CHALMERS

# Programming a Self Driving Bike

## Implementing a Balancing Algorithm in a Self Driving Bike

Bachelor's Thesis in Department of Electrical Engineering

ELLIOT ANDERSSON, HANNES HULTERGÅRD

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

Bachelor's thesis 2022

# Programming a Self Driving Bike

Implementing a Balancing Algorithm in a Self Driving Bike

ELLIOT ANDERSSON
HANNES HULTERGÅRD


CHALMERS

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2022

Programming a Self Driving Bike
Implementing a Balancing Algorithm in a Self Driving Bike
ELLIOT ANDERSSON, HANNES HULTERGÅRD



Supervisor: Jonas Sjöberg, Professor, Electrical Engineering
Examiner: Jonas Sjöberg, Professor, Electrical Engineering

Bachelor's Thesis 2022
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: The red version of the Autobike.

Typeset in LaTeX
Gothenburg, Sweden 2022

Programming a self driving bike
Implementing a balancing algorithm in a self driving bike
ELLIOT ANDERSSON, HANNES HULTERGÅRD
Department of Electrical Engineering
Chalmers University of Technology

# Abstract

The Autobike project was started at Chalmers in 2017 with the goal of creating a self driving bike, which could be used to test the safety systems in cars. The part of the project whose aim is the basis for this report is the implementation of a balancing algorithm in the existing program for one version of the bike, the so called *red bike*. This algorithm has the purpose of making the bike be able to balance on its own, whilst driving forward. The hardware required to reach this goal already exists, and will therefore be left as is. This bike uses the graphical programming environment LabVIEW for its main program, although in some situations or for some features, a text based programming language is preferred. LabVIEW can for this reason call shared libraries using the *Call Library Function Node*. In the case of this project, the balancing algorithm has been translated from Python to C. The algorithm uses the roll rate of the bike to calculate a duty cycle, this value is then sent to a motor which steers the front wheel. Code to control the motor which drives the bike forward has also been written. This code is called from LabVIEW using the same method as for the balancing algorithm, and from an inputted RPM value returns a command which is sent using UART from LabVIEW to the VESC motor controller. A secondary goal of the project has been to improve the development experience of the code by reorganizing code, removing unnecessary files, and uploading the code to a public GitHub repository. The result of implementing the balancing algorithm and the code to control the forward motor is the bike being able to balance on its own whilst driving forward.

Keywords: Autobike, bike, bicycle, self driving, balancing, LabVIEW, MyRIO, C, PID.

# Acknowledgements

This bachelor's thesis was written by to students at the Mechatronics Engineering program at Chalmers University of Technology during the spring of 2022.

We would like to thank our examiner and supervisor Jonas Sjöberg for his guidance and for giving us the opportunity to work on the Autobike. We would also like to thank Henrik Falk at Mälardalen University for helping us getting started with LabVIEW, and for answering any questions that we had about the program during the course of the project. Lastly, we would like to thank Guanzheng Wen and Antonin Le Noc who has worked with us on the Autobike and helped us with solving problems and testing the bike.

Elliot Andersson and Hannes Hultergård
Gothenburg, June 2022

# Contents

# 1

# Introduction

This introductory chapter initially presents the background behind the Autobike project, included in this background is when and why the project was created. The background also gives an overview of the current state of the Autobike project. The different aims of the given assignment are also described, this section clarifies what the main goal is and the sub-goals thereof. Finally the limitations and boundaries of the assignment, its aim and its goals, are formulated.

## 1.1 Background

As self driving cars are becoming more common, it is important that they can react properly to all possible traffic situations. One of these situations include cyclists, which have a unique behavior which the cars will need to be able to handle [1]. Due to this unique behavior, bikes should be classed as a distinct object when training the self driving algorithms. To achieve this, real bikes have to be used in the training. However, since there is a possibility of the cars hitting the bikes during development, it is not reasonable to have a person driving the bike. A need to develop a self driving bike that replicates a bicycle and a real cyclist as closely as possible has therefore appeared.

The Autobike project was started at Chalmers in 2017 with the goal of creating a self driving bike, which could be used to test the safety systems in cars. The bike needed to resemble a normal bike to the greatest extent possible, while also being able to balance on its own and follow a predetermined path. Two versions of the bike has been created at Chalmers, the *black bike* and *red bike*. The red bike was later improved upon by students at Mälardalen University (MDU) during 2021; it is this version of the bike which is in need of further development from this project.

The previous work on the red bike focused primarily on the hardware, as well as parts of the software made using the graphical programming language LabVIEW [2]. The main purpose of the LabVIEW code is to communicate with the sensors and I/O on the MyRIO-1900 [3] which controls the bike. The main software components which are yet to be completed or created for this version of the bike are the control

algorithm for the two motors of the bike, as well as the code for communication with the so called *forward motor*, which drives the bike forward [4].

The other version of the bike that was created at Chalmers, the black bike, is completely programmed in Python. This version has an algorithm for balancing the bike while it is moving forward. It is this algorithm which is supposed to be used in the red bike, but since LabVIEW and the MyRIO-1900 does not support Python, the algorithm must be converted into C.

## 1.2 Aim

The main aim or goal of the assignment is to further develop the software for the Autobike, so that the red bike can balance on its own, while driving forward. To guide the project towards reaching the main goal, a number of sub-goals have been created.

The first, and most important sub-goal is the adaption and implementation of the balancing algorithms written for the black bike. This algorithm must be changed and adapted to function together with the hardware and software used by the red bike. The desired result of implementing the algorithm is that the input signals for the steering motor can be calculated based on the signals from the bike's sensors as well as parameters set in the bike's software. The completed code should also guarantee that the bike is controlled within desired safety limits.

Another sub-goal is programming the software needed to communicate with the bike's forward motor. Achieving this aim will result in the ability to control the speed of the motor from the bike's software.

A secondary aim of the project is to improve the development experience of the bike's software. This includes organizing the project's files, and commenting all of the code so that it can be understood by future developers. Accomplishing this should result in it being easier to continue the development when this iteration of the project is completed.

## 1.3 Limitations

The main limitation of the given assignment is that the only part of the Autobike which will be changed is the software; the hardware will be left in its current state. Additionally, code that enables the bike to follow a predetermined path will not be written. It should therefore not be a goal that the bike can drive in a straight line either, since this would require knowledge about the current and past positions. Thirdly, the speed of the forward motor should not be able to be changed by the balancing algorithm.

# 2

# Theory

This chapter describes the theory behind the method of the project. It does this by describing the programs and hardware that has been used during the development of the Autobike, or in the final result. Note that the hardware presented below does not include all components of the Autobike, but instead only a selection which were deemed relevant to this project and report.

## 2.1 LabVIEW

LabVIEW is a graphical programming environment used to "develop automated research, validation, and production test systems" [2]. The language used in LabVIEW is called G, and programming is performed by connecting blocks called VI's (virtual instruments) or nodes using wires [5]. These VI's can either be a "sub-VI" written in G, or primitives built into G.

National Instruments (NI), the developers behind LabVIEW, describes the benefits of the G programming language as being easy for engineers and scientists to learn, as well as being more intuitive to debug than text based languages [6]. They also describe LabVIEW as allowing "automatic parallelization", since it is a dataflow language, in contrast to sequential languages like C where this is not possible. Figure 2.1 below shows an example of a LabVIEW program.

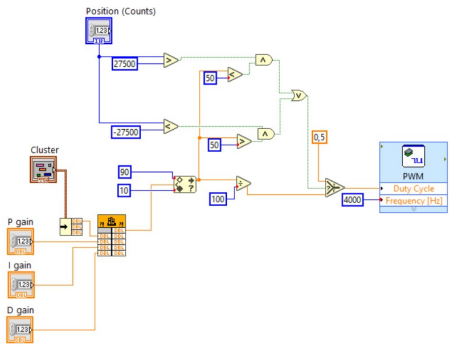



**Figure 2.1:** Example of a LabVIEW program.

In addition to the LabVIEW code exemplified above, each program has a corresponding front panel where the user can interact with indicators and input controls. [5]

### 2.1.1 Integration With Text Based Languages

Certain parts of a LabVIEW program may not be not suited for development in G, instead a text based language can be used and integrated into LabVIEW [6]. The integration can be either done by using the *Formula Node* to write code with a syntax similar to C, or using the *Call Library Function Node* to call a DLL (Dynamic-link library) or a shared library function [7]. The node takes in function parameters as specified in the node's settings, and uses their values when calling the specified function. When the called function completes, its value is returned and can be used by other parts of the LabVIEW program.

To be able to call a function using the Call Library Function Node the code must be compiled to either a DLL, or a shared library (a file with a .so filename extension) if using a Linux based target [8]. This library or DLL must then be uploaded to the target using for example SFTP (SSH file transfer protocol). If the target is Linux based the library should be uploaded to the `/usr/local/lib` directory. The name of the .so file should then be specified in LabVIEW inside the settings of the Call Library Function Node.

## 2.2 MyRIO-1900

The MyRIO-1900 is a device that combines multiple components such as an ARM processor, an FPGA (field programmable gate array)[1], as well as both analog and digital I/O (input / output) lines to help "students and educators complete real engineering projects in one semester" [3]. The RIO architecture that the myRIO uses combines a processor with the I/O of the device via an FPGA [10], [11]. In the RIO architecture, the FPGA is connected directly to the I/O and is used to offload critical and intensive tasks from the processor [11]. It also makes high throughput tasks that run on the FPGA more deterministic than if they had run on the processor.

## 2.3 ESCON Motor Controller

The ability to steer the front wheel of the bike is made possible by a the so called steering motor. The motor controller for the steering motor is an ESCON 50/5 designed by Maxon. This motor controller is a "PWM servo controller", which means it controls motors using Pulse Width Modulation (PWM). The PWM duty cycle range of the controller, meaning the range between which values the duty cycle can be varied, is 10% to 90% [12].

[1] An FPGA is an integrated circuit that consists of an array of logic gates, which can be configured by the user to create a specific hardware architecture [9].

### 2.3.1 ESCON Studio

The ESCON motor controller has an accompanying graphical user interface called ESCON Studio which features the ability to configure the controller's parameters [13]. These parameters can for example change the motor's maximum angular velocity or its acceleration. ESCON Studio also allows for the parameter configurations to be downloaded and transferred between different controllers. The software has other features, for example a diagnostic tool which can measure how the motor's speed follows a certain setpoint value.

## 2.4 Steering Motor Encoder

To measure the position of the steering motor, or how far the front wheel has been rotated, an encoder of the model HEDS-5540#A11 is used. The encoder translates rotational motion into a digital signal by using a codewheel that interrupts a light beam when it spins [14]. The encoder outputs a waveform signal with a resolution of 500 counts per revolution.

## 2.5 Inertial Measurement Unit (IMU)

The bike features an Inertial Measurement Unit, or IMU, the specific model of IMU used is a Pmod NAV designed by Digilent. This device features an accelerometer and a gyroscope, as well as a magnetometer and a barometer, the latter pair of which are not used in this project. Both the accelerometer and gyroscope makes measurements in relation to the 3 Cartesian coordinate axes [15]. The units of the given measurements are $g$ (circa $9.81 m/s^2$), and *dps* (degrees per second) for the accelerometer and gyroscope respectively; these measurements are given separately for each of the three axes [16].

## 2.6 UART Communication

Universal asynchronous receiver-transmitter, abbreviated to UART, is a device-to-device hardware communication protocol [17]. UART is used to transmit and receive serial data between two devices, this is achieved with only two wires for the transmitting and receiving ends. Each of the UART devices has two signals named transmitter (Tx), and receiver (Rx); the transmitter of one device should be connected to the receiver of the other, and vice versa.

The UART interface is, as its name implies, asynchronous, meaning it does not use a clock signal for synchronization. Instead a bitstream is generated by the transmitter, which is sampled by the receiver; synchronization is accomplished by using the same baud rate on both of the devices. The baud rate is the rate at which the information from the transmitter is sent.

