Development and implementation of an automatic brake system for the DTU Roadrunner

The report is written by:
Christian Hans Marker Johansen S132319

guidance counselor(s):
Henning Si Høj, DTU
Jens Christian Andersen, DTU

DTU Elektro
Automation and Control
Danmarks Tekniske Universitet
Elektrovej
2800 Kgs. Lyngby
Danmark
Tlf. : +45 4525 3576

studieadministration@elektro.dtu.dk

Project period:	01/02-2018 - 12/06-2018
ECTS:	10
Education:	Graduate
Program:	Electrical Engineering
Class:	Public
Remarks:	This report is submitted as partial fulfilment of the requirements for graduation in the above education at the Technical University of Denmark.
Copyrights:	

Technical University of Denmark

Christian Hans Marker Johansen *S132319*

Development and implementation of an automatic brake system for the DTU Roadrunner

Synthesis project, June - 2018

DTU Electrical Engineering
Department of Electrical Engineering

Abstract

The goal for the project described in this rapport is to design and implement a automatic brake for the DTU roadrunner car. To achieve this, mechanical calculations has been performed to find the best possible actuator for the brake, a control code has been made in C++ and tests have been performed to ensure that the brake work as intended.

Preface

The project is developed in cooperation with DTU Roadrunner.

Special thanks to:
Henning Si Høj for help with questions regarding software and administrative work.
Claus Suldrup Nielsen for help with questions regarding mechanics and the mounting of the actuator in the car.

Contents

1 System description **4**
1.1 Mechanical system 4
1.2 Software 5

2 Brake actuator calculations and considerations **5**
2.1 Calculations 5
2.2 Actuator 7

3 Simulink model **7**

4 Code **8**
4.1 Brake node 9
4.1.1 BrakeSystem.h file 9
4.1.2 BrakeSystem.cpp file 10
4.1.3 brake_node.cpp file 11
4.2 Position controller 13

5 Test **14**

6 Future improvements **17**

7 Conclusion **17**

Litteratur **18**

Figurer **19**

1 System description

The autonomous brake system consists of two parts. A software part which controls a mechanical part.

1.1 Mechanical system

The DTU Roadrunner brakes by using a hydraulic brake system. This system consists of a brake pedal, which is connected to a piston, which is housed in a cylinder. This cylinder is connected to a brake caliber, which house some brake pads, via a pipe, containing an as close as possible airless fluid. It works in such a way, that when the pedal is pressed, the piston is moved further into the cylinder, creating less room for the fluid, which is moved out of the cylinder. This way the fluid is moved in the system, but since the system is closed, pressure will increase in the system, at some point. The pressure moves the brake pads inside the brake calibers. These pads are pressed against the brake discs on the wheels. This makes the car loose speed and finally stop.
On figure 1.1.1 an illustration of the above mechanics is shown.
To brake the car automatically a electric actuator is inserted to press the pedal. On figure 1.1.2 the mounted brake can be seen. Since it must be possible for the driver to press the pedal at all times, the actuator has been placed at the side of the pedal. By using a hook, mounted on the pedal, the actuator can press the pedal without getting in the way of the driver.

