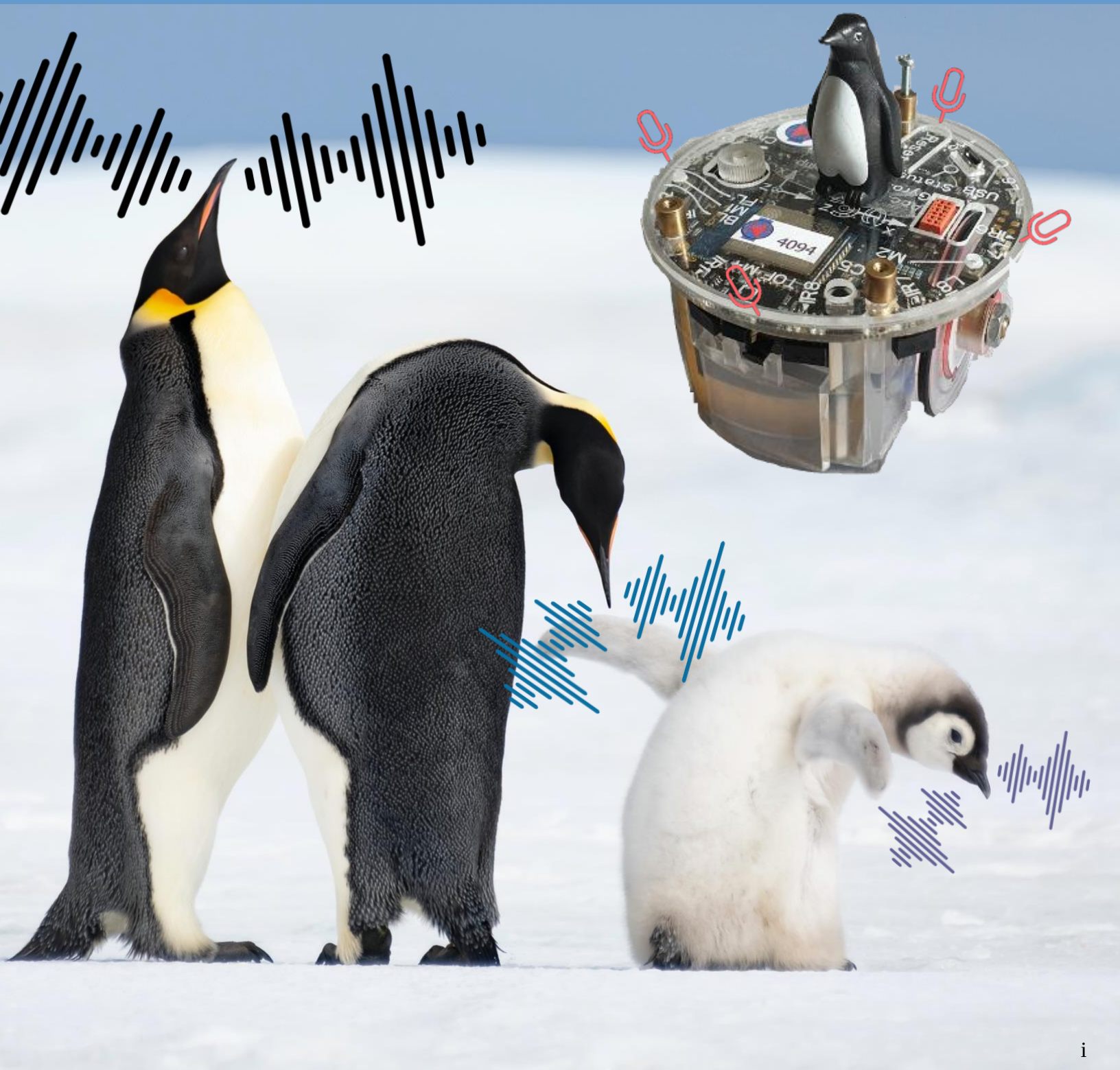
EPFL

Project « Microinformatique » spring 2020



Penguins

Built on an ePuck2 : http://www.e-puck.org

Authors : N. Schmid & T. Mayaud

TABLE OF CONTENTS

1. INTRODUCTION

1.1. A penguin story

Near the southern pole, living in a cold, cold world, live the emperor penguins. These penguins find a way to survive in their glacial environment, using their most critical social skills. One critical capability to survive, is being able to protect and feed their chicks (children). The emperor penguin chicks communicate with simple sounds, which the parents must recognize, locate, and go to.

Image 1 - Emperor penguins and its chicks[ii]

This behavior is what we set to emulate with the ePuck2 robot. Will penguins be beat at their own skill ? Read on to find out !

1.2. Technical implications

As different babies emit different frequencies, we will have multiple sources of sound, each at a specific, constant frequency. Using the four onboard microphones, our penguin impostor will find the direction of each individual simulated chick sound. This direction is an angle along the same plane as the four microphones. We should thus be able to navigate to any source, as long as it is on the same plane as our robot.

First, the robot will identify the different sources. Then, it will prompt us to input which of the sources it should go towards, and navigate towards it. The user interface is the terminal of a computer and as communication medium, Bluetooth is used with the serial protocol. There should be no intermediary obstacles, or the robot will just stop at the first one it finds on the way. If during one of the audio scans a killer whale is detected, the robot will try to escape from it by going into the opposite direction than that of the killer whale.

Once one of the sources is reached, the robot will stop, and wait for us to command it to go towards another of the sources it hears.

1.3. Work process overview

For this project, we will use the “Microtechnique” course tools, which include a customized and pre-configured eclipse IDE, and a GitHub repository at the following link : https://github.com/theophanemayaud/MTBA6MicroinfPinguins

As the COVID-19 crisis requires, we must do everything remotely. Fortunately, one of us two had the robot with him as EPFL was closed down. We split the work and did most of the testing through zoom using remote computer control. We explain in more details how this went on, in the result section.

LET THE RACE BETWEEN PENGUIN AND ROBOT BEGIN

The natural evolution of penguins 🐧; the silicon industry revolution 🤖; this contest will be hard and will test our skills, for we must show what we have learned as future EPFL engineers !

2. TECHNICAL REALISATIONS

2.1. General code structure

Our code has two main parts : the first part is the audio analysis, where we identify the different sources and their direction using the four microphones. The second part is the control of the movements of the robot, based on the directions given by part one, and also the distance to the obstacle given by the time of flight sensor. We make use of these two parts from the main.c file, of which you can see the main function "call graph" in Figure 1. We start by finding the different sources, and then ask the user which source he wants the robot to go to. The audio_processing.c library handles finding the sources and calculating their angles

