Case Study – Sensory Aided Waypoint Navigation

P. Aguilar - 11120782
A. Bodendorfer - 11071015
C. Hoeffer - 11060177

Automation & IT

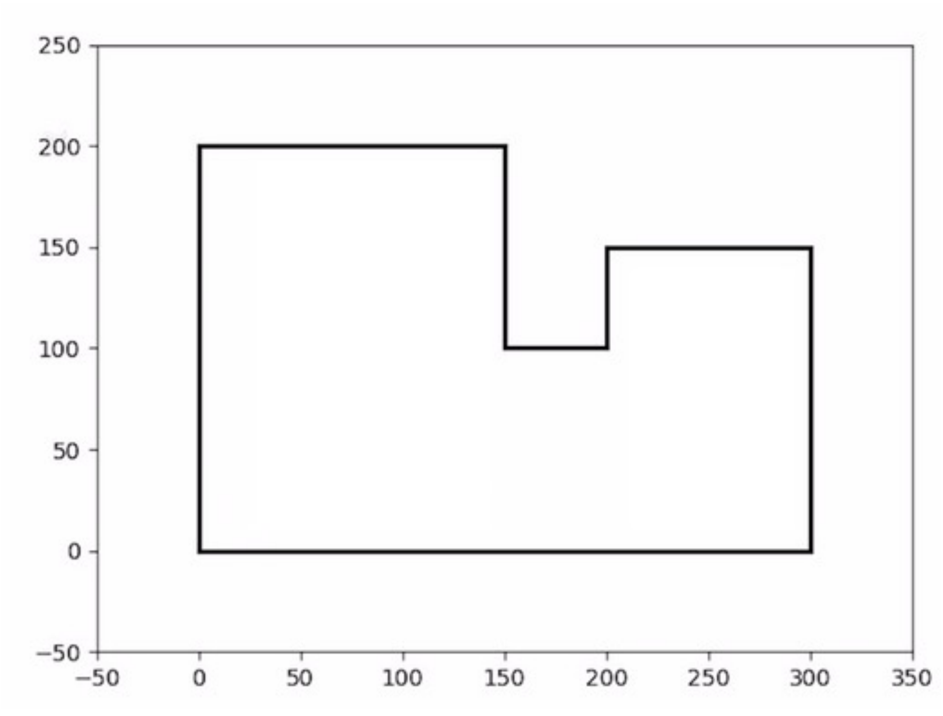
Technology Arts Sciences
TH Köln

[Type here]

Table of Contents

Introduction

The goal of this case study is the implementation of an autonomous robot able to navigate through given global coordinates in a walled environment. The exterior shape of a maze is given and has 13 points which the robot should visit at least once in a determined sequence. The robot should be able to recognize its starting position with the help of ultrasound sensors.



Outline of the given maze

The basic hardware for this robot a modified Activity Bot 360 robot kit with an additional ultrasonic sensor and a Gyroscope. At the end, a map is produced marking the commanded locations as well as the real (measured) locations. All software is written in Python 3.

The modules of the implementation are using the pigpio module to control the GPIOs of the Raspberry Pi.

The movement of the robot is done by a combination of the following functions:

• Turning on the spot

• Moving on a straight line - forward and backwards

• Scanning the surrounding with an ultrasonic sensor mounted on a servo

The turning and straight movements are controlled by four digital PID controllers. Each wheel is controlled by a cascade control, using a cascade of two PID controllers. The outer loops control the position while the inner loops control the speed of each wheel.

The modules developed for this provide simple APIs for turning and straight movements and also for scanning the surrounding or steering a servo.

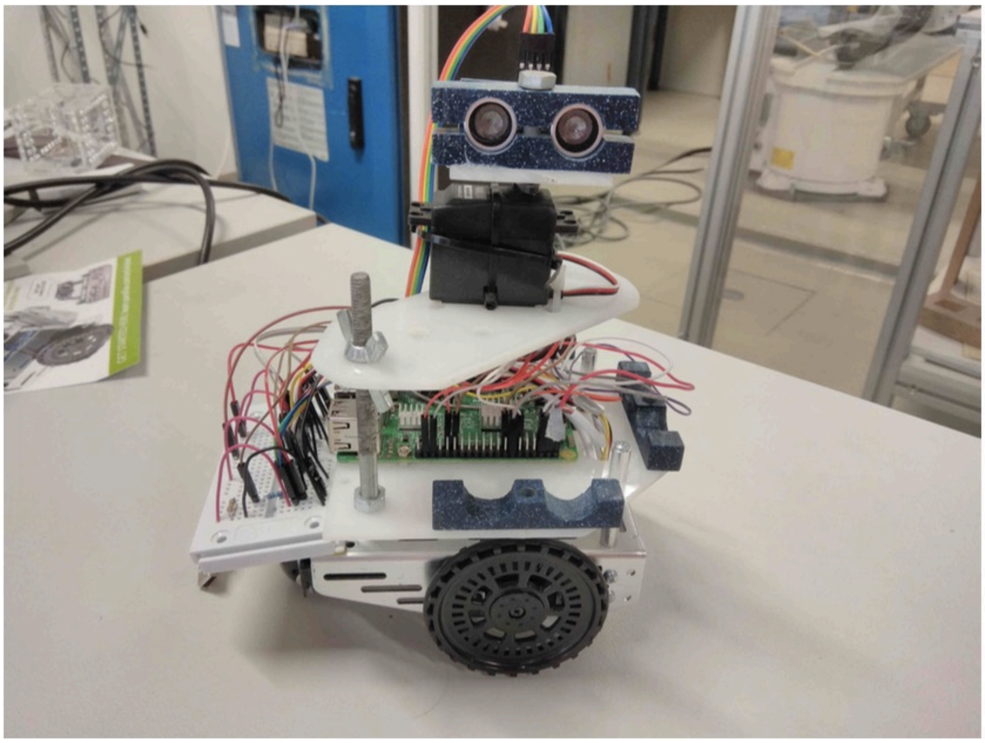
[Type here]

Most of the default values in the modules are those which are used while experimenting/developing with the demo implementation. They provide a good starting point for the range of the values. Pictures of the demo implementation can be found further down in this section.

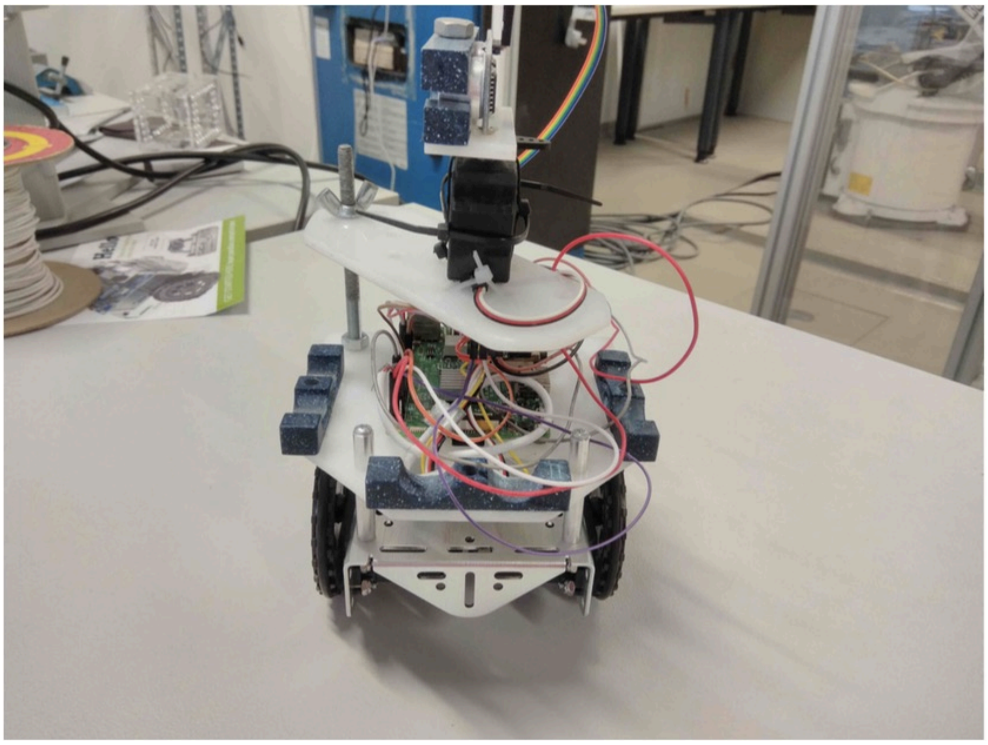
The programed modules also enable remote controlling the Raspberry Pis GPIOs. This enables use of the modules on a lap- top/computer and over e.g. WLAN remote controlling the Raspberry Pi which provides a WLAN hotspot, see re- mote_pin and pi_hotspot. This way, the robot can freely move with a power bank attached without any peripheral devices while programming/controlling it. The possibility of remote controlling the Raspberry Pi's GPIOs is a big advantage of the used pigpio module. This drastically improves the use of the modules, because then all programming/controlling can be done on a laptop/computer including using an IDE, having much more system resources and so on. It is also possible to use the modules on the Raspberry Pi itself and connect to it over VNC, see VNC. For both ways, using the modules on the Raspberry Pi itself or remote on a laptop/computer to control the Raspberry Pis GPIOs, no modifications have to be done in the source code of the modules.