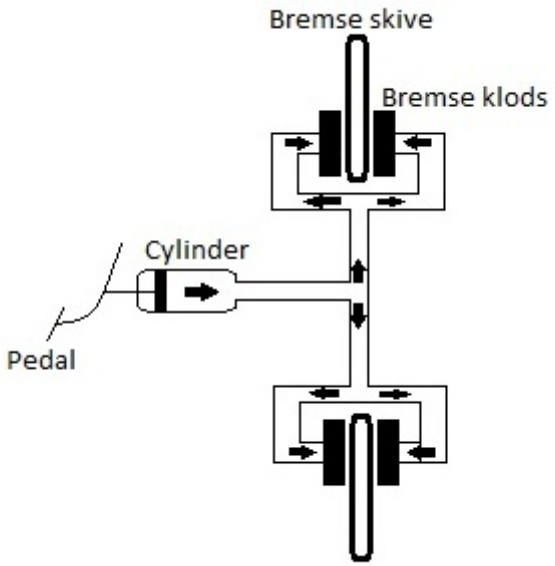

Figure 1.1.1: Illustration of a hydraulic brake

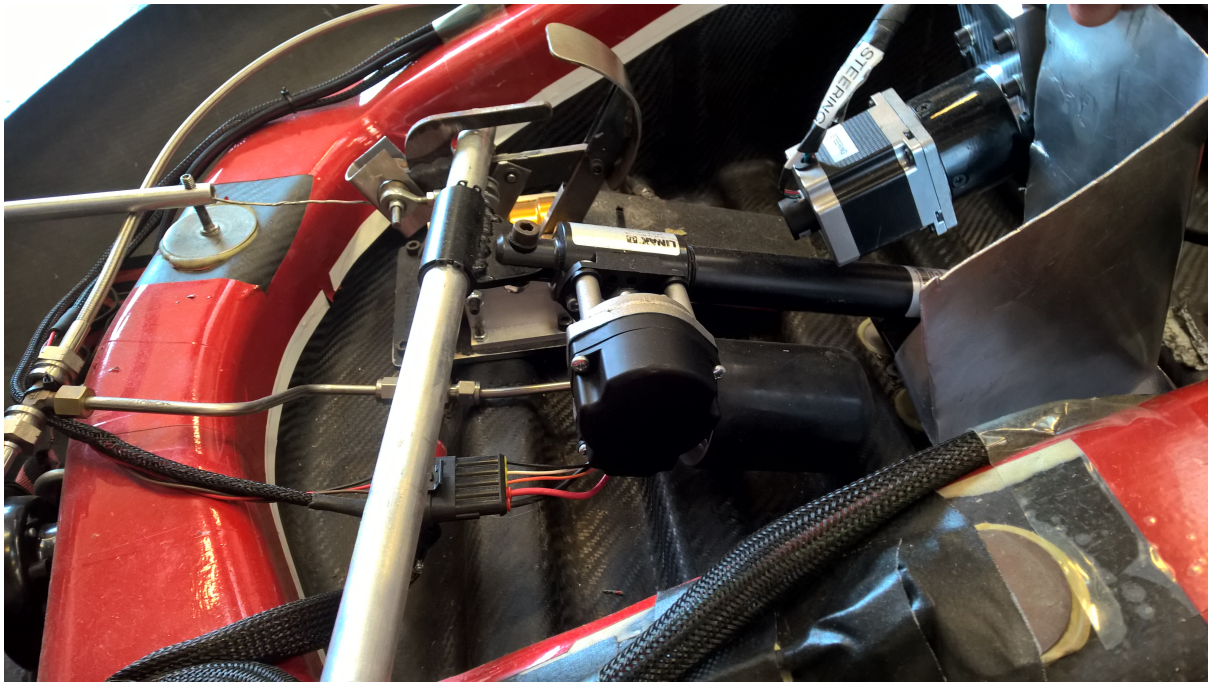

Figure 1.1.2: Picture of the mounted brake actuator

1.2 Software

The software for an autonomous system needs to control all parts of a bigger mechanical system. It needs to read inputs, analyse them and control the necessary actuators based on the analysed inputs. By separating the code into separate blocks it is easier to maintain, develop and control. A coding tool called ROS is used to easily do this in C++. The tool consists of several libraries which offer the wanted functionality, to have separate code blocks which can communicate variables between each other. ROS functions by creating a Topic message Type that the different nodes can advertise and/or subscribe to. This allows all nodes to be able to share needed information. The code blocks are called nodes, in the ROS language. For the autonomous system for the Roadrunner, each node controls a specific function of the car and a node called navigation can then call for the specific node, with the function it needs, for the operation it wants. Some nodes gather inputs from sensors and advertise them for the rest of the nodes, others control actuators based on some information they subscribe to from other nodes. The control software developed for the autonomous brake is one of the latter.

2 Brake actuator calculations and considerations

To make sure that the right actuator is bought, some calculations on the cars brake system has to be made. Specifically for this project it is important that the actuator is strong enough and fast enough.

2.1 Calculations

To find the best possible actuator the needed strength of the actuator is calculated as followed. First some constants are established.

Tire road friction coefficient $\alpha = 0.4$

Gravitational constant $g = 9.80665$

Weight pr. wheel $mh = 50$

Dynamic friction of the brake $mybk = 0.37$

Caliber piston movement $Lk = 0.0015$

Caliber piston radius $Dk = 0.032/2 = 0.0160$

Cylinder piston radius $dh = 0.022/2 = 0.0110$

Next the force needed on the wheel to brake the car is calculated. First the gravitational force acting on the car is calculated.

$$Fcar = 4 * mh * g$$

This is then multiplied with the friction constant.

$$FcarS = Fcar * alpha$$

Since the car brakes on all four wheels this is divided by 4 to find the gravitational force acting on each wheel.

$$Fw = FcarS/4$$

The needed force applied to the wheel is calculated by multiplying the gravitanional force acting on the wheel with the radius of the wheel. Since there are two brake pads on each wheel, this is then divided by two.

$$Twh = Fw * (0.54/2)$$

But since the brake pads are not placed directly on the wheels, but on a brake disc with a smaller radius. The above value is then divided by the brake disc's radius.

$$Fbk = Twh/0.08$$

The minimal brake force is then calculated by dividing by the Dynamic friction of the brake.

$$Fbkp = Fbk/mybk$$

Now the force that needs to be applied to the wheels have been calculated and the next step is to find the force that needs to be applied to the pedal, to reach the needed force on the wheels. The ratio between the force on the cylinder piston and the force on the brake pads are, the force on the cylinder piston divided by the surface area of the cylinder piston, is equal to the force applied on the wheel divided by the surface area of the caliber pistons, working on the brake pads.

$$Fmo/Ah = Fbkp/Ak$$

This ratio can be used to find the force needed to move the cylinder piston and thereby the pedal. First the surface area of the cylinder piston and the caliber piston is calculated.

$$Ah = dh^2 * pi$$

$$Ak = Dk^2 * pi$$

The needed pressure in the system to reach the wanted force on the brake pads can then be calculated by dividing the wanted force by the surface area of the caliber piston.

$$Pp = Fbkp/Ak$$

The needed force on the cylinder piston can then be calculated by multiplying the pressure with the surface area of the cylinder piston.

$$Fmo = Pp * Ah$$

It is here important to notice that pressure is force divided by surface area, meaning that the ratio equation above is still been used, the ratio is just substituted with pressure. This is done to illustrate that it is a pressure system that is being worked on, even though all calculations could have been made without ever using the word pressure.
Now that the needed force on the cylinder piston has been calculated, the force needed to be applied on the pedal can be calculated. This can be done by looking at the pedal as a lever.
The idea behind a lever is to increase strength in exchange for moving distance and thereby

time.
Between the cylinder piston and the pedal is a small lever with a 4 to 1 ratio, meaning the force applied the lever gives a 4 times stronger force applied to the piston. The pedal itself is a lever with a 1 to 1 ratio and only functions as an extension of the other lever to make it easier for a driver to press it. To calculate the needed force applied to the pedal the force needed on the cylinder piston is divided by the lever ratio.

$$FmoL = Fmo/4$$

Since there are two cylinders with a piston each the force on the lever is multiplied by 2.

$$FmoLTC = FmoL * 2$$

This gives 422.8 newton force.
This means that the actuator that is to be mounted in the car, has to be strong enough to move at least 422.8 Newton force.