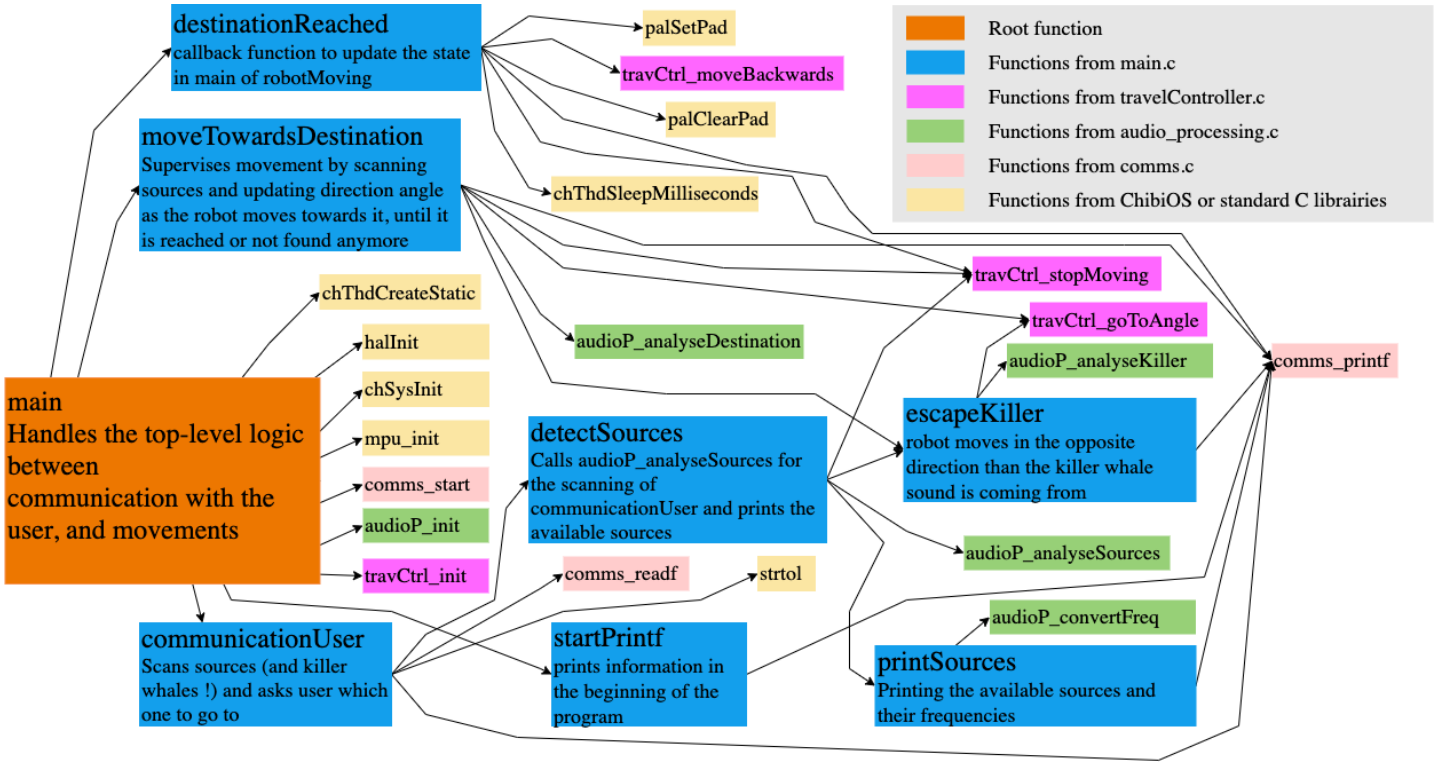


Figure 1 - main function call graph, as an overview of the general code architecture

MODULES USED

- **2 stepper motors** : controlled for direction by differential speed using a P structure and the angle of the sound source as input. Also controlled for movement using P controller with distance to obstacle as input.
- **4 microphones** : used to detect and locate multiple mono frequency sound sources, with Fourier transforms for phase shifts calculation between microphones and peak intensity identification.
- **Time-of-flight sensor (TOF)** : for distance measurement in order to stop before an obstacle.
- **Bluetooth module** : ask user which sound source to go to. Wireless required for mobility.
- **LEDs** : to signal when the source is reached or when a killer whale is detected.
- **Infrared proximity sensors** : backup to TOF for better detection of close obstacle to stop.

List 1 - modules of the epuck2 we used for our penguin

with respect to the robot. The travelController.c library handles the motors and their speeds depending on the angle and the distance to obstacles. We explain more in detail these two libraries in the following sections. We also programmed a third comms.c library which handles the communications via serial Bluetooth UART to the computer, displaying and reading user input. This library isn't explained in detail in a specific section, because it is short and simple, wrapping already existing functions for sending data, and reading data with the added specificity of checking the data.
In our code, we used many ChibiOS libraries, in order to use the ePuck modules easily. You can see what modules we used in List 1. Each of these modules uses a dedicated thread, which is created by ChibiOS, except for the LEDs as they do not need one. We actually create a thread for them to blink in a specific way, but this was just easier for one part of our application as we wanted good timing. From the main function, we initiate the other modules, then handle alternating between : asking the user which source to go to, then going to that source. We created some functions in main.c to simplify the main function and tie the sound analysis and the movements together, where it wasn't a task specifically tied to one or the other. It also ties in the communication with the user, which neither of the two libraries do.

2.2. Set up

To be able to analyze the audio input we have to know what kind of sound source we are dealing with. This is crucial because we have to treat the sound signal before we analyze it, such that we are able to filter the noise. Therefore, we make the following five assumptions for our sources, and about how our penguins are communicating:

1. The source emits a sinus wave at a specific frequency. Because we know that a sinus wave is in theory a dirac signal in the frequency domain, this way we simplify the audio analysis. In practice the dirac is a peak with high amplitude at the specific frequency that can be well distinguished from other signals.
2. The source emits constantly. A constant signal is easier to treat than an irregular one, because it is more predictable and the source should always be measured.
3. The sources are placed on a flat surface, because in the Antarctic where the penguins live, the usual environment is a flat ice desert. Therefore we do not have to deal with objects that have to be avoided on the way to the source, and we know that the sound can propagate in a more linear way, without reflections or deviations.
4. The sources are closer to the robot than a maximal radius. This is important because then we can filter all signals with a lower amplitude than a specific threshold we define.
5. The killer whales emit a sound at the frequency of 1000Hz. We chose this specific frequency so that the detection is easier. If we had chosen a larger bandwidth of frequencies, the chance for a wrong detection would have increased significantly (especially when for example someone is talking).

In practice we take the speakers from our smartphones as sound source. We are using the free application called *Tuning Fork*[iii] downloaded from the android store. It is a simple app which can emit a sound signal at a specific frequency.

2.3. Audio

2.3.1. Functions called from main

In audio there are three important functions that are called from main (see Figure 1):

audioP_analyseSources is called when the user has to choose a destination source. *audioP_analyseDestination* is the function which updates the angle of the destination, when the robot is moving towards a source. And *audioP_analyse_Killer* is called if a killer whale is detected, to escape from it. All three functions call *audio_analyseSpectre first,* which determines all peaks from the frequency specter (see chapter 2.3.3). They then call *audio_determineAngle* which calculates the angle of a specific source (see chapter 2.3.4). If either in *audioP_analyseSources* or in *audioP_analyseDestination* a frequency of 1000Hz is detected (which corresponds to a killer whale) the constant *AUDIOP__KILLER WHALE_DETECTED* is returned to main where the function *escapeKiller* will lead the robot away from the killer whale source. If in *audioP_analyseSources* the source is not found or the angle calculation returns errors for multiple times (defined by *NB_ERROR_DETECTED_MAX*), the constant *AUDIOP_SOURCE_NOT_FOUND* is returned to main. Then the program starts again by analyzing the sources and asking the user to which source the robot should go. The other two functions called from main are *audioP_convertFreq* and *audioP_init*: *audioP_convertFreq* is used to convert the frequency from the FFT-domain into a real frequency (see chapter 2.3.2) and *audioP_init* starts the processing thread of the microphones.

