;
20 float IRateRoll=3.5; float IRatePitch=IRateRoll;
   float IRateYaw=12;
21 float DRateRoll=0.03; float DRatePitch=DRateRoll;
   float DRateYaw=0;
22 float MotorInput1, MotorInput2, MotorInput3,
   MotorInput4;
23 float AccX, AccY, AccZ;
24 float AngleRoll, AnglePitch;

25 float KalmanAngleRoll=0,
   KalmanUncertaintyAngleRoll=2*2;
26 float KalmanAnglePitch=0,
   KalmanUncertaintyAnglePitch=2*2;
27 float Kalman1DOutput[]={0,0};

```

Initialize the same variables that you already needed for rate mode (project 12)

Initialize the accelerometer variables (project 14)

Define the Kalman variables (project 15)

```

28 float DesiredAngleRoll, DesiredAnglePitch;
29 float ErrorAngleRoll, ErrorAnglePitch;

```

Define the desired roll and pitch angles and corresponding errors for the outer loop PID controller

```

30 float PrevErrorAngleRoll, PrevErrorAnglePitch;
31 float PrevItermAngleRoll, PrevItermAnglePitch;

32 float PAngleRoll=2; float PAnglePitch=PAngleRoll;
33 float IAngleRoll=0; float IAnglePitch=IAngleRoll;
34 float DAngleRoll=0; float DAnglePitch=DAngleRoll;

```

Define the values necessary for the outer loop PID controller, including the P, I and D parameters

```

35 void kalman_1d(float KalmanState,
    float KalmanUncertainty, float KalmanInput,
    float KalmanMeasurement) {
36     KalmanState=KalmanState+0.004*KalmanInput;
37     KalmanUncertainty=KalmanUncertainty + 0.004
        * 0.004 * 4 * 4;
94     float KalmanGain=KalmanUncertainty * 1/
        (1*KalmanUncertainty + 3 * 3);
38     KalmanState=KalmanState+KalmanGain * (
        KalmanMeasurement-KalmanState);
39     KalmanUncertainty=(1-KalmanGain) *
        KalmanUncertainty;
40     Kalman1DOutput[0]=KalmanState;
        Kalman1DOutput[1]=KalmanUncertainty;
41 }

```

The Kalman filter function (project 15)

```

42 void battery_voltage(void) {
43     Voltage=(float)analogRead(15)/62;
44     Current=(float)analogRead(21)*0.089;
45 }
46 void read_receiver(void) {
47     ChannelNumber = ReceiverInput.available();
48     if (ChannelNumber > 0) {
49         for (int i=1; i<=ChannelNumber;i++){
50             ReceiverValue[i-1]=ReceiverInput.read(i);
51         }
52     }
53 }

```

Battery voltage function (project 9)

Receiver function (project 7)



```

54 void gyro_signals(void) {
55     Wire.beginTransmission(0x68);
56     Wire.write(0x1A);
57     Wire.write(0x05);
58     Wire.endTransmission();
59     Wire.beginTransmission(0x68);
60     Wire.write(0x1C);
61     Wire.write(0x10);
62     Wire.endTransmission();
63     Wire.beginTransmission(0x68);
64     Wire.write(0x3B);
65     Wire.endTransmission();
66     Wire.requestFrom(0x68,6);
67     int16_t AccXLSB = Wire.read() << 8 |
        Wire.read();
68     int16_t AccYLSB = Wire.read() << 8 |
        Wire.read();
69     int16_t AccZLSB = Wire.read() << 8 |
        Wire.read();
70     Wire.beginTransmission(0x68);
71     Wire.write(0x1B);
72     Wire.write(0x8);
73     Wire.endTransmission();
74     Wire.beginTransmission(0x68);
75     Wire.write(0x43);
76     Wire.endTransmission();
77     Wire.requestFrom(0x68,6);
78     int16_t GyroX=Wire.read()<<8 | Wire.read();
79     int16_t GyroY=Wire.read()<<8 | Wire.read();
80     int16_t GyroZ=Wire.read()<<8 | Wire.read();
81     RateRoll=(float)GyroX/65.5;
82     RatePitch=(float)GyroY/65.5;
83     RateYaw=(float)GyroZ/65.5;

84     AccX=(float)AccXLSB/4096-0.05;
85     AccY=(float)AccYLSB/4096+0.01;
86     AccZ=(float)AccZLSB/4096-0.11;

87     AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
        AccZ))*1/(3.142/180);

```

Gyro and accelerometer function (project 14)

Do not forget to put your own accelerometer calibration values [here](#) (project 14)

```

88     AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
      AccZ))*1/(3.142/180);
89 }
90 void pid_equation(float Error, float P , float I, float D,      PID function (project 12)
      float PrevError, float PrevIterm) {
91     float Pterm=P*Error;
92     float Iterm=PrevIterm+I*(Error+
      PrevError)*0.004/2;
93     if (Iterm > 400) Iterm=400;
94     else if (Iterm <-400) Iterm=-400;
95     float Dterm=D*(Error-PrevError)/0.004;
96     float PIDOutput= Pterm+Iterm+Dterm;
97     if (PIDOutput>400) PIDOutput=400;
98     else if (PIDOutput <-400) PIDOutput=-400;
99     PIDReturn[0]=PIDOutput;
100    PIDReturn[1]=Error;
101    PIDReturn[2]=Iterm;
102 }
103 void reset_pid(void) {
104     PrevErrorRateRoll=0; PrevErrorRatePitch=0;      PID reset function
      PrevErrorRateYaw=0;      (project 12)
105     PrevItermRateRoll=0; PrevItermRatePitch=0;
      PrevItermRateYaw=0;
106     PrevErrorAngleRoll=0; PrevErrorAnglePitch=0;
107     PrevItermAngleRoll=0; PrevItermAnglePitch=0;
108 }
109 void setup() {
110     pinMode(5, OUTPUT);
111     digitalWrite(5, HIGH);
112     pinMode(13, OUTPUT);
113     digitalWrite(13, HIGH);
114     Wire.setClock(400000);
115     Wire.begin();
116     delay(250);
117     Wire.beginTransmission(0x68);
118     Wire.write(0x6B);
119     Wire.write(0x00);
120     Wire.endTransmission();

```

Reset the PID error and integral values for the outer PID loop as well

Visualize the setup phase using the red LED

Communication with the gyroscope and calibration (project 4 and 5)



```

121  for (RateCalibrationNumber=0;
      RateCalibrationNumber<2000;
      RateCalibrationNumber++) {
122      gyro_signals();
123      RateCalibrationRoll+=RateRoll;
124      RateCalibrationPitch+=RatePitch;
125      RateCalibrationYaw+=RateYaw;
126      delay(1);
127  }
128  RateCalibrationRoll/=2000;
129  RateCalibrationPitch/=2000;
130  RateCalibrationYaw/=2000;

131  analogWriteFrequency(1, 250);
132  analogWriteFrequency(2, 250);
133  analogWriteFrequency(3, 250);
134  analogWriteFrequency(4, 250);
135  analogWriteResolution(12);

136  pinMode(6, OUTPUT);
137  digitalWrite(6, HIGH);
138  battery_voltage();
139  if (Voltage > 8.3) { digitalWrite(5, LOW);
140      BatteryAtStart=BatteryDefault; }
141  else if (Voltage < 7.5) {
142      BatteryAtStart=30/100*BatteryDefault; }
143  else { digitalWrite(5, LOW);
144      BatteryAtStart=(82*Voltage-580)/100*
      BatteryDefault; }

145  ReceiverInput.begin(14);
146  while (ReceiverValue[2] < 1020 ||
      ReceiverValue[2] > 1050) {
147      read_receiver();
148      delay(4);
149  }
150  LoopTimer=micros();
151 }

```

Set the PWM frequency to 250 Hz and the resolution to 12 bit for all motors (project 8)

Show the end of the setup process and determine the initial battery voltage percentage (project 9)

SAFETY RELATED LINES: Avoid accidental lift off after the setup process (project 12)

<pre> 152 void loop() { 153 gyro_signals(); 154 RateRoll-=RateCalibrationRoll; 155 RatePitch-=RateCalibrationPitch; 156 RateYaw-=RateCalibrationYaw; 157 kalman_1d(KalmanAngleRoll, 158 KalmanUncertaintyAngleRoll, RateRoll, AngleRoll); 159 KalmanAngleRoll=Kalman1DOutput[0]; 160 KalmanUncertaintyAngleRoll=Kalman1DOutput[1]; 161 kalman_1d(KalmanAnglePitch, 162 KalmanUncertaintyAnglePitch, RatePitch, AnglePitch); 163 KalmanAnglePitch=Kalman1DOutput[0]; 164 KalmanUncertaintyAnglePitch=Kalman1DOutput[1]; 165 read_receiver(); 166 DesiredAngleRoll=0.10*(ReceiverValue[0]-1500); 167 DesiredAnglePitch=0.10*(ReceiverValue[1]-1500); 168 InputThrottle=ReceiverValue[2]; 169 DesiredRateYaw=0.15*(ReceiverValue[3]-1500); 170 ErrorAngleRoll=DesiredAngleRoll- 171 KalmanAngleRoll; 172 ErrorAnglePitch=DesiredAnglePitch- 173 KalmanAnglePitch; 174 pid_equation(ErrorAngleRoll, PAngleRoll, 175 IAngleRoll, DAngleRoll, PrevErrorAngleRoll, 176 PrevItermAngleRoll); 177 DesiredRateRoll=PIDReturn[0]; 178 PrevErrorAngleRoll=PIDReturn[1]; 179 PrevItermAngleRoll=PIDReturn[2]; 180 pid_equation(ErrorAnglePitch, PAnglePitch, 181 IAnglePitch, DAnglePitch, PrevErrorAnglePitch, 182 PrevItermAnglePitch); 183 DesiredRatePitch=PIDReturn[0]; 184 PrevErrorAnglePitch=PIDReturn[1]; 185 PrevItermAnglePitch=PIDReturn[2]; </pre>	<p>Measure the rotation rates and subtract the calibration values (project 5)</p> <p>Calculate the roll and pitch angles through the Kalman filter (project 15)</p> <p>Calculate the desired angles from the receiver values</p> <p>Calculate the difference between the desired and the actual roll and pitch angles</p> <p>Calculate the desired roll and pitch angles through the outer loop PID controller</p>
--	--



172	ErrorRateRoll=DesiredRateRoll-RateRoll;	Calculate the difference between the desired and the actual roll, pitch and yaw rotation rates. Use these for the PID controller of the inner loop (project 12)
173	ErrorRatePitch=DesiredRatePitch-RatePitch;	
174	ErrorRateYaw=DesiredRateYaw-RateYaw;	
175	pid_equation(ErrorRateRoll, PRateRoll, IRateRoll, DRateRoll, PrevErrorRateRoll, PrevItermRateRoll);	
176	InputRoll=PIDReturn[0]; PrevErrorRateRoll=PIDReturn[1]; PrevItermRateRoll=PIDReturn[2];	
177	pid_equation(ErrorRatePitch, PRatePitch, IRatePitch, DRatePitch, PrevErrorRatePitch, PrevItermRatePitch);	
178	InputPitch=PIDReturn[0]; PrevErrorRatePitch=PIDReturn[1]; PrevItermRatePitch=PIDReturn[2];	
179	pid_equation(ErrorRateYaw, PRateYaw, IRateYaw, DRateYaw, PrevErrorRateYaw, PrevItermRateYaw);	
180	InputYaw=PIDReturn[0]; PrevErrorRateYaw=PIDReturn[1]; PrevItermRateYaw=PIDReturn[2];	
181	if (InputThrottle > 1800) InputThrottle = 1800;	
173	MotorInput1= 1.024*(InputThrottle-InputRoll -InputPitch-InputYaw);	Use the quadcopter dynamics equations (project 11)
174	MotorInput2= 1.024*(InputThrottle-InputRoll +InputPitch+InputYaw);	
175	MotorInput3= 1.024*(InputThrottle+InputRoll +InputPitch-InputYaw);	
176	MotorInput4= 1.024*(InputThrottle+InputRoll -InputPitch+InputYaw);	
182	if (MotorInput1 > 2000)MotorInput1 = 1999;	Limit the maximal power commands sent to the motors (project 12)
183	if (MotorInput2 > 2000)MotorInput2 = 1999;	
184	if (MotorInput3 > 2000)MotorInput3 = 1999;	
185	if (MotorInput4 > 2000)MotorInput4 = 1999;	

```

186 int ThrottleIdle=1180;
187 if (MotorInput1 < ThrottleIdle) MotorInput1 =
    ThrottleIdle;
188 if (MotorInput2 < ThrottleIdle) MotorInput2 =
    ThrottleIdle;
189 if (MotorInput3 < ThrottleIdle) MotorInput3 =
    ThrottleIdle;
190 if (MotorInput4 < ThrottleIdle) MotorInput4 =
    ThrottleIdle;

```

Keep the quadcopter motors running at minimally 18% power during flight

```

191 int ThrottleCutOff=1000;
192 if (ReceiverValue[2]<1050) {
193     MotorInput1=ThrottleCutOff;
194     MotorInput2=ThrottleCutOff;
195     MotorInput3=ThrottleCutOff;
196     MotorInput4=ThrottleCutOff;
197     reset_pid();
198 }
199 analogWrite(1,MotorInput1);
200 analogWrite(2,MotorInput2);
201 analogWrite(3,MotorInput3);
202 analogWrite(4,MotorInput4);

```

SAFETY RELATED LINES: make sure you are able to turn off the motors

Sent the commands to the motors

```

203 battery_voltage();
204 CurrentConsumed=Current*1000*0.004/3600+
    CurrentConsumed;
205 BatteryRemaining=(BatteryAtStart-
    CurrentConsumed)/BatteryDefault*100;
206 if (BatteryRemaining<=30) digitalWrite(5, HIGH);
207 else digitalWrite(5, LOW);

```

Keep track of battery level (project 9)

```

208 while (micros() - LoopTimer < 4000);
209 LoopTimer=micros();
210 }

```

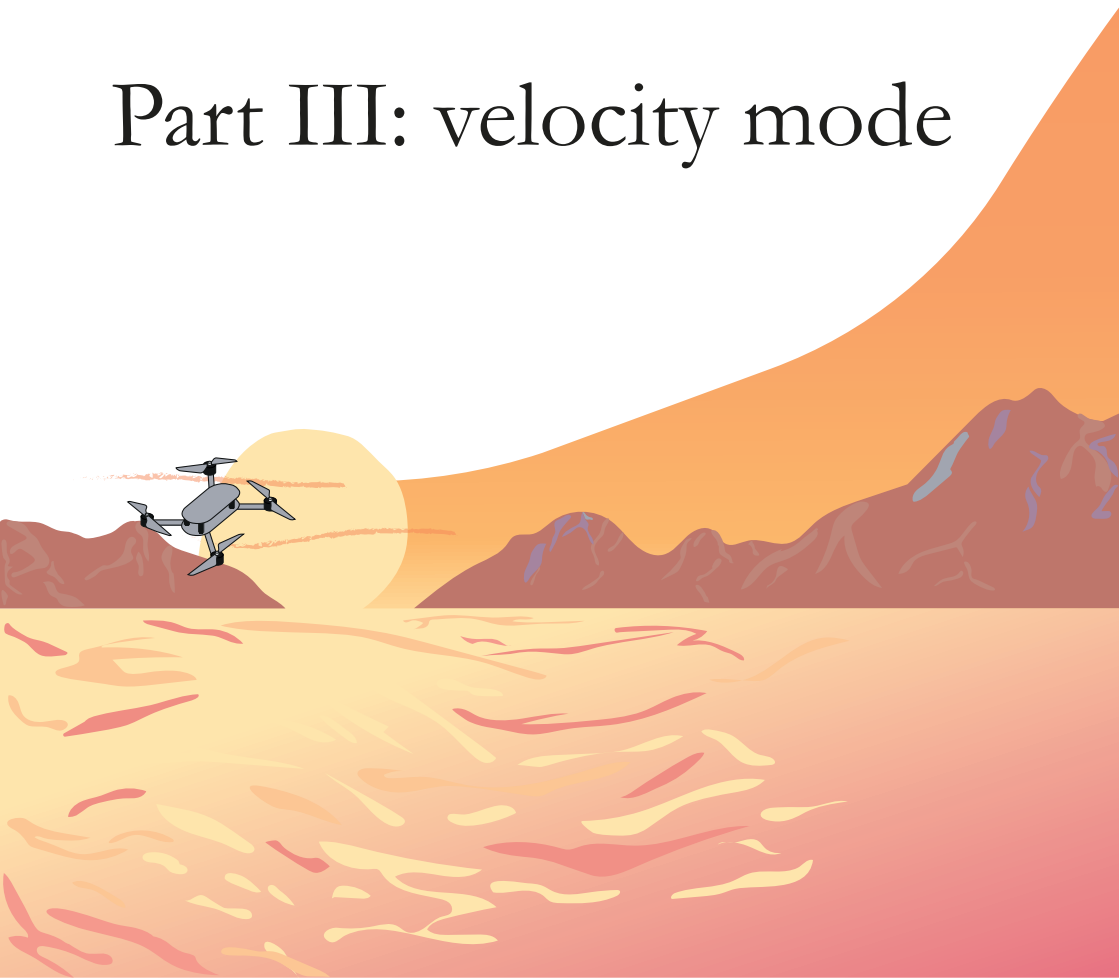
Finish the 250 Hz control loop

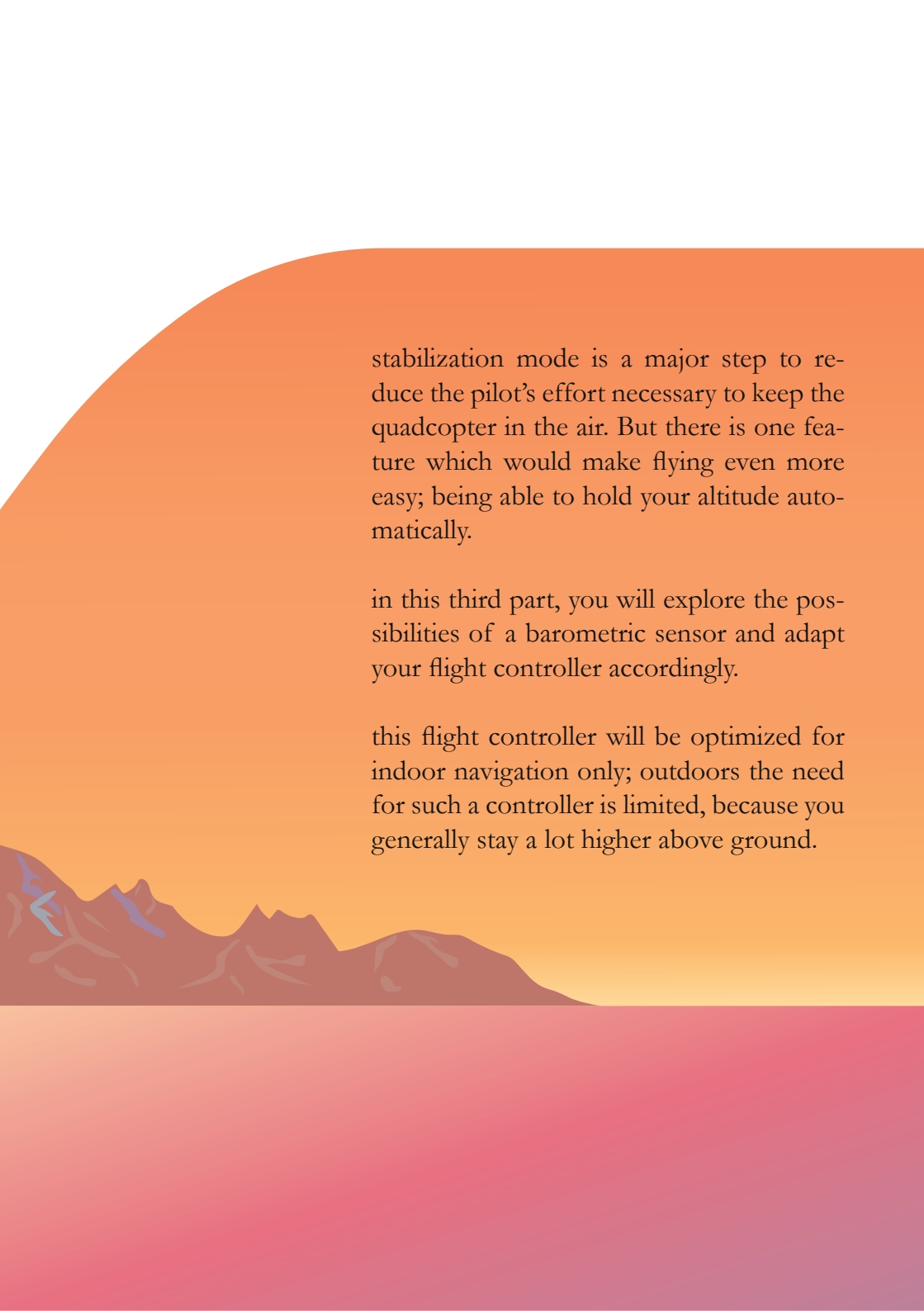
Start-up and flying your quadcopter

To start and fly your quadcopter with the new stabilize-mode controller, follow the same steps as you did with your rate-mode controller. You should notice that flying the quadcopter with your new flight controller is much easier than with the old one. To further simplify flying, a third and final flight controller will be developed in the next part to give you a better control over the altitude of the quadcopter.



Part III: velocity mode





stabilization mode is a major step to reduce the pilot's effort necessary to keep the quadcopter in the air. But there is one feature which would make flying even more easy; being able to hold your altitude automatically.

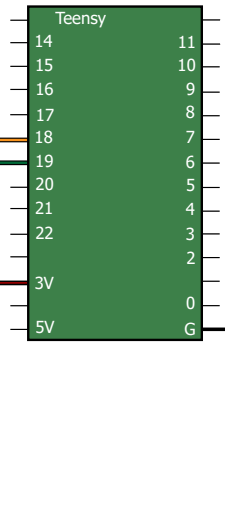
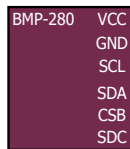
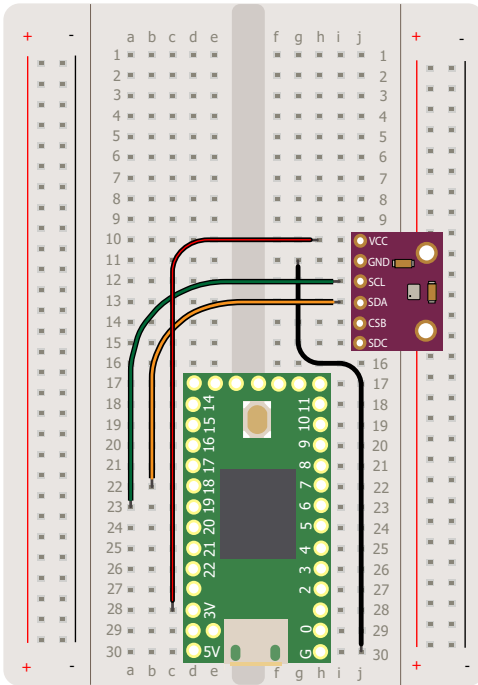
in this third part, you will explore the possibilities of a barometric sensor and adapt your flight controller accordingly.

this flight controller will be optimized for indoor navigation only; outdoors the need for such a controller is limited, because you generally stay a lot higher above ground.



Project 17

Measuring altitude



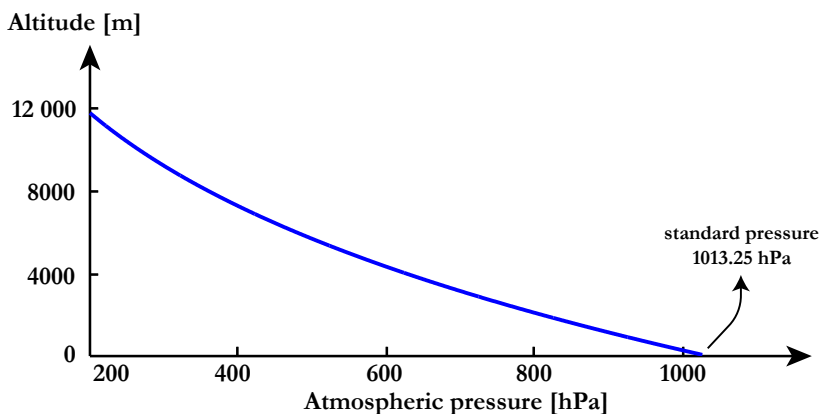
Use a barometer to measure the altitude

To help you holding your quadcopter at a constant altitude, you first need to be able to measure the altitude. You will use a barometric sensor for the altitude measurements: the BMP-280.

A barometric sensor is a sensor that measures the atmospheric pressure. Because the pressure decreases with increasing altitude, the relation between both can be used to measure the altitude. One of the advantages of a barometric sensor is its ability to detect very small pressure changes, making it a suitable measurement for your flight controller. The relation between the atmospheric pressure and the altitude is given through the barometric formula, which assumes in its standard form a constant temperature of 15°C and a standard pressure at sea level of 1013.25 hPa:

$$altitude = 44330 \cdot \left[1 - \left(\frac{pressure}{1013.25} \right)^{\frac{1}{5.255}} \right]$$

Where the altitude is given in meter and the pressure in hPa. Of course, the temperature when flying your drone is not always 15°C and the pressure at sea level also differs from 1013.25 hPa, depending on the weather. However, since you are only interested in the relative change of altitude between startup and a certain position, both the actual temperature and pressure at sea level does not matter for your flight controller. The relation between the altitude and pressure as given in the equation is plotted below.



Now you understand this essential piece of theory, let's connect the BMP-280 pressure sensor to your Teensy. Normally you already installed the sensor on your quadcopter: you can either choose to test directly on the printed circuit board of your quadcopter, or you can connect the sensor separately using your breadboard.

As with your MPU-6050, the SCL and SDA pins are connected to pins 19 and 18 of the Teensy respectively to be able to communicate through the I2C protocol. Because the BMP-280 sensor has a different address than the MPU-6050 sensor, you can use the same Teensy pins for communication with both sensors. Connect the ground of the sensor to the ground of the sensor. **Be very careful to connect the VCC pin of the sensor to the 3V output on your Teensy, not the 5V output!** The BMP-280 sensor is only 3V tolerant so powering it with 5V might damage it. When you're all wired up, let's start programming.

Coding

Each individual sensor is calibrated beforehand by the manufacturer. The calibration values are stored on the sensor's memory in the form of twelve trimming parameters; three for the temperature and nine for the pressure. They are all 16-bit signed or unsigned integers (see datasheet BMP-280). The names of the variables used in this part of the code correspond to the names in the datasheet of the BMP-280 sensor.

Define the altitude measured by the barometer as a global variable, together with the altitude at startup. To have a steady value for the altitude at startup, the average of a large integer number of altitude readouts will be taken (RateCalibrationNumber).

The I2C address for the BMP-280 is 0x76. The pressure and temperature data is read by starting a burst read from the 6 registers 0xF7 to 0xFC; the measurement of the raw temperature and pressure is spread out over three registers each. Request 6 bytes to read the registers; the data comes in unsigned 32 bit format.

The three registers for the temperature and three for the pressure are combined to form the raw, uncompensated and uncalibrated pressure (adc_P) and temperature (adc_T). The msb register contains bits 19 to 12, the LSB register contains bits 11 to 4 and the xlsb register contains bits 3 to 0 of the raw measurements.


