TU/e - Department of Mechanical Engineering
Eindhoven, 30 May 2016

Design and control of a three-axis gimbal

Internship
Quartile 1, 2015-2016
DC 2016.036

Supervisor: Prof. Wang, L.
Prof. dr. Nijmeijer, H.
Student: van Nispen, S.H.M. (Stephan) 0764290

Table of Contents

Chapter 1

Introduction

When a 3-axis gimbal is mounted to an Unmanned Aerial Vehicle, UAV, it can be used to feed smooth images from for instance a camera or LIDAR. The images can be used to map the surroundings which is necessary to let an UAV fly autonomously. A gimbal rules out vibrations and movement from the surroundings which is generally done by three brushless DC motors and an inertial measurement unit. Another characteristic is that the gimbal can rotate apart from the UAV. Figure 1.1 shows a gimbal which can be bought in a store [1].
Although a gimbal can be bought in a store, in this report a complete gimbal will be made from scratch. The benefit of not buying a gimbal, is gaining full control over materials, design and control algorithm. The problem with commercially available gimbals is the inability to change the software completely. A lot more freedom can be gained by writing the control algorithm and choosing the whole design and materials. The goal of this report is to design, model and control a gimbal to gain full control over the abilities of a gimbal.
This report starts by describing the process of choosing materials and designing a 3-axis gimbal in Chapter 2. To control the gimbal, general theory of a brushless DC motor is introduced in Chapter 3 after which the gimbal and actuators are modelled in Chapter 4. This model is inserted in `Simulink` in Chapter 5 and experiments are conducted with the designed gimbal in Chapter 6. Eventually the simulations are compared to the real measurement data in Chapter 7. Unfortunately due to late arrival of the chosen DC motors only position loop control is achieved.

Figure 1.1: *Example of a gimbal which can be bought in a store [1]*

Chapter 2

Materials and design

In this chapter the used materials are introduced and the design of the gimbal is explained. First the actuators of the gimbal are chosen, after which the choice of the controller and sensors is explained. Some additional materials for the gimbal are introduced to complete the overview of the used materials. At last the design of the gimbal is shown, together with the design decisions.

2.1 Actuators

The first step in choosing an actuator is deciding on the type. The main requirement of the actuator is that it should be fast enough to compensate the error without creating a blurry image when a camera is mounted. A servo motor seems a good choice for a gimbal because of relative simple control with regard to position. However, the deceleration of a servo motor is rather slow which means that with rapid changing angles the servo motor can not give a steady image. Another option is a brushless DC motor. This motor is normally used for constant rotation of for instance a fan. The advantage of a brushless DC motor is that it is capable of fast acceleration and deceleration and therefore it is suitable to use on a gimbal. One of the downsides of a brushless DC motor is that it is a bit harder to control than a servo motor. Having chosen the type of motor, a decision on the number of poles and coils can be made. The number of poles and coils are important to control the motor, the more poles and coils the more responsive the motor is to its input. A three-phase brushless DC motor with 14 poles and 12 coils is chosen as can be seen in Figure 2.1. Increasing the number of poles and coils would significantly increase the costs while the performance only slightly increases. Decreasing the number of poles and coils would reduce the costs but would also decrease the resolution significantly. Further details of the resolution and of a brushless DC motor can be found in Chapter 3.

Figure 2.1: *The eventually used brushless dc motor*

2.2 Controller and sensor

To control the motor an Arduino Nano V3.0 is used. The advantage of using this device, is that the code is written by the user so it is completely known what happens. The Arduino has six PWM output channels which is not sufficient to control three brushless DC motors, each motor needs three PWM output channels. Therefore, an extra PWM shield is needed to extend the number of PWM output channels.
At least one gyroscope is needed to measure the angle of the copter or camera. It is possible to control the gimbal using only one gyroscope. The gyroscope can be attached to the copter and the motor has to compensate for the angle made by the copter. However, it is hard to control the gimbal because the real angle of the camera will be unknown. An other option is to mount the gyroscope at the camera side. Now the angle of the camera can be determined exactly but control is a bit harder because the camera side will vibrate easier unless it is controlled well. A third option is to mount two gyroscopes, one at the camera side and one on the copter. When mounting two gyroscopes, it is possible to distinguish intentional movement and unintentional movement. First the gimbal will be controlled by using only one gyroscope attached at the camera side. The sensor used is a MPU6050 GY-521, which has six degrees of freedom, has a three axes gyroscope and a three axes accelerometer.To read out the data of the MPU6050 on the Arduino the I2C library of Jeff Rowberg is used [2].

2.3 Additional materials

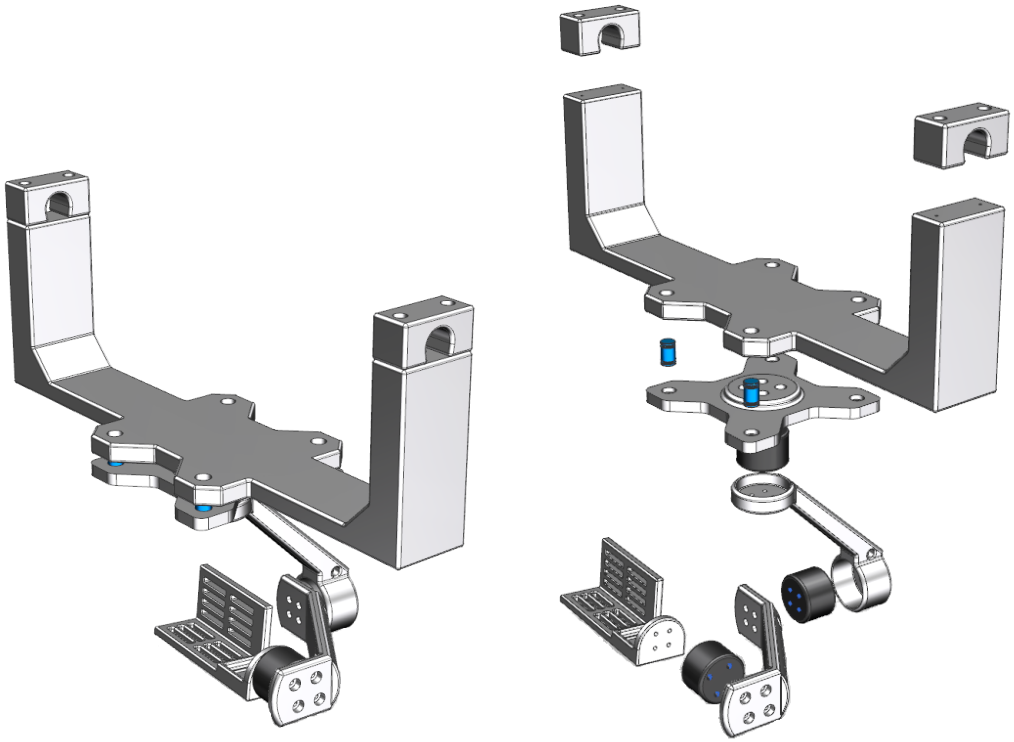
Apart from the main components other materials are needed. A LiPo battery is used as a power source to the actuators, Arduino and PWM shield. Also three H-bridges are needed to control the voltage to the motor and to make sure the back-EMF does not destroy the Arduino or PWM shield. Furthermore, damping is added in the design to eliminate vibrations from the UAV to the gimbal, making the image smoother. Eventually some capacitors are used to flatten out power outputs.

2.4 Design of the gimbal

The part above described the components necessary to move the gimbal but a frame is needed in between those components. The frame is designed in Siemens NX9.0 and is 3D printed. Figure 2.2 shows a normal and exploded view of all components, where the white parts are the 3D printed parts. The three motors and damping can also be seen in the assembly and are respectively black and blue. The gimbal is designed to carry a camera or other tooling with a maximum weight around 200 grams.

During the design process several choices had to be made. First the top part of the gimbal is designed, which connects the UAV and the gimbal. A requirement is that the gimbal will fit on already existing UAV's. These already existing UAV's all have two similar characteristics. The battery hangs in the centre below the UAV and the arms of a quadcopter, hexacopter or octocopter are all round and always in pairs directly opposite of each other. This explains the design of two clamps which fit tightly around the arms of one of the UAV's. These clamps are screwed on the top part which leaves enough room for a battery that is attached underneath the UAV's. For more stiffness the inside corners are chamfered. After designing most of the top part a connection to the motors has to be made. To rule out unwanted vibrations from the UAV an extra platform is introduced which is connected by dampers to the top platform. To leave space on the top platform for the Arduino, PWM shield, capacitors and H-bridges, the dampers are placed in a square which is wide enough to leave this space. The first actuator is attached to the extra platform, this extra platform is shaped to keep its stiffness but with as little material as possible. The first actuator is responsible for the yaw rotation, subsequently the arm attached to it is referred to as the yaw-arm. The arms attached to the second and third actuator are called the pitch-arm and roll-arm respectively. The design of all these arms is done collectively so that the pose in Figure 2.2 is roughly the equilibrium position when for instance a camera is attached to the roll-arm. On the roll arm some space is left to mount the MPU6050, near the camera mounting point. Furthermore, holes are added to the yaw-arm and pitch-arm to hide the wiring from sight. The yaw-arm saves space by inserting the two attached actuators in the arm. Besides saving space, it is practical to hide the wiring. The pitch-arm is made as light as possible so that the gimbal can stand in its equilibrium pose. The roll-arm has several slots to make good mounting of a camera possible. A camera can be mounted by using velcro tape.

The design and chosen materials will be used to model and simulate the movement of the gimbal. Before modelling the gimbal the theory of a brushless DC motor will be treated in more detail.



(a) (b)

Figure 2.2: *Design of the three axis gimbal*

Chapter 3

General theory of a brushless DC motor

The materials and design chosen in Chapter 2 can be used to model and simulate the movement of the gimbal. However, to model the actuators, a better understanding of the brushless DC motors is need. In this chapter the general theory of a brushless DC motor is introduced to eventually propose a basis for controlling the brushless DC motor.

A brushless DC motor consists of a rotor and stator, as can be seen in Figure 3.1. The outer circle is the rotor and the inner part is the stator. The rotor consists of permanent magnets, where the blue and red colour represent the north and south pole respectively. The stator has in this case three electromagnets which differ in polarity by alternating the current, a so called three-phase motor. Typically the three electromagnets differ a 120 degrees in phase. Figure 3.1 shows the rotor turning counter clockwise while the electromagnets are changing polarity. By changing the polarities at a constant frequency, the rotor will turn with a constant speed.

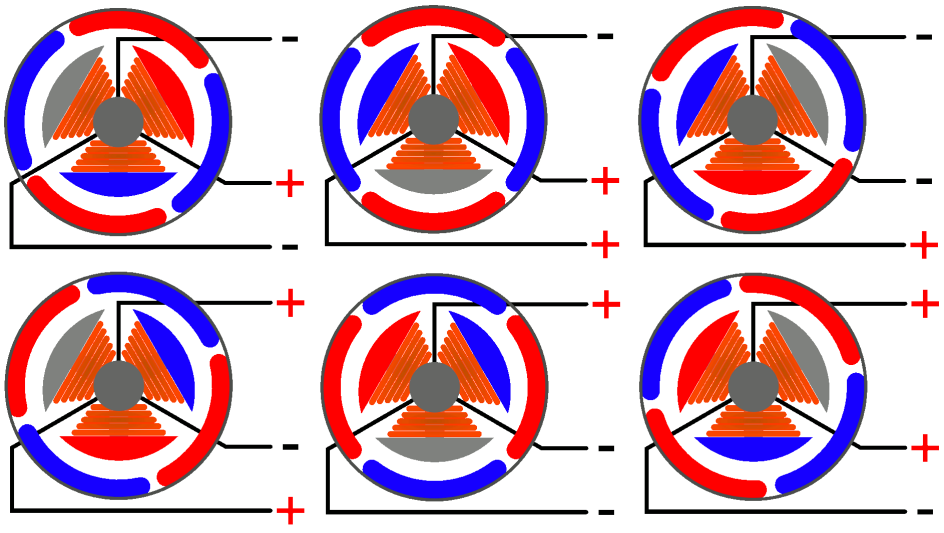


Figure 3.1: *Schematic of the operation of a brushless dc motor*

Taking a closer look shows that one electrical rotation, $\theta_{e,r}$, results in half a mechanical rotation, $\theta_{m,r}$. This depends on the number of poles, as stated in the following equation:

$$\theta_{m,r} = \frac{2}{\text{number of poles}}\,\theta_{e,r}. \tag{3.1}$$

The motor which is used has fourteen poles, which means that one electrical rotation is one seventh of a full mechanical rotation. Figure 3.1 shows only high and low signals, which would mean that each electrical rotation consists of six steps. The amount of steps can be extended when using PWM signals. A PWM signal can take a value between zero and 255, which means sending a voltage at zero per cent or 100 per cent respectively. In Figure 3.2 this is visualized.

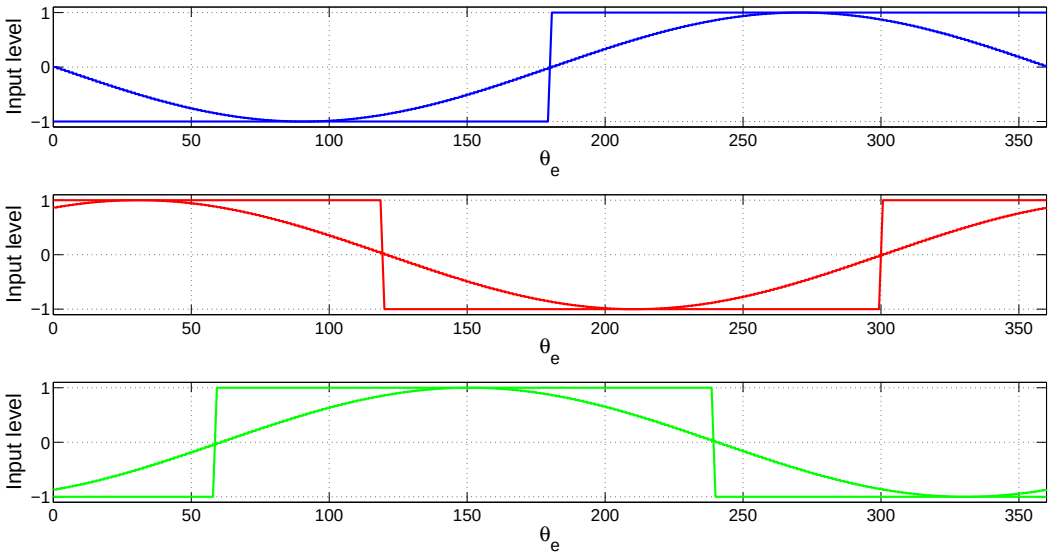


Figure 3.2: *Visualization of the signal send to a DC motor to rotate 360 electrical degrees when controlling it with a PWM signal having 255 steps and only a high or low signal*

This basic understanding of a brushless DC motor is necessary to control the gimbal. However, the gimbal needs position control instead of speed control. To accomplish this one can see that the alternating electromagnets are producing a rotating space vector which can point in virtually any direction. To calculate this vector, the following method is proposed to reduce computational time on the Arduino. In Figure 3.3 one electrical rotation is divided in six parts, where A, B and C are the voltage vectors. If voltage A is set to a PWM value of 255 and both B and C are kept zero the resulting vector is pointing in the A direction. To operate in area one, voltage A is set to the maximum PWM value while B can be varied and C is turned off. Looking at Table 3.1 shows which phases are used to operate in a certain area. Using this method means that every electrical rotation can be divided into six parts and subsequently these parts can be divided into 255 steps. Eventually every electrical rotation thus consist of 1524 steps (eliminate double scenarios) which make up one seventh of a mechanical rotation which thus consist approximately of 10.660 steps. Each step making up approximately 0.03 mechanical degrees.