2.3.2. Frequency conversion

The implementation of the FFT operates with 1024 sampling points. The frequency which the algorithm returns is not given in Hertz but in another unit. Therefore, we have to convert first the frequency to get a value in Hz – the scale that we are used to. By an experiment where we gave, as input to the robot defined frequencies, and measured the corresponding output values, we determined the formula for the conversion (see Figure 2.). It is a linear curve and represented in Figure 2. The R-squared value of the trend line is equal to one, which means that it is a perfect approximation. In our program we memorize the frequencies always in the FFT-scale. Only if we are printing out a frequency for the user or we are handing a frequency as argument to *audio_ConvertPhase* for the angle calculation, we are converting the frequency into Hz.

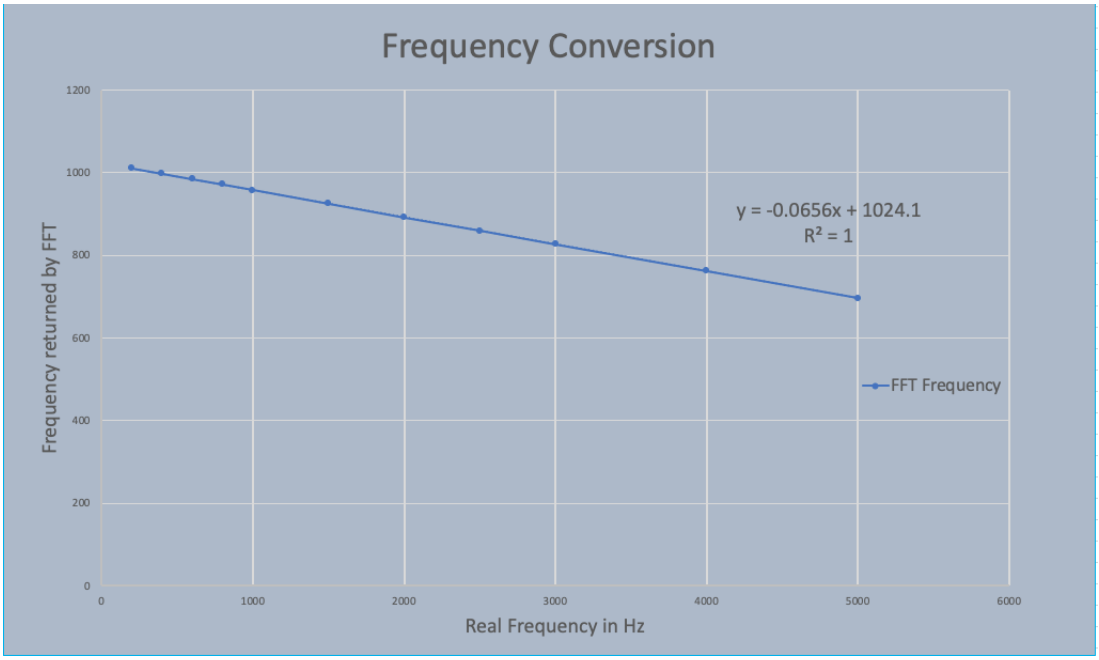


Figure 2 – Conversion of the frequency calculated by the FFT-algorithm (y) to a frequency in Hz (x)

This is done by *audio_ConvertFreq* with the reformulated formula of Figure 2: $x = 15611 - 15.244y$. Sometimes an inverse logic has to be applied. As the formula shows if the real frequency increases (x-value) the frequency returned by the FFT decreases (y-value).

2.3.3. Peak determination

audio_AnalyseSpectre is the function that analyses the sound input and determines the peaks of the frequency domain. First this function waits until the microphones delivered enough samples and filled a buffer of 1024 samples. The sampling frequency of the microphones is 16kHz. Every 10ms the thread of the microphones is calling the function *audio_processAudioData* and is writing 160 samples for each microphone into the buffer. Therefore, in average it will take 64ms to fill up the buffer. Because this is a long time for the microcontroller this is also approximately the time needed for the total cycle of detection of the sources and calculation of the angles. Afterwards, the buffers are copied into the *mic_data*-arrays to avoid any overwriting if the calculations take longer than 10ms.

In a second step, the function *audio_CalculateFFT* is called which calculates the Fast-Fourier-Transform (FFT) of the recorded sound and the amplitude of one of the microphones. Now, the actual sound analysis starts with *audio_Peak* (see Figure 3). This function scans the range of frequency used and detects all frequency-peaks. If no loud noise disturbs the signal, these peaks correspond to the sources that have to be detected.

We are running only through a part of the FFT and limit the detection to a range of frequencies. This was a choice due to multiple factors: We realized that if we make a measure with the microphones in complete silence and calculate the FFT, we obtain really large amplitudes between 0 and 150Hz. This is a noise coming from the microphones. As we see in Figure 4 the amplitudes are exceeding the threshold (15000,

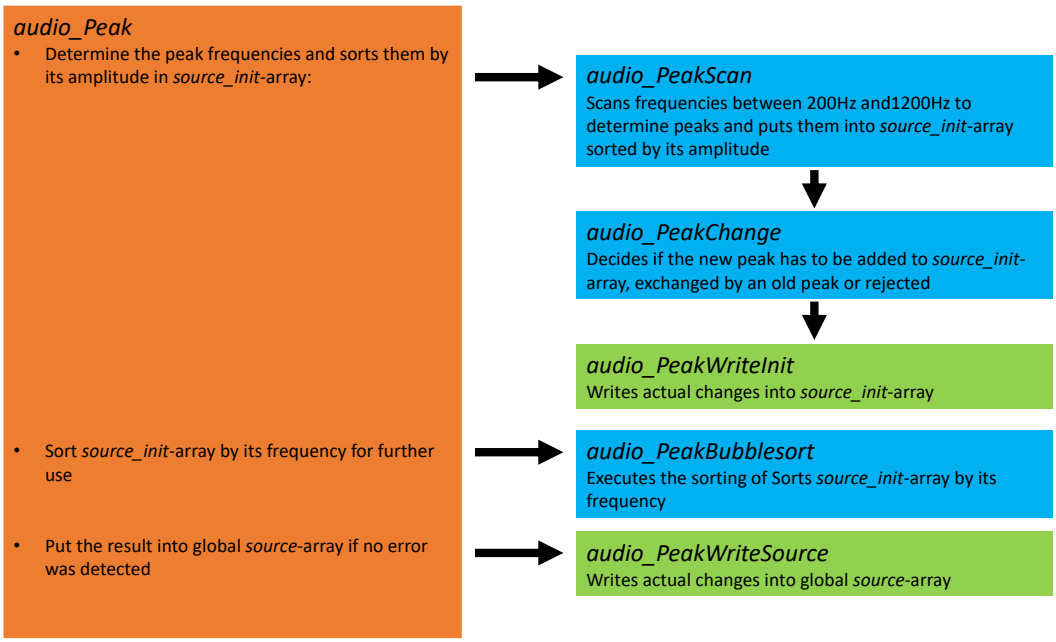


Figure 3 – audio_peak function with called sub-functions

amplitudes are always provided in the unit returned by the FFT algorithm) used to determine the peaks. Therefore, we filter all frequencies below 200Hz. The upper limit of 1200Hz was chosen out of practical

reasons: frequencies over 1200Hz get painful for the ears. This is why we are not using them for the sources and do not consider them in the calculations.
If inside the range of 200-1200Hz a frequency has an amplitude that is higher than a certain threshold (15000) then the value will be written into the *source_init*-array. This is a local and two-dimensional array