```
1 #include <Wire.h>
```

```
2 uint16_t dig_T1, dig_P1;  
3 int16_t dig_T2, dig_T3, dig_P2, dig_P3, dig_P4, dig_P5;  
4 int16_t dig_P6, dig_P7, dig_P8, dig_P9;
```

Define the pressure sensor calibration values

```
5 float AltitudeBarometer, AltitudeBarometerStartUp;  
6 int RateCalibrationNumber;
```

Define the altitude variables

```
7 void barometer_signals(void){
```

```
8     Wire.beginTransmission(0x76);  
9     Wire.write(0xF7);  
10    Wire.endTransmission();  
11    Wire.requestFrom(0x76,6);  
12    uint32_t press_msb = Wire.read();  
13    uint32_t press_lsb = Wire.read();  
14    uint32_t press_xlsb = Wire.read();  
15    uint32_t temp_msb = Wire.read();  
16    uint32_t temp_lsb = Wire.read();  
17    uint32_t temp_xlsb = Wire.read();
```

Make connection with the pressure sensor and read the raw uncombined pressure and temperature measurements

```
18    unsigned long int adc_P = (press_msb << 12) | (  
    press_lsb << 4) | (press_xlsb >>4);  
19    unsigned long int adc_T = (temp_msb << 12) | (  
    temp_lsb << 4) | (temp_xlsb >>4);
```

Construct the raw temperature and pressure measurements



To calculate the compensated and calibrated pressure, first the fine resolution temperature value t_{fine} needs to be determined from the raw temperature values and the trimming parameters. These calculations are entirely given by the manufacturer in the datasheet of the BMP-280 and are therefore not further explained here.

The compensated and calibrated pressure p (in Pa) is calculated with these lines from the raw pressure values and the trimming parameters. Once again, these lines are entirely given by the manufacturer in the datasheet of the BMP-280 and are therefore not further explained here.

Convert the pressure in Pa to the pressure in hPa and calculate the altitude from the standard pressure to the barometric formula. Multiply by 100 to convert from meter to centimetre. This marks the end of the barometric function; continue with the setup part of the code.



```

20 signed long int var1, var2;
21 var1 = (((adc_T >> 3) - ((signed long int)dig_T1
    <<1))) * ((signed long int)dig_T2) >> 11;
22 var2 = (((adc_T >> 4) - ((signed long int)dig_T1
    )) * ((adc_T >> 4) - ((signed long int)dig_T1)))
    >> 12) * ((signed long int)dig_T3) >> 14;
23 signed long int t_fine = var1 + var2;

```

Construct the fine resolution temperature value

```

24 unsigned long int p;
25 var1 = (((signed long int)t_fine >> 1) - (signed
    long int)64000);
26 var2 = (((var1 >> 2) * (var1 >> 2)) >> 11) * ((signed
    long int)dig_P6);
27 var2 = var2 + ((var1 * ((signed long int)dig_P5))
    << 1);
28 var2 = (var2 >> 2) + (((signed long int)dig_P4)
    << 16);
29 var1 = (((dig_P3 * (((var1 >> 2) * (var1 >> 2)) >> 13
    )) >> 3) + (((signed long int)dig_P2) *
    var1 >> 1)) >> 18;
30 var1 = (((32768 + var1) * ((signed long int)dig_P1))
    >> 15);
31 if (var1 == 0) { p = 0; }
32 p = (((unsigned long int) (((signed long int)
    1048576 - adc_P) - (var2 >> 12))) * 3125);
33 if (p < 0x80000000) { p = (p << 1) / ((unsigned
    long int)var1); }
34 else { p = (p / (unsigned long int)var1) * 2; }
35 var1 = (((signed long int)dig_P9) * ((signed long
    int) ((p >> 3) * (p >> 3)) >> 13)) >> 12;
36 var2 = (((signed long int)(p >> 2)) *
    ((signed long int)dig_P8)) >> 13;
37 p = (unsigned long int)((signed long int)p +
    ((var1 + var2 + dig_P7) >> 4));

```

Construct the compensated and calibrated pressure p

```

38 double pressure = (double)p / 100;
39 AltitudeBarometer = 44330 * (1 - pow(pressure
    / 1013.25, 1 / 5.255)) * 100;
40 }

```

Calculate the altitude



In the setup phase, you configure the BMP-280 in such a manner that the sensor is optimized for indoor navigation. The datasheet recommends to set the power mode of the sensor in normal mode, with an oversampling setting for the pressure (`osrs_p`) of x16 and the similar setting for the temperature (`osrs_t`) of x2. According to the datasheet, these settings correspond to `osrs_t[2:0]` bits of 010, `osrs_p[2:0]` bits of 101 and mode [1:0] bits of 11 in the data acquisition control register 0xF4, which has a layout that is visualised in the table:

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
F4	244	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]	
Binary representation		0	1	0	1	0	1	1	1

Converting the resulting binary representation of 01010111 to a hexadecimal value gives an address of 0x57.

The configuration register 0xF5, with a layout shown in the second table, sets the standby time `t_sb[2:0]`, the internal IIR filter `filter[2:0]` and the SPI interface `spi3w_en[0]`. For indoor navigation, the manufacturer recommends to set the IIR filter coefficient to 16 (101). As you do not use the SPI interface and the standby time is only helpful to reduce the power the device needs (which is anyway much smaller than the power the motors need), these remain on their default values (0):

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
F5	245	t_sb[2:0]			filter[2:0]			read only	spi3w_en[0]
Binary representation		0	0	0	1	0	1	0	0

Converting the resulting binary representation of 00010100 to a hexadecimal value gives an address of 0x14.

Import the twelve calibration (e.g. trimming) parameters from the sensor's memory. As they are stored in two's complement, you need to foresee 2x12 or 24 variables. The `i` variable will be used to indicate the trimming parameters in the subsequent while loop during import.

```
41 void setup() {
42     Serial.begin(57600);
43     pinMode(13, OUTPUT);
44     digitalWrite(13, HIGH);
45     Wire.setClock(400000);
46     Wire.begin();
47     delay(250);

48     Wire.beginTransmission(0x76);
49     Wire.write(0xF4);
50     Wire.write(0x57);
51     Wire.endTransmission();
```

Optimize the barometer for indoor navigation

```
52     Wire.beginTransmission(0x76);
53     Wire.write(0xF5);
54     Wire.write(0x14);
55     Wire.endTransmission();
```

Setup the configuration register

```
56     uint8_t data[24], i=0;
```

Import the calibration parameters



The register address of the first trimming parameter is 0x88 according to the data-sheet. Request 24 bytes such that you can pull the information from the 24 registers 0x88 to 0x9E.

Rearrange the trimming parameters, that are split in their two's complement values, such that they are readable in one single parameter. You need to carry out this step for all twelve parameters.

Before you will start your quadcopter, you need the altitude level from which you take off. Take the average of 2000 iterations to get a steady altitude reference level.

Now it is finally time to read the barometer in the loop part. Call the function and subtract the average startup altitude to get the altitude variation in flight.

```

57   Wire.beginTransmission(0x76);
58   Wire.write(0x88);
59   Wire.endTransmission();
60   Wire.requestFrom(0x76,24);
61   while(Wire.available()) {
62       data[i] = Wire.read();
63       i++;
64   }
65   dig_T1 = (data[1] << 8) | data[0];
66   dig_T2 = (data[3] << 8) | data[2];
67   dig_T3 = (data[5] << 8) | data[4];
68   dig_P1 = (data[7] << 8) | data[6];
69   dig_P2 = (data[9] << 8) | data[8];
70   dig_P3 = (data[11] << 8) | data[10];
71   dig_P4 = (data[13] << 8) | data[12];
72   dig_P5 = (data[15] << 8) | data[14];
73   dig_P6 = (data[17] << 8) | data[16];
74   dig_P7 = (data[19] << 8) | data[18];
75   dig_P8 = (data[21] << 8) | data[20];
76   dig_P9 = (data[23] << 8) | data[22]; delay(250);

```

```

77   for (RateCalibrationNumber=0;
        RateCalibrationNumber<2000;
        RateCalibrationNumber++) {
78       barometer_signals();
79       AltitudeBarometerStartUp+=
        AltitudeBarometer;
80       delay(1);
81   }
82   AltitudeBarometerStartUp/=2000;
83 }

```

```

84 void loop() {
85     barometer_signals();
86     AltitudeBarometer-=AltitudeBarometerStartUp;
87     Serial.print("Altitude [cm]: ");
88     Serial.println(AltitudeBarometer);
89     delay(50);
90 }

```

Calculate the altitude reference level

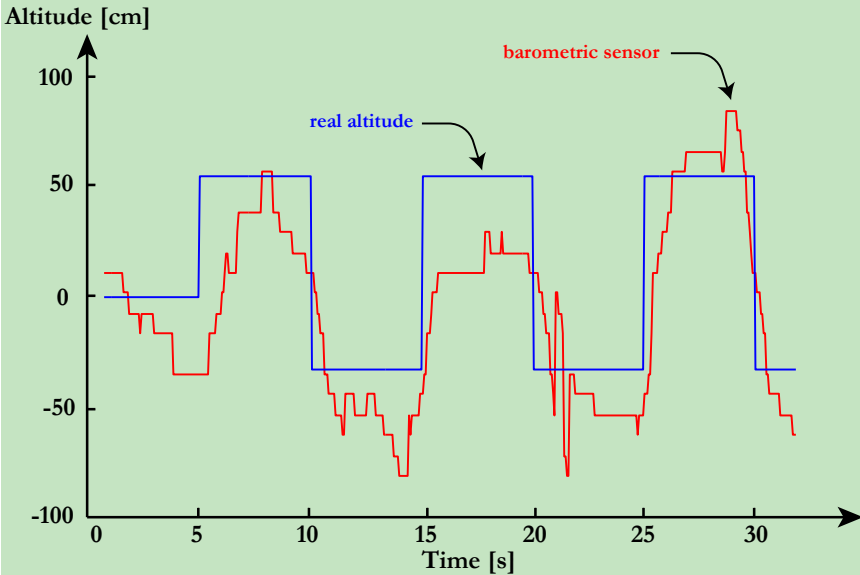
Read the barometer and print the altitudes



Testing the barometric sensor

Test the barometric sensor by moving your breadboard or quadcopter up and down. You will notice that the readings do not change very fast, are not very constant over a longer time and are overall not very accurate. This is illustrated by the figure below for changes in altitude between 50 cm and -30 cm. The reason for this poor performance are rapid pressure changes in the atmosphere, for example due to wind gusts or opening/closing windows when flying indoors.

The ‘jumpy’ performance of the sensor readings also mean that if you only use a barometer for your vertical velocity PID, its performance will not be very good. By now you have probably guessed already how you can solve this issue; use Kalman filter with another, complimentary measurement. In this case, the additional measurement will be the vertical velocity, obtained through the accelerometer.

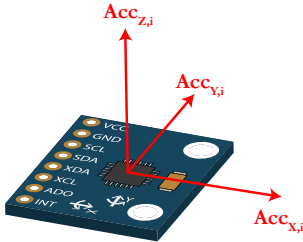






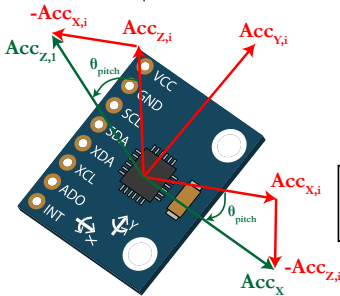
Project 18

Measuring vertical velocity



$$\cos \theta_{\text{pitch}} = \frac{\text{Acc}_{z,i}}{\text{Acc}_{z,1}}$$

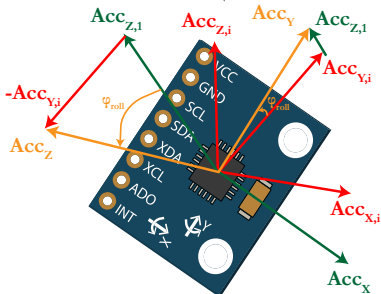
Pitch around the inertial Y axis
with angle θ_{pitch}



$$\sin \theta_{\text{pitch}} = \frac{-\text{Acc}_{z,i}}{\text{Acc}_x}$$

$$\cos \varphi_{\text{roll}} = \frac{\text{Acc}_{z,1}}{\text{Acc}_z}$$

Roll around the new X axis
with angle φ_{roll}



$$\sin \varphi_{\text{roll}} = \frac{\text{Acc}_{z,1}}{\text{Acc}_y}$$

$$\text{Acc}_{z,i} = -\text{Acc}_x \cdot \sin \theta_{\text{pitch}}$$

$$\text{Acc}_{z,i} = \text{Acc}_{z,1} \cdot \cos \theta_{\text{pitch}} = \text{Acc}_y \cdot \sin \varphi_{\text{roll}} \cdot \cos \theta_{\text{pitch}}$$

$$\text{Acc}_{z,i} = \text{Acc}_{z,1} \cdot \cos \theta_{\text{pitch}} = \text{Acc}_z \cdot \cos \varphi_{\text{roll}} \cdot \cos \theta_{\text{pitch}}$$

Measure vertical velocity with an accelerometer

The results from measuring the altitude using your barometer are not sufficiently accurate. Fortunately, your quadcopter hovers not only when the altitude stays the same, but also when the vertical velocity is equal to zero. Measuring this vertical velocity is surprisingly easy using your accelerometer.

Imagine that there exist three inertial directions for the acceleration of your quadcopter: $Acc_{x,i}$, $Acc_{y,i}$ and $Acc_{z,i}$. These are always aligned respectively horizontally and vertically to the surface of the earth and are coloured in red in the figure to the left. If your accelerometer or quadcopter turns, these red inertial axes keep pointing in the same direction. This means that $Acc_{z,i}$ will always be perpendicular to the earth's surface and is hence suitable for measuring the vertical velocity after integration.

When your quadcopter and its accelerometer rolls and pitches, it does not measure the acceleration in its inertial axes anymore, but it measures along the axes that are defined on the accelerometer itself. These are the acceleration vectors Acc_x , Acc_y and Acc_z . All three vectors will have a component that can be related back to the acceleration along the inertial Z axis $Acc_{z,i}$. Using basic trigonometry and the roll φ_{roll} and pitch θ_{pitch} angles, you can calculate the total acceleration along the inertial Z axis $Acc_{z,i}$ by adding the components of the acceleration Acc_x , Acc_y and Acc_z along this axis. This is visualized on the figure to the left. The acceleration in the X direction gives the following component in the inertial Z axis:

$$Acc_{z,i} = -Acc_x \cdot \sin(\theta_{pitch})$$

The acceleration in the Y and Z direction of the accelerometer give the following components in the inertial Z axis:

$$Acc_{z,i} = Acc_{z,1} \cdot \cos(\theta_{pitch}) = Acc_y \cdot \sin(\varphi_{roll}) \cdot \cos(\theta_{pitch})$$

$$Acc_{z,i} = Acc_{z,1} \cdot \cos(\theta_{pitch}) = Acc_z \cdot \cos(\varphi_{roll}) \cdot \cos(\theta_{pitch})$$

This means that the total acceleration in the inertial Z axis can be calculated by adding all separate components:

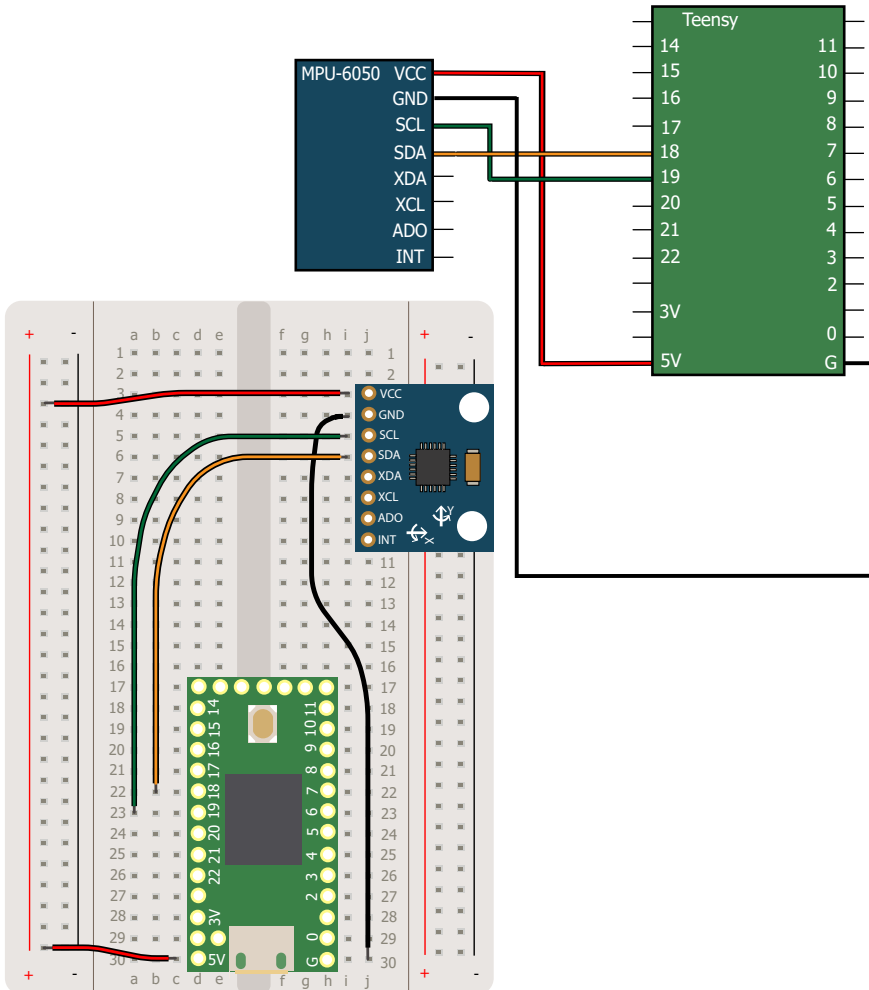
$$Acc_{z,i} = -Acc_x \cdot \sin(\theta_{pitch}) + Acc_y \cdot \sin(\varphi_{roll}) \cdot \cos(\theta_{pitch}) + Acc_z \cdot \cos(\varphi_{roll}) \cdot \cos(\theta_{pitch})$$

Let's test this mathematical representation for the acceleration in the inertial Z axis with the MPU-6050 and your Teensy.

Coding

To run this code, you once again need to connect your accelerometer to your Teensy. You can test this part either on a breadboard or directly on the printed circuit board of your quadcopter.

Define two additional variables for this part: the acceleration in the inertial Z axis $Acc_{Z,i}$ and the velocity in the inertial Z direction, which will be obtained by integrating $Acc_{Z,i}$.