To calculate the distance the actuator need to be able to move, the cylinder piston movement is first calculated. This can be calculated from the ratio between the caliber piston movement and the cylinder piston movement, since there are 4 calibers, the caliber piston movement is multiplied by 4.

$$Ah * Lh = Ak * 4 * Lk$$

Isolating for the cylinder piston movement gives.

$$Lh = Ak * 4 * Lk/Ah$$

Next the lever ratio has to be multiplied to the movement.

$$Gm = Lh * 4$$

After converting from meters to millimetres the result is 50.77.

2.2 Actuator

Based on the calculations above a actuator was chosen. The chosen actuator is a Linak LA30. This actuator has a force of 1800 Newton, a stroke length of 100 millimetres and a maximum speed of 52 mm/s. The actuator is much stronger than necessary and have a stroke length longer than needed, but it was the fastest that was available. The speed is very important since the actuator has to move a reasonable distance before slowing down the car. The 52 mm/s is without any load, but since the maximum load the brake will put on the actuator is nowhere near its maximum output, it will still move at a reasonable speed.
The actuator is a 12V, 20A actuator meaning that it uses 12V and at max load will pull a 20A current. The battery in the car is a 12V battery and should be able to provide the necessary current, since it will never need more than half the power to brake the car. The actuator works like a DC motor and moves forward when getting a positive voltage and backward when getting a negative voltage. Thanks to a screw mechanism, the actuator rod stay at the position it is in when the voltage is removed. This means that when controlling the position it does not need to oscillate around the target position. The voltage is just removed when at the target position.

3 Simulink model

To get an idea on how the brake control system will work, a simulink model of the brake system has been made. To control the braking of the car, a pressure controller is designed. This simulink model is meant to provide a starting point for the gains in the pressure controller. To do this a model of the brake system is made. It takes in a target pressure and returns the actual pressure in the system. As seen on figure 3.0.1 the model is made up of a controller and a plant. The controller is a simple PID controller. The plant is a representation of the actuator

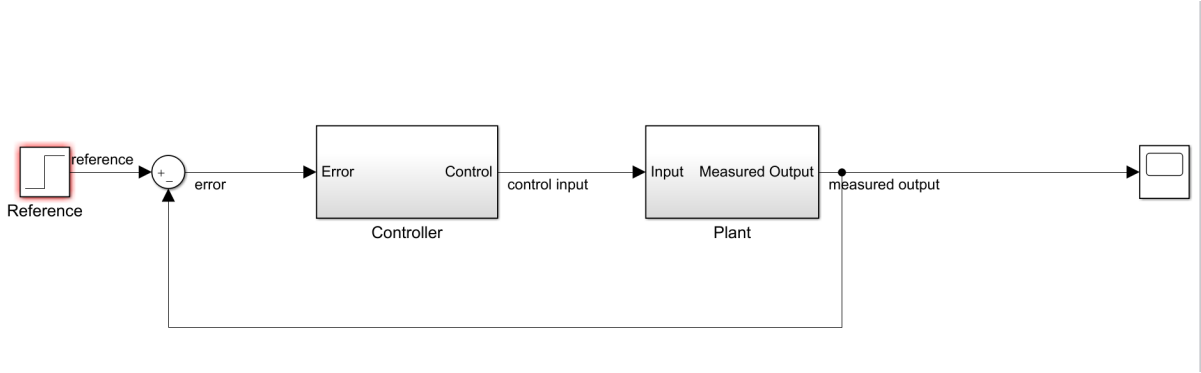

Figure 3.0.1: Model of the brake control system in simulink

and pressure sensor.

The PID controller takes in the error between the target and the actual pressure in the system and returns a voltage that represent the position the actuator moves to. On figure 3.0.2 the plant can be seen. As seen, the representation is very simple. It only consists of a look up table. This table is made from experiments, by slowly setting a position to the actuator and then measuring the pressure. The look up table then makes a translation from position to pressure. This is of cause a very limited representation of the system. To make a more telling representation of the system a full model of the actuator had to be made. This was not done, since to do that a lot of physical parameters needed to be found and since there were not enough time to make all the physical experiments before the brake had to work, a fast solution had to be found.
The plant takes in the position value from the controller and returns the pressure value in the system.

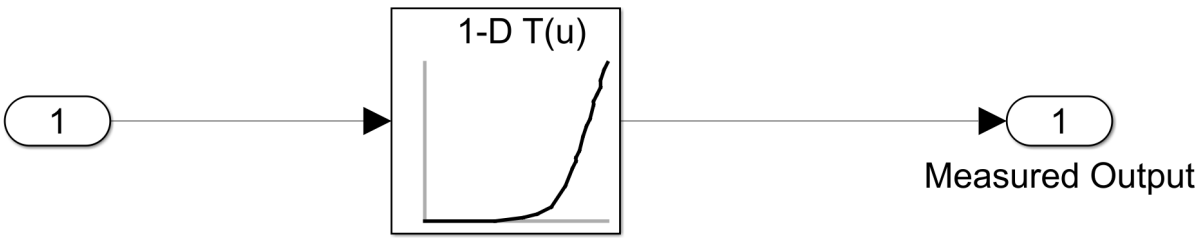

Figure 3.0.2: Model of the PID controller in simulink

4 Code

To control the brake a ROS node has been made. This node reads the pressure the navigation note tells it to set and calculates a position to sent to the actuator. A motor driver with build-in position controller was sent with the actuator. The motor driver is needed since the actuator use 12V and the computer controlling the system does not provide that much voltage. The motor driver receives a control voltage from the computer and uses the power supply battery, to provide the necessary voltage to the actuator based on the control voltage. The motor driver also need to know at what position the actuator is at and to know that a potentiometer is installed in the actuator. The potentiometer gives an value between 0 and 5.5 in the form of a voltage. To sent the control voltage from the computer to the motor driver, a PhidgetAnalog 4-Output converter is used. It reads voltage on the usb port and converts it to a voltage between -10V and 10V.