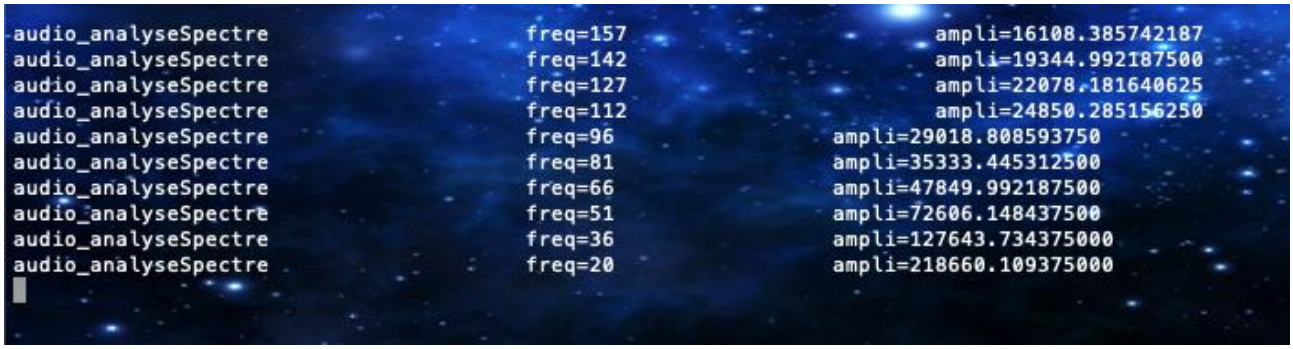


Figure 4 – Measure of the amplitude at low frequencies in completely silence

(left: printing function, middle: frequency in Hz, right: amplitude returned by the FFT)

containing the frequency and the amplitude, that initializes afterwards, if no errors are detected, the global *source*-array. At this point the *source_init*-array is sorted by its amplitudes. If a new frequency is detected either it is added as new element if the maximum number of elements is not yet reached or it is compared to the already existing elements if the array is already full. In both cases the array keeps always the structure having the first element with the lowest amplitude and the last element with the highest. This facilitates the comparison if a new element has to be added.
We wanted the number of sources to be variable, and not have to be predefined in the code (only the maximum number of possible sources has to be defined as constant *NB_SOURCES_MAX*). It is not possible to just take the highest peaks because we do not know how many peaks we should expect. In the beginning, we thought that we could take all frequencies that had an amplitude higher than our threshold, and this would provide us all peaks. But if the robot is close to one source, then multiple frequencies of this source have an amplitude higher than the threshold although we want to only detect the highest peak for this source. Therefore, not only the amplitude but also the frequency has to be compared. If the difference between two frequencies of the *source-init*-array is smaller than a threshold (45Hz) then only the element with the larger amplitude is kept. It was difficult to write an algorithm comparing at the same time the frequency and the amplitude, while also avoid running multiple times through the same loops and having a large number of cycles for the calculation. We minimized the number of cycles by running once through the range of frequencies (945-1011 in the FFT-scale corresponding to 200-1200Hz) and looping only through all existing sources if the amplitude is bigger than the amplitude threshold.
Afterwards, the *source-init*-array has to be sorted by its frequency, because the amplitudes may change if the robot is moving even though the frequency is constant. For later use, we need an array in which the order of the sources does not change in function of the position of the robot. In the end, if no errors are detected we copy the *source_init*-array into the global *source*-array which will be used for the angle calculation.
For the sources we create an array of fixed size defined by the constant *NB_SOURCES_MAX*. If there would be a bigger number of sources used, we should use a dynamic memory allocation to minimize the memory needed. We did not do that because in our project we are using normally under five sources and

therefore the memory used for the corresponding arrays is small. Nevertheless, it would be a way to extend the project especially if one would like to use more sources.

2.3.4. Angle calculation

The function *audio_determineAngle* calculates the angle from where the sound of the target source is coming with respect to the robot. The function returns an angle [-180°, 180°] if the calculation worked out, and an error constant *AUDIOP__ERROR* if an error is reported. It is necessary to call first the function *audio_analyseSpectre* to calculate the FFT and write the peaks into the *source*-array, because *audio_determineAngle* determines the angle with the last calculated FFT and *source*-array. Two parameters are given to *audio_determineAngle*: *source_index* indicates for which source the angle should be calculated and *go_towards_source* defines if the robot is moving towards or away from the source.

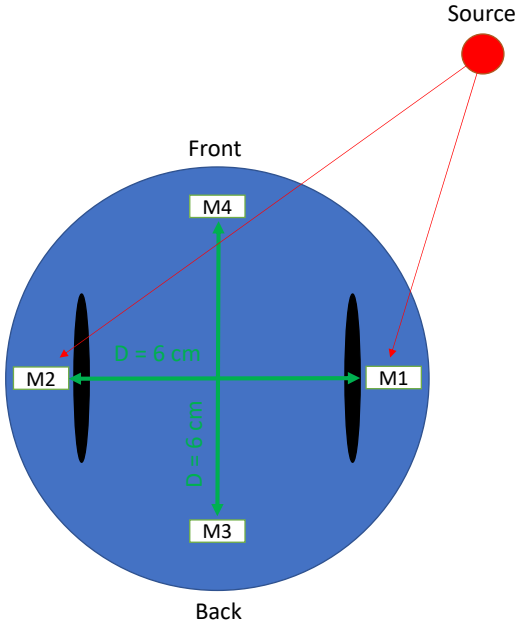


Figure 5 – View from above of EPUCK2 with microphones M1-M4 and sound source (blue: EPUCK2, white: microphones, black: wheels, green: distance measure, red: source)

As we see in Figure 5 the distance between the microphones M1-M2 and M3-M4 is around 6cm. We assume that the microphones are perfectly centered (we neglected the fact that M1 and M2 are slightly shifted towards the front). If a source emits a sound wave from a certain direction, the path that the wave makes is different for each microphone (except if the source is on the y-axes). This leads to a shift in time of the signal. After we calculated the FFT this time-shift is a phase-shift in the frequency domain. The difference in phase of a pair of microphones is evaluated by *audio_determinePhase*. As we see in Figure 6 the phase difference $\Delta\varphi$ for the pair M1-M2 is maximal or minimal if the source is on the x-axes and equal to zero if it is on the y-axes. Because we know the speed of sound in the air (v_{sound}), the distance between the two microphones (D) and we measure the frequency of the sound wave (f) we can calculate the maximal phase shift. Further,

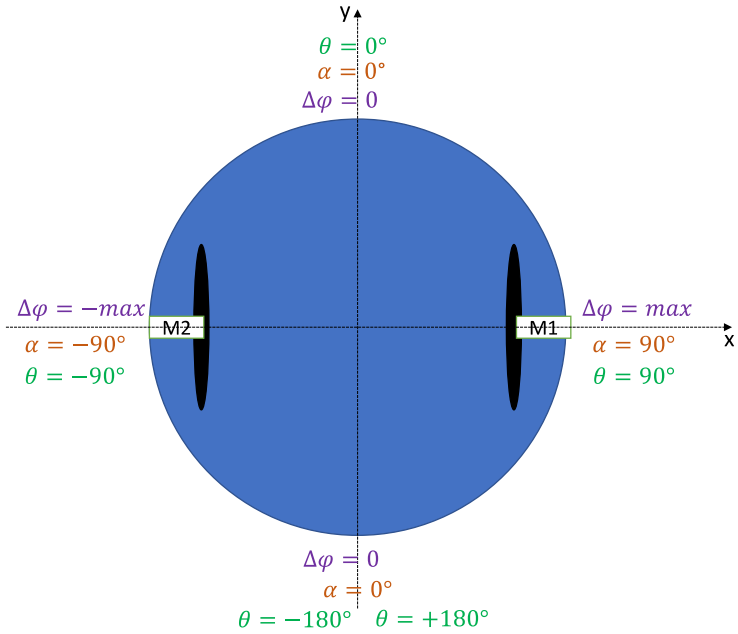


Figure 6 – View from above of EPUCK2 with phase difference and corresponding angle of microphone pair M1-M2 (blue: EPUCK2, white: microphones, black: wheels, violet: difference of phase, orange: angle[0°,±90°], green: angle [-180°,+180°])

we assume that the angle is changing linearly around the robot (this is an approximation but precise enough for our application). Then, we can calculate the angle (α) by normalizing the measured phase shift with the maximal phase shift possible. This is done with Eq. 1 in *audio_convertPhase*.