```

1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float AngleRoll, AnglePitch;
4 float AccX, AccY, AccZ;

```

Initialize the gyro and accelerometer variables (project 14)

```

5 float AccZInertial;
6 float VelocityVertical;

```

Define the acceleration and velocity variables

```

7 float LoopTimer;
8 void gyro_signals(void) {
9     Wire.beginTransmission(0x68);
10    Wire.write(0x1A);
11    Wire.write(0x05);
12    Wire.endTransmission();
13    Wire.beginTransmission(0x68);
14    Wire.write(0x1C);
15    Wire.write(0x10);
16    Wire.endTransmission();
17    Wire.beginTransmission(0x68);
18    Wire.write(0x3B);
19    Wire.endTransmission();
20    Wire.requestFrom(0x68,6);
21    int16_t AccXLSB = Wire.read() << 8 |
        Wire.read();
22    int16_t AccYLSB = Wire.read() << 8 |
        Wire.read();
23    int16_t AccZLSB = Wire.read() << 8 |
        Wire.read();
24    Wire.beginTransmission(0x68);
25    Wire.write(0x1B);
26    Wire.write(0x8);
27    Wire.endTransmission();
28    Wire.beginTransmission(0x68);
29    Wire.write(0x43);
30    Wire.endTransmission();
31    Wire.requestFrom(0x68,6);
32    int16_t GyroX=Wire.read()<<8 | Wire.read();
33    int16_t GyroY=Wire.read()<<8 | Wire.read();
34    int16_t GyroZ=Wire.read()<<8 | Wire.read();

```

Define the gyro/accelerometer function (project 14)



```

35   RateRoll=(float)GyroX/65.5;
36   RatePitch=(float)GyroY/65.5;
37   RateYaw=(float)GyroZ/65.5;

38   AccX=(float)AccXLSB/4096-0.05;
39   AccY=(float)AccYLSB/4096+0.01;
40   AccZ=(float)AccZLSB/4096-0.11;

41   AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
    AccZ))*1/(3.142/180);
42   AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
    AccZ))*1/(3.142/180);
43 }

```

Do not forget to put your own accelerometer calibration values [here](#) (project 14)

The acceleration in the inertial Z axis is calculated through the formula you have developed on the first page of this project. Do not forget to transform the roll and pitch angles from degrees to radians by multiplying them with $\pi/180$, as the sines and cosines functions in Arduino only accept radians. The resulting acceleration `AccZInertial` has the same units as `AccX`, `AccY` and `AccZ`, namely the gravitational constant equivalent `g`. Be mindful that for the flight controller, the angles will come from the one-dimensional Kalman filter to eliminate the effect of the quadcopter vibrations.

The acceleration in the inertial Z axis is equal to 1 g even when the accelerometer is stationary because of the gravitation. Therefore you need to subtract 1 g from the acceleration in order to obtain the values for the vertical acceleration of the quadcopter. Because 1 g is equal to 9.81 m/s^2 , you can multiply the acceleration with this constant to get the values in m/s^2 . The unit m/s^2 is too large to be practical so multiply the value with 100 cm/m to get the acceleration in cm/s^2 .

To obtain the vertical velocity, perform an integration by adding the previous velocity to the acceleration multiplied with the length of one loop, 0.004 seconds. You now have the velocity in cm/s .

Print out the value for the vertical velocity to be able to test your code.

```

44 void setup() {
45     Serial.begin(57600);
46     pinMode(13, OUTPUT);
47     digitalWrite(13, HIGH);
48     Wire.setClock(400000);
49     Wire.begin();
50     delay(250);
51     Wire.beginTransmission(0x68);
52     Wire.write(0x6B);
53     Wire.write(0x00);
54     Wire.endTransmission();
55 }
56 void loop() {
57     gyro_signals();
58     AccZInertial=-sin(AnglePitch*(3.142/180))*AccX
+cos(AnglePitch*(3.142/180))*sin(AngleRoll*
(3.142/180))* AccY+cos(AnglePitch*(3.142/180))*
cos(AngleRoll*(3.142/180))*AccZ;
59     AccZInertial=(AccZInertial-1)*9.81*100;
60     VelocityVertical=VelocityVertical
+AccZInertial*0.004;
61     Serial.print("Vertical velocity [cm/s]: ");
62     Serial.println(VelocityVertical);
63 while (micros() - LoopTimer < 4000);
64     LoopTimer=micros();
65 }

```

Communication with the gyroscope and calibration (project 4 and 5)

Calculate the acceleration in the inertial Z axis

Convert the acceleration to cm/s^2

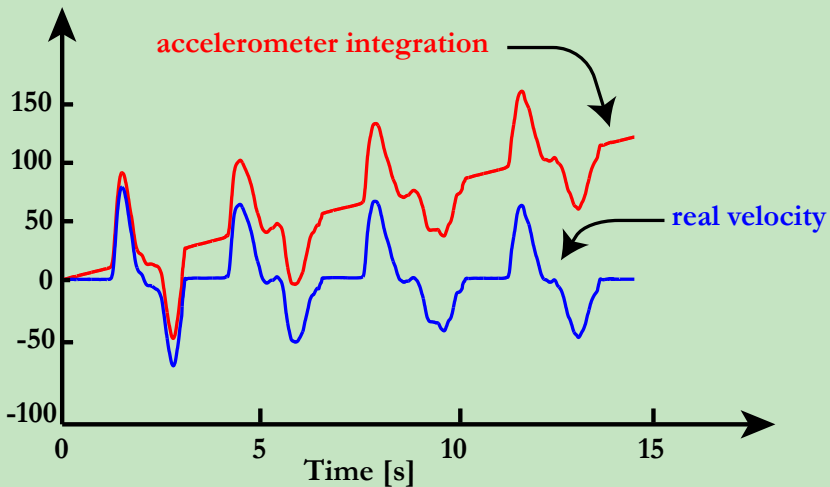
Calculate and print the vertical velocity



Testing the vertical velocity code

When testing the code on the previous pages, you will notice that the calculated vertical velocity changes linearly with time even when the accelerometer/quadcopter is not moving. The rate of change depends on how well you performed your accelerometer calibration as this is once again an example of accumulation of small integration errors. This issue is represented in the figure below, where the vertical velocity calculated through accelerometer integration is visualized, as opposed to the real vertical velocity. It is clear that the error on the vertical velocity becomes so large after a couple of seconds, that this method of measuring and calculating the vertical velocity gives not sufficiently satisfactory results to use in your control system. You will use a second Kalman filter to combine the altitude and vertical velocity measurements in order to obtain an accurate value for the vertical velocity.

Vertical velocity [cm/s]

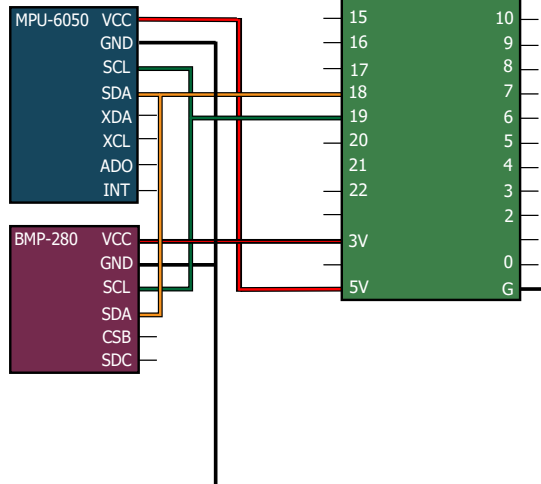
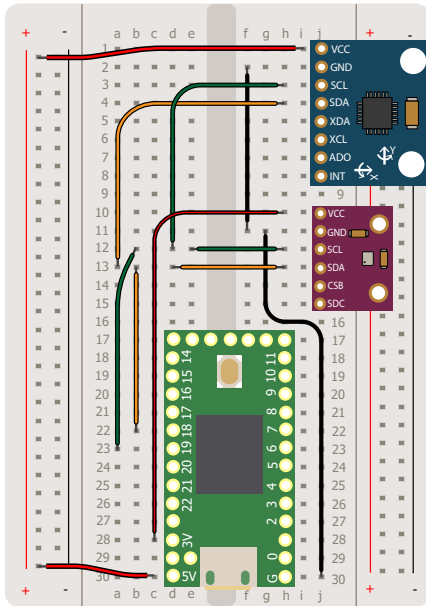






Project 19

The Kalman filter - two dimensions



Determine the vertical velocity accurately

Just as you did when measuring the roll and pitch angles, you will use another Kalman filter to combine the accelerometer and barometer measurements in order to obtain the vertical velocity. Because the state of your system contains two variables - the vertical velocity and the altitude - this Kalman filter will be two dimensional.

The approach you will follow to construct the two dimensional Kalman filter is very similar as the one dimensional case. However, you will now try to do it in a more structured way. The state of our system in this case consists of both the vertical velocity $Velocity_{kalman}$ and the altitude $Altitude_{kalman}$. Put both variables in the state vector S :

$$S = \begin{bmatrix} Altitude_{kalman} \\ Velocity_{kalman} \end{bmatrix}$$

Knowing that the measurement you will use to predict the vertical velocity and altitude is the acceleration in the z direction, $Acc_{z,inertial}$, you just need to integrate once to obtain the velocity in the z direction:

$$Velocity_{kalman}(k) = Velocity_{kalman}(k-1) + T_s \cdot Acc_{z,inertial}(k)$$

This gives a fully similar equation as the one you obtained when integrating the roll rate to get the roll angle in project 15. Integrating the above equation a second time gives you the altitude:

$$Altitude_{kalman}(k) = Altitude_{kalman}(k-1) + T_s \cdot Velocity_{kalman} + 0.5 \cdot T_s^2 \cdot Acc_{z,inertial}(k)$$

Both equations can be summarized in state space matrix form as:

$$\begin{bmatrix} Altitude_{kalman}(k) \\ Velocity_{kalman}(k) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix}}_{\mathbf{F}} \cdot \begin{bmatrix} Altitude_{kalman}(k-1) \\ Velocity_{kalman}(k-1) \end{bmatrix} + \underbrace{\begin{bmatrix} 0.5 \cdot T_s^2 \\ T_s \end{bmatrix}}_{\mathbf{G}} \cdot \underbrace{Acc_{z,inertial}(k)}_{\mathbf{U}}$$

This corresponds to the general equation for the state prediction $S(k)=F.S(k-1)+G.U(k)$. The uncertainty on this prediction is calculated through $P(k)=F.P(k-1)F^T+Q$. Because you will set both the altitude and velocity at startup to zero, the initial prediction for S at iteration $k=0$ is equal to:

$$S(k=0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Because this initial prediction for S is 100% accurate, the initial uncertainty on the prediction P can be set to zero as well:

$$P(k=0) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

To be able to calculate to prediction uncertainty, you still need the process uncertainty. Let's take a standard deviation of 10 cm/s² on the accelerometer values; together with the control matrix G and because Q is essentially the variance of the process uncertainty, you get:

$$Q = G \cdot G^T \cdot 10^2$$

The observation matrix H links the state with the measurement M; the error between both will be multiplied by the Kalman gain. M is in this case the altitude measured by the barometer, meaning that you can write the error between the measurement and the state as:

$$Altitude_{barometer}(k) - Altitude_{kalman}(k)$$

Writing this more generally in matrix form gives an observation matrix H equal to [1 0] and the measurement vector M equal to Altitude_{kalman}:

$$\boxed{\begin{matrix} Altitude_{kalman}(k) \\ \mathbf{M} \end{matrix}} - \boxed{\begin{matrix} [1 & 0] \\ \mathbf{H} \end{matrix}} \cdot \begin{bmatrix} Altitude_{kalman}(k) \\ Velocity_{kalman}(k) \end{bmatrix}$$

With the observation matrix H, calculate the intermediate matrix L(k)=H.P(k).H^T+R. R is the uncertainty on the barometer altitude measurement; let's take a standard deviation of 30 cm yielding R=30². From here onward, the Kalman gain can easily be calculated with the formula K=P(k).H^T.L(k)⁻¹ and the update of the prediction through S(k)=S(k)+K.(M-H.S). Finally the uncertainty on the predicted state is updated using the equation P(k)=(I-K.F)P(k). In this case, I is the 2x2 identity matrix:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

And this is all there is to it, you now have a two-dimensional Kalman filter! Let's try to implement your new filter in Arduino. For this part you need both the MPU-6050 gyro and the BMP-280 barometer, so either test the code directly on the printed circuit board of you quadcopter, or connect the sensor and Teensy separately using your breadboard as visualized on the previous page. **Be careful to connect the Vcc of the barometer to the 3.3V power source of your Teensy and not the 5V power source to avoid damaging the sensor.**

Two-dimensional form of the Kalman filter

1. Predict the current state of the system:

$$S(k) = F \cdot S(k-1) + G \cdot U(k)$$

S =state vector $\begin{bmatrix} \text{Altitude}_{\text{kalman}} \\ \text{Velocity}_{\text{kalman}} \end{bmatrix}$
 F =state transition matrix $\begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix}$
 G =control matrix $\begin{bmatrix} 0.5 \cdot T_s^2 \\ T_s \end{bmatrix}$
 U =input variable ($\text{Acc}_{z,\text{inertial}}$)

2. Calculate the uncertainty of the prediction:

$$P(k) = F \cdot P(k-1) \cdot F^T + Q$$

P =prediction uncertainty vector
 $P(k=0) = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
 Q =process uncertainty $G \cdot G^T \cdot 10^2$

3. Calculate the Kalman gain from the uncertainties on the predictions and measurements:

$$L(k) = H \cdot P(k) \cdot H^T + R$$

$$K = P(k) \cdot \frac{H^T}{L(k)} = P(k) \cdot H^T \cdot L(k)^{-1}$$

L = Intermediate matrix
 K =Kalman gain
 H =Observation matrix $\begin{bmatrix} 1 & 0 \end{bmatrix}$
 R =Measurement uncertainty (30^2)

4. Update the predicted state of the system with the measurement of the state through the Kalman gain:

$$S(k) = S(k) + K \cdot (M(k) - H \cdot S(k))$$

M =measurement vector ($\text{Altitude}_{\text{Kalman}}$)

5. Update the uncertainty of the predicted state:

$$P(k) = (I - K \cdot F) \cdot P(k)$$

I =unity matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$



Coding

Use a dedicated library to be able to implement and calculate with matrices: the BasicLinearAlgebra library. Make sure you import it by clicking on Sketch → include library → manage libraries because this is not a standard library. A namespace called BLA is used to define the matrices; for each matrix you need to declare the size; $H=[1 \ 0]$ for example is a 1x2 matrix while the state space S is a 2x1 matrix. Do not forget to define the altitude and vertical velocity that will be predicted with the Kalman filter.

Construct the function that will hold the two-dimensional Kalman filter. The acceleration value that will come from the accelerometer AccZInertial has to be transformed to the 1x1 matrix Acc. The prediction for the state space is calculated according to the formula seen in the theory of this project, together with the uncertainty on the prediction. Calculate the transpose of matrix F (F^T) by placing a \sim in front of the matrix. For the calculation of the Kalman gain, the matrix L should be inverted (L^{-1}) which you can do with the function **Invert**. The measurement matrix M consists only of the measured altitude with the barometer. Now everything is ready to calculate the updated state vector S ; extract the altitude (in cm) from the first position in the vector and the vertical velocity (in cm/s) from the second position. Do not forget to update the uncertainty on the prediction.

```

1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float AngleRoll, AnglePitch;
4 float AccX, AccY, AccZ;
5 float AccZInertial;
6 float LoopTimer;
7 uint16_t dig_T1, dig_P1;
8 int16_t dig_T2, dig_T3, dig_P2, dig_P3, dig_P4, dig_P5;
9 int16_t dig_P6, dig_P7, dig_P8, dig_P9;
10 float AltitudeBarometer, AltitudeBarometerStartUp;
11 int RateCalibrationNumber;

```

Include all variables for the gyro and accelerometer (project 14)

Include all variables for the barometer (project 17)

```

12 #include <BasicLinearAlgebra.h>
13 using namespace BLA;
14 float AltitudeKalman, VelocityVerticalKalman;
15 BLA::Matrix<2,2> F; BLA::Matrix<2,1> G;
16 BLA::Matrix<2,2> P; BLA::Matrix<2,2> Q;
17 BLA::Matrix<2,1> S; BLA::Matrix<1,2> H;
18 BLA::Matrix<2,2> I; BLA::Matrix<1,1> Acc;
19 BLA::Matrix<2,1> K; BLA::Matrix<1,1> R;
20 BLA::Matrix<1,1> L; BLA::Matrix<1,1> M;

```

Define the matrices for the two-dimensional Kalman filter

```

21 void kalman_2d(void) {
22     Acc = {AccZInertial};
23     S=F*S+G*Acc;
24     P=F*P*~F+Q;
25     L=H*P*~H+R;
26     K=P*~H*Invert(L);
27     M= {AltitudeBarometer};
28     S=S+K*(M-H*S);
29     AltitudeKalman=S(0,0);
30     VelocityVerticalKalman=S(1,0);
31     P=(I-K*H)*P;
32 }

```

Create the function that holds the two dimensional Kalman filter

```

33 void barometer_signals(void) {
34     Wire.beginTransmission(0x76);
35     Wire.write(0xF7);
36     Wire.endTransmission();
37     Wire.requestFrom(0x76,6);
38     uint32_t press_msb = Wire.read();

```

Calculate the altitude in cm from the barometric measurement (project 17)



```

39  uint32_t press_lsb = Wire.read();
40  uint32_t press_xlsb = Wire.read();
41  uint32_t temp_msb = Wire.read();
42  uint32_t temp_lsb = Wire.read();
43  uint32_t temp_xlsb = Wire.read();
44  unsigned long int adc_P = (press_msb << 12) | (
    press_lsb << 4) | (press_xlsb >>4);
45  unsigned long int adc_T = (temp_msb << 12) | (
    temp_lsb << 4) | (temp_xlsb >>4);
46  signed long int var1, var2;
47  var1 = (((adc_T >> 3) - ((signed long int)dig_T1
    <<1))) * ((signed long int)dig_T2) >> 11;
48  var2 = (((((adc_T >> 4) - ((signed long int)dig_T1
    )) * ((adc_T >>4) - ((signed long int)dig_T1)))
    >> 12) * ((signed long int)dig_T3) >> 14;
49  signed long int t_fine = var1 + var2;
50  unsigned long int p;
51  var1 = (((signed long int)t_fine >>1) - (signed
    long int)64000);
52  var2 = (((var1 >>2) * (var1 >>2)) >> 11) * ((signed
    long int)dig_P6);
53  var2 = var2 + ((var1 * ((signed long int)dig_P5))
    <<1);
54  var2 = (var2 >>2) + (((signed long int)dig_P4)
    <<16);
55  var1 = (((dig_P3 * (((var1 >>2) * (var1 >>2)) >> 13
    )) >>3) + (((signed long int)dig_P2) *
    var1) >>1)) >>18;
56  var1 = (((32768 + var1)) * ((signed long int)dig_P1))
    >>15);
57  if (var1 == 0) { p=0;}
58  p = (((unsigned long int) (((signed long int)
    1048576) - adc_P) - (var2 >>12)) * 3125;
59  if (p < 0x80000000) { p = (p << 1) / ((unsigned
    long int) var1);}
60  else { p = (p / (unsigned long int)var1) * 2; }
61  var1 = (((signed long int)dig_P9) * ((signed long
    int) ((p >>3) * (p >>3) >>13))) >>12;
62  var2 = (((signed long int)(p >>2)) *
    ((signed long int)dig_P8)) >>13;

```



```

63     p = (unsigned long int)((signed long int)p +
64         ((var1 + var2+ dig_P7) >> 4));
65     double pressure=(double)p/100;
66     AltitudeBarometer=44330*(1-pow(pressure
67         /1013.25, 1/5.255))*100;
68 }
69 void gyro_signals(void) {
70     Wire.beginTransmission(0x68);
71     Wire.write(0x1A);
72     Wire.write(0x05);
73     Wire.endTransmission();
74     Wire.beginTransmission(0x68);
75     Wire.write(0x1C);
76     Wire.write(0x10);
77     Wire.endTransmission();
78     Wire.beginTransmission(0x68);
79     Wire.write(0x3B);
80     Wire.endTransmission();
81     Wire.requestFrom(0x68,6);
82     int16_t AccXLSB = Wire.read() << 8 |
83         Wire.read();
84     int16_t AccYLSB = Wire.read() << 8 |
85         Wire.read();
86     int16_t AccZLSB = Wire.read() << 8 |
87         Wire.read();
88     Wire.beginTransmission(0x68);
89     Wire.write(0x1B);
90     Wire.write(0x8);
91     Wire.endTransmission();
92     Wire.beginTransmission(0x68);
93     Wire.write(0x43);
94     Wire.endTransmission();
95     Wire.requestFrom(0x68,6);
96     int16_t GyroX=Wire.read()<<8 | Wire.read();
97     int16_t GyroY=Wire.read()<<8 | Wire.read();
98     int16_t GyroZ=Wire.read()<<8 | Wire.read();
99     RateRoll=(float)GyroX/65.5;
100    RatePitch=(float)GyroY/65.5;
101    RateYaw=(float)GyroZ/65.5;
102    AccX=(float)AccXLSB/4096-0.05;
103    AccY=(float)AccYLSB/4096+0.01;

```

Define the gyro/ac-
celerometer function
(project 14)

Do not forget to put
your own accelerom-
eter calibration values
[here](#) (project 14)



```

99   AccZ=(float)AccZLSB/4096-0.11;
100  AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
    AccZ))*1/(3.142/180);
101  AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
    AccZ))*1/(3.142/180);
102 }
103 void setup() {
104   Serial.begin(57600);
105   pinMode(13, OUTPUT);
106   digitalWrite(13, HIGH);
107   Wire.setClock(400000);
108   Wire.begin();
109   delay(250);
110   Wire.beginTransmission(0x68);           Setup the MPU-6050
111   Wire.write(0x6B);                       (project 4)
112   Wire.write(0x00);
113   Wire.endTransmission();
114   Wire.beginTransmission(0x76);           Setup the BMP-280
115   Wire.write(0xF4);                       (project 17)
116   Wire.write(0x57);
117   Wire.endTransmission();
118   Wire.beginTransmission(0x76);
119   Wire.write(0xF5);
120   Wire.write(0x14);
121   Wire.endTransmission();
122   uint8_t data[24], i=0;
123   Wire.beginTransmission(0x76);

```

In the final setup step, you need to initialize the matrices and vectors that you will use in the two dimensional Kalman filter. Matrices F , G , H , I , Q and R stay constant throughout all iterations and were already defined in the theory part at the beginning of this project, with $T_s=0.004$ s. For matrix P and vector S , you only need to set the initial value, when k is equal to zero. As seen in the theory, you can set all elements of these matrices to zero since you know the exact starting altitude and speed, namely 0 cm and 0 cm/s respectively.

