DEGREE PROJECT IN MECHANICAL ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Riderless self-balancing bicycle

Derivation and implementation of a time variant linearized state space model for balancing a bicycle in motion by turning the front wheel

ARTHUR GRÖNLUND

CHRISTOS TOLIS

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INDUSTRIAL ENGINEERING AND MANAGEMENT

Riderless self-balancing bicycle

Derivation and implementation of a time variant linearized state space model for balancing a bicycle in motion by turning the front wheel

ARTHUR GRÖNLUND, CHRISTOS TOLIS

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

TRITA-ITM-EX 2018:58

Abstract

Self-driving vehicles are becoming more and more prevalent in society, with buses and cars close to being implemented in the public domain. Self-driving two-wheeled vehicles could be a solution for space-efficient transportation in cities, where space is becoming a larger issue.

The purpose of this project was to develop and implement a linearized time variant state space model for balancing such a two-wheeled vehicle in the form of a bicycle by turning its front wheel. To test the derived model a small demonstrator was built and experimented with.

The final conclusion was that the model could be a simple solution for balancing an electric bicycle. However, further experimentation on a bigger scale would have to be done to reach a more decisive conclusion.

Referat

Förarlös självbalanserande cykel

Självkörande fordon börjar bli en allt större verklighet i samhället, där bussar och bilar snart kan komma att implementeras på större skala. Självkörande tvåhjuliga fordon kan vara en möjlig lösning på mindre fordon i städer där utrymme blir mer och mer sparsamt.

Syftet med detta projekt har varit att ta fram och implementera en linjäriserad tidsvarierande tillståndsmodell för balansreglering av en elcykel genom vridning av framhjulet. För att testa modellen konstruerades en liten demonstrator med vilken experiment och tester utfördes.

Den slutsats som drogs var att modellen mycket väl skulle kunna vara en lösning för balansering av en elcykel, men att fortsatta undersökningar bör genomföras på en större skala för att en mer definitiv slutsats skall kunna dras.

Acknowledgements

We would like to thank the staff and students working at the M building at KTH for showing enthusiastic interest in our project and engaging in discussions during our test runs. The discussions led to further insights about the project and were very helpful.

Thank you also to our fellow FiM students and assistants for showing interest in the project and suggesting improvements, both during seminars and off-schedule.

A special thank you goes to Staffan Qvarnström for pointing us in the right direction whenever a problem arose. A thank you also to Stefan Ionescu for initially helping us out with the 3D-printing. Finally, a thank you to Kayla Kearns for proof-reading our report.

Contents

List of Figures

Abbreviations

CMG control moment gyroscope. 1

CS chip select. 3

DC direct current. 2, 4, 20

IDE integrated development environment. 3

IVP initial value problem. 15, 16

LED light emitting diode. 19

MEMS micro electromechanical system. 6, 7

MISO master in slave out. 3

MOSI master out slave in. 3

ODE ordinary differential equation. 4, 15

PI proportional integral. 14

PLA polylactic acid. 22

PWM pulse-width modulation. 3, 23, 25, 31

SCK serial clock. 3

SCL serial clock. 3

SDA serial data. 3

SPI serial peripheral interface. 3

SS slave select. 3

USB Universal Serial Bus. 3, 23

Nomenclature

$\mathbf{A}$	State matrix
$\mathbf{B}$	Input vector
$\mathbf{C}$	Output vector
$\mathbf{L}$	State space feedback vector
ρ	Air density
θ	Steer angle
φ	Bicycle lean angle
A_f	Frontal area of bicycle
C_d	Drag coefficient
C_R	Rolling resistance coefficient
F_B	Braking force
F_C	Centrifugal force of bicycle
F_D	Driving force
F_L	Force of air drag
F_R	Force of rolling resistance
F_{fB}	Force of friction between rear tyre and ground
F_{fF}	Force of friction between front tyre and ground
g	Acceleration due to earth's gravity
h_g	Length from ground to centre of mass
J_L	Moment of inertia of the bicycle, around ground contact point
J_S	Moment of inertia of steer assembly

$k_2\Phi$	DC motor constant
L	Length between rear and front axles
l_b	Length from rear axle to centre of mass
l_f	Length from front axle to centre of mass
m	Bicycle mass
N_B	Normal force between rear tyre and ground
N_F	Normal force between front tyre and ground
r_b	Radius of curvature, rear axle
r_f	Radius of curvature, front axle
r_g	Radius of curvature, centre of mass
R_{AS}	Electrical resistance of steering motor
v	Bicycle speed
x	Position of bicycle centre of mass in the x-direction
y	Position of bicycle centre of mass in the y-direction
z	Position of bicycle centre of mass in the z-direction

Chapter 1

Introduction

1.1 Background

With self driving vehicles such as cars and buses becoming more of a reality daily, the creation of smaller and consequently less energy consuming vehicles is becoming more and more relevant. Self driving motorcycles, and two-wheeled vehicles in general, could prove to be a more effective way of self driven transportation because of their advantage of taking up less space and energy for single person transportation. For these vehicles to be usable, a good and energy efficient balancing system needs to be implemented. Usually, one of three different stabilizing methods are used. The first method uses a gyroscope with motors attached to the gimbals and flywheel. This is called control moment gyroscope (CMG) [1]. The second method uses a displaceable weight to shift the bicycle's centre of gravity [2]. The third method utilizes the centrifugal force that arises when a bicycle is steered in a certain direction. This is referred to as "dynamic stabilization" [1]. An example of this method being used can be found in [3], where it was used to balance a full-sized electric bicycle, albeit with a fairly complicated system for regulation. Of these methods, the third method has been the subject of research in this project.

There are some obvious advantages to the CMG method and the displaceable weight method, as they can be active when the bicycle is standing still. The steering method only works when the bicycle has a forward speed. However, energy is required to keep a flywheel spinning at high speed or have the ability to move around a heavy mass. Turning a steering handle should be quite energy-efficient in comparison. Another problem using the steering method is that the bicycle will steer off its intended path.

1.2 Purpose

The purpose of this project has been to develop and evaluate whether a linearized time-variant state-space model can be utilized in balancing a bicycle using a dynamic stabilization technique.

1.3 General method

To research the above points of interest, a prototype bicycle using the chosen system for balancing was constructed. An Arduino Uno was utilized to do the calculations needed to automatically control the balance of the system. Two direct current (DC) motors, one for driving the bicycle forward and one for turning the front wheel, were used as well as an accelerometer and gyro for measuring the tilting angle.

Before the construction was carried out, a program for simulating the mechanics of the bicycle with and without automatic control was created. The purpose of the simulation was to tune the control variables without risking damage to hardware, and to determine the approximate requirements of parts used in construction of the prototype.

After constructing the demonstrator experiments were carried out to fine-tune the system and determine its feasibility.

Chapter 2

Theory

2.1 The Arduino Uno board

The Arduino Uno is an open-source board containing the ATmega328 microcontroller. This microcontroller can easily be programmed via the integrated Universal Serial Bus (USB) connection on the board and the Arduino Software integrated development environment (IDE). It has 13 digital pins which can be configured as input or output. Pins 3, 5, 6, 9, 10 and 11 can also be used to output a pulse-width modulation (PWM) signal. The board also has 6 analog input pins that are capable of measuring the connected input voltage with a 10-bit resolution [4].

The Uno also has the capability to communicate with other connected devices using the I^2C bus [4]. The I^2C bus uses two bus lines for communication, these are called serial data (SDA) and serial clock (SCL). It is possible to connect multiple devices along these lines by giving each device a unique address [5]. On revision 3 of the Uno, there are two separate pins for SDA and SCL. However, these are connected in parallel to the analog pins 4 and 5 respectively. This means that once I^2C is activated those pins cannot be used for other purposes [4].

One more way the Uno can communicate with other devices is via the serial peripheral interface (SPI) protocol. This interface uses 3 lines for communication called master out slave in (MOSI), master in slave out (MISO) and serial clock (SCK). One additional line, usually called slave select (SS) or chip select (CS), is also required for each connected device. On the Uno there are separate pins for the 3 communication lines but these are shared with digital pins 11, 12 and 13. Similar to the I^2C case, once SPI is activated these pins cannot be used for other purposes [6].

2.2 DC motors

DC motors are controlled by adjusting the voltage supplied to the motor. The relation between the speed of the motor and supplied voltage is given by the equation

$$U_A = k_2 \Phi \omega + R_A I_A, \tag{2.1}$$

where U_A is the supplied voltage, k_2 are some constructional parameters specific to each motor, Φ is the strength of the magnetic field, ω is the rotational speed, R_A is the motor's internal resistance and I_A is the current flowing through the motor. The torque generated by the motor is proportional to the current:

$$M = k_2 \Phi I_A. \tag{2.2}$$

Here, M denotes the torque [7]. Combining equations (2.1), (2.2) and using the rotational version of Newton's second law results in the second order ordinary differential equation (ODE)

$$\ddot{\theta} = \frac{k_2 \Phi}{J R_A} U_A - \frac{(k_2 \Phi)^2}{J R_A} \dot{\theta} \tag{2.3}$$

describing the rotation of the motor, where J is the moment of inertia of the motor axle. As can be seen by studying equation (2.3) closely, supplying a negative voltage to the motor will accelerate the motor in the opposite direction.

2.3 H-bridges

An H-bridge is a circuit which allows for controlling the direction in which a DC motor turns. A simplified, very basic version of an H-bridge is illustrated in Figure 2.1.

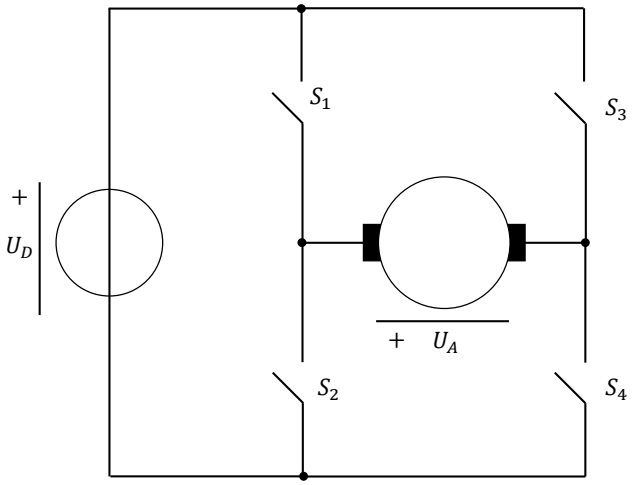


Figure 2.1. A simplified H-bridge circuit. Created using Microsoft PowerPoint.

If switches S_1 and S_4 are the only ones conducting current, the motor will spin in the normal direction, but if these are turned off and S_2 and S_3 are turned on, the motor will spin in the opposite direction. The switching is actually done with transistors which can be turned on or off by sending a signal [7].

2.4 Potentiometers

A potentiometer is an electrical component consisting of a resistive element, often an arc, with an adjustable contact called a wiper sliding over it. The component has three terminals, two connected to either end of the resistive part, and the third connected to the wiper. By letting a current pass through the component, the output current can be adjusted manually by adjusting the wiper. This is because the current will have to travel a longer or shorter distance through the resistive element depending on the wipers position. As such, the potentiometer can be seen as two resistors in series, with the wiper position deciding the resistance ratio between them [8].

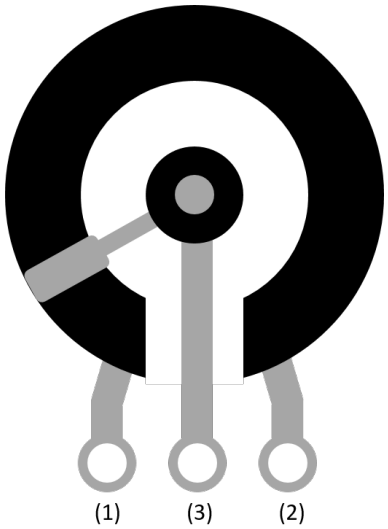


Figure 2.2. A sketch of a potentiometer with an arc of resistive material connected to two terminals, with a third terminal connected to a rotating wiper also in contact with the resistive arc. Created using Microsoft PowerPoint.

The outer terminals (1) and (2) connected to the resistive element in Figure 2.2 are connected to power and ground respectively when used in applications such as rotated angle measurements. For these applications, the middle terminal (3) is connected to the wiper acting as the data port from which voltage readings are made. Depending on the output voltage, the angle of the wiper, and thus the angle of the shaft, can be calculated.

2.5 Accelerometers

Accelerometers are electromechanical components that measure the acceleration that acts on their assigned axis. This means that an accelerometer at rest with an axis of measurement trained straight up would measure an acceleration of 1G, equal to the gravitation of the Earth. Note however that the direction of the acceleration is not the same as the attacking force, but rather the direction of the normal force due to the attacking one. As such, all measurements need to have their sign switched if the direction of attacking forces, and consequently acceleration, is required.

In principle the component is constructed as a simple damped mass-spring system, where the acceleration is calculated from the displacement of the spring. However, actual implementations of this system vary. A common example is the use of piezo-electric effects from crystal deformations caused by acceleration. These effects result in the generation of a voltage [9]. In most modern applications the accelerometer is a micro electromechanical system (MEMS), that often measures acceleration by

way of calculating the change or difference in capacitance in or between one or more electrodes when a mass is displaced between them [10].

2.6 Gyroscopes

Traditional gyroscopes are spinning discs whose orientation due to the law of conservation of angular momentum are unaffected by any rotation of the mount on which it sits. Because of this, they can be utilized to determine a change in orientation of, for example, an airplane. In addition to measuring the actual tilt of the device, a gyroscope can also be employed to measure the rate of rotation. This can be achieved by restricting the gyroscopes' freedom of movement , which due to the Coriolis effect results in a force where the gyro is restricted. By measuring this force the rate of rotation can then be derived.

In modern smaller scale electronics, the gyroscope usually takes the form of a MEMS, utilizing a similar, although modified, setup as the accelerometers in the above section. Instead of using a rotating disk, these MEMS use a vibrating mass to achieve the same effect [10].

2.7 Complementary filter

Not all sensors are perfect in all situations. Some might work very well in a low frequency spectra but poorly in high frequency ones, and other sensors may do the opposite. In order to obtain measurements that are equally valid in all spectra, a complementary filter may be utilized [11]. This combines the best properties of two sensors to achieve a better result than either sensor could achieve on their own. However, there exists a somewhat obvious drawback that two sensors have to be used instead of only one.

For example, in the case of angular measurements using a gyroscope and an accelerometer, the gyroscope only produces data that is good in the short term. This is due to the drift in the long term which is caused by the integration of errors, whilst the accelerometer only produces usable data in the long term, due to the low-pass filtering it has to go through [12]. Combining these two signals in such a way that the sum of their respective contribution equals to one, a measurement that is both fast and drift free is achieved.

2.8 Mechanics of an electric bicycle

In order to understand the problem at hand, and thus enable the creation of a system for automatic control, a mechanical analysis of the different driving situations of the bicycle was carried out. The goal of the mechanical analysis was to derive a differential equation relating the steering angle to the leaning angle of the bicycle.

This differential equation was needed for later use in creating the automatic control system in charge of balancing the bicycle, and also to analyze the physical limits involved.

The following analysis has been kept fairly simple, meaning a more in-depth analysis of the system could lead to better results. For example, the whole bicycle is considered to be rigid, and the self-balancing force originating from the wheels' rotation has been left out of the analysis. Therefore, their effects have not been taken into consideration in the derived equations.

2.8.1 Acceleration and braking

When analyzing the bicycle during acceleration and deceleration, a side-view of the bicycle was utilized when setting up the force equations, as shown in Figure 2.3. To simplify the process somewhat, the force resulting from air drag was assumed to act at the centre of mass.

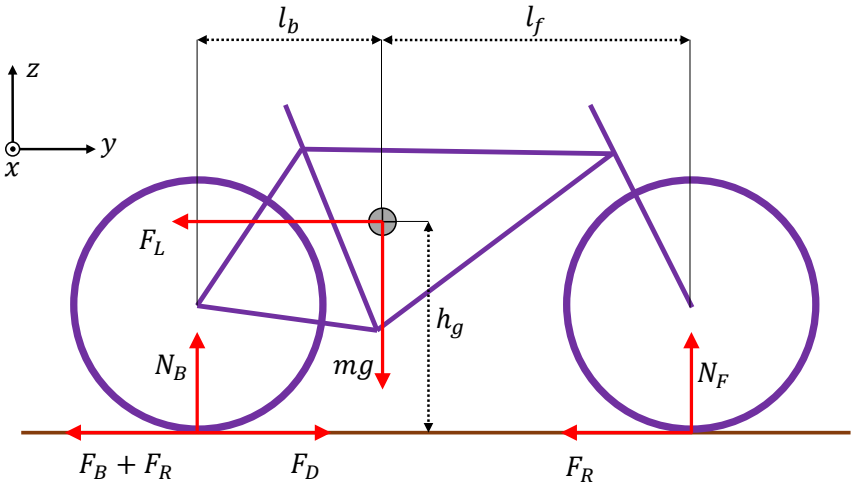


Figure 2.3. Forces acting on a bicycle during acceleration or braking, as seen from the side. Created using Microsoft PowerPoint.

Using Newton's second law,

$$\uparrow: N_f + N_b - mg = 0 \tag{2.4}$$

$$\rightarrow: F_D - F_L - 2F_R - F_B = m\ddot{y} \tag{2.5}$$

$$\circlearrowleft : h_g F_D + l_f N_F - 2h_g F_R - h_g F_B = 0 \tag{2.6}$$

where the force of air drag

$$F_L = \frac{1}{2}\rho A_f C_d \dot{y}^2. \tag{2.7}$$

Here, ρ is the density of air, A_f is the frontal area of the bicycle and C_d is the drag coefficient. A simplified model of the force due to rolling resistance is one where it is linearly proportional to the normal force with a constant:

$$F_R = C_R N \tag{2.8}$$

where N is the normal force against the ground and C_R is the coefficient of rolling resistance.

2.8.2 Relating the steering angle to the radius of curvature

While steering at a constant angle, the front and rear wheel and the centre of mass of the bicycle will trace out their own circles in the ground, as shown in Figure 2.4.

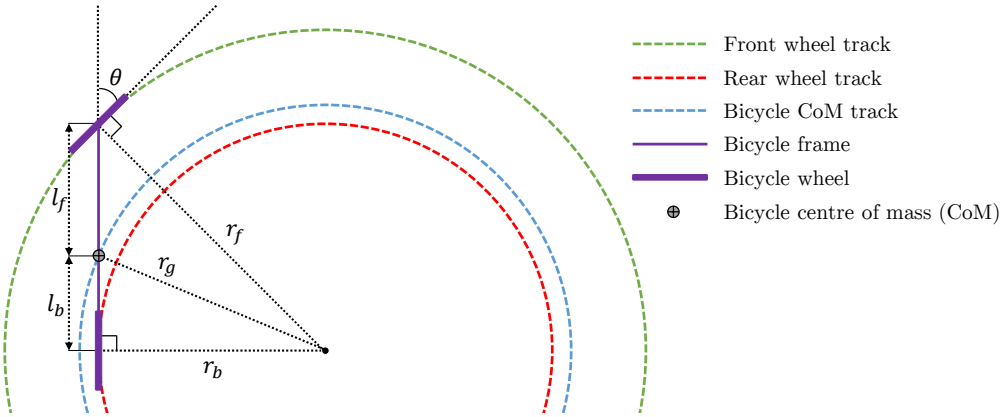


Figure 2.4. Geometry relating the different tracks of a bicycle. Created using Microsoft PowerPoint.