4.1 Brake node

The brake node is divided into three files. A .h file for initialisation, a .cpp file for functions, both called BrakeSystem and a .cpp file for the main script, called brake_node.

4.1.1 BrakeSystem.h file

The BrakeSystem.h file initialise and call necessary variables and functions. To begin with useful libraries are called.

ros/ros.h
The ros/ros.h is the main ROS library for all the ROS functions that is used for node building and communication.

std_srvs/Empty.h
The std_srvs/Empty.h is used for ROS service functions. These are functions that can be called from another script. The empty part is the choice of service. Empty means that the service is called without any data exchanged between the caller and the service function. If it said Trigger it would have been possible to see if the service was triggered or not.

std_srvs/SetBool.h
The std_srvs/SetBool.h is like std_srvs/Empty.h, but allows the service to tell if a bool was set.

dynamo_msgs/TeensyWrite.h, dynamo_msgs/TeensyRead.h and dynamo_msgs/BrakeStepper.h
The dynamo_msgs/TeensyWrite.h, dynamo_msgs/TeensyRead.h and dynamo_msgs/BrakeStepper.h allow for communication with messages from the stated nodes. These are used when calling ROS functions for subscribing and publicising.

fstream
The fstream library is not used, but is for file communication.

phidget21.h
The phidget21.h library is for communication with the PhidgetAnalog 4-Output converter. This library contains several functions needed for the control of any phidget device.

ros/console.h
The ros/console.h library provides functions for writing to the console.

boost
The three boost libraries are unused in this version of the code. The point of the libraries is to make a initialisation file, which the main script can read from to set values for the variables that may change due to physical changes and the calibrate service function can write to, making it possible to save values for the variables found during the calibration. This would mean that the user didn't have to calibrate every time the code is run to get the newest values for the node.

Next the constants the program uses are defined. These constants are values that must never change in the program.

Min_Pressure and Max_Pressure
The Min_Pressure and Max_Pressure constants are boundaries the code must work under and are the minimum and maximum pressure that the program can move the actuator to. The Min_Pressure constant is not used in the program, since it was found that it is easier to work with minimum position. The constant has been kept for possible future use. The Max_Pressure constant is used for finding the maximum allowed position during calibration.

KC, IC and DC
The KC, IC and DC are constants used for the PID controller function. These are the gains used to calculate the position from the pressure difference error in the PID controller function.

ANALOG_PHIDGET_SERIAL
The ANALOG_PHIDGET_SERIAL constant is the serial number for the PhidgetAnalog 4-Output converter, that the phidget functions need, to know where to sent their information.

WAIT_FOR_ATTACH_TIMEOUT
The WAIT_FOR_ATTACH_TIMEOUT constant is used by a phidget function, which check for connection with the PhidgetAnalog 4-Output converter, if the time it takes to connect in milliseconds is greater than that constant it will create an error message to the user.

Lastly a class is generated. This class contains some private variables only used in the class functions and some public variables and functions used outside the class. The CPhidgetAnalogHandle is a handle used by the phidget functions for communication with the PhidgetAnalog 4-Output converter.

4.1.2 BrakeSystem.cpp file

The BrakeSystem.cpp file contains all the function initialised in the .h file.

BrakePID(float target)
The first function is the BrakePID(float target). When called it takes in a target pressure and calculates a position, which is to be sent to the actuator to move to. This is done by first calculating the difference error between the target pressure and the actual measured pressure in the system. There are two actual pressure measurements, one for the front wheels and one for the back wheels. The BrakePID function only controls for the back pressure. It then generates an output gain from the calculated error and the PID gain constants. D gain in the PID has been set to 0, since the D gain in a PID often result in instability in physical systems with a big delay in time, like this brake system. The I gain is calculated from the calculated error, the frequency at which the code run and the I gain constant. To make sure that the position is not set to a value above or below what is physically possible for the system, the function output is checked against a preset maximum and minimum position value. If the output value is bigger than the maximum position it is set to the maximum position. The same is true for the minimum position check. The function then set a position between maximum and minimum.

SentToCon(double Vol)
The second function is SentToCon(double Vol). When called the function takes the given position in voltage and sent it to the position controller via the PhidgetAnalog 4-Output converter. It does this by using the phidget functions CPhidgetAnalog_setVoltage (phid, 0, Vol) and CPhidgetAnalog_setEnabled (phid, 0,1). The CPhidgetAnalog_setVoltage (phid, 0, Vol) function sents the voltage Vol to the output number 0 via the handle phid. The CPhidgetAnalog_setEnabled (phid, 0,1) function enables the output 0 via the handle phid, which allow the PhidgetAnalog 4-Output converter to sent the voltage, set on the output.

UpdateRef(float brakePot, float brake_press_1, float brake_press_2)
The third function is UpdateRef(float brakePot, float brake_press_1, float brake_press_2). When called this function takes in the measured values from the sensors interesting for the brake node and return them in a unit form, that is easier to work with. For the actuator position the measured value is multiplied with the maximum voltage divided by the maximum bit number. This gives the position in voltage instead of a bit value.

$$pot = (5.5/1024) * brakePot$$

For the pressure measurements the measured pressure is multiplied by a gain calculated from the maximum voltage given by the sensor and the voltage divider in the teensy board that

receives the measurement.

$$press1 = 0.001732418442 * brake\_press\_1 - 11$$

A original version of the pressure measurement of the back pressure is also saved to a new variable. The function set the calculated measurements to variables that are used elsewhere in the code.