```

124 Wire.write(0x88);
125 Wire.endTransmission();
126 Wire.requestFrom(0x76,24);
127 while(Wire.available()) {
128     data[i] = Wire.read();
129     i++;
130 }
131 dig_T1 = (data[1] << 8) | data[0];
132 dig_T2 = (data[3] << 8) | data[2];
133 dig_T3 = (data[5] << 8) | data[4];
134 dig_P1 = (data[7] << 8) | data[6];
135 dig_P2 = (data[9] << 8) | data[8];
136 dig_P3 = (data[11]<< 8) | data[10];
137 dig_P4 = (data[13]<< 8) | data[12];
138 dig_P5 = (data[15]<< 8) | data[14];
139 dig_P6 = (data[17]<< 8) | data[16];
140 dig_P7 = (data[19]<< 8) | data[18];
141 dig_P8 = (data[21]<< 8) | data[20];
142 dig_P9 = (data[23]<< 8) | data[22]; delay(250);
143 for (RateCalibrationNumber=0;
      RateCalibrationNumber<2000;
      RateCalibrationNumber++) {
144     barometer_signals();
145     AltitudeBarometerStartUp+=
146     AltitudeBarometer; delay(1);
147 }
148 AltitudeBarometerStartUp/=2000;

```

Calculate the altitude reference level (project 17)

```

149 F = {1, 0.004,
150     0, 1};
151 G = {0.5*0.004*0.004,
152     0.004};
153 H = {1, 0};
154 I = {1, 0,
155     0, 1};
156 Q = G * ~G*10*10;
157 R = {30*30};
158 P = {0, 0,
159     0, 0};
160 S = {0,
161     0};

```

Define the Kalman matrix values



Measure the acceleration in the inertial Z direction and the altitude for each iteration of $T_s=0.004$. Both measurements are subsequently used in the Kalman filter function to calculate the Kalman altitude and velocity.

Print out the Kalman-filtered values for both the altitude and the velocity.

Testing the two-dimensional Kalman filter

When you test the 2D Kalman filter, you will notice that the values for both the altitude and the vertical velocity are sometimes significantly off. This is due to changes in the atmospheric pressure which is unfortunately inherent to the type of measurement. However, when implementing the filter in the flight control controller and testing the quadcopter, you will not really notice this when flying because you use a velocity control and not an altitude control. Even if the measured velocity is slightly wrong, you will be able to hover the quadcopter by adjusting the throttle stick on your radiocontroller to match this velocity; this will happen intuitively when flying.

The evolution of the Kalman gain in time is given in the figure to the right. The Kalman gain is zero initially, because you have set the initial uncertainty matrix P to zero. After a few seconds, the Kalman gain reaches its steady state at 0.0033 for the altitude calculations and 0.0013 for the vertical velocity. This means physically that the Kalman filter relies heavily on the accelerometer integration and less on the barometer measurements, so the latter is used mostly to ensure that the altitude obtained from accelerometer integration does not diverges too far.

```

162   LoopTimer=micros();
163 }

```

```

164 void loop() {
165   gyro_signals();
166   AccZInertial=-sin(AnglePitch*(3.142/180))*AccX
+cos(AnglePitch*(3.142/180))*sin(AngleRoll*
(3.142/180))* AccY+cos(AnglePitch*(3.142/180))*
cos(AngleRoll*(3.142/180))*AccZ;
167   AccZInertial=(AccZInertial-1)*9.81*100;
168   barometer_signals();
169   AltitudeBarometer=AltitudeBarometerStartUp;
170   kalman_2d();

```

Calculate the Kalman altitude and velocity

```

171   Serial.print("Altitude [cm]: ");
172   Serial.print(AltitudeKalman);
173   Serial.print(" Vertical velocity [cm/s]: ");
174   Serial.println(VelocityVerticalKalman);

```

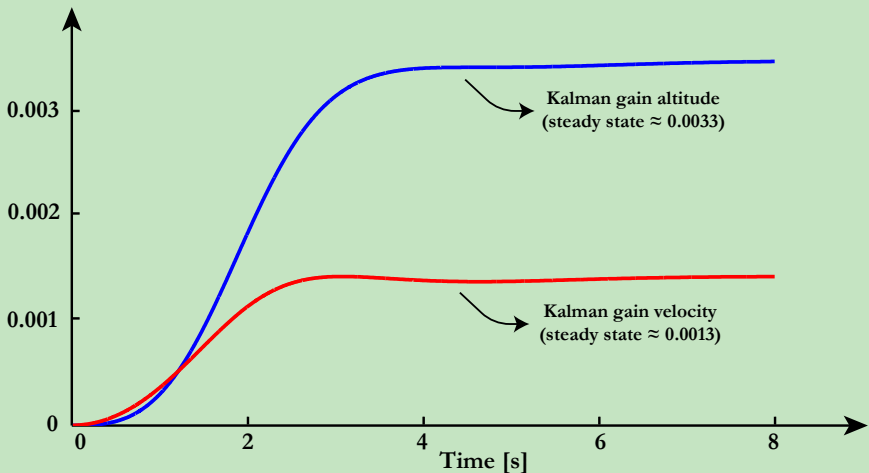
Print the altitude and velocity

```

175   while (micros() - LoopTimer < 4000);
176       LoopTimer=micros();
177 }

```

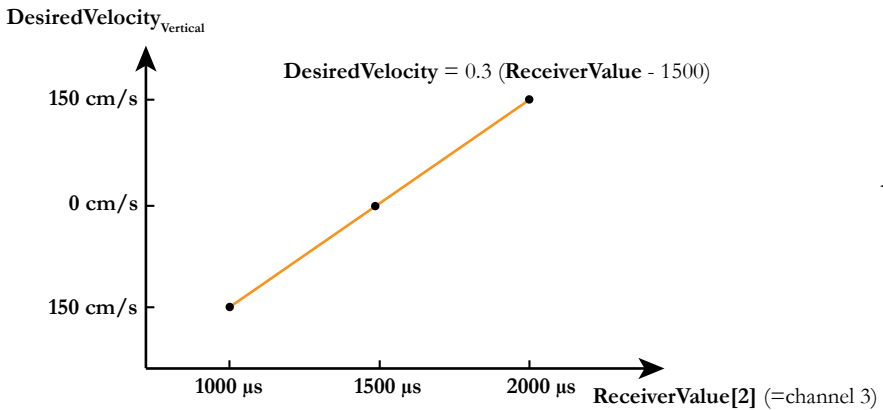
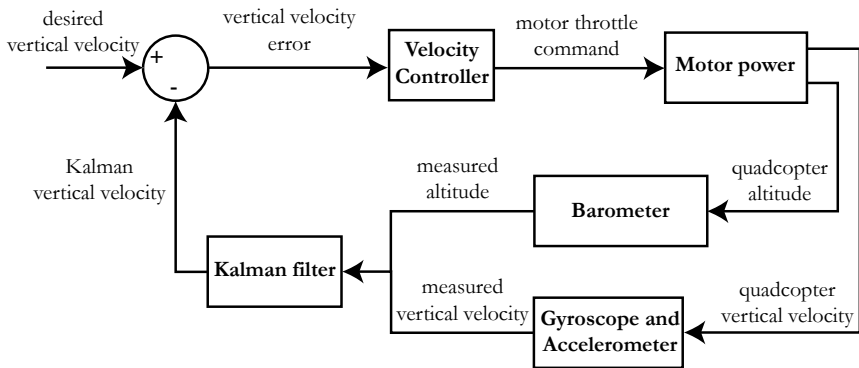
Kalman gain





Project 20

The flight controller: velocity mode



Hover your quadcopter with ease

The second flight controllers you programmed allowed you to stabilize your quadcopter based on its angles, which makes flying a lot easier. By additionally controlling the vertical velocity, the effort of flying will be reduced even more.

The flight controller you will program during this project will allow you to control the vertical velocity of your quadcopter, instead of the throttle. You can compare this with your own car; with the accelerator pedal, you control the acceleration of your car but not the speed. In order to stay at a constant speed, you almost continuously have to adjust the pedal in order to remain at a more or less equal speed. This was also true for your quadcopter when using the previous two flight controllers; you had to adjust the throttle stick almost continuously in order to hover. The velocity control that you will develop in this project is similar to the cruise control in your car; you set the speed once and the motor of your car will adjust its power in order to keep going at the required speed.

The control loop you will implement for the vertical velocity is a similar loop as you used for the rate control in your first flight controller. There are only two noticeable differences: the output for this controller is the throttle input command, instead of the roll, pitch and yaw commands. Furthermore, the measurement of the vertical velocity goes through the two dimensional Kalman filter as constructed in the previous project. The velocity controller is once again a PID controller for which you can use the already programmed function in Arduino. Good P, I and D parameters for the velocity control are:

- $P_{\text{Velocity Vertical}} = 3.5$
- $I_{\text{Velocity Vertical}} = 0.0015$
- $D_{\text{Velocity Vertical}} = 0.01$

Finally, you also need to decide how the vertical velocity input values correspond with the receiver commands. A too steep correlation will lead to very sensitive controls, while a more horizontal correlation will lead to insensitive controls. A good balance for the controls is when the maximal and minimal receiver values correspond with a vertical velocity of respectively ± 150 cm/s.

You are now ready to program your third and final flight controller, so let's start!

Coding

```

1 #include <Wire.h>
2 float RateRoll, RatePitch, RateYaw;
3 float RateCalibrationRoll, RateCalibrationPitch,
   RateCalibrationYaw;
4 int RateCalibrationNumber;
5 #include <PulsePosition.h>
6 PulsePositionInput ReceiverInput(RISING);
7 float ReceiverValue[]={0, 0, 0, 0, 0, 0, 0, 0, 0};
8 int ChannelNumber=0;
9 float Voltage, Current, BatteryRemaining, BatteryAtStart;
10 float CurrentConsumed=0;
11 float BatteryDefault=1300;
12 uint32_t LoopTimer;
13 float DesiredRateRoll, DesiredRatePitch,
   DesiredRateYaw;
14 float ErrorRateRoll, ErrorRatePitch, ErrorRateYaw;
15 float InputRoll, InputThrottle, InputPitch, InputYaw;
16 float PrevErrorRateRoll, PrevErrorRatePitch,
   PrevErrorRateYaw;
17 float PrevItermRateRoll, PrevItermRatePitch,
   PrevItermRateYaw;
18 float PIDReturn[]={0, 0, 0};
19 float PRateRoll=0.6; float PRatePitch=PRateRoll;
   float PRateYaw=2;
20 float IRateRoll=3.5; float IRatePitch=IRateRoll;
   float IRateYaw=12;
21 float DRateRoll=0.03; float DRatePitch=DRateRoll;
   float DRateYaw=0;
22 float MotorInput1, MotorInput2, MotorInput3,
   MotorInput4;
23 float AccX, AccY, AccZ;
24 float AngleRoll, AnglePitch;

25 float KalmanAngleRoll=0,
   KalmanUncertaintyAngleRoll=2*2;
26 float KalmanAnglePitch=0,
   KalmanUncertaintyAnglePitch=2*2;
27 float Kalman1DOutput[]={0,0};

```

Initialize the same variables that you already needed for rate mode (project 12)

Initialize the accelerometer variables (project 14)

Define the Kalman variables (project 15)


```

28 float DesiredAngleRoll, DesiredAnglePitch;
29 float ErrorAngleRoll, ErrorAnglePitch;
30 float PrevErrorAngleRoll, PrevErrorAnglePitch;
31 float PrevItermAngleRoll, PrevItermAnglePitch;
32 float PAngleRoll=2; float PAnglePitch=PAngleRoll;
33 float IAngleRoll=0; float IAnglePitch=IAngleRoll;
34 float DAngleRoll=0; float DAnglePitch=DAngleRoll;
35 uint16_t dig_T1, dig_P1;
36 int16_t dig_T2, dig_T3, dig_P2, dig_P3, dig_P4, dig_P5;
37 int16_t dig_P6, dig_P7, dig_P8, dig_P9;
38 float AltitudeBarometer, AltitudeBarometerStartUp;
39 float AccZInertial;
40 #include <BasicLinearAlgebra.h>
41 using namespace BLA;
42 float AltitudeKalman, VelocityVerticalKalman;
43 BLA::Matrix<2,2> F; BLA::Matrix<2,1> G;
44 BLA::Matrix<2,2> P; BLA::Matrix<2,2> Q;
45 BLA::Matrix<2,1> S; BLA::Matrix<1,2> H;
46 BLA::Matrix<2,2> I; BLA::Matrix<1,1> Acc;
47 BLA::Matrix<2,1> K; BLA::Matrix<1,1> R;
48 BLA::Matrix<1,1> L; BLA::Matrix<1,1> M;

```

Define the values necessary for the outer loop PID controller, including the P, I and D parameters (project 16)

Define the variables that you need for the two dimensional Kalman filter and barometer (project 19)

```

49 float DesiredVelocityVertical, ErrorVelocityVertical;
50 float PVelocityVertical=3.5;
   float IVelocityVertical=0.0015;
   float DVelocityVertical=0.01;
51 float PrevErrorVelocityVertical,
   PrevItermVelocityVertical;

```

Define the values necessary for the velocity PID controller, including the P, I and D parameters

```

52 void kalman_2d(void) {
53     Acc = {AccZInertial};
54     S=F*S+G*Acc;
55     P=F*P*~F+Q;
56     L=H*P*~H+R;
57     K=P*~H*Invert(L);
58     M = {AltitudeBarometer};
59     S=S+K*(M-H*S);
60     AltitudeKalman=S(0,0);
61     VelocityVerticalKalman=S(1,0);
62     P=(I-K*H)*P;
63 }

```

Define the two dimensional Kalman filter (project 19)



```

64 void barometer_signals(void) {
65     Wire.beginTransmission(0x76);
66     Wire.write(0xF7);
67     Wire.endTransmission();
68     Wire.requestFrom(0x76,6);
69     uint32_t press_msb = Wire.read();
70     uint32_t press_lsb = Wire.read();
71     uint32_t press_xlsb = Wire.read();
72     uint32_t temp_msb = Wire.read();
73     uint32_t temp_lsb = Wire.read();
74     uint32_t temp_xlsb = Wire.read();
75     unsigned long int adc_P = (press_msb << 12) | (
        press_lsb << 4) | (press_xlsb >>4);
76     unsigned long int adc_T = (temp_msb << 12) | (
        temp_lsb << 4) | (temp_xlsb >>4);
77     signed long int var1, var2;
78     var1 = (((adc_T >> 3) - ((signed long int)dig_T1
        <<1))) * ((signed long int)dig_T2) >> 11;
79     var2 = (((((adc_T >> 4) - ((signed long int)dig_T1
        )) * (adc_T >>4) - ((signed long int)dig_T1)))
        >> 12) * ((signed long int)dig_T3) >> 14;
80     signed long int t_fine = var1 + var2;
81     unsigned long int p;
82     var1 = (((signed long int)t_fine >>1) - (signed
        long int)64000);
83     var2 = (((var1 >>2) * (var1 >>2)) >> 11) * ((signed
        long int)dig_P6);
84     var2 = var2 + ((var1 * ((signed long int)dig_P5))
        <<1);
85     var2 = (var2 >>2) + (((signed long int)dig_P4)
        <<16);
86     var1 = (((dig_P3 * (((var1 >>2) * (var1 >>2)) >> 13
        )) >>3) + (((signed long int)dig_P2) *
        var1) >>1)) >>18;
87     var1 = (((32768 + var1) * ((signed long int)dig_P1))
        >>15);
88     if (var1 == 0) { p=0;}
89     p = (((unsigned long int)((signed long int)
        1048576) - adc_P) - (var2 >>12)) * 3125;
90     if(p < 0x80000000) { p = (p << 1) / ((unsigned
        long int) var1);}

```

Calculate the altitude in cm from the barometric measurement (project 17)

```

91     else { p = (p / (unsigned long int)var1) * 2; }
92     var1 = (((signed long int)dig_P9) * ((signed long
          int) (((p>>3) * (p>>3))>>13)))>>12;
93     var2 = (((signed long int)(p>>2)) *
          ((signed long int)dig_P8))>>13;
94     p = (unsigned long int)((signed long int)p +
          ((var1 + var2+ dig_P7) >> 4));
95     double pressure=(double)p/100;
96     AltitudeBarometer=44330*(1-pow(pressure
          /1013.25, 1/5.255))*100;
97 }

```

```

98 void kalman_1d(float KalmanState,
          float KalmanUncertainty, float KalmanInput,
          float KalmanMeasurement) {
99     KalmanState=KalmanState+0.004*KalmanInput;
100    KalmanUncertainty=KalmanUncertainty + 0.004
          * 0.004 * 4 * 4;
101    float KalmanGain=KalmanUncertainty * 1/
          (1*KalmanUncertainty + 3 * 3);
102    KalmanState=KalmanState+KalmanGain * (
          KalmanMeasurement-KalmanState);
103    KalmanUncertainty=(1-KalmanGain) *
          KalmanUncertainty;
104    Kalman1DOutput[0]=KalmanState;
          Kalman1DOutput[1]=KalmanUncertainty;
105 }

```

Define the 1D Kalman filter function (project 15)

```

106 void battery_voltage(void) {
107     Voltage=(float)analogRead(15)/62;
108     Current=(float)analogRead(21)*0.089;
109 }

```

Battery voltage function (project 9)

```

110 void read_receiver(void) {
111     ChannelNumber = ReceiverInput.available();
112     if (ChannelNumber > 0) {
113         for (int i=1; i<=ChannelNumber;i++) {
114             ReceiverValue[i-1]=ReceiverInput.read(i);
115         }
116     }
117 }

```

Receiver function (project 7)

```

118 void gyro_signals(void) {
119     Wire.beginTransmission(0x68);
120     Wire.write(0x1A);

```

Gyro and accelerometer function (project 14)



```

121 Wire.write(0x05);
122 Wire.endTransmission();
123 Wire.beginTransmission(0x68);
124 Wire.write(0x1C);
125 Wire.write(0x10);
126 Wire.endTransmission();
127 Wire.beginTransmission(0x68);
128 Wire.write(0x3B);
129 Wire.endTransmission();
130 Wire.requestFrom(0x68,6);
131 int16_t AccXLSB = Wire.read() << 8 |
    Wire.read();
132 int16_t AccYLSB = Wire.read() << 8 |
    Wire.read();
133 int16_t AccZLSB = Wire.read() << 8 |
    Wire.read();
134 Wire.beginTransmission(0x68);
135 Wire.write(0x1B);
136 Wire.write(0x8);
137 Wire.endTransmission();
138 Wire.beginTransmission(0x68);
139 Wire.write(0x43);
140 Wire.endTransmission();
141 Wire.requestFrom(0x68,6);
142 int16_t GyroX=Wire.read()<<8 | Wire.read();
143 int16_t GyroY=Wire.read()<<8 | Wire.read();
144 int16_t GyroZ=Wire.read()<<8 | Wire.read();
145 RateRoll=(float)GyroX/65.5;
146 RatePitch=(float)GyroY/65.5;
147 RateYaw=(float)GyroZ/65.5;

148 AccX=(float)AccXLSB/4096-0.05;
149 AccY=(float)AccYLSB/4096+0.01;
150 AccZ=(float)AccZLSB/4096-0.11;

151 AngleRoll=atan(AccY/sqrt(AccX*AccX+AccZ*
    AccZ))*1/(3.142/180);
152 AnglePitch=-atan(AccX/sqrt(AccY*AccY+AccZ*
    AccZ))*1/(3.142/180);
153 }

```

Do not forget to put your own accelerometer calibration values [here](#) (project 14)

```

154 void pid_equation(float Error, float P , float I, float D,
    float PrevError, float PrevIterm) {
155     float Pterm=P*Error;
156     float Iterm=PrevIterm+I*(Error+
        PrevError)*0.004/2;
157     if (Iterm > 400) Iterm=400;
158     else if (Iterm <-400) Iterm=-400;
159     float Dterm=D*(Error-PrevError)/0.004;
160     float PIDOutput= Pterm+Iterm+Dterm;
161     if (PIDOutput>400) PIDOutput=400;
162     else if (PIDOutput <-400) PIDOutput=-400;
163     PIDReturn[0]=PIDOutput;
164     PIDReturn[1]=Error;
165     PIDReturn[2]=Iterm;
166 }
167 void reset_pid(void) {
168     PrevErrorRateRoll=0; PrevErrorRatePitch=0;
        PrevErrorRateYaw=0;
169     PrevItermRateRoll=0; PrevItermRatePitch=0;
        PrevItermRateYaw=0;
170     PrevErrorAngleRoll=0; PrevErrorAnglePitch=0;
171     PrevItermAngleRoll=0; PrevItermAnglePitch=0;

172     PrevErrorVelocityVertical=0;
        PrevItermVelocityVertical=0;
173 }

174 void setup() {
175     pinMode(5, OUTPUT);
176     digitalWrite(5, HIGH);
177     pinMode(13, OUTPUT);
178     digitalWrite(13, HIGH);
179     Wire.setClock(400000);
180     Wire.begin();
181     delay(250);
182     Wire.beginTransmission(0x68);
183     Wire.write(0x6B);
184     Wire.write(0x00);
185     Wire.endTransmission();

```

PID function (project 12)

PID reset function (project 12)

Reset the PID error and integral values for the vertical velocity controller loop as well

Visualize the setup phase using the red LED

Setup the MPU-6050 (project 4)



```

186   Wire.beginTransmission(0x76);           Setup the BMP-280
187   Wire.write(0xF4);                       (project 17)
188   Wire.write(0x57);
189   Wire.endTransmission();
190   Wire.beginTransmission(0x76);
191   Wire.write(0xF5);
192   Wire.write(0x14);
193   Wire.endTransmission();
194   uint8_t data[24], i=0;
195   Wire.beginTransmission(0x76);
196   Wire.write(0x88);
197   Wire.endTransmission();
198   Wire.requestFrom(0x76,24);
199   while(Wire.available()){
200       data[i] = Wire.read();
201       i++;
202   }
203   dig_T1 = (data[1] << 8) | data[0];
204   dig_T2 = (data[3] << 8) | data[2];
205   dig_T3 = (data[5] << 8) | data[4];
206   dig_P1 = (data[7] << 8) | data[6];
207   dig_P2 = (data[9] << 8) | data[8];
208   dig_P3 = (data[11]<< 8) | data[10];
209   dig_P4 = (data[13]<< 8) | data[12];
210   dig_P5 = (data[15]<< 8) | data[14];
211   dig_P6 = (data[17]<< 8) | data[16];
212   dig_P7 = (data[19]<< 8) | data[18];
213   dig_P8 = (data[21]<< 8) | data[20];
214   dig_P9 = (data[23]<< 8) | data[22]; delay(250);
215   for (RateCalibrationNumber=0;
        RateCalibrationNumber<2000;
        RateCalibrationNumber++) {
216       gyro_signals();
217       RateCalibrationRoll+=RateRoll;
218       RateCalibrationPitch+=RatePitch;
219       RateCalibrationYaw+=RateYaw;
220       barometer_signals();           Calculate the altitude
221       AltitudeBarometerStartUp+=    reference level (pro-
222       AltitudeBarometer; delay(1);  ject 17)
223   }
224   RateCalibrationRoll/=2000;

```