Information is transferred in the form of a packet. The packet includes a start bit, the data frame, an eventual parity bit, and finally one or two stop bits. The start and stop bits indicate the borders of a packet, meaning its start and end. The data frame contains the actual data which is transferred, it has a maximum length of 9 bits if no parity bit is used, otherwise the maximum length is 8 bits. The parity bits can optionally be used to tell if the data frame has been changed during the transmission.

## 2.7 VESC

The VESC is a controller for brushless DC motors, also known as an ESC (electronic speed controller), created by Benjamin Vedder [18]. Both the hardware and firmware are open source, and available on GitHub[2]. Communication with the ESC can be done using either a PPM signal, analog, UART, I2C, USB or CAN [19].

Vedder describes in [20] that when communicating with the controller using UART, each packet must be wrapped in bytes containing information about itself. The packet should begin with a start byte with a value of either 2 or 3, depending on the length of the packet. This should be followed by one or two bytes that specifies the length of the packet, followed by the payload. At the end, two CRC[3] checksum bytes, and a stop byte with a value of 3 should be added. Vedder continues to write that a UART packet has to be sent "at regular intervals" to prevent the ESC from timing out; this can be either a specific send alive packet, or any other command that sets a value. The content of [20] is based on an example for an STM32F4 discovery board which implements UART communication with the VESC in C [22].

### 2.7.1 VESC Tool

The developer of the VESC has developed a graphical user interface for interacting with the motor controller [19]. This tool should be used to configure the ESC to work with the specific motor that is being used. It can also be used to test the motor and display real time data like the current drawn, or the temperature of the VESC.

[2]The hardware and firmware can be found on GitHub at `https://github.com/vedderb/bldc-hardware` and `https://github.com/vedderb/bldcrespectivly`.

[3]CRC, or cyclic redundancy check, is used to detect if the payload contains errors [21].

# 3

# Methods

The methods chapter describes the methods which have been used in an attempt to reach the end goal of the project. In addition to this, it is explained why these specific methods were chosen. Any additional information which might be needed to understand the used methods and reasoning behind them, is located within the previously presented theory section.

## 3.1 Configuring the Toolchain

The tools and software used to program and control the Autobike had already been decided to a large extent by the previous groups which worked on the project, as well as other stakeholders. It was therefore necessary to install these tools before the previous work could be examined to the fullest extent, and before further work could commence.

The installation process of these tools, will be presented in the upcoming subsections. Installation steps of other pieces of software will also be presented, together with their purpose and the reasons behind why they were decided to be used.

### 3.1.1 LabVIEW

The parts of the Autobike software that were already written had been done in LabVIEW. The LabVIEW program itself was therefore installed with the purpose of being able to view this code. Since a MyRIO-1900 had been selected to run the code, a special toolchain of plugins and programs which supports the configuration the device, as well as uploading and running code on it, also had to be installed. These programs were all first party tools available on the National Instruments website[1], or in the package manager which was installed alongside LabVIEW.

[1] All the required software for developing for the MyRIO is available in the *LabVIEW myRIO Software Bundle* available for download on the National Instruments website: `https://www.ni.com/sv-se/support/downloads/software-products/download.labview-myrio-software-bundle.html`

It should be noted that the installation of LabVIEW and the necessary packages is more time consuming and problem-ridden than what might first be expected. The installation itself is not uncommon to take multiple hours, especially if the computer's storage space runs out in the middle of an installation. This step was further delayed due to downloads from the National Instruments website not working for several days.

### 3.1.2 Text Editor and Build Tools for C Code

Some of the more advanced algorithms used in the Autobike were better suited to be written using C code, and called by the main LabVIEW program. These algorithms include code written for another version of the bike with the purpose of controlling the steering motor, as well as code needed in the communication with the forward motor.

To be able to use C code with LabVIEW, a toolchain capable of both writing C code, but also (cross) compiling it, was required. The toolchain would also need to create shared libraries as well as uploading these to the target. The selected software, and the rationale behind the choices are further discussed below.

#### 3.1.2.1 Text editor / IDE: Visual Studio Code

National Instruments has published a guide for setting up a C development toolchain using the IDE (integrated development environment) Eclipse [23]. However due to preferential reasons and the project members' familiarity with Visual Studio Code (VS Code), it was chosen as the editor instead of Eclipse.

The basics of the previously mentioned guide can still be followed even though the details differ. There is a guide published on the National Instruments forum that describes how VS Code can be set up to work with the MyRIO as well [24]. Although this is not an official guide, it is authored by a NI employee. After installing VS Code, the *C/C++ for Visual Studio Code* extension is recommended as it allows for code-completion aids and debugging capabilities for code written in C [25].

#### 3.1.2.2 Compiler and Build Tools: GCC, CMake and Ninja

Before the C code can be called from within LabVIEW when it is running on a MyRIO, it must first be compiled and built into a shared library as mentioned in section 2.1.1. The previously mentioned guide for setting up VS Code to develop for a MyRIO also goes through how to set up the needed build tools [24].

The compiler used in the guide is a version of GCC made to run on Windows, and compile the code for Linux running on ARMv7; this compiler is used since VS Code is running on a computer using windows and the myRIO has an ARM processor. To aid the build process CMake, which is "a family of tools designed to build, test and package software", is used to generate makefiles [26]. These files can then be

used by Ninja, a build system, to run the compiler and create a shared library or `.so` file [27].

#### 3.1.2.3 FTP Client: Filezilla

When the shared library or `.so` file has been created, this file needs to be transferred to the myRIO. To accomplish this, the guide recommends Filezilla, which is a "cross-platform FTP, FTPS and SFTP client" [28]. Filezilla was used to connect to the myRIO and upload the files as described in 2.1.1.

## 3.2 Examining Previous Work and the Current State of the Project

When the tools necessary for viewing and running LabVIEW code had been set up, the existing code was examined and tested. This was done in order to learn what parts of the Autobike's software had not yet been completed. It was concluded that all hardware except the forward motor could be interacted with in some way from the main LabVIEW program.

All sensors, including the GPS, gyro, accelerometer, hall sensor (for measuring the forward speed), and position sensor (to measure the steering wheel position) was visually represented by graphs or similar on the LabVIEW front panel. The steering motor duty cycle could be controlled using a slider, and the emergency stop could stop the program except for in some edge cases (it should however be noted that the emergency stop always physically broke the circuit).

In addition to the main LabVIEW program, several other VIs were present in the folder handed over by the previous project group. Some of these were used as sub-VIs in the main program, but the majority of them seemed to be test code that had been used during the development process. One of these programs appeared to be an attempt at communicating with the forward motor; attempts to make the code to work were however unsuccessful, as further described in 3.4.

## 3.3 Steering Motor

At the start of this project progress related to being able to control the steering motor had already been made by the previous project group. Excluding the hardware, LabVIEW code which could communicate with the steering motor controller, using a PWM signal, had already been completed. The duty cycle of the PWM signal had a value ranging from 0% to 100%, where a values above 50% would rotate the motor clockwise, and values below 50% rotate the motor anti-clockwise, a value of 50% would stop the motor.

The code consisted of three states: a start state, the control loop, and the end state. The start state made sure the motor was still and thereafter disabled the pin where

the signal was sent, and waited until the user told the motor to start with a "Go" button. After being told to start, the control loop would be entered and the pin was enabled, which allowed the user to control the speed and direction of the motor using a slider; the slider changed the PWM signal sent to the motor controller. Finally the end state would be entered if a stop signal was given, for example by activating the physical emergency stop button, this state also stops the motor and disables the pin.

Testing the code by starting the program, pressing the "Go" button, and changing the PWM signal, the code proved to be working as intended; the motor could rotate in both directions and change speed accordingly. Therefore it was decided the code was going to be used, with minor alterations in the form of exchanging the slider for the previously mentioned control algorithm.

The following subsections describes the methodology and process of completing the software for the steering motor. This includes identifying and translating relevant code from another version of the Autobike, using the gyroscope built into the IMU together with the algorithm to control the steering motor, and implementing safety limits to the motor's range of motion.

### 3.3.1 Configuring the ESCON Motor Controller

Since the black bike has the same forward motor and controller, as well as a functioning configuration of these, the ESCON motor controller parameter configuration was exported using ESCON Studio and uploaded to the controller of the red bike. After copying the configuration, the diagnostics tool in ESCON Studio was run to validate that the motor worked, and that its angular velocity followed the setpoint value. The most important settings included in the transferred configuration includes the maximum angular velocity of 4000 RPM, and an increase of the motor's acceleration.

### 3.3.2 Transferring Control Algorithm From the Black Bike

To reach the end goal of creating a bike which can balance on its own, code that was written for the other version of the bike, the so called black bike, were to be converted as to work with the current version which this project relates to. The software for the black bike had been written completely in Python, however it still used a MyRIO.

Before the code could be used, the parts of the code which are relevant to the current bike, had to be identified. This process, followed by the conversion of the code to C, and its integration with LabVIEW is described below.

#### 3.3.2.1 Identifying Relevant Code

Since the software used on the black bike was written entirely in Python, a large amount of the code had the same or similar functionality as the LabVIEW code

for the red bike. For reference, the folder that was received contained in total 156 python files with a combined 20 253 lines of code[2]. Code which was used for loading simulations from CSV files were also present in most of files. From this it could be concluded that almost all of the received code was not relevant for the red bike and would not have to be used.

Trajectory tracing is also not a part of this project's goal, meaning the only part of the code that would be needed was the code which relates to keeping the bike stable. This code was located in the file `controller.py` and more specifically in the function `keep_the_bike_stable`. The function also called another function, `update`, which implemented a PID controller in parallel form according to:

$$u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de}{dt}$$

The value returned from this function is then further sent to another function, named `controller_set_handlebar_angular_velocity`. This final function calculates the angular velocity of the motor based on some safety limits, and calls a function to convert the angular velocity to a duty cycle. In Python, this conversion is done using the `interp` function from the *NumPy* library with the velocity on the x-axis, and PWM on the y-axis. In C this as been converted to `50 + rpm * 40.0 / 4000.0`, where 4000 is the maximum angular velocity (in RPM) of the motor. This gives a value between 10 and 90 when `rpm` is in the range of the motor's angular velocity.

#### 3.3.2.2 Converting Python to C

The previous project group that worked on the bike considered using a language called Cython to compile the Python code to C, and thus avoid having to manually convert the code [4]. The group mentions that the translation created code which was confusing and hard to understand, and is was concluded that "Cython is not useful for this project". It became further apparent that Cython was not needed after the relevant Python code had been identified. This conclusion was made based on the relatively small amount of code that would actually have to be converted. It was therefore decided that the code was going to be converted manually.

The conversion process started with a new C project being created and the identified Python functions being copied there. The code was then adapted to fit the C syntax, at the same time as irrelevant code was removed. This irrelevant code mainly included lines that was only called when a simulation file were present, but some code was also removed since it did not have any effect on the rest of the program (for example variables that were written to, but then never used). The safety limits were also removed and not implemented in the C code because it was decided they would be created in LabVIEW instead. This decision was based upon two reasons:

[2] As reported by running the following commands in a Linux terminal:
`find . -mindepth 1 -type f -name "*.py" -printf x | wc -c` and
`wc -l ‘find . -type f -name "*.py"‘`

to facilitate a better understanding and easier debugging of the code by having as much code written in LabVIEW as possible, as well as being able to use LabVIEW to log when a safety limit had been exceeded.