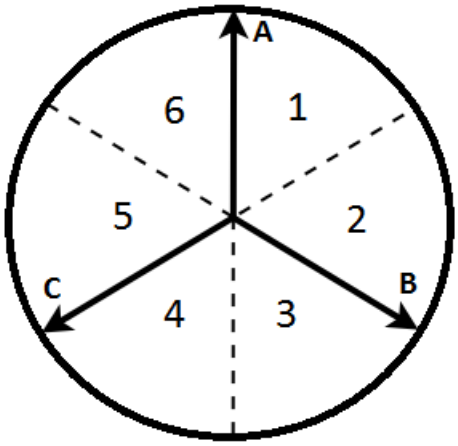


Figure 3.3: *Schematic of a brushless dc motor*

Table 3.1: *PWM values to determine the area of the vector in Figure 3.3 where X means to be calculated*

	PWM_A (PWM value)	PWM_B (PWM value)	PWM_C (PWM value)
Area 1	255	X	0
Area 2	X	255	0
Area 3	0	255	X
Area 4	0	X	255
Area 5	X	0	255
Area 6	255	0	X

The exact PWM value to create a certain resulting vector in Figure 3.3 can be determined by using geometric functions if the operating area is known. For each operating area (3.2) till (3.7) are used. Note that the PWM value in the second part of the equations is at its maximum value of 255 but this is left out to make the equations more universal.

$$\text{Area 1:} \quad \mathrm{PWM_B} = \text{round}\left(\frac{\tan(\theta_e)\mathrm{PWM_A}}{\sin(\frac{2}{3}\pi) - \cos(\frac{2}{3}\pi)\tan(\theta_e)}\right) \qquad 0 \leq \theta_e \leq 60, \tag{3.2}$$

$$\text{Area 2:} \quad \mathrm{PWM_A} = \text{round}\left(-\frac{\tan(\theta_e)\cos(\frac{2}{3}\pi)\mathrm{PWM_B} - \sin(\frac{2}{3}\pi)\mathrm{PWM_B}}{\tan(\theta_e)}\right) \qquad 60 \leq \theta_e \leq 120, \tag{3.3}$$

$$\text{Area 3:} \quad \mathrm{PWM_C} = \text{round}\left(\frac{\tan(\theta_e)\cos(\frac{2}{3}\pi)\mathrm{PWM_B} - \sin(\frac{2}{3}\pi)\mathrm{PWM_B}}{\sin(\frac{4}{3}\pi) - \cos(\frac{4}{3}\pi)\tan(\theta_e)}\right) \qquad 120 \leq \theta_e \leq 180, \tag{3.4}$$

$$\text{Area 4:} \quad \mathrm{PWM_B} = \text{round}\left(\frac{\tan(\theta_e)\cos(\frac{4}{3}\pi)\mathrm{PWM_C} - \sin(\frac{4}{3}\pi)\mathrm{PWM_C}}{\sin(\frac{2}{3}\pi) - \cos(\frac{2}{3}\pi)\tan(\theta_e)}\right) \qquad 180 \leq \theta_e \leq 240, \tag{3.5}$$

$$\text{Area 5:} \quad \mathrm{PWM_A} = \text{round}\left(\frac{\sin(\frac{4}{3}\pi)\mathrm{PWM_C} - \tan(\theta_e)\cos(\frac{4}{3}\pi)\mathrm{PWM_C}}{\tan(\theta_e)}\right) \qquad 240 \leq \theta_e \leq 300, \tag{3.6}$$

$$\text{Area 6:} \quad \mathrm{PWM_C} = \text{round}\left(\frac{\tan(\theta_e)\mathrm{PWM_A}}{\sin(\frac{4}{3}\pi) - \cos(\frac{4}{3}\pi)\tan(\theta_e)}\right) \qquad 300 \leq \theta_e \leq 360(0). \tag{3.7}$$

In (3.2) till (3.7) the electrical angle θ_e can easily be replaced by the mechanical angle θ_m when using (3.1) in the following way:

$$\theta_e = \frac{\theta_m \text{number of poles}}{2}. \tag{3.8}$$

The general theory of a brushless DC motor is now known and can be used to model the actuator and moreover the above equations will be the basis of controlling the position of the brushless DC motor. The equations will be implemented in the simulations of the movement of the gimbal but before these can be performed a model of the gimbal has to be created.

Chapter 4

Modelling the gimbal

The chosen materials and design and the general theory of a brushless DC motor are now known. This knowledge is used in this chapter to model the gimbal. First the kinematics of the gimbal will be described after which the dynamics are taken into account. Also the IMU is modelled so the movement of an UAV, on which a gimbal is attached, can be simulated. Eventually, friction is taken into account to complete the model.

4.1 Kinematics

First the kinematics of the system are derived. This means only geometry is taken into account when describing the translational and angular position and their time derivatives of the system [3]. It is assumed that the gimbal can be seen as a rigid multi body system. This makes it possible to attach coordinate frames to each body by which the motion of the body can be described. To describe the motion, individual transformations between the coordinate systems are combined. The gimbal consists out of three revolute joints with each one degree of freedom. Thus the gimbal has in total three degrees of freedom, roll, pitch and yaw. These angles are used to describe the system. In figures 4.1 and 4.2 the front and side view of the gimbal are given respectively. Note that also the individual coordinate systems are shown.

4.1.1 Transformation matrices

To describe the transformation from one frame to another, both a rotation matrix and a translational vector are introduced. For rotations around the x, y and z axis, standard rotation matrices are given as:

$$
\begin{aligned}
R_x(\theta_3) = R_3^2(\theta_3) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_3 & -\sin\theta_3 \\ 0 & \sin\theta_3 & \cos\theta_3 \end{bmatrix}, \\
R_y(\theta_2) = R_2^1(\theta_2) &= \begin{bmatrix} \cos\theta_2 & 0 & \sin\theta_2 \\ 0 & 1 & 0 \\ -\sin\theta_2 & 0 & \cos\theta_2 \end{bmatrix}, \\
R_z(\theta_1) = R_1^0(\theta_1) &= \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix},
\end{aligned}
\tag{4.1}
$$

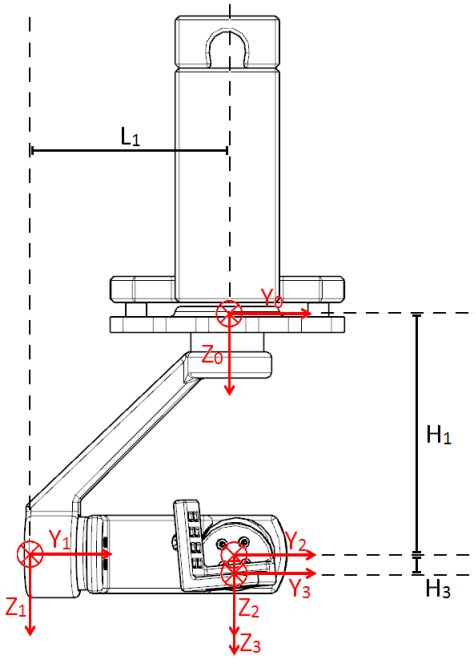


Figure 4.1: *Side view of the gimbal with axes attached*

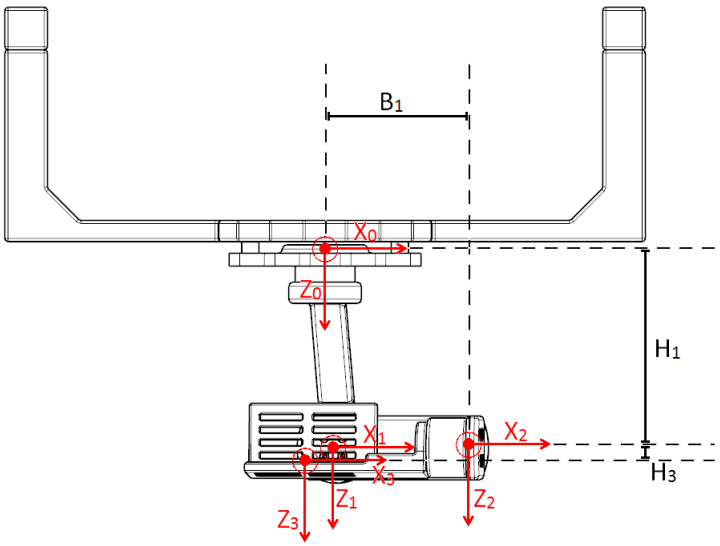


Figure 4.2: *Front view of the gimbal with axes attached*

where the superscript and subscript indicate the rotation from one frame to the other respectively. The translational vectors are equal to the distance between the coordinate frames and are given as:

$$\begin{aligned} d_1^0(\theta) &= \begin{bmatrix} L_1 \sin\theta_1 \\ L_1 \cos\theta_1 \\ H_1 \end{bmatrix}, \\ d_2^1(\theta) &= \begin{bmatrix} B_1 \cos\theta_2 \\ L_1 \\ B_1 \sin\theta_2 \end{bmatrix}, \\ d_3^2(\theta) &= \begin{bmatrix} -B_1 \\ H_3 \sin\theta_3 \\ H_3 \cos\theta_3 \end{bmatrix}. \end{aligned} \tag{4.2}$$

Combining both the rotational matrix and the translational vector results in the homogeneous transformation matrix from frame zero to one as:

$$T_1^0 = \begin{bmatrix} R_1^0 & d_1^0 \\ 0 & 1 \end{bmatrix}. \tag{4.3}$$

Knowing all homogeneous transformation matrices gives the resulting homogeneous transformation matrix from frame zero to three as:

$$T_3^0 = T_1^0 T_2^1 T_3^2 = \begin{bmatrix} R_3^0 & d_3^0 \\ 0 & 1 \end{bmatrix}, \tag{4.4}$$

where R_3^0 is:

$$R_3^0 = \begin{bmatrix} c_1c_2 & c_1s_2s_3 - c_3s_1 & s_1s_3 + c_1c_3s_2 \\ c_2s_1 & c_1c_3 + s_1s_2s_3 & c_3s_1s_2 - c_1s_3 \\ -s_2 & c_2s_3 & c_2c_3 \end{bmatrix}, \tag{4.5}$$

and d_3^0 is:

$$d_3^0 = \begin{bmatrix} c_1c_3H_3s_2 - H_3s_1s_3 \\ 2c_1L_1 + c_1H_3s_3 + c_3H_3s_1s_2 \\ H_1 + 2B_1s_2 + c_2c_3H_3 \end{bmatrix}. \tag{4.6}$$

In (4.5) and (4.6), c and s are equal to cos and sin respectively. The subscript of c or s indicates the angle, θ_1, θ_2 or θ_3. The kinematics of the gimbal are determined so the next step is to introduce the dynamics.

4.2 Dynamics

To derive a dynamic model for a rigid body which is subject to kinematic constraints the Newton-Euler approach or the Lagrangian approach can be used. In this case the Lagrangian approach is used to describe the dynamics of the gimbal [4]. First the dynamics of the overall gimbal will be presented after which the dynamics of the actuators is introduced.

4.2.1 Gimbal

The gimbal can be seen as an n joints robot. Describing a robot with n joints following the Lagrange equation:

$$Q_i = \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \omega_i}\right) - \frac{\partial \mathcal{L}}{\partial \theta_i} \quad i = 1, 2, \ldots, n. \tag{4.7}$$

Here $\mathcal{L}$ is the Lagrangian which is defined by:

$$\mathcal{L} = K - V, \tag{4.8}$$

where K is the kinetic energy and V is the potential energy. Furthermore, θ_i are the angles, roll, pitch and yaw, of the gimbal and ω_i the derivatives of the angles. The variable Q_i denotes the generalised non-conservative forces such as the torques from the actuators. Equation 4.7 can also be expressed in matrix form:

$$Q = D(\Theta)\ddot{\Omega} + H(\Theta, \Omega) + G(\Theta), \tag{4.9}$$

with D being the inertial-type matrix, H being the velocity coupling vector and G the gravitational vector. Since the link transformation matrices are known, (4.9) can be written in summation form:

$$Q_i = \sum_{j=1}^{n} D_{ij}(\theta)\ddot{\theta}_j + \sum_{j=1}^{n}\sum_{k=1}^{n} H_{ikm}\dot{\theta}_k\dot{\theta}_m + G_i. \tag{4.10}$$

The inertial-type matrix is written as:

$$D_{ij} = \sum_{r=\max(i,j)}^{n} \operatorname{tr}\left(\frac{\partial\, {}^0T_r}{\partial \theta_i}\, {}^r\bar{I}_r \left(\frac{\partial\, {}^0T_r}{\partial \theta_j}\right)^T\right), \tag{4.11}$$

the velocity coupling as:

$$H_{ijk} = \sum_{r=\max(i,j,k)}^{n} \operatorname{tr}\left(\frac{\partial^2\, {}^0T_r}{\partial \theta_j \partial \theta_k}\, {}^r\bar{I}_r \left(\frac{\partial\, {}^0T_r}{\partial \theta_i}\right)^T\right), \tag{4.12}$$

and the gravitational vector as:

$$G_i = -\sum_{r=i}^{n} m_r g^T \frac{\partial\, {}^0T_r}{\partial \theta_i}\, {}^r\mathbf{r}_r, \tag{4.13}$$

where tr is the trace of a matrix, ${}^r\mathbf{r}_r$ is the translational position of the center of mass of body r expressed in frame r. The matrix ${}^r\bar{I}_r$ is the pseudo inertia matrix of body r expressed in frame r. This matrix is compatible with the homogeneous transformation matrix and is given as:

$$\bar{I} = \begin{bmatrix} \frac{-I_{xx}+I_{yy}+I_{zz}}{2} & I_{xy} & I_{xz} & mx \\ I_{yx} & \frac{I_{xx}-I_{yy}+I_{zz}}{2} & I_{yz} & my \\ I_{zx} & I_{zy} & \frac{I_{xx}+I_{yy}-I_{zz}}{2} & mz \\ mx & my & mz & m \end{bmatrix}, \tag{4.14}$$

with x,y and z the coordinates of the body's centre of mass expressed in frame r. The mass moment of inertia of body r is expressed as I_{ii}. Note that the inertia of the camera is already included in frame three. The equations in this section describe the dynamics of the overall gimbal. In the next section the dynamics of the actuators is introduced.