Using the geometry in Figure 2.4, it is concluded that

$$\tan(90° - \theta) = \frac{r_b}{L} \quad \Rightarrow \quad r_b = L\cot\theta, \tag{2.9}$$

where $L = l_f + l_b$ is the total length between the ground contact points of the tyres. The radius of curvature for the bicycle's centre of mass is now easily calculated

using the Pythagorean theorem,

$$r_g = \sqrt{l_b^2 + L^2 \cot^2 \theta}. \tag{2.10}$$

If r_g in equation (2.10) is considered as a function of θ, it is clear that r_g always will be positive. However, if it is desirable that r_g also conveys information about the direction of a turn, equation (2.10) can be rewritten as

$$r_g = \frac{\sqrt{l_b^2 + (L^2 - l_b^2)\cos^2 \theta}}{\sin \theta}. \tag{2.11}$$

Now, r_g will be negative for negative values of θ, indicating a left turn, and positive for positive values of θ, indicating a right turn.

2.8.3 Turning and instability

As opposed to the side-way approach of the acceleration and deceleration situation, a back-view of the bicycle was chosen when analyzing the forces present during turning and tilting. These are illustrated in Figure 2.5.

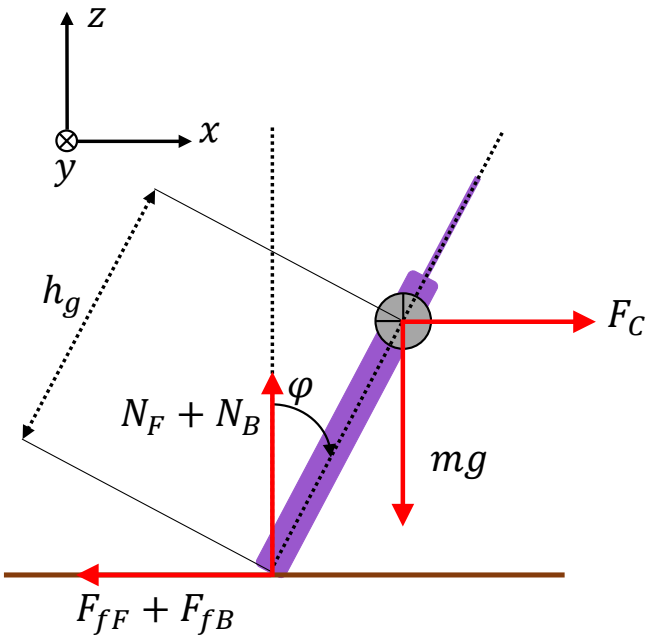


Figure 2.5. Forces acting on a bicycle during a left turn, as seen from behind. Created using Microsoft PowerPoint.

Using the rotational version of Newton's second law at the point where the tyre contacts the ground yields

$$h_g mg \sin\varphi + h_g F_C \cos\varphi = J_L \ddot{\varphi}, \tag{2.12}$$

where φ is the lean angle, J_L is the moment of inertia of the bicycle around the ground contact point and

$$F_C = -m\frac{\dot{y}^2}{r_g} \tag{2.13}$$

is the centrifugal force due to the bicycle having a circular motion. The reason for the negative sign in equation (2.13) is to make sure that the centrifugal force points in the right direction. Since the bicycle in Figure 2.5 is turning to the left, r_g will be negative because of how the steering angle was defined. The reasoning behind defining the angles this way is so that positive values on both θ and φ means "to the right". Combining equations (2.11), (2.12) and (2.13) results in

$$\ddot{\varphi} = \frac{h_g mg}{J_L}\sin\varphi - \frac{h_g m}{J_L\sqrt{l_b^2 + (L^2 - l_b^2)\cos^2\theta}}\dot{y}^2 \sin\theta\cos\varphi. \tag{2.14}$$

An analysis of the above equations leads to the realization that in order to balance the bicycle when it is tilting, it needs to turn in the same direction as the tilt. This will cause a centrifugal force with a magnitude that depends on bicycle speed and steer angle, pointing in the opposite direction. By choosing the right combination of steer angle and bicycle speed, the tilting acceleration can be canceled out or reversed.

2.9 Automatic control

2.9.1 Balance regulator

For automatic control of the balance, a linear model was chosen to keep the regulator as simple as possible. The linearization of equation (2.14) was carried out by doing a Taylor-expansion around the point ($\varphi = 0, \theta = 0, \dot{y} = v$ m/s) with any terms of higher order than the first derivative being ignored. This resulted in

$$\frac{d^2\varphi}{dt^2} \approx T_1(v) = \frac{h_g mg}{J_L}\varphi - \frac{v^2 m h_g}{J_L L}\theta. \tag{2.15}$$

The error of this model is shown in Figure 2.6, where there are large relative errors for some combinations of φ and θ.

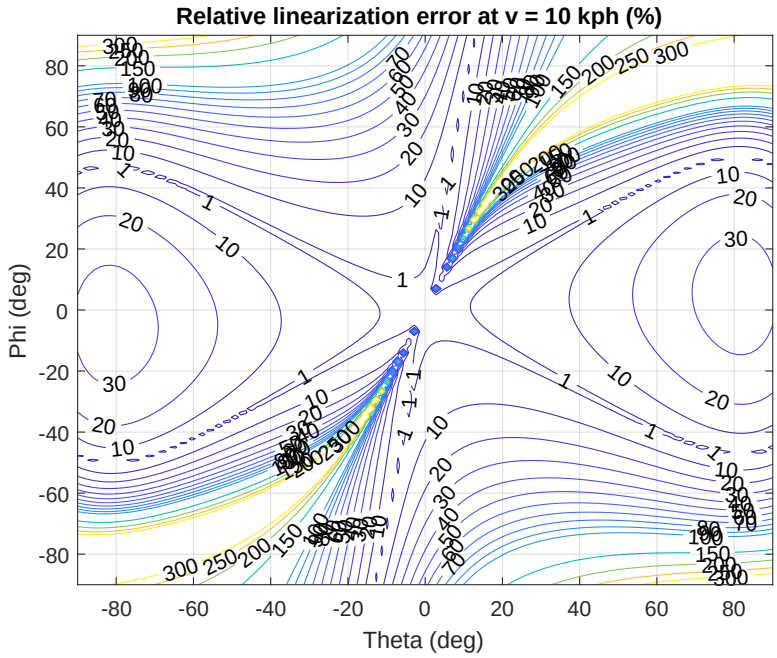


Figure 2.6. Relative error due to linearization for different combinations of lean (φ) and steer angle (θ). Created using MATLAB.

Using the linearized model, a simple state space model can be derived using the following method [13]:

Let the state variables

$$x_1 = \varphi \tag{2.16}$$

$$x_2 = \dot{\varphi}, \tag{2.17}$$

then

$$\dot{x_1} = \dot{\varphi} = x_2 \tag{2.18}$$

$$\dot{x_2} = \ddot{\varphi} = \frac{mh_g g}{J_L} x_1 - \frac{mh_g v^2}{J_L L} \theta, \tag{2.19}$$

and the matrices for the state space model become

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ \frac{mh_g g}{J_L} & 0 \end{bmatrix} \tag{2.20}$$

$$\mathbf{B}(v) = \begin{bmatrix} 0 \\ -\frac{mh_g v^2}{J_L L} \end{bmatrix} \tag{2.21}$$

$$\mathbf{C} = \begin{bmatrix} 1 & 0 \end{bmatrix}. \tag{2.22}$$

Now, a feedback-loop is created with

$$\theta = -\mathbf{L}\mathbf{x} + \varphi_{ref}, \tag{2.23}$$

where $\mathbf{x}$ is our state vector, φ_{ref} is the desired lean angle and $\mathbf{L}$ is the feedback vector which will be constructed by choosing the placement of the poles. The poles of the closed system is given by the eigenvalues of the matrix $\mathbf{A} - \mathbf{BL}$ and therefore the poles are the solution to the equation

$$\det(\lambda \mathbf{I} - (\mathbf{A} - \mathbf{BL})) = 0. \tag{2.24}$$

Solving this equation yields

$$\lambda = \frac{mh_g v^2}{2J_L L} l_2 \pm i \sqrt{-\frac{m^2 h_g^2 v^4}{4J_L^2 L^2} l_2^2 - \frac{mh_g g}{J_L} - \frac{mh_g v^2}{J_L L} l_1}. \tag{2.25}$$

It can be seen that the real part is given by

$$\mathrm{Re}(\lambda) = \frac{mh_g v^2}{2J_L L} l_2, \tag{2.26}$$

therefore

$$l_2 = \frac{2J_L L}{mh_g v^2} \mathrm{Re}(\lambda). \tag{2.27}$$

The imaginary part is given by

$$\mathrm{Im}(\lambda) = \sqrt{-\mathrm{Re}(\lambda)^2 - \frac{mh_g g}{J_L} - \frac{mh_g v^2}{J_L L} l_1}. \tag{2.28}$$

Solving for l_1 results in

$$l_1 = -\frac{J_L L}{mh_g v^2}\left(\mathrm{Im}(\lambda)^2 + \mathrm{Re}(\lambda)^2 + \frac{mh_g g}{J_L}\right) \tag{2.29}$$

and therefore the feedback vector is

$$\mathbf{L} = \begin{bmatrix} -\frac{J_L L}{mh_g v^2}\left(\mathrm{Im}(\lambda)^2 + \mathrm{Re}(\lambda)^2 + \frac{mh_g g}{J_L}\right) & \frac{2J_L L}{mh_g v^2}\mathrm{Re}(\lambda) \end{bmatrix}, \tag{2.30}$$

where the 2 poles λ can be chosen and v is the current speed of the bicycle.

2.9.2 Steering motor

To ensure that the bicycle was steering in the intended direction at all times, a regulator for the steering was also constructed. A state space model was also chosen for this regulator, as both the steer angle and steer rotational speed were to be measured directly. Using the same method as in Section 2.9.1 and the differential equation (2.3) resulted in the matrices and vectors

$$\mathbf{A}_S = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{(k_2\Phi)_S^2}{J_S R_{AS}} \end{bmatrix}, \tag{2.31}$$

$$\mathbf{B}_S = \begin{bmatrix} 0 \\ \frac{(k_2\Phi)_S}{J_S R_{AS}} \end{bmatrix}, \tag{2.32}$$

$$\mathbf{C}_S = \begin{bmatrix} 1 & 0 \end{bmatrix}, \tag{2.33}$$

where the subscript S specifies that the variables belong to the motor controlling the steering. As this regulator is time-invariant, the feedback vector $\mathbf{L}_S$ was calculated by usage of MATLAB's place() function.

2.9.3 Driving motor

Automatic control for the driving motor was also implemented to ensure that the bicycle maintained the desired speed. For this regulator, a simple proportional integral (PI) type was chosen, as only the rotational speed of the motor is measured.

2.10 The Runge-Kutta 4-method

Runge-Kutta are a family of methods for approximating ODEs numerically. The four indicates that it is a fourth order method, meaning that every time the step size is halved, the error is reduced by a factor of 2^4 [14]. Given the initial value problem (IVP)

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases} \tag{2.34}$$

one step using the Runge-Kutta 4 method is given by

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{2.35}$$

where h is the step size, and

$$k_1 = f(t_i, y_i), \tag{2.36}$$

$$k_2 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right), \tag{2.37}$$

$$k_3 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right), \tag{2.38}$$

$$k_4 = f(t_i + h, y_i + hk_3). \tag{2.39}$$

The ODE is approximated by reiterating (2.35).

2.11 Approximating second order ODEs

A second order ODE can be solved by the Runge-Kutta 4 method by rewriting it as a system of first order ODEs. Given the IVP

$$\begin{cases} y'' = f(t, y, y') \\ y(0) = y_0 \\ y'(0) = y_0' \end{cases} \tag{2.40}$$

let

$$\mathbf{u} = \begin{bmatrix} y \\ y' \end{bmatrix}, \tag{2.41}$$

then

$$\frac{d\mathbf{u}}{dt} = \begin{bmatrix} y' \\ y'' \end{bmatrix} = \begin{bmatrix} u_2 \\ f\left(t, u_1, u_2\right) \end{bmatrix} = \mathbf{F}\left(\mathbf{u}\right), \tag{2.42}$$

and the new IVP

$$\begin{cases} \frac{d\mathbf{u}}{dt} = \mathbf{F}\left(\mathbf{u}\right) \\ \mathbf{u}_0 = \begin{bmatrix} y(0) \\ y'(0) \end{bmatrix} \end{cases} \tag{2.43}$$

has been given, which can be solved using the Runge-Kutta 4 method.

Chapter 3

Demonstrator

A prototype unit was constructed in order to verify the theory and to perform experiments to answer the research questions.

3.1 Simulation

A simulation was carried out by solving equations (2.14) and (2.3) numerically using the Runge-Kutta 4 method in MATLAB. One more differential equation was also solved describing the bicycle's forward acceleration. The written MATLAB code can be found in appendix C. The developed regulators were implemented in the simulation to test them in theory. The simulation was also used to test if different motors available for purchase would have adequate performance.

Figure 3.1 shows the outcome of different events and bicycle commands. In this simulation, the bicycle is already up to speed (7 km/h) at $t = 0$. At $t = 1$ s, the bicycle is given the command to turn left by changing the lean angle reference to $-15°$. At $t = 2$ s, randomly distributed disturbing forces are introduced for the rest of the simulation. These are biased to the left for 3 seconds and to the right thereafter, creating a simplified model of shifting wind. At $t = 3$ s, the bicycle is commanded to increase speed to 12 km/h. At $t = 6$ s, the bicycle is commanded to turn to the right. As can be seen in the forces plot, the simulation predicts that tyres will be a problem as the friction exceeds the limit when the random forces are introduced.

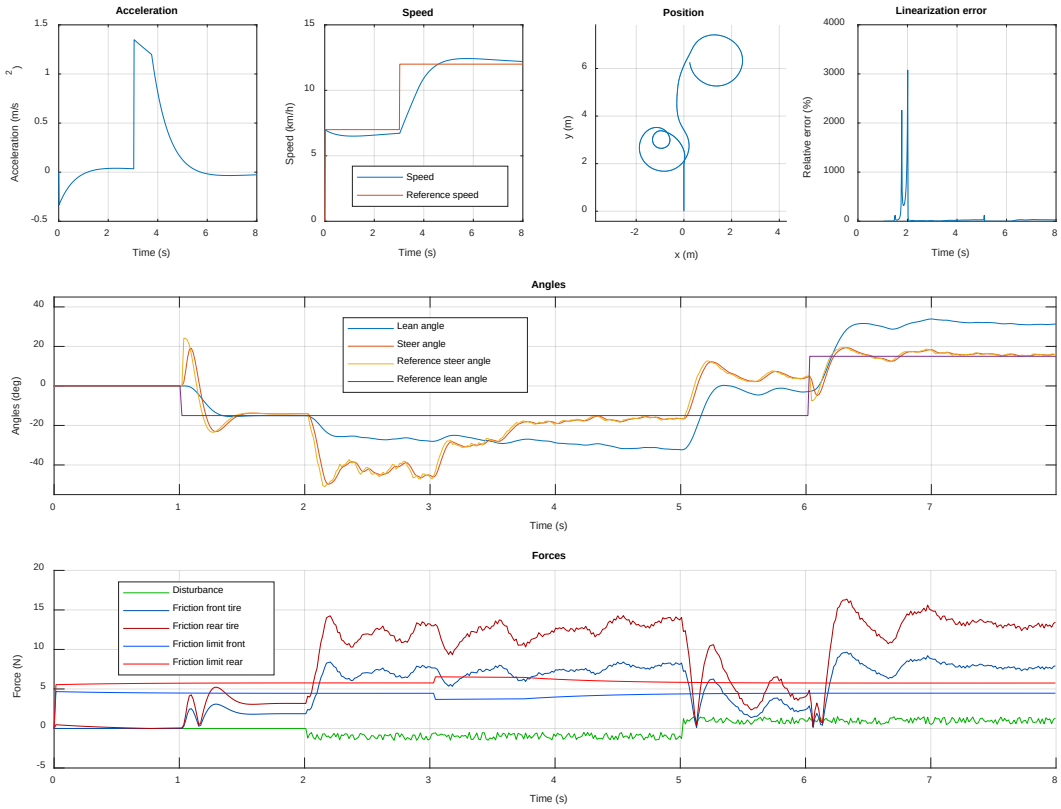


Figure 3.1. Simulating the bicycle system with different events and bicycle commands. Created using MATLAB.

3.2 Component selection

Using the simulation program, requirements on motor performance were identified. The steering motor needed to be fast enough to achieve the desired steer angle quickly, but also have enough torque to accelerate up to speed quickly. The goal was to find a motor with the right properties to simplify the construction of the prototype by not requiring gearing. The choice of motor for controlling the steering assembly fell on Nidec DMN37KA. The motor has a rated power output of 9.2W, a no-load speed of 4300 rpm and a stall torque of 0.16 Nm. To control the steering motor, an Olimex BB-VNH3SP30 motor driver was used, due to them being able to handle the required current to drive the motors.

For the accelerometer and the gyroscope, the GY-521 MPU 6050 breakout board was chosen, as it combines both components in a small package for a relatively low price. The MPU 6050 chip also only requires two ports for transferring data from

all six axes, meaning the freed up ports can be used for other purposes. As for the specific break out board, the GY-521 was chosen due to it having the AD0-pin included, which makes it possible to change the I^2C address, which is needed in order to use more than one.

To measure the angle of the front wheel, the potentiometer WAL305 5K from Contelec was chosen because of its resolution of 0.3 degrees. The hole in its center was also large enough for the motor shaft of the steering motor above to fit.

A simple photo micro sensor was used along with a 3D-printed encoder wheel to measure the velocity of the bicycle.

Finally, the pulley and driving belt were chosen after some simulation and estimation of the required speeds of the bicycle, which resulted in a desired drive ratio of 4.

3.3 Electronics

In addition to the sensors, motor drivers and the Arduino Uno, two simple circuit boards were constructed. One power supply board which distributes the power to all components and one user interface board which has two buttons and five LEDs. The circuitry for both boards and the wiring of all the electrical components are available in appendix A.

The power supply board has three 12 V output connectors which supply power to the motors and nine 5 V output connectors that supply power to the sensors and the Arduino. The input to the board is 12 V from the battery bank, and an LM7805 regulates the voltage down to 5 V. On each side of the LM7805 there are capacitors that help with stability and transient responses. It is the same circuit as recommended in the data sheet [15], except that a 47 µF capacitor was used on the input side instead of 33 µF, as there was one readily available during construction.

The interface board was created to be able to turn on or off the program at a given time. This is done with the two buttons available on the board. The board also contains five LEDs to be able to receive some sort of feedback, which for example could be helpful when troubleshooting code.

The mounted electronics on the bicycle is shown in Figure 3.2.