Another change that was made to the control algorithm when converting it into C code was the removal of code that depended on the time, i.e. the integral and derivative action in the PID controller. The integral of the error was instead implemented by adding the current error to the previous errors once every time the function was called. The derivative of the error was calculated as the difference between the current and previous error (the error the last time the function was called). The resulting function is shown below.

```
1  double pid(double reference, double currentValue, double Kp, double
       Ki, double Kd) {
2      double error = reference - currentValue;
3
4      integral += error;
5      derivative = error - previousError;
6
7      if (integral < -windupGuard) {
8         integral = -windupGuard;
9      } else if (integral > windupGuard) {
10         integral = windupGuard;
11     }
12
13     previousError = error;
14
15     return Kp * error + Ki * integral + Kd * derivative;
16 }
```

### 3.3.3 Automating Steering Motor Control

To automate the control of the steering motor, meaning automatically calculating the duty cycle deciding which direction and how quickly the motor should rotate, a shared library file of the converted C code was created and uploaded to the MyRIO. The bike's roll rate[3], read from the IMU's gyroscope, as well as the three respective PID gains were then connected to a *Call Library Function Node* in LabVIEW. The node was set to call one of the functions from the C algorithm. The called function uses these four parameters and returns the calculated duty cycle.

The received duty cycle can theoretically have a value outside the range of 10% to 90%. Because of this the duty cycle is coerced to fall within this range using the *In Range and Coerce* function in LabVIEW.

For safety reasons the range which the wheel can turn within needs to be limited. By using the steering motor encoder, the position of the steering motor can be calculated in degrees from when the wheel is straight. As mentioned in 2.4 the encoder has a resolution of 500 counts per revolution, but when turning the wheel a full rotation,

[3]Roll rate in this case means the speed of which the bike rotates along its lengthwise axis.

the resulting counts are approximately 220 000; the reason for this is unknown. Thus the resulting factor for converting between counts and degrees is 360/220000. In LabVIEW the rotational position is then compared with a limitation, configured from the main program's front panel. The current limit is set to 45 degrees in each direction, as this is the limit used for the black bike. If the limit is exceeded the duty cycle is set to 50%, meaning the motor should stop.

The duty cycle is then sent to a node in LabVIEW which creates a PWM signal. This signal replaces the previously existing slider and is in turn sent to the hardware pin (pin 19) which the steering motor controller is connected to.

#### 3.3.3.1 Calibrating Gyroscope

The IMU's placement at an angle resulted in none of its axes correctly corresponding to the roll rate of the bike. Therefore the misalignment had to be changed or compensated for to make the gyroscope's readings as accurate as possible. Code could be written which compensates for this by combining readings from multiple axes, but this was considered to be too complicated in comparison with rearranging the IMU. Even though the project's limitations included that the hardware of the Autobike would not be changed, the rubber block which the IMU is placed in, was measured and cut so that the IMU's Z-axis corresponded with the bike's roll rate.

When the automated control of the steering motor was tested, it was discovered that the motor rotated even though the bike was not moving. During an investigation of the graphs that plotted the gyroscopes values, it appeared as though the average value of the rotational rate around every axes of the gyroscope was non-zero, even when the bike was stationary. This seems to be a case of a "constant bias error" [29].

To counteract the error, the bike was kept stationary over a time while the values returned from the gyroscope was recorded. The average of these recordings was then calculated and the negative of this average was added to the gyroscope's output for each axis to correct the error. It should be noted that the error or "constant bias error" might drift over time, meaning the error might change. If this occurs the average offset could be calculated again, or more sophisticated methods could be utilized as mentioned in [29].

## 3.4 Forward Motor

When the project was received, the main LabVIEW program contained no code with the purpose of controlling the forward motor. The reason for this code not being created was that "the command used for controlling the forward motor controller were not found" as described in the previous project's report [4].

The MyRIO communicates using UART with the forward motor controller, which will be refereed to as *the VESC* from this point on. The hardware necessary to control the forward motor using UART was implemented; the RX and TX wires

had been connected between the MyRIO and the VESC, which in turn is connected to the forward motor. In LabVIEW there existed a VI which sent a command to the UART port connected to the VESC, however this program was undocumented and did not appear to work when it was tested.

Based on the previously created code, or lack there of, the LabVIEW code to communicate with the VESC using UART and consequently control the forward motor had to be created from scratch. This process with be described in the upcoming subsections. This process consist of configuring the VESC, encoding the correct UART command, successfully sending this command using LabVIEW, as well as some general testing and tuning.

### 3.4.1 Configuring the VESC

The development of the code meant to control the forward motor started with trying to use the already existing test program to send a UART command to the VESC. Even though the command appeared to match the description from 2.7, the motor did not react. Why this was the case was unknown, which created doubts concerning which part of the setup caused the problem.

In an attempt to narrow down the cause of the problem, a computer running the VESC Tool was connected directly to the VESC via USB. Using this tool, an *experiment* was made where a fixed RPM was sent to the controller. This resulted in a high pitched noise, as well as close to no movement of the motor.

It was hypothesized that the VESC was misconfigured in regard to the specifications of the forward motor. To try to fix this issue, the setup wizard built into the tool was used. In the wizard, the requested values and parameters was set to that of the motor [30]. The values which could not be found in the forward motor's data sheet were set to the recommended values. When running the same experiment as before and setting the motor's RPM to a fixed value, the motor now turned at what seemed to be the correct speed, without any screeching.

The VESC Tool is also used to make sure the VESC is set to use UART as its communication interface. This is done by changing the *APP to Use* to UART under *App Settings* > *General*. The baud rate can also now be accessed and changed, but is kept at the default value of 115 200 bps.

### 3.4.2 Encoding UART Command

The UART communication with the VESC is based on sending various commands which should make the motor react in a certain way. Apart from the blog post by Vedder cited in section 2.7 as well as his example for the STM32F4 discovery board, there exists no documentation for how the UART commands sent to the VESC should look.

The first attempt at encoding a UART command was made almost entirely in LabVIEW by setting a RPM value in the front panel, flipping its endianness, and wrapping it in start and stop bytes as well as checksum bytes. The payload was also split into separate bytes in the same way as in Vedder's example. Due to lacking documentation regarding the UART implementation, and there being no easy way of knowing exactly what bytes a correctly encoded message should contain, this method was later abandoned.

It was instead decided that a shared library would be created based on the example made by Vedder; the shared library could then be called using a *Call Library Function Node* in LabVIEW. This method circumvented the problem of having to write the communication in a, for this purpose, undocumented language. Instead the correct command could be returned from a C function based on an inputted RPM value.

To create this library which could be called by LabVIEW, the code that created the desired UART commands had to be identified. The functions to call and send specific commands was located in the `bldc_interface.c` file. The three functions that have been adapted to work with the myRIO are `bldc_interface_set_rpm`, `bldc_interface_set_current`, as well as `bldc_interface_send_alive`. They have also been renamed to `setRpm`, `setCurrent`, and `sendAlive` respectively, to better represent their new purpose. These functions was chosen since they were deemed to be the most useful commands related to controlling the forward motor. Functions later called by these functions have also been adapted and, in some cases, simplified. The functions `buffer_append_int32` and `buffer_append_float32` converts an integer or float values to a big-endian array, and `send_packet_no_fwd` then calls a function to build and send the actual packet. In the adapted code, there is instead a function named `buildPacket` which builds and returns the packet. This function also calls `crc16` to calculate and add the checksum bytes. Finally the packet is copied to an array which is passed as a pointer to `setRPM` and `setCurrent`.

### 3.4.3 Sending UART Command Using LabVIEW

To create the UART command using LabVIEW, initially the desired speed of the bike in km/h is converted to the RPM of the bike's pedals. This conversion is based on the dimension of the back wheel combined with the gear ratio from the pedals to the wheel. The converted value is then sent to a *Call Library Function Node* which is set to call the correct function from the C code, after it has been compiled and uploaded to the myRIO. The called function has an additional input parameter in the form of a pointer to an array. This array is initialized in LabVIEW, so that the pointer of it can be sent to the C code. When the C code runs, the pointer is used to populate the array with the values of the command which should be sent to the VESC.

The actual transmission of the command is accomplished using the *UART* block in LabVIEW. The block is configured to use the UART channel corresponding to

the connection with the VESC, the baud rate is also set to 115 200 bps which is the same value as was previously configured in the VESC Tool. The transmission is placed inside of a loop to prevent the VESC from going to sleep.

The loop which which was desired to be used was a *Timed Loop*, which executes at a precise rate specified by the user. Though due to problems with the program crashing, the type of loop was changed to a *While Loop*; the cause of these crashes are unknown. A delay of one second was also introduced inside the loop to prevent it from throttling the rest of the system. The change of loop type did resolve the program crashing, though did also introduce another issue with recording loop times. This is further discussed in 3.7.5.

## 3.5 Cleaning up and Organizing the Code

In an attempt to make it easier for both the current and future people working on the Autobike project, all of the LabVIEW and C code that is used has been moved into a single folder. Previously it was separated into two different folders which had to be placed in specific locations on the development computer for LabVIEW to be able to detect them. The folder has also been uploaded to a monolithic public GitHub repository[4]. The purpose of this is to make it easier to collaborate on the code, as well as making it possible to keep track of changes. It should also make it easier for others to take over development of the bike, as this was previously unnecessarily tedious. However, due to how LabVIEW functions, there is still one file which has to be manually located when opening LabVIEW on a new computer, this file is located inside `labview/bin` together with all other files that the LabVIEW project uses.

The LabVIEW code that was received from the previous project group has been cleaned up, meaning that the main program has been separated into several sub-VIs, and unused code has been removed. The main advantage of creating sub-VIs is to reduce the area which the LabVIEW code uses; a large area makes it hard to locate code since you can not zoom inside LabVIEW. The code that has been removed are mainly VIs that were either old versions of the main VI, or old code used for testing individual components (presumably from before this was integrated into the main VI). By removing unused VIs the potential confusion about what a VI does, and which VI is the most recent one, is mitigated.

Care has also been taken to document the new code that has been written, as well as the old code, so that anyone can continue the development process without having to interpret all of the code without any guidance. In addition to writing comments inline in the C and LabVIEW files, a `README.md` file has also been created in the root of the repository. This file lists all the folders that are currently present and explains their purpose. When relevant, this file also contains instructions for how

[4]https://github.com/Hannnes1/autobike

the content of the folder should be used. A user manual for the LabVIEW program has also been written. This can be found in appendix A.

## 3.6 Logging of Control and Sensor Signals

With the purpose of being able to record and log the control and sensor signals of the bike, a logging system had to be created. This system would allow for future logs of a test being plotted and analyzed, helping the tuning process of the system. The logs could also be used to troubleshoot potential problems with the bike's performance.

In this specific case to be able to examine the data from performed tests, values from sensors, actuators, and the balancing algorithm added to the logging system. LabVIEW has built in functionality for this using the *TDMS file format* [31], which includes blocks for opening and creating, streaming to, and closing TDMS files. These files can then be opened using a number of different methods. Some of these methods include a special LabVIEW VI, Excel, or MATLAB using the *Data Acquisition Toolbox*.

In the main LabVIEW program used on the bike, a TDMS file is either created or replaced in the specified location using a *TDMS Open* block. In each location where data should be logged, a *TDMS Write* block is added and a group name and channel name is specified. Data is inputted as an array where each element corresponds to one channel (when the data layout has been specified as *interleaved*). Each time that data is sent to the TDMS block, new data points are added to the channels. Therefore by placing the block inside of a loop, data from each iteration of the loop will be logged to the file. When a test is completed and all of the test data has been recorded, the file is closed using a *TDMS Close* block before the program finishes.

MATLAB was chosen as the method for analyzing the data which the logging system gathers. MATLAB was specifically chosen due to the flexibility it grants regarding how the data can be presented. The code to do perform the plotting is presented in appendix B.3,

## 3.7 Testing and Validating

During the development of the bike's software, several tests have been done continuously to ensure that the developed features worked as expected. These tests include both simpler and informal tests designed to validate the bike's most basic functions, and more formal tests where values of different signals were logged.

To simplify the process of replicating the tests, a testing protocol was created. In this protocol, all tests and results have been documented so that they can easily be understood and reproduced, both for this and future versions of the bike. Steps which had to be taken to reach a satisfactory result have also been documented.

