PID Regulated Balancing Cube

Thesis WIP

SEBASTIAN BRANDMAIER, DENIS RAMSDEN

Bachelor's Thesis at ITM
Supervisor: Nihad Subasic
Examiner: Nihad Subasic

TRITA-ITM-EX 2020:33

Abstract

This projects purpose were to do a research of a field using mechanical parts, electronics and coding given a limited set of resources. In this project a balancing cube were constructed with the intent of it balancing on of one of its edges using a PID-controller to counteract the gravity pulling it down. To be able to this a good understanding of the system was required to have the right components for the job. A DC motor were used with two reaction wheel on each side of its axle using the moment of inertia to convert the torque from the motor to a angle acceleration of the cube. An IMU was used to determine the angle difference of the centre of mass from a fixed point in the room and this value were used as an input in the PID-controller. Different method were used to determine the P, I and D components of the PID-controller to create a stable system. Even though different methods were used no one successfully made the cube balance perfectly.

Keywords: Mechatronics, PID-control, reaction wheel, Arduino, cube

Referat

PID-regulerande Balanserande Kub

Syftet med detta projekt är att undersöka olika forskningsfrågor med hjälp av att använda mekaniska delar, elektronik och kod med en begränsad mängd resurser. Detta projekt består av att en balanserande kub var konstruerad där tanken var att den ska balansera på en av sina kanter med hjälp av en PID-kontroll för att motverka sin egen tyngd. För att göra detta behövdes en bra förståelse för systemets uppbyggnad för att kunna införskaffa sig rätt komponenter för projektet. En DC motor användes för detta ändamål och den använde två reaktionshjul på vardera sida av axeln och använde tröghetsmomentet från dessa för att överföra momentet från motorn till en vinkelacceleration på kuben. En IMU användes för att mäta vinkeländringen av masscentrumet från en fixt punk i rummet och detta värde användes som input i PID-kontrollen. Olika metoder användes för att bestämma P, I och D komponenterna av PID-kontrollen som gav ett stabilt system. Trots att olika metoder användes lyckades ingen av dessa att få kuben att balansera felfritt.

Nyckelord: Mekatronik, PID-kontroll, reaktionshjul, Arduino, kub

Acknowledgements

We would like to thank our Seshagopalan Thorapalli Muralidharan that has help us to manufacture all the necessary parts and guided us through this project.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Purpose . . . 1
- 1.3 Scope . . . 1
- 1.4 Method . . . 2

2 Theory **3**
- 2.1 System dynamics . . . 3
  - 2.1.1 Reaction wheel . . . 4
  - 2.1.2 PID-controller . . . 4
- 2.2 Components . . . 5
  - 2.2.1 Motors . . . 5
  - 2.2.2 Sensors . . . 6
  - 2.2.3 Arduino . . . 7
  - 2.2.4 H-bridge . . . 7

3 Design **9**
- 3.1 Implementations of components . . . 9
  - 3.1.1 Gyroscope . . . 9
  - 3.1.2 Hall effect sensor . . . 9
  - 3.1.3 Motor driver . . . 10
- 3.2 Shell . . . 10
- 3.3 Reaction wheel . . . 12
- 3.4 Measure of motor speed and direction . . . 13
- 3.5 PID-controller . . . 14
- 3.6 Code . . . 15

4 Results **17**
- 4.1 IMU . . . 17
- 4.2 Motor . . . 18
  - 4.2.1 PID . . . 18

5 Discussion **19**

Bibliography **21**

Appendices **22**

A Electric scheme chart **24**

B Datasheet for the motor **25**

C Source code **27**
C.1 Motor experiment in Matlab 27
C.2 Arduino code in C++ 28

List of Figures

2.1 Gravitational force acting on the cube in horizontal position. Made in Solid Edge . . . 3
2.2 Angular velocity over time when a voltage of 24V is applied on the motor. Made in Matlab . . . 6
2.3 A schematic image of a H-bridge. Source: `https://www.build-electronic-circuits.com/wp-content/uploads/2018/11/H-bridge-switches.png` . . . 8

3.1 MPU-6050 - Gyroscope/Accelerometer. Source: `https://www.electrokit.com/uploads/productimage/41016/41016233.jpg` . . . 9
3.2 Hall effect sensor. Source: `https://www.electrokit.com/uploads/productimage/41015/41015710.jpg` . . . 10
3.3 Motor driver VNH3SP30 Source: `https://www.electrokit.com/uploads/productimage/41014/41014059.jpg` . . . 10
3.4 A Solid Edge CAD that shows the cube shell with reaction wheels and motor installed. Made in Solid Edge . . . 11
3.5 Section view from the side of cube in Solid Edge CAD showing the motor holders. Made in Solid Edge . . . 12
3.6 Reaction wheel from the side. Made in Solid Edge . . . 13
3.7 Sensors for measuring wheel speed. Made in Pixlr Editor . . . 14
3.8 Flowchart for the balancing cube. Made in draw.io. . . . 16

4.1 Measured values from the IMU over time with and without a capacitor. Made in Matlab. . . . 17
4.2 Comparison of theoretical rotating speed from *equation 2.12* and measured values. Made in Matlab. . . . 18

5.1 Comparison of theoretical rotating speed from *equation 2.12* and measured values, adjusted moment of inertia. Made in Matlab . . . 20

Chapter 1

Introduction

1.1 Background

The project we have chosen is to construct a cube that can balance on one of its edges. The idea is to make it balance by rotating a reaction wheel with a motor creating a torque with the moment of inertia making the cube counteracts its own weight. The acceleration of the wheels is then adjusted relative to the angle between the cube and the horizontal plane to keep the cube stable.

1.2 Purpose

We were inspired by the fact that the design of the physical product seems very simple, but in reality it is based on a complex control theory problem. Although the final product will not solve any everyday problems, the technology is important as it provides insight into how control theory can be applied in real projects. The experience of controlling motors and handling signals can benefit later projects since similar problems exists in many areas. Examples of areas where this technology is useful are for designing segways, rockets and aircrafts etc. This study will mainly focus on answering following questions:

- What is a convenient way to construct a cube with the necessary hardware to balance on an edge using a reaction wheel?
- Is it possible with the resources available to balance the cube with a PID controller and what tuning method should be used?

1.3 Scope

The focus of this project is to design and construct a fully functional prototype of a product. This regarding a a budget of 1000 SEK and a project deadline of three months. Other control systems could work for the same control problem but in this project a PID controlled system will be analyzed.

1.4 Method

A physical Cube will be constructed to make experiment to be able to answer the study questions and all construction of the necessary components will take place at the KTH Prototype Center. Arduino uno will be used as a development platform in this project to control the components. Mathematical models of the system will be made to be able to analyze it and create a PID control system making the cube balance. After that the components for the PID will be tuned until the system is fully stable.

Chapter 2

Theory

2.1 System dynamics

To be able to stabilize the cube on its corner a relationship between the different affecting factors is needed and a model of the system dynamics is set up. M_{max} is the maximum torque needed to counteract the construction´s own weight at a horizontal position and is given by equation 2.1 where a mechanical equilibrium gives this relationship between the cube's mass m, gravitation g and the horizontal length to its center of gravity $\frac{l}{2}$ according to

$$M_{max} = mg\frac{l}{2}. \tag{2.1}$$

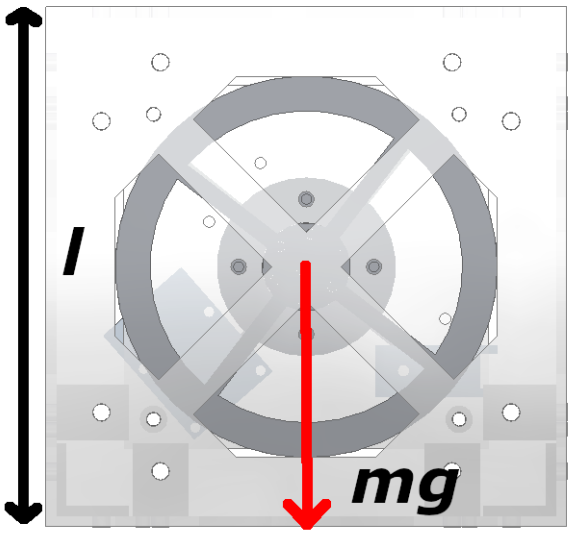


Figure 2.1. Gravitational force acting on the cube in horizontal position. Made in Solid Edge

This sets the requirement for needed torque from the motor. An important aspect is to make the construction mechanically stable because the mechanical stability will set a fundamental stability to the construction that gives a good starting point for the automatic control later. This is done by letting the center of mass being centered in the middle of the cube.