4.2.2 Actuator

As a part of the total dynamics, the actuator dynamics is described in this section. A model of the actuator, a brushless dc motor, is developed based on the rotor reference frame. This frame is called a dq-frame, which is the translation of the three vectors of a three phase motor into a 2D space vector representation. This transformation is done by first transforming to a static frame, called $\alpha\beta$ frame, using the Clarke transformation matrix [5] :

$$\begin{bmatrix} v_\alpha \\ v_\beta \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \end{bmatrix} \begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix}. \tag{4.15}$$

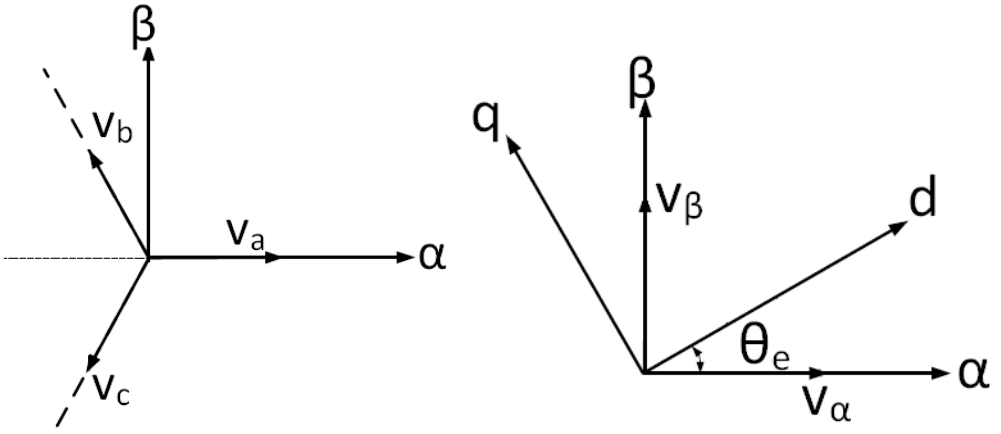
To transform from this static $\alpha\beta$ frame to a dynamic dq frame which is attached to the rotor, the Park transformation matrix [5] is used:

$$\begin{bmatrix} d \\ q \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \tag{4.16}$$

combining both the Clark and Park transformation matrices results in:

$$\begin{bmatrix} V_q \\ V_d \end{bmatrix} = \frac{2}{3} \begin{bmatrix} \cos(\theta_e) & \cos(\theta_e - \frac{2}{3}\pi) & \cos(\theta_e - \frac{4}{3}\pi) \\ -\sin(\theta_e) & -\sin(\theta_e - \frac{2}{3}\pi) & -\sin(\theta_e - \frac{4}{3}\pi) \end{bmatrix} \begin{bmatrix} V_a \\ V_b \\ V_c \end{bmatrix}. \tag{4.17}$$

The Clarke transformation is shown in Figure 4.3(a), from the three vector representation to the static two vector representation. The transformation from the static $\alpha\beta$ frame to the dynamic dq frame is shown in Figure 4.3(b).



(a) Transformation from the three vectors representation to the two vector representation

(b) Transformation between the static $\alpha\beta$ frame to the dq frame

Figure 4.3: *Visualisation of the Clark and Park transformation respectively*

By using (4.17), v_q and v_d can be expressed as:

$$\begin{aligned} v_q &= R_s i_q + L_s \frac{\delta i_q}{\delta t} + \omega_e L_s i_d + \omega_e \phi_{mg}, \\ v_d &= R_s i_d + L_s \frac{\delta i_d}{\delta t} - \omega_e L_s i_q, \end{aligned} \tag{4.18}$$

where R_s is the electrical resistance in Ohm, i the current in q or d direction in Ampere, L_s the inductance in Ohm seconds, ω_e the electrical velocity in radians per second and ϕ_{mg} is the flux

induced by the permanent magnets of the rotor in weber. For a surface mounted Permanent Magnet Synchronous Motor (PMSM) the inductances are equal to each other:

$$L_d = L_q = L_s. \tag{4.19}$$

The electromagnetic torque can be determined by:

$$T_e = \frac{3}{2}p(\phi_{mg} i_q), \tag{4.20}$$

where p is the amount of pole pairs. The equations above are used to determine the complete model of a PMSM. The rotation of a motor can be described by this dynamic equation:

$$J_m \frac{\delta \omega_m}{\delta t} = T_e - B_v \omega_m - T_L, \tag{4.21}$$

where J_m is the total inertia in kilogram square meter, B_v the viscous friction coefficient in Newton meter second and T_L the load torque in Newton meter. Replacing the mechanical velocity with the electrical velocity and rewriting (4.18) results in:

$$\begin{aligned} \frac{\delta i_d}{\delta t} &= \frac{1}{L_d}(v_d - R_s i_d + \omega_e L_q i_q), \\ \frac{\delta i_q}{\delta t} &= \frac{1}{L_q}(v_q - R_s i_q - \omega_e L_d i_d - \omega_e \phi_{mg}), \\ \frac{\delta \omega_e}{\delta t} &= \frac{p}{J_m}(\frac{3}{2}p\phi_{mg} - \frac{B_v}{p}\omega_e - T_L), \qquad \omega_e = p\omega_m. \end{aligned} \tag{4.22}$$

The equations above are used to describe the dynamics of the actuators. In the next part the influence of the movement of an UAV on the readings of the IMU is modelled.

4.3 IMU

The influence of the movement of an UAV on the readings of the IMU is modelled in this section. First the readings of the IMU without the zero frame moving are determined. This is done by describing the angular velocity of the IMU, using the local angular velocities of the joints [4]:

$$\Phi = J\Omega, \tag{4.23}$$

with Ω being:

$$\Omega = \begin{bmatrix} \omega_1 & \omega_2 & \omega_3 \end{bmatrix}^T. \tag{4.24}$$

Because all the joints are revolute, the Jacobian used in (4.23) can be expressed using the rotation matrices:

$$J = \begin{bmatrix} \left(R_2^1 R_3^2\right)^T \theta_1^1 & R_3^{2T}\theta_2^2 & \theta_3^3 \end{bmatrix}, \tag{4.25}$$

where θ_1^1, θ_2^2 and θ_3^3 are expressed as:

$$\theta_1^1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \theta_2^2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \qquad \theta_3^3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}. \tag{4.26}$$

Substituting (4.25) and (4.26) in (4.23), results in:

$$\begin{bmatrix} \phi_{\text{roll}} \\ \phi_{\text{pitch}} \\ \phi_{\text{yaw}} \end{bmatrix} = \begin{bmatrix} -\sin(\theta_2) & 0 & 1 \\ \cos(\theta_2)\sin(\theta_3) & \cos(\theta_3) & 0 \\ \cos(\theta_2)\cos(\theta_3) & \sin(\theta_3) & 0 \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix}. \tag{4.27}$$

To simulate the gimbal hanging underneath an UAV, the readings of the IMU should be included in the model. This is done by translating the velocities acting on the zero frame to the IMU, which is close to frame three. When the velocities acting on frame zero are interpreted as if the joints produce them while the zero frame is fixed, gives:

$$\Omega_\Phi = J_{\text{joints}}\Phi_0, \tag{4.28}$$

where the Jacobian, J_{joints}, is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_1) & -\sin(\theta_1) \\ -\sin(\theta_2) & \cos(\theta_2)\sin(\theta_1) & \cos(\theta_1)\cos(\theta_2) \end{bmatrix}. \tag{4.29}$$

Eventually these angular velocities are combined with the following equation, with the Jacobian from (4.25),

$${}^3_3\Phi = J(\Omega_\phi + \Omega). \tag{4.30}$$

Furthermore, white noise is added since the measurements of the gyroscope and accelerometer are not noise free. Note that the I2C library of Jeff Rowberg [2] uses quaternions to avoid the problem of singularities, also known as gimbal lock, which is a known problem with Euler's rotation theorem [6]. A singularity occurs when two rotational axes align and thus a degree of freedom is lost. This is solved by using Quaternions. The quaternion is an extension of the complex number and is written as a scalar plus a vector denoted as (4.31),

$$q = q_0 + q_1 i + q_2 j + q_3 k, \qquad i^2 = j^2 = k^2 = ijk = -1, \tag{4.31}$$

describing a single rotation around a specific axis. Having four parameters to describe the angles results in not having gimbal lock. In the model it is assumed that gimbal lock will not occur since the angles made by the gimbal are relatively small, less than 40°. The readings of the IMU hanging underneath an moving UAV are now modelled. In the next section the influence of friction on the gimbal is described.

4.4 Friction

This section describes the influence of friction on the gimbal. The torques, Q, are influenced by friction which in this case is assumed to be linear but this should be validated when further research is conducted. The torques Q can be written as:

$$Q = \tau_a - \tau_s \text{sgn}(\Omega) - \tau_d \Omega. \tag{4.32}$$

Here τ_a is the torque from the actuators, τ_s the static friction of the joints and τ_d the dynamic friction. Both τ_s and τ_d can differ per joint and direction. The sgn function indicates the sign of τ_s.

This chapter describes the model of the gimbal. The materials and design of Chapter 2 and the PWM equations of Chapter 3 can now be used to simulate the movement of the gimbal, which is done in the next chapter.

Chapter 5

Simulation model

In this chapter a simulation model in `Simulink` is created which can be used to simulate the movement of the gimbal. The materials and design of Chapter 2, the PWM equations of Chapter 3 and the model of Chapter 4 are used for the implementation in `Simulink`. In the next section the implementation of the model and parameters in `Simulink` is explained. The simulation model will be used in the next chapter to compare measurement results with simulation results.

5.1 Implementation

The `Simulink` model is divided into several blocks, which represent a subsystem. Every block of the model will be treated separately and the total implemented model can be found in Figure 5.1. First the inputs of the model are explained after which the implementation of the PWM converter is shown. Then the implementation of the DC motor is treated. Furthermore, the implementation of the gimbal block is explained. Lastly, the controller is implemented and explained.

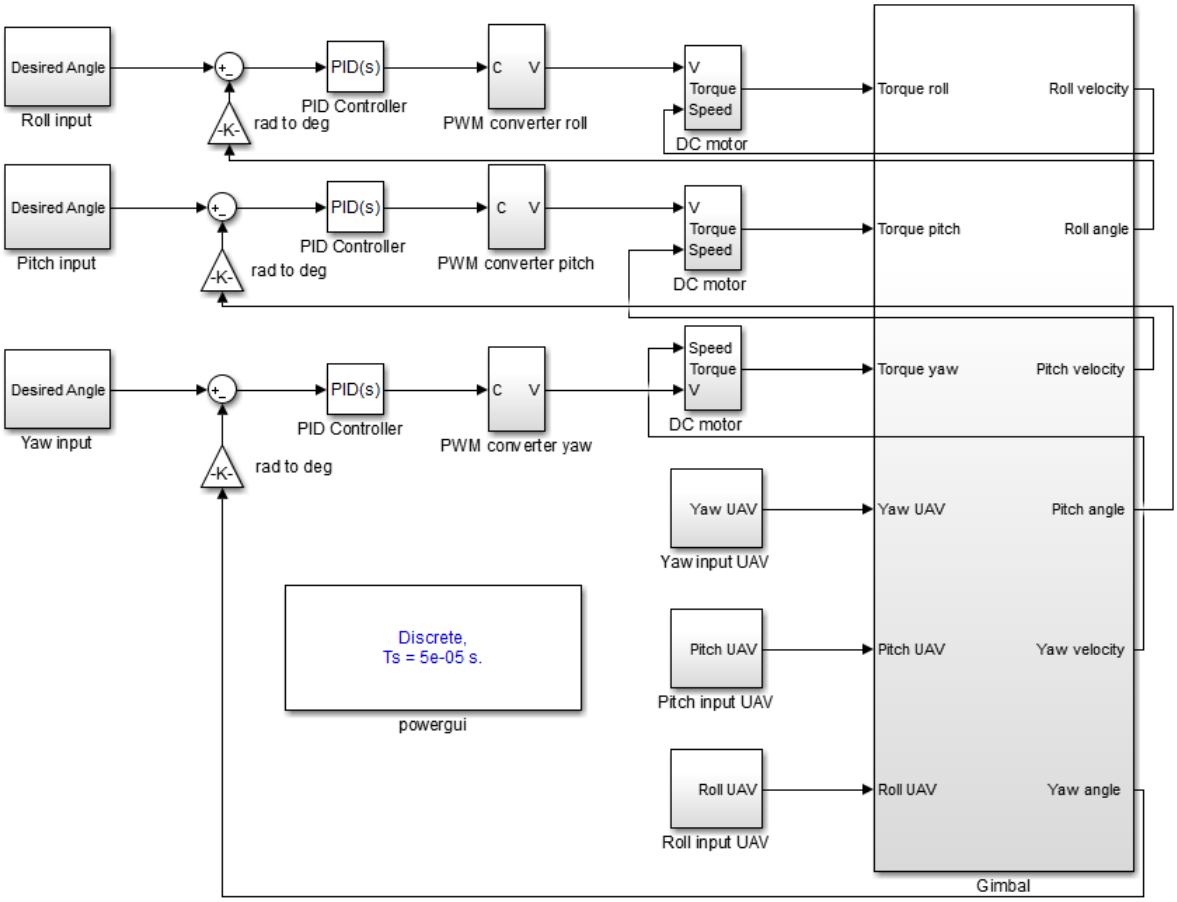


Figure 5.1: `Simulink` *model to simulate the motion of a gimbal*

The inputs are the desired angles, roll, pitch and yaw, of the gimbal and the angles made by the UAV. So in total there are six inputs to the model. An example of an input subsystem is shown in Figure 5.2. The input angles for the gimbal will generally be constants but it is possible to create a trajectory for each individual angle. The input angles for the UAV will generally be trajectories to mimic a flight of an UAV. Note that all the input angles are in degrees.

Figure 5.2: *An example of an input subsystem to the model*

The PWM converter is purely the implementation of (3.2) till (3.7) from Chapter 3 in a `Matlab` function, which can be found in Appendix A. Using if statements the right equation is selected and the corresponding PWM values are determined. The input of the function is the controlled roll, pitch or yaw angle of the gimbal and the output is a vector containing the three PWM values and the phase corresponding to these values. In Figure 5.3 the PWM converter subsystem is shown.

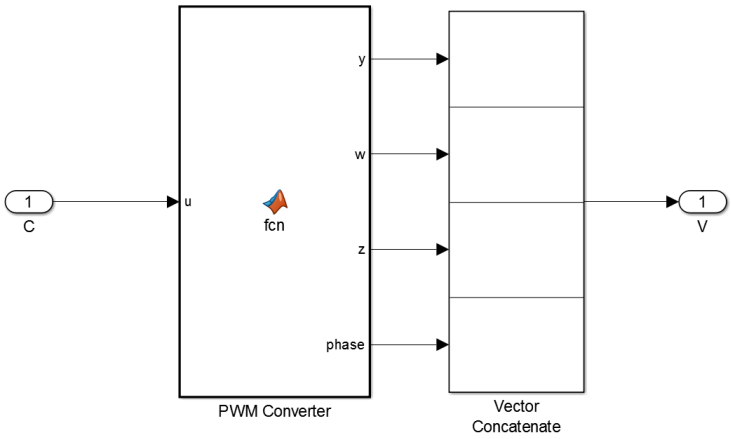


Figure 5.3: *The PWM converter subsystem in the* `Simulink` *model*

The implementation of the DC motor is a bit more complicated, as can be seen in Figure 5.4. First every output PWM signal of the PWM converter, so three per motor, is converted to a voltage. This conversion is done using the *Controlled Voltage Source* block in `Simulink`, with as source type DC and initial amplitude zero Volt. The input is the PWM value of the PWM converter, the minus side is connected to the ground and the plus side is the output voltage which is connected to the DC motor. For the DC motor the *Permanent Magnet Synchronous Machine* (PMSM) is used in `Simulink`. The configuration of the PMSM is standard with three phases, a trapezoidal back EMF waveform and speed as mechanical input. Since no parameters of the DC motor are known also the parameters of the PMSM are standard and thus unchanged.