In this section the tests which have been performed are presented together with each test's purpose. More detailed descriptions and procedures are described in the test protocol found in appendix C. The most important results from the testing are also presented in the upcoming chapter 4.

### 3.7.1 Testing the Basic Functionality of the Forward Motor

The first test of the forward motor was described in 3.4.1. Summarized, it was performed by using the VESC Tool to run the motor at a fixed RPM. Later on, when the code to send UART commands via LabVIEW had been created, a fixed RPM was instead sent to the motor via one of these commands. Both of the tests mentioned above were performed when the bike was stationary and elevated into the air, this to insure the bike would not start moving. During these tests, the load on the motor was varied by pressing a foot against the back wheel of the bike with different pressures. The purpose of this was to ensure that the controller in the VESC worked as intended and the speed of the motor stayed constant during varying loads.

### 3.7.2 Testing the Basic Functionality of Steering Motor

To ensure that controller in the ESCON worked, meaning the speed of the forward motor followed the setpoint value, a test was performed in ESCON Studio. This test plotted the measured velocity of the motor, in comparison with the setpoint velocity, which was periodically changed.

Another test of the forward motor was performed by running the existing LabVIEW code and adjusting the duty cycle of the motor using the slider on the front panel. During this test the bike's front wheel was suspended midair so it could rotate freely. After the balancing algorithm had been created and connected to control the steering motor, it was tested by leaning the bike and examining if the steering motor rotated in the same direction. This purpose of the latter test was to insure the duty cycle of the steering motor followed the changing roll rate of the bike.

### 3.7.3 Testing the Gyroscope

To test that the gyroscope measured the correct roll rate, the bike was leaned to a predetermined angle. The roll rate which was measured during the time of the leaning was then integrated to calculate an approximate angle (not accounting for drift in the gyro), which could be compared with the predetermined angle the bike was tilted to.

### 3.7.4 Testing the Balancing Algorithm

After concluding that both motors worked separately, the balancing algorithm was able to be tested with both motors running. Two types of tests were performed, the first using a bike roller, and the second allowing the bike to move unaided. Both of these tests are described in the subsections below.

#### 3.7.4.1 Bike Roller Test

The bike roller test made it possible to replicate the effects of going forwards while not requiring a large amount of space. The test consisted of placing the bike on the roller and turning on both motors. For safety the bike was held, though this was done whilst trying to not manually interfere in its movement. The aim of this test was to examine if the motors worked in tandem, and if it seemed as the bike could potentially balance on its own. It should be noted that a large restricting factor of this test is the limited space the bike can move sideways before reaching the boundary of the roller.

#### 3.7.4.2 Unaided Test

To test the bike using an approach closer to a real world environment, a series of tests were performed outdoors in a flat and open area. This test made it possible to drive the bike forward, as well as allowing it to move sideways, mitigating the limitation of the previous bike roller test. To prevent the bike from falling and potentially being damaged, two people ran alongside the bike ready to catch in the case it would fall.

For the purpose of being able to subsequently analyze the bike's behavior, the tests where both filmed and recorded using the logging system presented in 3.6. Multiple tests were made with different values for the PID gains, to see how changing these values would affect the bikes performance.

The data which was chosen to be recorded, logged and plotted with MATLAB include the position and speed of the steering motor, the duty cycle sent to the steering motor, as well as the roll rate of the bike. The results of some of these tests are shown in section 4.3.

### 3.7.5 Testing the Program's Loop Times

While not being apart of the project's aim in itself, the performance of the program is of interest. This is because the bike's balancing could potentially be impacted if parts of the program takes too long to run. To be able to measure the performance, the time to run one iteration of each loop in the LabVIEW program was recorded. These times are then added together to get a total loop time for the program. The members of the previous project group have done a more in depth examination of this performance, which means that the values which are retrieved during the current testing can be compared with those values.

It should be noted that because a while-loop had to be used for the forward motor control, the time it takes to run cannot be measured. However, since a delay of one second exists in the loop, this is the loop time unless the other code in the loop takes longer to compute (which it does not). Additionally, the loop time of the forward motor control does not matter in this stage of the project, since the speed of the motor will be kept constant.

# 4

# Results

This chapter presents the results of the project. Primarily the result corresponding to the main aim of the project, meaning the bike's ability to balance, will be presented. The results correlating to the sub-goals: how the bike performs with regards to the steering motor, forward motor and balancing algorithm are also presented. Finally the reasons behind the results, and the effects of them are discussed. Further more detailed results from the individual tests are documented in appendix C.

## 4.1 Steering Motor

The steering motor, when viewed as a standalone component fully works as intended. That is, it is disabled until the *GO* button is pressed in LabVIEW, and then it can be controlled by changing the duty cycle between 10% and 90%. It also stops when the duty cycle is set to 50%, as expected. When the emergency stop is pressed either physically or in the software, the duty cycle is set to 50%, and the pin is disabled.

When connecting the control of the steering motor to the balancing algorithm and safety limit, the motor turns towards the same direction which the bike is leaning, and the duty cycle is set to 50% when the safety limits is reached. However, the steering wheel has the possibility of overshooting the safety limit because of the inertia of the steering wheel and handlebar. Whether this happens also depends on how fast it was rotating before the limit was reached.

The controls for the motors, separated into two boxes are depicted in figure 4.1 below. Inside of the left box, the steering motor controls are located. In these controls there exist inputs for changing each of the PID gains labeled P, I, and D. The box also has a field for changing the angle limit (in degrees from when the wheel is straight). Below the controls mentioned above are outputs from the steering motor encoder which shows its current position, velocity and acceleration. For the purpose of calibrating the encoder, there exists a button which resets the encoder position whenever it is pressed. At the bottom of the box, the duty cycle sent to the motor from the balancing algorithm is shown. At the bottom right of the image, under the boxes, is the *GO* button which starts the two motors.

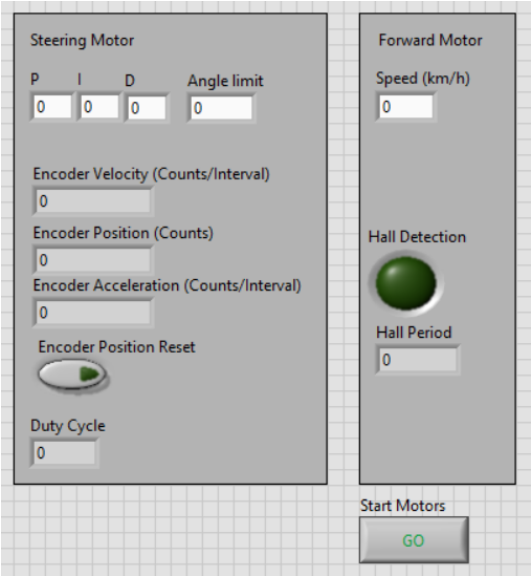



**Figure 4.1:** The motor controls in LabVIEW, with the steering motor control on the left side

## 4.2 Forward Motor

The speed of the forward motor can successfully be set using the LabVIEW front panel as seen on the right side of figure 4.1 above, and also changed while the program is running. The forward motor only starts when *GO* is pressed, and it stops when the emergency stop is pressed; the same behavior that the steering motor has.

The motor can not only be controlled by setting its speed directly, but also by changing its current. Controlling the motor using current can be accomplished if the function called by the *Call Library Function Node* is changed to `setCurrent`. Additional methods of controlling the motor can be added by creating and calling additional C functions based on the example by Vedder.

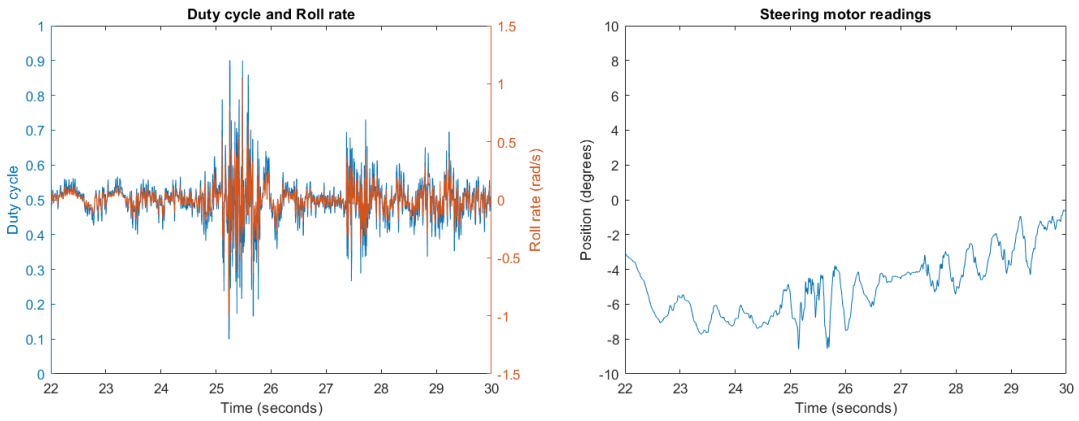
## 4.3 Balancing Algorithm

During the initial testing of the balancing algorithm, it was concluded that the steering motor behaved as expected in terms of rotation direction in relation to the roll rate of the bike. Meaning when the bike was leaned, the front wheel rotated towards the same direction.

When the bike was then placed on the roller, it seemed as if the bike could stay upright for some time before it had to be caught to prevent it from falling. The algorithm would make the front wheel rotate to the correct direction, but the amplitude of these rotations was gradually increasing until the border of the roller was reached. A real, conclusive result could not be made from this test; the reasons

for this are both because of human interference, as well as the narrow width of the roller.

The final and most reliable test of the balancing algorithm was done outdoors as described in 3.7.4.2, and the P gain was set to 5 (I and D were set to 0). The data from these tests, recorded with the logging system, are presented in the images below.




**(a)** The roll rate of the bike and duty cycle sent from the balancing algorithm.

**(b)** The position of steering motor during the test.

**Figure 4.2:** Plotted data from the final test.

In the test presented above the roll rate of the bike is relatively stable around 0. There only major deviance is around the 25:th second, which is caused by the bike passing over a bump in the road. As seen in blue line in figure 4.2a which represents the duty cycle, the rapid change in duty cycle also leads to a change by the control algorithm as a correction. The steering motor has some oscillations with and amplitude of about 2° and period of roughly one second. That is however nothing which had any noticeable effect on the overall performance; the test lasted for 8 seconds, and had to be aborted when the bike drove too close to obstacles in the testing area.

## 4.4 Loop Times

For the loop time tests all of the bikes hardware was enabled so that all control signal calculations would be made, and the resulting time would be as accurate possible. The total loop time was recorded as being between 2155 µs and 2420 µs. Compared to the previous project group's measurements, the new loop times are at worst is 240 µs slower than the worst case before the logging system and C algorithms were added [4].

## 4.5 Discussion

The main part of the result which should be discussed is that the forward motor control stopped working before the final test. Meaning that the UART commands sent to the VESC did not result in the forward motor moving. The reasons for this are unknown, but the problem has been isolated to the VESC, perhaps a broken trace related to the UART connections are the cause. During the final test the bike was instead accelerated by pushing it, before being let go. Pushing the bike should though not affect the overall result, compared to if the forward motor was still working.

The improvement of the development experience is one aim which was presented in 1.2, this aim is however not mentioned above. The reason being this decision it that it is difficult to assess if the aim has been reached or not, since it can be considered individual to a certain degree. What can be said is that all of the code which is used has been moved to a single location, which should make it easier to find the code. It should also now be much clearer what code is the most recent one. The code itself has been organized and separated into sub-VIs which should also make it easier to find the relevant code, this in combination with more comments should also make it easier to understand it.

What the result of the final test, and the data in figure 4.2 shows is that the bike can be kept stable by the balancing algorithm, even when the surface is uneven. There are some low frequency osculations of about the same frequency in all of the plotted data, which indicates that primarily the P gain of the PID controller could be tuned for better results.