Pictures of the demo implementation

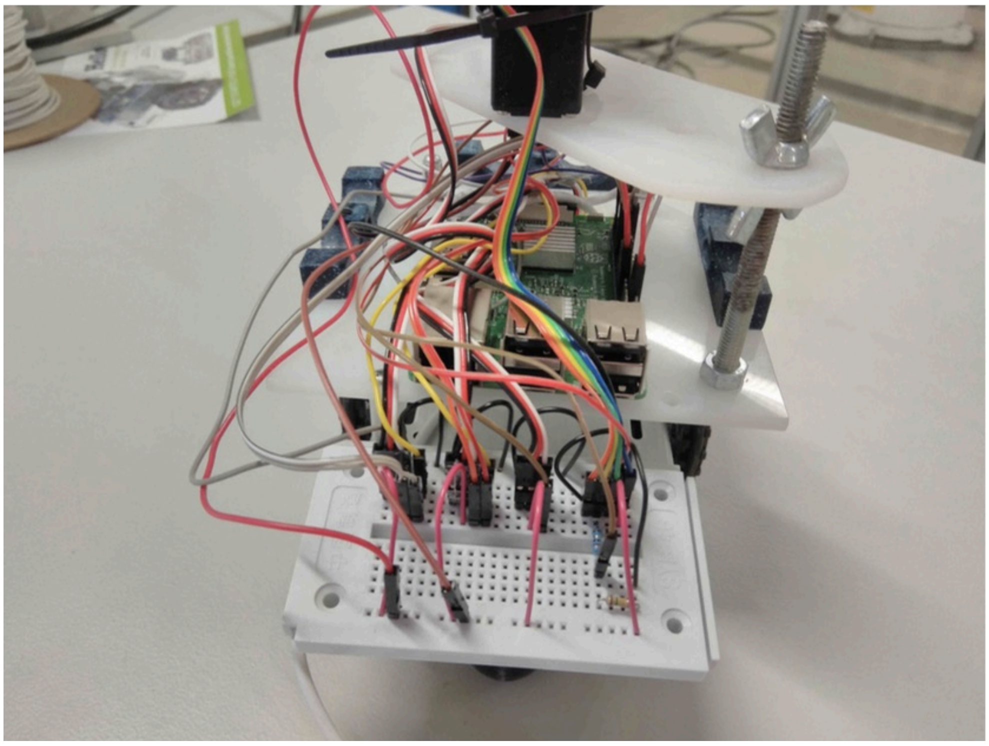
View from the right side.



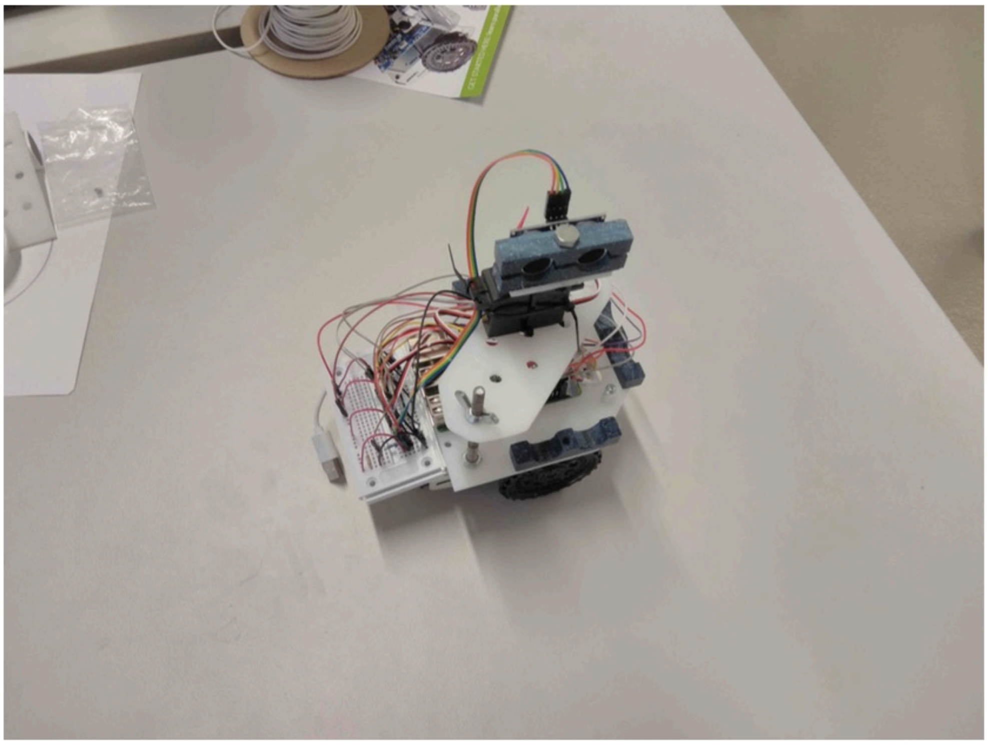
View from the front side.



View from the back side.

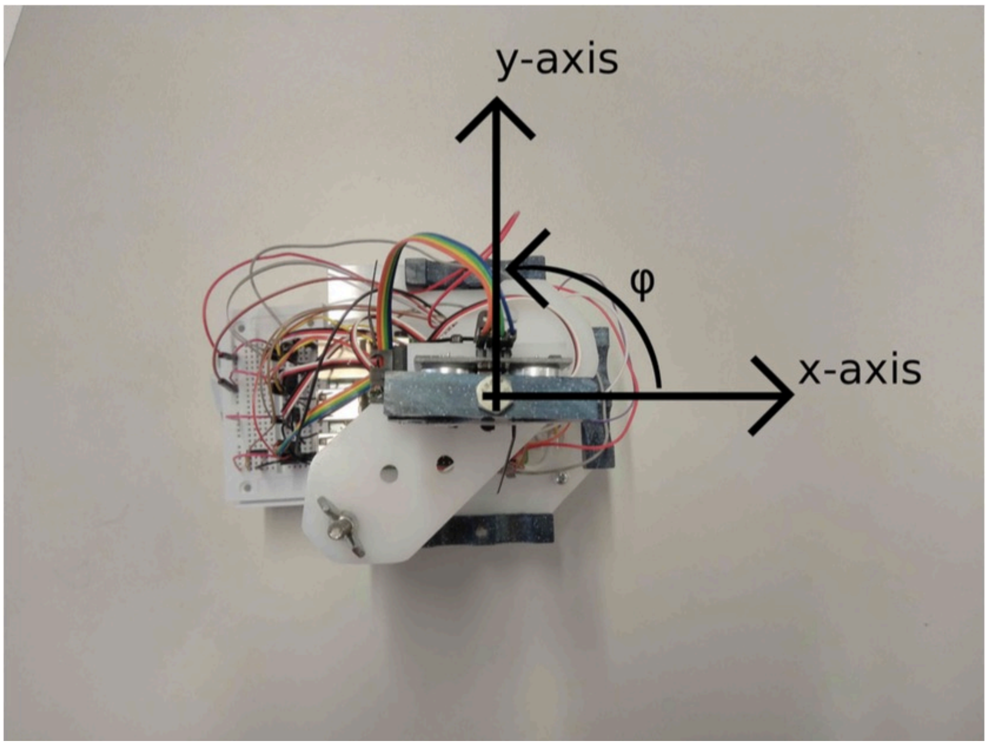


View from the top



Used local coordinate system

The following picture shows the local coordinate system which is used in lib_motion.