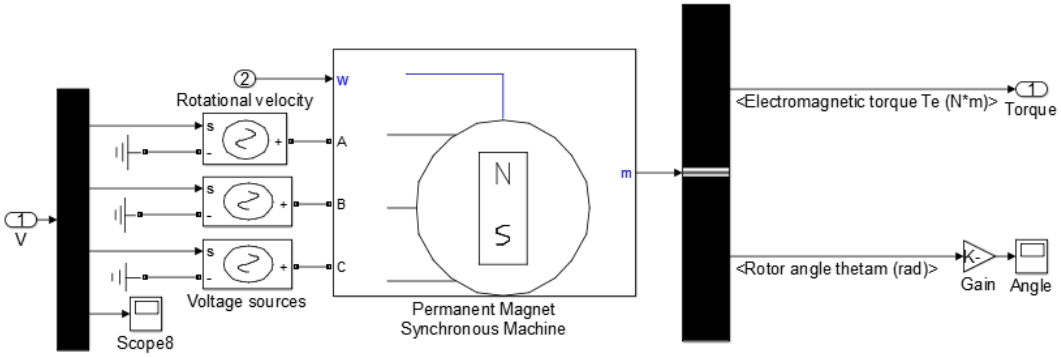


Figure 5.4: *The DC motor subsystem in the* `Simulink` *model*

The implementation of the gimbal subsystem is done by dividing this subsystem into two separate subsystems, as shown in Figure 5.5. One subsystem is the IMU and the other subsystem is the dynamics of the gimbal. The inputs to the IMU are the roll, pitch and yaw angles of the UAV and the gimbal and angular velocities of the gimbal. The output of the IMU are the measured roll, pitch and yaw angles of the gimbal relative to the fixed world. The inputs to the dynamics of the gimbal are the vector of measured angles from the IMU and the angular velocities and torques of the gimbal.

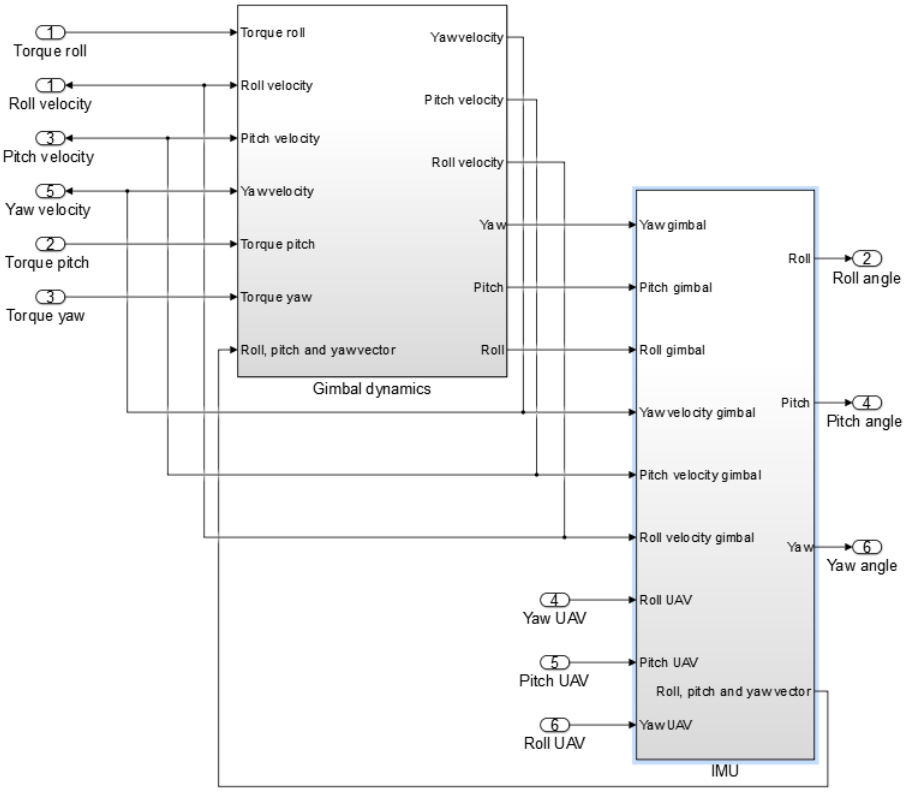


Figure 5.5: *The subsystem of the gimbal, including a subsystem for the dynamics and the IMU*

The IMU subsystem is shown in Figure 5.6. The `Matlab` function consists of (4.23) till (4.30) from Chapter 4 and can be found in Appendix B. The output of the `Matlab` function are the ideal measured angles. These angles are disturbed by adding white noise since this is a reasonable assumption.

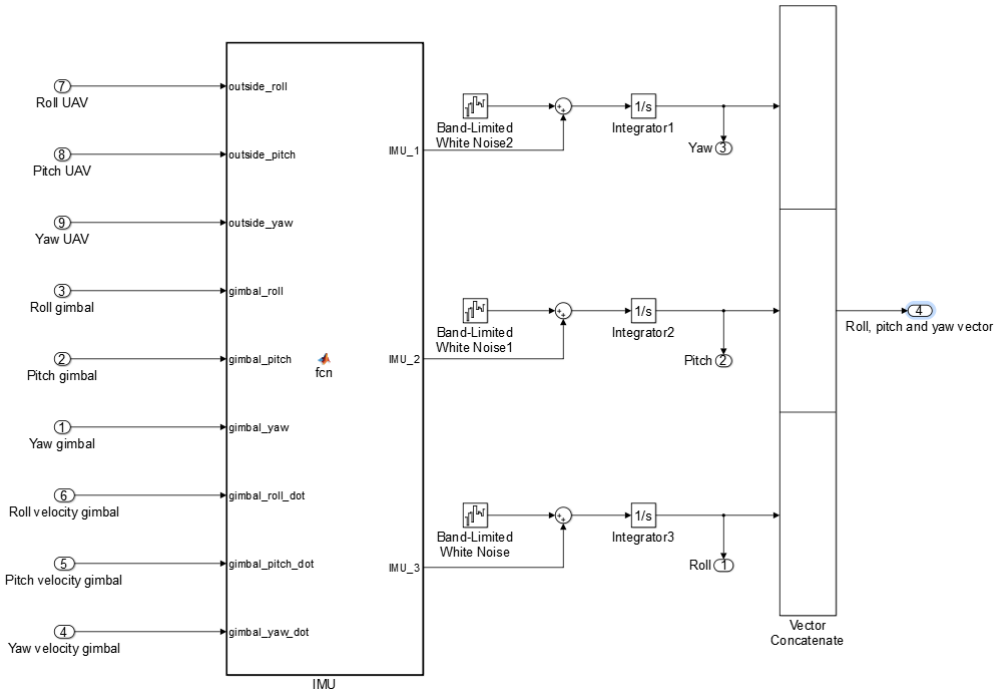


Figure 5.6: *The subsystem of the IMU*

The dynamics of the gimbal are calculated in the subsystem shown in Figure 5.7. The `Matlab` function contains (4.7) till (4.14) and (4.32) and can be found in Appendix C.

The controller in the simulation model is a relatively simple PID controller, which is only used in the position loop to control the gimbal, as can be seen in Figure 5.1. The PID controller is the standard *PID Controller* block from `Simulink`. It is chosen to only control the position loop due to the ease of implementation and limited time frame of this project. A PID controller is proposed since it is commonly used in the industry [7].

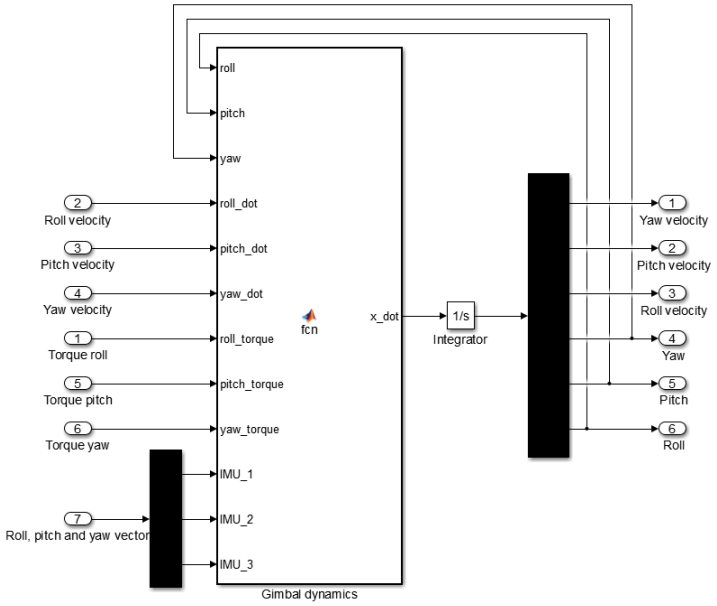


Figure 5.7: *The subsystem of the gimbal dynamics*

All the values such as the inertias, masses and distances to the centres of mass are extracted out of the 3D model in Siemens NX9.0. The values for the static and dynamic friction are assumed to be small, especially the static friction since no brushess are present.

In this chapter the implementation of the model from Chapter 4 in `Simulink` is explained. This `Simulink` model can now be used to simulate the movement of the gimbal. The simulation results will be compared with experimental results. The next chapter explains how the experimental data is acquired.

Chapter 6

Acquirement of experimental data

In this chapter it is explained how the experimental data is acquired. This experimental data will be used in the next chapter to compare the experimental results to the simulation results. First the materials described in Chapter 2 are combined to be able to control the actuators of the gimbal. After this the measurement data of the MPU6050 is analysed for drifting of the yaw, since this is a known problem. Then one single actuator is controlled to check if the implemented algorithm on the Arduino works. Eventually all three axis of the gimbal are controlled by an extended version of the algorithm for one axis. To obtain stable parameters for each P, I and D value the Ziegler-Nichols tuning method is used.

6.1 Control of the actuators of the gimbal

To connect all the materials described in Chapter 2 initially a breadboard together with a lot of wires is used. This is done to test a simple code on the Arduino and to check if every component is functioning properly and to get familiar with coding on the Arduino. Coding on the Arduino is done in `C++`. To implement all the devices compactly, a Printed Circuit Board (PCB) is designed in Eaglecad. The PCB is designed such that an Arduino, PWM driver, three motor drivers, two capacitors, one MOSFET and a connection to the MPU6050 fit on it. Figure 6.1 shows the design, where the blue lines lay on the bottom side of the PCB and the red lines on top of the PCB. These lines are the connections between the components and are replacing all the cables which were originally used on the breadboard.

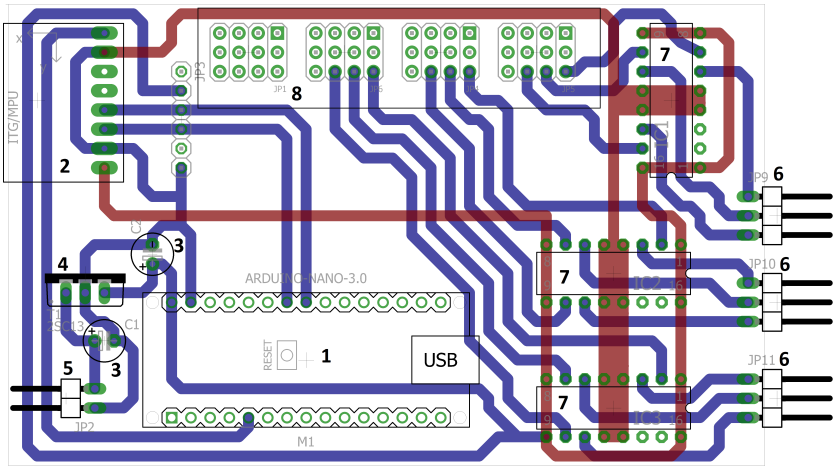


Figure 6.1: *PCB design made in Eaglecad*

Each component on the PCB is numbered, as can be seen in Figure 6.1. Number one is the Arduino, number two is the MPU6050, number three is the capacitor, number four the capacitor, number five a two-pins connector for the battery, number six a three-pins connector to connect an actuator, number seven the H-bridge and number eight the PWM shield.

6.1.1 Yaw drift

The data of the MPU6050 is analysed on drifting angles. This is done after calibrating the device, which is important to get the correct data. The roll, pitch and yaw angles are measured while the MPU6050 was not moving. The results are shown in Figure 6.2, which shows the switching transient for approximately the first 1800 measurement cycles, corresponding to roughly the first 18 seconds. The update speed of the MPU6050 lays around 100 Hz, the sample rate is thus 100 Hz, so 100 measurement cycles correspond roughly to 1 second. After 18 seconds the the roll, pitch and yaw seem to stabilize.

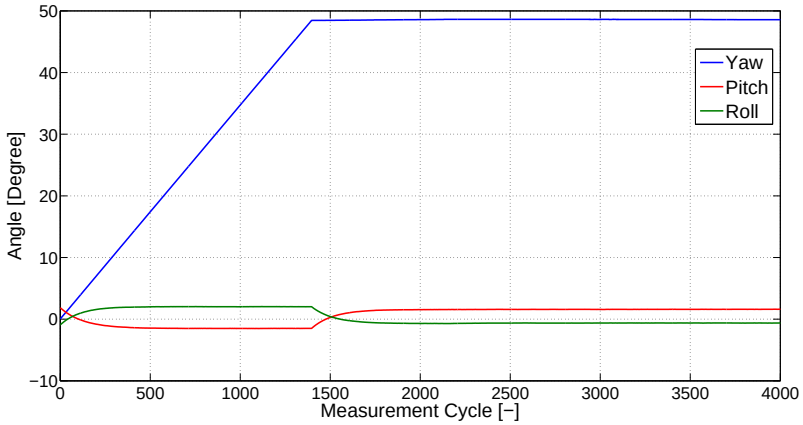


Figure 6.2: *Roll, pitch and yaw with switching transient*

When taking a closer look at the yaw angle, it can be seen that it is drifting as shown in Figure 6.3. This figure shows measurement data after the MPU6050 measured for twenty seconds and the data is translated to zero at the first measurement cycle after these twenty seconds. It can be seen that the drifting has a random slope which is maximal one degree per 14000 measurement cycles, roughly 140 seconds. Since the typical flight time of a small UAV is in the order of minutes, it is assumed that the drifting is too slow to noticeably disturb the image made by the device attached to the gimbal.