CalibrateBrake()
The fourth function is CalibrateBrake(). When this function is called it sets the position of the actuator to 0 and then check if there is any difference in the actuator position measurement for a set time. If there are no difference the posMin variable used by the BrakePID function, is set to the measured position. This way of finding the minimum position that the actuator is allowed to move to, works because there is a parameter in the position controller that can be set to force a minimum movement point. The minimum position the PID is allowed to set is then set to the same as the position controller. Next the function find a value for the posMax variable used by the BrakePID function, during the check if the maximum position has been exceeded. It does this by slowly increasing the position sent to the actuator and checking if the maximum pressure has been reached, if so then the variable posMax is set to the measured actuator position. The function return a bool that is true when the calibration is done.

phidget attachment
The fifth, sixth and seventh functions are used by the phidget library to tell the user if the PhidgetAnalog 4-Output converter has been attached, detached or run into an error. These functions allow to choose what happens in the code if one of these happens. As of the version of the code that is described in this rapport, these functions does nothing more than inform the user.

BrakeSystem(float time)
The eighth function is BrakeSystem(float time). This function is called once every time the main script is run. This function set variables to a start value for start up and run functions that only need to be run once. These functions include some phidget functions for creating the phidget handle and communication with it. Then a function for establishing communication with the PhidgetAnalog 4-Output converter is run and a check if this was done correctly and if not a error message is given to the user. Lastly the the actuator position is set to 0.

close()
The last function is close(). This function is only called once, when the main script is closed by the user. It disables connection to the actuator, thereby setting it to position 0, close all communication with the PhidgetAnalog 4-Output converter and delete the phidget handle.

4.1.3 brake_node.cpp file

The brake_node.cpp script is the main script, that is run when the user wish to control the brake autonomously. It begins by defining two constants, these are defined here, because they are standard to the ROS node. These are DEBUG_LOGGER and CONTROL_LOOP_RATE. Next the BrakeSystem class is called. After that a enumeration is made containing different states the node can be in.
Next is a ROS service function called StartCalibration(). This function sets up a ROS service, which means that this function will run, whenever the function is called from another console. The function set the node state to the calibrate state. This is used in the main function to decide what to do. This calibration service function is not finished and is untested.
The next two functions are subscription callbacks for the ROS communication. These functions are called whenever there is an update on one of the subscribed topics. The first subscription function is the brakePowerCallback. When this function is called it sets the target pressure for

the PID, to the pressure value given by the publicising node. It gets this value from the brakePower topic in the systems node network's list of topics. The second subscription function is teensyCallback. When this function is called it call the UpdateRef function from the BrakeSystem.cpp file. After that another service function is created, which is the aliveCallback. When this function is called the node is forcibly closed, due to an error in the ROS communication. The function sets the node to fatal state. This function is a standard function for all nodes in the autonomous car system.
After the subscription functions and the aliveCallback service function, the main function is called. When the main function is called, the function first do some ROS initialisations. First the node state is set to NODE_STATE_WAIT, to tell that the node is not ready for use yet. Next the function ros::init is called. This function creates the node in the ROS node network. After that the function ros::NodeHandle is called. This function creates a handle for the node, which is used doing communication. These two functions are standard for ROS nodes.
After the ROS node is set up in the network of nodes, the function ros::Subscriber is called for each subscription function. This function subscribe the node to the stated topic in the systems node network and links it to the subscription function. Next the function ros::ServiceServer is called for each service function. This function creates service servers on the systems node network and links them to the service functions stated. All of these functions, that work by checking if there is an update on a value in the systems node network, is run by the function ros::spinOnce. When this function is called a check for updates on the subscriptions and services are done. This means that even if the subscription functions and service functions are never called directly in the code, they get called by ros::spinOnce, if there is an update on the subscribed topic. After this, the function ros::Rate rate(CONTROL_LOOP_RATE) is called. When this function is called the rate class is created. The class is a ROS class that help ruining the code at a given frequency. It is given the frequency given by the constant CONTROL_LOOP_RATE.
Everything written above happens once, when the script is run. For the controlling of the brake to run continuously a while loop is used. This loop run as long as ros::ok() returns true. ros::ok() is a ROS function that always return true, as long as the node has not been shutdown externally or Ctrl+C has not been pressed in the console. In the while loop the first thing that happens is that a ros::spinOnce() is run. After that, three if statements check if the sensor readings are within expected limits. If a sensor reading is outside the expected limits, the state of the node is set to NODE_STATE_WAIT, the position of the actuator is set to the latest calculated position and a warning is written to the user. If the sensor readings are not outside the expected limits the node state is set to NODE_STATE_OK.
After this three if statements are checked. Other if statements check what state the node is in. Dependent on what state the node is in, different codes are run. There are four different states that is checked for, NODE_STATE_OK, NODE_STATE_FATAL_ERROR, NODE_STATE_TEST and NODE_STATE_CALIBRATE.

NODE_STATE_FATAL_ERROR
The first if statement check if the node is in NODE_STATE_FATAL_ERROR. If this is true then an error message is sent to the user and the while loop is broken.