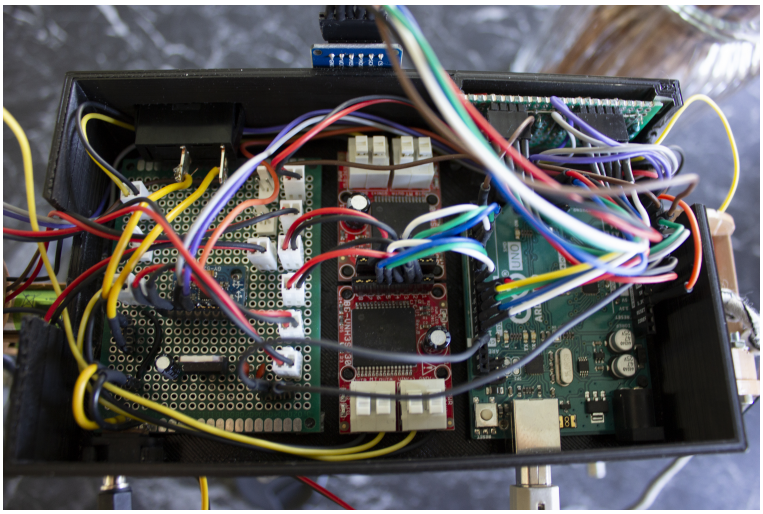


Figure 3.2. Photograph of the box housing the electronics.

3.4 Construction

Once the components needed were identified a prototype was modeled in Solid Edge ST9. First, the size of the bicycle was decided on by choosing appropriate wheel sizes, where limitations in the manufacturing process had to be considered. It was decided that the wheel sizes would be 160 mm in diameter. The modeling then started with an idea that the batteries should be located in the frame to keep the center of mass around the middle of the bicycle. Mounts for a box to contain the electronics were modeled at the top of the frame. Due to size limitations in 3D-printing, the frame was split into two parts, and the rear part that holds the rear wheel and driving motor was modeled later.

Focus was then shifted to finishing the steer assembly while considering availability and price of components such as ball bearings. The result is shown in Figure 3.3, where the steering DC motor is mounted on top of the steer assembly, held in place by a cover. The angle sensor (linear potentiometer) that is mounted on the motor axle is also visible.

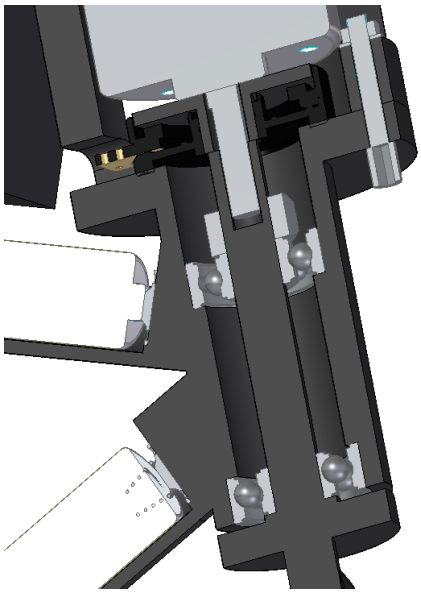


Figure 3.3. CAD model of the steer assembly. Created using Solid Edge ST9.

After the steer assembly was finished, the rear part of the the frame was modeled, shown in Figure 3.4, along with a mount for the driving motor. This mount included rails so that the driving motor's position could be adjusted to ensure proper tension in the timing belt that transfers the torque to the rear axle.

Figure 3.4. CAD model of the rear part of the bicycle. Created using Solid Edge ST9.

The completed CAD model is shown in Figure 3.5. For drawings of the 3D-printed parts and an exploded assembly, refer to appendix B.

Figure 3.5. CAD model of the completed bicycle. Created using Solid Edge ST9. Rendered in KeyShot 6.3.

All the brown parts and the black electronic box in the figure were 3D-printed on an Ultimaker 2 with polylactic acid (PLA) as material. The wheels were created by laser cutting acrylic, and the tyres were simply made by gluing rubber directly onto the wheels. The torque transfer from the rear axle to the rear wheel is via two collars glued to each side of the wheel. The same method was used to attach and fix the pulley and the pulse code wheel to the rear axle. This method was probably not ideal to create a long lasting or a serious construction, but the decision was made to decrease model complexity, cost and save time. In all cases the glue used was cyanoacrylate, usually called "super glue". The final construction is shown in Figure 3.6.

Figure 3.6. Photograph of the bicycle.

3.5 Experimental setup

In order to ascertain and test the feasibility of the system, a significant amount of testing was done. To collect data on how the bicycle reacted and ran during the tests, in addition to simple observations, a micro SD card reader situated on a breakout board was installed on the Arduino. This was due to the fact that because the bicycle had to move during testing, data could not be transfered directly to a computer via a USB cable. Data on PWM levels, angles, accelerations, loop times and velocities were then written to an SD card in CSV-format for later analysis in MATLAB. A few of the later test runs were also filmed and analyzed frame by frame to get a better understanding of what was happening.

To reduce the risk of the demonstrator breaking during testing, mainly from falling over and/or driving into walls, two thin ropes were fastened to the frame of the bicycle. One rope was was attached at the front of the bicycle and another at the back, both seen in Figure 3.6 above. This ensured that the bicycle could be lifted in the case of danger. A section of code was also added to the Arduino, where one of the accelerometers was utilized in such a way that a quick pull in the vertical direction of the bicycle would turn off all motors.

Because of the nature of the chosen way of balancing, the bicycle had to reach a sufficient velocity before any actual balancing could realistically be performed. The initial idea was to simply use the above mentioned ropes to keep the bicycle upright until a hard coded velocity had been reached. However, this particular approach was harder than initially estimated due to the difficulties with keeping the bicycle upright while maintaining an appropriate amount of friction against the ground.

To solve this issue, a single support wheel was constructed and installed on the right outermost part of the rear axle. This wheel was made to be slightly smaller than the regular wheels, so that when the bicycle reached a sufficient velocity and balance regulation was initiated, it could drive without the support wheel touching the ground. The resulting size of the wheel was such that when resting on the support wheel, the bicycle tilted slightly below 10 degrees.

The experiments were mainly conducted in a corridor with plastic flooring, with a few test runs being done on stone flooring as well as asphalt.

For the most part, the testing was done in groups of a few test drives at a time, after which the data was analyzed in MATLAB. Potential problems and areas of improvement both when it comes to construction and regulator variables were determined. Improvements were implemented on these areas and another group of tests were conducted. When the Arduino code was finished and most parameters were tweaked, the support wheel could be removed. The bicycle was then held upright for approximately one second while it accelerated up to speed, after it was released to balance on its own.

Chapter 4

Results

The results from the experiments were mixed. On some occasions, the bicycle managed to keep itself balanced. On others it seemed that tyre friction was a limiting factor. Some times the Arduino locked up, which caused the PWM signals to stop working. This set the motor speeds to either be fully on or fully off, sometimes resulting in damage to the bicycle fork.

Figure 4.1 shows a typical run where the bicycle exceeds the frictional limit between the tyres and the flooring. This was verified by a frame-by-frame analysis of a recording of the test run. This particular test run was conducted on dusty plastic flooring. The bicycle was lifted at approximately 1.5 seconds to prevent damage, hence the increase in speed. The lean angle starts at approximately 8 degrees due to the fact that this run utilized the support wheel on the right side of the bicycle until it got up to speed. The steer reference angle was initially -4 degrees, as a counter measure to the support wheel dragging the bicycle to the right.

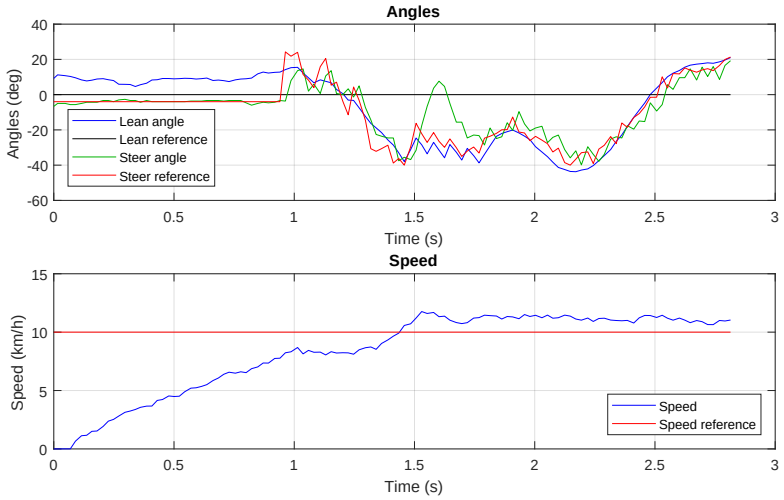


Figure 4.1. A typical experimental result where the bicycle loses friction. Created using MATLAB.

Figure 4.2 below shows data from one of the longest test runs conducted where the bicycle managed to balance itself. The test was carried out on parquet flooring without the use of the support wheel. The bicycle was held upright until the balance regulator was turned on. This was programmed to happen when the speed passes 6 km/h, as can be seen at around the 0.8 second mark in the plot.

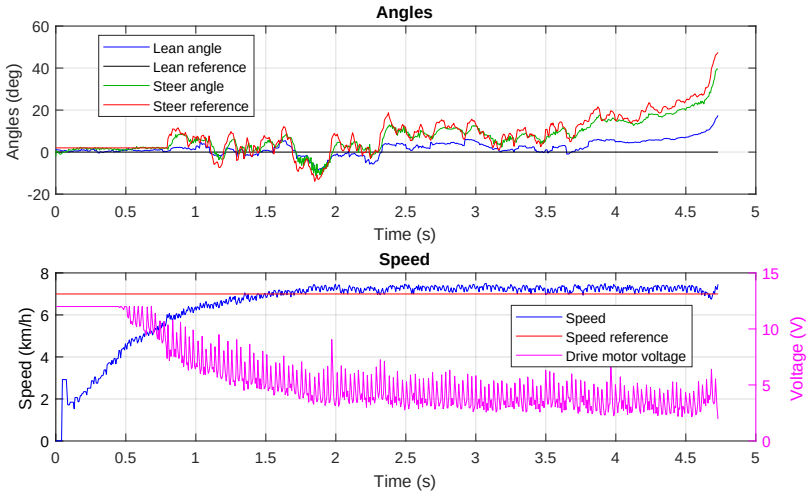


Figure 4.2. A typical experimental result where the bicycle managed to keep itself balanced. Created using MATLAB.

Figure 4.3 below is a simulation using the same initial conditions as the experimental setup used in Figure 4.2 for comparison.

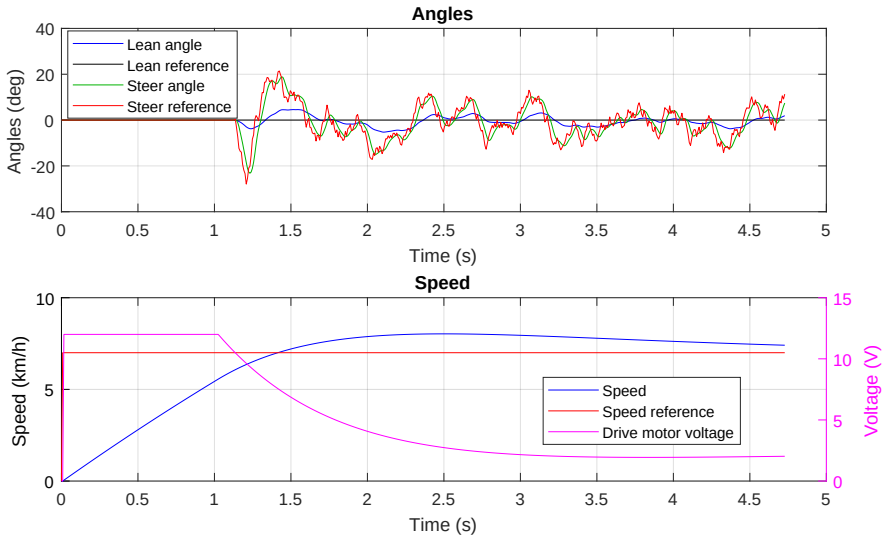


Figure 4.3. Results of simulating the bicycle, matching the initial values of the experimental run in Figure 4.2. Created using MATLAB.

Chapter 5

Discussion and conclusions

5.1 Discussion

Firstly, the general outcome of testing (focusing on Figure 4.1) will be addressed. In this figure, the bicycle fell a few moments after the regulator was initiated. After filming a few cases of this occurring and doing a frame by frame analysis of what was happening, the most likely cause of this issue was insufficient friction between the wheels and the ground. In order to straighten itself up from leaning on the support wheel, the bicycle needed to turn right first so as to create a centrifugal force to pull the bicycle up. To stop the bicycle from tipping over too much on the other side, it naturally had to turn left which can also be seen in aforementioned figure. When examining this and similar cases in the video material, the bicycle can be seen to continue traveling in whatever direction it was heading prior to the change in steer angle, solidifying the suspicion that friction is the main problem in these cases.

The wheels were a challenge to construct in general, not only in terms of trying to mimic the friction of an actual tire. When laser cutting them from plexiglas, the rim of the wheels ended up with a slight sloped face, as the laser did not cut through the entire 5 mm thickness at once. While this might not be a major issue, it might very well still be a contributing factor to the issues of the system. Another possible issue with the wheels is the fastening of rubber on them. This rubber fastening proved to be more difficult than anticipated which resulted in a somewhat uneven amount of glue being placed around the edge, making the wheels not as smooth as they ought to be. Another possible reason is that the wheels do not have proper tires. As such they are not filled with air as actual bicycle tires are, meaning they do not provide any dampening of vibrations from the ground, but rather transfer them straight to the rest of the bicycle and consequently the sensors.

Another weak point of the constructed demonstrator, both in terms of vibrations and overall durability, is the bicycle fork on the front wheel. Because of its thin

design, it flexes upward more than can be considered reasonable due to the weight of the demonstrator. Not only does this lead to a lot of unnecessary vibrations due to it flexing back and forth during driving, but it also broke three times during testing.

Despite the wheels not providing any dampening, there were other sources of vibrations. For example, the drive and steering motor contribute to a lot of vibrations, both of which could have been isolated better. The overall small size of the demonstrator also contributed to more severe vibrations. This is because any smaller cavities or bumps in the flooring were much bigger in relation to the demonstrator as opposed to what would be the case with an actual bicycle. All of these vibrations combined with general sensor noise led to a need to filter some signals. However, the signals were still quite noisy even after processing them, which in turn might have led to increased instability of the bicycle.

Despite the above problems, the bicycle did manage to balance itself sometimes, as can be seen in test runs such as Figure 4.2. Of course, this particular test was fairly short, as the bicycle had to be stopped before it crashed into a wall. Regardless, given the behavior of the system in the graph as well as the observations of how it looked from the outside, it seems very likely that it would have continued balancing a fair bit longer, given enough free space to drive on. This points toward the regulator and the underlying idea being correct. Something that might prove problematic in the regulator is the spots in the domain with large relative errors visible in Figure 2.6. These unfortunately seem to coincide with the desired lean angles and their appropriate steer angles. However, because the system won't stay in any lean/steer angle configuration for more than a very short moment, these errors should not have too big of an impact for small values of steer and lean angles. In Figure 3.1, showing a simulation of the system, the bicycle can be seen passing one of these spots around the 2 second mark. The error spikes but quickly reduces again.

The biggest issue in determining the feasibility of the constructed system, even if the above problems are ignored and only the good test runs are considered, is the unfortunately short distances the bicycle has been allowed to run in. This is in part another byproduct of the small stature of the demonstrator. As stated earlier, even relatively small imperfections in the ground had a much bigger impact on the system than they would have had on an actual real-sized bicycle. Because of this, the location in which testing could have been conducted became limited, as large halls with clear floors made of a material with good friction, were hard to come by. Using a real bicycle for the testing was unfortunately not possible however, because of the restricted nature of the available budget of around 1000 Swedish crowns. This budget was slightly exceeded to create this demonstrator.

Other smaller issues that contributed to difficulties in determining the feasibility of the system include how the electrical components react and work together, as well as their implementation. As it was not possible to record data straight from the

Arduino via USB during testing, an SD card (as stated in Section 3.5) for data to be written onto had to be installed. The problem with this particular solution was that the library needed to write onto the card takes up around 30-40 % of the dynamic memory of the Arduino Uno. This led to memory issues and possibly instability of the microcontroller.

Another component that caused problems was the MPU 6050, particularly the one assigned as the lean angle measurer, which caused the Arduino to freeze fairly frequently. The reason behind this isn't entirely clear, but we suspect that it is caused by disturbances in the data transfer between the Arduino and MPU through the I^2C bus. This bus is designed primarily for short distance communication only. The disturbances in question could originate from the PWM signals sent to the drive motor, whose cables lie close to the communication cables of the MPU.

The final issue that made testing and tweaking of the system difficult was the batteries and their supplied voltage. Because the bicycle in its current state pulls current straight from the batteries without a voltage regulator in-between, the voltage supplied to the motors changes as the batteries become increasingly drained. Voltage supply variation affects the speed of the motors, which in turn affects the system as a whole, and thus, seemingly promising regulator parameters would have to be changed after a while.

5.2 Conclusion

The system has quite a few issues that makes balancing difficult, mainly the friction with the ground. This stemmed from bad wheels and a low weight of the system. The actual regulator however, does seem to work fairly well. This is because when the bicycle did keep its balance, it seemingly would have continued to do so given enough space for it to move in. More testing has to be conducted before a more decisive conclusion can be drawn, preferably done on a larger demonstrator.

5.3 Suggested improvements

In light of the above discussion, it is evident that a lot of things can be done to improve both the overall performance of the system itself, as well as the overall ease of testing the system. Following is a few improvements that should be taken into account, should further testing with a new demonstrator be conducted.

Firstly, and perhaps most importantly, simply using an actual bicycle, or at least a bigger demonstrator for testing could fix a lot of the bigger problems with the system. The increased weight of the system would increase its grip with the ground significantly, potentially eliminating or at least decreasing the risk of the wheels slipping. Having proper wheels with air-filled tires would not just provide some well needed damping, but would decrease the system vibrations caused by smaller

surface imperfections. An example of a surface with smaller imperfections is asphalt. Another side effect of being able to drive on asphalt is that testing could be conducted on more or less any empty parking lot, completely removing the issue of finding a good testing location. The flexing problems of the fork would also be solved by using a real bicycle, as it would be constructed from a much sturdier material like steel as compared to the material now used in the demonstrator. Of course, even if an up-scaling was not done, simply making the fork out of a stiffer material would solve these flexing problems. However, increasing the size of the demonstrator comes with some obvious drawbacks, mainly the fact that the motors would have to be a lot more powerful, and consequently a lot more expensive. As would the battery pack needed to power these motors.

Electronics wise, the components used in this demonstrator should work fine for the most part even in an up-scaled model. One exception would be the potentiometer used for angular readings on the front wheel which would have to be replaced by a larger version. It is technically possible to use the accelerometer/gyro already situated on the front wheel to calculate the steer angle, which is something that was initially tested on this demonstrator. This is not recommended however, as the implementation of this was prone to a lot of errors on top of having a restricted domain of allowed steer and lean angles. Some kind of potentiometer or encoder solution would therefore be preferred. Also, to avoid memory issues in the micro controller, a board such as the Arduino Mega or a Raspberry Pi would be recommended, even though a Uno does seem to work, albeit barely with the SD library installed.