$$\alpha = \frac{\Delta\varphi * v_{sound}}{4f * D} \qquad \text{Eq. 1}$$

The calculated angle (α) varies from 0° to $\pm$90° and back to 0°. It is impossible to evaluate if the source is in the positive y-plane or in the negative one. Therefore, the second pair of microphones M3-M4 (see Figure 5) is used to calculate a second angle to determine the specific quadrant from which the sound wave is coming. In the end, the two angles are projected on the same axes (because one pair is shifted by 90° with respect to the other), transformed into an angle between +180° and -180° (θ in Figure 6) and averaged. This calculation is executed in *audio_determineAngle*. The result is the angle of the source with respect to the robot (θ). The averaging of the two angles not only helps to determine the y-plane from which the sound wave is coming, but also as we saw during testing it improves a lot the accuracy of the angle.
In the function *audio_determinePhase* we test if the phase difference calculated is physically possible for frequencies below 1200Hz. If this is not the case, then we return the error constant *AUDIOP__ERROR* and we re-do the calculations with new measurements. For example, this is the case if there is a loud noise disturbing the source signal. To improve further the stability of the angle we use an exponential moving average inside *audio_determineAngle* (similar to Figure 12). The exponential moving average is described by Eq. 2.

$$\theta_{ema} = \theta_{ema} * WEIGHT + \theta_{new} * (1 - WEIGHT). \qquad \text{Eq. 2}$$

The result is really satisfying. We can observe that angles below $\pm$120° are really precise. If the absolute value of the angle increases further, the accuracy of the precision decreases. This is because the exponential moving average cannot be calculated if the angle jumps from a positive to a negative value or vis-versa. In this case no moving average is calculated, but the new value is directly taken as result. This is no big issue because the robot turns anyways towards the source and operates around an angle of zero. Only if the robot wants to escape a killer whale it is moving away from the source and the angles are around $\pm$180° which causes sometimes problems. Therefore, we calculate when we are escaping a killer whale the angles in a different way: the angle $\theta = 0°$ is not anymore like normally in the front of the robot (see Figure 6), but in the back of the robot because this side faces the source when moving away from it. With this adjustment the angles are not anymore around $\pm$180° and the moving average can be calculated perfectly fine.

2.4. Movement control process

2.4.1. Overview

The control of the robot with the motors, uses two input parameters. The first is the angle of the source to be reached, which is from -180° on the left, to 180° on the right, integer values. The second is the distance to an obstacle in front. The angle is determined by other functions (see chapter 2.3), and updated through a callback provided by the movement control process initialization function. The distance is simply measured with the time of flight sensor, which returns, in millimeters, how far the path is clear of obstacles in front. The whole controller is started by *travCtrl_init*, which starts a thread that runs every 10 milliseconds.

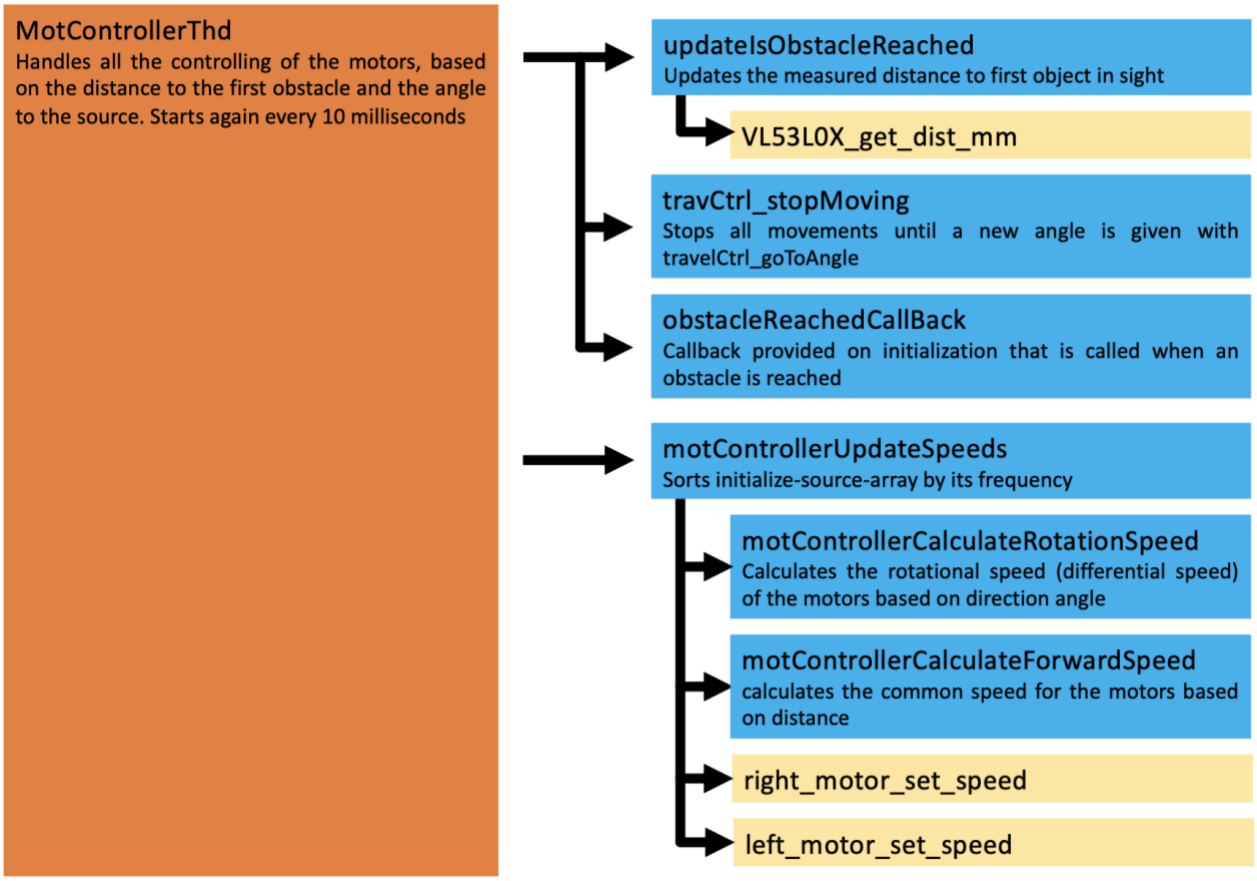


Figure 7 – travelController.c library architecture, explaining function calls and function roles. (orange: main controller thread, blue: subfunctions called from travelController.c , yellow: subfunctions called from ChibiOS)

We can see Figure 7 what functions the thread calls. It will update and check the distance using the time of flight sensor, if it's smaller than the set threshold of 3.5cm which means it has arrived at the obstacle. It also uses the proximity sensors to check for that minimum distance, through a threshold we experimentally set. Because the time of flight sensor is only in one direction, the four proximity sensors used, left, front left, front right and right, prevent hitting obstacles when closing in on them from wider angles. If the robot has not yet arrived at an obstacle, the controller thread will then call the function *motControllerUpdateSpeeds* which updates the motor speeds, based on the angle of the direction and the distance. This motor speeds update function is detailed further specifically. The controller thread is never stopped, but it uses a specific variable to check whether it should do anything or not. When there is nothing to be done, it just reschedules itself to run 10ms later, in order to check again whether it should do something. We decided on a 10ms periodicity as it was fast enough for our application, in fact we could even have lowered it without much noticeable effect, but we wanted to take advantage of the robot capabilities and have smoother control.The travelController.c library exposes four public functions, to initialize, start going to a specific angle, stop, or perform a specific back up maneuver. The library also requires upon initialization to be provided with a function pointer (callback) to call when it reaches an obstacle. All the rest is done internally.