NODE_STATE_TEST
The second if statement check if the node is in NODE_STATE_TEST. If this is true then a small test code is run. This part of the code is meant for small tests on the movement of the actuator. A few variables that is needed for the test is initialised above the if statement. First the test code prints some useful measurements to the console for the user to read. Next it check if it has been under 10 seconds since the code started to run. If this is true then the function brakeSystem.BrakePID(test) is called and right after that the function brakeSystem.SentToCon(brakeSystem.position) is called. This first calculates a position from a target pressure called test. After that the test position is sent to the position control and thereby the actuator. The test pressure can be chosen by the user in the code as one of the variables initialised above the if statement. If it has been more than 10 seconds since the code started to run, the function brakeSystem.SentToCon(brakeSystem.posMin) is called. This sets the actua-

tor to its start position.

NODE_STATE_CALIBRATE
The third if statement check if the node is in NODE_STATE_CALIBRATE. If this is true the calibration code is run. This code first print relevant measurements and variable values to the console for the user. It then call the funtion CalOK = brakeSystem.CalibrateBrake() and then brakeSystem.SentToCon(brakeSystem.position). This will first run the calibration function, which will find a position that is then sent to the position contol and thereby the actuator. After that a check is made for the return value of the calibrate function. If this is true then the node state is set to NODE_STATE_OK otherwise the state remains NODE_STATE_CALIBRATE and the calibration continue to run.

NODE_STATE_OK
The fourth if statement check if the node is in NODE_STATE_OK. if this is true then the main brake control code is run. This code first print to the console, that the brake node is OK and ready to use. After that a check is made to see if there has been a change in the requested power that the brake needs to move to. If this is true the integral part of the PID is reset to 0. This is done to avoid an accumulation of the integral part of the PID, which would otherwise tell the actuator to move very fast to a very far point. Next there is a check if a engage command has been sent to brake node. If this is true then some relevant measurements are sent to the console for the user. After that a check is made to see if the requested power is 0. If this is true, then the function brakeSystem.SentToCon(brakeSystem.posMin) is called, which moves the actuator to the start position. This is done because it is faster to tell the actuator to move to a specific position, then to make it use the PID and since the start position is constant there is no reason to use control to reach it. If the requested power is not 0, then the function brakeSystem.BrakePID(brakeSystem.power) is called followed by the function brakeSystem.SentToCon(brakeSystem.position). This first calculates a position from the requested pressure and then sent the position to the position controller and thereby the actuator. If the check for the engage command is not true, the function brakeSystem.SentToCon(brakeSystem.posMin) is called. It sets the position of the actuator to its start position.

Lastly in the while loop the function rate.sleep() is called. This is a ROS function that pause the program long enough for it to run at the frequency, set with the ros::Rate function.
When the while loop is broken and just before the node shutdown the function brakeSystem.close() is called, to close down all phidget communication.

4.2 Position controller

The driver with a build-in position controller is a TR-EM-288-SAF. To program this some parameters can be change via the attachable screen TR-EM-236. There are 25 parameters that can be set to a desired value. Below are the relevant ones described. For more information on the rest, see reference [4].

Speed, Backward
This parameter sets the speed of the actuator in its backward direction. The speed is stated in procent %, 0 to 100. For this project it has been set to 90%. This is because if it is set to more than 90% the position controller overshoots more than wanted.

Speed, Forward
This parameter sets the speed of the actuator in its forward direction. The speed is stated in procent %, 0 to 100. For this project it has been set to 70%. This is because if it is set to more than 70% the position controller overshoots more than wanted.

Current limit, OUT
This parameter sets the maximum allowed current consumption in the forwards direction. The

current is stated in ampere, 0.1 to 20. For this project it has been set to 100 equivalent to 10 ampere. This is for security reasons, since it is better if the position controller shuts down than if the actuator push too hard and break the brake. The 10 ampere is approximately equivalent to 25 of bar pressure on the brakes.

Current limit, IN
This parameter sets the maximum allowed current consumption in the backwards direction. The current is stated in ampere, 0.1 to 20. For this project it has been set to 100 equivalent to 10 ampere. Since the actuator only pushes and lets go of the brake in the system, this parameter is not very important, but has been changed to fit the current limit in the opposite direction.

Read fault values
This parameter decides what kind of output port 13 has. It can be set to 1, 2, 3 or 4. If it is set to 1 the port will go high in case of a fault in the driver. If it is set to 2 the port will go high when the targeted position has been reached. If it is set to 3 the port will output the current position of the actuator 0 to 5 volt. If it is set to 4 the port will output the current position of the actuator 0.5 to 4.5 volt and if a fault happens in the driver it will output 0 volt. For this project is has been set to 3, because it was the plan to sent this output to the teensy to give the position value to the computer. It turned out that it was much easier to sent the position directly to the teensy from the potentiometer. For future work it would be good to set this parameter to 2 and connect it to the teensy. In combination with the reset port 11, it would then be possible to make some code, that reset the driver if an error occurred.

Max. stroke length OUT
This parameter sets the maximum allowed position in the forward direction. The position is stated in procent %, 0 to 500, where 500 is equivalent to 50%. For this project this parameter has been set to 90. This allow for a small overshoot when the the actuator moves to its maximum allowed position, while keeping the actuator from going further than needed.

Max. stroke length IN
This parameter sets the maximum allowed position in the backwards direction. The position is stated in procent %, 0 to 500, where 500 is equivalent to 50%. For this project this parameter has been set to 250. This moves the actuators starting position to a point right before it touches the brake and saves some time when moving the actuator.

5 Test

To test if the brake work as intended the test node state described in section 4 was used. The first thing that was tested was the PID gain constants. These were found by testing the brake and changing the gain slowly. To find a place to start the simulink model was used. The gain values that looked good in the model was KD = 0.01 and ID = 0.017. Testing this on the actual system gave a much slower result. This was to be expected, since the model does not have any delay and operates at the computers speed. The gains for the real system ended up been KD = 0.02 and ID = 0.017. The output of the actual system for different target pressures can be seen on figure 5.0.1, 5.0.2, 5.0.3 and 5.0.4. The test first set the target pressure to one of four values and after 10 seconds it sets the target pressure to 0.