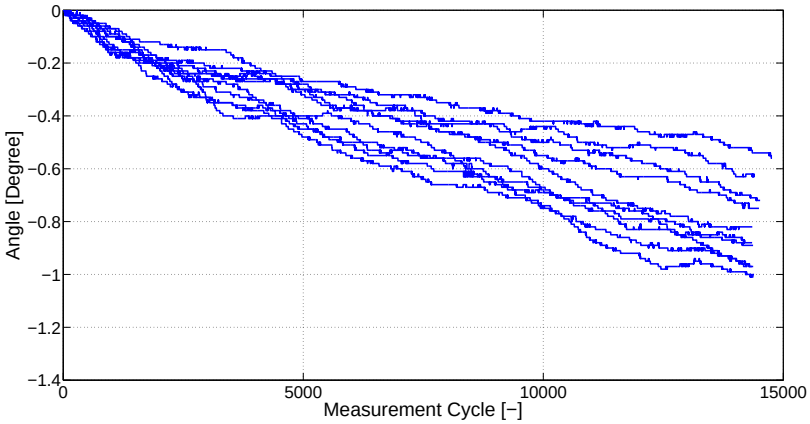


Figure 6.3: *Drifting yaw after stabilization of the MPU6050 and translated to zero at the start*

6.1.2 Single axis

A single axis is first tested during open-loop operation and here after during closed-loop operation. The code used for both modes of operation is found in Appendix D and implemented on the Arduino. Instead of testing directly on the gimbal, a test rig is used to test the control algorithm implemented on the Arduino. The test rig consists of a rigid base where to the motor is mounted, on top of the motor a platform is mounted where a breadboard and the MPU6050 sits on.

During open-loop operation the system is operating as expected. Putting a ramp as input, resulted in movement of the gimbal as would be expected. In Figure 6.4 the open-loop measurement of a decreasing ramp as input is shown. This shows that the system is working as desired and the closed-loop algorithm can be implemented.

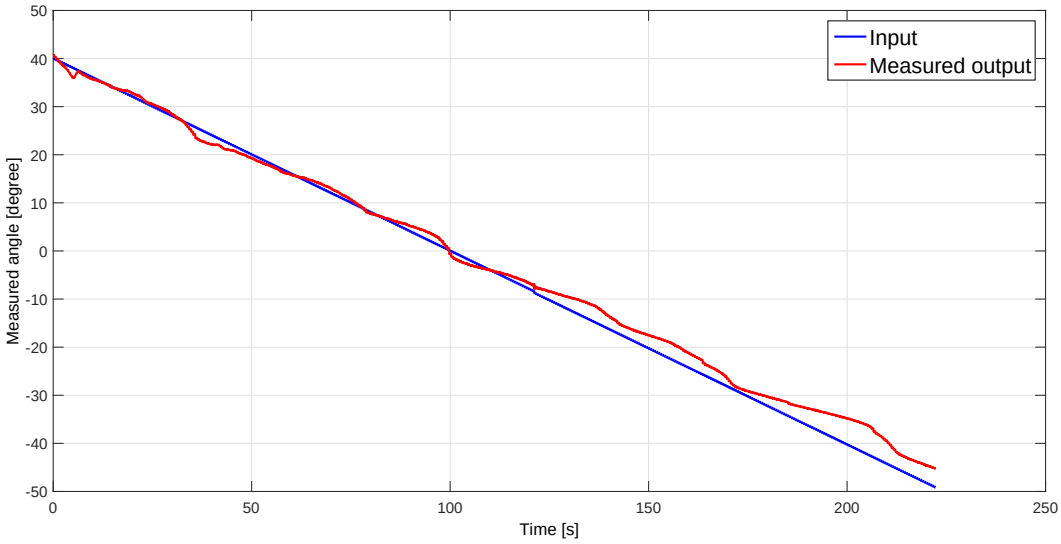


Figure 6.4: *Open-loop ramp down measurement, where the blue line is the input signal and the red line is the measured output*

After implementing the closed-loop control algorithm, without a controller, on the Arduino and testing it on a single motor, severe unstable behaviour is seen. Therefore a PID controller, which is commonly used in the industry, is proposed [7]. The control scheme for a single axis can be seen in Figure 6.5. In this scheme $\theta_{m,r}$ is the desired mechanical angle while θ_m is the actual measured angle. Since a PID controller is used, an anti-windup compensation ensures no integral windup will occur. Note that normally cascade control is used to control the position of a PMSM [5]. The inner loop current control is in this case not possible due to the lack of sensors and knowledge of the PMSMs. Furthermore the velocity loop is not integrated because of the late arrival of the PMSMs and Arduino code which had to prove to work.

A controller is manually tuned to test the control algorithm and after this it is implemented on the gimbal with the same controller values. The closed loop step responses for the roll are shown in Figure 6.6. The PID controller has the values 0.15, 9 and 0.0025 for the P,I and D respectively. As can be seen the step trajectory is roughly followed by the system.

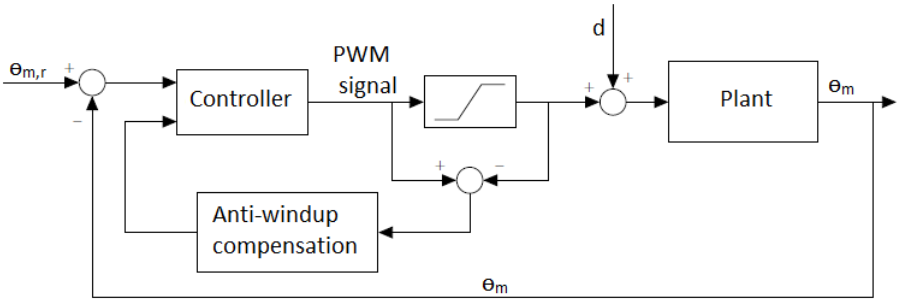


Figure 6.5: *Control scheme to control a single axis*

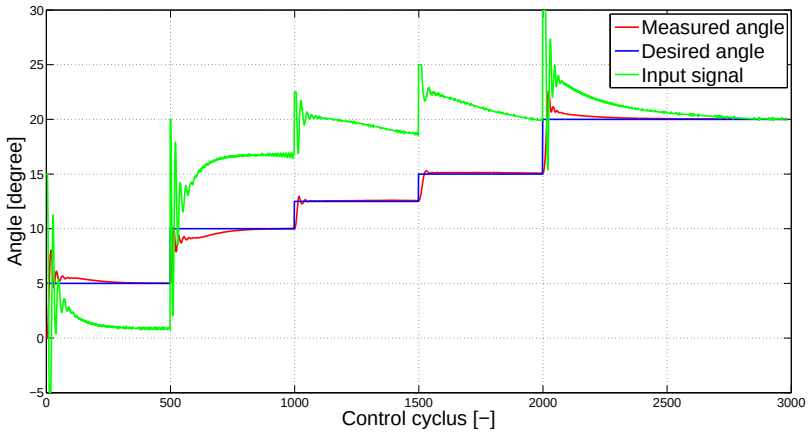


Figure 6.6: *Response of the system following a reference trajectory*

6.1.3 Three axis

Controlling three axes of the gimbal is done in a similar fashion as controlling one single-axis. The control scheme shown in Figure 6.5 is applied to all three axes. Instead of trail-and-error, the Ziegler-Nichols closed-loop tuning method is implemented [8]. This method only applies a gain as controller, so the integral and derivative action are discarded. By creating a small disturbance in the loop, for instance by changing the set point or disturbing the system by hand, the system begins to oscillate. The gain is tuned to the value K_u where the oscillations become constant having a period P_u. If K_u and P_u are known, the P, I and D values of the Ziegler-Nichols method can be determined following Table 6.1.

Table 6.1: *Closed-loop calculations of P, I and D*

	P	I	D
P	$\frac{K_u}{2}$		
PI	$\frac{K_u}{2.2}$	$\frac{P}{\frac{P_u}{1.2}}$	
PID	$\frac{K_u}{1.7}$	$\frac{P}{\frac{P_u}{2}}$	$\frac{PP_u}{8}$

After finding the right PWM values to set the yaw, pitch and roll axis to approximately zero degrees, the Ziegler-Nichols closed-loop tuning method is used to determine the initial values of the PID controller for the roll axis. Figure 6.7 shows a change of set point and consequently an approximately constant oscillation is observed. The value of K_u is 0.3 and P_u is 0.11 seconds which can be seen in the figure. Note that one measurement cycle makes up 0.01 seconds. This results in the initial values of 0.1765, 3.2086 and 0.0024 for respectively P, I and D. Later on these values will be tuned manually to achieve a better controller.

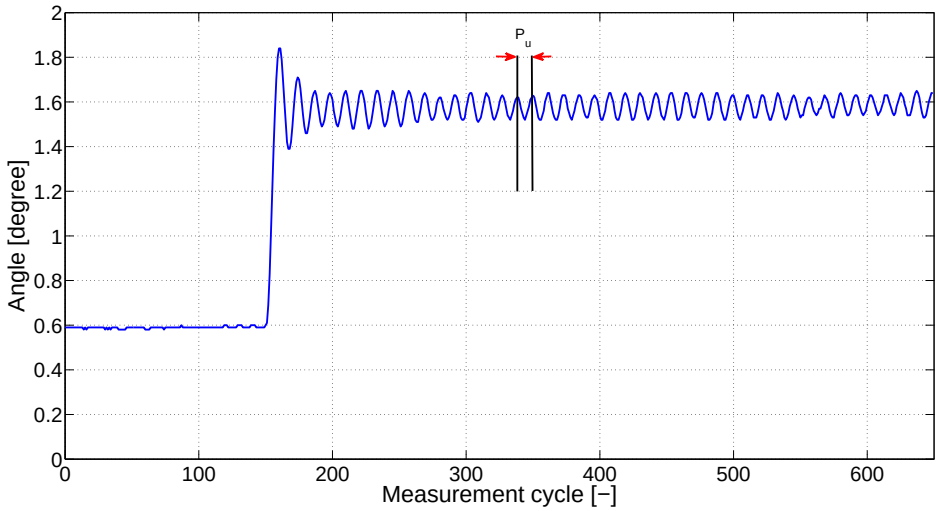


Figure 6.7: *Change of set point of the roll axis*

Here after the pitch is tuned but now the set point is kept constant and an external disturbance, in the form of a push, is applied. This results in Figure 6.8. Where K_u is 0.4 and P_u is 0.31 seconds which results in the initial PID values of respectively 0.2353, 1.5180 and 0.0091. The same is done for the yaw axis which results in a K_u of 0.45 and a P_u of 0.3, giving the initial PID values of respectively 0.2647, 1.7647 and 0.0099.

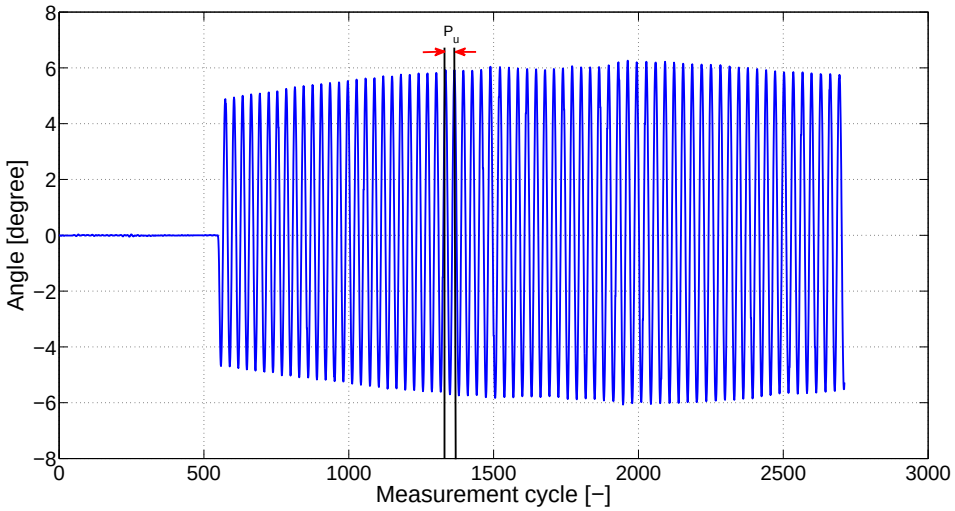


Figure 6.8: *External disturbance on the pitch axis*

After implementing all three initial PID controllers, a step response for every axis separately is done. Figure 6.9 shows the results, note that although the results are in one plot, the measurements are all done separate. As can be seen the roll, pitch and yaw are still vibrating for almost three seconds after a step of only one degree. Especially the roll and yaw have significant overshoot.

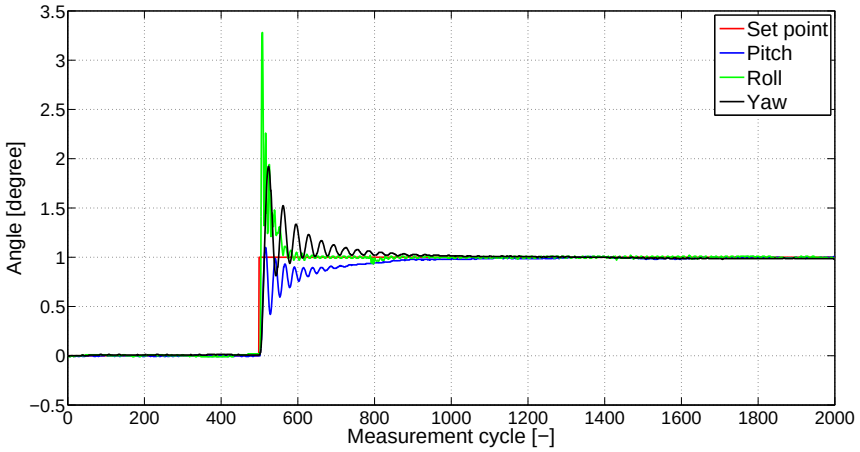


Figure 6.9: *Step response of every individual axis combined in one figure*

Manually tuning each axis should give better results but no significant improvement is seen. To achieve better results it is recommended to add at least the velocity control loop and for further improvement also the most inner control loop of the current. Unfortunately real time measurement of each axis when mimicking the motion of an UAV was not possible because this would slow down the sample rate resulting in an unstable system.

In this chapter it is explained how the experimental data is acquired. First the materials described in Chapter 2 are combined to be able to control the actuators of the gimbal. After this the measurement data of the MPU6050 is analysed for drifting of the yaw and one single actuator is controlled to confirm if the implemented algorithm on the Arduino works. Eventually all three axis of the gimbal are controlled by an extended version of the algorithm for one axis. To obtain stable parameters for each P, I and D value the Ziegler-Nichols tuning method is used. In the next chapter the simulation and experimental results of both the open-loop and closed-loop are compared.

Chapter 7

Comparison experimental and simulation data

In this section the results obtained in Chapter 5 and Chapter 6 of respectively the simulations and experiments are compared. First the open-loop results are compared after which the closed-loop results are compared.