2.4.2. Motors speed update

For this part of the movement control, there is one general concept. We have a forward speed, and a rotational speed. The forward movement, depending on how far the obstacle is, is regulated by a P structure, which sets a common, forward, speed value for both motors. On top of this forward speed, a rotational speed value (differential speed between the two motors) is obtained through a P controller using the direction angle as input. As this differential speed value is calculated positive for positive angles, and the robot needs to turn right in this situation,

PSEUDOCODE :

motControllerUpdateSpeeds

```
calculate rotational speed
if the angle is within ± 40° (in front)
    calculate forward speed
right speed = forward speed - differential speed
left speed = forward speed + differential speed
update exponential moving average of left and
                        right speeds with latest values
set motors to offset calculated exponential
                        moving average speeds
```

Pseudocode 1 - function motControllerUpdateSpeeds

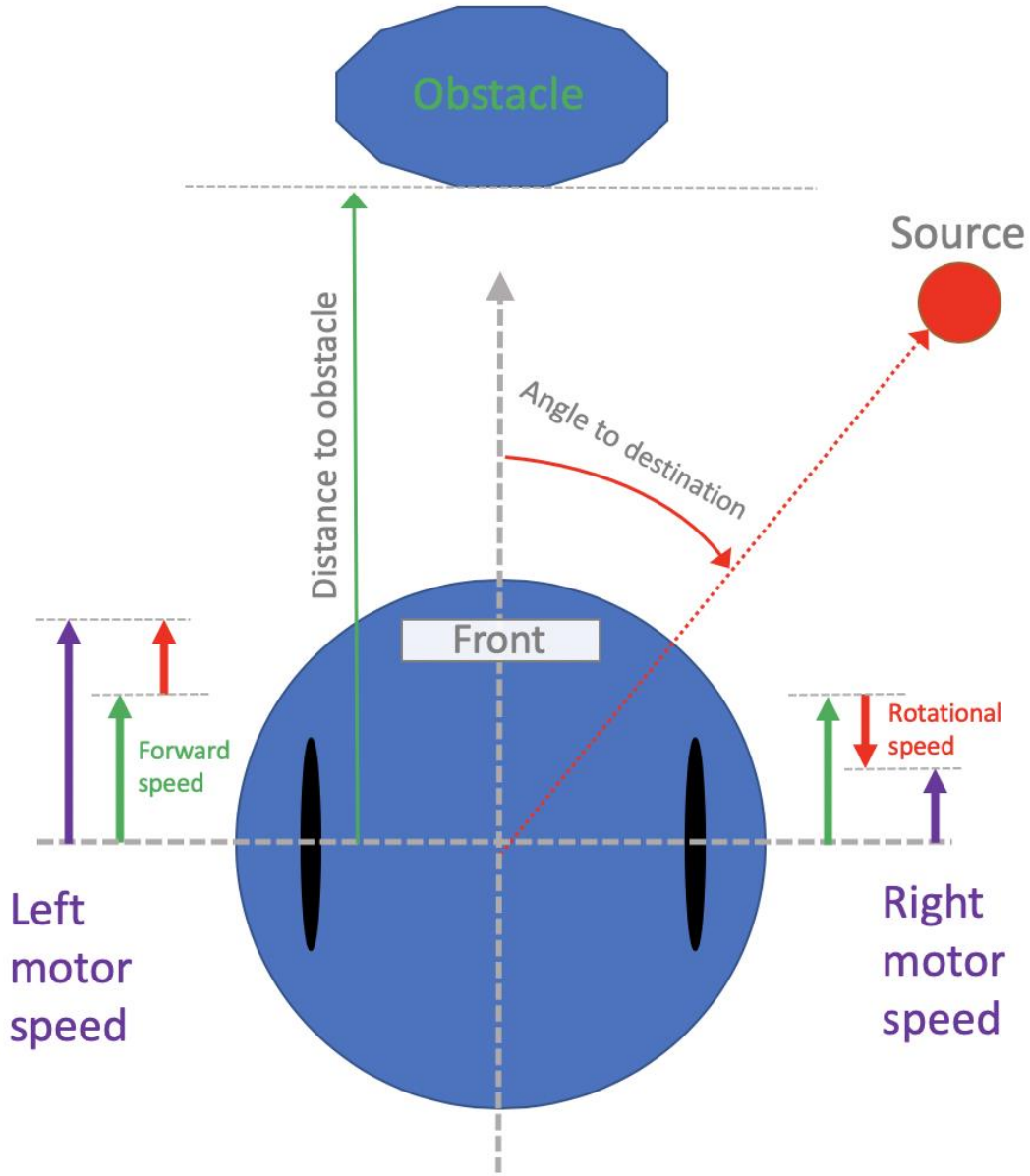


Figure 8 - Forward and differential speed concept break down, depending on angle to source and distance to obstacle.

the right motor speed is defined and set as the common speed minus the differential speed. And the left motor speed is the common speed plus the differential speed. In this way, our robot when needing to turn right, will slow down the right motor, and speed up the left motor. For the left side, the behavior is similar. This is outlined through Pseudocode 1 as the logic for calculations is detailed step by step. We also represent this graphically in Figure 8 : we show the rotational speed in red, which relates to the angle of destination, and the forward speed in green which relates to the distance from the first obstacle in front. Also you see how these forward and rotational speeds subtract for the right motor, and add up for the left motor.

2.4.3. Rotation speed details

The rotational speed is calculated with the direction angle, using a proportional controller. We considered using other types of control, but we found that the precision was good enough with only a proportional controller. Also, since the angles are not very precise, in absolute or relative terms, they introduce an error much bigger than the types of errors we could reduce through a PI structure (amongst other possible structures). Indeed, the final error with a P controller, as we observed in our experimentation, was not noticeable. We also did not add a derivation because the angles were too uncertain and varied too quickly.