2.1.1 Reaction wheel

The most suitable shape of a reaction wheel is a ring because it provides the greatest moment of inertia per mass that will result in a decreased acceleration. The theoretical moment of inertia for a ring can be calculated with the ring´s mass m and it´s radius r as [2]

$$I_{ring} = mr^2. \tag{2.2}$$

When the motor accelerate the reaction wheel generates torque. Due to Newtons third law, a reaction torque that acts on the cube in the opposite direction is also generated. This can be described by the equation below with I_{ring} as the wheels theoretical moment of inertia and the angle acceleration $\dot{\omega}_{motor}$ of it as [2]

$$M_{motor} = I_{ring}\dot{\omega}_{motor}. \tag{2.3}$$

By adjusting the speed and acceleration of the motor the angle between the cube and the surface can be controlled. In this project a Newtonian model was used that can be described as [2]

$$I_{tot}\ddot{\theta} = mglsin(\theta) - M_{motor} \tag{2.4}$$

2.1.2 PID-controller

A PID-controller is the most commonly used controller in the industry. The controller use a reference point that is the value it want to reach and compare it with its current value and this difference is the error. It then uses the error to stabilize the system with three components. PID means that the controller consists of three components: a proportional, an integrative and a derivative component. The P-component is used to adjust the gain, the I-component is used to reduce the static error, and the D-component compensates for changes in the system that increases the stability. [8]
Since arduino includes a built in a PID-function within its libraries it will be used in this project. The PID-function has three parameters K_P, K_I, and K_D that will be choosen experimentally. [3] The ideal PID-regulator can be written as:

$$u(t) = K_P e(t) + K_I \int_{t_0}^{t} e(\tau)d\tau + K_D \frac{de(t)}{dt}. \tag{2.5}$$

There is different ways a PID-controller can be tuned to stabilize a system. One method is the Ziegler-Nicholas method where analysis of the system could give a

good approximation of the three components creating the PID-controller. Given a system the simplest method is to increase the gain for the system until a steady state oscillation occurs, giving the K_{cr}. This with the given oscillation period P_{cr} can with this method give a good starting point for tuning the PID. The method suggests the following values in relation with the analysed valued above: [6]

$$K_P = 0.6K_{cr}, \tag{2.6}$$

$$T_I = 0.5P_{cr}, \tag{2.7}$$

$$T_D = 0.125P_{cr}. \tag{2.8}$$

Using equation 2.5 the PID-controller can now be written as

$$u(t) = K_P(e(t) + \frac{1}{T_I}\int_{t_0}^{t} e(\tau)d\tau + T_D\frac{de(t)}{dt)}. \tag{2.9}$$

2.2 Components

2.2.1 Motors

In this project a brushed DC motor was used. A brushed motor means that it rotates internally. DC stands for direct current which implies that ohm's law and Kirchhoff's voltage law can be used in calculations. The relationship between voltage, current and velocity can therefore be describe by [9]

$$U_A = R_A I_A + K_2\Phi\omega. \tag{2.10}$$

The torque that the motor generates is proportional to the current and is mathematically described as [9]

$$M = K_2\Phi I_A. \tag{2.11}$$

Combining equation 2.5 and 2.6 results in that the total motor torque is [9]

$$M = \frac{K_2\Phi}{R_A}U - \frac{K_2\Phi^2\omega}{R_A}. \tag{2.12}$$

$K_2\Phi$ is a constant that can be calculated by measuring the speed of the rotor shaft when there is no load on the motor. This gives that the voltage is direct proportional to the rotating speed ($U_A = K_2\Phi\omega$). In this project it was instead found in the data sheet for the motor. R_a is the terminal resistance can be measured, but was also found in the data sheet. [9]

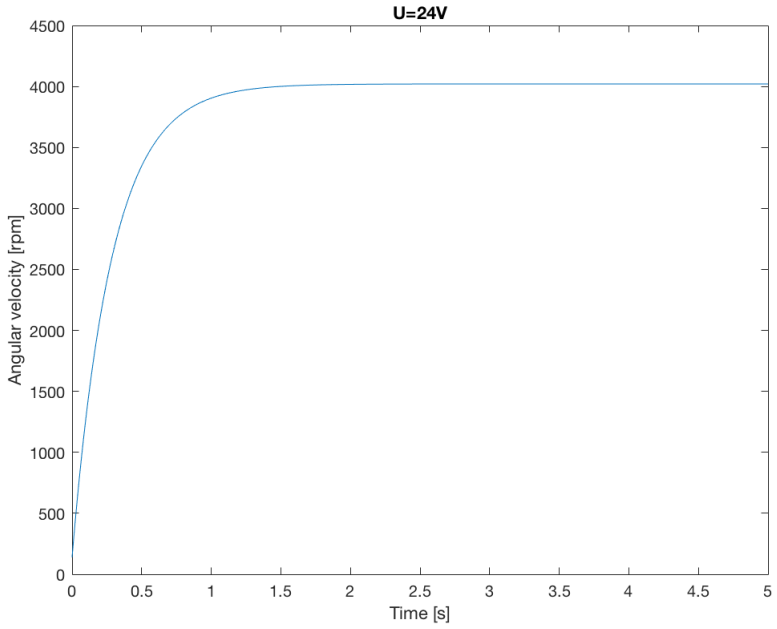


Figure 2.2. Angular velocity over time when a voltage of 24V is applied on the motor. Made in Matlab

The equation 2.7 and 2.3 was used to plot how the angular velocity of the flywheel changes over time when the voltage 24V is applied on the motor. The angular velocity was calculated with the Euler forward method and depends on the torque that was received from equation 2.7. In the data sheet the ”no load speed” for the motor is 3900 rpm. The reason why the motor passes this speed despite there is load on the motor might be because the used model does not consider internal friction of the motor. A theoretical value of the load was calculated by approximating the flywheel as a ring. The moment of inertia could the be calculated with equation 2.2.

2.2.2 Sensors

Hall effect sensor

To be able to control the angular velocity of the reaction wheels a Hall effect sensor is used. The sensor can sense a magnetic field due to the disturbance of electrons and sends a signal to the micro controller when it sense one. If a magnet is placed on the reaction wheel the hall effect sensor will sense a disturbance for every spin of the wheel. With that the micro controller can give us an accurate angular velocity. With two hall effect sensors it is possible to determine the acceleration direction in cases were the motor is directed in a different way than the velocity, for example during deacceleration.

Gyro/accelometer

To know how the cube is positioned according to the room a gyro/accelometer is used. The gyrometer´s origo is set to the position the cube is supposed to balance in. When the cube is falling in some direction the gyrometer will give a signal to the micro controller of this change in angle. The micro controller will then do actions to stabilize the cube and bring its position back to origo.