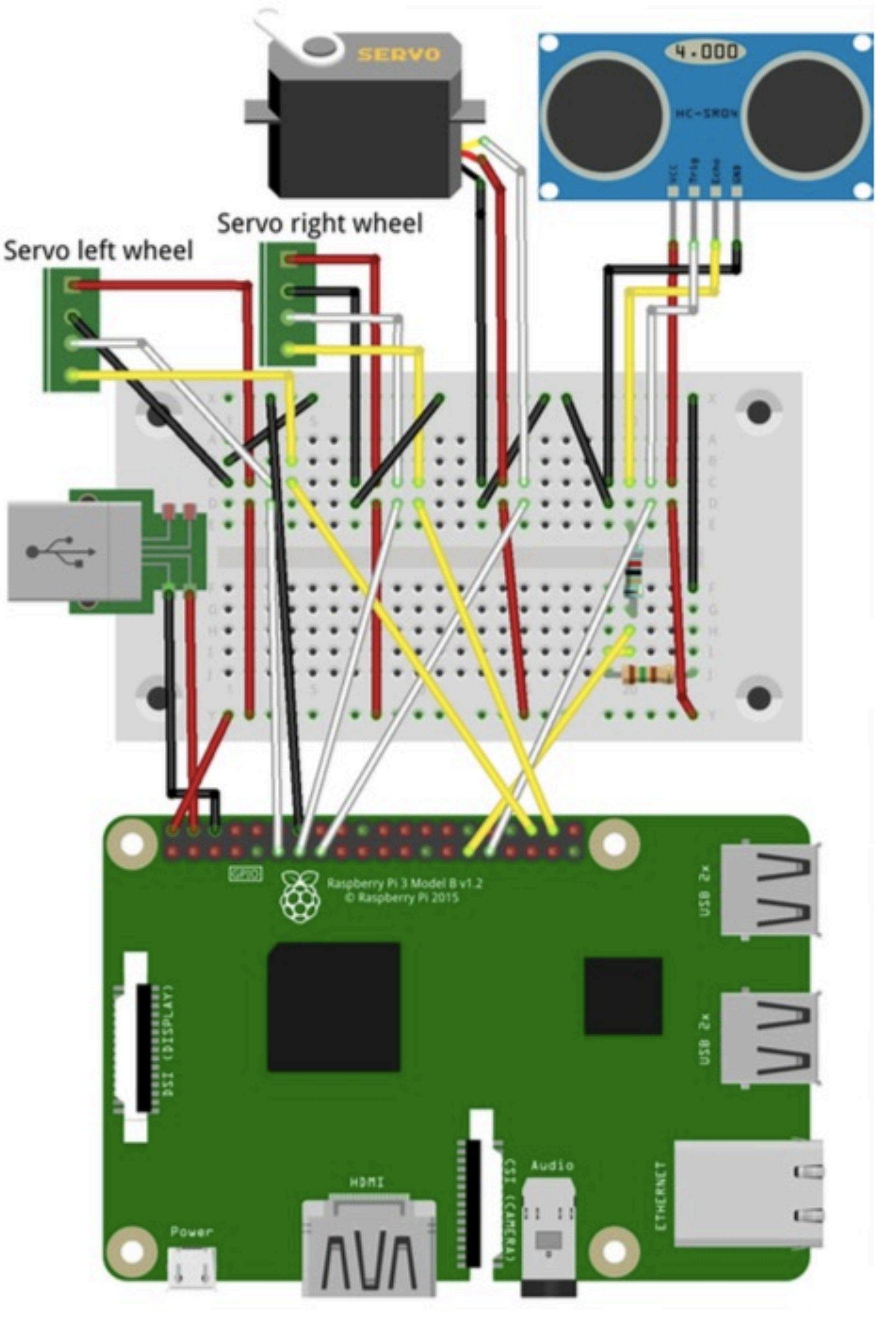


Cabling of the demo implementation

Below a Fritzing sheet which illustrates the cabling of the demo implementation. The voltage divider is built out of two resistors, blue one with 82 Ohm and gold one with 150 Ohm.

Warning: The voltage divider is very important. The Echo Pin of the HC-SR04 outputs a PWM with the same voltage as VCC Pin (5 V in this case) which needs to be converted to 3.3 V. 3.3 V is the max voltage the Raspberry Pi can handle on a GPIO, otherwise it might get damaged! The chosen resistors for the voltage divider convert 5 V to 3.23 V. Setup: The output of the Echo Pin is connected to the blue 82 Ohm resistor. At its end, the GPIO is connected and then the golden 150 Ohm resistor at whose end ground is connected.

Warning: The 5 V of the USB power supply should be connected to the 5 V Pin of the Raspberry Pi directly, as shown in the Fritzing sheet. The USB power supply does not only power the Raspberry Pi itself, also the three servos and maybe more devices, which get added in the future, are powered over it. Therefore, powering all devices over the Raspberry Pi's micro-USB port should be avoided, because otherwise all needed current of all devices would be conducted through the Raspberry Pi.



General Software

As mentioned, all libraries and programs are written for Python 3. The following libraries were used in the development of this case study.

General Libraries

- time: Used for specific delays, such as pauses between intgration samples and calibration
- math: Baic mathematical functions
- json: Communication between threads is done using json encapsulement
- pigpio: Control of the General Purpose Input Outputs (sensor communication and actuator commanding)
- numpy: Trigonometric functions

Threading Library

- threading

Gyroscope Libraries

- mpu6050: Controlling library for the MPU6050 Accelerometer and Gyroscope
- python3-smbus (dependency): Uses the inherent i2c structs and unions

Additional Library

- matplotlib: Graphical library for map creation.

Additionally, two libraries were specifically developed for this robot. One lib motion which controls the movement of the robot and lib scanner in charge of the rotation of the ultrasound sensor located on top of the robot.

Sensor Modelling

Two sensors are used to better determine the position of the robot as a way to complement the information from the encoders. An ultrasound sensor is used to measure distances to the walls of the maze and a gyroscope to measure the spin of the robot during motion.

Ultrasound

A special library was programmed to interreact with the sensor and the motor. This is further explained in the lib_motion chapter.

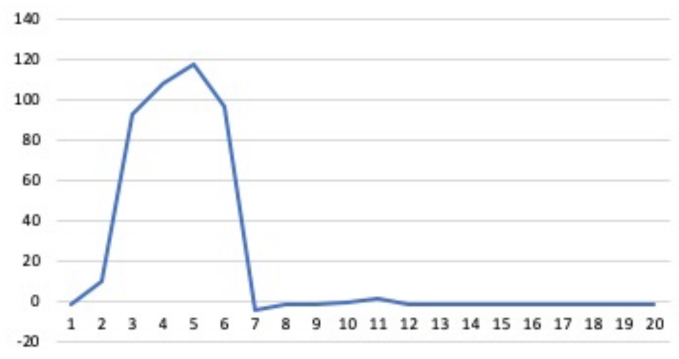
To model the ultrasound sensor, a simple covariance matrix was built by measuring the distance from a fixed origin position to a wall located at fixed distances. The mean of the error was calculated to be of 2.3 cm.

Gyroscope

The MPU 6050 gyroscope was used in this experiment. Once the mpu6050 and python3-smbus dependencies have been installed, the process to obtain readings from the sensor is as follows:

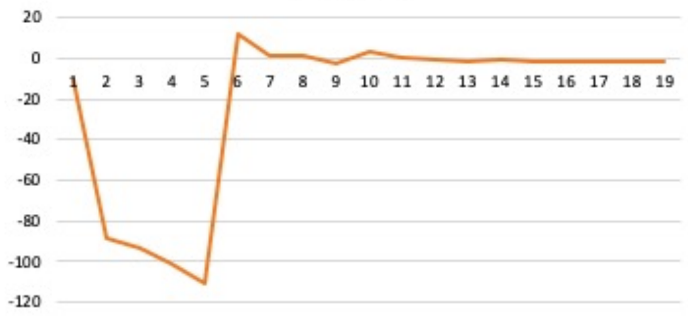
```
#Required libraries
from mpu6050 import mpu6050
from time import sleep

#Instance of the sensor
sensor = mpu6050(0x68)

#Pause to calibrate the sensor
print("waiting for sensor to calibrate")
sleep(2)

while True:
	#Read the data from the sensor in an endless loop
    accel_data = sensor.get_accel_data()
    gyro_data = sensor.get_gyro_data()
    temp = sensor.get_temp()
    #print data in the Z axis
    print(" z: ", str(gyro_data['z']))
    sleep(.5)
```

The following graph portrays the measurement of a 45º spin commanded to the robot



The measurement from the gyroscope in integrated and gives a value of 43.1º.

The following graph portrays the measurement of a -45ºdegree spin commanded to the robot:



The measurement is of 44.3º.