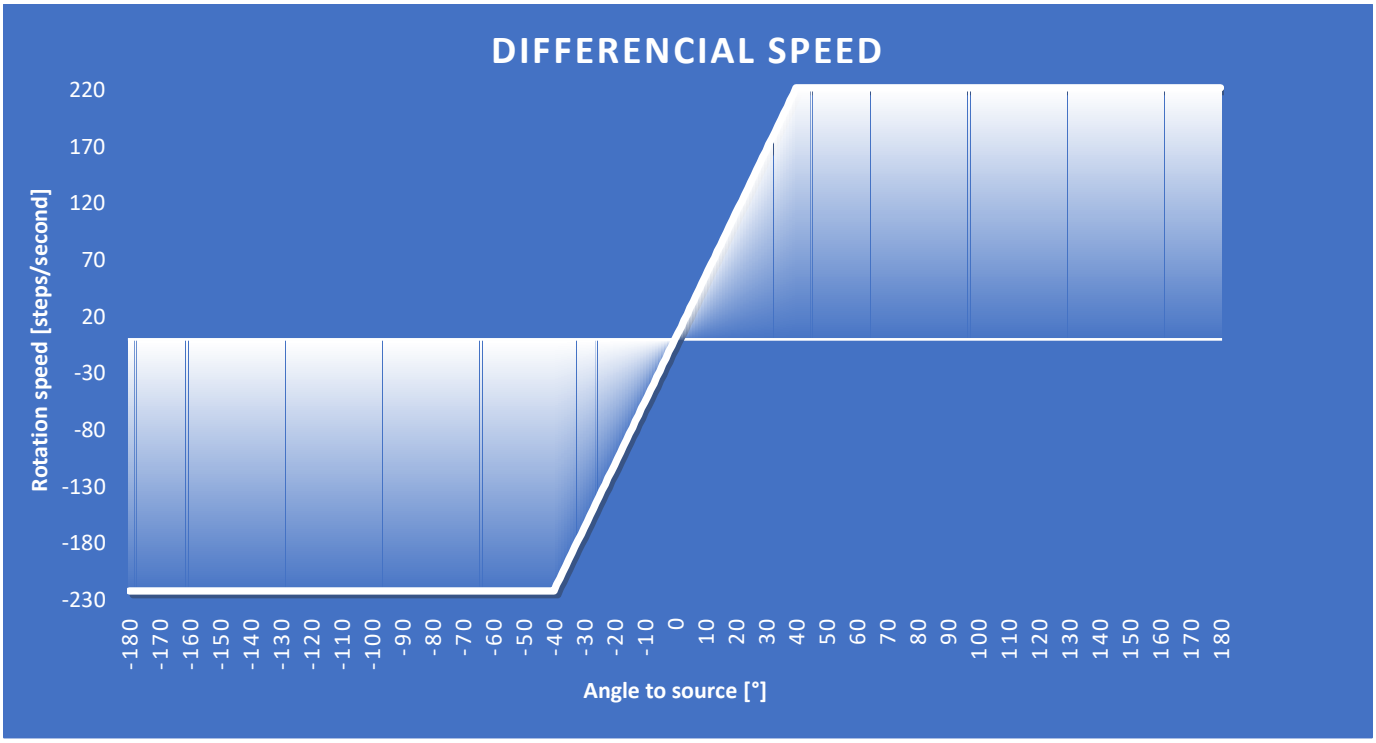


Figure 9 - Differential speed control curve, calculated values in steps per second for angles to the source from -180° to 180°

We control the rotation of the robot throughout the whole movement, with a rotation speed linearly proportional to the angle of direction to go to. In order to better set up the response of our robot, we set the response to be linear only up to 40° (*MOT_MAX_ANGLE_TO_CORRECT*), after which the rotation speed is constant at 222 steps per second (*MOT_MAX_DIFF_SPS_FOR_CORRECTION*). This enabled us to have a bigger response for small angles, but keep a reasonable speed for the maximum angles the robot could see. We settled on 222 steps per second and 40° after trying different angles and seeing what made the robot behave best. For the type of control, we also considered using an easing (centered sigmoid) function, to have a steeper speed curve for small angles, and avoid having a specific limit to the linearity of the correction. But this proved much too sophisticated for our application, which worked well with linear

control. Therefore, we use two parameters to control the slope of the speed to the angle: the max angle for linear correction, and the max rotation speed desired. As we offset the speeds, which is explained later, we have in effect also a third parameter to consider, the minimum motor speed to be applied, but this one is not intended to be configurable so we do not include its effect here. You can see the resulting control curve we apply in Figure 9.

2.4.4. Forward speed details

The forward speed, which means the part of the speed applied to the motors that is the same for both, is controlled by the distance, in millimeter, obtained through the time of flight sensor. We also set this forward speed to zero when the angle is not within ± 40°. We took such a limit to the forward speed control, because we wanted the robot to really only go towards the source,. For angles over 45° (*MOT_MAX_ANGLE_TO_CORRECT* which we reused from the rotational control because it made for a good transition for both) we are in fact using only about half the energy to move towards the source, and the other half to move in another, unneeded direction. Therefore we settled on 40° because it seemed to be the most efficient from our tests, where we saw that the robot would be fast and not get stuck moving too much to the sides. As it was the same value as for a rotational parameter, we reused the same constant.

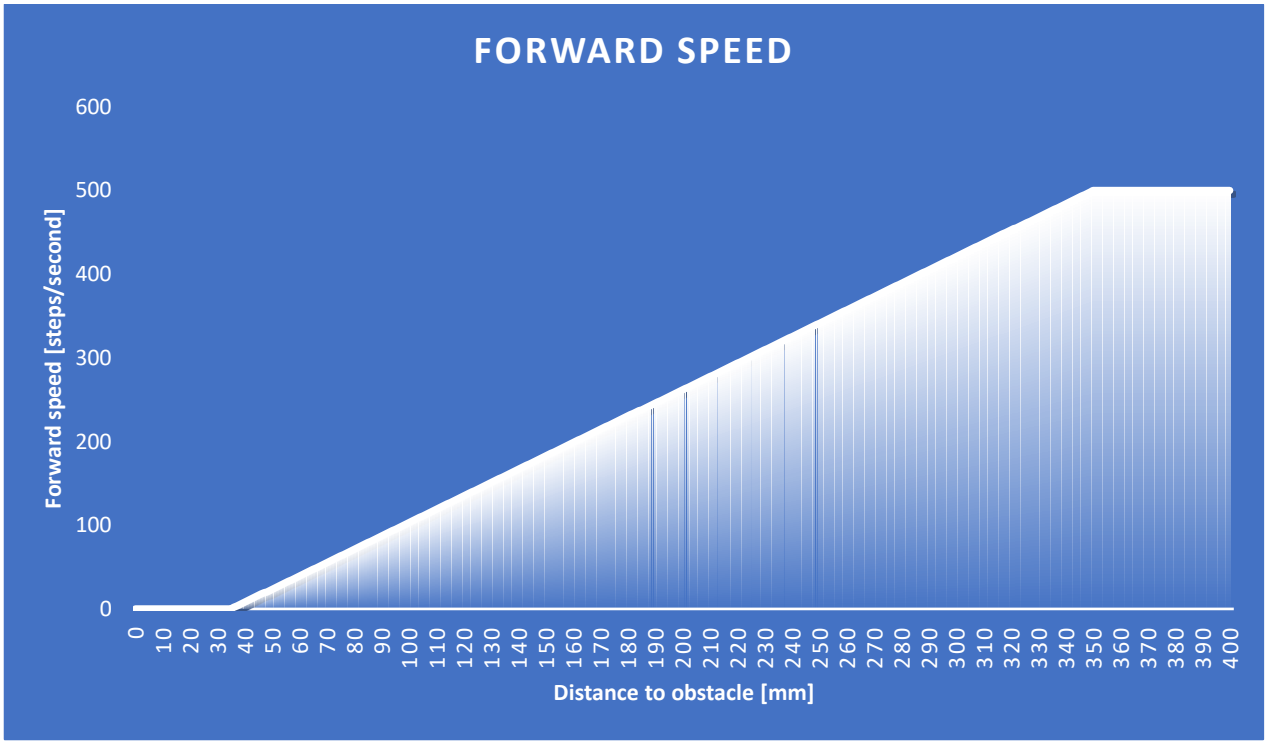


Figure 10 - Forward speed control curve, calculated values in steps per second distances in millimeters from 0 to 396, to be prolonged for higher distances as a constant