There are some high frequency oscillations in the roll rate which might be caused by the IMU not being properly mounted, something which leads to vibrations; especially when the surface is uneven. Improper mounting of hardware can also be found elsewhere in the bike, which could be contributing to a worse balancing performance. These oscillations also creates high frequency changes in the duty cycle. Though this is generally not a problem since the steering wheel cannot change direction with such a high frequency; the dynamics of the bike could be described as acting as a low pass filter.

# 5

# Conclusion

The main goal of the project was to develop the software for the Autobike so that it could balance on its own whilst driving forward. In practice this meant completing two sub-goals: implementing an algorithm which, using the roll rate of the bike, could calculate the correct duty cycle for the steering motor, as well as writing software to encode a UART command which then could be used to control the forward motor.

As described in the results, the two sub-goals as well as the primary aim of the project were reached. The forward motor can be controlled from LabVIEW, and the steering motor is controlled by the balancing algorithm. The balancing algorithm can also successfully keep the bike stable, at least until the forward velocity becomes too low. What this minimum velocity is has not been determined, and will vary depending on what the gains of the PID controller are. Even if the some pieces of hardware were not properly mounted, this did not affect the performance in any meaningful way in this stage of the development. If the controller were to be tuned further, the hardware should probably be securely mounted so that the best possible foundation is achieved.

The secondary goal of the project, improving the future development experience, is as previously discussed difficult to judge if it has been reached. The wording "improving" does not contain the degree of which this should be done, meaning any small improvement could be interpreted as reaching the goal. However, continuously during the project's development, files have been organized and cleaned up; both for the benefit of the current developers, but more importantly for any future developers. The question of whether this goal has been reached or not is therefore best judged by these future developers.

## 5.1 Future work

The priority for any future work should be to fix forward motor. Even if this is not something which affected the results of this report, the speed might become more important in the future if the current algorithms are improved, and additional ones

are added. To accurately set the forward speed and to increase how far the bike can be driven, the forward motor would have to work.

Future work should also be focused on improving the hardware, and primarily its mounting. The IMU is the most important hardware component for the balancing algorithm, and will therefore have to be mounted so that its axis aligns with the axis of the bike. It also has to be mounted in such a way that it does not move around or vibrate excessively.

# Bibliography


[1] Argo AI, *Autonomous Vehicle Guidelines for Safe Driving Around Cyclists*, Dec. 2021. [Online]. Available: https://www.argo.ai/wp-content/uploads/2021/12/ArgoAI_BikeGuidelines-WithCopyright.pdf.

[2] National Instruments, *What Is LabVIEW?* Apr. 2022. [Online]. Available: https://www.ni.com/sv-se/shop/labview.html.

[3] ——, *myRIO-1900*, Apr. 2022. [Online]. Available: https://www.ni.com/sv-se/support/model.myrio-1900.html.

[4] V. Aronsson Karlsson, L. Bridén, M. Zawari, and G. Sherif, "PROJECT AUTOBIKE," Ph.D. dissertation, Mälardalen University, Västerås, Aug. 2022.

[5] H. A. Andrade and S. Kovner, "Software synthesis from dataflow models for G and LabVIEW TM," in *Conference Record of the Asilomar Conference on Signals, Systems and Computers*, vol. 2, 1998, pp. 1705–1709. DOI: 10.1109/acssc.1998.751616. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=751616.

[6] National Instruments, *Benefits of Programming Graphically in LabVIEW*, 2022. [Online]. Available: https://www.ni.com/sv-se/innovations/white-papers/13/benefits-of-programming-graphically-in-ni-labview.html.

[7] National instruments, *Call Library Function Node*, Mar. 2018. [Online]. Available: https://zone.ni.com/reference/en-XX/help/371361R-01/glang/call_library_function/.

[8] National Instruments, *Creating Shared Library For LabVIEW Real-Time or VeriStand on NI Linux RT Target*, Oct. 2021. [Online]. Available: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000YGNdCAO&l=sv-SE.

[9] M. Trochimiuk, *FPGA programming—how it works and where it can be used*, Apr. 2021. [Online]. Available: https://codilime.com/blog/fpga-programming-how-it-works-and-where-it-can-be-used/.

[10] National Instruments, *The LabVIEW RIO Architecture: A Foundation for Innovation*, Oct. 2020. [Online]. Available: https://www.ni.com/sv-se/innovations/white-papers/13/the-labview-rio-architecture--a-foundation-for-innovation.html.

[11] ——, *From Student to Engineer: Preparing Future Innovators With the NI LabVIEW RIO Architecture - NI*, Feb. 2022. [Online]. Available: https://

www.ni.com/sv-se/innovations/white-papers/14/from-student-to-engineer--preparing-future-innovators-with-the-n.html.

[12] Maxon, *ESCON 50/5 Hardware Reference*, Aug. 2021. [Online]. Available: https://www.maxongroup.com/medias/sys_master/root/8930313371678/409510-ESCON-50-5-Hardware-Reference-En.pdf.

[13] ESCON, *ESCON Overview*, Apr. 2014. [Online]. Available: https://docs.rs-online.com/ed0a/0900766b81376a21.pdf.

[14] Avago Technologies, *HEDM-55xx/560x & HEDS-55xx/56xx*, Nov. 2014.

[15] Digilent, "Pmod NAV Reference Manual," Jan. 2017. [Online]. Available: https://digilent.com/reference/_media/reference/pmod/pmodnav/pmod_nav_rm.pdf.

[16] STMicroelectronics, *iNEMO inertial module: 3D accelerometer, 3D gyroscope, 3D magnetometer*, Mar. 2015. [Online]. Available: https://www.st.com/resource/en/datasheet/lsm9ds1.pdf.

[17] M. Grace Legaspi and E. Peňa, "UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter," *Analog Dialogue*, vol. 54, no. 4, 2020. [Online]. Available: https://www.analog.com/en/analog-dialogue/articles/uart-a-hardware-communication-protocol.html.

[18] B. Vedder, *A custom BLDC motor controller (a custom ESC)*, 2014. [Online]. Available: http://vedder.se/2014/01/a-custom-bldc-motor-controller/.

[19] ——, *VESC – Open Source ESC*, Jan. 2016. [Online]. Available: http://vedder.se/2015/01/vesc-open-source-esc/.

[20] ——, *Communicating with the VESC using UART*, Oct. 2015. [Online]. Available: http://vedder.se/2015/10/communicating-with-the-vesc-using-uart/.

[21] Technopedia, *What is Cyclic Redundancy Check (CRC)?* Sep. 2020. [Online]. Available: https://www.techopedia.com/definition/1793/cyclic-redundancy-check-crc.

[22] B. Vedder, *vedderb/bldc_uart_comm_stm32f4_discovery: A project that demonstrates how to communicate between the VESC and a STM32F4 discovery board using UART*, Dec. 2018. [Online]. Available: https://github.com/vedderb/bldc_uart_comm_stm32f4_discovery.

[23] National Instruments, *Getting Started with C/C++ Development Tools for NI Linux Real-Time, Eclipse Edition*, Jan. 2021. [Online]. Available: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000YHR7CAO&l=sv-SE.

[24] C. J, *NI Linux Real-Time Cross Compiling: Using the NI Linux Real-Time Cross Compile Toolchain with Visual Studio Code*, Mar. 2020. [Online]. Available: https://forums.ni.com/t5/NI-Linux-Real-Time-Documents/NI-Linux-Real-Time-Cross-Compiling-Using-the-NI-Linux-Real-Time/ta-p/4026449.

[25] G. Van Liew, eli, J. Reid, A. Wang, Pine, M. Bierner, J. Sola, F. Bluemle, and E. Rashedi, *C++ programming with Visual Studio Code*, May 2022. [Online]. Available: https://code.visualstudio.com/docs/languages/cpp.

[26] Kitware, *CMake*, 2022. [Online]. Available: https://cmake.org/.

[27] J. N. Hasse, N. Weber, E. Martin, G. Costa, and T. Ikuta, *Ninja, a small build system with a focus on speed*, 2022. [Online]. Available: https://ninja-build.org/.

[28] FileZilla, *Client Features*, 2022. [Online]. Available: https://filezilla-project.org/client_features.php.

[29] I. Beavers, "The Case of the Misguided Gyro," *Analog Dialogue*, vol. 51, no. 3, Mar. 2017. [Online]. Available: https://www.analog.com/en/analog-dialogue/raqs/raq-issue-139.html.

[30] Shimano, *DU-E6010*, 2022. [Online]. Available: https://bike.shimano.com/en-EU/product/component/citytrek-ebike-e6000/DU-E6010.html.

[31] National Instruments, *The NI TDMS File Format*, May 2022. [Online]. Available: https://www.ni.com/sv-se/support/documentation/supplemental/06/the-ni-tdms-file-format.html.

# A

# LabVIEW User Manual

This appendix goes through the main LabVIEW program, and explains how it should be used from a user's perspective. The program is launched by opening project found under `labview/red_bike.lvproj` in LabVIEW 2019. Later versions might work as well, but the 2019 version is recommended for the best compatibility. Inside the project, open `myRIO-1900/bin/TestMain.vi`. The first time the program is opened on a new computer, LabVIEW might give a warning that a `.lvbitx` file could not be found. This file is located in `labviw/bin`, and can manually be pointed to when requested.

With the program opened, the controls which can be interacted with to control be bike are located in the top left of the front panel. These controls are shown in image A.1 below.

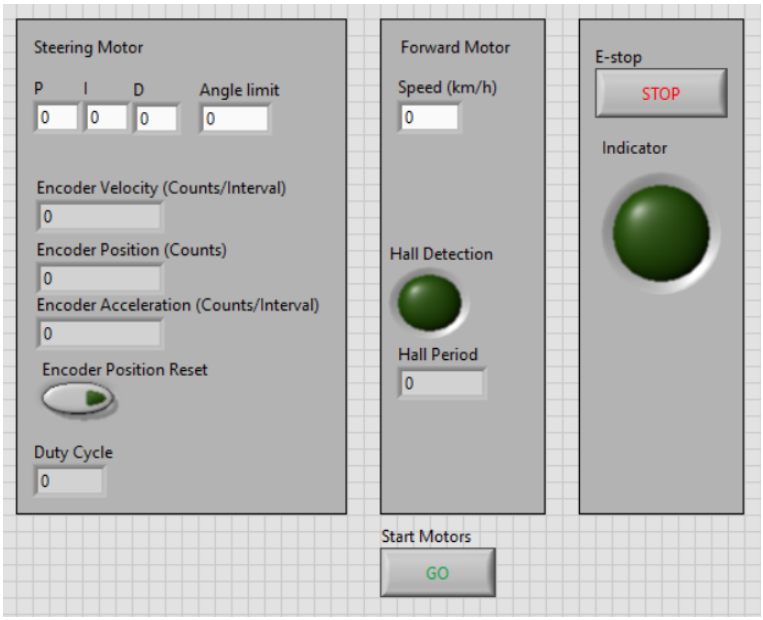



**Figure A.1:** The controls which can be interacted with in the LabVIEW front panel.

In the leftmost box in the picture contains the controls for the steering motor. Here, the PID gains and the angle limit (in degrees from the center), can be changed. Below this are the readouts from the encoder, and a button to reset its position. This button should be pressed when the front wheel is straight to calibrate the encoder before starting any of the motors. At the bottom of the box is the duty cycle which is sent from the balancing algorithm to the steering motor.

The middle box at the center of the image contains the controls for the forward motor. The only thing which can be changed here is the RPM, note that the RPM are measured in relation with the pedals (for the red bike). Below the forward motor controls is a button which starts the two motors. It should be mentioned that the program can still be running without affecting the motors, if this button is not pressed.