7.1 Open-loop comparison

A simple open-loop comparison is done to check if the output of the simulation model corresponds to the measured output. The roll of both the model and measurement is shown together with the actual desired roll in figures 7.1 and 7.2. The two figures are made because differences can occur in moving in the positive and negative roll direction. It can be seen that the simulation approximates the real measurement, especially at smaller angles. At larger angles the measurement and simulation data start to differ, especially when looking at the negative direction plot.

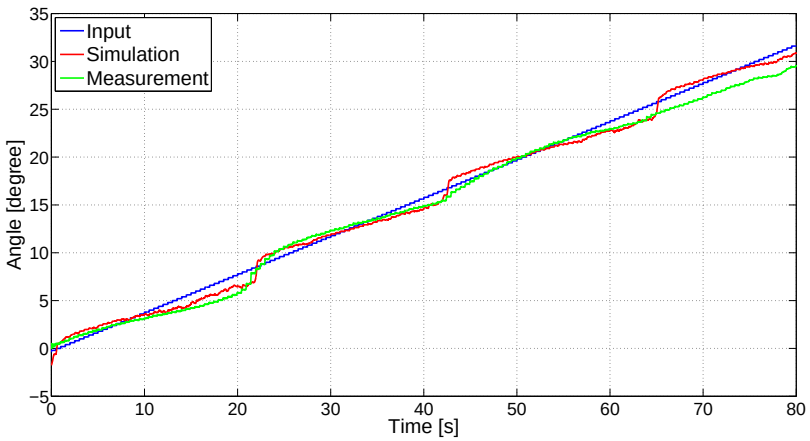


Figure 7.1: *Simulation and real measurement of the roll rotation of the gimbal in positive direction*

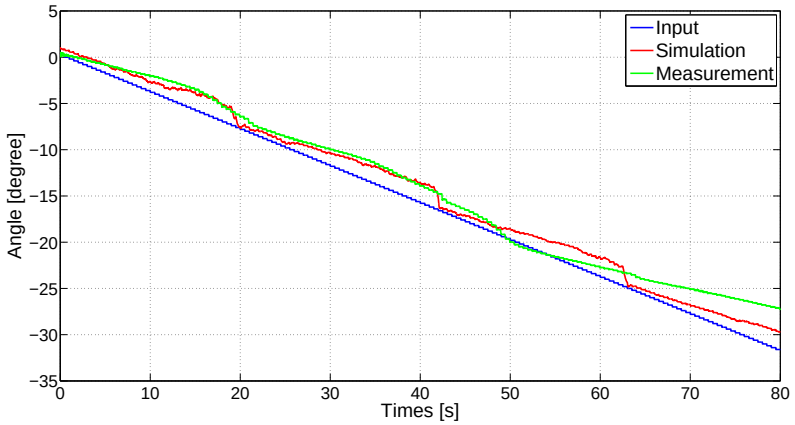


Figure 7.2: *Simulation and real measurement of the roll rotation of the gimbal in negative direction*

Both figures indicate that no large discrepancies are present between the model and measurements. The closed-loop measurements and simulations can now be compared.

7.2 Closed-loop comparison

The closed-loop measurements and simulations are compared using the same PID controller. The values of the PID controller are determined in Chapter 6 by using the Ziegler-Nichols method and are respectively 0.2647, 1.7647 and 0.0099. Note that only a feedback is present for the position. In Figure 7.3 the simulation and measurement data are shown. As seen with the open-loop simulation, for small angles the simulation and measurement data are approximately the same. For larger angles the simulation and measurement data differ more.

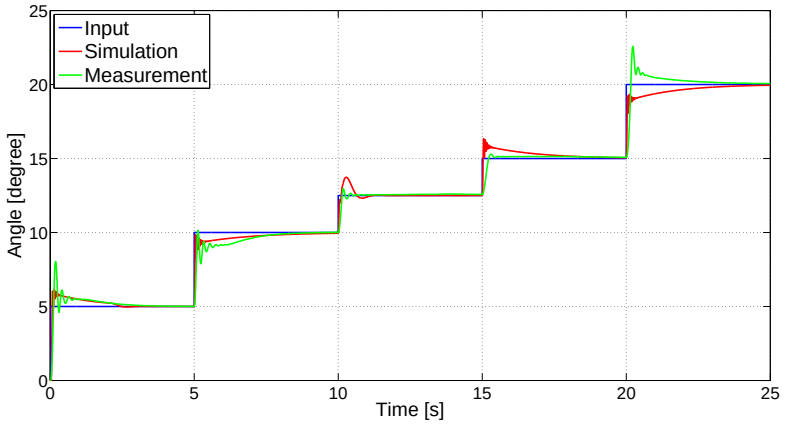


Figure 7.3: *Simulation and real measurement of the roll rotation of the gimbal using steps*

It would be expected that the gimbal is performing reasonably when looking at the results. Although the results look promising, some unexpected stuttering of the roll motor is seen when moving the top of the gimbal around. This behaviour is only seen for the roll angle and only when the platform has to keep a certain attitude in comparison to the surrounding world. The code on the Arduino is checked, but no errors can be found. All other hardware is also checked for any faults, but non are found. Also the motors were exchanged, also without any success of solving the problem. Eventually the time frame was too short to solve the stuttering of the roll motor. In the next chapter a conclusion is drawn and recommendations are given.

Chapter 8

Conclusion and recommendations

In this report a gimbal is made from scratch. First the material choices are explained in Chapter 2 after which the general theory of a brushless DC motor is introduced in Chapter 3. The gimbal and actuators are then modelled in Chapter 4. This model is inserted in `Simulink` in Chapter 5 and experiments are conducted with the designed gimbal in Chapter 6. Eventually the simulations are compared to the real measurement data in Chapter 7.

The goal of this report is to design, model and control a gimbal to gain full control over the abilities of a gimbal. The design is made without any noteworthy issues. Also the model of the gimbal is made. During the implementation of the model in Chapter 5, it became clear that more data is needed of the used actuators. This data has to be obtained by either measurements or by contacting the manufacturer for further specifications of the actuator. Control of the gimbal is achieved by implementing a feedback loop for the position. This resulted in some unexpected instability of the system, since the simulation data did not show instability. It was chosen to implement a PID controller, which is common in the industry, and this controller was tuned using the Ziegler-Nichols tuning method. Eventually the gimbal followed a trajectory rather well and the measurement data was comparable with the simulation data. Only unexplainable stuttering of the roll angle was unsolved. The goal of this report is achieved, although improvement of the gimbal is certainly needed. Full control of the gimbal's abilities is possible by programming it on the Arduino and full understanding of the gimbals behaviour can thus be achieved.

8.1 Recommendations

Since the gimbal is not yet achieving stable visualization of a camera image some recommendations are made. Besides using some more advanced control strategies, one can consider a linear quadratic controller. Some further suggestions are given here.

The sample rate of the control system is approximately 100 Hz. This can be considered rather slow, so a higher sample rate can give increased performance. The sample rate is limited because of the use of the Jeff Rowberg library to read out the MPU6050 data. Although this is a downside of the library, the benefits were considered more important. The main benefit of the library is that the drifting of the yaw angle is limited.

As said in Chapter 6 only the position control loop is used. To ensure better tracking of a reference signal it is recommended to implement at least an inner control loop for the velocity. For further improvement it is also advised to include the most inner control loop for the current. Both inner loops are normally controlled using a PI controller [5]. When implementing these loops it is advised to improve the model parameters first.

As earlier mentioned, the use of a second attitude measurement unit will give the ability to know the difference between intentional and unintentional movement. This will give the ability to use forward kinematics to control each axis. A second attitude measurement unit is normally already present on the UAV itself so only a connection between the controller and this device has to be made.

Bibliography

[1] Hobbyking. 3 axis brushless gimbal for multi rotor or aircraft. `http://www.hobbyking.com/hobbyking/store/__66150__FeiyuTech_G3_3_Axis_Brushless_Gimbal_for_Multi_Rotor_or_Aircraft.html`. [Online; accessed October 15, 2015].

[2] J. Rowberg. I2C device library. `http://www.i2cdevlib.com/`. [Online; accessed September 29, 2015. Librabry for I2C communication].

[3] N. van de Wouw. *Multibody Dynamics: lecture notes*. Eindhoven University of Technology, 2013. [Lecture notes course 4J400 Eindhoven University of Technology].

[4] R. N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and control*. Springer US, 2nd edition, 2010.

[5] L. Wang, S. Chai, D. Yoo, L. Gan, and K. Ng. *PID and predictive control of electrical drives and power converters using MATLAB/Simulink*. Wiley-IEEE Press, December 2014.

[6] P. Corke. *Robotics, Vision and Control - Fundamental Algorithms in MATLAB*, volume 73 of *Springer Tracts in Advanced Robotics*. Springer-Verlag Berlin Heidelberg, 2011.

[7] K. J. Seong, H. G. Kang, B. Y. Yeo, and H. P. Lee. The stabilization loop design for a two-axis gimbal system using LQG/LTR controller. In *SICE-ICASE, 2006. International Joint Conference*, pages 755–759, Oct 2006.

[8] J. Grant J. Bennet, A. Bhasin and W. C. Lim. PID tuning via classical methods. `https://controls.engin.umich.edu/wiki/index.php/PIDTuningClassica`, November 2007. [Online; accessed November 2, 2015].

Appendix A

Matlab function PWM converter

The code found below is used in the `Simulink` model of Chapter 5, which is shown in Figure 5.1. The subsystem of the PWM converter is shown in Figure 5.3. It calculates the correct PWM value and phase as an input for a three phase DC motor. The code is the same for the roll, pitch and yaw PWM converter.

```
% This function is used in the Simulink model of Chapter 5 to calculate the
% correct PWM value and phase as input for a three phase DC motor. More
% information can be found in Chapter 5 and Figure 5.1.3.

function [y,w,z,phase] = fcn(u)
 %Number of poles of DC motor
nop             =   14;
%Converting the input angle to radians
theta_error     =   (u*2*pi)/360;
%Converting the mechanical angle to electrical
theta_e         =   (theta_error * nop) / 2;
%Making sure that the value of the angle is between 0 and 2*pi
theta_e         =   mod(theta_e,(2*pi));
if(theta_e < 0)
    theta_e = theta_e + (2*pi);
end


%Implementing (3.0.2) till (3.0.7)
if theta_e <2/6*pi
PWM = (tan(theta_e)*5)/(sin(2/3*pi)-cos(2/3*pi)*tan(theta_e));
phase =1;
elseif theta_e < 4/6*pi
PWM = -(tan(theta_e)*cos(2/3*pi)*5-sin(2/3*pi)*5)/(tan(theta_e));
phase =2;
elseif theta_e < 6/6*pi
phase =3;
PWM = (tan(theta_e)*cos(2/3*pi)*5-sin(2/3*pi)*5)/(sin(4/3*pi) ...
    -cos(4/3*pi)*tan(theta_e));
elseif theta_e < 8/6*pi
phase =4;
PWM = (tan(theta_e)*cos(4/3*pi)*5-sin(4/3*pi)*5)/(sin(2/3*pi) ...
    -cos(2/3*pi)*tan(theta_e));
elseif theta_e < 10/6*pi
phase =5;
PWM = (sin(4/3*pi)*5 - tan(theta_e)*cos(4/3*pi)*5)/tan(theta_e);
else
phase =6;
PWM = (tan(theta_e)*5)/(sin(4/3*pi)-cos(4/3*pi)*tan(theta_e));
end


%Implementing Table 3.0.1 to assign the proper PWM values
```

```
44 if (phase == 1)
45     y =5;
46     w =PWM;
47     z =0;
48 elseif (phase == 2)
49     y =PWM;
50     w =5;
51     z =0;
52 elseif (phase == 3)
53     y =0;
54     w =5;
55     z =PWM;
56 elseif (phase == 4)
57     y =0;
58     w =PWM;
59     z =5;
60 elseif (phase == 5)
61     y =PWM;
62     w =0;
63     z =5;
64 else
65     y =5;
66     w =0;
67     z =PWM;
68 end
```

Appendix B

Matlab function IMU

The code found below is used in the `Simulink` model of Chapter 5, which is shown in Figure 5.1. The code is part of the subsystem of the IMU, shown in Figure 5.6, which is part of the subsystem of the gimbal, shown in Figure 5.5. It calculates the correct roll, pitch and yaw angles relatively to the fixed world frame.

```
% In this function the measurement of the IMU is implemented. This function
% is used in the Simulink model explained in Chapter 5 and this function is
% shown in Figure 5.1.6.
function [IMU_1,IMU_2,IMU_3]  = fcn(outside_roll,outside_pitch ...
    ,outside_yaw,gimbal_roll,gimbal_pitch,gimbal_yaw,gimbal_roll_dot ...
    ,gimbal_pitch_dot,gimbal_yaw_dot)
%Assiging the angles made by the gimbal to theta_1,2 and 3
theta_1 = gimbal_yaw;
theta_2 = gimbal_pitch;
theta_3 = gimbal_roll;

 %Determining the jacobian of the joints as in (4.3.7)
 J_0        =   [1                  0                          0;
                 0               cos(theta_1)               -sin(theta_1);
                 -sin(theta_2)   cos(theta_2)*sin(theta_1)   ...
                 cos(theta_1)*cos(theta_2)];
%Make the vector of frame zero angles
Phi_0       =   [outside_yaw; outside_pitch; outside_roll];

%Angular velocities acting on frame zere as if the joints produce them
%while the zero fram is fixed as in (4.3.6)
Omega_phi   =   J_0*Phi_0;

%Making the vector of the angular velocities of the gimbal
Omega       =   [gimbal_yaw_dot ;gimbal_pitch_dot ;gimbal_roll_dot];

%Determine the Jacobian as in (4.3.5)
J           =   [cos(theta_2)*cos(theta_3)  sin(theta_3)    0;
                 cos(theta_2)*sin(theta_3)  cos(theta_3)    0;
                 -sin(theta_2)              0               1];

%Determine the angles made comparing to the world frame, as in (4.3.8)
A = J*(Omega_phi + Omega);

%Output
IMU_1 = A(1);   %Yaw
IMU_2 = A(2);   %Pitch
IMU_3 = A(3);   %Roll
```