Regarding the issues with the Arduino freezing due to the MPU, a simple upscaling of the system could possibly fix this as well. That is, if the issue is related to disturbances in the communication due to PWM signals. An upscaling would mean that the wires could be more easily separated. Otherwise, using shielded cables could solve these issues.

Lastly in terms of electronics, simply installing a voltage regulator between the battery pack and the rest of the system to allow for more consistent voltages is a must, to avoid changes in system behavior due to the batteries becoming drained.

Finally, software wise, there might be room for further optimization in the code. In its current state however, it has been stripped of most, if not all, unnecessary processes. Also, a change from the complementary filter to a Kalman filter would likely help to improve the overall accuracy of the measured lean angle.

Bibliography

[1] H. Yetkin, S. Kalouche, M. Vernier, G. Colvin, K. Redmill, and U. Ozguner, "Gyroscopic stabilization of an unmanned bicycle." American Automatic Control Council, June 2014, pp. 4549–4554.

[2] M. Yamakita, A. Utano, and K. Sekiguchi, "Experimental study of automatic control of bicycle with balancer." IEEE, October 2006, pp. 5606–5611.

[3] M. Baquero-Suárez, J. Cortés-Romero, J. Arcos-Legarda, and H. Coral-Enriquez, "A robust two-stage active disturbance rejection control for the stabilization of a riderless bicycle," *Multibody System Dynamics*, pp. 1–29, January 2018.

[4] "Arduino Uno Rev3," Accessed: 2018-05-13. [Online]. Available: https://store.arduino.cc/arduino-uno-rev3

[5] NXP Semiconductors, "UM10204 I2C-bus specification and user manual," Accessed: 2018-05-14. [Online]. Available: https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[6] "Arduino - SPI," Accessed: 2018-05-18. [Online]. Available: https://www.arduino.cc/en/Reference/SPI

[7] H. Johansson, *Elektroteknik*. Stockholm: Institutionen för maskinkonstruktion, Tekniska högsk., 2006.

[8] "Fundamentals of Potentiometers," Accessed: 2018-05-19. [Online]. Available: http://www.resistorguide.com/potentiometer/

[9] "Introduction to Accelerometer Measurement," Accessed: 2018-05-19. [Online]. Available: https://www.omega.com/prodinfo/accelerometers.html

[10] Majid Dadafshar, "Accelerometer and Gyroscopes Sensors: Operation, Sensing, and Applications," Accessed: 2018-05-19. [Online]. Available: https://www.maximintegrated.com/en/app-notes/index.mvp/id/5830

[11] W. Higgins, "A comparison of complementary and kalman filtering," *Aerospace and Electronic Systems, IEEE Transactions on*, vol. AES-11, no. 3, pp. 321–325, May 1975.

[12] Sanghyuk Park, Jonathan How, "MIT Lecture on Complementary and Kalman filters," Accessed: 2018-05-19. [Online]. Available: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-333-aircraft-stability-and-control-fall-2004/lecture-notes/lecture_15.pdf

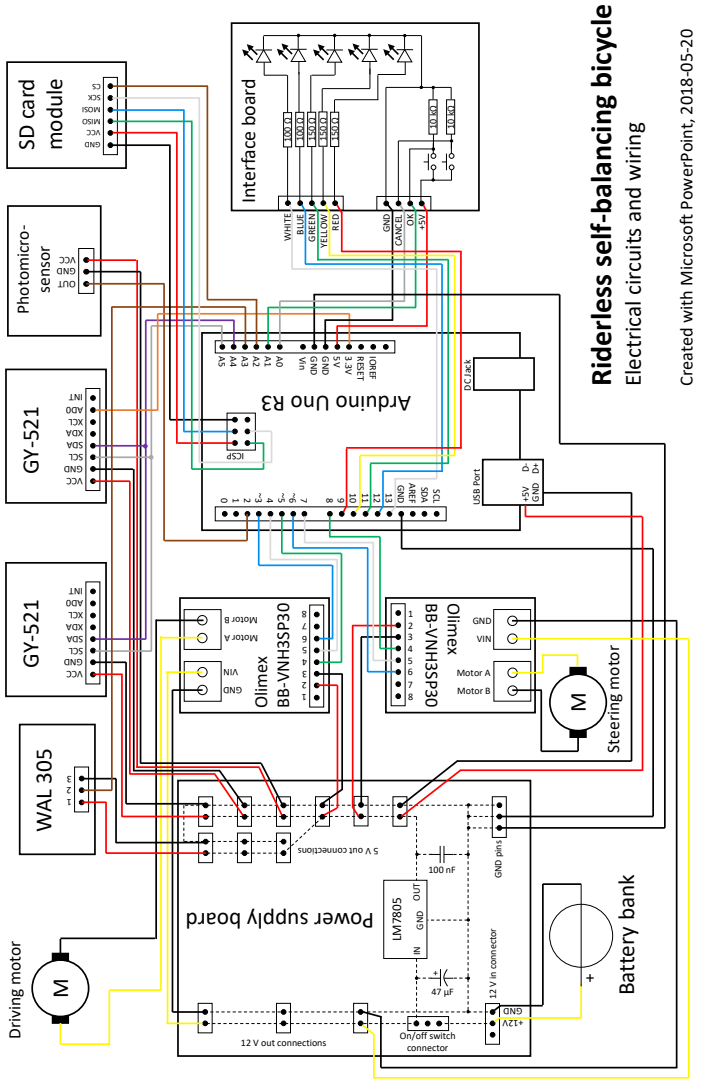
[13] T. Glad, *Reglerteknik : grundläggande teori*, 4th ed. Lund: Studentlitteratur, 2006.

[14] T. Sauer, *Numerical analysis*, second edition, pearson new international edition.. ed. Harlow, Essex: Pearson, 2014.

[15] Fairchild Semiconductor, "LM78XX/LM78XXA 3-Terminal 1A Positive Voltage Regulator," Accessed: 2018-05-19. [Online]. Available: http://hades.mech.northwestern.edu/images/6/6c/LM7805.pdf

Appendix A

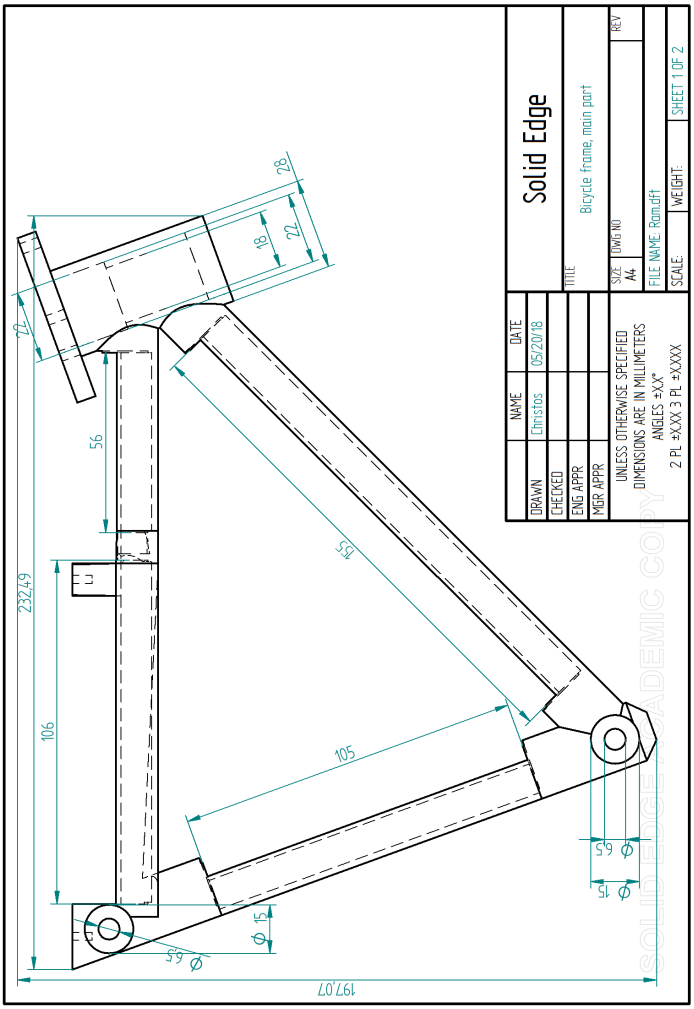
Electrical circuits and wiring

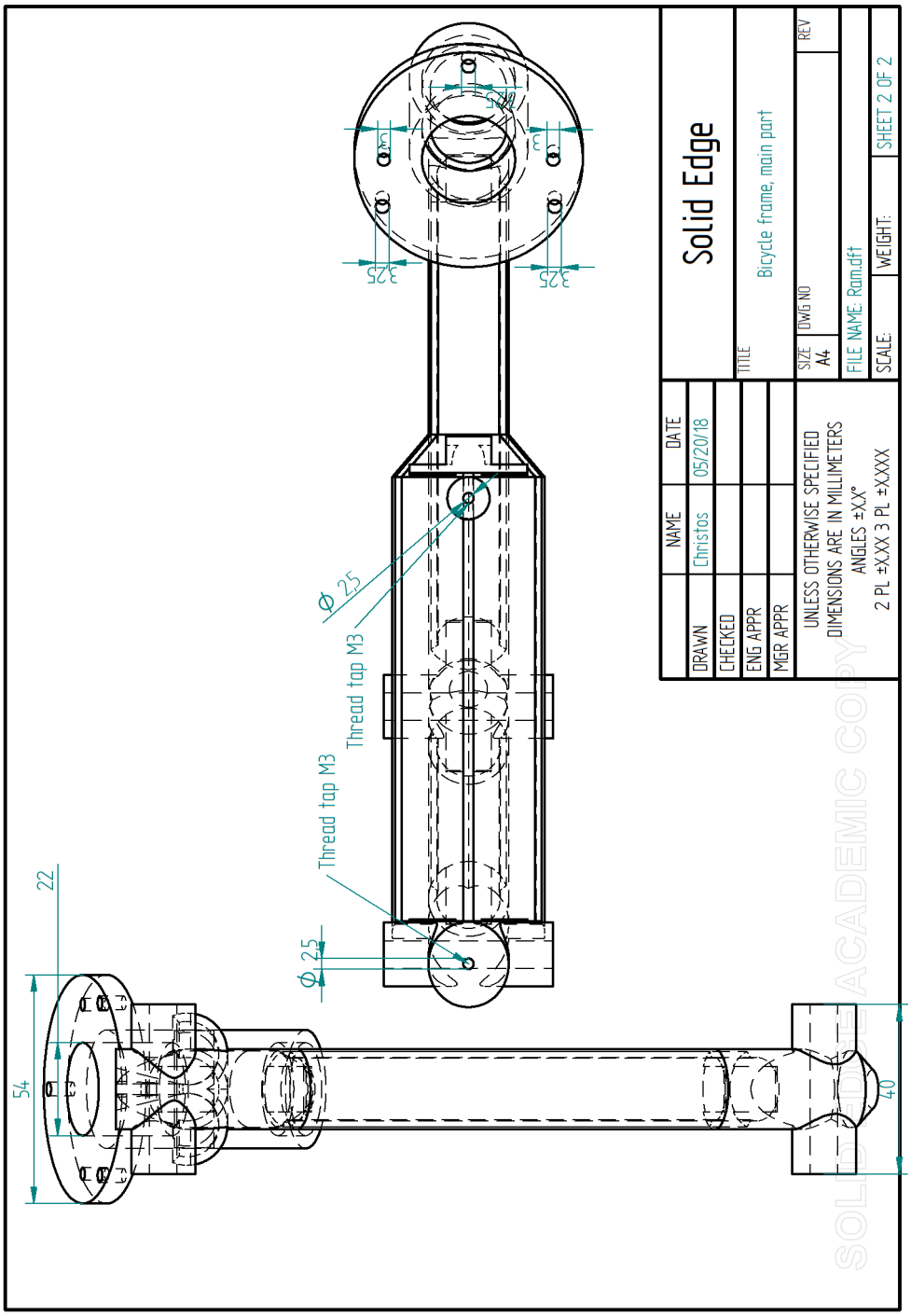


Appendix B

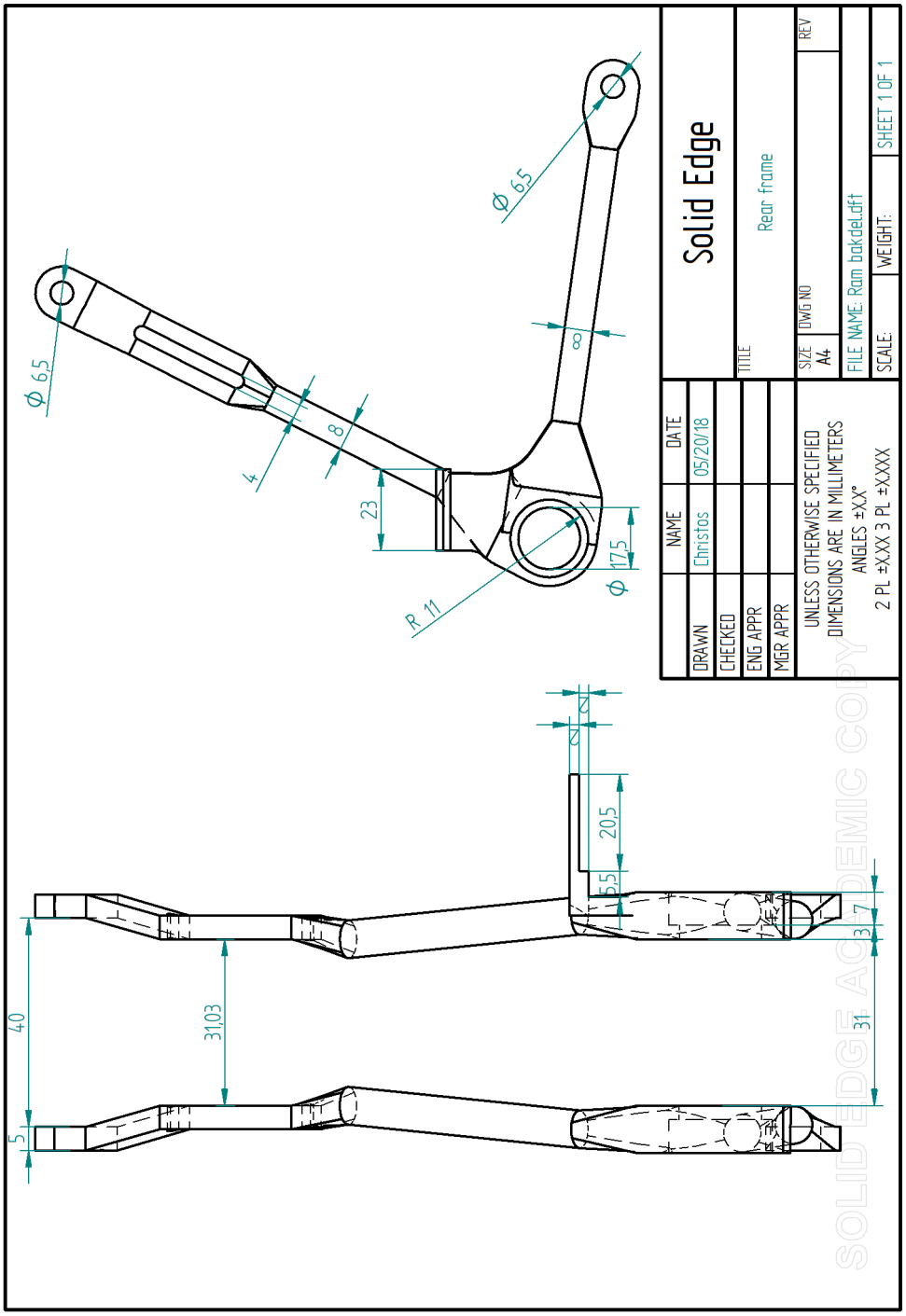
CAD drawings

B.1 Frame

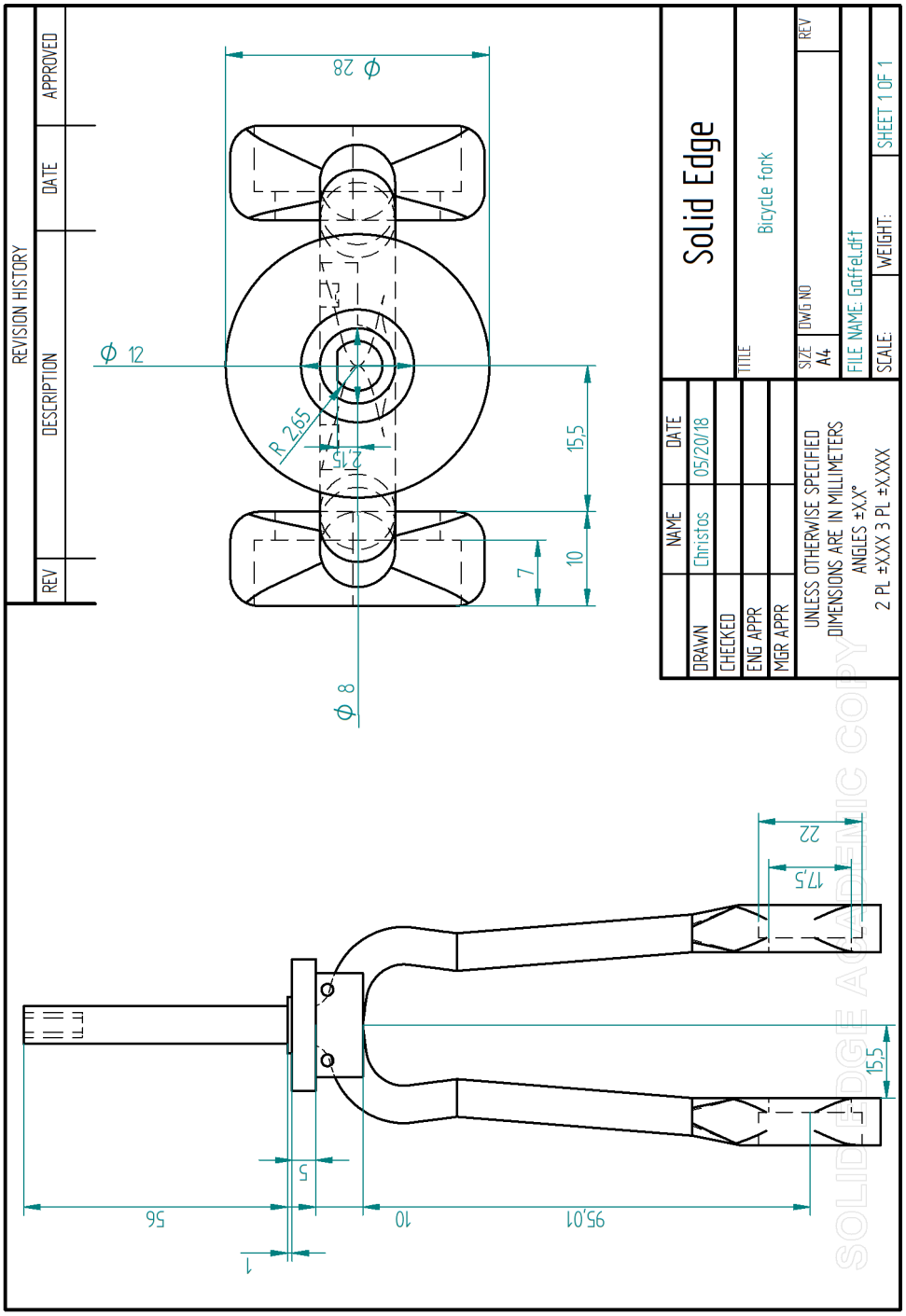



Solid Edge
Bicycle frame, main part
TITLE
SIZE A4
DWG NO
REV
FILE NAME: Ram.dft
SCALE:
WEIGHT:
SHEET 2 OF 2
NAME
DATE
DRAWN
Christos
05/20/18
CHECKED
ENG APPR
MGR APPR
UNLESS OTHERWISE SPECIFIED
DIMENSIONS ARE IN MILLIMETERS
ANGLES ±X.X°
2 PL ±X.XX 3 PL ±X.XXX
Thread tap M3
Thread tap M3
Ø 25
Ø 25
22
54
40
3.25
3.25
SOLID EDGE ACADEMIC COPY