A covariance of the error of this sensor was built by repeating the same experiment 20 times and set to 2.7º.

Waypoint Navigation

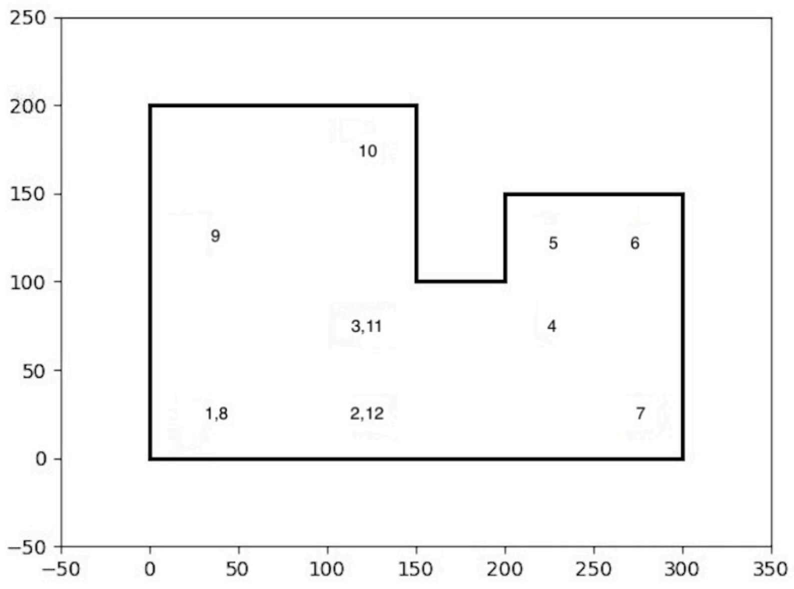
For this goal, 13 global coordinates were given within a previously constructed maze with defined obstacles.

The coordinates for said points are stored in a double precision two-dimensional array that correspond to the middle point of the cells in an imaginary grid in the maze as follows:

pointL = [[25.0, 25.0],

[Type here]

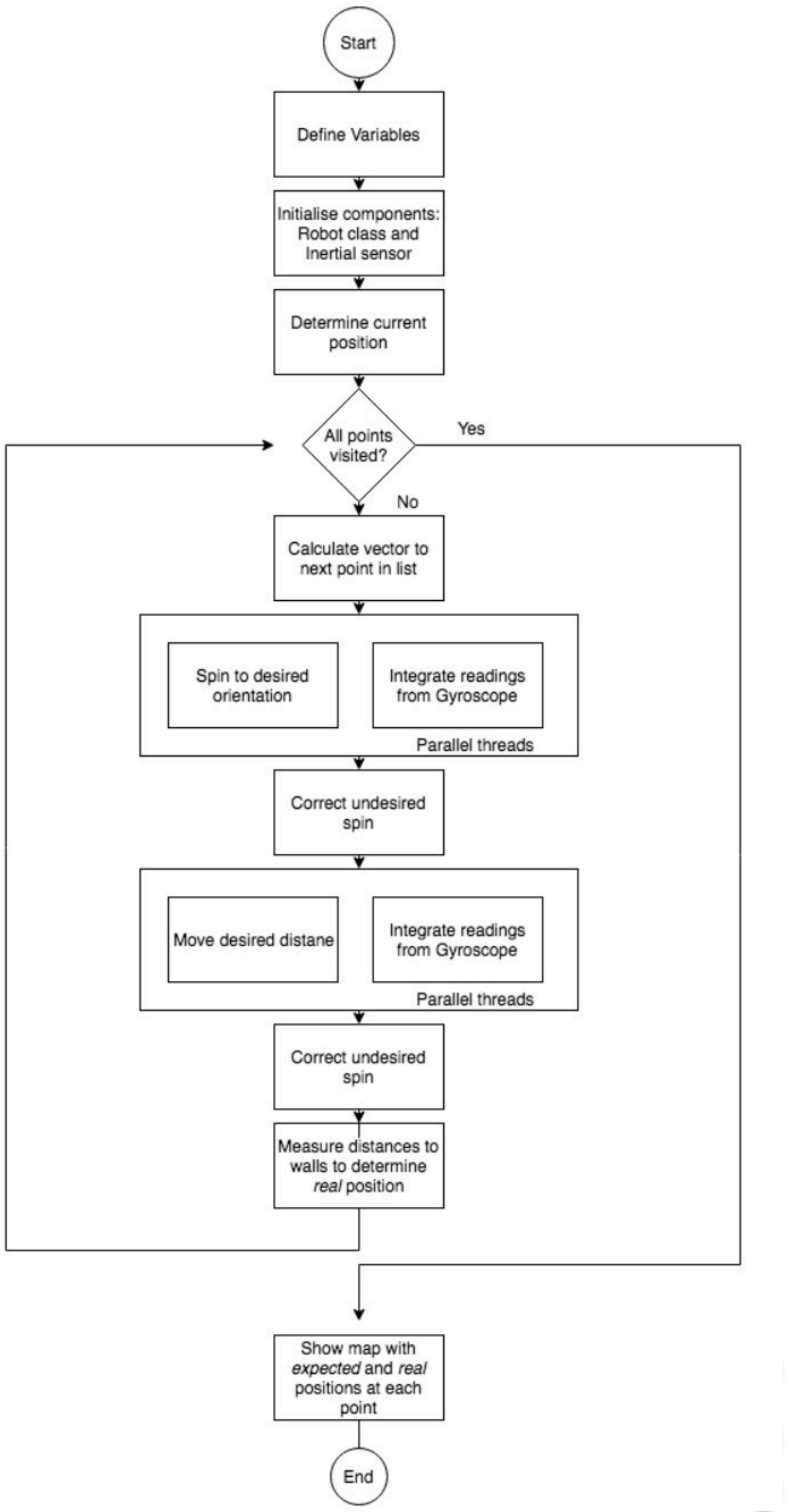
```
[125.0, 25.0],
[125.0, 75.0],
[225.0, 75.0],
[225.0, 125.0],
[275.0, 125.0],
[275.0, 25.0],
[25.0, 25.0],
[25.0, 125.0],
[125.0, 175.0],
[125.0, 75.0],
[125.0, 25.0],
[25.0, 25.0]]
```



All coordinates are in centimeters. Navigation between points is done by calculating movement vectors between the current position, stored in a 3-element array (x, y, θ)

Algorithm

A general description of the algorithm is shown in the following diagram:


Start
Define Variables
Initialise components: Robot class and Inertial sensor
Determine current position
All points visited?
Yes
No
Calculate vector to next point in list
Spin to desired orientation
Integrate readings from Gyroscope
Parallel threads
Correct undesired spin
Move desired distane
Integrate readings from Gyroscope
Parallel threads
Correct undesired spin
Measure distances to walls to determine real position
Show map with expected and real positions at each point
End

Initial Settings

The main variables defined here and used throughout the execution are defined globally such as robot position and the matrix containing some information about the map, such as distances to the walls of the landmark matrix. Aside from the object instantiation that control the robot movement through the motion class there is a two second delay for the calibration of the MPU6050. Worthy of mention is an approximately constant offset value obtained from the Gyroscope, calculated at around -1.29811 degrees per second. The position of the robot is also determined in this section.

Determining Initial Position

The robot starts with a given an orientation, in this case 90◦ using the global frame of reference. The robot does a 5-angle distance sweep at [0◦, 45◦, 90◦, 135◦, 180◦] to obtain a histogram that characterizes a location. This is the compared to the stored distances in optimal conditions to determine the starting point of the robot.

Driving to Next Position

It is assumed the position the robot starts in is exactly on the point the initial sweep determined. The destination point is obtained from the pointL array. The movement vector is then calculated using basic trigonometry and commanded to the robot using the motion library.

Determining the real position on arrival

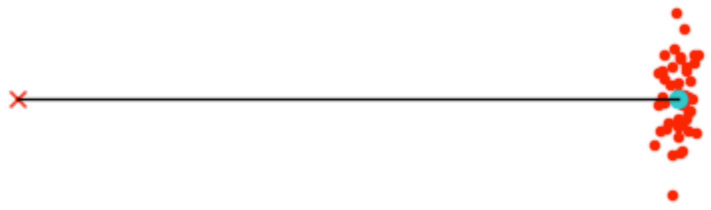
As soon as the vector is commanded, a parallel thread is started that reads and stores the information from the gyroscope. Once the destination point is reached, the readings from the gyroscope are integrated using a sampling time of 0.1 seconds.