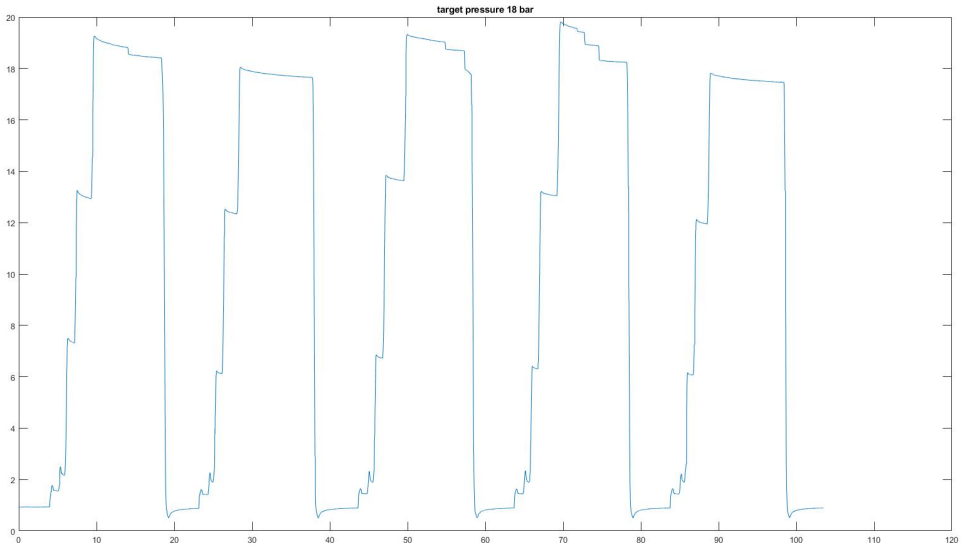

Figure 5.0.1: Pressure measurements for target 18 bar test.

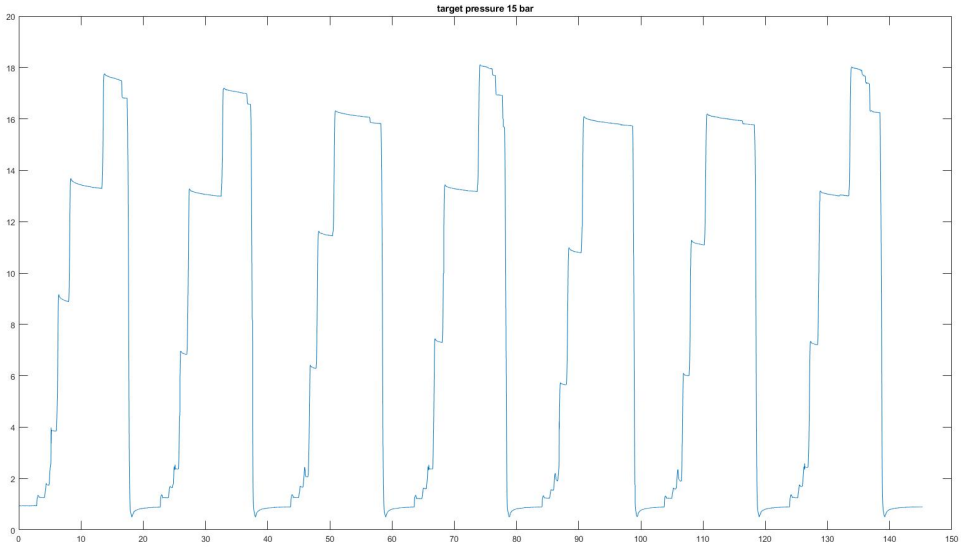

Figure 5.0.2: Pressure measurements for target 15 bar test.

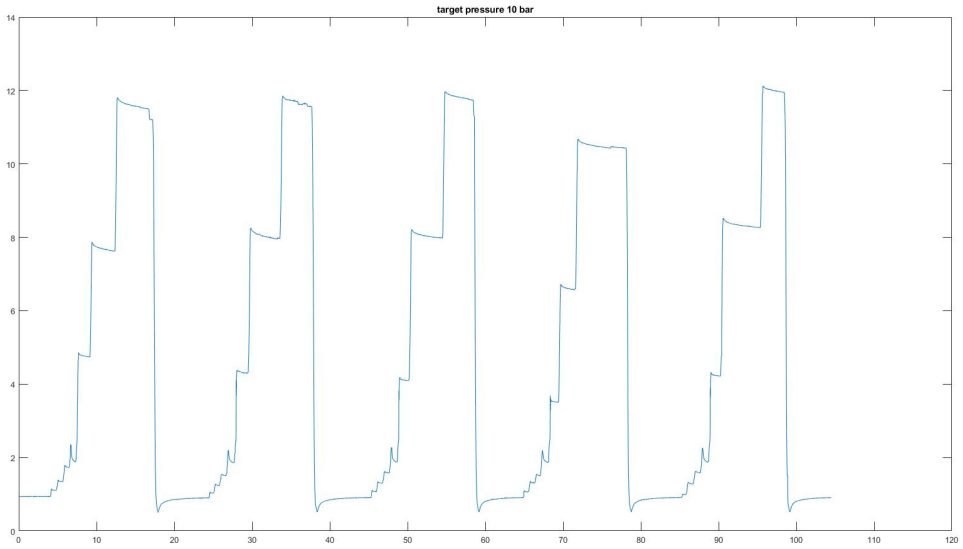

Figure 5.0.3: Pressure measurements for target 10 bar test.