2.2.3 Arduino

For this project an Arduino UNO micro controller was used. Arduino is an open source development platform which makes it easy to control motors and sensor. This is done using built in input and output pins. Arduino UNO uses an ATmega328P processor which is programmable in a language very similar to C++. It has many of built in libraries which makes it easy to control sensors and motors. Arduino UNO is an 8 bit controller which uses a reference voltage of 5V. Therefore when it sends analog signals it converts an int between 0-255 to a PWM signal between 0-5V. [7],[5]

2.2.4 H-bridge

To drive a DC-motor with higher voltage it is necessary to use a H-bridge. This is due to the Arduino´s inability to output voltage over 5 V. A H-bridge is used to control an external power source with small PWM-signals described in section 2.2.3. Conventionally an H-bridge is made of four transistors that not only makes it possible to change the output voltage, it can also change the current direction. An other reason why it is important to use an H-bridge is because DC-motors also can work as generators. Therefore if some external force would make the motor start spinning it would induce a current which could destroy the arduino. [9]

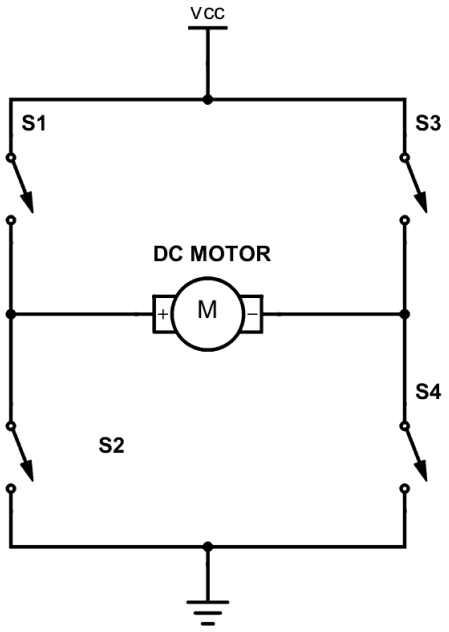


Figure 2.3. A schematic image of a H-bridge.
Source: `https://www.build-electronic-circuits.com/wp-content/uploads/2018/11/H-bridge-switches.png`

Chapter 3

Design

3.1 Implementations of components

3.1.1 Gyroscope

The gyroscope used in this project is the MPU-6050 accelerometer and gyroscope. It give values for the change of angle on the z-axle which was used as input for the PID-controller to control the control system.

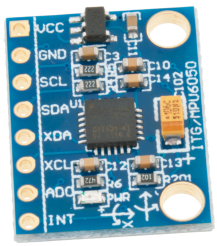


Figure 3.1. MPU-6050 - Gyroscope/Accelerometer.
Source: `https://www.electrokit.com/uploads/productimage/41016/41016233.jpg`

3.1.2 Hall effect sensor

The hall effect sensors were used to measure the angular velocity of the reaction wheel. This was done by mounting two magnets on the reaction wheel and the sensors on the motor holder giving a good distance between these. The sensors sends a signal every time the magnet is close to it with a function in Arduino that measure past time gives us the angular velocity. This because one of the sensors will send the signal right before the other.

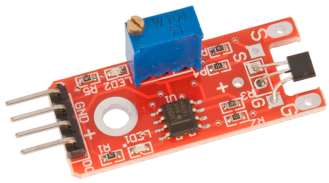


Figure 3.2. Hall effect sensor.
Source: `https://www.electrokit.com/uploads/productimage/41015/41015710.jpg`

3.1.3 Motor driver

A H-bridge is used to control the voltage over the motor and the choice of the specific H-bridge depended on it to be able to output a current as high as 12 A and control a voltage up to 24 V. The best suited motor driver was the VNH3SP30 and therefore it was used. It can output a current up to 30 A and voltages up to 40 V. [11], [10]

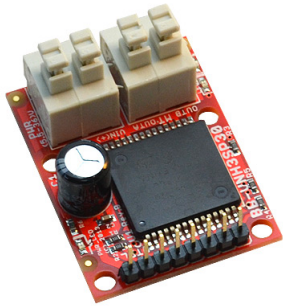


Figure 3.3. Motor driver VNH3SP30
Source: `https://www.electrokit.com/uploads/productimage/41014/41014059.jpg`

3.2 Shell

The cube is designed to have all of its components within its sides with the sides acting as a protective shell and its corner as good balancing points. The shell will consist of six plates that make up the cubes sides that will be laser cut from acrylic glass with an thickness of three millimeter. Acrylic glass was chosen because of its low density and good enough yield strength.

In every corner three v-brackets is used to connect all plates to each other using M5 screws and nuts, that is visible in figure 3.4. To connect the motor to the

cube we are using two other plate acting as motor holders that is connected to two opposite sides with distance screws in each corner of the plates, see figure 3.5. The motor is mounted on the motor holders with M4 screws. The two reaction wheels is placed on the motor axle on opposite sides between the side of the cube and the motor holders, see figure 3.5.

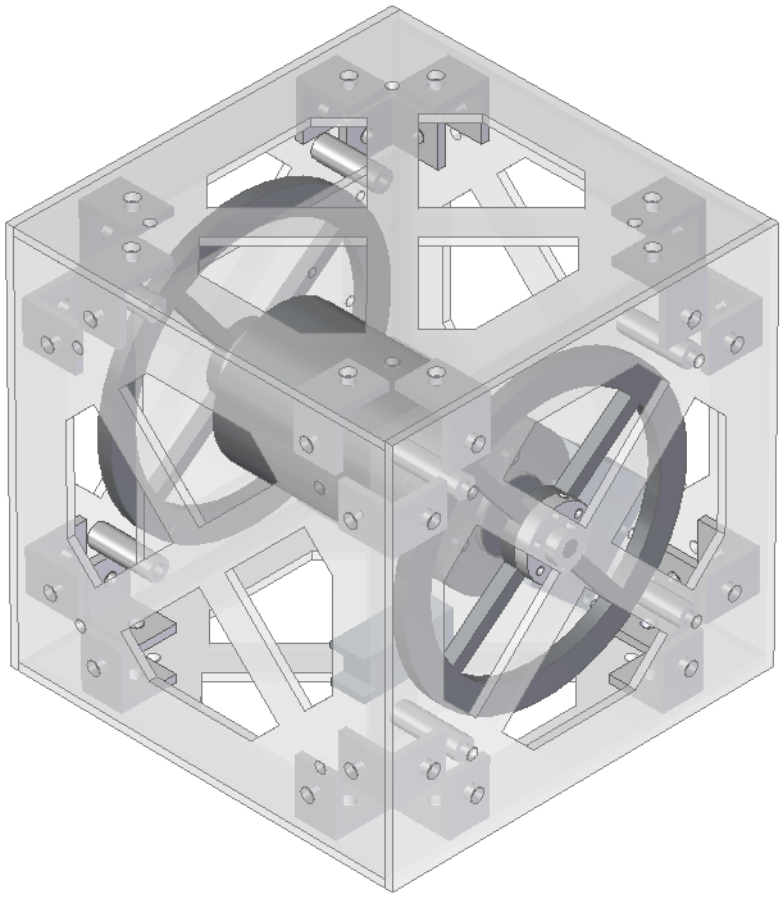


Figure 3.4. A Solid Edge CAD that shows the cube shell with reaction wheels and motor installed. Made in Solid Edge

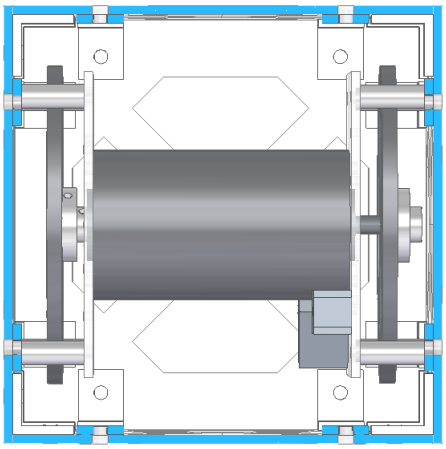


Figure 3.5. Section view from the side of cube in Solid Edge CAD showing the motor holders. Made in Solid Edge

3.3 Reaction wheel

The wheel's purpose is to generate as much moment of inertia per mass, therefore the shape of a ring. Two wheels with a radius of 5,5 centimeter and the weight of 150 g where created to generate it.

The reaction wheel is made of steel and was water cut because of its simplicity with the wheel only needed to be cut in one dimension. The wheel is then mounted on the motor axle with a hub that is connected directly on the reaction wheel with six M3 screws and then locked on the axle with two set screws from the hub

A requirement for the wheel is to be able to store enough energy that it takes for the cube to tilt to an angle of 45 °. The kinetic energy that the wheel can store is [1]

$$T = \frac{l}{2} I_{ring} \omega_{max}^2 \tag{3.1}$$

where ω_{max} is the maximum speed [rad/s] and I_{ring} is the moment of inertia that is described in *equation 2.2*. The energy it needs to store is the difference in potential energy between the states when it is lying horizontal on the surface and balancing at a 45 ° angle. This is mathematically described [1]

$$V = \sqrt{2} m_{cube} g \frac{l}{2} \tag{3.2}$$

where m_{cube} is the mass of the cube [kg], g is gravitational acceleration $[m/s^2]$ and l is the length of the cube. By using *equation 2.2* and setting V equal to T the wheels minimum mass can be calculated to

$$m_{min,ring} = \frac{\sqrt{2} m_{cube} g}{\omega_{max}^2 r_{ring}^2} \tag{3.3}$$

The Buehler 24V motor has a rated speed at 3000 rpm and with that as ω_{max} it gives that the wheel must weight at least 93 g. Since the wheel is not ideally a ring a mass of 150 g gives a big margin.