To determine the spinning error, the spin read by the gyroscope can is subtracted from the commanded spin. This is used to estimate the orientation of the robot at the end of the execution of the vector movement. A correction of the spin is done in order to leave the robot on either 0◦, 45◦, 90◦, 135◦ or 180◦ orientation.

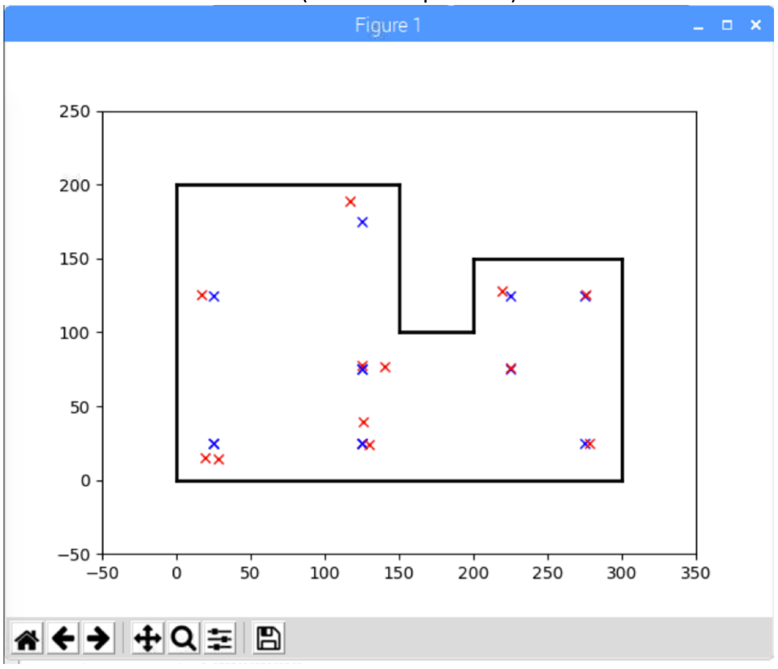
To determine the real position on the XY plane, a sweeping of the ultrasound sensor is done to determine where we actually landed. The possible position of the robot is calculated with a process similar to the Monte Carlo localization, where using a covariance matrix we obtain possible end positions. Since both the orientation is known and the distance to the walls in the expected landing position, we can determine which one of the calculated points is closer to reality.

The covariance matrix was calculated by commanding a straight line and measuring the deviation from the expected arrival point.

$$\begin{bmatrix} 0{,}13 & 0{,}0 \\ 0{,}0 & 0{,}27 \end{bmatrix}$$



With this information, a map can be generated. In blue we see the expected position after a run and in red the real (measured position).



A comparison between the real (measured position) and the real (photographed) can be done with the following image using position 5:

[Type here]

Developed Modules Installation

The following steps to set up a working environment are valid for both, the Raspberry Pi and a laptop/computer. Remember, the modules can be used on a laptop/computer for remote controlling the Raspberry Pis GPIOs or on the Raspberry Pi itself without any modifications of the source code of the modules. On the Raspberry Pi, the latest Raspbian version and afterwards all available updates should be installed. The latest Raspbian image can be downloaded from Raspbian Downloads.

Getting files

One option is creating a folder named 360pibot in the Raspberry Pis home folder

```
cd ~
mkdir 360pibot
```

and then connect the Raspberry Pi to the internet, open the projects github page, download the repository as a .zip file and unzip it in the created 360pibot folder.

Another option is to clone the git repository to the home folder. This will automatically create the 360pibot folder and download the project files. First, git will be installed, then the repository will be cloned.

```
sudo apt-get update
sudo apt-get install git
cd ~
git clone https://github.com/choeffer/360pibot
```

Both methods for getting the project files are also working on a laptop/computer.

Installing needed modules

Next step is to install the needed modules. They can be installed in the global Python 3 environment or in a virtual Python 3 environment. The latter has the advantage that the packages are isolated from other projects and also from the system wide installed global once. If things get messed up, the virtual environment can just be deleted and created from scratch again. First, installing with a virtual environment will be explained, afterwards with using the global Python 3 environment.

With a virtual environment

On a Raspberry Pi first ensure that the packages python3-venv and python3-pip are installed. This has also to be checked on Debian based distributions like Ubuntu/Mint.

```
sudo apt-get update
sudo apt-get install python3-venv python3-pip
```

Afterwards, navigate to the created 360pibot folder and create a virtual environment named venv and activate it. An activated virtual environment is indicated by a (venv) in the beginning of the terminal prompt.

```
cd ~
cd 360pibot
python3 -m venv venv
source venv/bin/activate
```

With the activated virtual environment install the needed pigpio module inside.

```
pip3 install pigpio
```

Deactivating the activated virtual environment can be done later by just typing deactivate in the terminal where the virtual environment is activated.

```
deactivate
```

Without a virtual environment

On a Raspberry Pi first ensure that the package python3-pip is installed. This has also to be checked on Debian based distributions like Ubuntu/Mint. Then, the pigpio module will be installed in the global Python 3 environment.

```
sudo apt-get update
sudo apt-get install python3-pip
pip3 install pigpio
```

Raspberry Pi

The following steps are specific to the Raspberry Pi. It is necessary to install the pigpio package, enable starting the pigpio daemon at boot and then doing a reboot to activate the pigpio daemon. For the demo implementation the package from the Raspbian repository is installed. This ensures that the package is good integrated in the system, even if it might be older.

```
sudo apt-get update
sudo apt-get install pigpio
sudo systemctl enable pigpiod
sudo reboot
```

If the Raspberry Pi's GPIOs are not responding anymore, it might help to restart the pigpio daemon on the Raspberry Pi. For that, SSH into the Raspberry Pi if remotely working with it, otherwise use the local terminal, and execute the following two commands.

```
sudo systemctl daemon-reload
sudo systemctl restart pigpiod.service
```

Hotspot and remote access

An important step which improves programming/controlling the Raspberry Pi is to make it remotely accessible. This can be done by connecting the Raspberry Pi to a WLAN network or by enabling a hotspot on it. This is recommended before using it. Setting up a hotspot will not be covered here, because the official documentation is good and updated regularly to match the latest Raspbian changes.

After enabling a hotspot on the Raspberry Pi and being connected with your laptop/computer, the following steps are needed to remote control the Raspberry Pi's GPIOs.

First, in the Raspberry Pi configuration Remote GPIO has to be enabled. This can be done via GUI or sudo raspi-config. This will allow remote connections while the pigpio daemon is running.

Then, the environment variable has to be set while or before launching Python 3 or an IDE. This variable will point to the IP address (and optional port) on which the Raspberry Pi is accessible. This can be on its own provided hotspot/network or on a WLAN it is connected to.

```
PIGPIO_ADDR=192.168.1.3 python3 hello.py
PIGPIO_ADDR=192.168.1.3 python3 code .
```

Implementation detail

Calibrating 360°servo

The following code calibrates a Parallax Feedback 360° High-Speed Servo 360_data_sheet. The values dcMin and dcMax are later needed in lib_motion.control . For more informations, see lib_para_360_servo. calibrate_pwm(). This example is included as calibrate.py .

Note: As seen in the example code, one pigpio.pi() instance can be passed to all used classes or modules. This reduces the number of parallel threads which gets started. If using one pigpio.pi() instance for each used class or module, so multiple instances in one script, more parallel threads will be started, which is not necessary.

```
import time
import pigpio
import lib_para_360_servo
#define GPIO for each servo to read from
gpio_l_r = 16
gpio_r_r = 20
#define GPIO for each servo to write to
```

```
gpio_l_w = 17
gpio_r_w = 27
pi = pigpio.pi()
#### Calibrate servos, speed  = 0.2 and -0.2
#choose gpio_l_w/gpio_l_r (left wheel), or gpio_r_w/gpio_r_r
#(right wheel) accordingly
servo = lib_para_360_servo.write_pwm(pi = pi, gpio = gpio_r_w)
#buffer time for initializing everything
time.sleep(1)
servo.set_speed(0.2)
wheel = lib_para_360_servo.calibrate_pwm(pi = pi, gpio = gpio_r_r)
servo.set_speed(0)
#http://abyz.me.uk/rpi/pigpio/python.html#stop

pi.stop()
```

In this case, for the left wheel for duty_cycle_min / dcMin 27.3 should be chosen, so the smallest out of 27.3 and 31.85. For duty_cycle_max / dcMax 969.15 should be chosen, so the biggest out of 964.6 and 969.15. For the right wheel, for duty_cycle_min / dcMin 27.3 and for duty_cycle_max / dcMax 978.25 accordingly.