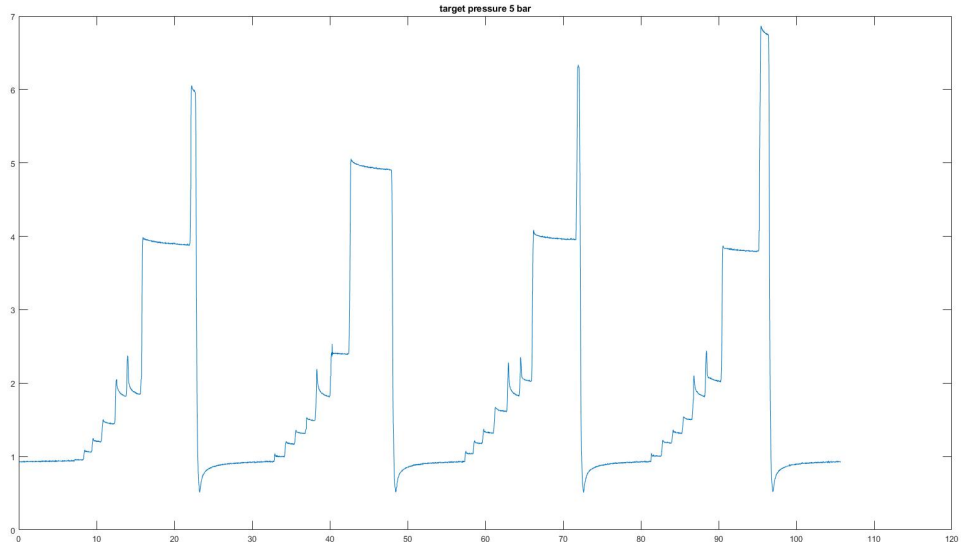

Figure 5.0.4: Pressure measurements for target 5 bar test.

The test is run 4 to 7 times. As seen on the figures it takes about 5 seconds for the actuator to reach the target pressure. This is of cause not desired. The reason for this is that as seen on the figures, the actuator takes breaks at different positions as it moves. Why it does this has not been discovered. As seen on the figures the problem does not occur when moving to 0, where it does not use the PID and just sets a position. Due to this, a hypothesis has been made, that states that it might be something with the PID control. For the deadline of this rapport it has not been possible to find a solution to this problem.
Even with the above mentioned problem, the test show that the brake system work and that it is possible to set a target pressure and move the brake to that point.

6 Future improvements

Check for both pressure measurements in PID. As the code is right now the PID controller only calculates the position gain from the pressure measurement from the back brakes. It would be best to find a way where both the front and back measurement was included, when calculating the position. A good way to do this has not been found.

The phidget functions for attachment warnings need to be finished. As the code is right now they only print a warning message to the user and nothing more. If there is a fault in the phidget communication, something needs to be done to avoid sending a bad signal to the position controller. What to do with these functions has not been decided.

The calibration state in the node has not been tested extensively and may have some unforeseen faults. Also the ROS service call to the calibration has not been tested at all. As the code is right now, NODE_STATE_CALIBRATE will most likely be overwritten by one of the other states before reaching the calibration function. This needs to be fixed, before trying to use the calibration function.

For future work it would be good to set parameter 9 on the driver to 2 and connect it to the teensy. In combination with the drivers reset port 11, it would then be possible to make some code, that reset the driver if an error occurred. Although this require that the teensy is extended with two more ports one to output to the drivers port 11 and one to read from the drivers port 13.

7 Conclusion

A working brake system has been designed and implemented for the DTU Roadrunner. The brake system works together with many other systems and both mechanical, electrical and programming has been worked with in this project. Not only was the mechanical part of the system calculated to find the best actuator, but through out the project, mechanical parameters had to be considered, because if the code did not work, something mechanical might break. It was a constant concern, and reprecautions had to be made every time something new was introduced to the system.
To make the actuator move and know where it was, a sensor and a driver had to be installed. This led to a lot of wiring and voltage and current design decisions. Since it had to work in cooperation with a already existing system it had to be design in a way that allowed for this.
A ROS node was programmed to control the rest based on information gathered from the other ROS nodes and sensors.
To make the entire autonomous car work, everybody working on it had to work together and make sure that their different parts could work together as well.
There are still some improvements to be made, but a brake system has been implemented and a lot has been learned from work on a big project with many different professions involved.

References

[1] Platform Integration for Autonomous Self-Driving Vehicles, Master's thesis, Henning Si Høj, DTU, (© Henning Si Høj, June 2017)

[2] Udvikling af hovedcylindder til bremsesystem, Nikolai Nilsson, DTU, (© Nikolai Nilsson, June 2015)

[3] LA30 - datablad, Linak, https://cdn.linak.com/-/media/files/data-sheet-source/en/linear-actuator-la30-data-sheet-eng.ashx

[4] TR-EM-288-SAF - User guide, Linak, http://www.linakthirdparty.com/products/drivers.aspx?product=TR-EM-288-SAF+-+Positioning+driver

List of Figures

Figur 1.1.1	Illustration of a hydraulic brake	4
Figur 1.1.2	Picture of the mounted brake actuator	5
Figur 3.0.1	Model of the brake control system in simulink	8
Figur 3.0.2	Model of the PID controller in simulink	8
Figur 5.0.1	Pressure measurements for target 18 bar test.	15
Figur 5.0.2	Pressure measurements for target 15 bar test.	15
Figur 5.0.3	Pressure measurements for target 10 bar test.	16
Figur 5.0.4	Pressure measurements for target 5 bar test.	16