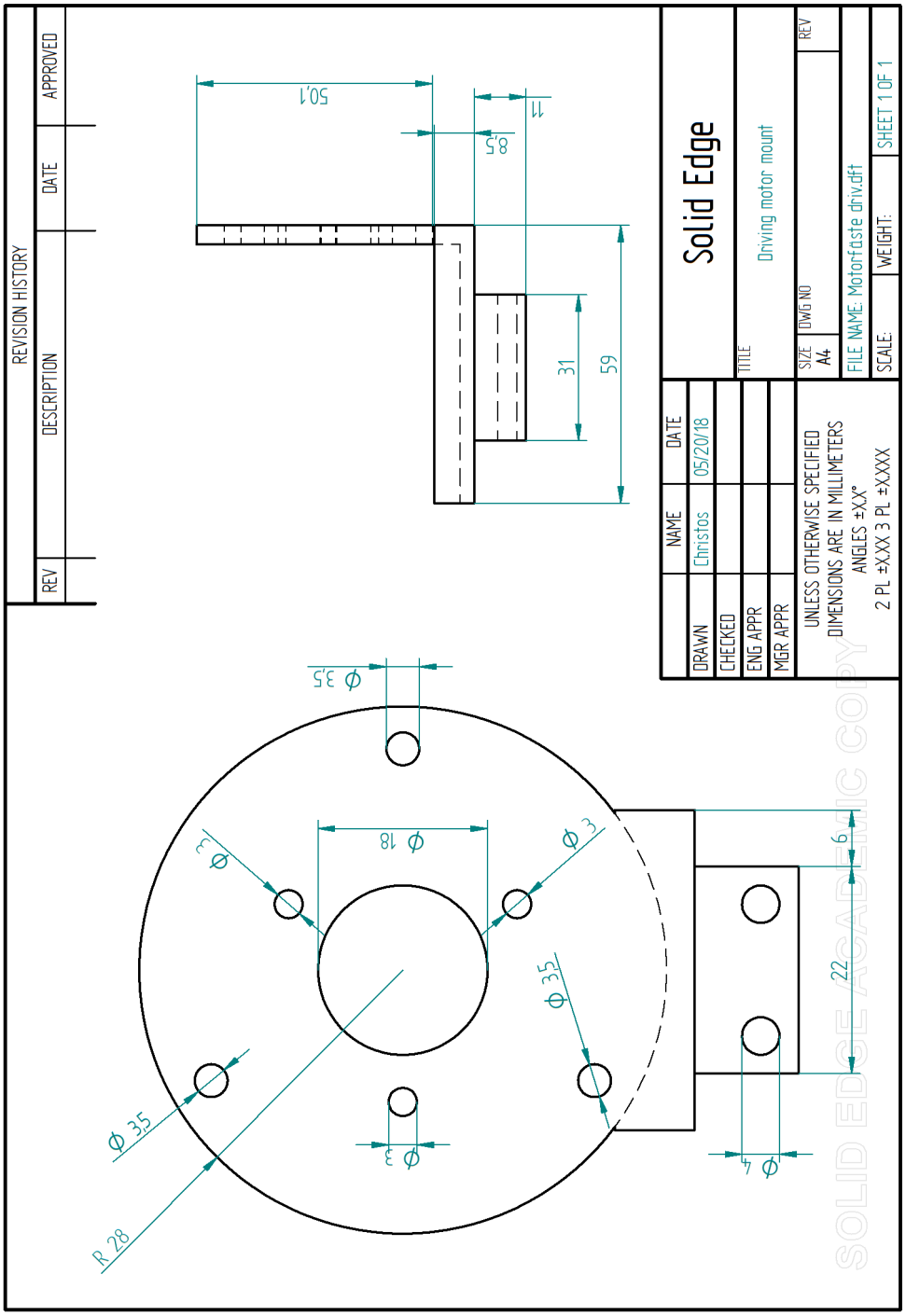
B.2 Rear frame



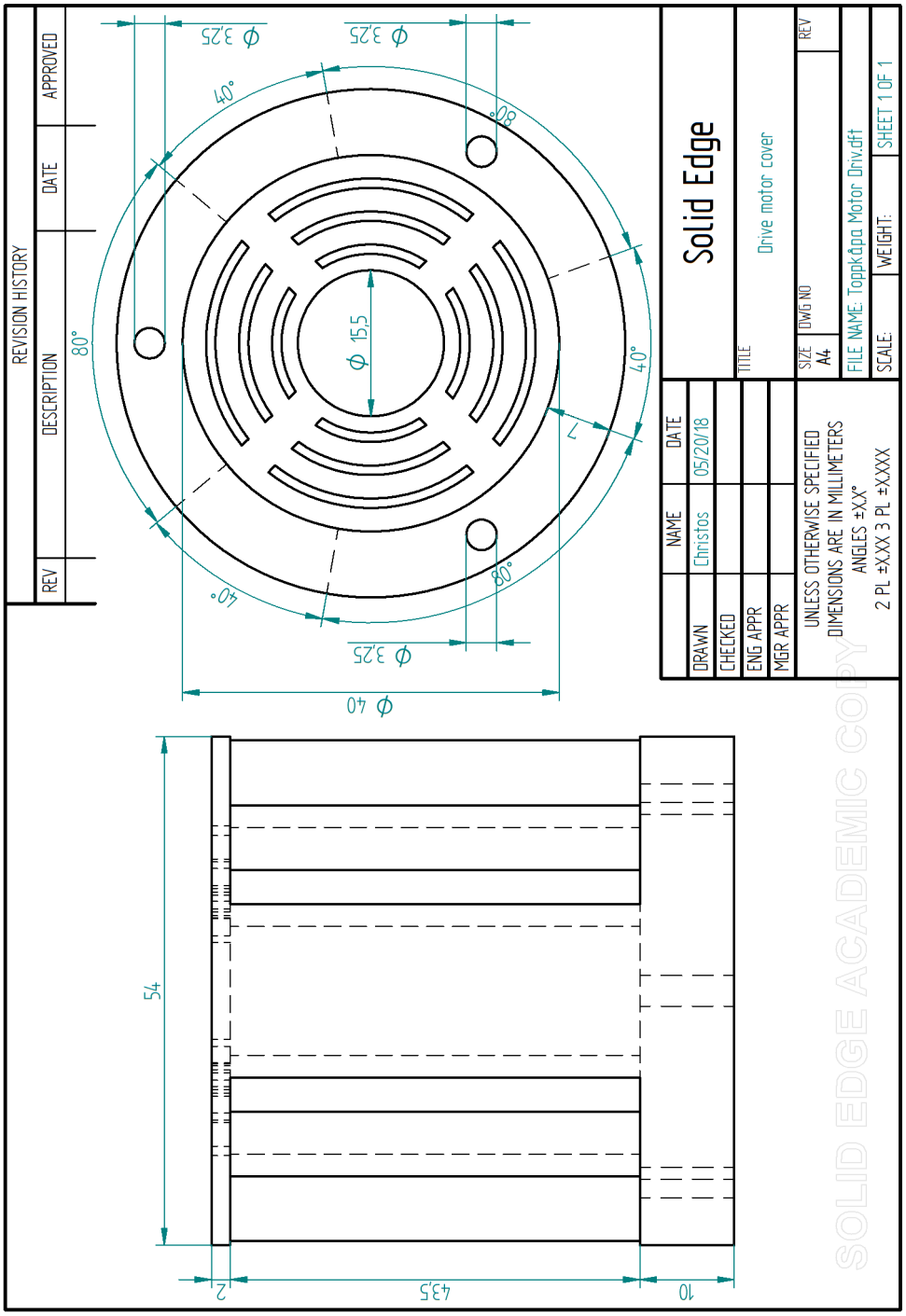
B.3 Fork


REVISION HISTORY
REV
DESCRIPTION
DATE
APPROVED
Solid Edge
Bicycle fork
TITLE
SIZE A4
DWG NO
REV
FILE NAME: GaffeL.dft
SCALE:
WEIGHT:
SHEET 1 OF 1
NAME
DATE
DRAWN
Christos
05/20/18
CHECKED
ENG APPR
MGR APPR
UNLESS OTHERWISE SPECIFIED
DIMENSIONS ARE IN MILLIMETERS
ANGLES ±X.X°
2 PL ±X.XX 3 PL ±X.XXX
28
12
R 2.65
15.5
7
10
8
22
17.5
15.5
5
56
10
95.01
1
SOLID EDGE ACADEMIC COPY