Emergency stop

The following code sets the speed of the two used Parallax Feedback 360° High-Speed Servos 360_data_sheet back to zero to stop both wheels. This might be needed e.g. if a script raises an exception and stops executing before setting the speed of the servos back to zero. In this case, the servos will continue rotating with the last set speed. This example is included as emergency_stop.py .

```
import time
import pigpio
import lib_para_360_servo
#define GPIO for each servo to write to
gpio_l = 17
gpio_r = 27
pi = pigpio.pi()
servo_l = lib_para_360_servo.write_pwm(pi = pi, gpio = gpio_l)
servo_r = lib_para_360_servo.write_pwm(pi = pi, gpio = gpio_r)
#buffer time for initializing everything
time.sleep(1)
servo_l.set_speed(0)
servo_r.set_speed(0)
#http://abyz.me.uk/rpi/pigpio/python.html#stop

pi.stop()
```

Moving the robot

The following code lets the robot turning four times 45 degree to the left, then moving 20 cm (200 mm) forwards, then 20 cm backwards and in the end turning two times 90 degree to the right. This example is included as move_robot. py.

Note: The PID controller values have a strong influence on the robot’s movement. Therefore, try out different values than the default once if the robot is not moving as expected and also to get a feeling of their influence.

[Type here]

Note: Sometimes the end of a movement takes some time. One reason could be noise in the position measurement of the wheels and therefore not correctly recognizing the reached set-point. So, do not be confused if this happens and the robot takes some time and seems not to proper function anymore. Just be patient. This might be solved with increasing the deadband of the outer PID controllers, which on the other hand would also decrease the accuracy of the movement. At the moment a rather small deadband is chosen. A simple but effective workaround is to give the robot a little poke and let it reach the set-point again. See lib_motion.control.move() for more informations.

Note: Sometimes the speed changes abruptly. This might be caused by noise in the position measurement of the wheels from which the rotation speed measurement (ticks/s) is calculated indirectly. Increasing the sliding window size might stabilize it. But this would also increase the delay between calculated and real speed of the wheels and therefore also increase the response time of the speed controllers. At the moment a rather small window size is chosen. See lib_motion.control.move() for more information.

```
import pigpio
import lib_motion
pi = pigpio.pi()
robot = lib_motion.control(pi = pi)

a=0
while a < 4:

   robot.turn(45)
   a+=1
robot.straight(200)
robot.straight(-200)

a=0
while a < 2:

   robot.turn(-90)
   a+=1
#http://abyz.me.uk/rpi/pigpio/python.html#callback
robot.cancel()
#http://abyz.me.uk/rpi/pigpio/python.html#stop

pi.stop()
```

Moving standard servo

The following code steers the standard servo stand_data_sheet2 . First to the middle position, then to the max right, then to max left and finally to 45 degree (regarding reached max left and max right). For more information, see lib_scanner.para_standard_servo . This example is included as move_stand_servo.py .

```
import time
import pigpio
import lib_scanner
pi = pigpio.pi()
servo = lib_scanner.para_standard_servo(pi = pi, gpio = 22)

servo.middle_position()
time.sleep(2)
servo.max_right()
time.sleep(2)
servo.max_left()
time.sleep(2)
servo.set_position(degree = 45)
#http://abyz.me.uk/rpi/pigpio/python.html#stop

pi.stop()
```

Scanning the surrounding

The following code scans the surrounding of the robot in all five default angles and prints out the result. This example is included as scanning.py .

Warning: Make sure that the min_pw and max_pw values are carefully tested before using this example, see **Warning** in lib_scanner.para_standard_servo . The passed values min_pw and max_pw for the created ranger object are just valid for the demo implementation!

```
import pigpio
import lib_scanner
pi = pigpio.pi()
"""
.. warning::
   Make sure that the ``min_pw`` and ``max_pw`` values are carefully tested
   **before** using this example, see **Warning** in
   :class:`lib_scanner.para_standard_servo` . The passed values ``min_pw``
   and ``max_pw`` for the created ranger object are just valid for the
   demo implementation!
"""
ranger = lib_scanner.scanner(pi = pi, min_pw=600, max_pw=2350)
distances = ranger.read_all_angles()

print(distances) #http://abyz.me.uk/rpi/pigpio/python.html#callback

ranger.cancel()
#http://abyz.me.uk/rpi/pigpio/python.html#stop

pi.stop()
```

Simple collision avoiding algorithm

The following code implements a simple collision avoiding algorithm. The robot will turn 45 degree to the left if there is any obstacle closer than 40 cm at the five default measuring angles. If not, the robot will drive 20 cm forward. This example is included as no_collision.py .

[Type here]

Warning: Make sure that the min_pw and max_pw values are carefully tested before using this example, see **Warning** in lib_scanner.para_standard_servo . The passed values min_pw and max_pw for the created ranger object are just valid for the demo implementation!

360pibot API

In this section the use of each module is documented. In the source code it is commented, if necessary, how each part of the code is working and what it is intended to do.

lib_scanner

Module for making measurements with a HC-SR042 ultrasonic sensor and rotating it with a Parallax Standard Servo stand_data_sheet.

This module includes three classes. One for making the measurements with an HC-SR042 ultrasonic sensor lib_scanner.hcsr04, one for searing a Parallax Standard Servo stand_data_sheet4 lib_scanner.para_standard_servo and one which combines the first two to scan the surrounding lib_scanner. scanner .

class lib_scanner.**hcsr04**(pi,trigger,echo,pulse_len=15)

Makes measurements with a HC-SR042 ultrasonic sensor.

This class allows making measurements with a HC-SR042 ultrasonic sensor. A trigger signal will be sent to a defined GPIO Pin trigger and a PWM will be received on a defined GPIO Pin echo. With the received PWM the distance to an object is calculated.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.
- **trigger** (int) – GPIO identified by their Broadcom number, see elinux.org. To this

  GPIO the trigger pin of the HC-SR042 has to be connected.

- **echo** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO

  the echo pin of the HC-SR042 has to be connected.

- **pulse**_len (int,float) – Defines in microseconds the length of the pulse which is sent on the trigger GPIO. Default: 15, taken from the data sheet (10µs) and added 50%, to have a buffer to surely trigger the measurement.

cancel()
Cancel the started callback function.

This method cancels the started callback function if initializing an object. As written in the pigpio callback3 documentation, the callback function may be cancelled by calling the cancel function after the created instance is not needed anymore.

read(temp_air=20, upper_limit=4, number_of_sonic_bursts=8, added_buffer=2, debug=False) Measures the distance to an object.

This method triggers a measurement, does all the calculations and returns the distance in meters.

Parameters

- **temp_air** (int,float) – Temperature of the air in degree Celsius. Default: 20.
- **upper_limit** (int,float) – The upper measurement limit in meters. Default: 4, upper limit taken from the data sheet HC-SR042.
- **number_of_sonic_bursts** (int) – The number of sonic bursts the sensor will make. Default: 8, taken from the data sheet HC-SR042.
- **added_buffer** (int,float) – The added safety buffer for waiting for the distance measurement to complete. Default: 2, so 100% safety buffer.
- **debug** (bool) – Controls if debugging printouts are made or not. For more details, have a look at the source code. Default: False, so no printouts are made.

Returns Measured distance in meters.

Return type float

class lib_scanner.**para_standard_servo**(pi, gpio, min_pw=1000, min_degree=-90, max_degree=90, max_pw=2000)

This class steers a Parallax Standard Servo and should also work with other servos which have a 50Hz PWM for setting the position. The position of the Parallax Standard Servo can be set between -90 (min_degree) and +90 (max_degree) degree.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.
- **gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the signal wire of the servo has to be connected.
- **min_pw** (int) – Min pulsewidth, see Warning, carefully test the value before! Default: 1000, taken from set_servo_pulsewidth.
- **max_pw** (int) – Max pulsewidth, see Warning, carefully test the value before! Default: 2000, taken from set_servo_pulsewidth.
- **min_degree** (int) – Min degree which the servo is able to move. Default: -90, taken from stand_data_sheet.
- **max_degree** (int) – Max degree which the servo is able to move. Default: +90, taken from stand_data_sheet.
- **max_left**() - Sets the position of the servo to -90 degree, so min_degree (max left, counter-clockwise).
- **max_right**() - Sets the position of the servo to 90 degree, so max_degree (max right, clockwise).
- **middle_position**() - Sets the position of the servo to 0 degree, so middle position.
- **set_position(**degree) - Sets position of the servo in degree.