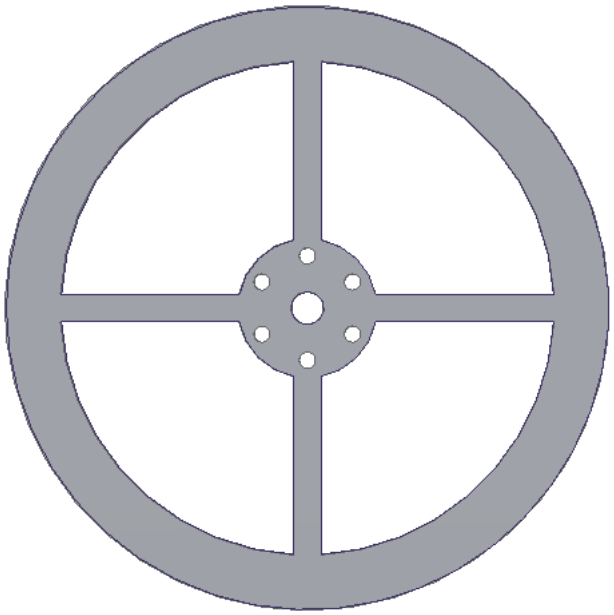


Figure 3.6. Reaction wheel from the side. Made in Solid Edge

3.4 Measure of motor speed and direction

To be able to control the DC motor it was necessary to measure the speed and direction of motor. Since there was no rotary encoder available a method described in the research *"Reaction Wheel Stabilized Stick"* [12] was used instead. This method uses two hall effect sensor and two magnets, but the number of magnets was increased to four to reduce vibrations that occur if the reaction wheel is unbalanced. The mounting of the magnets is shown below in figure 3.7.

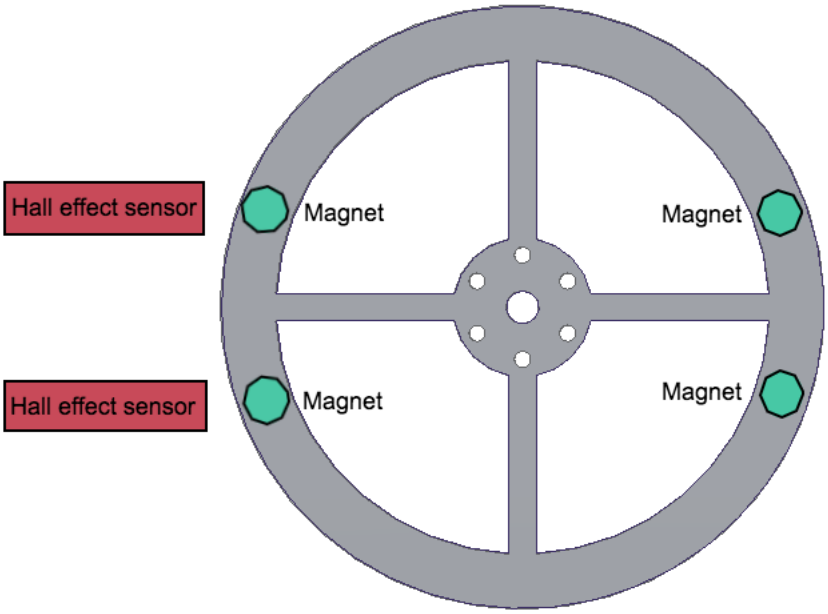


Figure 3.7. Sensors for measuring wheel speed. Made in Pixlr Editor

To measure the wheel speed a built in function in called *micros()* was used. It returns the time in microseconds since the Arduino was powered on. If both hall effect sensors detects a magnet the wheel speed [rad/s] can be calculated by [4]

$$\omega = \frac{\pi}{\Delta t} \tag{3.4}$$

where Δt [s] is the time difference between the current and the previous measure. To determine which direction the wheel is rotating a variable called *state* was defined. Before both sensors detects a magnet at the same time, one of the magnets will have been detected by one of the hall effect sensor. This information is stored in the state variable and is used to determine the wheel rotating direction.

3.5 PID-controller

To construct a stable system with a PID-controller, the three components K_P, K_I and K_D need to be tuned right. With a block diagram on the system with a transfer function it is possible to theoretically find these components.

For this system, equation 2.4 can be used for small angles with the assumption that $sin(\theta) \approx \theta$ and with the laplace transform of equation 2.4 the transfer function for this system can be given with

$$\theta(s)s^2 I_{tot} = M_{motor}(s) - mgl\theta(s), \tag{3.5}$$

where the output of the system can be written as

$$\theta(s) = \frac{1}{s^2 I_{tot} + mgl\theta} M_{motor}(s). \tag{3.6}$$

The laplace tranform of the equation 2.8 can be written as:

$$M_{motor}(s) = \frac{K_2\Phi}{R_A}U(s) - \frac{K_2\Phi^2}{R_A}\omega(s). \tag{3.7}$$

3.6 Code

The code was written in the Arduino IDE. At the beginning of the document all variables was defined. In the setup function the I/O pins was initialize as inputs or outputs. To make it easy to use the IMU a library written by Jeff Rowberg was downloaded and included in the file. With this library it comes an example code that was rewritten to suit the rest of the program. Also a PID library made by Brett Beauregard was downloaded and included.

The PID function uses an input signal to calculate an output signal depending on the parameter K_p, K_i and K_d. The used input signal is the a angle that is received from the IMU. The output signal is used as a PWM signal and regulates the speed of the motor. The PID values was calculated in the loop function and is therefore updated repetitively when the program is running.

The analog output was used on the hall effect sensors to detect the magnets if the magnet field was greater than a set value. This method worked better than using the digital output. Since the void function is running repetitively a boolean variable was used to prevent that the code does not measure the velocity many times when a magnet is detected. If the boolean is 1 when the sensor detects a magnet it measures the velocity, after it is set to 0. When the sensor doesn't detect a magnet anymore the boolean is set to 1 again.

Below in *figure 3.8* is a flowchart of the balancing cube that describes the structure of written code for the Arduino.