```

225 RateCalibrationPitch/=2000;
226 RateCalibrationYaw/=2000;
227 AltitudeBarometerStartUp/=2000;
228 F = {1, 0.004,
229     0, 1};
230 G = {0.5*0.004*0.004,
231     0.004};
232 H = {1, 0};
233 I = {1, 0,
234     0, 1};
235 Q = G * ~G*10*10;
236 R = {30*30};
237 P = {0, 0,
238     0, 0};
239 S = {0,
240     0};

```

Setup the matrices for the two-dimensional Kalman filter (project 19)

```

241 analogWriteFrequency(1, 250);
242 analogWriteFrequency(2, 250);
243 analogWriteFrequency(3, 250);
244 analogWriteFrequency(4, 250);
245 analogWriteResolution(12);

```

Set the PWM frequency to 250 Hz and the resolution to 12 bit for all motors (project 8)

```

246 pinMode(6, OUTPUT);
247 digitalWrite(6, HIGH);
248 battery_voltage();
249 if (Voltage > 8.3) { digitalWrite(5, LOW);
250     BatteryAtStart=BatteryDefault; }
251 else if (Voltage < 7.5) {
252     BatteryAtStart=30/100*BatteryDefault ;}
253 else { digitalWrite(5, LOW);
254     BatteryAtStart=(82*Voltage-580)/100*
        BatteryDefault; }

```

Show the end of the setup process and determine the initial battery voltage percentage (project 9)

```

255 ReceiverInput.begin(14);
256 while (ReceiverValue[2] < 1020 ||
        ReceiverValue[2] > 1050) {
257     read_receiver();
258     delay(4);
259 }
260 LoopTimer=micros();
261 }

```

SAFETY RELATED LINES: Avoid accidental lift off after the setup process (project 12)



```

262 void loop() {
263     gyro_signals();
264     RateRoll=RateCalibrationRoll;
265     RatePitch=RateCalibrationPitch;
266     RateYaw=RateCalibrationYaw;

267     kalman_1d(KalmanAngleRoll,
KalmanUncertaintyAngleRoll, RateRoll, AngleRoll);
268     KalmanAngleRoll=Kalman1DOutput[0];
KalmanUncertaintyAngleRoll=Kalman1DOutput[1];
269     kalman_1d(KalmanAnglePitch,
KalmanUncertaintyAnglePitch, RatePitch, AnglePitch);
270     KalmanAnglePitch=Kalman1DOutput[0];
KalmanUncertaintyAnglePitch=Kalman1DOutput[1];

271     AccZInertial=-sin(AnglePitch*(3.142/180))*AccX
+cos(AnglePitch*(3.142/180))*sin(AngleRoll*
(3.142/180))* AccY+cos(AnglePitch*(3.142/180))*
cos(AngleRoll*(3.142/180))*AccZ;
272     AccZInertial=(AccZInertial-1)*9.81*100;
273     barometer_signals();
274     AltitudeBarometer=AltitudeBarometerStartUp;
275     kalman_2d();

276     read_receiver();

277     DesiredAngleRoll=0.10*(ReceiverValue[0]-1500);
278     DesiredAnglePitch=0.10*(ReceiverValue[1]-1500);
279     DesiredRateYaw=0.15*(ReceiverValue[3]-1500);

280     DesiredVelocityVertical=0.3*(ReceiverValue[2]-
1500);
281     ErrorVelocityVertical=DesiredVelocityVertical-
VelocityVerticalKalman;
282     pid_equation(ErrorVelocityVertical,
PVelocityVertical, IVelocityVertical,
DVelocityVertical, PrevErrorVelocityVertical,
PrevItermVelocityVertical);

```

Measure the rotation rates and subtract the calibration values (project 5)

Calculate the roll and pitch angles through the Kalman filter (project 15)

Calculate the vertical acceleration and the altitude (project 18)

Calculate the desired angles from the receiver (project 16)

Calculate the desired velocity from the receiver and start the PID loop for the throttle


```
283 InputThrottle=1500+PIDReturn[0];
    PrevErrorVelocityVertical=PIDReturn[1];
    PrevItermVelocityVertical=PIDReturn[2];
```

Because the zero velocity point and thus the hover point will be around the point where the throttle stick is in the middle (1500 μ s), add this value to the PID output

```
284 ErrorAngleRoll=DesiredAngleRoll-
    KalmanAngleRoll;
285 ErrorAnglePitch=DesiredAnglePitch-
    KalmanAnglePitch;
```

Calculate the difference between the desired and the actual roll and pitch angles (project 16)

```
286 pid_equation(ErrorAngleRoll, PAngleRoll,
    IAngleRoll, DAngleRoll, PrevErrorAngleRoll,
    PrevItermAngleRoll);
```

Calculate the desired roll and pitch rotation rates through the outer loop PID controller (project 16)

```
287 DesiredRateRoll=PIDReturn[0];
    PrevErrorAngleRoll=PIDReturn[1];
    PrevItermAngleRoll=PIDReturn[2];
```

```
288 pid_equation(ErrorAnglePitch, PAnglePitch,
    IAnglePitch, DAnglePitch, PrevErrorAnglePitch,
    PrevItermAnglePitch);
```

```
289 DesiredRatePitch=PIDReturn[0];
    PrevErrorAnglePitch=PIDReturn[1];
    PrevItermAnglePitch=PIDReturn[2];
```

```
290 ErrorRateRoll=DesiredRateRoll-RateRoll;
291 ErrorRatePitch=DesiredRatePitch-RatePitch;
292 ErrorRateYaw=DesiredRateYaw-RateYaw;
```

Calculate the difference between the desired and the actual pitch, roll and yaw rotation rates. Use these for the PID controller of the inner loop (project 12)

```
293 pid_equation(ErrorRateRoll, PRateRoll, IRateRoll,
    DRateRoll, PrevErrorRateRoll,
    PrevItermRateRoll);
```

```
294 InputRoll=PIDReturn[0];
    PrevErrorRateRoll=PIDReturn[1];
    PrevItermRateRoll=PIDReturn[2];
```



```

295  pid_equation(ErrorRatePitch, PRatePitch,
        IRatePitch, DRatePitch, PrevErrorRatePitch,
        PrevItermRatePitch);
296  InputPitch=PIDReturn[0];
        PrevErrorRatePitch=PIDReturn[1];
        PrevItermRatePitch=PIDReturn[2];
297  pid_equation(ErrorRateYaw, PRateYaw,
        IRateYaw, DRateYaw, PrevErrorRateYaw,
        PrevItermRateYaw);
298  InputYaw=PIDReturn[0];
        PrevErrorRateYaw=PIDReturn[1];
        PrevItermRateYaw=PIDReturn[2];

299  if (InputThrottle > 1800) InputThrottle = 1800;
                                     Limit the throttle val-
                                     ue to 80% (project
                                     12)

300  MotorInput1= 1.024*(InputThrottle-InputPitch-
        InputRoll-InputYaw);
                                     Use the quadcopter
301  MotorInput2= 1.024*(InputThrottle+InputPitch-
        InputRoll+InputYaw);
                                     dynamics equations
302  MotorInput3= 1.024*(InputThrottle+InputPitch+
        InputRoll-InputYaw);
                                     (project 11)
303  MotorInput4= 1.024*(InputThrottle-InputPitch+
        InputRoll+InputYaw);

304  if (MotorInput1 > 2000)MotorInput1 = 1999;
305  if (MotorInput2 > 2000)MotorInput2 = 1999;
306  if (MotorInput3 > 2000)MotorInput3 = 1999;
307  if (MotorInput4 > 2000)MotorInput4 = 1999;
                                     Limit the maximal
                                     power commands
                                     sent to the motors
                                     (project 12)

308  int ThrottleIdle=1180;
309  if (MotorInput1 < ThrottleIdle) MotorInput1 =
        ThrottleIdle;
310  if (MotorInput2 < ThrottleIdle) MotorInput2 =
        ThrottleIdle;
311  if (MotorInput3 < ThrottleIdle) MotorInput3 =
        ThrottleIdle;
312  if (MotorInput4 < ThrottleIdle) MotorInput4 =
        ThrottleIdle;
                                     Keep the quadcop-
                                     ter minimally at 18%
                                     power during flight

```

```

313 int ThrottleCutOff=1000;
314 if (ReceiverValue[2]<1050) {
315     MotorInput1=ThrottleCutOff;
316     MotorInput2=ThrottleCutOff;
317     MotorInput3=ThrottleCutOff;
318     MotorInput4=ThrottleCutOff;
319     reset_pid();
320 }
321 analogWrite(1,MotorInput1);
322 analogWrite(2,MotorInput2);
323 analogWrite(3,MotorInput3);
324 analogWrite(4,MotorInput4);

325 battery_voltage();
326 CurrentConsumed=Current*1000*0.004/3600+
    CurrentConsumed;
327 BatteryRemaining=(BatteryAtStart-
    CurrentConsumed)/BatteryDefault*100;
328 if (BatteryRemaining<=30) digitalWrite(5, HIGH);
329 else digitalWrite(5, LOW);