  This method sets the servo position in degree. Minus is to the left, plus to the right. **Parameters degree** (int,float) – Should be between min_degree (max left) and max_degree (max right), otherwise the value will be limited to those values.

- **set_pw**(pulse_width) - Sets pulsewidth of the PWM.

  This method allows setting the pulse width of the PWM directly. This can be used to test which min_pw and max_pw is appropriate. For this the min_pw and max_pw are needed to be set very small and very big, so that they do not limit the set pulse width. Normally they are used to protect the servo by limiting the pulse width to a certain range.

- **Parameters pulsewidth** (int,float) – Pulsewidth of the PWM signal. Will be limited to min_pw and max_pw.

class lib_scanner.**scanner**(pi, trigger=6, echo=5, pulse_len=15, temp_air=20, upper_limit=4, number_of_sonic_bursts=8, added_buffer=2, gpio=22, min_pw=1000, max_pw=2000, min_degree=-90, max_degree=90, angles=[-90, -45, 0, 45, 90], time_servo_reach_position=3, debug=False)

Scans the surrounding with the help of the hcsr04 and para_standard_servo classes.

This class steers the servo position and triggers measurements with the ultrasonic sensor. With a passed list, measurements will be made at the defined positions. A dict will be returned with the measured distances at the defined positions.

Parameters

pi (pigpio.pi) – Instance of a pigpio.pi() object.

trigger (int) – GPIO identified by their Broadcom number, see elinux.org . To this

GPIO the trigger pin of the HC-SR04 has to be connected.
echo (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO

the echo pin of the HC-SR04 has to be connected.

temp_air (int,float) – Temperature of the air in degree celsius. Default: 20.

upper_limit (int,float) – The upper measurement limit in meters. Default: 4, upper limit taken from the data sheet HC-SR04.

number_of_sonic_bursts (int) – The number of sonic bursts the sensor will make. Default: 8, taken from the data sheet HC-SR04 .

added_buffer (int,float) – The added safety buffer for waiting for the distance measurement. Default: 2, so 100% safety buffer.

gpio (int) – GPIO identified by their Broadcom number, see elinux.org[1] . To this GPIO the signal wire of the servo has to be connected.

min_pw (int) – Min pulsewidth, see Warning, carefully test the value before! Default: 1000, taken from set_servo_pulsewidth.

max_pw (int) – Max pulsewidth, see Warning, carefully test the value before! Default: 2000, taken from set_servo_pulsewidth .

min_degree (int) – Min degree which the servo is able to move. Default: -90, taken from stand_data_sheet .

max_degree (int) – Max degree which the servo is able to move. Default: +90, taken from stand_data_sheet .

angles (list) – List of positions where the servo moves to and the ultrasonic sensor will make measurements.

time_servo_reach_position (int,float) – Time in seconds to wait until the servo moves from one to another position. This needs to be tested for each servo. Default: 3, this should be sufficient to safely (incl. lot of safety buffer) reach each position before the measurement is made. If the servo is not oscillating after reaching each position, even a value of 0.35 was working fine with the demo implementation.

debug (bool) – Controls if debugging printouts and measurements are made or not. For more details, have a look at the source code. Default: False, so no printouts and measurements are made.

pulse_len (int,float) – Defines in microseconds the length of the pulse, which is sent on the trigger GPIO. Default: 15, taken from the data sheet (10µs) and added 50%, to have a buffer to surely trigger the measurement.

cancel()
Cancel the started callback function.

This method cancels the started callback function if initializing an object. As written in the pigpio callback documentation, the callback function may be cancelled by calling the cancel function after the created instance is not needed anymore.

read_all_angles()
Moves servo and makes measurements at every defined position.

This method moves the servo to every position defined in list angles, makes a measurement there and afterwards returns a dict with the distance in meter for every position.

Returns Measured distances in meters for each position defined in angles.

Return type dict

lib_para_360_servo

Module for setting the speed and reading the position of a Parallax Feedback 360° High-Speed Servo 360_data_sheet.

This module includes three classes. One for setting the speed lib_para_360_servo.write_pwm , one for reading the position lib_para_360_servo.read_pwm and one for calibrating a servo to determine the appropriate dcMin / dcMax values needed in lib_motion lib_para_360_servo.calibrate_pwm .

class lib_para_360_servo.**calibrate_pwm**(pi,gpio,measurement_time=120) Calibrates a Parallax Feedback 360° High-Speed Servo with the help of the read_pwm class.

This class helps to find out the min and max duty cycle of the feedback signal of a servo. This values (dcMin / dcMax) are then needed in lib_motion to have a more precise measurement of the position. The experience has shown that each servo has slightly different min/max duty cycle values, different than the once provided in the data sheet 360_data_sheet. Values smaller and bigger than the printed out once as "duty_cycle_min/duty_cycle_max" are outliers and should therefore not be considered. This can be seen in the printouts of smallest/biggest 250 values. There are sometimes a few outliers. Compare the printouts of different runs to get a feeling for it.

Note: The robot wheels must be able to rotate free in the air for calibration. Rotating forward or backward might sometimes give slightly different results for min/max duty cycle. Choose the smallest value and the biggest value out of the forward and backward runs. Do both directions three times for each wheel, with speed = 0.2 and -0.2. Then chose the values.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.

- **gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the feedback wire of the servo has to be connected.
- **measurement_time** (int,float) – Time in seconds for how long duty cycle values will be collected, so for how long the measurement will be made. Default: 120. Returns Printouts of different measurements

  At the moment, the period for a 910 Hz signal is hardcoded, as in read_pwm() .

class lib_para_360_servo.**read_pwm**(pi,gpio)

Reads position of a Parallax Feedback 360° High-Speed Servo 360_data_sheet. This class reads the position of a Parallax Feedback 360° High-Speed Servo. At the moment, the period for a 910 Hz signal is hardcoded.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.
- **gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the feedback wire of the servo has to be connected.

cancel()

Cancel the started callback function.

This method cancels the started callback function if initializing an object. As written in the pigpio callback documentation, the callback function may be cancelled by calling the cancel function after the created instance is not needed anymore.

read()
Returns the recent measured duty cycle.

This method returns the recent measured duty cycle. Returns Recent measured duty cycle Return type float

class lib_para_360_servo.**write_pwm**(pi, gpio, min_pw=1280, max_pw=1720, min_speed=-1, max_speed=1)

Steers a Parallax Feedback 360° High-Speed Servo 360_data_sheet .

This class steers a Parallax Feedback 360° High-Speed Servo. Out of the speed range, defined by min_speed and max_speed, and the range of the pulsewidth, defined by min_pw and max_pw, the class allows setting the servo speed and automatically calculates the appropriate pulsewidth for the chosen speed value.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.
- **gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the control wire of the servo has to be connected.
- **min_pw** (int) – Min pulsewidth, see Warning, carefully test the value before! Default: 1280, taken from the data sheet 360_data_sheet.
- **max_pw** (int) – Max pulsewidth, see Warning, carefully test the value before! Default: 1720, taken from the data sheet 360_data_sheet.
- **min_speed** (int) – Min speed which the servo is able to move. Default: -1.
- **max_speed** (int) – Max speed which the servo is able to move. Default: 1.

max_backward()
Sets the speed of the servo to -1, so min_speed (max backwards, counter-clockwise)

max_forward()
Sets the speed of the servo to 1, so max_speed (max forward, clockwise)

set_pw(pulse_width)
Sets pulsewidth of the PWM.

This method allows setting the pulsewidth of the PWM directly. This can be used to test which min_pw and max_pw are appropriate. For this the min_pw and max_pw are needed to be set very small and very big, so that they do not limit the set pulsewidth. Normally they are used to protect the servo by limiting the pulsewidth to a certain range.

Parameters **pulsewidth** (int,float) – Pulsewidth of the PWM signal. Will be limited to min_pw and max_pw.

set_speed(speed)
Sets speed of the servo.

This method sets the servos rotation speed. The speed range is defined by by min_speed and max_speed .

Parameters **speed** (int,float) – Should be between min_speed and max_speed , oth- erwise the value will be limited to those values.

stop()
Sets the speed of the servo to 0.

lib_motion

Module for moving the robot.
This module includes the method lib_motion.control.move() which is the core of the movement controlling.