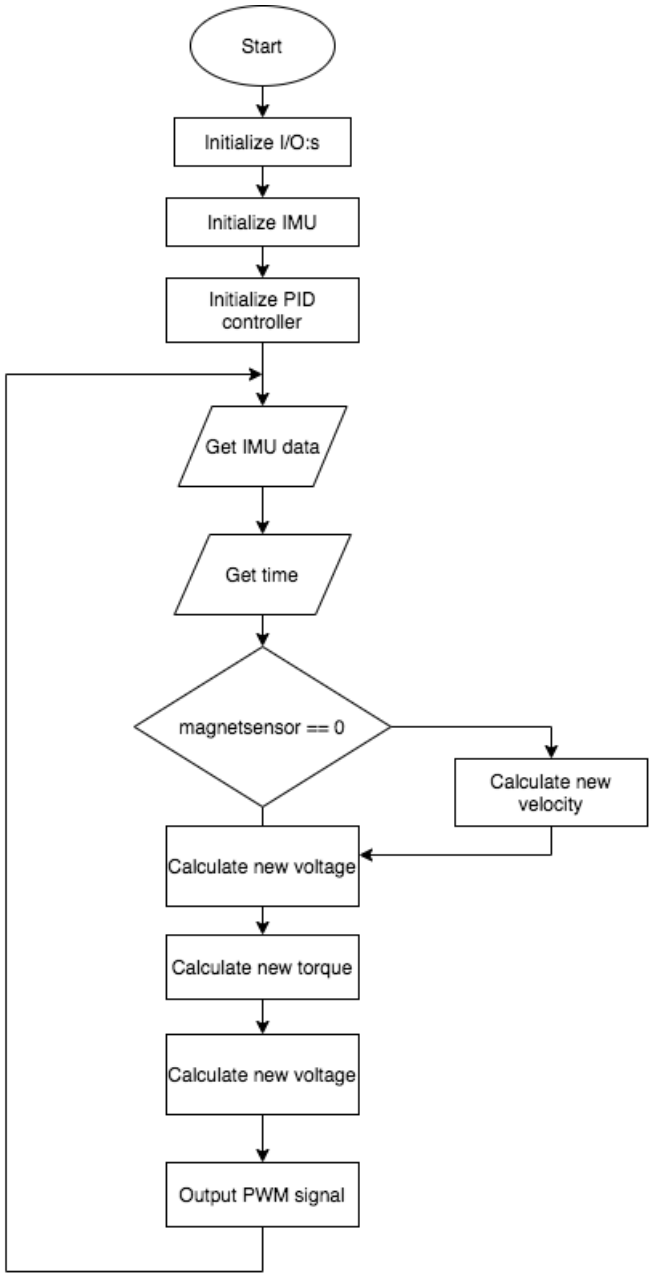


Figure 3.8. Flowchart for the balancing cube. Made in draw.io.

Chapter 4

Results

4.1 IMU

At the beginning of the development process the MPU-6050 that was used as an IMU did not seem to work correctly. It took many seconds for the IMU to stabilize and it did not always stabilize at the same values. This problem was solved with a capacitor. The MPU-6050 is sensitiv to voltage variations and the voltage supply from the arduino is not always reliable. The capacitor compensated for this and the result is shown in *figure 4.1*. A similar problem also occurred when the gain was increased in the PID-controller. By adding a capacitor over the power supply it helped sustain a steady voltage even when the gain was increased.

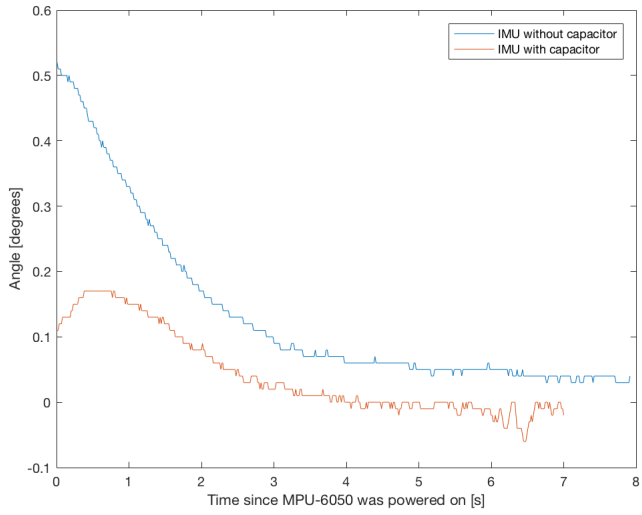


Figure 4.1. Measured values from the IMU over time with and without a capacitor. Made in Matlab.

4.2 Motor

An experiment was done to determine the deviation between the mathematical model used in *figure 2.2* and the real dynamics of the motor. All constant values for the mathematical model was received from the datasheet for the motor. The used motor was a Buhler 24V with terminal resistance $R_a = 2\Omega$ and the torque constant $k_2\phi = 5.7Ncm$. In *figure 4.2* it is shown that there is a big difference between the theoretical values and the measured values. [10]

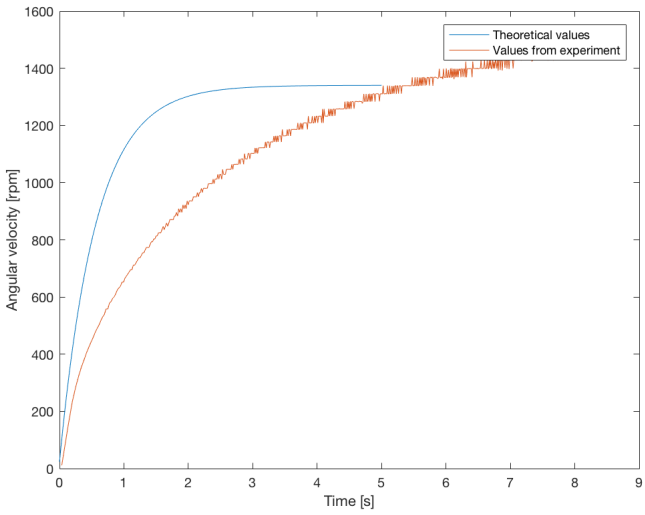


Figure 4.2. Comparison of theoretical rotating speed from *equation 2.12* and measured values. Made in Matlab.

4.2.1 PID

The PID components were determined by experiments on the cube. The K_P component was increased until the value made the cube counteract its own weight when falling to one side. Though the controller was able to counteract the cube falling it couldn't counteract the response from the controller and the cube fell to the other side, the system was unstable. To compensate for this response a K_D component was added that will counteract the response based on the speed of the error change. When the cube oscillates with only the K_P component the K_D component will decrease the oscillation, making the system more stable. Due to the controller having the error as input it will only respond when an error occur making the system never settling to the set point. Adding a K_I component calculating the integral of the error over time will make the system even more stable. Though using this method to find accurate components for the PID controller the system in this project could not stabilize completely, but made the system many times more stable than using no controller.

Chapter 5

Discussion

During this project a research has been done in how a balancing cube should be constructed. The chosen design of screwing the sides of the cube together with three attachments in each corner resulted in a very robust design. Using laser printers to cut out all the necessary parts from acrylic glass also benefited the construction since this method gave high precision. It worked very well to use two plates as motor attachments. This design was very stable and also allowed all the sensor to be mounted on the plates. A disadvantage with this design is that it only works for two dimensional balancing. For multidimensional balancing another motor must be added that would not fit in this construction.

No PID parameters where found that could balance the cube. There are many possible reasons to why it didn't work. Despite that a lot of focus have been on developing a mechanically balanced cube there is a risk that weight is not equally distributed on both sides. Also the angles from the gyro does not always settle at the same point that causes problem in the controlling of the cube. Also the weight of the motor makes it harder to control the cube because a low power to weight ratio reduces the maximum deviation angle from the equilibrium point that is when the angle is zero. There are also methods to iterate the PID parameters that is something that could be investigated.

This project was inspired by Cubli, a cube that was built at ETH Zurich. For the cube to be able to jump up from horizontal position to balancing on an edge it would require a voltage of approximately 36 V. An alternative to this would be to design a brake system with servomotors that can perform the *"jump"* action and then a smaller motor can balance the cube for smaller angles. This would not only reduce the required voltage and current supply, but also free space which is very important if the project would expand to multi dimensional balancing. It would also be interesting to use brushless motors since they generally have a higher torque per weight ratio.

At first the dynamic model of the DC motor described in *section 2.2.1* was intended to be used to regulate the motor torque so it corresponded with the computed output from the PID controller. Since it was difficult to know how reliable the measurement of the calculated velocity from the hall effect sensors were, the output was instead directly used to control the PWM signal to the motor.

Another thing that can be analysed is why the result from the experiment and the theoretical values deviated so much in *figure 4.2*. When the angular velocity is small it looks like the two lines are parallel that is good because the torque is proportional to the slope. When the velocity gets higher it differs a lot. This might be because the internal friction is not considered in mathematical model that can be assumed to be proportional to the rotating velocity. Also the moment of inertia might have been underestimated. [1]

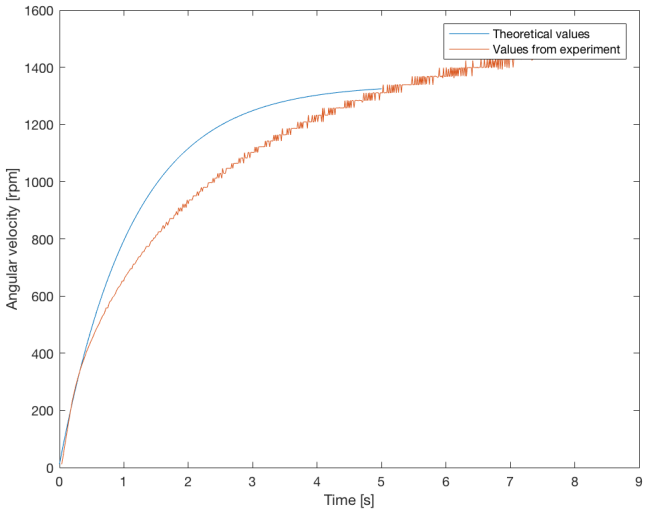


Figure 5.1. Comparison of theoretical rotating speed from *equation 2.12* and measured values, adjusted moment of inertia. Made in Matlab

In *figure 5.1* the reaction wheels moment of inertia has been increased with a factor of 2. Now the model looks more similar to the experimental values, but they curves still have different characteristics and therefore no conclusion can be taken.