330 while (micros() - LoopTimer < 4000);
331 LoopTimer=micros();
332 }

```

SAFETY RELATED LINES: stop the motors when throttle stick is fully down

Sent the commands to the motors

Keep track of battery level (project 9)

Finish the 250 Hz control loop

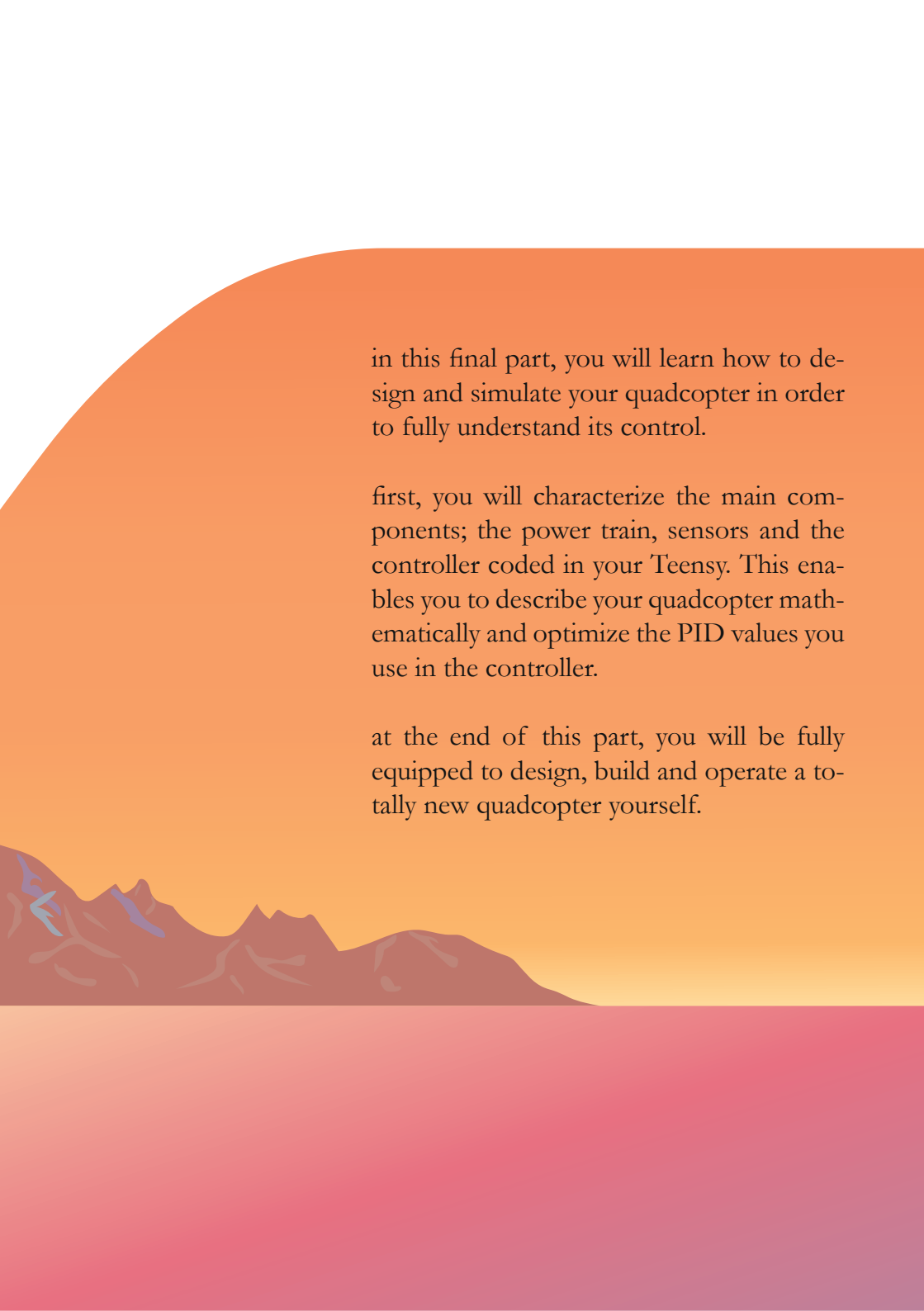
Start-up and flying your quadcopter

To start and fly your quadcopter with the new vertical velocity-mode flight controller, follow the same steps as with your rate-mode controller and your stabilize-mode controller. Remember that this controller is meant for flying indoors. You should notice that flying the quadcopter with your new flight controller is even easier as the controller facilitates the hovering. Congratulations, you have successfully reached the end of the practical part of this manual!



Part IV: quadcopter design and simulation





in this final part, you will learn how to design and simulate your quadcopter in order to fully understand its control.

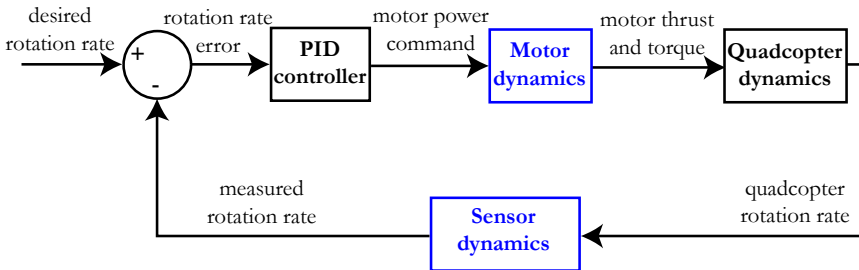
first, you will characterize the main components; the power train, sensors and the controller coded in your Teensy. This enables you to describe your quadcopter mathematically and optimize the PID values you use in the controller.

at the end of this part, you will be fully equipped to design, build and operate a totally new quadcopter yourself.



Project 21

Motor and sensor simulation



Your two bladed propeller generates a less thrust than the three bladed 3035 propeller, although it is driven by a motor that turns less fast (4500 kV instead of 5000 kV). The reason is simple; the three bladed propeller ‘catches’ more air under its surface, generating more thrust but also requiring the motor to overcome more air resistance; this means that the current consumption increases as well.

The thrust and current relation with the motor power evolves almost linearly; this means that you can simply perform linear regression in order to estimate the throttle-thrust and throttle-current relationship. For the GEPRC 1105 5000 kV motor and 3018 propeller combination, these relationships are:

$$\begin{aligned}\text{thrust [g]} &= 160 \cdot \text{throttle [0 : 1]} \\ \text{current [A]} &= 4.4 \cdot \text{throttle [0 : 1]} + 0.132\end{aligned}$$

Where the throttle is a value between 0 (0%) and 1 (100%). The measurements from which these linear regressions are constructed were performed at a constant battery voltage of 7.8 V.

Learn to characterize your motors and sensors

Understanding how to characterize the motors and sensors you use to fly is critical for the design and simulation of your quadcopter. The static and dynamic aspects of both components will be investigated thoroughly during this theoretical project.

When starting to think about the design and simulation of your quadcopter, it is useful to take a step back and look to the basic loop of the rate-mode controller, visualized in the figure to the left. You start with a desired rotation rate, given by the position of the sticks of your radiocontroller. The desired rotation rate is fed to your PID controller and gives a motor power command. The power command is subsequently converted to a certain amount of thrust generated by each individual motor, which in turn will affect the movement (e.g. rotation rate) of your quadcopter. The rotation rate is then measured with your sensor, compared to the desired rotation rate and fed back into your PID controller to start the second loop. In this project, you will characterize two important components of this control loop: the motors and the sensors.

Static motor modelling

When modelling your brushless motors, you are primarily interested in the relation between the throttle and the thrust provided by your motor-propeller combination, as well as the current consumption. **A general rule in quadcopter design says that the maximal thrust generated by your four motors should be equal to two times the quadcopter weight, to ensure enough flexibility during flight.** This means that one motor should generate about half the thrust necessary to hover the quadcopter. Your quadcopter weighs around 250 gram (g), which means that the maximal thrust of one motor should be equal to at least 125 g.

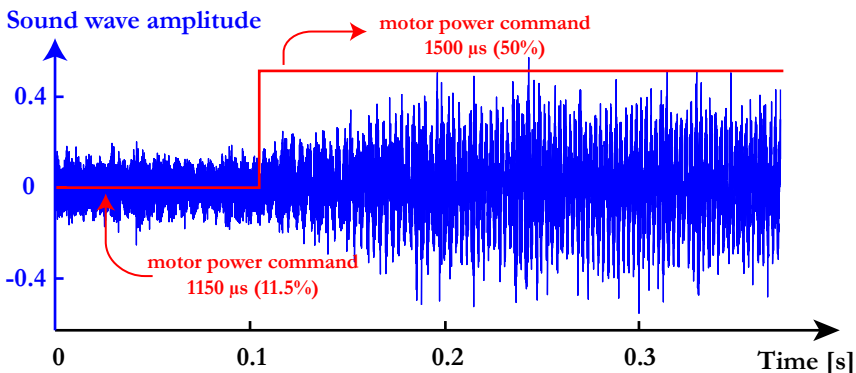
The relation between the thrust and throttle can be measured with a thrust bench; you can create this device yourself by mounting your motor on a load cell connected to a microchip (for example the HX711), amplifying the load cell signals and connecting this in turn to your Teensy. When you have a lab power supply, you can also measure the current that goes to the motors during testing. The datasheet of the motor manufacturer usually gives the maximal thrust and current for a given type of propeller. The thrust and current data for the motors used in this project are visualised on the next pages: the relation between the motor thrust and current is displayed each time with respect to the throttle level. To highlight the importance of the propeller used, your two bladed propeller (Gemfan 3018) is compared to a three bladed propeller (Gemfan 3035).

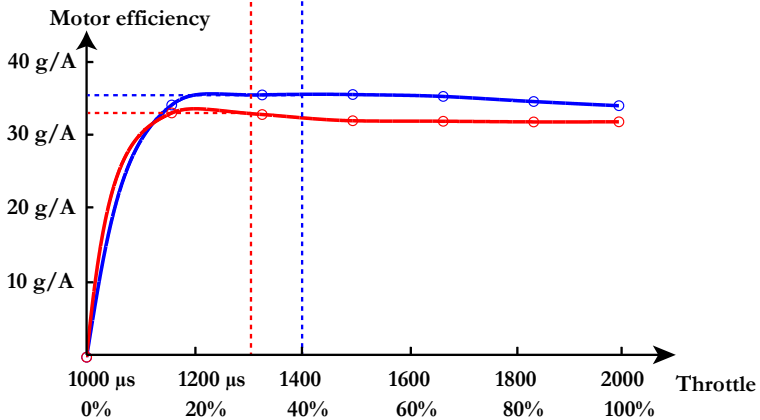
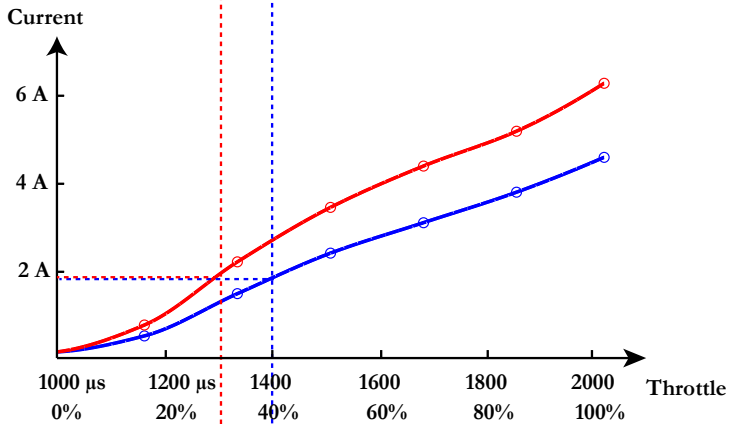
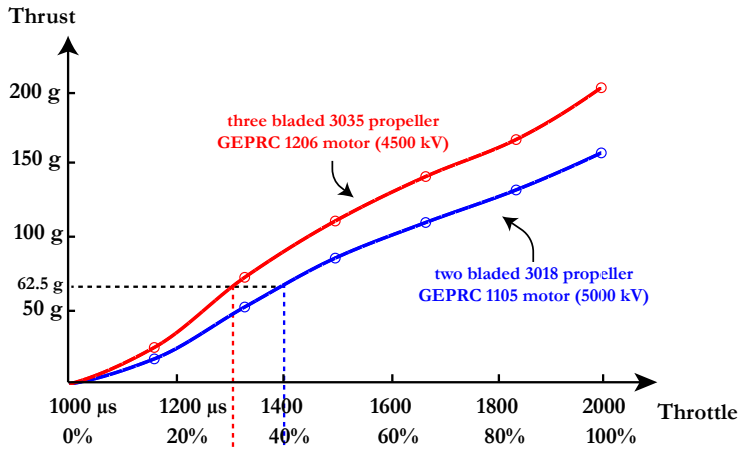
This is important because a higher/lower voltage will of course generate a higher/lower thrust and current. For a quadcopter that weighs about 250 g, the motors need to provide a thrust equal to $4 \times 62.5 \text{ g}$ when hovering. Using the formula from the previous page, this corresponds to a power level for one motor of $62.5 / 160 = 0.4$ or 40%. The current consumption in this case will be equal to $4.4 \cdot 0.4 + 0.132 = 1.9$ Ampere (A) for each motor, or 7.4 A for all motors together. This means that a 1300 mAh (=1.3 Ah) battery will be able to keep your quadcopter $1.3 \text{ Ah} / 7.8 \text{ A} = 0.17 \text{ h}$ or 10 minutes in the air. In reality this value will be lower, as you do not only hover but also move the quadcopter around and accelerate, consuming more energy than during static hovering. The above calculation gives you nonetheless a good upper limit of the flight time.

By dividing the thrust with the consumed current, the motor efficiency (in g/A) can also be determined as displayed on the figure to the right. Two bladed propellers will be slightly more efficient than three bladed propellers and the larger the propeller, the more efficient as well.

Dynamic motor modelling

Besides the generated thrust and the consumed current of your motor, another important parameter is necessary for modelling; how fast does your motor accelerate when increasing the motor power? This is important because your PID controller generates a new command every 0.004 seconds, but your motor generally needs more time to accelerate/decelerate up to this desired value. The easiest way to measure this (usually very short) time delay is by recording the sound; both your motor and your propeller generate noise which changes in amplitude and frequency when their speed increases or decreases. You can configure your Teensy in such a way that it gives a step increase for one of the motor power commands and record the sound with your cellphone during this step increase. The evolution of the sound wave amplitude is visualised in the figure below:





At 0.11 seconds, the motor power command changes from 1150 μs or 11.5% to 1500 μs or 50%. Immediately the amplitude of the sound coming from the motor/propeller increases. It takes around 0.1 to 0.2 seconds before the amplitude reaches a new steady state. To characterize the time delay more accurately, you will not look at the amplitude change, but at the frequency change of the sound during acceleration.

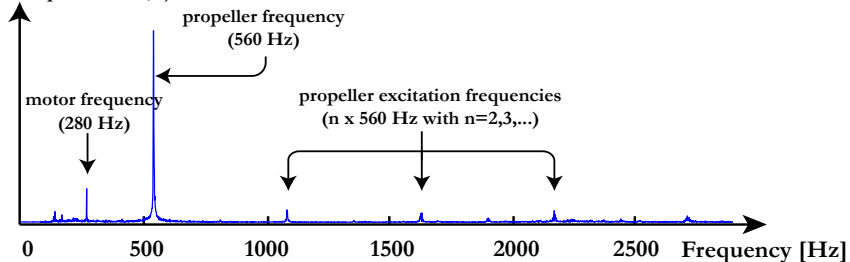
Let's now take a fast Fourier transform to transform the sound signal from the time domain to the frequency domain. By doing this, you will lose the information on the time, so let's do it first for the part at which the motor power turns stationary at 1150 μs or 11.5% throttle. You know that your motor has a 5000 kV rating. Since kV is the equivalent for rpm/V and the test was done with a constant battery voltage of 7.8 V, your motor frequency at full throttle (100 %) will be equal to:

$$5000 \frac{\text{rpm}}{\text{V}} \cdot 7.8 \text{ V} = 39\,000 \text{ rpm or } \frac{39\,000 \text{ rounds/min}}{60 \text{ s/min}} = 650 \text{ Hz}$$

Since your propeller has two blades, the frequency of the sound emitted by the propellers will be equal to $2 \times 650 \text{ Hz} = 1300 \text{ Hz}$ at full throttle and without losses. When not at full throttle, the ESC will lower the average voltage in order for the motor to spin slower. At 11.5% motor power, the fast Fourier transform of the emitted sound gives the frequency spectrum displayed on the figure below. The first peak in the spectrum is due to the sound emitted by the motor; it is situated at 280 Hz for this low motor power, meaning that the motor turns at $280 \text{ Hz} \times 60 \text{ s/min} = 16\,800 \text{ rpm}$. The second and largest peak is exactly equal to $2 \times 280 \text{ Hz}$ or 560 Hz: this is the two-bladed propeller frequency. Further in the spectrum, some small additional peaks are recorded at equal intervals; these are the propeller excitation frequencies.

Sound wave magnitude

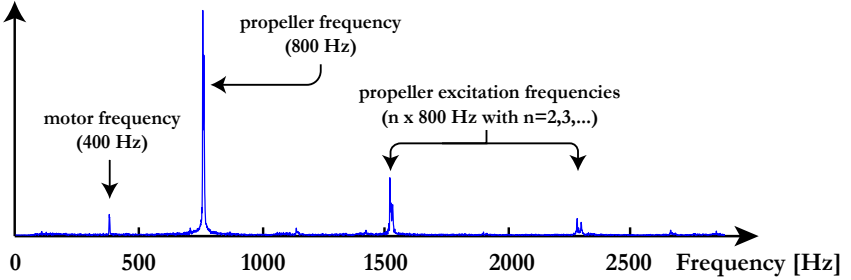
(motor power: 1150 μs)



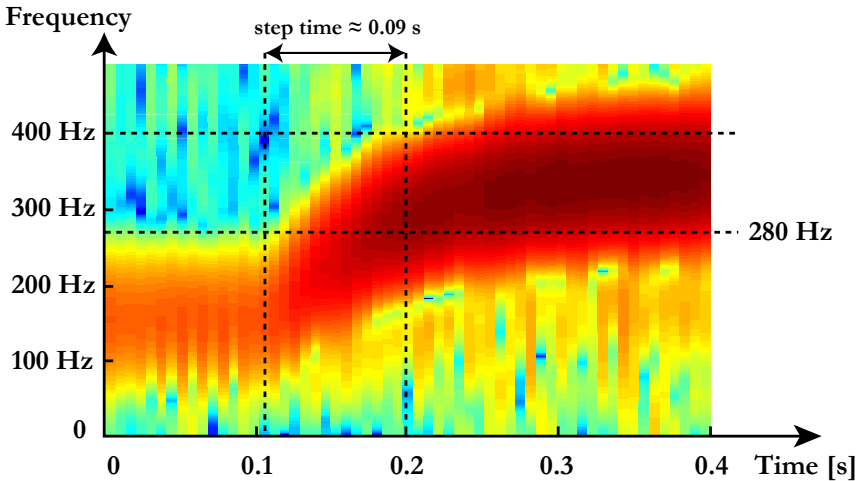
A second fast Fourier transform is used to visualize the frequency after the motor power increase, at 1500 μs or 50% power. The first peak coming from the sound of the motor is now situated at 400 Hz, meaning that the motor turns at $400 \text{ Hz} \times 60 \text{ s/min} = 24\,000 \text{ rpm}$. The propeller frequency is now equal to $2 \times 400 \text{ Hz}$ or 800 Hz.

In the higher frequency ranges, you find once again the propeller excitation frequencies.

Sound wave magnitude
(motor power: 1500 μ s)



This means that purely by recording the sound emitted by the motor, our fast Fourier transform reveals that the motor frequency increases from 280 Hz at 11.5% throttle to 400 Hz at 50% throttle. However, with this transformation all information with regard to time is lost. This can be solved with the so-called spectrogram; with a spectrogram, the frequencies are displayed for each chosen time interval allowing you to follow the change in frequency over time. The spectrogram for the acceleration of our motor (so frequency range up to 500 Hz) is given in the picture below. You can nicely follow the frequency increase from 280 Hz at a motor power of 11.5% to 400 Hz at a motor power of 50%. The time it takes for the motor to accelerate up to 400 Hz is equal to 0.09 seconds. This key piece of information will enable you to model the transfer function for the motor.



Transfer function for the motor response

You will now describe the time response of the motor compared to the command you have given to the motor mathematically. This is done using a transfer function. When looking at the behaviour of the motor visualized on the spectrogram, it resembles a first order response to a step input. Consider a general example in which a step input command of 1 is given at time = 0 seconds to your motor. Mathematically, this can be described as:

$$\begin{aligned}\text{input}(\text{time}) &= 0 \text{ if } \text{time} < 0 \\ \text{input}(\text{time}) &= 1 \text{ if } \text{time} > 0\end{aligned}$$

Describe now the first order response of a motor using one time-related parameter τ :

$$\text{output}(\text{time}) = (1 - e^{-\frac{t}{\tau}}) \cdot \text{input}(\text{time})$$

As visualised on the figures to the right with different step-response lengths, τ is defined as the time at which the motor output reaches 95% of the desired value divided by 3. For our motor, this gives a time parameter τ of $\tau=0.09$ seconds/ $3=0.03$ seconds. For further calculations, your transfer function needs to be written in the frequency domain, not the time domain. This means that you need the Laplace transform of the above equation. The Laplace transform of 1 is easy and equal to $1/s$. In this notation, s is not the unit of time, but a complex number with the form $s = \sigma + i \cdot \omega$. You also know that the Laplace transform of e^{-at} is equal to $1/(s+a)$ by convention, so transforming the above equation to the Laplace domain gives you:

$$\text{output}(s) = \left(\frac{1}{s} - \left(\frac{1}{s + \frac{1}{\tau}} \right) \right) \cdot \text{input}(s)$$

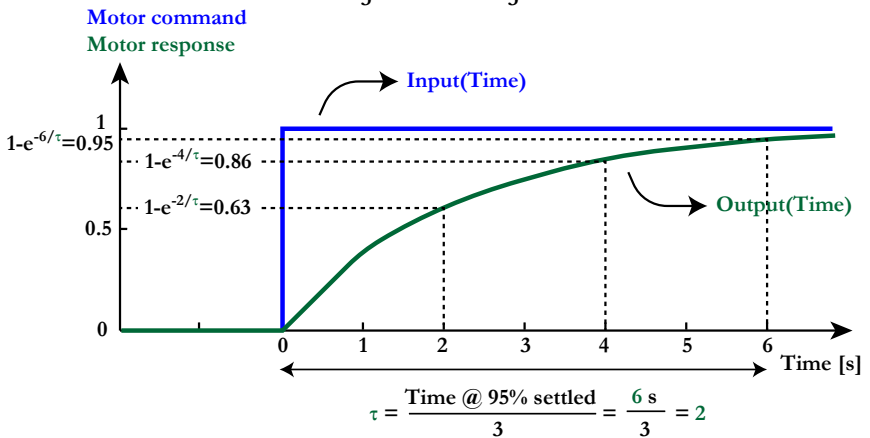
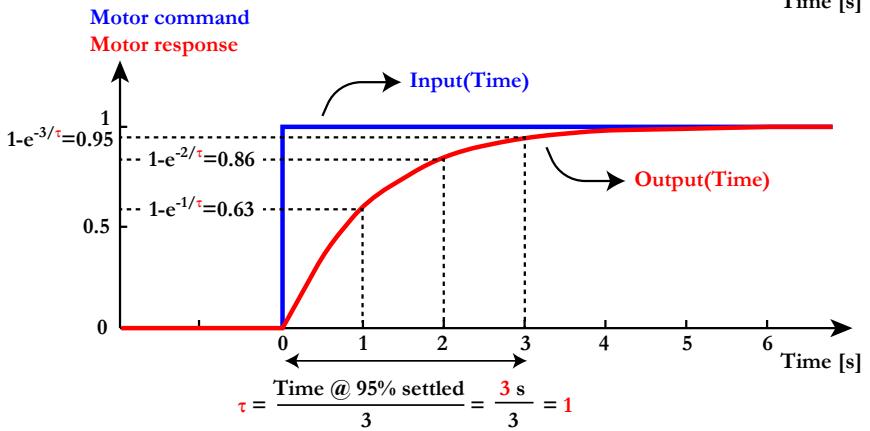
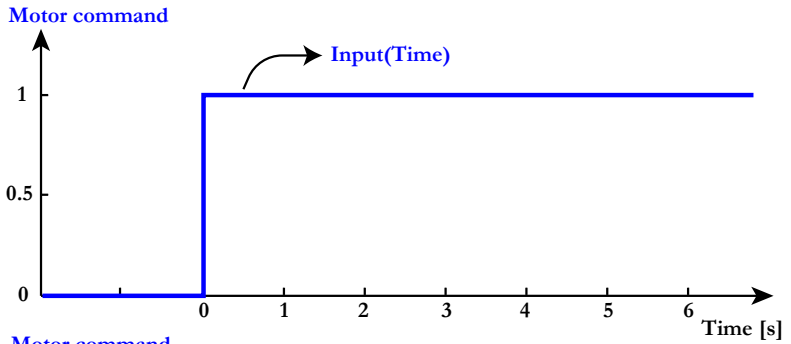
$$\text{output}(s) = \frac{s + \frac{1}{\tau} - s}{s \cdot \left(s + \frac{1}{\tau} \right)} \cdot \text{input}(s)$$

$$\text{output}(s) = \frac{1}{\tau \cdot s + 1} \cdot \frac{\text{input}(s)}{s}$$

And $\text{input}(s)/s$ represents the unit step input considered in this example. In reality, you can have any input that you want, giving the final transfer function for a first order response:

$$\text{output}(s) = \frac{1}{\tau \cdot s + 1} \cdot \text{input}(s)$$

With τ being equal to 0.03 seconds for your motor. This transfer function is a good approximation for the dynamics of your motor.



Dynamic sensor modelling

The state of your quadcopter (absolute angle and angular rates) is measured by a separate measurement system; the MPU-6050 sensor. This measurement system also has a dynamic response, as it does not respond instantaneously to the demands of your Teensy to provide some data on the quadcopter state. However, the sampling time of your MPU-6050 is equal to 1 kHz or 1000 Hz, which is much faster than the speed of your control loop (250 Hz). In addition, you also configured a 10 Hz low pass filter for both the accelerometer and the gyroscope in your code. With this filter, you are able to filter out unwanted high-frequency motor vibrations. Hence to model the time response of your MPU-6050 accurately, you only need to model its low-pass filter as its has the lowest frequency. The transfer function of any low-pass filter in the frequency domain is equal to:

$$\text{output}(s) = \frac{\omega_c}{s + \omega_c} \cdot \text{input}(s)$$

where $\omega_c = 2\pi \cdot f_c$ with f_c being the cutoff frequency of the filter, 10 Hz in this case. This means that the frequency domain transfer function for your sensor becomes:

$$\text{output}(s) = \frac{2 \cdot \pi \cdot 10}{s + 2 \cdot \pi \cdot 10} \cdot \text{input}(s)$$

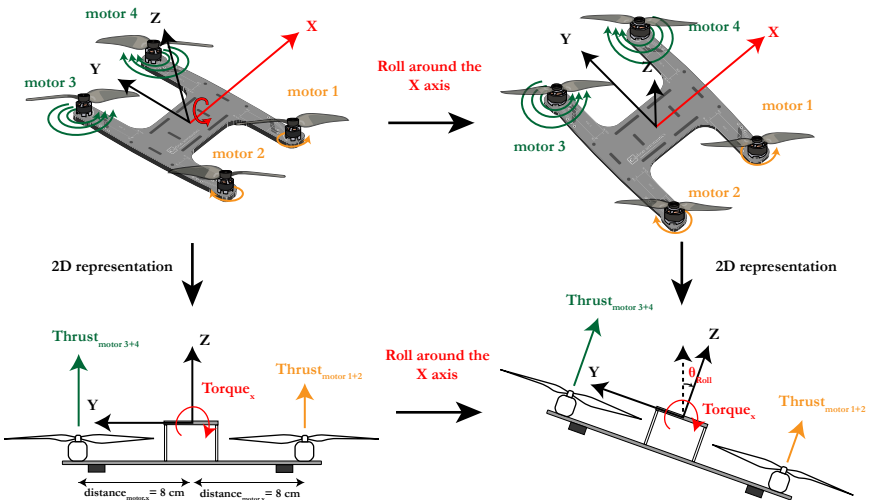
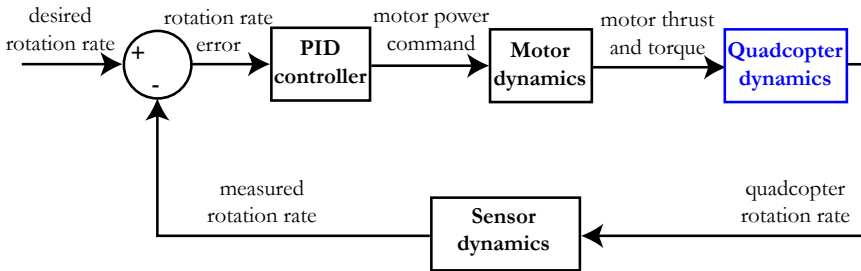
Now what does this practically mean in the time domain? Well, the transfer function can be transformed to $1/(1/(2\pi \cdot 10)s + 1)$, which gives a first order step response τ of $1/2\pi \cdot 10$ or 0.016 seconds. With this response, the same reasoning holds as with the motor response; any inputs that occur faster than $3\tau = 3 \cdot 0.016$ seconds will be significantly attenuated. An input of 0.016 seconds ($=\tau$) for example, is attenuated to 63% of its value as $1 - e^{-0.016/0.016} = 0.63$. A fast vibration with a response of 0.001 seconds gets almost fully attenuated: $1 - e^{-0.001/0.016} = 0.06$ or 6%. This is why configuring a 10 Hz low-pass filter in your MPU-6050 sensor was sufficient to filter out the high-frequency vibrations coming from your motors.





Project 22

Quadcopter dynamics simulation