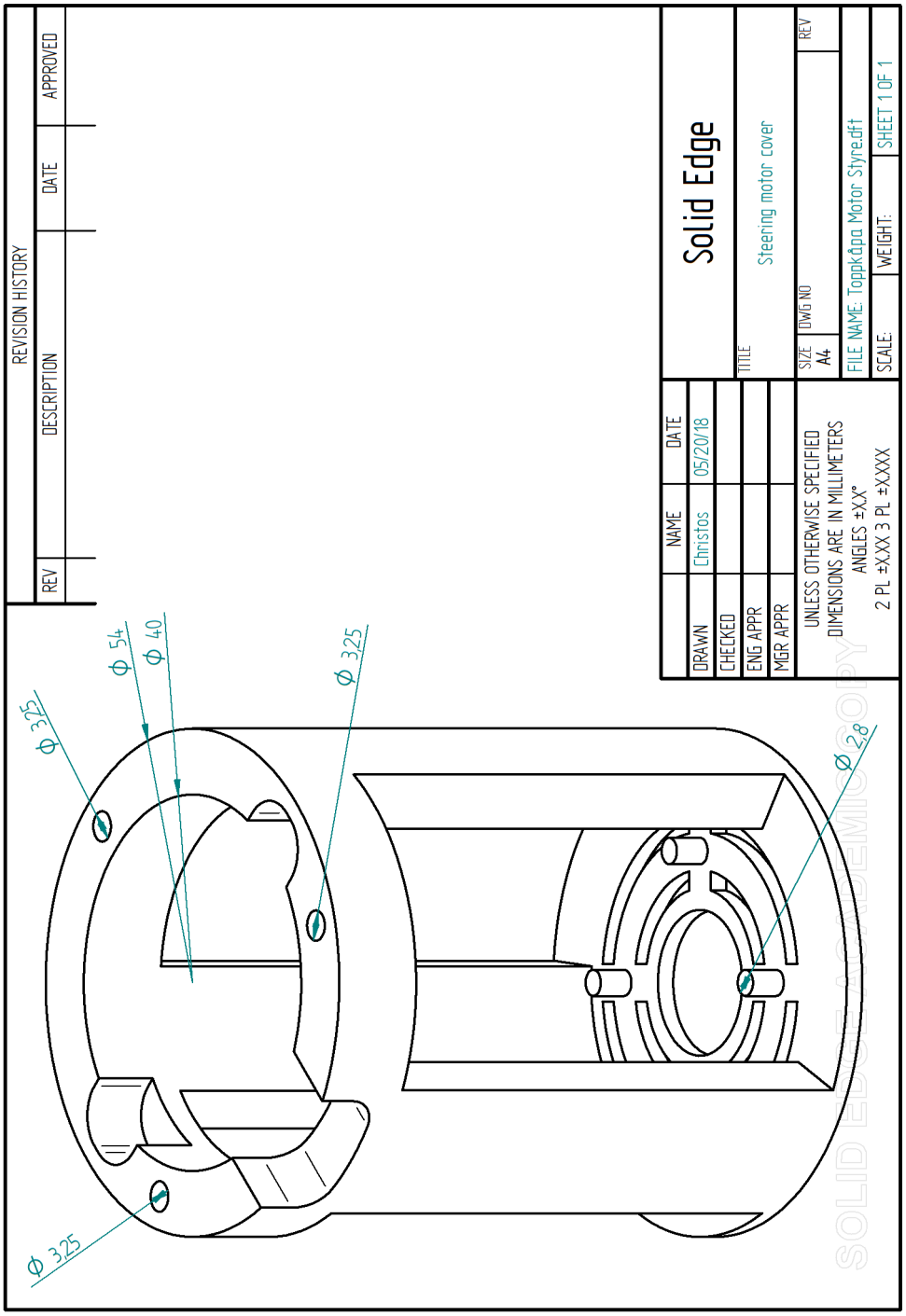
B.4 Drive motor mount



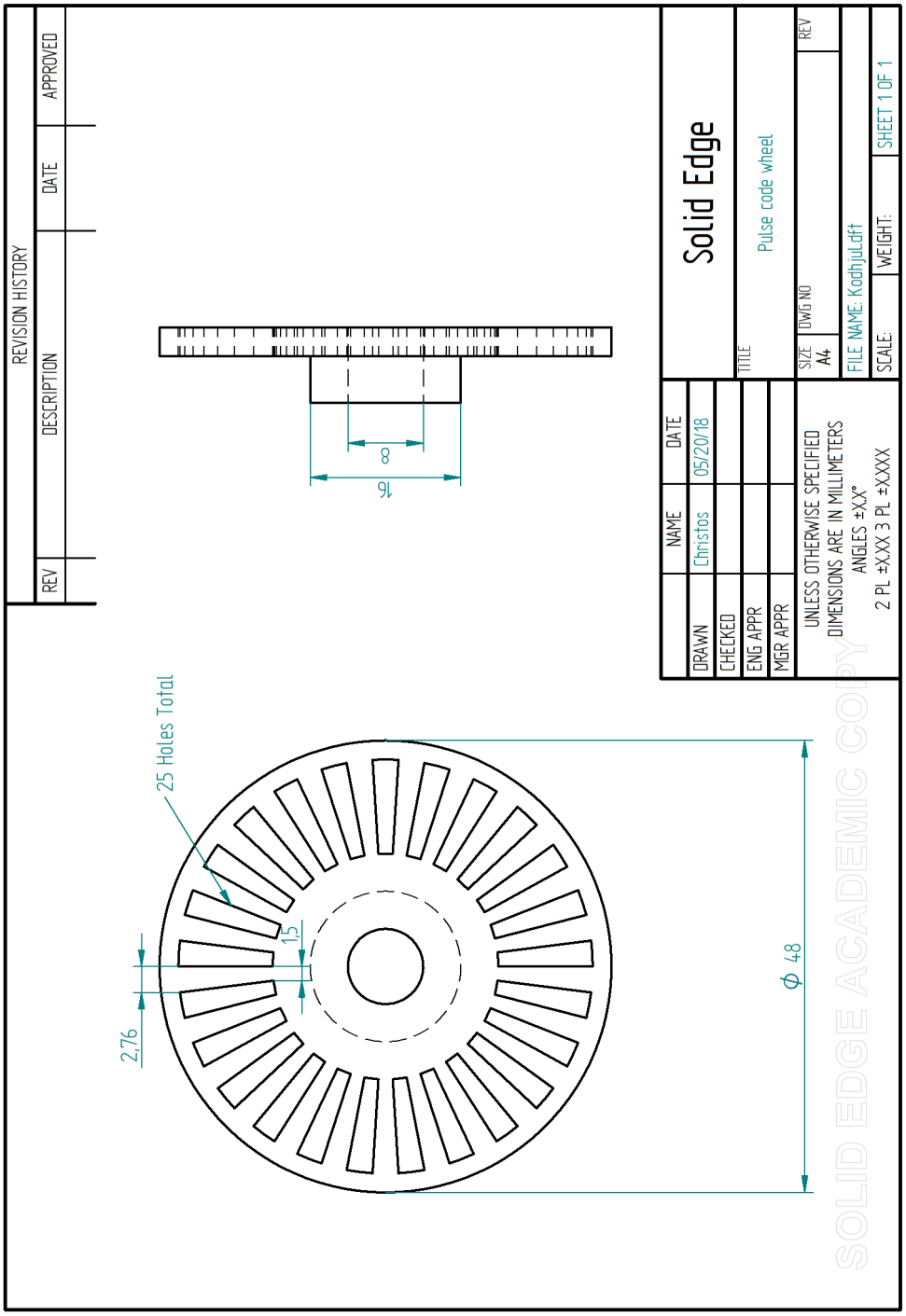
B.5 Drive motor cover



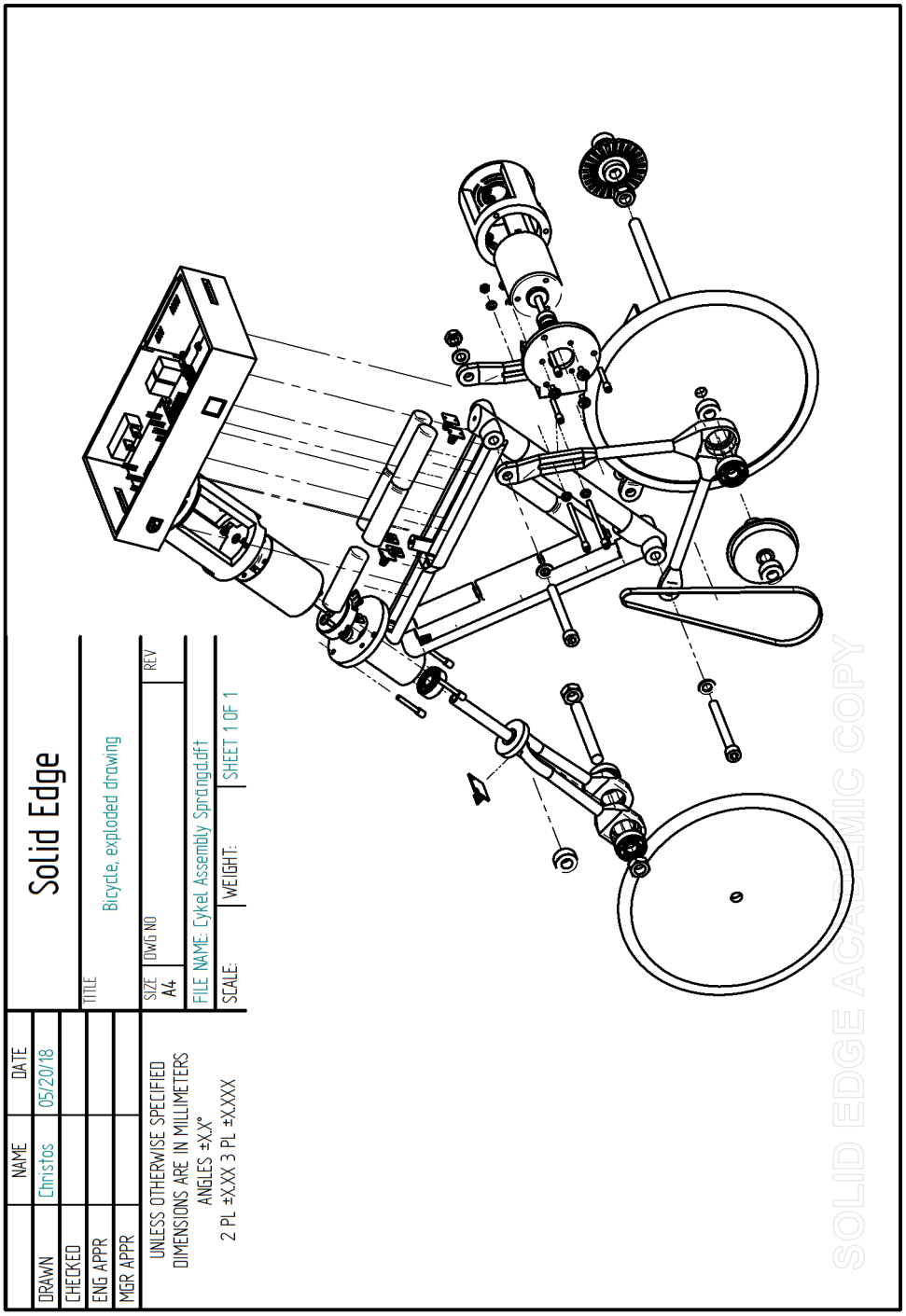
B.6 Steer motor cover



B.7 Pulse code wheel



B.8 Exploded bicycle assembly



Appendix C

MATLAB-code for simulation

```
%   MATLAB-code for simulating the bicycle system.
%
%   By: Arthur Grönlund & Christos Tolis
%   Course: MF133X degree project in mechatronics
%   University: KTH - Kungliga Tekniska Högskolan
%
%   Purpose: This program simulates the bicycle system
%            to test the performance of the controllers and to evaluate
%            component requirements. Also used to identify potential
%            problem areas needing improvement.
%
%            The program is split into three sections,
%            1 - Defines the physical properties of the bicycle and
%                electronics. Linearizes the differential equation
%                describing the balance of the system.
%            2 - Controller design (pole placements and parameters)
%            3 - Simulates the bicycle and presents the results.
%
%   Last change: 2018-05-25
%

clear;
clc;

% Bicycle constants
rho = 1.21;                             % Density of air
A_f = 0.3*0.1/2;                        % Frontal area of vehicle, approximation
C_d = 1.0;                              % Coefficient of drag, approximation
g = 9.82;                               % Gravitational acceleration
L = 0.314;                              % Distance between front and rear axle, measured in Solid Edge ST9
l_b = 0.1385;                           % Distance from CoG to rear axle, measured in Solid Edge ST9
l_f = L - l_b;                          % Distance from CoG to front axle
h_g = 0.1655;                           % Distance from ground to CoG, measured in Solid Edge ST9
r_d = 0.08;                             % Wheel radius
m_d = 0.119;                            % Mass of wheel, calculated by Solid Edge ST9
J_D = 0.000000060;                      % Moment of inertia of driving motor axle, calculated by Solid Edge ST9
%J_B = 1/2*m_d*r_d^2;                   % Moment of inertia of rear wheel axle, approximation
J_B = 0.00039;                          % Moment of inertia of rear wheel axle, calculated by Solid Edge ST9
J_F = J_B;                              % Moment of inertia of front wheel axle
%J_S = 1/12*m_d*r_d^2;                  % Moment of inertia, steer assembly, approximation
J_S = 0.000232;                         % Moment of inertia, steer assembly, calculated by Solid Edge ST9
m = 1.6;                                % Vehicle mass, measured
```

```
%J_L = 1/3*m*L^2;                       % Moment of inertia for leaning angle,
    approximation
J_L = 0.060;                            % Moment of inertia for leaning angle,
    calculated by Solid Edge ST9
my_d = 0.65;                             % Friction coefficient tyre/ground,
    approximated
i_d = 4;                                % Gearing ratio driving motor axle / rear axle
eta = 0.95;                             % Timing belt efficiency, approximated
i_s = 1;                                % Gearing ratio steering motor axle / steer
    assembly

% Electronic "constants"
U_battery = 1.2;                        % Voltage, one battery
R_batteri = 45e-3;                      % Resistance, one battery
s = 10;                                 % Batteries in series
p = 1;                                  % Batteries in parallel
U_bank = s*U_battery;                   % Total voltage of battery bank
R_bank = s*R_batteri;                   % Total resistance of battery bank
R_FET = 5e-3;                           % Resistance of FET transistor or H-bridge

% Driving motor Nidec DMN37KA
M_ND = 0.0245;                                          % Nominal torque of driving
    motor
I_ND = 1.2;                                             % Nominal current of driving
    motor
U_ND = 12;                                              % Nominal voltage of driving
    motor
omega_ND = 3600*2*pi/60;                                % Nominal angular speed of
    driving motor
k2phi_D = M_ND / I_ND;                                  % Motor constant of driving
    motor
omega_0D = U_ND / k2phi_D;                              % No load speed of driving motor
M_stallD = M_ND * (omega_0D / (omega_0D - omega_ND)) ;  % Stall torque of driving motor
I_stallD = M_stallD / k2phi_D;                          % Stall current of driving motor
R_AD = U_ND / I_stallD;                                 % Resistance of driving motor

% Steering motor Nidec DMN37KA
M_NS = 0.0245;                                          % Nominal torque of driving
    motor
I_NS = 1.2;                                             % Nominal current of driving
    motor
U_NS = 12;                                              % Nominal voltage of driving
    motor
omega_NS = 3600*2*pi/60;                                % Nominal angular speed of
    driving motor
k2phi_S = M_NS / I_NS;                                  % Motor constant of driving
    motor
omega_0S = U_NS / k2phi_S;                              % No load speed of driving motor
M_stallS = M_NS * (omega_0S / (omega_0S - omega_NS)) ;  % Stall torque of driving motor
I_stallS = M_stallS / k2phi_S;                          % Stall current of driving motor
R_AS = U_bank / I_stallS;                               % Resistance of driving motor

% Linearization of Alpha
[x, y] = meshgrid(-pi/2:pi/128:pi/2, -pi/2:pi/128:pi/2);

v_lin = 10/3.6;  % For plots
v_test = 10/3.6; % For plots

% Derived function of phi
f2 = @(theta, phi, v) m*g./J_L.*h_g.*sin(phi) - m.*sin(theta) ./ (J_L*sqrt(l_b^2 + cos(
    theta).^2*(L^2 - l_b^2))) .* v.^2 .* h_g .* cos(phi);
% Taylor expansion of phi, 1st order
T1 = @(theta, phi, v) (m*g*h_g/J_L .* phi - m.*v.^2.*h_g./(J_L*L) .* theta);

figure;
mesh(x*180/pi, y*180/pi, f2(x, y, v_test));
hold on;
mesh(x*180/pi, y*180/pi, T1(x, y, v_test));
title('d^2phi/dt^2 vs linearization');
xlabel('Theta');
ylabel('Phi');
zlabel('Angular acceleration of Phi');
colorbar;