Bibliography

[1] Nicholas Apazidis. *Mekanik 1 Statik och partikeldynami.* Studentlitteratur, 2013.

[2] Nicholas Apazidis. *Mekanik 2 Partikelsystem, stel kropp och analytisk mekanik.* Studentlitteratur, 2012.

[3] Arduino.cc. *PID.* `https://playground.arduino.cc/Code/PIDLibraryConstructor/` retrieved 2020-03-29

[4] Arduino.cc. *micros().* `https://www.arduino.cc/reference/en/language/functions/time/micros/` retrieved 2020-05-11

[5] Arduino.cc. *What is arduino?.* `https://www.arduino.cc/en/Guide/Introduction` retrieved 2020-05-12

[6] Dr. Hodge Jenkins. *Tuning for PID Controllers*, `http://faculty.mercer.edu/jenkins_he/documents/TuningforPIDControllers.pdf`, retrieved 2020-05-12

[7] Arduino.cc. *Ardunio uno rev3.* `https://store.arduino.cc/arduino-uno-rev3` retrieved 2020-05-12

[8] Torkel Glad, Lennart Ljung. *Reglerteknik: grundläggande teori.* Studentlitteratur, 2006

[9] Hans Johansson. *Elektroteknik.* Instutitionen för maskinkonstruktion och mekatronik, 2013

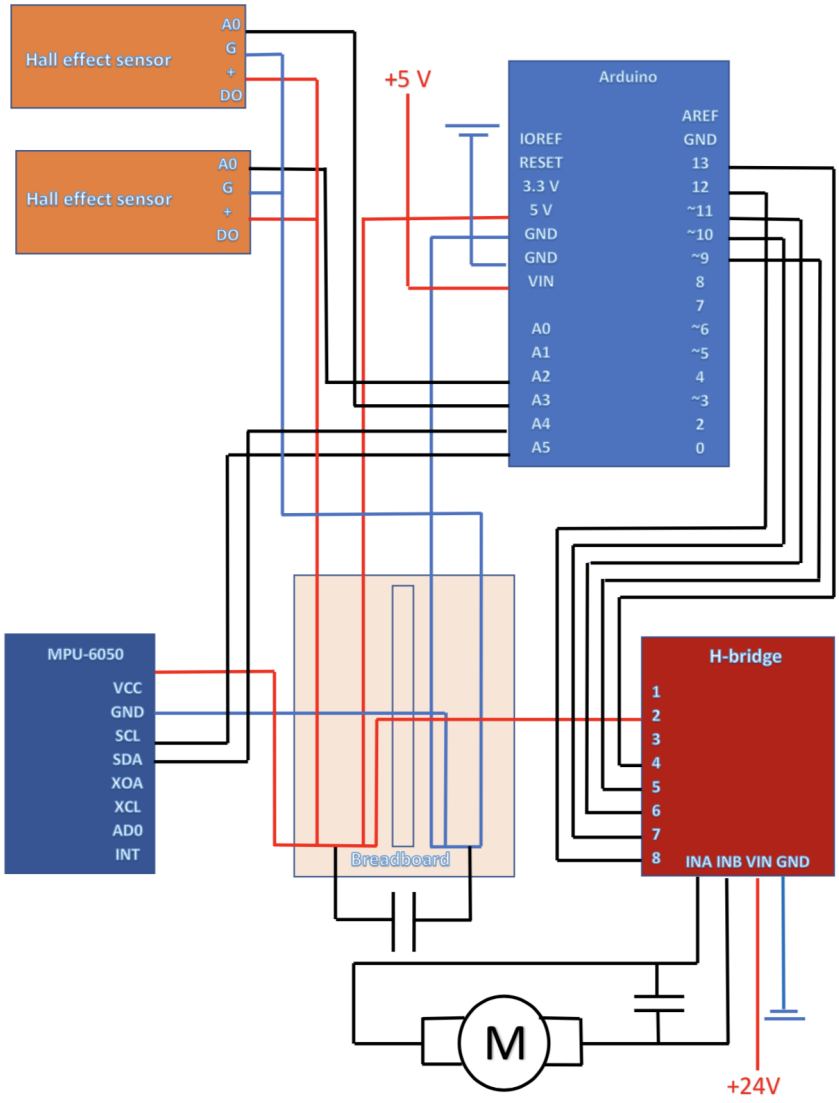
[10] Buehlermotor. *DC Motor 51x88.* `https://www.buehlermotor.com/fileadmin/user_upload/stock_service/datasheets/DC-Motor_51x88__1.13.044.2XX.pdf?fbclid=IwAR1ru7WgIEmUk8lEha7CSzmgLtvLIcKwwQ4AvKP0RA68nEo4glcVtn7Yeh0`, retrieved 2020-05-11

[11] ST. *Fully integrated H-bridge motor driver.* `https://www.electrokit.com/uploads/productfile/41014/vnh3sp30.pdf`, retrieved 2020-05-12

[12] Pontus Gräsberg, Bill Lavebratt. *Reaction Wheel Stabilized Stick.* KTH. 2019. oai:DiVA.org:kth-264481

Appendix A

Electric scheme chart



Appendix B

Datasheet for the motor

Type / Baureihe 1.13.044.XXX				235	236
Characteristics*	**Nenndaten***				
Rated voltage	Nennspannung	U/V	V	12	24
Rated power	Nennleistung	P_N	W	50	50
Rated torque	Nenndrehmoment	T_N/M_N	Ncm	15	15
Rated speed	Nenndrehzahl	n_N	rpm/min^{-1}	3000	3000
Rated current	Nennstrom	I_N	A	6.2	3.1
No load characteristics*	**Leerlaufdaten***				
No load speed	Leerlaufdrehzahl	n_0	rpm/min^{-1}	3980	3900
No load current	Leerlaufstrom	I_0	A	0.4	0.2
Stall characteristics*	**Anlaufdaten***				
Stall torque	Anlaufmoment	T_S/M_H	Ncm	64	64
Stall current	Anlaufstrom	I_S/I_H	A	24	12
Performance characteristics*	**Leistungsdaten***				
max. Output power	max. Abgabeleistung	P_{max}	W	70	70
max. Constant torque	max. Dauerdrehmoment	T_{max}/M_{max}	Ncm	10	10
Motor parameters*	**Motorparameter***				
Weight	Gewicht	G	g	765	765
Rotor inertia	Läuferträgheitsmoment	J	gcm^2	180	180
Terminal resistance	Anschlusswiderstand	R	Ohm	0.5	2.0
Inductance	Induktivität	L	mH	0.5	1.0
Mech. time constant	Mech. Zeitkonstante	τ_m	ms	13	13
Electr. time constant	Elektr. Zeitkonstante	τ_e	ms	1.0	0.5
Speed regulation constant	Drehzahregelkonstante	R_n	rpm/Ncm	60	60
Torque constant	Drehmomentkonstante	k_I/k_M	Ncm/A	2.8	5.7
Thermal resistance	Thermischer Widerstand	R_{th}	K/W	9.5	9.5
Thermal time constant	Thermische Zeitkonstante	τ_{th}	min	9.5	9.5
Axial play	Axialspiel		mm	< 0.01	< 0.01
Direction of rotation	Drehrichtung			bidirectional / bidirektional	