	Input _{Throttle}	Input _{Roll}
output motor 1 = 25% power =	50%	-25%
output motor 2 = 25% power =	50%	-25%
output motor 3 = 75% power =	50%	+25%
output motor 4 = 75% power =	50%	+25%

Describe how the quadcopter moves in space

A mathematical discussion on quadcopter dynamics is essentially an analysis of its roll, pitch, yaw and throttle reactions. As you already saw when programming your the flight controller, these movements can all be calculated separately; this means that all four movements can be described independent of each other. While it is not fully true that these movements are independent from each other, it will prove to be a very good approximation.

The previous project ended with the mathematical description of the motors and sensor dynamics using transfer functions. In this project, you will try to find the transfer function for the roll, pitch and yaw rotation rates, and also for the vertical velocity. These four transfer functions will form the "quadcopter dynamics". All movements of the quadcopter are essentially generated by the thrust and torque developed by your motors, in combination with the gravitation force that acts on it during flight. You will describe the roll motion of the quadcopter to derive the roll dynamics and proceed subsequently with the other movements.

Roll and pitch motion

Let's first try to describe the roll motion of the quadcopter in mathematical terms. To rotate the quadcopter in the roll direction, you need to apply a torque around the stationary x-axis: Torque_x. The resulting angular acceleration Acceleration_{roll} in the roll direction depends on this applied torque, but also on the distribution of mass of the object. This mass distribution is described by the moment of inertia I_x, which results in the following relation:

$$\text{Acceleration}_{\text{roll}} [\text{rad/s}] = \frac{\text{Torque} [N \cdot m]}{I_x [kg \cdot m^2]}$$

Where N is the unit Newton (equivalent to kg.m²/s²) and rad the angle unit radians. In order to convert the angular acceleration from radians/s² to °/s², the formula needs to be multiplied with 180/π (°/rad):

$$\text{Acceleration}_{\text{roll}} [^\circ/\text{s}^2] = \frac{\text{Torque}_x [N \cdot m]}{I_x [kg \cdot m^2]} \cdot \frac{180}{\pi}$$

Torque around the x axis

The torque around the x axis depends on the combination of the thrust from all four motors and the distance of each motor to the x axis:

$$\text{Torque}_x [N \cdot m] = \text{Thrust}_{\text{motor } 3+4} [N] \cdot \text{distance}_{\text{motor},x} [m] - \text{Thrust}_{\text{motor } 1+2} [N] \cdot \text{distance}_{\text{motor},x} [m]$$

You already determined the thrust of one motor (in gram g):

$$\text{thrust [g]} = 160 \cdot \text{throttle [0 : 1]}$$

This can be transformed to N (or $\text{kg}\cdot\text{m}^2/\text{s}^2$) by multiplying the result with 9.81 m/s^2 (the gravitational constant) divided by 1000 g/kg . Moreover, you do not input a motor power between 0 and 1 but rather a value between $0 \mu\text{s}$ and $1000 \mu\text{s}$ through the parameter $\text{input}_{\text{roll}}$ (although in the PID loop it is restricted to a maximal value of $400 \mu\text{s}$). The formula transforms to:

$$\text{thrust [N]} = 160 \cdot \left(\frac{\text{input}_{\text{roll}}[0 : 1000]}{1000} \right) [g] \cdot \frac{9.81 [m/s^2]}{1000 [g/kg]}$$

$$\text{thrust [N]} = 0.160 \cdot (\text{input}_{\text{roll}}[0 : 1000]) [g] \cdot \frac{9.81 [m/s^2]}{1000 [g/kg]}$$

The value of $\text{input}_{\text{roll}}$ is equal for all four motors, but has a positive sign for motors 3 and 4 and a negative sign for motors 1 and 2. This means that the torque formula can be rewritten as:

$$\text{Torque}_x [N\cdot m] = 4 \cdot (0.160 \cdot \text{input}_{\text{roll}}[0 : 1000]) [g] \cdot \frac{9.81 [m/s^2]}{1000 [g/kg]} \cdot \text{distance}_{\text{motor},x} [m]$$

Moment of inertia around the x axis

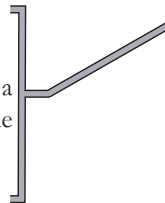
The moment of inertia I_x in turn describes the weight distribution of the quadcopter and the distance of all components to the stationary x axis. Using the two-dimensional approximation of the quadcopter, the moment of inertia can be calculated as:

$$I_x [kg \cdot m^2] = \sum_{i=1} (\text{mass}_i [kg] \cdot \text{distance}_{\text{mass},x}^2 [m^2])$$

Where mass_i is the mass of component i , and $\text{distance}_{\text{mass},x}$ the distance between component i and the x-axis. Most weight of the quadcopter is situated very close to the x-axis: the battery and all the on-board electronics are stacked along the x-axis. Because this gives a very small distance, their contribution to I_x is negligible. The weight of the frame is very small meaning that this contribution is also negligible. Therefore, the only components with a non-negligible weight and sufficiently far from the x-axis are the four motors and the ESCs. The formula reduces to:

$$I_x = 4 \cdot \text{mass}_{\text{motor}} \cdot \text{distance}_{\text{motor},x}^2 + 4 \cdot \text{mass}_{\text{ESC}} \cdot \text{distance}_{\text{ESC},x}^2$$

Knowing that the mass of each motor (including propeller and bolts) is 8 g with a distance of 8 cm to the x axis, and the ESC weighs 7 g with a distance of 4 cm to the x axis, you now have everything to calculate the angular roll acceleration.



Roll and pitch dynamics

Introducing the formulas for the torque and the moment of inertia in the angular roll acceleration formula gives:

$$\text{Acceleration}_{\text{roll}} [^\circ/s^2] = \frac{180}{\pi} \cdot \frac{4 \cdot 0.160 \cdot \text{input}_{\text{roll}} \cdot \frac{9.81}{1000} \cdot 0.08}{4 \cdot 0.008 \cdot 0.08^2 + 4 \cdot 0.007 \cdot 0.04^2}$$

Which results in a very simple formula for the angular acceleration in the time domain:

$$\text{Acceleration}_{\text{roll}} [^\circ/s^2] = 115 \cdot \text{input}_{\text{roll}}$$

The angular roll acceleration can be replaced by the derivative of the roll rate, giving:

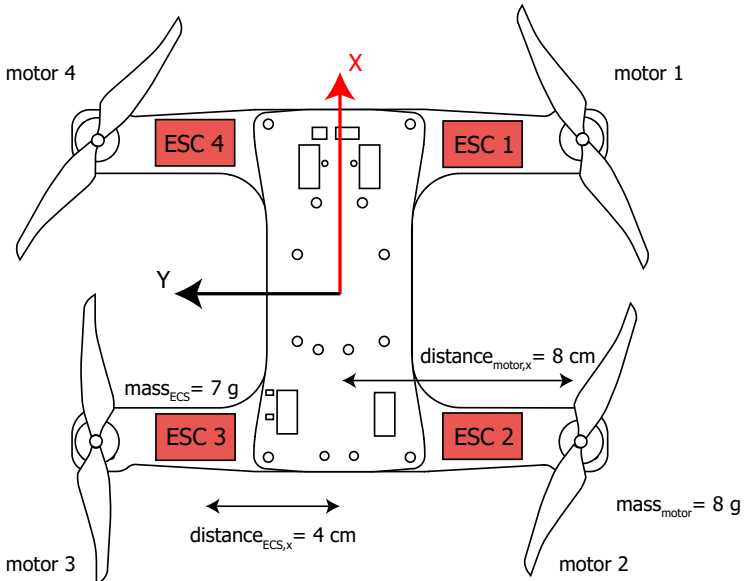
$$\frac{d}{dt} (\text{Rate}_{\text{roll}} [^\circ/s]) = 115 \cdot \text{input}_{\text{roll}}$$

For the simulation of the PID controller, you want to have your equations in the frequency domain rather than the time domain. You can transform the time domain to the frequency domain by performing a Laplace transform to both sides of the equation:

$$s \cdot \text{Rate}_{\text{roll}}[s] = 115 \cdot \text{input}_{\text{roll}}[s]$$

In this notation, s is not the unit of time, but a complex number with the form $s = \sigma + i \cdot \omega$. This gives you finally the relation between the quadcopter roll rate and the input command given by the PID controller, which is called your quadcopter transfer function:

$$\text{Rate}_{\text{roll}}[s] = \frac{115}{s} \cdot \text{input}_{\text{roll}}[s]$$



You can construct a fully similar reasoning for the pitch dynamics, eventually giving the following formula:

$$\text{Rate}_{\text{pitch}}[s] = \frac{115}{s} \cdot \text{input}_{\text{pitch}}[s]$$

Yaw motion

The same basic equation for the angular acceleration in the yaw direction holds as for the roll and pitch direction, with the only difference being the torque and moment of inertia, which should be calculated around the stationary z axis:

$$\text{Acceleration}_{\text{yaw}} [^\circ/s^2] = \frac{\text{Torque}_z [N \cdot m]}{I_z [kg \cdot m^2]} \cdot \frac{180}{\pi}$$

Torque around the z axis

The torque around the z axis is equal to the sum of all four motor torques; the motor torque is opposite to the rotation direction of each motor. When the motor is spinning in a clockwise direction, the force of the air pushed against the propellers creates a torque in the counter-clockwise direction. The total torque around the z axis is then equal to:

$$\text{Torque}_z [N \cdot m] = -\text{Torque}_{\text{motor } 1} + \text{Torque}_{\text{motor } 2} - \text{Torque}_{\text{motor } 3} + \text{Torque}_{\text{motor } 4}$$

The torque for each motor is related to the motor current through the torque coefficient K_T :

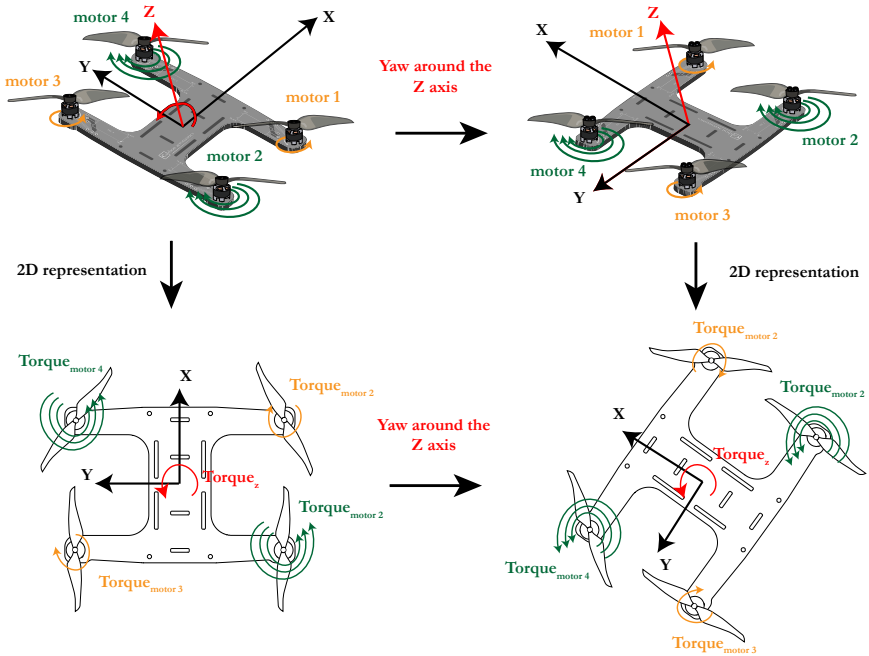
$$\text{Torque}_{\text{motor}} [N \cdot m] = K_T \left[\frac{N \cdot m}{A} \right] \cdot \text{Current} [A]$$

This torque coefficient K_T can be estimated using the kV rating of the motor, which is in our case equal to 5000 kV or 5000 rpm/V:

$$K_T = \frac{60}{2 \cdot \pi \cdot K_v} = \frac{60}{2 \cdot \pi \cdot 5000} = 0.0019 \frac{N \cdot m}{A}$$

You already determined that the current consumed by one motor is described by the formula below, which is already corrected for $\text{input}_{\text{yaw}}$ between 0 and 1000 μs instead of a throttle between 0 and 1:

$$\text{Current} [A] = 4.4 \cdot \frac{\text{input}_{\text{yaw}}[0 : 1000]}{1000} + 0.132$$



	Input _{Throttle}	Input _{Yaw}
output motor 1 = 25%	power = 50%	- 25%
output motor 2 = 75%	power = 50%	+ 25%
output motor 3 = 25%	power = 50%	- 25%
output motor 4 = 75%	power = 50%	+ 25%

The value of input_{yaw} is equal for all four motors, but has a positive sign for motors 2 and 4 and a negative sign for motors 1 and 3. The torque formula results in:

$$\text{Torque}_z [N \cdot m] = 0.0019 \cdot 4 \cdot \left(4.4 \cdot \frac{\text{input}_{yaw} [0 : 1000]}{1000} \right)$$

Moment of inertia around the z axis

The moment of inertia I_z describes the weight distribution of the quadcopter and its distance to the stationary z axis. In the two-dimensional representation, this gives:

$$I_z [kg \cdot m^2] = \sum_{i=1} [\text{mass}_i [kg] \cdot (\text{distance}_{\text{mass},i,x}^2 [m^2] + \text{distance}_{\text{mass},i,y}^2 [m^2])]$$

Notice that not only the distance of the mass to the x axis matters, but also the distance of the mass to the y axis. This is different compared to I_x , as the two-dimensional representation in that case meant that distance_{mass,i,z} is approximated as being zero. Once again only the components with a considerable mass that are far enough from the quadcopter X and Y axes are the motors and the ESCs.



This gives the following formula:

$$I_z = 4 \cdot \text{mass}_{\text{motor}} \cdot (\text{distance}_{\text{motor},x}^2 + \text{distance}_{\text{motor},y}^2) + 4 \cdot \text{mass}_{\text{ECS}} \cdot (\text{distance}_{\text{ESC},x}^2 + \text{distance}_{\text{ESC},y}^2)$$

Where the distance of the motors and the ESCs from the Y axis are equal to 5 cm.

Yaw dynamics

Introducing both formulas in the angular yaw acceleration formula gives:

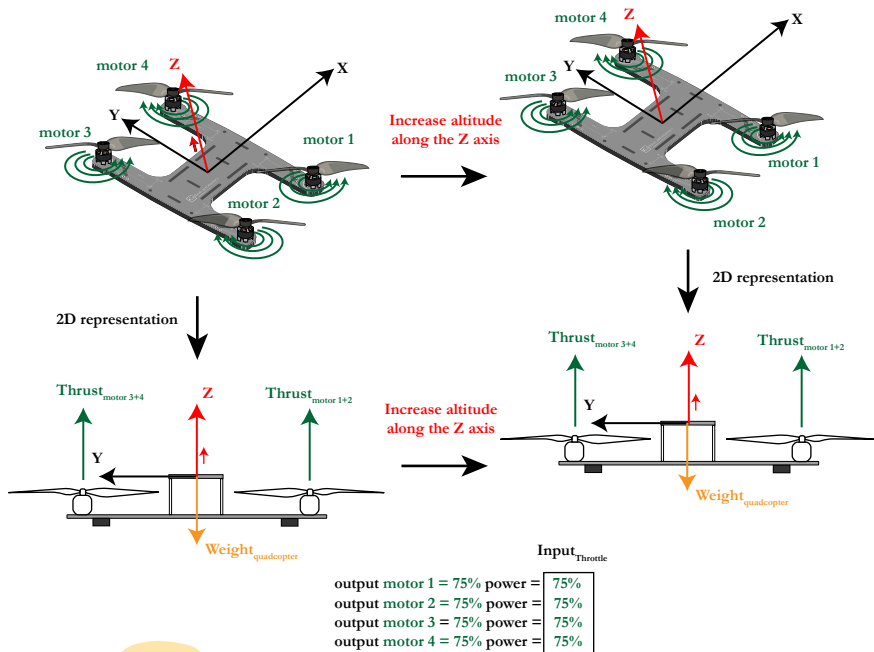
$$\text{Acceleration}_{\text{yaw}} [^\circ/s^2] = \frac{180}{\pi} \cdot \frac{0.0019 \cdot 4 \cdot \frac{4.4}{1000} \cdot \text{input}_{\text{yaw}}}{4 \cdot 0.008 \cdot (0.08^2 + 0.05^2) + 4 \cdot 0.007 \cdot (0.04^2 + 0.05^2)}$$

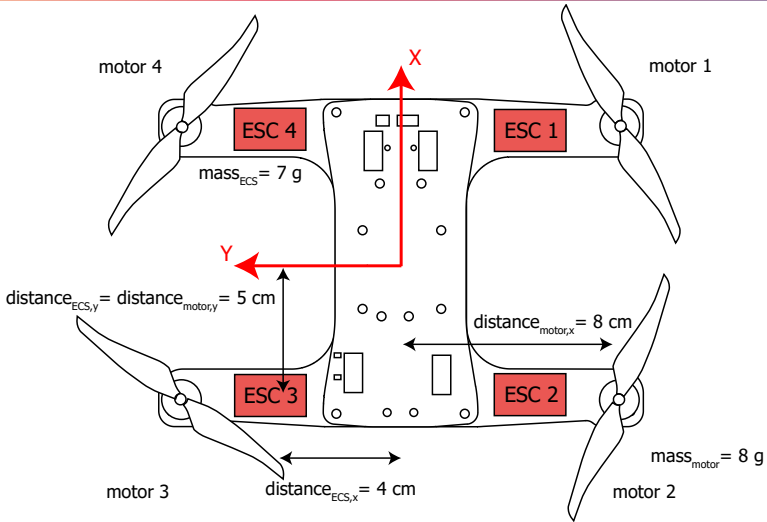
Which results again in a very simple formula for the angular acceleration in the time domain:

$$\text{Acceleration}_{\text{yaw}} [^\circ/s^2] = 4.8 \cdot \text{input}_{\text{yaw}}$$

Doing a similar Laplace transformation as before together with the integration of the acceleration results in the transfer function for the yaw dynamics:

$$\text{Rate}_{\text{Yaw}} [s] = \frac{4.8}{s} \cdot \text{input}_{\text{yaw}} [s]$$





Vertical velocity dynamics

The mathematical description of the quadcopter dynamics along the stationary Z axis is rather intuitive; the resulting acceleration along this axis is equal to the thrust delivered by the motors minus the gravity acting on the quadcopter mass. Write this force balance as:

$$\text{mass}_{\text{quadcopter}} [kg] \cdot \text{Acceleration}_z [m/s^2] = \text{thrust}_{\text{motor } 1+2+3+4} [N] - \text{mass}_{\text{quadcopter}} [kg] \cdot 9.81 m/s^2$$

And the thrust of all four motors was already calculated previously as:

$$\text{thrust}_{\text{motor } 1+2+3+4} [N] = 4 \cdot (0.160 \cdot \text{input}_{\text{throttle}} [0 : 1000]) \cdot \frac{9.81}{1000} = 0.0063 \cdot \text{input}_{\text{throttle}} [0 : 1000]$$

In the flight controller, you programmed $\text{Input}_{\text{throttle}}$ as $\text{InputThrottle} = 1500 + \text{PIDReturn}[0]$, because the zero velocity and thus the hover point will be at the point where the throttle stick is in the middle (1500 μs). This means that if you take this into account, the throttle stick at 1500 μs and gravity acting on the quadcopter mass cancel each other out. Knowing that the quadcopter mass is equal to 250 g, the dynamic equation of the acceleration along the z axis becomes:

$$\begin{aligned} 0.250 [kg] \cdot \text{Acceleration}_z [m/s^2] &= 0.0063 \cdot \text{input}_{\text{throttle}} [0 : 1000] \\ \text{Acceleration}_z [m/s^2] &= 0.025 \cdot \text{input}_{\text{throttle}} \end{aligned}$$

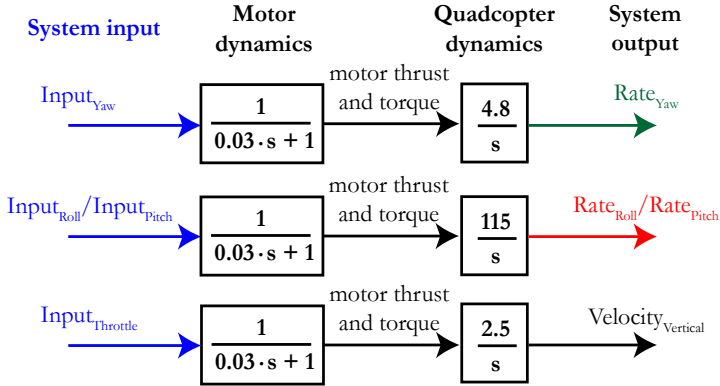
When coding the flight controller, you did not measure the vertical velocity in m/s but rather in cm/s. Just multiply the equation by 100 cm/m to get the correct units:

$$\text{Acceleration}_z [cm/s^2] = 2.5 \cdot \text{input}_{\text{throttle}}$$

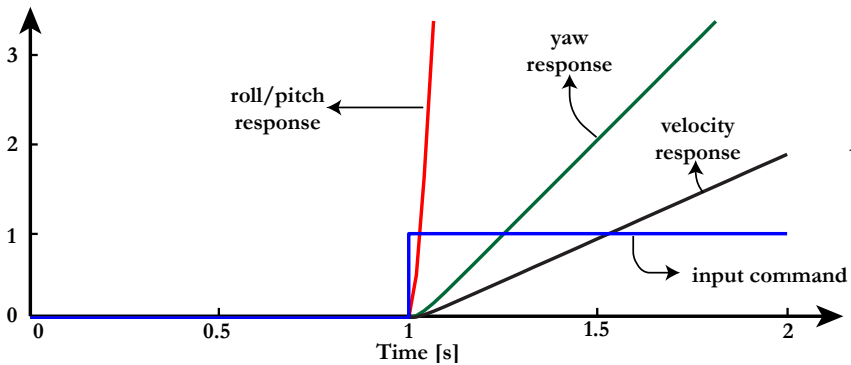


Doing the same Laplace transformation as before together with the integration of the acceleration results in the transfer function of the vertical velocity dynamics:

$$\text{Velocity}_{\text{Vertical}} [s] = \frac{2.5}{s} \cdot \text{input}_{\text{throttle}}[s]$$



Roll/Pitch/Yaw rate [°/s]
Vertical velocity [cm/s]



Open loop system response

You have now characterized the quadcopter dynamics for the roll, pitch, yaw and vertical velocity movements. Combined with the motor dynamics, you can simulate the response of the system to a command in the time domain (usually done with dedicated software). This is called the open loop system response and is visualised in the diagram to the left. The quadcopter dynamics for all movements are summarized below:

$$\text{Rate}_{\text{Roll}}(s) = \frac{115}{s} \cdot \text{Input}_{\text{Roll}}(s)$$

$$\text{Rate}_{\text{Pitch}}(s) = \frac{115}{s} \cdot \text{Input}_{\text{Pitch}}(s)$$

$$\text{Rate}_{\text{Yaw}}(s) = \frac{4.8}{s} \cdot \text{Input}_{\text{Yaw}}(s)$$

$$\text{Velocity}_{\text{Vertical}}(s) = \frac{2.5}{s} \cdot \text{Input}_{\text{Throttle}}(s)$$

The value of the nominator is the only part that changes for each of the quadcopter dynamics. When you simulate an input step response in the time domain and evaluate the so called open loop system response, the system output gives an indication of the inherent (in)stability for all four movements. The results are visualized in the figure to the left. You can see that the roll and pitch response happens much faster than the velocity response. This depends off course on the value of the nominator in the quadcopter dynamics transfer function: a large nominator (115) causes a much faster response than a small nominator (2.5). The quadcopter dynamics are an example of a meta-stable system, as control theory requires the poles of the system to be in the left half plane (e.g. negative). Since the pole of the system lies in this case exactly in the origin ($s=0$), it means that the system is meta-stable; the system will continuously increase at the same rate in an open loop. The pole of the motor dynamics is equal to $0.03s+1=0$ or $s=-33$ and thus very stable, meaning it does not have an impact on the open loop system as a whole.

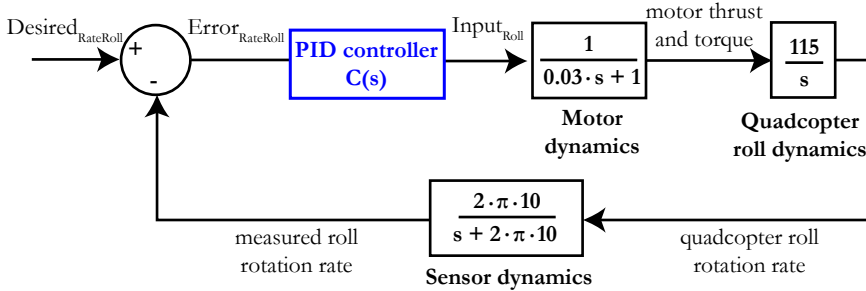