```

```
figure;
abs_lin_error = abs(f2(x, y, v_test) - T1(x, y, v_test));
rel_lin_error = abs_lin_error ./ abs(f2(x, y, v_test)) * 100;
contour(x*180/pi, y*180/pi, rel_lin_error, 'Showtext', 'on', 'Levellist', [1 10 20 30 40
     50 60 70 80 90 100 150 200 250 300]);
title(['Relative linearization error at v = ' num2str(3.6*v_test) ' kph (%)']);
xlabel('Theta (deg)');
ylabel('Phi (deg)');
zlabel('Error');
grid on;

%% Controller stuff
s = tf('s');

% Cykel
A_cykel = [0,         1;
     m*g*h_g/(J_L), 0];
B_cykel = @(v) [0;
     -m*v^2*h_g / (J_L*L)];
C_cykel = [1 0];

PP_Im = 6;
PP_Re = -7;

L_cykel = @(v) [-J_L*L/(m*h_g*v^2) * (PP_Im^2 + PP_Re^2 + h_g*m*g/J_L), 2*J_L*L*PP_Re /
    (m*h_g*v^2)];
l0_cykel = @(v) -J_L*L / (m*h_g*v^2) * (PP_Im^2 + PP_Re^2);
sys_cykel_pp = @(v) ss(A_cykel - B_cykel(v)*L_cykel(v), B_cykel(v), C_cykel, 0) *
    l0_cykel(v);

v_vec = [5/3.6 7/3.6 9/3.6 10/3.6 15/3.6];

figure;
t_cykel = 0:0.01:15;

for n = 1:length(v_vec)
    [phi_cykel, t_cykel, x_cykel] = lsim(sys_cykel_pp(v_vec(n)), 5*pi/180*ones(1, length
        (t_cykel)), t_cykel);
    info_cykel = stepinfo(phi_cykel, t_cykel);

    thetaref_cykel = zeros(length(t_cykel), 1);
    for j = 1:length(thetaref_cykel)
        thetaref_cykel(j) = - L_cykel(v_vec(n)) * x_cykel(j, :)' + l0_cykel(v_vec(n));
    end

    subplot(2, 3, n);
    plot(t_cykel, phi_cykel * 180/pi);
    hold on;
    plot(t_cykel, thetaref_cykel * 180/pi);
    title(['Step response of bike system at v = ' num2str(v_vec(n) * 3.6) ' km/h']);
    grid on;
    xlabel('Time (s)');
    ylabel('Angle (degrees)');
    plot([info_cykel.SettlingTime info_cykel.SettlingTime], [info_cykel.SettlingMin
        info_cykel.SettlingMax], 'k');
    text(info_cykel.PeakTime, info_cykel.Peak * 1.05 * 180/pi, ['Overshoot: ' num2str(
        info_cykel.Overshoot, 4) ' %']);
    text(info_cykel.RiseTime, 0.5, ['Rise time: ' num2str(info_cykel.RiseTime, 4) ' s'])
        ;
    text(info_cykel.SettlingTime, info_cykel.SettlingMin * 180/pi, ['Settling time (2%):
         ' num2str(info_cykel.SettlingTime, 4) ' s']);
    legend('Phi (lean angle)', 'Requested theta (steer angle)', 'location', 'southeast')
        ;
end

subplot(2, 3, 6);
pzmap(sys_cykel_pp(v_vec(1)));
title('Pole placement of bike system');

% Motor, steering
A_motor_s = [0, 1;
             0, -i_s^2*k2phi_S^2/(J_S*R_AS)];
B_motor_s = [0;
             i_s*k2phi_S/(J_S*R_AS)];
```

```
C_motor_s = [1 0];
Im_s = 30;
Re_s = 40;
L_motor_s = place(A_motor_s, B_motor_s, [(-Re_s + 1i*Im_s) (-Re_s - 1i*Im_s)]);
sys_motor_s_pp0 = ss(A_motor_s - B_motor_s*L_motor_s, B_motor_s, C_motor_s, 0);
l0_motor_s = 1 / dcgain(sys_motor_s_pp0);
sys_motor_s_pp = sys_motor_s_pp0 * l0_motor_s;

[theta_motor_s, t_motor_s, x_motor_s] = step(sys_motor_s_pp);
u_motor_s = zeros(length(t_motor_s), 1);
for j = 1:length(u_motor_s)
    u_motor_s(j) = - L_motor_s * x_motor_s(j, :)' + l0_motor_s;
end

info_motor_s_pp = stepinfo(sys_motor_s_pp);

figure;
subplot(1, 2, 1);
plot(t_motor_s, theta_motor_s);
hold on;
plot(t_motor_s, u_motor_s);
title('Steering motor State space model');
grid on;
plot([info_motor_s_pp.SettlingTime info_motor_s_pp.SettlingTime], [info_motor_s_pp.
    SettlingMin info_motor_s_pp.SettlingMax], 'k');
text(info_motor_s_pp.PeakTime, info_motor_s_pp.Peak * 1.05, ['Overshoot: ' num2str(
    info_motor_s_pp.Overshoot, 4) ' %']);
text(info_motor_s_pp.RiseTime, 0.5, ['Rise time: ' num2str(info_motor_s_pp.RiseTime, 4)
    ' s']);
text(info_motor_s_pp.SettlingTime, info_motor_s_pp.SettlingMin, ['Settling time (2%): '
    num2str(info_motor_s_pp.SettlingTime, 4) ' s']);
legend('Theta', 'Motor voltage');

subplot(1, 2, 2);
pzmap(sys_motor_s_pp);
title('Steering motor State space model');


% Motor, drive force
Kp = 12;
Ki = 6;
Kd = 0;
G_D = k2phi_D * i_d * eta / ( (r_d*(m + (J_D*i_d^2*eta + J_B + J_F) / r_d^2)*(R_bank +
    R_AD + R_FET))*s + eta*k2phi_D^2*i_d^2/r_d);
F_D = Kp + Ki/s + Kd*s;
sys_motor_d = feedback(F_D*G_D, 1);
info_motor_d = stepinfo(sys_motor_d);

[gammaspeed_motor_d, t_motor_d] = step(sys_motor_d);
[u_motor_d, ~] = step(F_D / (1 + F_D*G_D), t_motor_d);

figure;
subplot(1, 2, 1);
plot(t_motor_d, gammaspeed_motor_d);
hold on;
plot(t_motor_d, u_motor_d);
title('Driving motor PID');
grid on;
plot([info_motor_d.SettlingTime info_motor_d.SettlingTime], [info_motor_d.SettlingMin
    info_motor_d.SettlingMax], 'k');
text(info_motor_d.PeakTime, info_motor_d.Peak * 1.05, ['Overshoot: ' num2str(
    info_motor_d.Overshoot, 4) ' %']);
text(info_motor_d.RiseTime, 0.5, ['Rise time: ' num2str(info_motor_d.RiseTime, 4) ' s'])
    ;
text(info_motor_d.SettlingTime, info_motor_d.SettlingMin, ['Settling time (2%): '
    num2str(info_motor_d.SettlingTime, 4) ' s']);
legend('Speed', 'Motor voltage');

subplot(1, 2, 2);
pzmap(sys_motor_d);
title('Driving motor PID');


%% Simulation
```

```
% Variables - initial values
U_AD = 0.0001;                      % Voltage over driving motor (if set to 0 things
    break....)
U_AS = 0;                           % Voltage over steering motor
theta_ref = 0;                      % Desired steering angle
FDist = 0;                          % Disturbing force
Cr = 0.004;                         % Rolling resistance (set to 0 to not simulate)
simulate_phi_acceleration = 1;      % Indicates whether we should simulate falling over
    or not
simulate_air_resistance = 1;        % Indicates whether we should simulate air
    resistance or not
simulate_drive_force = 1;           % Indicates whether we should simulate driving force
phi_ref = 0;                        % Desired lean angle
v_ref = 7/3.6;                      % Desired speed (m/s)
regulateSteering = 0;               % Don't regulate steering initially
timeAttainedSpeed = 9999999;        % Variable for logging the time that the bicycle
    attained a certain speed

% Functions that remain constant throughout simulation
r_g = @(theta) sqrt(l_b^2 + cos(theta).^2*(L^2 - l_b^2)) ./ sin(theta);         % Curve
    radius
F_D = @(U_AD, v) i_d*eta/r_d*(k2phi_D * (U_AD - k2phi_D*i_d*v/r_d) / (R_bank + R_AD +
    R_FET));  % Driving force
F_B = 0;                                                                        %
    Braking force
F_L = @(v) 1/2*rho*A_f*C_d*v^2;                                                 % Air
    resistance
N_B = @(U_AD, v) l_f/L*m*g + h_g/L*F_D(U_AD, v);                                % Normal
     force, rear tire
N_F = @(U_AD, v) l_b/L*m*g - h_g/L*F_D(U_AD, v);                                % Normal
     force, front tire
F_Rb = @(U_AD, v) sign(v)*Cr*N_F(U_AD, v);                                      %
    Rolling resistance, rear tire
F_Rf = @(U_AD, v) sign(v)*Cr*N_B(U_AD, v);                                      %
    Rolling resistance, front tire
F_C = @(v, theta) -m*v^2/r_g(theta);                                            %
    Centrifugal force
F_G = m*g;                                                                      %
    Gravitational force

% Runge-Kutta-4 setup
h = 0.015;                                      % Step length Runge-Kutta 4
t = 0:h:8;                                     % Simulation time vector
s0 = 0;                                         % Initial position
v0 = 7/3.6;                                     % Initial speed
phi0 = 0;                                       % Initial lean angle
omega_phi0 = 0;                                 % Initial lean rotational speed
theta0 = 0;                                     % Initial steer angle
omega_theta0 = 0;                               % Initial steer rotational speed

U = zeros(6, length(t));                        % Allocate memory for the solution
U(:, 1) = [s0; v0; phi0; omega_phi0; theta0; omega_theta0;];

% Allocate vectors that store information about the simulation
Acceleration = zeros(1, length(t));
Speed = zeros(1, length(t));
PhiSpeed = zeros(1, length(t));
Gamma = zeros(1, length(t));
GammaSpeed = zeros(1, length(t));
Distance = zeros(1, length(t));
Phi = zeros(1, length(t));
PhiAcceleration = zeros(1, length(t));
Theta = zeros(1, length(t));
ThetaSpeed = zeros(1, length(t));
ThetaAcceleration = zeros(1, length(t));
LinearizationError = zeros(1, length(t));
XPos = zeros(1, length(t));
YPos = zeros(1, length(t));
Beta = zeros(1, length(t));
ForceDriving = zeros(1, length(t));
ForceBraking = zeros(1, length(t));
ForceDisturb = zeros(1, length(t));
ForceAirResist = zeros(1, length(t));
ForceRollingResistFront = zeros(1, length(t));
```

```
292 ForceRollingResistRear = zeros(1, length(t));
293 ForceFrictionFront = zeros(1, length(t));
294 ForceFrictionRear = zeros(1, length(t));
295 FrictionLimitFront = zeros(1, length(t));
296 FrictionLimitRear = zeros(1, length(t));
297 VoltageSteeringMotor = zeros(1, length(t));
298 CurrentSteeringMotor = zeros(1, length(t));
299 TorqueSteeringMotor = zeros(1, length(t));
300 PowerSteeringMotor = zeros(1, length(t));
301 VoltageDrivingMotor = zeros(1, length(t));
302 CurrentDrivingMotor = zeros(1, length(t));
303 TorqueDrivingMotorAxle = zeros(1, length(t));
304 TorqueRearAxle = zeros(1, length(t));
305 PowerDrivingMotor = zeros(1, length(t));
306 ReferenceTheta = zeros(1, length(t));
307 ReferencePhi = zeros(1, length(t));
308 ReferenceSpeed = zeros(1, length(t));
309 SpeedError = zeros(1, length(t));
310 GammaIntegralError = zeros(1, length(t));
311
312 for j = 2:length(t)
313     % Update the functions, in case some parameters have changed
314     F_ff = @(U_A, v, theta) l_b/L * abs(F_C(v, theta) - FDist);
315     F_fb = @(U_A, v, theta) sqrt(l_f/L * (F_C(v, theta) - FDist)^2 + F_D(U_A, v)^2);
316     acceleration = @(U_AD, v) (simulate_drive_force*F_D(U_AD, v) -
          simulate_air_resistance*F_L(v) - F_Rf(U_AD, v) - F_Rb(U_AD, v)) / ...
317                 (m + (J_D*i_d^2*eta + J_B + J_F) / r_d^2);
318     phi_acceleration = @(v, phi, theta) simulate_phi_acceleration*(m*g./J_L.*h_g*sin(phi
          ) - m.*v^2.*h_g./(J_L*r_g(theta)) .* cos(phi) + FDist/J_L);
319
320     theta_acceleration = @(U_A, omega_theta) i_s*k2phi_S / (J_S*R_AS) * U_A - i_s^2*
          k2phi_S^2/(J_S*R_AS)*omega_theta;
321
322     F = @(u) [u(2);                                  % dy/dt
323               acceleration(U_AD, u(2));              % d2y/dt2
324               u(4);                                  % dphi/dt
325               phi_acceleration(u(2), u(3), u(5));    % d2phi/dt2
326               u(6);                                  % dtheta/dt
327               theta_acceleration(U_AS, u(6));];      % d2theta/dt2
328
329     % The Runge-Kutta-4 iteration
330     un = U(:, (j - 1));
331     k1 = F(un);
332     k2 = F(un + h/2*k1);
333     k3 = F(un + h/2*k2);
334     k4 = F(un + h*k3);
335
336     U(:, j) = un + h/6 * (k1 + 2*k2 + 2*k3 + k4);
337
338     % Collect information
339     Distance(j) = U(1, j);
340     Speed(j) = U(2, j);
341     Acceleration(j) = acceleration(U_AD, Speed(j));
342     Gamma(j) = U(1, j) * i_d / r_d;
343     GammaSpeed(j) = U(2, j) * i_d / r_d;
344     Phi(j) = U(3, j);
345     Theta(j) = U(5, j);
346     PhiSpeed(j) = U(4, j);
347     ThetaSpeed(j) = U(6, j);
348     PhiAcceleration(j) = phi_acceleration(Speed(j), Phi(j), Theta(j));
349     ThetaAcceleration(j) = theta_acceleration(U_AS, U(6, j));
350     LinearizationError(j) = abs(f2(Theta(j), Phi(j), Speed(j)) - T1(Theta(j), Phi(j),
          Speed(j))) / abs(f2(Theta(j), Phi(j), Speed(j))) * 100;
351     ForceDriving(j) = F_D(U_AD, Speed(j));
352     ForceBraking(j) = - F_B;
353     ForceDisturb(j) = FDist;
354     ForceAirResist(j) = simulate_air_resistance * F_L(Speed(j));
355     ForceRollingResistFront(j) = F_Rf(U_AD, Speed(j));
356     ForceRollingResistRear(j) = F_Rb(U_AD, Speed(j));
357     Beta(j) = Beta(j - 1) + (Distance(j) - Distance(j - 1)) / (sqrt(l_b^2 + cos(Theta(j)
          ).^2*(L^2 - l_b^2)) ./ sin(Theta(j)));
358     XPos(j) = XPos(j - 1) + (Distance(j) - Distance(j - 1))*sin(Beta(j));
359     YPos(j) = YPos(j - 1) + (Distance(j) - Distance(j - 1))*cos(Beta(j));
360     ForceFrictionFront(j) = F_ff(U_AD, Speed(j), Theta(j));
```

```
361     ForceFrictionRear(j) = F_fb(U_AD, Speed(j), Theta(j));
362     FrictionLimitFront(j) = N_F(U_AD, Speed(j))*my_d;
363     FrictionLimitRear(j) = N_B(U_AD, Speed(j))*my_d;
364     VoltageSteeringMotor(j) = U_AS;
365     TorqueSteeringMotor(j) = i_s*k2phi_S/R_AS * (U_AS - k2phi_S*ThetaSpeed(j));
366     CurrentSteeringMotor(j) = TorqueSteeringMotor(j) / k2phi_S;
367     PowerSteeringMotor(j) = TorqueSteeringMotor(j) * ThetaSpeed(j);
368     VoltageDrivingMotor(j) = U_AD;
369     TorqueRearAxle(j) = i_d*k2phi_D/R_AD * (U_AD - k2phi_D*GammaSpeed(j));
370     TorqueDrivingMotorAxle(j) = k2phi_D/R_AD * (U_AD - k2phi_D*GammaSpeed(j));
371     CurrentDrivingMotor(j) = TorqueDrivingMotorAxle(j) / k2phi_D;
372     PowerDrivingMotor(j) = TorqueDrivingMotorAxle(j) * GammaSpeed(j);
373     ReferenceSpeed(j) = v_ref;
374     ReferencePhi(j) = phi_ref;
375     ReferenceTheta(j) = theta_ref;
376     SpeedError(j) = v_ref - Speed(j);
377
378     % Simulation logic begins here
379     if (abs(Phi(j)) > pi/2)     % Vehicle fell over
380         FD = 0;                 % There is no longer a driving force
381         Cr = 1;                 % Increase the rolling resistance so the vehicle comes
                to a halt
382         simulate_phi_acceleration = 0;     % Stop simulating the leaning part
383         U(4, j) = 0;            % Alpha speed is now zero
384     end
385
386     % Check if friction was lost
387     %if (ForceFrictionRear(j) > N_B(U_AD, Speed(j))*my_d)       % Rear wheel traction
          was lost
388     %    break;
389     %end
390     %if (ForceFrictionFront(j) > N_F(U_AD, Speed(j))*my_d)      % Front wheel traction
          was lost
391     %    break;
392     %end
393
394     % Automatic control begins here
395     delta_t = (t(j) - t(j - 1));
396
397     % Balancing regulator
398     theta_ref = - L_cykel(Speed(j))*[Phi(j); PhiSpeed(j)] + l0_cykel(Speed(j)) * phi_ref
          ;
399     if theta_ref > 60*pi/180       % Dont request steering angles higher than a set
          value
400         theta_ref = 60*pi/180;
401     elseif theta_ref < -60*pi/180
402         theta_ref = -60*pi/180;
403     end
404
405     % Steer motor regulator
406     U_AS = - L_motor_s*[Theta(j); ThetaSpeed(j)] + l0_motor_s * theta_ref;
407
408     % Speed regulator (cruise control)
409     U_AD = Kp*SpeedError(j) + Ki*delta_t*sum(SpeedError) + Kd*(SpeedError(j) -
          SpeedError(j - 1)) / delta_t;
410
411     if U_AS > 12        % Maximum voltage available for motors is U_bank
412         U_AS = 12;
413     elseif U_AS < -12
414         U_AS = -12;
415     end
416     if U_AD > U_bank
417         U_AD = U_bank;
418     elseif U_AD < 0
419         U_AD = 0;
420     end
421
422     % Change some parameters over time
423     if (t(j) > 1)
424         phi_ref = -15*pi/180;
425     end
426
427     if (t(j) > 6)
428         phi_ref = 15*pi/180;
```

```
429     end
430
431     if (t(j) > 3)
432         v_ref = 12/3.6;
433     end
434
435     % Introduce some disturbances
436     if (t(j) > 2 && t(j) < 5)
437         FDist = -1 + 0.5*rand();
438     elseif (t(j) > 5)
439         FDist = 0.5 + rand();
440     else
441         FDist = 0;
442     end
443
444 end
445
446 % Plot results
447 figure;
448 subplot(3, 4, 1);
449 plot(t, Acceleration);
450 title('Acceleration');
451 ylabel('Acceleration (m/s^2)');
452 xlabel('Time (s)');
453 grid on;
454
455 subplot(3, 4, 2);
456 plot(t, Speed * 3.6);
457 hold on;
458 plot(t, ReferenceSpeed * 3.6);
459 title('Speed');
460 xlabel('Time (s)');
461 ylabel('Speed (km/h)');
462 legend('Speed', 'Reference speed', 'location', 'southeast');
463 grid on;
464
465 subplot(3, 4, 3);
466 line(XPos, YPos, t);
467 title('Position');
468 xlabel('x (m)');
469 ylabel('y (m)');
470 grid on;
471 axis equal;
472
473 subplot(3, 4, 4);
474 plot(t, LinearizationError);
475 title('Linearization error');
476 xlabel('Time (s)');
477 ylabel('Relative error (%)');
478 grid on;
479
480 subplot(3, 4, [5 6 7 8]);
481 plot(t, Phi*180/pi);
482 hold on;
483 plot(t, Theta*180/pi);
484 plot(t, ReferenceTheta*180/pi);
485 plot(t, ReferencePhi*180/pi);
486 axis([t(1) t(end) -55 45]);
487 title('Angles');
488 xlabel('Time (s)');
489 ylabel('Angles (deg)');
490 grid on;
491 legend('Lean angle', 'Steer angle', 'Reference steer angle', 'Reference lean angle', '
      location', 'southeast');
492
493 subplot(3, 4, [9 10 11 12]);
494 %plot(t, ForceDriving);
495 hold on;
496 %plot(t, ForceBraking);
497 plot(t, ForceDisturb, 'Color', [0 0.7 0]);
498 %plot(t, ForceAirResist);
499 plot(t, ForceFrictionFront, 'Color', [0 0.3 0.7]);
500 plot(t, ForceFrictionRear, 'Color', [0.7 0 0]);
501 plot(t, FrictionLimitFront, 'Color', [0 0.3 1]);
```

```
plot(t, FrictionLimitRear, 'Color', [1 0 0]);
title('Forces');
xlabel('Time (s)');
ylabel('Force (N)');
grid on;
legend( ... %'Driving force', 'Braking force',
       'Disturbance', ...
       'Friction front tire', 'Friction rear tire', ...
       'Friction limit front', 'Friction limit rear');


% Steering motor info
figure;
subplot(3, 3, 1);
plot(t, VoltageSteeringMotor);
title('Voltage, Steering Motor');
xlabel('Time (s)');
ylabel('Voltage (V)');
grid on;

subplot(3, 3, 4);
plot(t, CurrentSteeringMotor);
title('Current, Steering Motor');
xlabel('Time (s)');
ylabel('Current (A)');
grid on;

subplot(3, 3, 7);
plot(t, VoltageSteeringMotor .* CurrentSteeringMotor);
title('Electrical Power, Steering Motor');
xlabel('Time (s)');
ylabel('Power (W)');
grid on;

subplot(3, 3, 2);
plot(t, TorqueSteeringMotor);
title('Torque, Steering Axle');
xlabel('Time (s)');
ylabel('Torque (Nm)');
grid on;


subplot(3, 3, 5);
plot(t, ThetaSpeed * 60 / (2*pi));
title('Angular speed, Steering Axle');
xlabel('Time (s)');
ylabel('Angular speed (rpm)');
grid on;

subplot(3, 3, 8);
plot(t, PowerSteeringMotor);
title('Power, Steering Axle');
xlabel('Time (s)');
ylabel('Power (W)');
grid on;

% Driving motor info
figure;
subplot(3, 3, 1);
plot(t, VoltageDrivingMotor);
title('Voltage, Driving Motor');
xlabel('Time (s)');
ylabel('Voltage (V)');
grid on;

subplot(3, 3, 4);
plot(t, CurrentDrivingMotor);
title('Current, Driving Motor');
xlabel('Time (s)');
ylabel('Current (A)');
grid on;

subplot(3, 3, 7);
plot(t, VoltageDrivingMotor .* CurrentDrivingMotor);
```

```
576 title('Electrical Power, Driving Motor');
577 xlabel('Time (s)');
578 ylabel('Power (W)');
579 grid on;
580
581 subplot(3, 3, 5);
582 plot(t, GammaSpeed * 60/(2*pi));
583 title('Angular speed, Driving Motor Axle');
584 xlabel('Time (s)');
585 ylabel('Angular speed (rpm)');
586 grid on;
587
588 subplot(3, 3, 8);
589 plot(t, PowerDrivingMotor);
590 title('Power, Driving Motor Axle');
591 xlabel('Time (s)');
592 ylabel('Power (W)');
593 grid on;
594
595 subplot(3, 3, 2);
596 plot(t, TorqueDrivingMotorAxle);
597 title('Torque, Driving Motor Axle');
598 xlabel('Time (s)');
599 ylabel('Torque (Nm)');
600 grid on;
601
602 subplot(3, 3, 3);
603 plot(t, TorqueRearAxle);
604 title('Torque, Rear Axle');
605 xlabel('Time (s)');
606 ylabel('Torque (Nm)');
607 grid on;
608
609 subplot(3, 3, 6);
610 plot(t, GammaSpeed * 60/(2*pi) / i_d);
611 title('Angular speed, Rear Axle');
612 xlabel('Time (s)');
613 ylabel('Angular speed (rpm)');
614 grid on;
615
616 subplot(3, 3, 9);
617 plot(t,  PowerDrivingMotor ./ (VoltageDrivingMotor .* CurrentDrivingMotor) * 100);
618 title('Efficiency, driving motor');
619 xlabel('Time (s)');
620 ylabel('Efficiency (%)');
621 grid on;
622
623 % Figure that match experiment data figures
624 figure;
625 subplot(2, 1, 1);
626 plot(t, Phi*180/pi, 'b');
627 hold on;
628 plot(t, ReferencePhi*180/pi, 'k');
629 plot(t, Theta*180/pi, 'Color', [0 0.7 0]);
630 plot(t, ReferenceTheta*180/pi, 'r');
631 grid on;
632 legend('Lean angle', 'Lean reference', 'Steer angle', 'Steer reference', 'location', '
      southwest');
633 xlabel('Time (s)');
634 ylabel('Angles (deg)');
635 title('Angles');
636
637 sub2 = subplot(2, 1, 2);
638 plot(t, Speed*3.6, 'b');
639 hold on;
640 plot(t, ReferenceSpeed*3.6, 'r');
641 grid on;
642 xlabel('Time (s)');
643 ylabel('Speed (km/h)');
644 yyaxis right;
645 plot(t, VoltageDrivingMotor, 'm');
646 ylabel('Voltage (V)');
647 legend('Speed', 'Speed reference', 'Drive motor voltage', 'location', 'southeast');
648 title('Speed');
```

```
649 sub2.YAxis(2).Color = [1 0 1];
```