Appendix C

Source code

C.1 Motor experiment in Matlab

```
clear all, close all, clc %clean command window and workspace
}
%% Mathematical motor model:

format long

%mehanical constants
m_cube = 2 % Total mass of th cube [kg]
l_cube = 15*10^(-2) % length of cube [m]
m_wheel = 0.300 % flywheel mass[kg]
r_wheel = (11/2)*10^(-2) % radius of the flywheel [m]
I_wheel = m_wheel*r_wheel^2 % momen of inertia of the flywheel [ kgm^2]
I_motor = 180*10^(-7) % momen of inertia of the rotor shaft [ kgm^2]

J = I_wheel+I_motor;


%motor varibles:
omega = 0; % Angular velocity
u = 8 % Voltage [V]


%Motor specifications:
Ra = 2 % Terminal resistance [ohm]
Kt = 5.7*10^(-2) %Torque constant Nm/A
U_max = 8 % max voltage [V]

% Other constants
N = 500 % Number of iterations


%%Motor dynamics:
t = 5 % tid [s]
```

```
dU = U_max/(N-1)
dt = t/(N-1)

T = [0:dt:t];



M = []
OMEGA = [];

for x = T
   torque = (Kt/Ra)*U_max-(((Kt^2)/Ra))*omega;

   acceleration = torque/J;
   omega = omega + acceleration * (t/(N-1));

  OMEGA = [OMEGA omega];


  M = [M torque];
end

OMEGA = OMEGA*60/(2*pi)

figure

plot(T,OMEGA)
xlabel('Time [s]')
ylabel('Angular velocity [rpm]')
set(gcf,'color','w');
hold on
%title('U=24V')

%% Values from experiment

data = load("kex/8v_motor.txt")

Time = (data(1:666,1)/(10^6))-1.3;
rpm = data(1:666,2)*0.5;
plot(Time, rpm)
legend('Theoretical values', 'Values from experiment')
```