Physically, the behaviour of the open loop system means that you can control the vertical velocity of the quadcopter manually and you do not necessarily need a control loop. The first rate controller you developed did not have vertical velocity control yet you were able to control the altitude of the quadcopter pretty well by constantly adjusting the throttle. It is a different story with the roll, pitch and yaw rate; trying to manually control these rates is practically impossible for the reaction times of a human being. That is why the rate mode flight controller is the minimal controller you need to fly your quadcopter. The mathematical development of this controller that automatically stabilizes your quadcopter will be explained in the next project.





Project 23

Quadcopter PID controller



The integral term

A more difficult transformation is the transformation of the integral term; the bilinear transformation is used together with the property of the z-transform that $a(k-1)$ in the discrete domain is equivalent to $z^{-1} \cdot a(z)$ in the z-domain:

$$I_{\text{term}}(k) = I_{\text{term}}(k-1) + I \cdot (\text{Error}(k) + \text{Error}(k-1)) \cdot \frac{T_s}{2}$$

z-transform

$$I_{\text{term}}(z) = z^{-1} \cdot I_{\text{term}}(z) + I \cdot (\text{Error}(z) + z^{-1} \cdot \text{Error}(z)) \cdot \frac{T_s}{2}$$

$$(1 - z^{-1}) \cdot I_{\text{term}}(z) = I \cdot (1 + z^{-1}) \cdot \text{Error}(z) \cdot \frac{T_s}{2}$$

$$\frac{z-1}{z} \cdot I_{\text{term}}(z) = I \cdot \frac{z+1}{z} \cdot \text{Error}(z) \cdot \frac{T_s}{2}$$

$$I_{\text{term}}(z) = I \cdot \frac{T_s}{2} \cdot \frac{z+1}{z-1} \cdot \text{Error}(z)$$

bilinear transform

$$I_{\text{term}}(s) = I \cdot \frac{T_s}{2} \cdot \frac{\left(\frac{1+s \cdot \frac{T_s}{2}}{1-s \cdot \frac{T_s}{2}} + 1\right)}{\left(\frac{1+s \cdot \frac{T_s}{2}}{1-s \cdot \frac{T_s}{2}} - 1\right)} \cdot \text{Error}(s)$$

$$I_{\text{term}}(s) = I \cdot \frac{T_s}{2} \cdot \frac{1+s \cdot \frac{T_s}{2} + 1-s \cdot \frac{T_s}{2}}{1+s \cdot \frac{T_s}{2} - 1+s \cdot \frac{T_s}{2}} \cdot \text{Error}(s)$$

Design your quadcopter controller

During the previous project, you have characterized the quadcopter dynamics. The goal of this characterization is the design of a mathematical controller that is capable of stabilizing your quadcopter. The PID controller that you used for your flight code will be designed in this project.

Your goal is once again to construct a transfer function for the controller that you want to use. Remember that a PID controller consists of three terms:

$$\text{motor input}(k) = P_{\text{term}}(k) + I_{\text{term}}(k) + D_{\text{term}}(k)$$

which, in the discrete time domain of your flight controller program (where each k is a new iteration that takes $T_s = 0.004$ s or 250 Hz), translates to:

$$\text{motor input}(k) = P \cdot \text{Error}(k) + I_{\text{term}}(k-1) + I \cdot (\text{Error}(k) + \text{Error}(k-1)) \cdot \frac{T_s}{2} + D \cdot \left(\frac{\text{Error}(k) - \text{Error}(k-1)}{T_s} \right)$$

Now you want to transform this discrete representation to the s -domain. Such a transformation is only possible by first transforming the discrete time representation to the frequency domain using the z -transform. Next the z -domain will be transformed to the s -domain using the bilinear or Tustin transformation:

$$z = e^{-s \cdot T_s} = \frac{1 + s \cdot \frac{T_s}{2}}{1 - s \cdot \frac{T_s}{2}}$$

The proportional term

The easiest transformation is the one of the proportional term, as it requires only the transformation of the in- and outputs to the z and s domains:

$$P_{\text{term}}(k) = P \cdot \text{Error}(k)$$

z -transform

$$P_{\text{term}}(z) = P \cdot \text{Error}(z)$$

bilinear transformation

$$P_{\text{term}}(s) = P \cdot \text{Error}(s)$$

As expected, the result is just a constant P multiplied with the error. Let's continue with the integral term.

$$I_{\text{term}}(s) = I \cdot \frac{T_s}{2} \cdot \frac{2}{2 \cdot s \cdot \frac{T_s}{2}}$$

$$I_{\text{term}}(s) = I \cdot \frac{1}{s}$$

The derivative term

The transformation of the derivative term from the discrete to the s-domain is very similar to the work you have already carried out for the integral term:

$$D_{\text{term}}(s) = D \cdot \frac{\text{Error}(k) - \text{Error}(k-1)}{T_s}$$

z-transform

$$D_{\text{term}}(z) = D \cdot \frac{\text{Error}(z) - z^{-1} \cdot \text{Error}(z)}{T_s}$$

$$D_{\text{term}}(z) = D \cdot \frac{1}{T_s} \cdot \frac{z-1}{z} \cdot \text{Error}(z)$$

bilinear transform

$$D_{\text{term}}(s) = D \cdot \frac{1}{T_s} \cdot \frac{\left(\frac{1+s \cdot \frac{T_s}{2}}{1-s \cdot \frac{T_s}{2}} - 1\right)}{\left(\frac{1+s \cdot \frac{T_s}{2}}{1-s \cdot \frac{T_s}{2}}\right)} \cdot \text{Error}(s)$$

$$D_{\text{term}}(s) = D \cdot \frac{1}{T_s} \cdot \frac{1+s \cdot \frac{T_s}{2} - 1 + s \cdot \frac{T_s}{2}}{1+s \cdot \frac{T_s}{2}} \cdot \text{Error}(s)$$

$$D_{\text{term}}(s) = D \cdot \frac{1}{T_s} \cdot \frac{2 \cdot s \cdot \frac{T_s}{2}}{1+s \cdot \frac{T_s}{2}} \cdot \text{Error}(s)$$

$$D_{\text{term}}(s) = D \cdot \frac{s}{1+s \cdot \frac{T_s}{2}} \cdot \text{Error}(s)$$

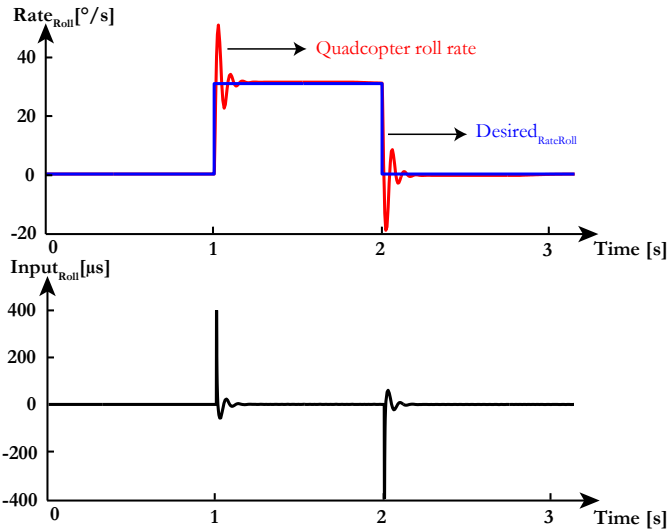
When taking into account that T_s is equal to 0.004 seconds, the transport function for the derivative term becomes:

$$D_{\text{term}}(s) = D \cdot \frac{s}{1+s \cdot 0.002} \cdot \text{Error}(s)$$

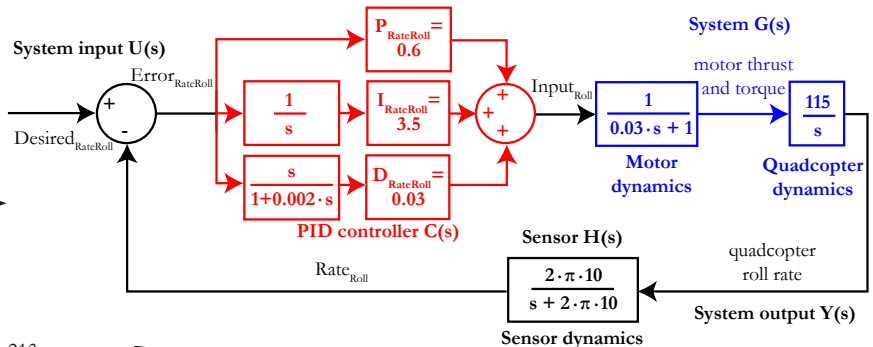
Closed control loop response

During the previous projects you learned how to describe the quadcopter, motor and sensor dynamics in a mathematical way, subsequently transforming them to the frequency domain. Now that you can describe the PID controller as well, you are able to construct the full control loop. The figure on the right shows the full quadcopter rate control loop, with the P, I and D values that you used for your rate mode flight controller. For the pitch and yaw rate loop, the figure would be exactly the same, only the PID values and the quadcopter dynamics transfer function would change. You can simulate the time-domain response of the full closed control loop with dedicated software.

Let's assume your desired roll rate changes from 0 to $30^\circ/\text{s}$ and then back to 0 in respectively one second. The resulting quadcopter roll rate for the closed control loop is displayed on the figure below. At each step change, the quadcopter roll rate overshoots the desired roll rate with almost $20^\circ/\text{s}$, but within half a second it is stable again. When looking at the commands sent to the motor, they follow the desired roll rate also closely. Notice that the commands saturate at $\pm 400 \mu\text{s}$; this was already programmed in the flight controller and can additionally be simulated as well.



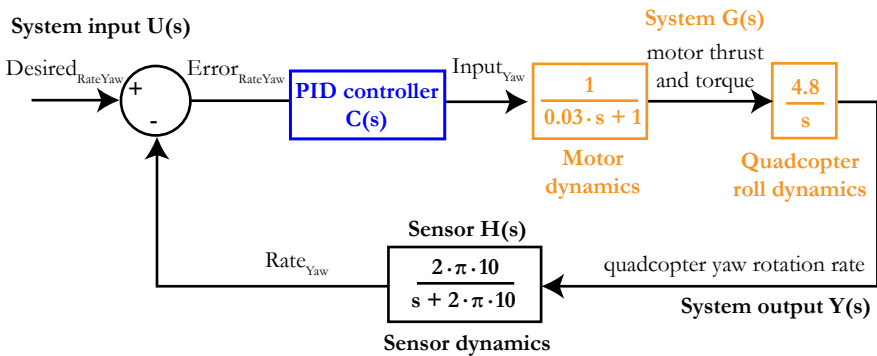
With the mathematical representation of the full control system, it is possible to optimize the response of the system by adjusting the PID values. In reality, the system is first simulated using the mathematical representation, then the optimal PID values are chosen given a desired system response. Because the mathematical representation is an approximation of the real physical system, further tuning of the PID values will always be necessary when testing your quadcopter.



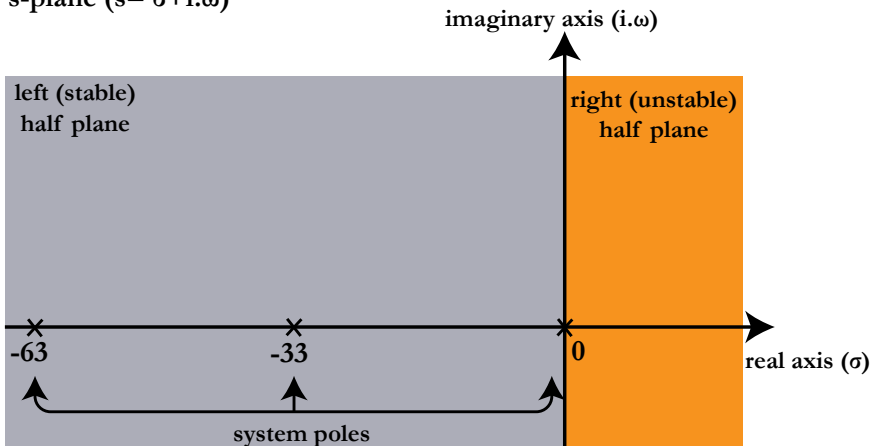


Project 24

Estimate the PID values



s-plane ($s = \sigma + i \cdot \omega$)



Tune your controller for a smooth flight

You are now capable to mathematically simulate the full system and control loop. The goal of this simulation is to 'tune' your PID controller in order to get a stable flight. In essence, you need to determine the best values for the P, I and D parameters.

Once you have the mathematical representation for the system and the sensor, a first estimation of the PID values for the controller is usually calculated using dedicated software. However, the optimal values can also be calculated by hand using the root locus method. In this project you will derive an estimation of the PID values for the yaw rate controller. A similar approach can be followed for the other controllers.

The closed loop transfer function of the full yaw rate controller can be mathematically constructed using the figure on the left: the output of the system $Y(s)$ is related to the system input $U(s)$ through:

$$Y(s) = C(s) \cdot G(s) \cdot [U(s) - H(s) \cdot Y(s)]$$

Rewriting the equation such that the system output $Y(s)$ is isolated gives:

$$Y(s) = C(s) \cdot \frac{G(s)}{1 + C(s) \cdot G(s) \cdot H(s)} \cdot U(s)$$

The closed-loop poles s^* of this system are the zeros of the denominator:

$$1 + C(s^*) \cdot G(s^*) \cdot H(s^*) = 0$$

$$C(s^*) \cdot G(s^*) \cdot H(s^*) = -1$$

$G(s)$ and $H(s)$ are determined by the physical system and cannot be chosen by you (considering the quadcopter design remains fixed that is). But you have full control over the controller $C(s)$ to stabilize the system; strictly speaking it does not even have to be a PID controller. Let's first look at the characteristics of the physical system:

$$G(s) \cdot H(s) = \frac{1}{0.03 \cdot s + 1} \cdot \frac{4.8}{s} \cdot \frac{2 \cdot \pi \cdot 10}{s + 2 \cdot \pi \cdot 10}$$

The system contains three poles:

$$s = 0$$

$$s = \frac{-1}{0.03} = -33$$

$$s = -2 \cdot \pi \cdot 10 = -63$$

The pole at zero is the only one that makes the system meta-stable, as the other poles are situated in the stable left half-plane far from the imaginary axis. To stabilize the system, you will have to add a zero near the origin to cancel out the pole. This means that you need a controller of the type:

$$C(s) = K \cdot (s + b) \text{ with } b \text{ near zero}$$

To make sure you do not have a steady-state error in the response, you put another pole at zero to finally obtain a controller of the PI type:

$$C(s) = K \cdot \frac{s + b}{s}$$

With b and K the parameters that you will calculate using the root locus method. The full open loop system now becomes:

$$C(s) \cdot G(s) \cdot H(s) = \frac{1}{0.03 \cdot s + 1} \cdot \frac{4.8}{s} \cdot \frac{2 \cdot \pi \cdot 10}{s + 2 \cdot \pi \cdot 10} \cdot K \cdot \frac{s + b}{s}$$

This is a fourth order system, because the highest power of s in the denominator is s^4 when you multiply all fractions. However, as b will be close to zero, it will help to cancel out the two poles at zero. This means that the system will behave as a second order system. The step response of a general second order system can be characterized by two parameters: the maximal overshoot and the settling time, which is defined as the time at which the response goes to within 2% of the desired value. Let's say that you want the system to settle within 0.5 seconds with a maximal overshoot of 10%:

- overshoot (OS)=10%
- settling time (t_{settling})=0.5 s

The damping ratio ζ for a second order system is defined by:

$$\xi = \frac{-\ln\left(\frac{OS}{100}\right)}{\sqrt{\pi^2 + \ln\left(\frac{OS}{100}\right)^2}} = \frac{-\ln\left(\frac{10}{100}\right)}{\sqrt{\pi^2 + \ln\left(\frac{10}{100}\right)^2}} = 0.59$$

You already defined the settling time as the time necessary to reach 2% (=0.02) of the desired response. This is related to the damping ratio through the formula:

$$t_{\text{settling}} = \frac{-\ln(0.02)}{\xi \cdot \omega_n}$$

Where ω_n is the natural frequency of the second-order system. Calculate the natural frequency by inverting the above formula:

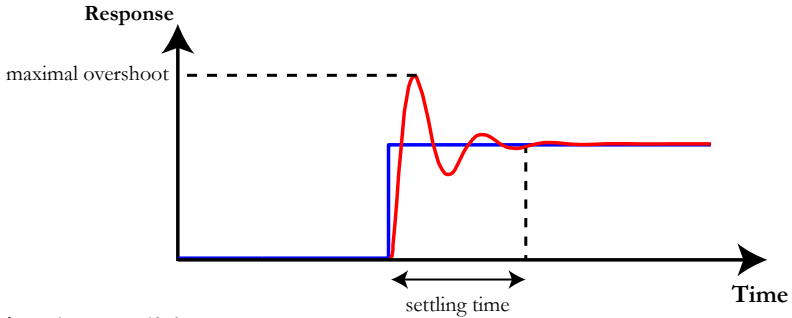
$$\omega_n = \frac{-\ln(0.02)}{0.59 \cdot 0.5 \text{ s}} = 13 \text{ rad/s}$$

The desired poles of your full system are then equal to:

$$P_{\text{desired}} = -\omega_n \cdot \xi \pm i \cdot \omega_n \cdot \sqrt{1 - \xi^2}$$

$$P_{\text{desired}} = -7.8 \pm i \cdot 10.7$$

The b and K parameters will be determined using the angle and magnitude conditions of the root locus method.



Angle condition

The angle condition of the root locus says that the sum of the angles of the open loop poles minus the sum of the angles of the open loop zeros has to be equal to 180° :

$$\sum \theta_{poles} - \sum \theta_{zeros} = 180^\circ$$

Remember that your system contained the four poles and you want to add a zero at b. The angle condition becomes:

$$\theta_2 + \theta_3 + \theta_4 + \theta_5 - \theta_1 = 180^\circ$$

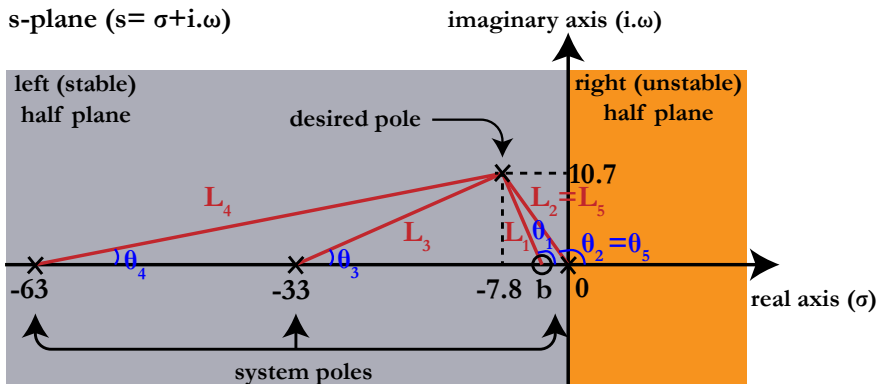
Using basic trigonometry, the angles can be determined (see figure below):

$$\tan(180^\circ - \theta_2) = \frac{10.7}{7.8} \rightarrow \theta_2 = \theta_5 = 126^\circ$$

$$\tan(\theta_3) = \frac{10.7}{33 - 7.8} \rightarrow \theta_3 = 23^\circ$$

$$\tan(\theta_4) = \frac{10.7}{63 - 7.8} \rightarrow \theta_4 = 11^\circ$$

s-plane ($s = \sigma + i\omega$)



This gives for θ_1 :

$$\theta_1 = 126^\circ + 23^\circ + 11^\circ + 126^\circ - 180^\circ = 106^\circ$$

From which you can calculate the value for parameter b, again using trigonometry and the figure on the previous page:

$$\begin{aligned} \tan(180^\circ - \theta_1) &= \frac{10.7}{7.8 - b} \\ \tan(180^\circ - 106^\circ) &= \frac{10.7}{7.8 - b} \rightarrow b = 4.7 \end{aligned}$$

Your PID controller becomes equal to $C(s) = K(s - 4.7)/s$.

Magnitude condition

You will now determine the final parameter K using the magnitude condition of the root locus:

$$K_{\text{global}} = \frac{\prod L_{\text{poles}}}{\prod L_{\text{zeros}}}$$

In which K_{global} comprises not only the parameter K, but the full amplification throughout the open loop system:

$$\begin{aligned} C(s) \cdot G(s) \cdot H(s) &= \frac{1}{0.03 \cdot s + 1} \cdot \frac{4.8}{s} \cdot \frac{2 \cdot \pi \cdot 10}{s + 2 \cdot \pi \cdot 10} \cdot K \cdot \frac{s + b}{s} \\ C(s) \cdot G(s) \cdot H(s) &= \frac{\frac{1}{0.03} \cdot 4.8 \cdot 2 \cdot \pi \cdot 10}{s + \frac{1}{0.03}} \cdot \frac{s + b}{s} \cdot K \end{aligned}$$

$= K_{\text{global}}$

Meaning that:

$$K_{\text{global}} = K \cdot \frac{1}{0.03} \cdot 4.8 \cdot 2 \cdot \pi \cdot 10$$

The magnitude condition of the root locus is then equal to:

$$\begin{aligned} 10053 \cdot K &= \frac{L_2 \cdot L_3 \cdot L_4 \cdot L_5}{L_1} \\ K &= \frac{\sqrt{7.8^2 + 10.7^2} \cdot \sqrt{(33 - 7.8)^2 + 10.7^2} \cdot \sqrt{(63 - 7.8)^2 + 10.7^2} \cdot \sqrt{7.8^2 + 10.7^2}}{10053 \cdot \sqrt{(7.8 - 4.7)^2 + 10.7^2}} \end{aligned}$$

This gives $K = 2.4$ and now you finally know the parameters of your PI controller:

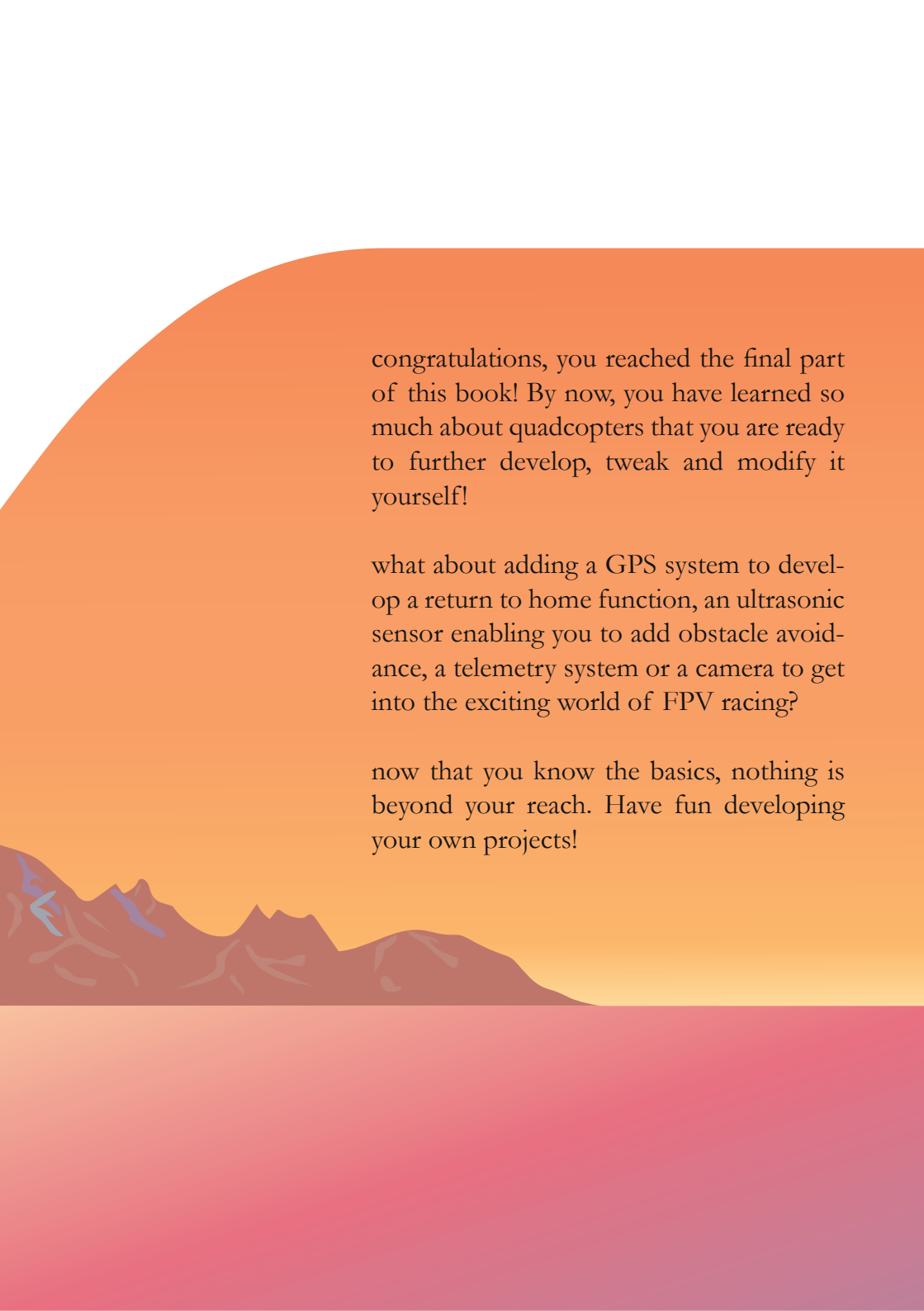
$$\begin{aligned} C(s) &= 2.4 \cdot \frac{s + 4.7}{s} \\ C(s) &= 2.4 + \frac{11.3}{s} \end{aligned}$$

The PI values for the yaw rate you used in your flight controller are equal to 2 and 12, which is very close to your calculated estimation of 2.4 and 11.2. In practice, you will choose 2.4 and 11.2 as initial values when testing your flight controller then slightly tune the parameters to further improve the handling of your quadcopter.



Part V: expanding your horizon





congratulations, you reached the final part of this book! By now, you have learned so much about quadcopters that you are ready to further develop, tweak and modify it yourself!

what about adding a GPS system to develop a return to home function, an ultrasonic sensor enabling you to add obstacle avoidance, a telemetry system or a camera to get into the exciting world of FPV racing?

now that you know the basics, nothing is beyond your reach. Have fun developing your own projects!

This manual helps you to develop, program and construct your own quadcopter with the help of 24 small projects, explaining the essentials on aeronautics, electronics and embedded programming along the way.

All components and code used in this manual are fully hackable and adaptable, giving you the opportunity to create your own unique quadcopter.

Ca rbon aeronautics