At the right is a software implementation of the emergency stop. From the program's perspective, this acts exactly as if the hardware emergency stop was to be pressed.

The path to the log file (including its desired name) can be specified in the text field which is located at the absolute bottom of the image. The path leads to a file **on the MyRIO** where the LabVIEW program is running, and must be inside the `/home/lvuser/` directory. This is due to the program not having *sudo* privileges in the Linux OS which is running on the device. If no file exists in the specified path, a file will be created. If a file does exist, it will be replaced and overwritten when the program starts. It is therefore recommended to change the file name before each time the program is to be run. To open the log files, they have to be downloaded from the MyRIO using for example FileZilla, and then opened using any of the methods mentioned in 3.6.

The rest of the front panel consists of the values from the various sensors. On the right of the controls depicted in the figure above are three graphs illustrating the values from the IMU's gyroscope. The topmost graph corresponds with the roll rate which is used by the balancing algorithm. The front panel also shows GPS data and accelerometer values, these are however not used by any part of the program at the moment.

In summary, the following steps must be performed to start the bike and log its sensor and control signal data:

1. Set the PID gains, and angle limit and RPM to something non-zero

2. Set the path to the logging file to somewhere inside the `/home/lvuser/` directory. For example `/home/lvuser/log1.tdms`

3. Start the LabVIEW program

4. Reset the encoder position when steering wheel is straight

5. Press *GO*

Note that, to be able to start the LabVIEW program, the hardware emergency stop has to be up, and the hardware reset button has to be pressed. The program is ready when the light next to the reset button shines green.

# B

# Code

This Appendix contains some of the code which have been used on the bike, or during the project. Some of the code, for example the main LabVIEW program, is difficult to include in the appendix, but can be found on the GitHub repository together with all of the other pieces of code at the following URL: `https://github.com/Hannnes1/autobike`

## B.1 Balancing Algorithm

```
1  // balancing.h
2
3  #ifndef _BALANCING_H_
4  #define _BALANCING_H_
5
6  #define gearRatio 111.0
7  #define pi 3.141592
8
9  #define windupGuard 6
10
11 double integral = 0;
12 double derivative = 0;
13
14 double previousError = 0;
15
16 extern double stabilizeBike(double rollRate, double Kp, double Ki,
       double Kd);
17
18 double calculateSteeringPWM(double angularVelocity);
19
20 double pid(double reference, double currentValue, double Kp, double
       Ki, double Kd);
21
22 #endif
```

```
// balancing.c

#include "balancing.h"

extern double stabilizeBike(double rollRate, double Kp, double Ki, double Kd) {
    double steeringRate = pid(0, rollRate, Kp, Ki, Kd);

    // Send Steering Rate Reference value to steering motor controller
    double steeringPWM = calculateSteeringPWM(steeringRate);

    return steeringPWM;
}

// Take in the wanted angular velocity of the handlebar,
// And return the required duty cycle for that velocity.
double calculateSteeringPWM(double angularVelocity) {
    // Convert from angular velocity (rad/s) of the handlebar,
    // to rpm of the motor.
    double rpm = -angularVelocity * 30 / pi * gearRatio;

    // Convert from rpm to duty cycle,
    // 4000 is the maximum speed of the motor (configured in Escon Studio).
    return 50 + rpm * 40.0 / 4000.0;
}

double pid(double reference, double currentValue, double Kp, double Ki, double Kd) {
    double error = reference - currentValue;

    integral += error;
    derivative = error - previousError;

    if (integral < -windupGuard) {
       integral = -windupGuard;
    } else if (integral > windupGuard) {
        integral = windupGuard;
    }

    previousError = error;

    return Kp * error + Ki * integral + Kd * derivative;
}
```

## B.2 VESC UART Encoder

```
// common.h

#ifndef _COMMON_H_
#define _COMMON_H_

typedef unsigned char uint8_t;
```

```
typedef unsigned   uint32_t;
typedef short  int16_t;
typedef unsigned short  uint16_t;

unsigned short crc16(unsigned char *buf, unsigned int len);

#endif
```

```
// common.c

#include "common.h"

// CRC Table.
const unsigned short crc16_tab[] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};

// Calculates checksum bytes, based on payload and crc16_tab above.
unsigned short crc16(unsigned char *buf, unsigned int len) {
    unsigned int i;
    unsigned short cksum = 0;
    for (i = 0; i < len; i++) {
        cksum = crc16_tab[(((cksum >> 8) ^ *buf++) & 0xFF)] ^ (
           cksum << 8);
```

```
47      }
48      return cksum;
49  }
```

```c
// vescEncoder.h

#ifndef _VESC_H_
#define _VESC_H_

#include "common.h"

unsigned short crc16(unsigned char *buf, unsigned int len);

extern void sendAlive(uint8_t *array);

extern int setRpm(uint8_t* array, int rpm);

extern int setCurrent(uint8_t *array, int current);

void bufferAppendFloat32(uint8_t* buffer, float number, float scale
    , int *index);

void bufferAppendInt32(uint8_t* buffer, int number, int *index);

uint8_t *buildPacket(uint8_t *data, unsigned int len);

uint8_t txBuffer[512 + 6];

#endif
```

```c
// vescEncoder.c

#include "vescEncoder.h"
#include "common.h"

#include <string.h>

static unsigned char buffer[1024];

// Calculates command used to keep the motor alive.
extern void sendAlive(uint8_t *array) {
    int index = 0;

    buffer[index++] = 30; // Number correspondning to the keepAlive
        command.

    // Build the packet for the command.
    uint8_t *packet = buildPacket(buffer, index);

    // Copy calculated packet to the referance array.
        for (uint8_t i = 0; i < 10; i++) {
        array[i] = packet[i];
    }
}

// Calculates command used to set the current of the motor.
// Based on the given parameter "current".
extern int setCurrent(uint8_t *array, int current) {
        int index = 0;
```

```
29
30          buffer[index++] = 6; // Number correspondning to the
                setCurrent command.
31
32      // Rescale the current and swap its endianness.
33          bufferAppendFloat32(buffer, (float) current, 1000.0, &index
                );
34
35      // Build the packet for the command.
36      uint8_t *packet = buildPacket(buffer, index);
37
38      // Copy the calculated packet unto the referance array.
39          for (uint8_t i = 0; i < 10; i++) {
40          array[i] = packet[i];
41      }
42          return current;
43  }
44
45  // Calculates command used to set the rpm of the motor.
46  // Based on the given parameter "rpm".
47  extern int setRpm(uint8_t *array, int rpm) {
48      int index = 0;
49
50      rpm *= 250; // Convert to the actual RPM of the bike's pedals.
51
52      buffer[index++] = 8; // Number correspondning to the setRpm
            command.
53
54      // Swpan the endianness of the rpm.
55      bufferAppendInt32(buffer, rpm, &index);
56
57      // Build the packet for the command.
58      uint8_t *packet = buildPacket(buffer, index);
59
60      // Copy the calculated packet unto the referance array.
61      for (uint8_t i = 0; i < 10; i++) {
62          array[i] = packet[i];
63      }
64      return rpm;
65  }
66
67  // Builds the packet based on the given command and payload.
68  // See http://vedder.se/2015/10/communicating-with-the-vesc-using-
        uart/ for more information.
69  uint8_t *buildPacket(uint8_t *data, unsigned int len) {
70      int bufferIndex = 0;
71
72      if (len <= 256) {
73          txBuffer[bufferIndex++] = 2;
74          txBuffer[bufferIndex++] = len;
75      } else {
76          txBuffer[bufferIndex++] = 3;
77          txBuffer[bufferIndex++] = len >> 8;
78          txBuffer[bufferIndex++] = len & 0xFF;
79      }
80
```

```
81      memcpy(txBuffer + bufferIndex, data, len);
82      bufferIndex += len;
83
84      unsigned short crc = crc16(data, len);
85      txBuffer[bufferIndex++] = (uint8_t)(crc >> 8);
86      txBuffer[bufferIndex++] = (uint8_t)(crc & 0xFF);
87      txBuffer[bufferIndex++] = 3;
88
89      return txBuffer;
90  }
91
92  // Rescales the given "number" with "scale", before swapping its
        endianness with "bufferAppendInt32".
93  void bufferAppendFloat32(uint8_t* buffer, float number, float scale
        , int *index) {
94      bufferAppendInt32(buffer, (int)(number * scale), index);
95  }
96
97  // Swaps the endianness of the given "number".
98  void bufferAppendInt32(uint8_t *buffer, int number, int *index) {
99      buffer[(*index)++] = number >> 24;
100     buffer[(*index)++] = number >> 16;
101     buffer[(*index)++] = number >> 8;
102     buffer[(*index)++] = number;
103 }
```

## B.3 MATLAB Test Data Plotting

```
% To plot the logged data, change the file path to the correct log
% file. It might also be necessary to uncomment / comment some
% parts of the code, depending on what you want to plot. The xlim
% and ylim of some plots will also have to be adjusted.

% Read all of the data in the TDMS file.
data = tdmsread("Logs/testLog.tdms");

%% Encoder

% ‘data‘ contains one cell per group. Each cell contains
% a table with one column per channel. In this case, the
% first channel in the first group contains the time,
% and the second channel contains the position etc.
time1 = table2array(data{1,1}(:,1));
position = table2array(data{1,1}(:,2));
velocity = table2array(data{1,1}(:,3));

figure;
%yyaxis left;
% 611 is the conversion ratio from counts on the encoder
% to degrees.
plot(time1-time1(1),position/611);
ylim([-10,10]);
ylabel("Position (degrees)");
hold on;
%yyaxis right;
%plot(time1-time1(1),velocity/611*1000);
%ylim([-50,50]);
%ylabel("Velocity (degrees/s)");
xlabel("Time (seconds)");
xlim([22,30]);
title("Steering motor readings");
hold off;

%% Gyro
time2 = table2array(data{1,2}(:,1));
X = table2array(data{1,2}(:,2));
Y = table2array(data{1,2}(:,3));
Z = table2array(data{1,2}(:,4));

figure;
hold on;
plot(time2-time2(1),X);
plot(time2-time2(1),Y);
plot(time2-time2(1),Z);
ylabel("Angular velocity (rad/s)");
xlabel("Time (seconds)");
title("Gyro");
%yyaxis right;

angle = zeros(length(Z),1);
for i = 1:length(Z) - 1
    angle(i + 1) = angle(i) + (time2(i+1) - time2(i)) * Z(i);
```

```
55 end
56 %plot(time2-time2(1),angle);
57 %ylabel("Lean angle (radians)");
58 legend("X", "Y", "Z", "Lean angle");
59
60 %% Steering motor
61
62 time3 = table2array(data{1,3}(:,1));
63 pwm = table2array(data{1,3}(:,2));
64
65 figure;
66 plot(time3-time3(1),pwm);
67 ylim([0,1]);
68 ylabel("Duty cycle");
69 xlabel("Time (seconds)");
70 title("Steering motor control");
71
72 %% Combined
73
74 figure;
75 yyaxis left;
76 plot(time3-time2(1),pwm);
77 ylim([0,1]);
78 ylabel("Duty cycle");
79 hold on;
80 yyaxis right;
81 plot(time2-time2(1),Z);
82 ylabel("Roll rate (rad/s)");
83 ylim([-1.5,1.5]);
84 xlim([22,30]);
85 xlabel("Time (seconds)");
86 title("Duty cycle and Roll rate");
87
88 %% PWM vs angular velocity
89
90 pwm01 = tdmsread("Logs/pwm01.tdms");
91 pwm02 = tdmsread("Logs/pwm02.tdms");
92 pwm03 = tdmsread("Logs/pwm03.tdms");
93 pwm04 = tdmsread("Logs/pwm04.tdms");
94
95 time01 = table2array(pwm01{1,1}(:,1));
96 velocity01 = table2array(pwm01{1,1}(:,3));
97 position01 = table2array(pwm01{1,1}(:,2));
98
99 time02 = table2array(pwm02{1,1}(:,1));
100 velocity02 = table2array(pwm02{1,1}(:,3));
101
102 time03 = table2array(pwm03{1,1}(:,1));
103 velocity03 = table2array(pwm03{1,1}(:,3));
104
105 time04 = table2array(pwm04{1,1}(:,1));
106 velocity04 = table2array(pwm04{1,1}(:,3));
107
108 %%
109
110 figure;
```

```
111 hold on;
112 plot(time01 - time01(1) - 1.56,velocity01/611*1000);
113 plot(time02 - time02(1) - 0.9,velocity02/611*1000);
114 plot(time03 - time03(1) - 1.59,velocity03/611*1000);
115 plot(time04 - time04(1) - 0.78,velocity04/611*1000);
116 xlabel("Time (seconds)");
117 ylabel("Velocity (degrees/second)");
118 xlim([-1,9]);
119 legend("duty cycle = 0.1", "duty cycle = 0.2", "duty cycle = 0.3",
       "duty cycle = 0.4", 'Location','northwest');
120 title("Relation between duty cycle and angular velocity");
```

# C

# Testing protocol

The following appendix documents tests that have been performed during the project. They are documented in such a way that they can be repeated to validate the functionality of the balancing algorithm on either this or another version of the Autobike in the future.