Appendix C

Matlab function gimbal dynamics

The code found below is used in the `Simulink` model of Chapter 5, which is shown in Figure 5.1. The code is part of the subsystem of the gimbal dynamics, shown in Figure 5.7, which is part of the subsystem of the gimbal, shown in Figure 5.5. Following the equations in Chapter 4 the angles and velocities of the gimbal.

```
% Following Chapter 4 to implement the dynamics of the gimbal in this
% function which is used in the Simulink model. The function is used in
% Figure 5.1.7. More information can be found in Chapter 5
function x_dot  = fcn(roll,pitch,yaw,roll_dot,pitch_dot,yaw_dot,roll_torque, ...
pitch_torque,yaw_torque,IMU_1,IMU_2,IMU_3)
%Assign roll, pitch and yaw angles made by the gimbal
theta_1 = yaw;
theta_2 = pitch;
theta_3 = roll;

%Assign roll, pitch and yaw angular velocities made by the gimbal
om_1 = yaw_dot;
om_2 = pitch_dot;
om_3 = roll_dot;

%Assign the measured angles of the gimbal refered to the world frame
alpha_1 = IMU_1;
alpha_2 = IMU_2;
alpha_3 = IMU_3;

%% Fill in measured parameters from the gimbal
joints = 3;      %amount of joints

%Measurements of the lengths, as in figures 4.1.1 and 4.1.2
l_1 =72/1000;    %measured in mm so divided by 1000
h_1 =90/1000;
b_1 =67/1000;
h_3 =8/1000;

%Masses measured in NX
m = [0.025 0.027 0.174];

%Gravitiy constant
g=9.81;

%Distances from x_0 to x_1 and x_1 to x_2 and so on from figures 4.1.1 and 4.1.2
x_1 =0;
x_2 =-28/1000;
x_3 =2/1000;

y_1 =27/1000;
y_2 =-29/1000;
y_3 =-2/1000;
```

```
44
45 z_1 =38/1000;
46 z_2 =0;
47 z_3 =-18/1000;
48
49 %Pseudo inertia matrices as in (4.2.8)
50 I11=[(-85/1000000+51/1000000+38/1000000)/2      0                    ...
51     0                                    m(1)*x_1;
52      0                                (85/1000000-51/1000000+38/1000000)/2 ...
53      -40/1000000                              m(1)*y_1;
54      0                                           -40/1000000    ...
55      (85/1000000+51/1000000-38/1000000)/2     m(1)*z_1;
56      m(1)*x_1                                    m(1)*y_1       ...
57      m(1)*z_1                              m(1)];
58 I22=[(-34/1000000+41/1000000+72/1000000)/2 33/1000000 0 m(2)*x_2;
59      33/1000000 (34/1000000-41/1000000+72/1000000)/2 0 m(2)*y_2;
60      0 0 (34/1000000+41/1000000-72/1000000)/2 m(2)*z_2;
61      m(2)*x_2 m(2)*y_2 m(2)*z_2 m(2)];
62 I33=[(-96/1000000+149/1000000+73/1000000)/2 -1/1000000 -2/1000000 m(3)*x_3;
63      -1/1000000 (96/1000000-149/1000000+73/1000000)/2 7/1000000 m(3)*y_3;
64      -2/1000000 7/1000000 (96/1000000+149/1000000-73/1000000)/2 m(3)*z_3;
65      m(3)*x_3 m(3)*y_3 m(3)*z_3 m(3)];
66
67 %Combining in a matrix
68 I = cell(1,3);
69 I{1} = I11;
70 I{2} = I22;
71 I{3} = I33;
72
73 %Combining in a matrix
74 rr1 = [x_1; y_1; z_1; 1];
75 rr2 = [x_2; y_2; z_2; 1];
76 rr3 = [x_3; y_3; z_3; 1];
77 rr = cell(1,3);
78 rr{1} = rr1;
79 rr{2} = rr2;
80 rr{3} = rr3;
81
82
83
84 %% initialize parameters/ making matrices
85 theta = [theta_1 theta_2 theta_3];
86 om = [om_1 om_2 om_3];
87 H = cell(joints,1);
88 T= cell(1,joints);
89 G = sym(zeros(joints,1));
90
91 %Rotation matrices as in (4.1.1)
92 R01 = [cos(theta_1) -sin(theta_1) 0; sin(theta_1) cos(theta_1) 0; 0 0 1];
93 R12 = [cos(theta_2) 0 sin(theta_2); 0 1 0; -sin(theta_2) 0 cos(theta_2)];
94 R23 = [ 1 0 0; 0 cos(theta_3) -sin(theta_3); 0 sin(theta_3) cos(theta_3)];
95 R03 = R01*R12*R23;
96
97 %Translational vectors as in (4.1.2)
98 d01 =[l_1*sin(theta_1); -l_1*cos(theta_1); h_1];
99 d12 = [ b_1*cos(theta_2); l_1; b_1*sin(theta_2)];
100 d23 = [-b_1; h_3*sin(theta_3); h_3*cos(theta_3)];
101
102 %Homogenous transformation matrices as in (4.1.3)
103 T01 = [R01 d01; 0 0 0 1];
104 T12 = [R12 d12; 0 0 0 1];
105 T23 = [R23 d23; 0 0 0 1];
106 T02 = T01*T12;
107 T03 = T01*T12*T23;
108 T{1} =T01;
109 T{2} = T02;
110 T{3} = T03;
111
112 %Correct gravity with the angles
113 gg = R03*[-sin(alpha_2); cos(alpha_2)*sin(alpha_3);cos(alpha_2)*cos(alpha_2)] ...
```

```
        * g;
gg =[gg;0];


%Using (4.2.6) the velocity coupling matrix is determined as:
for i=1:joints
for j=1:joints
    for k=1:joints

        r=max([i,j,k]);
        H{i}(j,k) = trace(diff(diff(T{r},theta(j)),theta(k)) * ...
            I{r}*transpose(diff(T{r},theta(i))))*om(j)*om(k);
    end
end
end


for i=1:joints
H{i} = sum(sum(H{i}));
H{i} = simplify(H{i});
end

%Using (4.2.5) the intertial-type matrix is determined:
D = sym(zeros(joints,joints));

for i=1:joints
    for j=1:joints
       for r=max([i,j]):joints;
        Z = trace(diff(T{r},theta(j))*I{r}*transpose(diff(T{r},theta(i))));

        D(i,j) =D(i,j)+ Z;
       end
    end
end

D=simplify(D);

%Using (4.2.7) the gravitational vector is determined as:
for i = 1:joints
    for r = i: joints
   G(i) =G(i) - m(r)*transpose(gg)*diff(T{r},theta(i))*rr{r};
    end
end

H=[H{1};H{2}; H{3}];



%Assign torque from the DC motor
T1 = yaw_torque;
T2 = pitch_torque;
T3 = roll_torque;

%Assign small values for the dynamic and static friction
t_s = 2*10^(-17)*[1; 1; 1];
t_d = 0.001*[1; 1; 1];

%Make a vector of the angular velocities
Omega = [om_1; om_2; om_3];

%Determine the torques Q by (4.4.1)
Q = [T1-t_s(1)*sign(Omega(1))-t_d(1)*(Omega(1));
    T2-t_s(2)*sign(Omega(2))-t_d(2)*(Omega(2));
    T3-t_s(3)*sign(Omega(3))-t_d(3)*(Omega(3))];

%Determine x_dot following (4.2.3) where x_dot is a vector of the three
%angular accelerations and three velocites of the gimbal
x_dot = [inv(D)*(Q-H-G); Omega];
```