Appendix D

Arduino C code

```
1  /* Arduino Uno C code for the bicycle
2   * Version 3
3   *
4   * By: Arthur Grönlund & Christos Tolis
5   * Course: MF133X degree project in mechatronics
6   * University: KTH - Kungliga Tekniska Högskolan
7   *
8   * Purpose: This program controls the bicycle unit and implements
9   *          a linearized, time-variant state space controller for automatic
10  *          balancing of the bicycle prototype. Also implements automatic control
11  *          for the two DC motors.
12  *
13  *          When started, red LED will flash every second to indicate that the program
        is ready.
14  *          Pressing the right button will start the bicycle. The left button (or
        lifting the bicycle)
15  *          will shut off the program.
16  *
17  * Hardware used: Arduino Uno Rev3
18  *                2 x MPU6050 accelerometer&gyro sensors
19  *                1 x Angle sensor (Linear potentiometer)
20  *                1 x custom built pulse code wheel with an optical sensor
21  *                2 x VNH3SP30 motor drivers
22  *                1 x SD-card module connected via ICSP pins
23  *
24  * Version history: version 1 - included usage of all the LED's and allowed selection of
25  *                              different "programs" for different bicycle behaivours.
26  *                  version 2 - added SD card module for logging data. This meant that 4
        of
27  *                              the LED's could not be used and therefore program
        selection
28  *                              had to be removed. Code now consists of two states: "on"
        or "off".
29  *                  version 3 - Optimized code for speed. Loop time cut from 15-25 ms to
        6-10 ms.
30  *                              Switched SD library from SD.h to SdFat.h
31  *
32  * Last change: 2018-05-25
33  *
34  * Compiled with gcc -O2 flag. O3 makes the program slightly too large.
35  *
36  */
37
38 #include "PinChangeInt.h"                // This library helped against encoder bounces
39 #include "Wire.h"                        // For I2C bus communication
40 #include "I2Cdev.h"                      // Used by MPU6050.h
41 #include "MPU6050.h"                     // Accelerometer/Gyro library
42 #include "Filters.h"                     // Library for some generic filters such as
        low/high pass
43 #include <SPI.h>                         // Enables SPI communication, used by SdFat.h
```

```
44 #include "SdFat.h"                          // SD card library, this is slightly lighter
       than Arduino's SD.h
45
46 // PIN definitions
47 #define LED_WHITE 13                       // Unused due to SD card module using SPI
48 #define LED_BLUE 12                        // Unused due to SD card module using SPI
49 #define LED_GREEN 11                       // Unused due to SD card module using SPI
50 #define LED_YELLOW 10                      // Unused due to SD card module using SPI
51 #define LED_RED 9
52 #define MOTOR_STEER_B 8
53 #define MOTOR_STEER_A 7
54 #define MOTOR_STEER_PWM 6
55 #define MOTOR_DRIVE_B 4
56 #define MOTOR_DRIVE_A 5
57 #define MOTOR_DRIVE_PWM 3
58 #define SPEED_SENSOR_INTERRUPT 2
59 #define BUTTON_OK A1
60 #define BUTTON_CANCEL A0
61 #define SD_CARD_CS A2
62 #define STEER_ANGLE_SENSOR A3
63
64 // Unit states
65 #define STATE_STARTUP 1
66 #define STATE_PROGRAM_RED 2
67
68 // Bicycle constants
69 #define g 9.82                                 // Gravitational acceleration
70 #define L 0.314                                // Distance between front and rear axle,
       measured in Solid Edge ST9
71 #define l_b 0.1385                             // Distance from CoG to rear axle, measured
        in Solid Edge ST9
72 #define l_f 0.1755                             // Distance from CoG to front axle
73 #define h_g 0.1655                             // Distance from ground to CoG, measured in
        Solid Edge ST9
74 #define r_d 0.08                               // Wheel radius
75 #define m_d 0.119                              // Mass of wheel, calculated by Solid Edge
       ST9
76 #define J_D 0.000000060                        // Moment of inertia of driving motor axle,
        calculated by Solid Edge ST9
77 #define J_B 0.00039                            // Moment of inertia of rear wheel axle,
       calculated by Solid Edge ST9
78 #define J_F J_B                                // Moment of inertia of front wheel axle
79 #define J_S 0.000232                           // Moment of inertia, steer assembly,
       calculated by Solid Edge ST9
80 #define m 1.6                                  // Vehicle mass, measured
81 #define J_L 0.060                              // Moment of inertia for leaning angle,
       calculated by Solid Edge ST9
82
83 // Electronic "constants"
84 #define U_bank 12.0                            // Total voltage of battery bank
85
86 // Controller constants
87 // Balancing controller
88 #define PP_Im 6.0                              // Pole placement of balancing controller:
       Imaginary part
89 #define PP_Re -7.0                             // Pole placement of balancing controller:
       Real part
90
91 // These are constants in the feedback vector for the balancing controller,
92 // calculate these here so that they don't have to be calculated every loop.
93 float L_cykel1_factor = -J_L*L / (m*h_g) * (PP_Im*PP_Im + PP_Re*PP_Re + h_g*m*g/J_L);
94 float L_cykel2_factor = 2*J_L*L*PP_Re / (m*h_g);
95 float l0_cykel_factor = -J_L*L / (m*h_g) * (PP_Im*PP_Im + PP_Re*PP_Re);
96
97 // Speed controller
98 #define Kp 12                                  // Speed controller: Proportional constant
99 #define Ki 6                                   // Speed controller: Integral constant
100
101 // Steer motor controller
102 //#define  L_motor_s1 20.3739                  // Steer motor controller - Pole placement:
        Im 10, Re 20
103 //#define  L_motor_s2 1.6095                   // Steer motor controller - Pole placement:
        Im 10, Re 20
```

```
104 //#define L_motor_s1 45.841166435183943      // Steer motor controller - Pole placement:
        Im 15, Re 30
105 //#define L_motor_s2 2.424445543209810       // Steer motor controller - Pole placement:
        Im 15, Re 30
106 //#define L_motor_s1 75.3833                 // Steer motor controller - Pole placement:
        Im 25, Re 35
107 //#define L_motor_s2 2.8319                  // Steer motor controller - Pole placement:
        Im 25, Re 35
108 #define L_motor_s1 101.8692587448532         // Steer motor controller - Pole placement:
        Im 30, Re 40
109 #define L_motor_s2 3.2393996131686           // Steer motor controller - Pole placement:
        Im 30, Re 40
110 #define l0_motor_s L_motor_s1                // Static gain
111
112 // Other constants
113 #define lean_gyro_percentage 0.98            // Complementary filter setting,
      accelerometer contribution factor will be 1 - lean_gyro_percentage
114
115 // Data logging (SD card) objects
116 SdFat sd;
117 SdFile dataFile;
118
119 // Buffers for converting numerical values to strings when logging to file
120 char vBuffer[9];
121 char v_refBuffer[9];
122 char steerAngleBuffer[9];
123 char steerAngle_refBuffer[9];
124 char U_ASBuffer[9];
125 char U_ADBuffer[9];
126 char leanAngleBuffer[9];
127 char leanAngle_refBuffer[9];
128 char ssxBuffer[9];
129 char lsyBuffer[9];
130 char laxBuffer[9];
131 char lazBuffer[9];
132 char dtBuffer[9];
133
134 // State variables, initial values
135 float leanAngle = 0;                         // Lean angle
136 float steerAngle = 0;                        // Steer angle
137 volatile float v = 0;                        // Bicycle speed. Needs to be volatile
      since it can change unexpectedly due to interrupts, otherwise compiler optimizations
       might make wrong assumptions about this variable.
138 float U_AS = 0;                             // Steer motor voltage
139 float U_AD = 0;                             // Drive motor voltage
140 float speedErrorHistory = 0;                // Accumulative. If this gets very large
      the PI-controller will get sluggish, and this might need clearing. (Aka integral
      windup)
141 int state = STATE_STARTUP;                  // Current state of the program
142 bool okPressed = false;                     // True when OK button is pressed
143 bool cancelPressed = false;                 // True when Cancel button is pressed
144 bool regulateSteering = false;              // Determines if steering should be
      regulated or not
145
146 // References, initial values
147 float leanAngle_ref = 0;                    // Desired lean angle
148 float v_ref = 7/3.6;                        // Desired bicycle speed
149 float steerAngle_ref = 0;                   // Desired steer angle
150
151 // Sensor objects
152 MPU6050 leanSensor(0x69);                   // Lean angle sensor
153 MPU6050 steerSensor(0x68);                  // Steer angle sensor, has AD0 set to HIGH.
154
155 // Filter objects, second argument is cutoff frequency in Hz for one pole filters.
156 FilterOnePole steerRefFilter(LOWPASS, 5);       // Filter the steer ref angle cause of
      the noise in the gyro data. This will cause a delay in the regulator and the cutoff
      should therefore not be set too low.
157 FilterOnePole liftDetectionFilter(LOWPASS, 2);  // Filter for the lift detection code so
       the unit doesn't shut off due to simple vibrations.
158 FilterOnePole driveMeasureFilter(LOWPASS, 5);   // Smooths out speed measurements a bit.
159
160
161 // Raw measurement variables from sensors
162 int lean_acc_x, lean_acc_z;
```

```
163 int lean_speed_y;
164 int steer_speed_x;
165
166 // Calibration "constants" - determined at startup
167 int calib_lean_speed_y;
168 int calib_steer_speed_x;
169 int calib_lean_acc_x, calib_lean_acc_z;
170
171 // SI unit variables for acceleration and angular velocity
172 float lax, laz, lsy, ssx;
173
174 // Other
175 unsigned long lastTimeCodeWheelPass = 0;              // Time of last interrupt signal
        from speed sensor.
176 unsigned long timeAttainedSpeed = 0;                  // Variable for holding the time
        that bicycle attained a certain speed
177 unsigned long lastLoopTime = 0;                       // Time when last dt (change in
        time) was calculated
178
179 // Interrupt service routine (ISR): run every time a change is registered at the pulse
        code wheel.
180 void speedCodeWheelPass() {
181   float dt = (float) (micros() - lastTimeCodeWheelPass) * 1E-6;
182   lastTimeCodeWheelPass = micros();
183
184   v = driveMeasureFilter.input(0.0100530 / dt);
185 }
186
187 // Flashes red LED every second, for one second
188 void flashRedLED() {
189   if ((millis() / 1000) % 2 == 0) {
190     digitalWrite(LED_RED, HIGH);
191   }
192   else {
193     digitalWrite(LED_RED, LOW);
194   }
195 }
196
197 // Balacing controller
198 float calcTheta_ref(float phi, float phiSpeed, float phi_ref) {
199   float v2 = v*v;
200   float L_cykel1 = L_cykel1_factor / v2;
201   float L_cykel2 = L_cykel2_factor / v2;
202   float l0_cykel = l0_cykel_factor / v2;
203
204   float theta_ref = -L_cykel1*phi - L_cykel2*phiSpeed + l0_cykel*phi_ref;     //
        Balancing controller, feedback loop
205
206   // Don't request larger angles than 50 deg (0.872664 rad)
207   if (theta_ref > 0.872664)
208     theta_ref = 0.872664;
209   else if (theta_ref < -0.872664)
210     theta_ref = -0.872664;
211
212   return theta_ref;
213 }
214
215 // Steer controller
216 float calcU_AS(float theta, float thetaSpeed, float theta_ref) {
217   float U_AS = -L_motor_s1*theta - L_motor_s2*thetaSpeed + l0_motor_s*theta_ref;    //
        Steer motor controller, feedback loop
218
219   if (U_AS > U_bank)        // Makes sure the regulator doesn't request higher voltages
        than the battery bank can provide.
220     U_AS = U_bank;
221   else if (U_AS < -U_bank)
222     U_AS = -U_bank;
223
224   return U_AS;
225 }
226
227 // Speed controller
228 float calcU_AD(float dt) {
229   float speedError = v_ref - v;
```

```
230   speedErrorHistory += speedError;
231   float U_AD = Kp*speedError + Ki*dt*speedErrorHistory;
232
233   if (U_AD > U_bank)      // Makes sure the regulator doesn't request higher voltages
          than the battery bank can provide.
234     U_AD = U_bank;
235   else if (U_AD < 0)
236     U_AD = 0;
237
238   return U_AD;
239 }
240
241 // Creates a new file on the SD card, with increasing trailing numbers in the file name.
242 void createFile() {
243   int filenumber = 1;
244   char filename[9];
245
246   strcpy(filename, "1.csv");
247   while (sd.exists(filename)){
248     filenumber += 1;
249     itoa(filenumber, filename, 10);
250     strcat(filename, ".csv");
251   }
252
253   dataFile.open((const char*) filename, O_CREAT | O_WRITE);
254 }
255
256 void collectSensorData() {
257   int trashMemory;    // Dummy variable
258   leanSensor.getMotion6(&lean_acc_x, &trashMemory, &lean_acc_z, &trashMemory, &
          lean_speed_y, &trashMemory); // This was slightly faster than getting the values
          separately.
259   steer_speed_x = steerSensor.getRotationX();
260
261   steerAngle = - (float) (analogRead(STEER_ANGLE_SENSOR) - 504) * 0.00579503;
262
263   // Convert to SI units and switch direction on steer angular velocity (it is mounted "
          upside down")
264   lax = ((float) lean_acc_x + calib_lean_acc_x) * 0.000599365;
265   laz = ((float) lean_acc_z + calib_lean_acc_z) * 0.000599365;
266   lsy = ((float) lean_speed_y + calib_lean_speed_y) * 1.33231E-4;
267   ssx = - ((float) steer_speed_x + calib_steer_speed_x) * 0.00106526;
268 }
269
270 void calcLeanAngle(float dt) {
271   float leanAngleAcc = -atan2(lax, laz);      // Lean angle according to accelerometer
272   float leanAngleGyrChange = lsy * dt;        // Change in lean angle according to gyro
273   leanAngle = lean_gyro_percentage*(leanAngle + leanAngleGyrChange) + (1 -
          lean_gyro_percentage)*leanAngleAcc;   // Complementary Filter
274 }
275
276 void logData(float dt) {
277   // Convert floats to C strings
278   dtostrf(v, 8, 3, vBuffer);
279   dtostrf(v_ref, 8, 3, v_refBuffer);
280   dtostrf(steerAngle, 8, 3, steerAngleBuffer);
281   dtostrf(steerAngle_ref, 8, 3, steerAngle_refBuffer);
282   dtostrf(U_AS, 8, 3, U_ASBuffer);
283   dtostrf(U_AD, 8, 3, U_ADBuffer);
284   dtostrf(leanAngle, 8, 3, leanAngleBuffer);
285   dtostrf(leanAngle_ref, 8, 3, leanAngle_refBuffer);
286   dtostrf(ssx, 8, 3, ssxBuffer);
287   dtostrf(lsy, 8, 3, lsyBuffer);
288   dtostrf(lax, 8, 3, laxBuffer);
289   dtostrf(laz, 8, 3, lazBuffer);
290   dtostrf(dt, 8, 3, dtBuffer);
291
292   // File structure: [v, v_ref, leanAngle, leanAngle_ref, steerAngle, steerAngle_ref,
          U_AS, U_AD, ssx, lsy, lax, laz, dt]
293   dataFile.print(vBuffer); dataFile.print(","); dataFile.print(v_refBuffer); dataFile.
          print(","); dataFile.print(leanAngleBuffer); dataFile.print(","); dataFile.print(
          leanAngle_refBuffer); dataFile.print(",");
294   dataFile.print(steerAngleBuffer); dataFile.print(","); dataFile.print(
          steerAngle_refBuffer); dataFile.print(","); dataFile.print(U_ASBuffer); dataFile.
```

```
        print(","); dataFile.print(U_ADBuffer); dataFile.print(",");
295   dataFile.print(ssxBuffer); dataFile.print(","); dataFile.print(lsyBuffer); dataFile.
        print(","); dataFile.print(laxBuffer); dataFile.print(","); dataFile.print(
        lazBuffer); dataFile.print(",");
296   dataFile.print(dtBuffer);
297
298   dataFile.println("");
299
300   /* One could do the logging by letting the sd library convert the floats, such as:
301    *
302    *  dataFile.print(v, 3); dataFile.print(","); dataFile.print(v_ref, 3); dataFile.
         print(","); dataFile.print(leanAngle, 3); dataFile.print(","); dataFile.print(
         leanAngle_ref, 3); dataFile.print(",");
303    *  dataFile.print(steerAngle, 3); dataFile.print(","); dataFile.print(steerAngle_ref,
          3); dataFile.print(","); dataFile.print(U_AS, 3); dataFile.print(","); dataFile.
         print(U_AD, 3); dataFile.print(",");
304    *  dataFile.print(ssx, 3); dataFile.print(","); dataFile.print(lsy, 3); dataFile.
         print(","); dataFile.print(lax, 3); dataFile.print(","); dataFile.print(laz, 3);
         dataFile.print(","); dataFile.print(dt, 3);
305    *  dataFile.println("");
306    *
307    *  This method might save some RAM (since the buffers/char arrays aren't needed), but
          takes about 4-5 ms longer to execute.
308    */
309 }
310
311 // System setup, run once when unit is turned on.
312 void setup() {
313   // Pin configurations
314   pinMode(LED_RED, OUTPUT);
315   pinMode(MOTOR_DRIVE_A, OUTPUT);
316   pinMode(MOTOR_DRIVE_B, OUTPUT);
317   pinMode(MOTOR_DRIVE_PWM, OUTPUT);
318   pinMode(MOTOR_STEER_A, OUTPUT);
319   pinMode(MOTOR_STEER_B, OUTPUT);
320   pinMode(MOTOR_STEER_PWM, OUTPUT);
321
322   pinMode(SPEED_SENSOR_INTERRUPT, INPUT_PULLUP);
323   pinMode(BUTTON_OK, INPUT);
324   pinMode(BUTTON_CANCEL, INPUT);
325
326   // Join I2C bus
327   Wire.begin();
328
329   // Initialize MPU6050 sensors
330   leanSensor.initialize();
331   steerSensor.initialize();
332
333   // Configure sensors
334   leanSensor.setDLPFMode(4);              // Activates the internal lowpass filter:
        Cutoffs at 20Hz, 8.3 ms delay.
335   steerSensor.setDLPFMode(4);
336   steerSensor.setFullScaleGyroRange(3);   // Uses the full range of the gyroscope for
        steering, 0 - 2000 deg / s.
337
338   // Test connections to sensors, if successful light up LED for half a second.
339   if (leanSensor.testConnection() && steerSensor.testConnection());{
340     digitalWrite(LED_RED, HIGH);
341   }
342   delay(500);
343   digitalWrite(LED_RED, LOW);
344
345   if (!sd.begin(SD_CARD_CS, SD_SCK_MHZ(50))) {
346     sd.initErrorHalt();
347   }
348
349   // Attach interrupt function to speed sensor pin
350   PCintPort::attachInterrupt(SPEED_SENSOR_INTERRUPT, speedCodeWheelPass, CHANGE);
351
352   // Calibrate gyros and accelerometers.
353   lean_acc_x = leanSensor.getAccelerationX();
354   lean_acc_z = leanSensor.getAccelerationZ();
355   lean_speed_y = leanSensor.getRotationY();
356   steer_speed_x = steerSensor.getRotationX();
```

```
357
358   calib_lean_speed_y = 0 - lean_speed_y;
359   calib_steer_speed_x = 0 - steer_speed_x;
360   calib_lean_acc_x = 0 - lean_acc_x;        // Calibration without support wheel (unit
          should be straight up and not moving)
361   calib_lean_acc_z = 16384 - lean_acc_z;    // Calibration without support wheel (unit
          should be straight up and not moving)
362   //calib_lean_acc_x = -2394 - lean_acc_x;      // Calibration with support wheel
363   //calib_lean_acc_z = 16209 - lean_acc_z;      // Calibration with support wheel
364 }
365
366 // Main loop when unit was just started or a program was shut off
367 // Flashes red LED every second
368 void loopStartup() {
369   // Check any input from buttons
370   bool okValue = digitalRead(BUTTON_OK);
371
372   if (okValue == HIGH)
373     okPressed = 1;
374   else if (okValue == LOW && okPressed) {
375     okPressed = 0;
376     speedErrorHistory = 0;
377     v = 0;
378
379     createFile();
380
381     state = STATE_PROGRAM_RED;
382   }
383
384   // LED indications
385   flashRedLED();
386
387   collectSensorData();
388
389   // Calculate change in time
390   unsigned long now = micros();
391   float dt = (now - lastLoopTime) * 1E-6;
392   lastLoopTime = now;
393
394   calcLeanAngle(dt);
395
396   // Automatic Control
397   //steerAngle_ref = calcTheta_ref(leanAngle, lsy, leanAngle_ref);
398   //U_AS = calcU_AS(steerAngle, ssx, steerAngle_ref);
399   //U_AD = calcU_AD(dt);
400
401   // Turn off motors
402   digitalWrite(MOTOR_DRIVE_A, LOW);
403   digitalWrite(MOTOR_DRIVE_B, LOW);
404   analogWrite(MOTOR_DRIVE_PWM, 0);
405   digitalWrite(MOTOR_STEER_A, LOW);
406   digitalWrite(MOTOR_STEER_B, LOW);
407   analogWrite(MOTOR_STEER_PWM, 0);
408 }
409
410 void loopProgramRed() {
411   // If user pressed cancel button, turn off the program
412   bool cancelValue = digitalRead(BUTTON_CANCEL);
413
414   if (cancelValue == HIGH)
415     cancelPressed = 1;
416   else if (cancelValue == LOW && cancelPressed) {
417     cancelPressed = 0;
418     dataFile.close();
419     regulateSteering = false;
420     state = STATE_STARTUP;
421
422     return;
423   }
424
425   // LED indications
426   digitalWrite(LED_RED, HIGH);
427
428   collectSensorData();
```

```
429
430   // Detect if unit was lifted up and stop program if that's the case
431   if (liftDetectionFilter.input(laz) > 11.5) {
432     dataFile.close();
433     regulateSteering = false;
434     state = STATE_STARTUP;
435
436     return;
437   }
438
439   // Calculate change in time
440   unsigned long now = micros();
441   float dt = (now - lastLoopTime) * 1E-6;
442   lastLoopTime = now;
443
444   calcLeanAngle(dt);
445
446   // Automatic Control
447   steerAngle_ref = calcTheta_ref(leanAngle, lsy, leanAngle_ref);
448
449   // With support wheel:
450   // Only regulate the steering when the bicycle has attained the desired speed.
451   // Otherwise steer slightly to the left to keep as straight as possible.
452
453   if (regulateSteering) {
454     //if (now > timeAttainedSpeed + 1000)
455       //leanAngle_ref = -0.174532;  // -10 deg
456   }
457   else {
458     steerAngle_ref = 0;
459
460     if (v > 1.6667) {       // 6 km/h
461       regulateSteering = true;
462       timeAttainedSpeed = micros();
463     }
464   }
465
466   steerAngle_ref = steerRefFilter.input(steerAngle_ref);
467
468   U_AS = calcU_AS(steerAngle, ssx, steerAngle_ref);
469   U_AD = calcU_AD(dt);
470
471   // Send PWM signals to motor drivers
472   digitalWrite(MOTOR_DRIVE_A, HIGH);
473   digitalWrite(MOTOR_DRIVE_B, LOW);
474   analogWrite(MOTOR_DRIVE_PWM, map(U_AD, 0, 12, 0, 255));
475
476   if (U_AS < 0) {
477     U_AS = map(abs(U_AS), 0, 12, 2, 12);
478     digitalWrite(MOTOR_STEER_A, HIGH);
479     digitalWrite(MOTOR_STEER_B, LOW);
480     analogWrite(MOTOR_STEER_PWM, map(abs(U_AS), 0, 12, 24, 255));
481   }
482   else {
483     U_AS = -map(abs(U_AS), 0, 12, 2, 12);
484     digitalWrite(MOTOR_STEER_A, LOW);
485     digitalWrite(MOTOR_STEER_B, HIGH);
486     analogWrite(MOTOR_STEER_PWM, map(abs(U_AS), 0, 12, 24, 255));
487   }
488
489   // Log data to SD card
490   logData(dt);
491 }
492
493 // Arduino main loop - selects appropriate loop depending on which state unit is in
494 void loop() {
495   switch (state) {
496     case STATE_STARTUP:
497       loopStartup();
498       break;
499     case STATE_PROGRAM_RED:
500       loopProgramRed();
501       break;
502   }
```

```
503 }
```

TRITA-ITM-EX 2018:58