## C.1 Angular velocity of the Forward Motor Follows the Setpoint Value

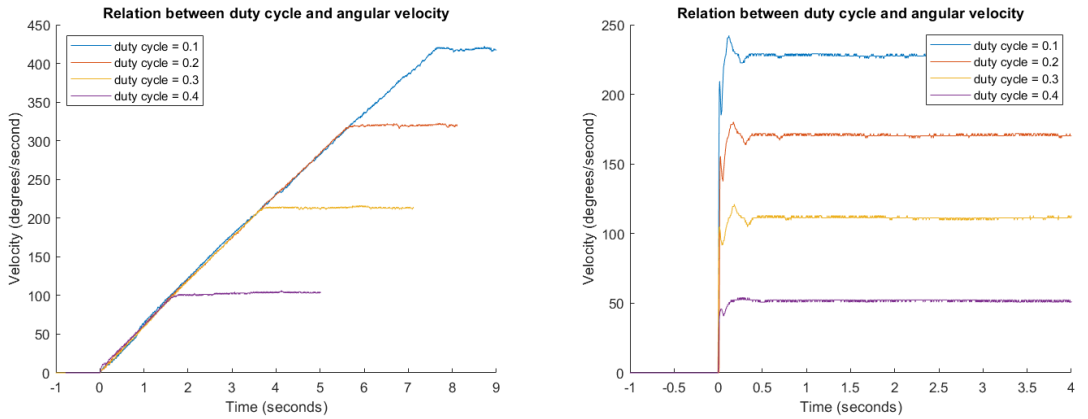
To test that the Angular velocity of the forward motor follows the setpoint value, the *Auto Tuning* feature in ESCON Studio was used. The results presented in the image below are that the actual speed (red line) follows the setpoint speed (blue line), with the exception of the initial acceleration and deceleration.


**Figure C.1:** The results page after running the Auto Tuning in ESCON Studio.

## C.2 Duty Cycle Compared With the Angular Velocity of the Front Wheel

In this test, the bike was lifted into the air and the balancing algorithm was bypassed so that the duty cycle sent to the steering motor could be controlled directly. The angle limit was also bypassed, so that the motor could spin freely several revolutions. The duty cycle was then set to 0.4, and the velocity of the steering motor was recorded. This velocity is also the speed that the bikes handlebar turns with, since the gear ratio between the handlebar and the motor is 1:1. The test was then repeated, but with the duty cycle set to 0.3, 0.2, and 0.1.

The initial result from this test was that the motor took 8 seconds to accelerate from stationary to the maximum velocity as seen in image C.2a. This was later fixed by uploading the ESCON configuration from the black bike as described in 3.3.1. The fixed result is plotted in C.2b.




**(a)** Duty cycle vs velocity with the wrong motor configuration.

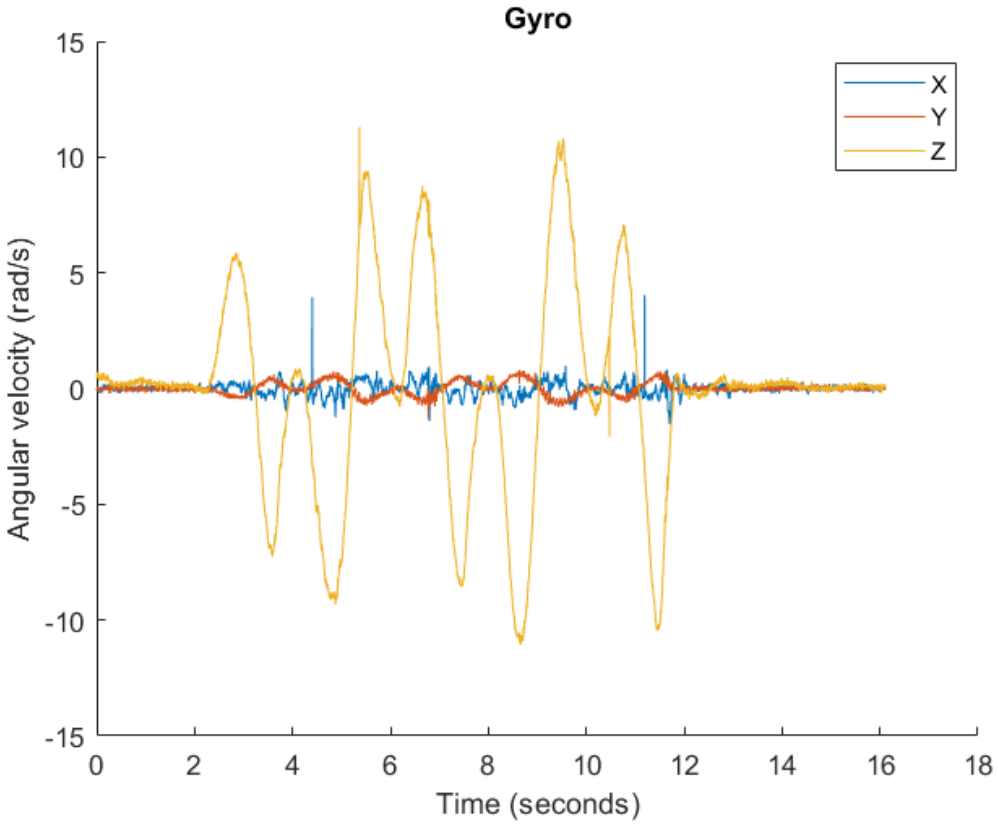
**(b)** The position and velocity of steering motor.

**Figure C.2:** Test results of varying duty cycles compared with the angular velocity of the steering motor.

## C.3 Gyroscope Measurements When Tilting the Bike

This test was performed by tilting the bike back and forth. While logging the data from the gyroscope for each axis. Before performing these tests, the mount for IMU had been fixed so that the IMU was no longer placed at an angle. The data from the test is illustrated in the figure below.

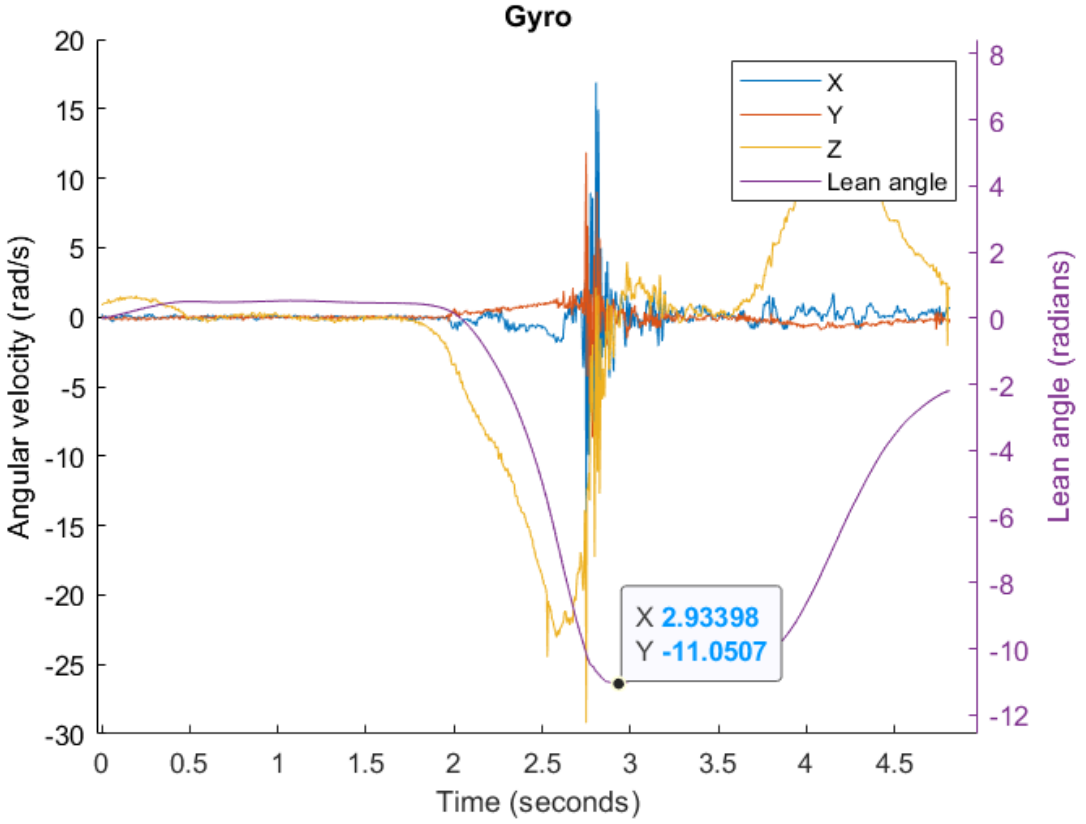
What can be seen is that the Z-axis is the axis which is affected the most. This is also the axis which should be affected by tilting the bike side to side. It can also be seen that the Y-axis has some very minor movements, which is because the current mount is not very exact, and it is difficult to precisely adjust the placement angle.



## C.4 Gyroscope and Position

This test was performed by leaning the bike towards a wall and measuring the angle the electronics box has when it touches the wall; in this case the angle was 20°. The bike was then leaned from an upright position until it hit the wall, whilst the data from the gyroscope for each axis was logged. The data from this test is illustrated in the figure below, together with the integral of the gyroscope's data for the Z-axis, which corresponds with how much the bike has been tilted.