The module imports lib_para_360_servo .

class lib_motion.**control**(pi,width_robot=102,diameter_wheels=66,unitsFC=360,dcMin_l=27.3, dcMax_l=969.15, dcMin_r=27.3, dcMax_r=978.25, l_wheel_gpio=16, r_wheel_gpio=20, servo_l_gpio=17, min_pw_l=1280, max_pw_l=1720, min_speed_l=-1, max_speed_l=1, servo_r_gpio=27, min_pw_r=1280, max_pw_r=1720, min_speed_r=-1, max_speed_r=1, sam- pling_time=0.01, Kp_p=0.1, Ki_p=0.1, Kd_p=0, Kp_s=0.5, Ki_s=0, Kd_s=0)

Controls the robot movement.

This class controls the robot’s movement by controlling the two Parallax Feedback 360° High-Speed Servos 360_data_sheet. The robots local coordinate system is defined in the following way. X-axis positive is straight forward, y-axis positive is perpendicular to the x-axis in the direction to the left wheel. The center (0/0) is where the middle of the robots chassis is cutting across an imaginary line through both wheels/servos. Angle phi is the displacement of the local coordinate system to the real-world coordinate system. See Used local coordinate system for a picture of it.

Parameters

- **pi** (pigpio.pi) – Instance of a pigpio.pi() object.

- **width_robot**(int)–Width of the robot in mm, so distance between middle right wheel and middle left wheel. Default: 102 mm, measured.
- **diameter_wheels** – Diameter of both wheels. Default: 66 mm, measured and taken from the products website wheel_robot.
- **unitsFC** (int) – Units in a full circle, so each wheel is divided into X sections/ticks. This value should not be changed. Default: 360
- **dcMin_l** (float) – Min duty cycle of the left wheel. Default: 27.3, measured with method lib_para_360_servo.calibrate_pwm() .
- **dcMax_l** (float) – Max duty cycle of the left wheel. Default: 969.15, measured with method lib_para_360_servo.calibrate_pwm().
- **dcMin_r** (float) – Min duty cycle of the right wheel. Default: 27.3, measured with method lib_para_360_servo.calibrate_pwm().
- **dcMax_r** (float) – Max duty cycle of the left wheel. Default: 978.25, measured with method lib_para_360_servo.calibrate_pwm() , see Examples .
- **l_wheel_gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the feedback wire of the left servo has to be connected.
- **r_wheel_gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the feedback wire of the right servo has to be connected.
- **servo_l_gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the control wire of the left servo has to be connected.
- **min_pw_l**(int)–Min pulse width, carefully test the value before! Default: 1280, taken from the data sheet 360_data_sheet.
- **max_pw_l** (int) – Max pulsewidth, see Warning, carefully test the value before! De- fault: 1720, taken from the data sheet 360_data_sheet.
- **min_speed_l** (int) – Min speed which the servo is able to move. Default: -1, so that the speed range is also scaled between -1 and 1 as the output of the inner control loop.
- **max_speed_l** (int) – Max speed which the servo is able to move. Default: 1, so that the speed range is also scaled between -1 and 1 as the output of the inner control loop.
- **servo_r_gpio** (int) – GPIO identified by their Broadcom number, see elinux.org. To this GPIO the control wire of the right servo has to be connected.
- **min_pw_r**(int)–Min pulse width, carefully test the value before! Default: 1280, taken from the data sheet 360_data_sheet .
- **max_pw_r** (int) – Max pulse width, carefully test the value before! Default: 1720, taken from the data sheet 360_data_sheet .
- **min_speed_r** (int) – Min speed which the servo is able to move. Default: -1, so that the speed range is also scaled between -1 and 1 as the output of the inner control loop.
- **max_speed_r** (int) – Max speed which the servo is able to move. Default: 1, so that the speed range is also scaled between -1 and 1 as the output of the inner control loop.
- **sampling_time** (float) – Sampling time of the four PID controllers in seconds. De- fault: 0.01. 1. PWM of motor feedback is 910Hz (0,001098901 s), so position changes cannot be recognized faster than 1.1 ms. Therefore, it is not needed to run the outer control loop more often and update the speed values which have a 50 Hz (20ms) PWM. 2. Tests of the runtime of the code including the controller part have shown that writing the pulse width (pi.set_servo_pulsewidth()) in lib_para_360_servo.write_pwm. set_pw() is the bottleneck which drastically slows down the code by the factor ~400 (0,002 seconds vs 0,000005 seconds; runtime with vs without writing pulsewidth). 3. For recognizing the RPMs of the wheels 10ms is needed to have enough changes in the position. This was found out by testing. See method move() for more information.

- **Kp_p**(int,float)–KpvalueoftheouterPIDcontrollers,seemethodmove()formore informations. Default: 0.1.
- **Ki_p**(int,float)–KivalueoftheouterPIDcontrollers,seemethodmove()formore informations. Default: 0.1.
- **Kd_p**(int,float)–KdvalueoftheouterPIDcontrollers,seemethodmove()formore informations. Default: 0.
- **Kp_s**(int,float)–KpvalueoftheinnerPIDcontrollers,seemethodmove()formore informations. Default: 0.5.
- **Ki_s**(int,float)–KivalueoftheinnerPIDcontrollers,seemethodmove()formore informations. Default: 0.
- **Kd_s**(int,float)–KdvalueoftheinnerPIDcontrollers,seemethodmove()formore informations. Default: 0.

cancel()
Cancel the started callback function.

This method cancels the started callback function if initializing an object. As written in the pigpio callback documentation, the callback function may be cancelled by calling the cancel function after the created instance is not needed anymore.

move(number_ticks=0, straight=False, turn=False) Core of motion control.

This method controls the movement of the robot. It is called from lib_motion.control.turn() or lib_motion.control.straight() and is not ment to be called directly. Four digital PID con- trollers are used to make two cascade control loops, one cascade control loop for each wheel. Each cascade control loop has the same parameters (P/I/D parameters), so that both wheels are controlled in the same way. Chosen default: Outer control loop is a PI controller; inner control loop is a P controller.

The outer loop is a position controller, the inner loop a speed controller. After both wheels have reached their set- point (position), the method waits one second before the movement is marked as finished.

This ensures that overshoots/oscillations are possible and that both wheels can independently reach their set-point (position).

The I part of each PID controller is limited to -1 and 1, so that the sum of the errors is not integrated till in- finity which means to very high or low values which might cause problems.

The output value of each inner PID controller is scaled between -1 and 1 and the output value of each outer PID controller is limited to -1 and 1. This ensures that no scaling factors are introduced in the P/I/D parameters and also that the output of each PID controller matches the speed range of the servos, defined in lib_para_360_servo. write_pwm.set_speed() .

A sliding median window is used to filter out the noise in the rotation speed measurement (ticks/s) which is done indirectly by measuring the position of the servo. Also, a dead- band filter after the error calculation of the outer control loop is implemented.

This adjustment helps to make the controllers more stable, e.g. filter out outliers while calculating the rotation speed and therefore avoid high value changes/jumps or avoid oscillations after reaching the set-point (position). The sample time of the digital PID controllers can also be freely chosen and does not influence the P/I/D parameters, the rotation speed measurement or the time before the movement is marked as finished.

Parameters

- **number_ticks** (int,float) – Number of ticks the wheels have to move.
- **straight** (bool) – True or False, if robot should move straight. Default: False.
- **turn** (bool) – True or False, if robot should turn. Default: False.

straight(distance_in_mm)
Moves the robot about x mm forward or backward.

This method moves the robot x mm forward or backward. Positive distance values move the robot forward (regarding the local x-axis), negative distance values backward (regarding the local x-axis), see picture in Used local coordinate system, where the local coordinate system of the robot is shown. This method calls lib_motion.control.move() which controls the movement of the robot.

Parameters **distance_in_mm** (int,float) – Distance the robot has to move. **turn**(degree)

Turns the robot about x degree.

This method turns the robot x degree to the left or to the right. Positive degree values turn the robot to the left, negative degree values to the right, see picture in Used local coordinate system, where the local coordinate system of the robot is shown. This method calls lib_motion.control.move() which controls the movement of the robot.

Parameters **degree** (int,float) – Degree the robot has to turn.

Conclusion

The experimental runs done at the lab showed that the current state of the robot allows for precise location only for a limited number of steps. After arriving at 7 or 8 points, the accuracy of the estimation is not enough to avoid getting close to walls. The proximity to walls hinders the robot's ability to turn and causes further error in the location approximation.

Further work can be done to give the robot "smarter" location capabilities such as a Kalman filter or another sensor like a camera. The Raspberry Pi facilitates further sensor integration both in the form of additional GPIO pins nor currently used or the USB ports still available.

The programmed libraries were done in a robust way so that they can be easily used by similar projects using similar hardware. They are installed as a regular python library and can be parametrized so use different pins as the ones picked for this implementation.