C.2 Arduino code in C++

```
#include <PID_v1.h>


// Variables for gyro
double roll_deg;

//Variables for PID
double input;
```

```
double output;
double setpoint;
double Kp = 5, Ki=0, Kd=0;

PID PID_controller(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);
double pwmSignal;


// Variables for motor
const int inaPin = 13;
const int inbPin = 9;
const int pwmPin = 11;
const int diagaPin = 10;
const int diagbPin = 12;
const int buttonPin = 2;
const int trimPin = A0;
int i = 0;

// I2C device class (I2Cdev) demonstration Arduino sketch for MPU6050
    class using DMP (MotionApps v2.0)
// 6/21/2012 by Jeff Rowberg <jeff@rowberg.net>
// Updates should (hopefully) always be available at https://github.com
    /jrowberg/i2cdevlib
//
// Changelog:
//      2013-05-08 - added seamless Fastwire support
//                 - added note about gyro calibration
//      2012-06-21 - added note about Arduino 1.0.1 + Leonardo
    compatibility error
//      2012-06-20 - improved FIFO overflow handling and simplified
    read process
//      2012-06-19 - completely rearranged DMP initialization code and
    simplification
//      2012-06-13 - pull gyro and accel data from FIFO packet instead
    of reading directly
//      2012-06-09 - fix broken FIFO read sequence and change interrupt
     detection to RISING
//      2012-06-05 - add gravity-compensated initial reference frame
    acceleration output
//                 - add 3D math helper file to DMP6 example sketch
//                 - add Euler output and Yaw/Pitch/Roll output formats
//      2012-06-04 - remove accel offset clearing for better results (
    thanks Sungon Lee)
//      2012-06-01 - fixed gyro sensitivity to be 2000 deg/sec instead
    of 250
//      2012-05-30 - basic DMP initialization working

/* ============================================
I2Cdev device library code is placed under the MIT license
Copyright (c) 2012 Jeff Rowberg

Permission is hereby granted, free of charge, to any person obtaining a
     copy
of this software and associated documentation files (the "Software"),
```

```
    to deal
in the Software without restriction, including without limitation the
    rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or
    sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included
     in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
     OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY
    ,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
     THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
     FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
    IN
THE SOFTWARE.
===============================================
*/

// I2Cdev and MPU6050 must be installed as libraries, or else the .cpp
    /.h files
// for both classes must be in the include path of your project
#include "I2Cdev.h"

#include "MPU6050_6Axis_MotionApps20.h"
//#include "MPU6050.h" // not necessary if using MotionApps include
    file

// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE
    implementation
// is used in I2Cdev.h
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include "Wire.h"
#endif

// class default I2C address is 0x68
// specific I2C addresses may be passed as a parameter here
// AD0 low = 0x68 (default for SparkFun breakout and InvenSense
    evaluation board)
// AD0 high = 0x69
MPU6050 mpu;
//MPU6050 mpu(0x69); // <-- use for AD0 high

/*
    ===========================================================
```

```
    NOTE: In addition to connection 3.3v, GND, SDA, and SCL, this sketch
    depends on the MPU-6050's INT pin being connected to the Arduino's
    external interrupt #0 pin. On the Arduino Uno and Mega 2560, this is
    digital I/O pin 2.
 *
    ============================================================================
     */

/*
    ============================================================================

    NOTE: Arduino v1.0.1 with the Leonardo board generates a compile
        error
    when using Serial.write(buf, len). The Teapot output uses this
        method.
    The solution requires a modification to the Arduino USBAPI.h file,
        that
    is fortunately simple, but annoying. This will be fixed in the next
        IDE
    release. For more info, see these links:

    http://arduino.cc/forum/index.php/topic,109987.0.html
    http://code.google.com/p/arduino/issues/detail?id=958
 *
    ============================================================================
     */



// uncomment "OUTPUT_READABLE_QUATERNION" if you want to see the actual
// quaternion components in a [w, x, y, z] format (not best for parsing
// on a remote host such as Processing or something though)
//#define OUTPUT_READABLE_QUATERNION

// uncomment "OUTPUT_READABLE_EULER" if you want to see Euler angles
// (in degrees) calculated from the quaternions coming from the FIFO.
// Note that Euler angles suffer from gimbal lock (for more info, see
// http://en.wikipedia.org/wiki/Gimbal_lock)
//#define OUTPUT_READABLE_EULER

// uncomment "OUTPUT_READABLE_YAWPITCHROLL" if you want to see the yaw/
// pitch/roll angles (in degrees) calculated from the quaternions
    coming
// from the FIFO. Note this also requires gravity vector calculations.
// Also note that yaw/pitch/roll angles suffer from gimbal lock (for
// more info, see: http://en.wikipedia.org/wiki/Gimbal_lock)
#define OUTPUT_READABLE_YAWPITCHROLL

// uncomment "OUTPUT_READABLE_REALACCEL" if you want to see
    acceleration
// components with gravity removed. This acceleration reference frame
    is
// not compensated for orientation, so +X is always +X according to the
// sensor, just without the effects of gravity. If you want
```

```
    acceleration
// compensated for orientation, us OUTPUT_READABLE_WORLDACCEL instead.
//#define OUTPUT_READABLE_REALACCEL

// uncomment "OUTPUT_READABLE_WORLDACCEL" if you want to see
    acceleration
// components with gravity removed and adjusted for the world frame of
// reference (yaw is relative to initial orientation, since no
    magnetometer
// is present in this case). Could be quite handy in some cases.
//#define OUTPUT_READABLE_WORLDACCEL

// uncomment "OUTPUT_TEAPOT" if you want output that matches the
// format used for the InvenSense teapot demo
//#define OUTPUT_TEAPOT



#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0
    = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42
    bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor
    measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel
    sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel
    sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll
    container and gravity vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = { '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00,
    '\r', '\n' };



// ================================================================
// ===                      INTERRUPT DETECTION ROUTINE                  ===
// ================================================================
```

```
volatile bool mpuInterrupt = false;     // indicates whether MPU
    interrupt pin has gone high
void dmpDataReady() {
    mpuInterrupt = true;
}



// ================================================================
// ===                      INITIAL SETUP                       ===
// ================================================================

void setup() {
    //Setup for PID
    setpoint = 20;

    PID_controller.SetMode(AUTOMATIC);
    PID_controller.SetTunings(Kp,Ki,Kd);
    PID_controller.SetOutputLimits(-255, 255);

    // join I2C bus (I2Cdev library doesn't do this automatically)
    #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
    #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
    #endif

    // I/O:s for motor pins
    pinMode(buttonPin, INPUT);
    pinMode(inaPin, OUTPUT);
    pinMode(inbPin, OUTPUT);
    pinMode(pwmPin, OUTPUT);
    pinMode(diagaPin, INPUT);
    pinMode(diagbPin, INPUT);
    pinMode(trimPin, INPUT);

    // Example code for MPU6050 downloaded

    // initialize serial communication
    // (115200 chosen because it is required for Teapot Demo output,
        but it's
    // really up to you depending on your project)
    Serial.begin(115200);
    while (!Serial); // wait for Leonardo enumeration, others continue
        immediately

    // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or
        Ardunio
    // Pro Mini running at 3.3v, cannot handle this baud rate reliably
        due to
    // the baud timing being too misaligned with processor ticks. You
        must use
    // 38400 or slower in these cases, or use some kind of external
```

```
    separate
// crystal solution for the UART timer.

// initialize device
Serial.println(F("Initializing I2C devices..."));
mpu.initialize();

// verify connection
Serial.println(F("Testing device connections..."));
Serial.println(mpu.testConnection() ? F("MPU6050 connection
    successful") : F("MPU6050 connection failed"));

// wait for ready
Serial.println(F("\nSend any character to begin DMP programming and
    demo: "));
while (Serial.available() && Serial.read()); // empty buffer
while (!Serial.available());                 // wait for data
while (Serial.available() && Serial.read()); // empty buffer again

// load and configure the DMP
Serial.println(F("Initializing DMP..."));
devStatus = mpu.dmpInitialize();

// supply your own gyro offsets here, scaled for min sensitivity
mpu.setXGyroOffset(220);
mpu.setYGyroOffset(76);
mpu.setZGyroOffset(-85);
mpu.setZAccelOffset(1788); // 1688 factory default for my test chip

// make sure it worked (returns 0 if so)
if (devStatus == 0) {
    // turn on the DMP, now that it's ready
    Serial.println(F("Enabling DMP..."));
    mpu.setDMPEnabled(true);

    // enable Arduino interrupt detection
    Serial.println(F("Enabling interrupt detection (Arduino
        external interrupt 0)..."));
    attachInterrupt(0, dmpDataReady, RISING);
    mpuIntStatus = mpu.getIntStatus();

    // set our DMP Ready flag so the main loop() function knows it'
        s okay to use it
    Serial.println(F("DMP ready! Waiting for first interrupt..."));
    dmpReady = true;

    // get expected DMP packet size for later comparison
    packetSize = mpu.dmpGetFIFOPacketSize();
} else {
    // ERROR!
    // 1 = initial memory load failed
    // 2 = DMP configuration updates failed
    // (if it's going to break, usually the code will be 1)
    Serial.print(F("DMP Initialization failed (code "));
```

```
        Serial.print(devStatus);
        Serial.println(F(")"));
    }

    // configure LED for output
    pinMode(LED_PIN, OUTPUT);
}



// ================================================================
// ===                    MAIN PROGRAM LOOP                     ===
// ================================================================

void loop() {
    // if programming failed, don't try to do anything
    if (!dmpReady) return;

    // wait for MPU interrupt or extra packet(s) available
    while (!mpuInterrupt && fifoCount < packetSize) {
        // other program behavior stuff here
        // .
        // .
        // .
        // if you are really paranoid you can frequently test in
            between other
        // stuff to see if mpuInterrupt is true, and if so, "break;"
            from the
        // while() loop to immediately process the MPU data
        // .
        // .
        // .
    }

    // reset interrupt flag and get INT_STATUS byte
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is
        too inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
        // reset so we can continue cleanly
        mpu.resetFIFO();
        Serial.println(F("FIFO overflow!"));

    // otherwise, check for DMP data ready interrupt (this should
        happen frequently)
    } else if (mpuIntStatus & 0x02) {
        // wait for correct available data length, should be a VERY
            short wait
        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
```

```
// read a packet from FIFO
mpu.getFIFOBytes(fifoBuffer, packetSize);

// track FIFO count here in case there is > 1 packet available
// (this lets us immediately read more without waiting for an
    interrupt)
fifoCount -= packetSize;

#ifdef OUTPUT_READABLE_QUATERNION
    // display quaternion values in easy matrix form: w x y z
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    Serial.print("quat\t");
    Serial.print(q.w);
    Serial.print("\t");
    Serial.print(q.x);
    Serial.print("\t");
    Serial.print(q.y);
    Serial.print("\t");
    Serial.println(q.z);
#endif

#ifdef OUTPUT_READABLE_EULER
    // display Euler angles in degrees
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetEuler(euler, &q);
    Serial.print("euler\t");
    Serial.print(euler[0] * 180/M_PI);
    Serial.print("\t");
    Serial.print(euler[1] * 180/M_PI);
    Serial.print("\t");
    Serial.println(euler[2] * 180/M_PI);
#endif

#ifdef OUTPUT_READABLE_YAWPITCHROLL
    // display Euler angles in degrees
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
    //Serial.print("ypr\t");
    //Serial.print(ypr[0] * 180/M_PI);
    //Serial.print("\t");
    //Serial.print(ypr[1] * 180/M_PI);
    //Serial.print("\t");
    //Serial.println(ypr[2] * 180/M_PI);
#endif

#ifdef OUTPUT_READABLE_REALACCEL
    // display real acceleration, adjusted to remove gravity
    mpu.dmpGetQuaternion(&q, fifoBuffer);
    mpu.dmpGetAccel(&aa, fifoBuffer);
    mpu.dmpGetGravity(&gravity, &q);
    mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity);
    Serial.print("areal\t");
```

```
            Serial.print(aaReal.x);
            Serial.print("\t");
            Serial.print(aaReal.y);
            Serial.print("\t");
            Serial.println(aaReal.z);
        #endif

        #ifdef OUTPUT_READABLE_WORLDACCEL
            // display initial world-frame acceleration, adjusted to
                remove gravity
            // and rotated based on known orientation from quaternion
            mpu.dmpGetQuaternion(&q, fifoBuffer);
            mpu.dmpGetAccel(&aa, fifoBuffer);
            mpu.dmpGetGravity(&gravity, &q);
            mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity);
            mpu.dmpGetLinearAccelInWorld(&aaWorld, &aaReal, &q);
            Serial.print("aworld\t");
            Serial.print(aaWorld.x);
            Serial.print("\t");
            Serial.print(aaWorld.y);
            Serial.print("\t");
            Serial.println(aaWorld.z);
        #endif

        #ifdef OUTPUT_TEAPOT
            // display quaternion values in InvenSense Teapot demo
                format:
            teapotPacket[2] = fifoBuffer[0];
            teapotPacket[3] = fifoBuffer[1];
            teapotPacket[4] = fifoBuffer[4];
            teapotPacket[5] = fifoBuffer[5];
            teapotPacket[6] = fifoBuffer[8];
            teapotPacket[7] = fifoBuffer[9];
            teapotPacket[8] = fifoBuffer[12];
            teapotPacket[9] = fifoBuffer[13];
            Serial.write(teapotPacket, 14);
            teapotPacket[11]++; // packetCount, loops at 0xFF on
                purpose
        #endif

        // blink LED to indicate activity
        blinkState = !blinkState;
        digitalWrite(LED_PIN, blinkState);
    }


    input = roll_deg = ypr[1] * 180/M_PI;
    //Serial.println(roll_deg);

    PID_controller.Compute();

    Serial.print(input);
    Serial.print("\t");
    Serial.println(output);
```

```
    if(output >= 0){
      pwmSignal = output;
      digitalWrite(inaPin, LOW); //CW direction of motor.
      digitalWrite(inbPin, HIGH);
    } else {
      pwmSignal = -1*output;
      digitalWrite(inaPin, HIGH); //CCW direction of motor.
      digitalWrite(inbPin, LOW);
    }

    analogWrite(pwmPin, pwmSignal);

}
```