Appendix D

Arduino code for open- and closed-loop operation

The code found below is used on the Arduino to control the attitude of the gimbal. The code is used in Chapter 6 and works with all materials and the design described in Chapter 2. The PWM converter introduced in Chapter 3 is implemented in the code. Both open- and closed-loop operation is performed using this code.

```
// ------------------------------------------------------------------------
// This script can be used to control a three axis gimbal which is made of
// the materials described in the report above.
// ------------------------------------------------------------------------


int phase = 1;    // initialize the phase of a DC motor, always starts in ...
    phase 1
int i = 0;        // initialize counter
const double Pi = 3.14159;  // initialize pi

// 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 file for library to work
#include <MPU6050_6Axis_MotionApps20.h>  // include file for library to work
#include <Adafruit_PWMServoDriver.h>  // include file for library to work
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x40);   // Set adress ...
    for the PWM servo driver

MPU6050 mpu;  // Define MPU6050 as mpu

// 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

#include <math.h>     // include file for library to work
#include "get_angle.h"    // include file to calculate the angle
#include "motor_driving.h"  // include file so the motor can be driven

// initializing yaw components
double theta_last_y=0;   // set last measured yaw on zero
double theta_all_y=0;   // set total yaw on zero
double theta_error_y=0;   // set the yaw error on zero
double theta_m_y=0;     // set mechanical yaw angle on zero
double error_y=0;      // set the error in yaw on zero
double nop = 14;  //Number Of Poles nop
float theta_r_y = 0;
float yaw_offset_1 = 0;   // set yaw offset to zero
```

```
39 float yaw_offset_2 = 0;   // set yaw offset to zero
40 float roll_offset_1 = 0;  // set roll offset to zero
41 float pitch_offset_1 = 0; // set pitch offset to zero
42 double P_c_y=0;           // set P of yaw controller on zero
43 double I_c_y=0;           // set I of yaw controller on zero
44 double D_c_y=0;           // set D of yaw controller on zero
45 float time_2y;            // initialize time of yaw on
46 float time_3y;            // initialize time of yaw on
47 double theta_before_windup_y;   // initialize yaw before windup
48 double theta_after_windup_y;    // initialize yaw after windup
49 double windup_y;                // initialize windup of yaw
50 double real_error_y;            // initialize real yaw error
51 float sampling_y;               // initialize sampling of yaw
52
53 // initializing roll components
54 double theta_last_r=0;      // set last roll to zero
55 double theta_all_r=0;       // set total roll to zero
56 float theta_error_r=0;      // set roll error to zero
57 double theta_m_r=0;         // set mechanical roll to zero
58 double error_r=0;           // set roll error to zero
59 float theta_r_r = 0;
60 double P_c_r=0;             // set P of roll controller to zero
61 double I_c_r=0;             // set I of roll controller to zero
62 double D_c_r=0;             // set D of roll controller to zero
63 float time_2r;              // initialize roll time
64 float time_3r;              // initialize roll time
65 double theta_before_windup_r;   // initialize roll before windup
66 double theta_after_windup_r;    // initialize roll after windup
67 double windup_r;                // initialize windup of roll
68 double real_error_r;            // initialize real roll error
69 float sampling_r;               // initialize sampling of roll
70
71 // initializing pitch components
72 double theta_last_p=0;        // set last pitch to zero
73 double theta_all_p=0;         // set total pitch to zero
74 double theta_error_p=0;       // set pitch error to zero
75 double theta_m_p=0;           // set mechanical pitch to zero
76 double error_p=0;             // set pitch error to zero
77 float theta_r_p = 0;
78 double P_c_p=0;               // set P of pitch controller to zero
79 double I_c_p=0;               // set I of pitch controller to zero
80 double D_c_p=0;               // set D of pitch controller to zero
81 float time_2p;                // initialize pitch time
82 float time_3p;                // initialize pitch time
83 double theta_before_windup_p;   //initialize pitch before windup
84 double theta_after_windup_p;    // initialize pitch after windup
85 double windup_p;                // initialize windup of pitch
86 double real_error_p;            // initialize real pitch error
87 float sampling_p;               // initialize sampling of pitch
88
89 double P_r_c;
90
91
92 double output;      // initialize eventual output
93
94
95 // Setup of the script, set all correct values and initialize the MPU6050 and ...
       set DC motors to zero roll pitch and yaw
96 void setup() {
97   Serial.begin(115200);  // Serial port where data can be read
98   pwm.begin();           // initialize PWM shield
99   pwm.setPWMFreq(1600);  // 1600 This is the maximum PWM frequency
100  sampling_y = 0.01;     // set sampling time to 0.01
101
102  uint8_t twbrbackup = TWBR;     // save I2C bitrate
103  // must be changed after calling Wire.begin() (inside pwm.begin())
104  TWBR = 12; // upgrade to 400KHz!
105
106  //Motor 1 (yaw)  PWM values are chosen such that roll, pitch and yaw are ...
         approximately zero
```

```
107      pwm.setPWM(13, 0, 4095 );
108      pwm.setPWM(14, 0, 4095);
109      pwm.setPWM(15, 0, 0 );
110      Serial.println("First motor on");
111      //Motor 2 pitch
112      pwm.setPWM(9, 0, 4095 );
113      pwm.setPWM(10, 0, 4095);
114      pwm.setPWM(11, 0, 0 );
115      Serial.println("Second motor on");
116      //motor 3 roll
117      pwm.setPWM(6, 0, 4095 );
118      pwm.setPWM(7, 0, 4095);
119      pwm.setPWM(8, 0, 0 );
120      Serial.println("Third motor on");
121      delay(4000);  // small delay so vibrations are gone
122
123
124
125
126    setup_angles();  // initialize the MPU6050
127    for (int an = 0; an < 1800; an++)    // read out angles so the offset can be ...
           corrected
128    {
129      get_angles1();
130    }
131    // Set the offsets
132    yaw_offset_1 = yaw_measured_1 ;
133    pitch_offset_1 = pitch_measured_1;
134    roll_offset_1 = roll_measured_1;
135    Serial.println("Setup done");
136  }
137
138
139
140  void loop() {
141    get_angles1();  // read out angles to make sure no NAN's are returned
142
143    i = i + 1;     // counter plus one
144
145
146  // different angle steps can be implemented here. In this case after 500 ...
       counts the roll will step to 5 degrees
147    if (i < 500)
148    {
149      theta_r_y = 0;
150      theta_r_p = 0;
151      theta_r_r = 0;
152    }
153    else if (i < 1000)
154    {
155      theta_r_y = 0;
156      theta_r_p = 0;
157      theta_r_r = 5;
158    }
159
160
161  //compute roll pitch yaw error, if error is larger than 360 degrees, subtract ...
       360, is the same point on a circle
162    theta_error_y = theta_r_y - yaw_measured_1+yaw_offset_1;
163    real_error_y = theta_error_y;
164  if (theta_error_y >360)
165  {
166    theta_error_y = theta_error_y - 360;
167  }
168
169    theta_error_p = theta_r_p - pitch_measured_1+pitch_offset_1;
170    real_error_p = theta_error_p;
171  if (theta_error_p >360)
172  {
173    theta_error_p = theta_error_p - 360;
```

```
}

  theta_error_r = theta_r_r + roll_measured_1 - roll_offset_1;
  real_error_r = theta_error_r;
if (theta_error_r >360)
{
  theta_error_r = theta_error_r - 360;
}

    // compute the gains, P
  P_c_y =0.35* theta_error_y;
  P_c_p =0.29* theta_error_p;
  P_c_r =0.075* theta_error_r;


  time_3y = time_2y;  // set previous measured time to time_3y
  time_2y = micros();  // set measured time to time_2y
  sampling_y = (time_2y - time_3y)/1000000 ;   // determine sampling time

// first time the sampling time will be large so if statement
 if(sampling_y < 1)
 { // Determine I action of controllers
  I_c_y = I_c_y + 15*theta_error_y * sampling_y;
  I_c_p = I_c_p + 18*theta_error_p * sampling_y;
  I_c_r = I_c_r + 4*theta_error_r * sampling_y;
 }
 // Determine D actions of controllers
  D_c_y =0.12* (yaw_measured_1 - theta_last_y)/sampling_y;
  D_c_p =0.09* (pitch_measured_1 - theta_last_p)/sampling_y;
  D_c_r = 0.0015* (roll_measured_1 - theta_last_r)/sampling_y;

  // Store last measured angles
  theta_last_y = yaw_measured_1;
  theta_last_p = pitch_measured_1;
  theta_last_r = roll_measured_1;


//Set the output signal of the controllers, 8.57 is added due to ...
    initialization of the motors
  theta_error_y = -P_c_y + 8.57 - I_c_y  + D_c_y ;
  theta_error_p = 8.57 - P_c_p - I_c_p  + D_c_p  ;
  theta_error_r = 8.57 - P_c_r  - I_c_r + D_c_r;

// Determine output
output = theta_error_r;

//Convert control signals to radians and to electrical angles
  theta_error_y= (theta_error_y * 2 * Pi) / 360;
  theta_e_y = (theta_error_y * nop) / 2;

  theta_error_p= (theta_error_p * 2 * Pi) / 360;
  theta_e_p = (theta_error_p * nop) / 2;

  theta_error_r= (theta_error_r * 2 * Pi) / 360;
  theta_e_r = (theta_error_r * nop) / 2;

// Make sure electrical angles are smaller than 2 pi and larger than 0
  theta_e_y = fmod(theta_e_y, (2 * Pi));
  if (theta_e_y < 0)
  {
    theta_e_y = theta_e_y + (2 * Pi) ;
  }

  theta_e_p = fmod(theta_e_p, (2 * Pi));
  if (theta_e_p < 0)
  {
    theta_e_p = theta_e_p + (2 * Pi) ;
  }

  theta_e_r = fmod(theta_e_r, (2 * Pi));
```

```
243   if (theta_e_r < 0)
244   {
245     theta_e_r = theta_e_r + (2 * Pi) ;
246   }
247
248 // Eventual drive motors
249   PWM_drive_motor(theta_e_y);
250   PWM_drive_motor(theta_e_p);
251   PWM_drive_motor(theta_e_r);
252
253 }
```

```
1  /* ============================================
2  I2Cdev device library code is placed under the MIT license
3  Copyright (c) 2012 Jeff Rowberg
4
5  Permission is hereby granted, free of charge, to any person obtaining a copy
6  of this software and associated documentation files (the "Software"), to deal
7  in the Software without restriction, including without limitation the rights
8  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9  copies of the Software, and to permit persons to whom the Software is
10 furnished to do so, subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in
13 all copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
21 THE SOFTWARE.
22 ===============================================
23 */
24
25
26
27
28 #define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
29 bool blinkState = false;
30
31 // MPU control/status vars
32 bool dmpReady = false;  // set true if DMP init was successful
33 uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
34 uint8_t devStatus;      // return status after each device operation (0 = ...
       success, !0 = error)
35 uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
36 uint16_t fifoCount;     // count of all bytes currently in FIFO
37 uint8_t fifoBuffer[64]; // FIFO storage buffer
38
39 // orientation/motion vars
40 Quaternion q;           // [w, x, y, z]         quaternion container
41 VectorInt16 aa;         // [x, y, z]            accel sensor measurements
42 VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor ...
       measurements
43 VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor ...
       measurements
44 VectorFloat gravity;    // [x, y, z]            gravity vector
45 float euler[3];         // [psi, theta, phi]    Euler angle container
46 float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and ...
       gravity vector
47
48 //// MPU control/status vars
49 //bool dmpReadyC2 = false;  // set true if DMP init was successful
50 //uint8_t accelgyroIC2IntStatus;   // holds actual interrupt status byte from MPU
51 //uint8_t devStatusC2;      // return status after each device operation (0 = ...
       success, !0 = error)
```

```
52 //uint16_t packetSizeC2;    // expected DMP packet size (default is 42 bytes)
53 //uint16_t fifoCountC2;     // count of all bytes currently in FIFO
54 //uint8_t fifoBufferC2[64]; // FIFO storage buffer
55 //
56 //// orientation/motion vars
57 //Quaternion qC2;           // [w, x, y, z]         quaternion container
58 //VectorFloat gravityC2;    // [x, y, z]            gravity vector
59 //float yprC2[3];           // [yaw, pitch, roll]   yaw/pitch/roll container ...
       and gravity vector
60
61
62 float yaw_measured_1;
63 float pitch_measured_1;
64 float roll_measured_1;
65 //float yaw_measured_2;
66
67
68 // ================================================================
69 // ===               INTERRUPT DETECTION ROUTINE                ===
70 // ================================================================
71
72 volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin ...
       has gone high
73 void dmpDataReady() {
74     mpuInterrupt = true;
75 }
76
77
78
79
80 // ================================================================
81 // ===                      INITIAL SETUP                       ===
82 // ================================================================
83
84 void setup_angles() {
85     // join I2C bus (I2Cdev library doesn't do this automatically)
86     #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
87         Wire.begin();
88         TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
89     #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
90         Fastwire::setup(400, true);
91     #endif
92
93     // initialize serial communication
94     // (115200 chosen because it is required for Teapot Demo output, but it's
95     // really up to you depending on your project)
96     Serial.begin(115200);
97     while (!Serial); // wait for Leonardo enumeration, others continue immediately
98
99     // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or Ardunio
100    // Pro Mini running at 3.3v, cannot handle this baud rate reliably due to
101    // the baud timing being too misaligned with processor ticks. You must use
102    // 38400 or slower in these cases, or use some kind of external separate
103    // crystal solution for the UART timer.
104
105    // initialize device
106    Serial.println(F("Initializing I2C devices..."));
107    mpu.initialize();
108
109    // verify connection
110    Serial.println(F("Testing device connections..."));
111    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : ...
           F("MPU6050 connection failed"));
112
113
114    // load and configure the DMP
115    Serial.println(F("Initializing DMP..."));
116    devStatus = mpu.dmpInitialize();
117
118
```

```
119
120 // supply your own gyro offsets here, scaled for min sensitivity
121     mpu.setXGyroOffset(122);
122     mpu.setYGyroOffset(36);
123     mpu.setZGyroOffset(59);
124     mpu.setXAccelOffset(-699);
125     mpu.setYAccelOffset(-3413);
126     mpu.setZAccelOffset(1354); // 1688 factory default for my test chip
127
128
129
130     // make sure it worked (returns 0 if so)
131     if (devStatus == 0) {
132         // turn on the DMP, now that it's ready
133         Serial.println(F("Enabling DMP..."));
134         mpu.setDMPEnabled(true);
135
136         // enable Arduino interrupt detection
137         Serial.println(F("Enabling interrupt detection (Arduino external ...
                interrupt 0)..."));
138         attachInterrupt(0, dmpDataReady, RISING);
139         mpuIntStatus = mpu.getIntStatus();
140
141         // set our DMP Ready flag so the main loop() function knows it's okay ...
                to use it
142         Serial.println(F("DMP ready! Waiting for first interrupt..."));
143         dmpReady = true;
144
145         // get expected DMP packet size for later comparison
146         packetSize = mpu.dmpGetFIFOPacketSize();
147     } else {
148         // ERROR!
149         // 1 = initial memory load failed
150         // 2 = DMP configuration updates failed
151         // (if it's going to break, usually the code will be 1)
152         Serial.print(F("DMP Initialization failed (code "));
153         Serial.print(devStatus);
154         Serial.println(F(")"));
155     }
156
157     // configure LED for output
158     pinMode(LED_PIN, OUTPUT);
159 }
160
161
162
163
164 // ================================================================
165 // ===                      MAIN PROGRAM LOOP                   ===
166 // ================================================================
167
168 void get_angles1() {
169     // if programming failed, don't try to do anything
170     if (!dmpReady) return;
171
172     // wait for MPU interrupt or extra packet(s) available
173     while (!mpuInterrupt && fifoCount < packetSize) {
174
175     }
176
177     // reset interrupt flag and get INT_STATUS byte
178     mpuInterrupt = false;
179     mpuIntStatus = mpu.getIntStatus();
180
181     // get current FIFO count
182     fifoCount = mpu.getFIFOCount();
183
184     // check for overflow (this should never happen unless our code is too ...
            inefficient)
185     if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
```

```
186         // reset so we can continue cleanly
187         mpu.resetFIFO();
188         Serial.println(F("FIFO overflow!"));
189 
190     // otherwise, check for DMP data ready interrupt (this should happen ...
        frequently)
191     } else if (mpuIntStatus & 0x02) {
192         // wait for correct available data length, should be a VERY short wait
193         while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
194 
195         // read a packet from FIFO
196         mpu.getFIFOBytes(fifoBuffer, packetSize);
197 
198         // track FIFO count here in case there is > 1 packet available
199         // (this lets us immediately read more without waiting for an interrupt)
200         fifoCount -= packetSize;
201 
202             // display Euler angles in degrees
203             mpu.dmpGetQuaternion(&q, fifoBuffer);
204             mpu.dmpGetGravity(&gravity, &q);
205             mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
206 
207 
208  yaw_measured_1 =  ypr[0] * 180/M_PI;
209  pitch_measured_1 = ypr[1] *180/M_PI;
210  roll_measured_1 = ypr[2] * 180/M_PI;
211 
212 
213 
214 
215         // blink LED to indicate activity
216         blinkState = !blinkState;
217         digitalWrite(LED_PIN, blinkState);
218     }
219 }
```

```
 1 // This script is used to drive a three phase DC motor
 2 
 3 // determine sine and cosine values
 4 double s23= sin((2*Pi)/3);
 5 double c23= cos((2*Pi)/3);
 6 double s43= sin((4*Pi)/3);
 7 double c43= cos((4*Pi)/3);
 8 
 9 // set error roll pitch yaw to zero
10 double theta_e_p=0;
11 double theta_e_r=0;
12 double theta_e_y=0;
13 
14 // set maximum pwm
15 float Max_pwm = 4095;
16 
17 //initialize pwm
18 int PWM;
19 // function that drives the motor
20 void PWM_drive_motor(double theta_e){
21 
22  // Using equations 3.02 till 3.07 of the report to determine PWM value
23  if (theta_e == theta_e_y || theta_e == theta_e_p || theta_e == theta_e_r)
24   {
25     if (theta_e < (2 * Pi) / 6)
26   {
27     PWM = (tan(theta_e) * 4095) / (s23 - (c23 * tan(theta_e)));
28     phase = 1;
29   }
30   else if (theta_e < (4 * Pi) / 6)
31   {
32     PWM = -(tan(theta_e) * c23 * 4095 - s23 * 4095) / (tan(theta_e));
```

```
33      phase = 6;
34    }
35    else if (theta_e < (6 * Pi) / 6)
36    {
37      PWM = (tan(theta_e) * c23 * 4095 - s23 * 4095) / (s43 - c43 * tan(theta_e));
38      phase = 5;
39    }
40    else if ( theta_e < (8 * Pi) / 6)
41    {
42      PWM = (tan(theta_e) * c43 * 4095 - s43 * 4095) / (s23 - c23 * tan(theta_e));
43      phase = 4;
44    }
45    else if ( theta_e < (10 * Pi) / 6)
46    {
47      PWM = (s43 * 4095 - tan(theta_e) * c43 * 4095) / tan(theta_e);
48      phase = 3;
49    }
50    else
51    {
52      PWM = (tan(theta_e) * 4095) / (s43 - c43 * tan(theta_e));
53      phase = 2;
54    }
55    }
56
57
58  // knowing pwm and phase results in values to steer a DC motr, roll pitc and yaw
59    if(theta_e == theta_e_r)
60    {
61    if (phase == 1)
62    {
63      pwm.setPWM(13, 0, 4095 );
64      pwm.setPWM(14, 0, PWM );
65      pwm.setPWM(15, 0, 0 );
66    }
67    else if (phase == 6)
68    {
69      pwm.setPWM(13, 0, PWM );
70      pwm.setPWM(14, 0, 4095 );
71      pwm.setPWM(15, 0, 0 );
72    }
73    else if (phase == 5)
74    {
75      pwm.setPWM(13, 0, 0 );
76      pwm.setPWM(14, 0, 4095 );
77      pwm.setPWM(15, 0, PWM );
78    }
79    else if (phase == 4)
80    {
81      pwm.setPWM(13, 0, 0 );
82      pwm.setPWM(14, 0, PWM );
83      pwm.setPWM(15, 0, 4095 );
84    }
85    else if (phase == 3)
86    {
87      pwm.setPWM(13, 0, PWM );
88      pwm.setPWM(14, 0, 0 );
89      pwm.setPWM(15, 0, 4095 );
90    }
91    else if (phase == 2)
92    {
93     pwm.setPWM(13, 0, 4095 );
94      pwm.setPWM(14, 0, 0 );
95      pwm.setPWM(15, 0, PWM );
96    }
97    }
98
99
100
101
102
```

```
103 else if(theta_e == theta_e_p)
104   {
105   if (phase == 1)
106   {
107     pwm.setPWM(9, 0, 4095 );
108     pwm.setPWM(10, 0, PWM );
109     pwm.setPWM(11, 0, 0 );
110   }
111   else if (phase == 6)
112   {
113     pwm.setPWM(9, 0, PWM );
114     pwm.setPWM(10, 0, 4095 );
115     pwm.setPWM(11, 0, 0 );
116   }
117   else if (phase == 5)
118   {
119     pwm.setPWM(9, 0, 0 );
120     pwm.setPWM(10, 0, 4095 );
121     pwm.setPWM(11, 0, PWM );
122   }
123   else if (phase == 4)
124   {
125     pwm.setPWM(9, 0, 0 );
126     pwm.setPWM(10, 0, PWM );
127     pwm.setPWM(11, 0, 4095 );
128   }
129   else if (phase == 3)
130   {
131     pwm.setPWM(9, 0, PWM );
132     pwm.setPWM(10, 0, 0 );
133     pwm.setPWM(11, 0, 4095 );
134   }
135   else if (phase == 2)
136   {
137    pwm.setPWM(9, 0, 4095 );
138     pwm.setPWM(10, 0, 0 );
139     pwm.setPWM(11, 0, PWM );
140   }
141   }
142
143
144
145 else if(theta_e == theta_e_y)
146   {
147     PWM = PWM+tresshold;
148     if(PWM  >4095)
149     {
150       PWM = 4095;
151     }
152   if (phase == 1)
153   {
154
155     pwm.setPWM(6, 0, Max_pwm+tresshold );
156     pwm.setPWM(7, 0, PWM );
157     pwm.setPWM(8, 0, tresshold );
158     }
159   else if (phase == 6)
160   {
161
162     pwm.setPWM(6, 0, PWM );
163     pwm.setPWM(7, 0, Max_pwm+tresshold );
164     pwm.setPWM(8, 0, tresshold );
165
166   }
167   else if (phase == 5)
168   {
169     pwm.setPWM(6, 0, tresshold );
170     pwm.setPWM(7, 0, Max_pwm+tresshold );
171     pwm.setPWM(8, 0, PWM );
172
```

```
173    }
174    else if (phase == 4)
175    {
176      pwm.setPWM(6, 0, tresshold );
177      pwm.setPWM(7, 0, PWM );
178      pwm.setPWM(8, 0, Max_pwm +tresshold);
179        }
180
181
182    else if (phase == 3)
183    {
184      pwm.setPWM(6, 0, PWM );
185      pwm.setPWM(7, 0, tresshold );
186      pwm.setPWM(8, 0, Max_pwm +tresshold );
187      }
188
189    else if (phase == 2)
190    {
191     pwm.setPWM(6, 0, Max_pwm +tresshold );
192      pwm.setPWM(7, 0, tresshold );
193      pwm.setPWM(8, 0, PWM  );
194      }
195    }
196
197
198  }
```