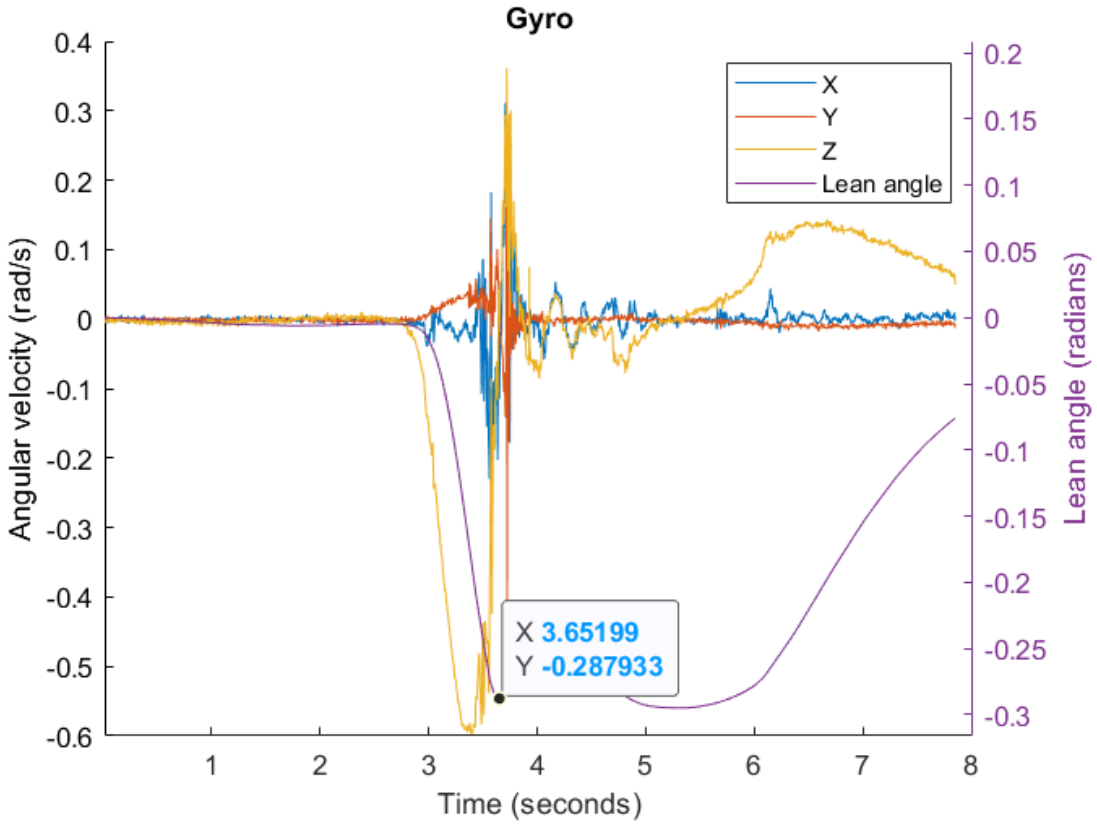
The maximum angle of the bike is recorded as -11 radians, when it should have been $20 * \frac{\pi}{180} \approx 0.35$ radians. The conclusion which can be made from this is that the gyro recordings are wrong with a factor of $\frac{11}{0.35} = 31.4$. The reason for this could not be found, so to compensate for the error, all readings were divided by 31.4 inside of the LabVIEW program before being used in any calculations.



### C.4.1 After Multiplying With the Calculated Factor

The next test was performed after the factor which compensates for the error was added. The test was otherwise performed exactly as the previous test, the only difference being the measured angle was 16°. The data from the test is shown in the figure below.

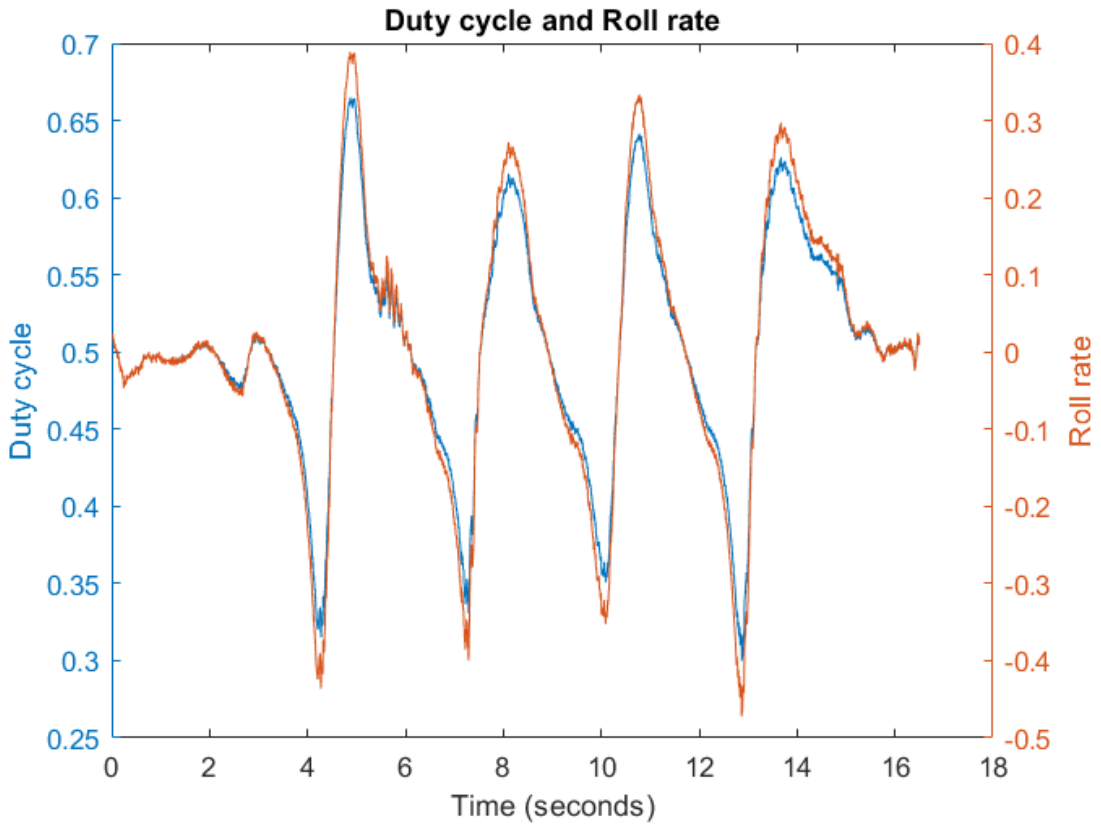
This graph shows that the maximum angle is -0.288 radians, or 16.5°. The values of the angular velocities is also more reasonable. All future tests presented below have also been performed with the the correcting factor of 31.4.



## C.5 Duty Cycle Compared With the Gyroscope's Measurements

This test was performed by allowing the bike to naturally fall a few centimeters, before catching it. The roll rate (data from the Z-axis of the gyroscope), together with the duty cycle given by the balancing algorithm is displayed in the figure below. During this test, the P gain of the balancing controller was set to 1.

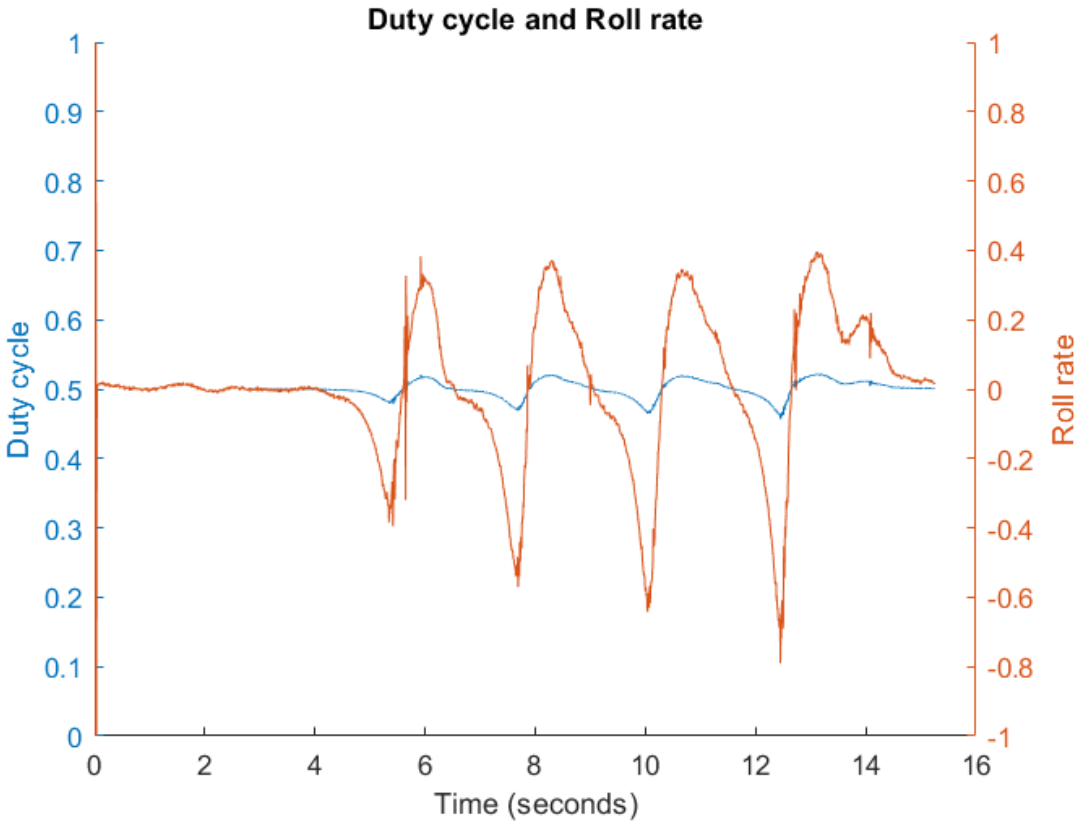
The result from this test is that the duty cycle follows the roll rate almost perfectly. It also never reaches the upper and lower limits of 0.9 and 0.1 respectively.



### C.5.1 After Adjusting the Balancing Algorithm

This test was performed exactly as the previous one. The difference being the balancing algorithm had been changed so that the maximum duty cycle of 0.9 is only reached when the roll rate multiplied with the P gain is the same as the steering motor's maximum angular velocity, which is set to 4000 RPM in ESCON Studio. This is how the algorithm is indented to work. The upcoming figure illustrates the data from the test.

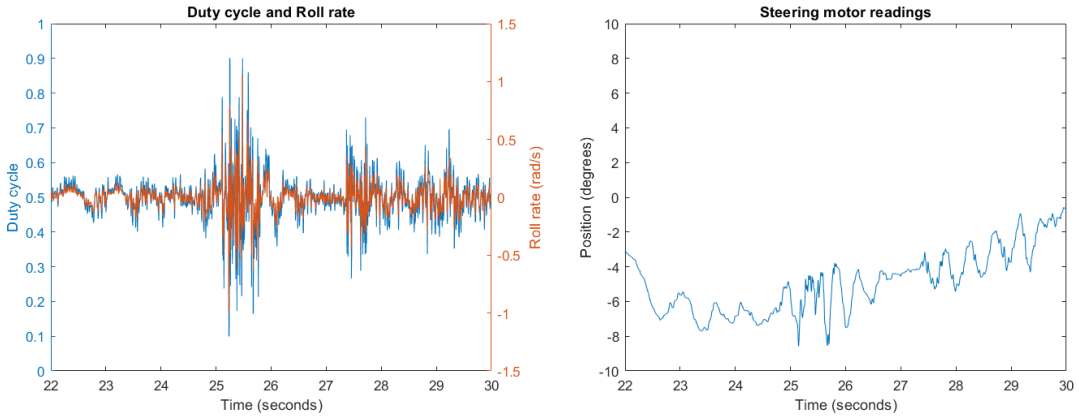
In the data below, it can be seen that the duty cycle is much lower for the same roll rates. This can be affected by increasing the P gain of the balancing controller.



## C.6 Unaided Outdoor Test

When performing the final tests outdoors in an open area, the motor controller for the forward motor (the VESC) was broken (this is further discussed in section 4.5), so the bike had to be pushed by hand. The test was started by enabling the steering motor, pushing the bike up to speed and then letting it go. The P gain of the balancing algorithm was initially set to 1. This resulted in the bike having close to no control response, which lead to it falling over and having to be caught within a second after letting go.

The P gain was then increased to 5, and the test was repeated. The results of this can be seen in figure C.3. Here it is shown that the roll rate as well as the duty cycle is close to their center throughout the test, except for around the 25:th second when the bike drove over a bump. It can also be seen that the duty cycle reacts to the roll rate and that the steering motor follows the duty cycle.




**(a)** Duty cycle and the roll rate of the bike. **(b)** The position of the forward motor.

**Figure C.3:** The final test results of running the bike outside on an open area.

**DEPARTMENT OF ELECTRICAL ENGINEERING**
**CHALMERS UNIVERSITY OF TECHNOLOGY**
Gothenburg, Sweden
www.chalmers.se


CHALMERS