The actual control of the forward speed is accomplished through a proportional control similar to the rotational speed. We use a maximum forward speed of 500 steps per second (*MOT_MAX_NEEDED_SPS*), a maximum distance for the linear control of 350 millimeters (*MAX_DISTANCE_VALUE_MM*), and a minimum distance at which the robot will stop of 35 millimeters (*STOP_DISTANCE_VALUE_MM*). Using these three parameters which we set experimentally to have a good behavior, we are able to configure the reactivity of our robot. As we offset the speeds, which is explained later in the section 2.4.5 Additional details, we have in effect also a fourth parameter to consider, the minimum motor speed to be applied. But

this parameter is not intended to be configurable so we do not include its effect here. As for the rotational speed control, we considered other types of control structures, such as PI or using specific curves, but we concluded after testing that the extra layer of complexity to the code and the computing time was not necessary for the types of movements we needed to accomplish. Indeed, we could not see noticeable differences

In short, when being between ± 40° where the linear control is in place, we calculate the forward speed to distance curve, through our three parameters, max forward speed, max distance for linear correction and minimum distance at which to stop, seen Figure 10.

On top of this control, we added an infrared proximity sensors condition to stop the robot when one of the four forward facing sensors detects an object too close. Indeed, sometimes the time of flight sensor could not detect the objects that were on the sides, but still close enough to be hit, so we decided to improve reliability.

2.4.5. Additional details

For the time of flight sensor, we used exponential moving average values with a weight of 0.8 (EMA_WEIGHT_TOF), because otherwise we observed values that were not smooth enough for the direct usage in the forward speed controller. Such an average is calculated with Eq. 3.

$$newAveragedValue = wheight * oldAveragedValue + (1 - weight) * newMeasurement \quad \text{Eq. 3}$$

This type of average is enabled through a simple static variable within the function, which we update on each new measurement to add the new measurement and reduce the old measurements in importance.

You can see in Figure 11 how the weights of the past measurements decrease exponentially over time, and on Figure 12 which part of the averaged value is determined by values how far back, as we did an integral of the weights to time value. We notice that after 10 measurements, the average value becomes 90% renewed, so we can say our reaction time to a sudden change will take this many updates to really reflect the new value, so with a 10 millisecond controller refresh time, this means a 100 millisecond reaction time to sudden changes. This is sufficient for our application as the robot does not move quickly, there is no high reaction time needed. Also, because it moves relatively slowly, the robot will not see sudden changes of distance to obstacle, so the averaged value will already be adding in small changes gradually.

On top of calculating exponential moving averages for the time of flight measurements, we decided after experimentations to set motor speeds using a similar exponential moving average strategy. Before implementing this, we were often setting sudden changes in speed to the motors, which caused them to make jitter noises that were none too reassuring. Therefore, we considered different ways of smoothing changes, while preserving good reaction times. A linear smoothing approach was not reactive enough, and an exponential moving average was both highly reactive, and sufficiently smoothing to the motors. We set an exponential weight of 0.9 (*EMA_WEIGHT*) through experimentation, which, using the same types of analysis as Figure 12 gives us a 90% response time of 20 refreshes so 200 millisecond. In the worst case scenario, even added, for the forward speed component, to the time of flight averages response times, this gives an approximate 300 millisecond reaction time small enough relative to the robot movement speeds.

Another detail is that we offset speeds that we set to the motors. This is because we found in our experiments, that the motors would not work well, or not work at all, for speeds under 100-200 steps per second. They seemed sometimes to be rusty, or started vibrating. After asking on the forums about this and being told it

might be due to faulty motors, we decided to fix it through our code. We simply offset the exponentially averaged values for the right and left motor by 150 steps/second (*MOT_MIN_SPEED_SPS*). For each motor, depending if the averaged total speed value is positive or negative, we add or subtract respectively 150 steps/second to it and set this to the motor. Doing this fixes the problem of low speeds while keeping the forward and rotational speed logic intact. We need to keep in mind though, that when setting the max wanted forward speed and max wanted rotational speed, there will be the *MOT_MIN_SPEED_SPS* value added or subtracted, such that the final forward and rotational speeds will be faster. To prevent errors that could have risen from this interdependence, we clearly stated to pay attention to this via comments in our code where we defined our constants.

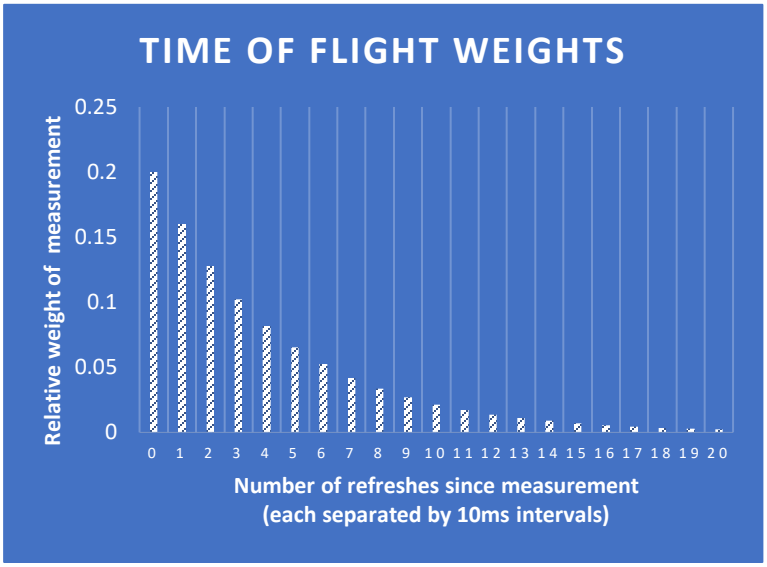


Figure 11 - Time of flight exponential moving average weight curve for past measurements.

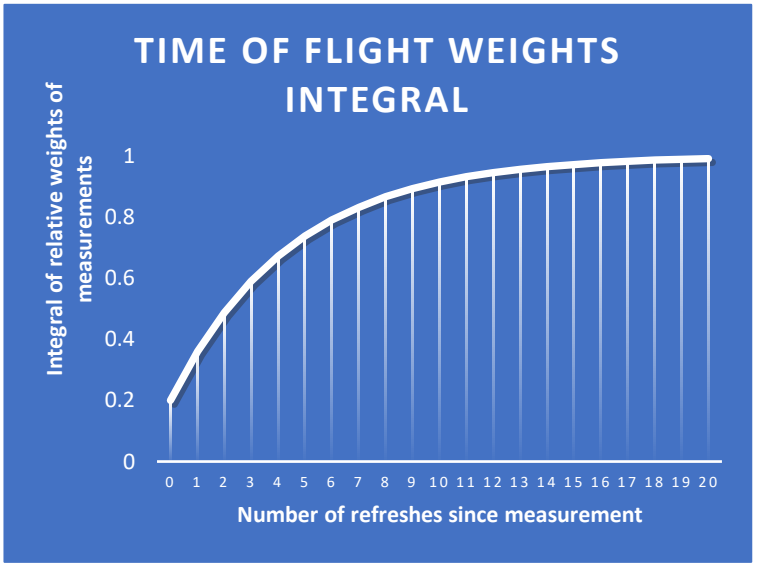


Figure 12 - Time of flight integral of exponential moving average weights, to see how far back values have a significant impact on average.

In the end, our left and right motor speeds can be plotted as a surface in 3D, for all the possible angles and a good distance range, as seen Figure 13.

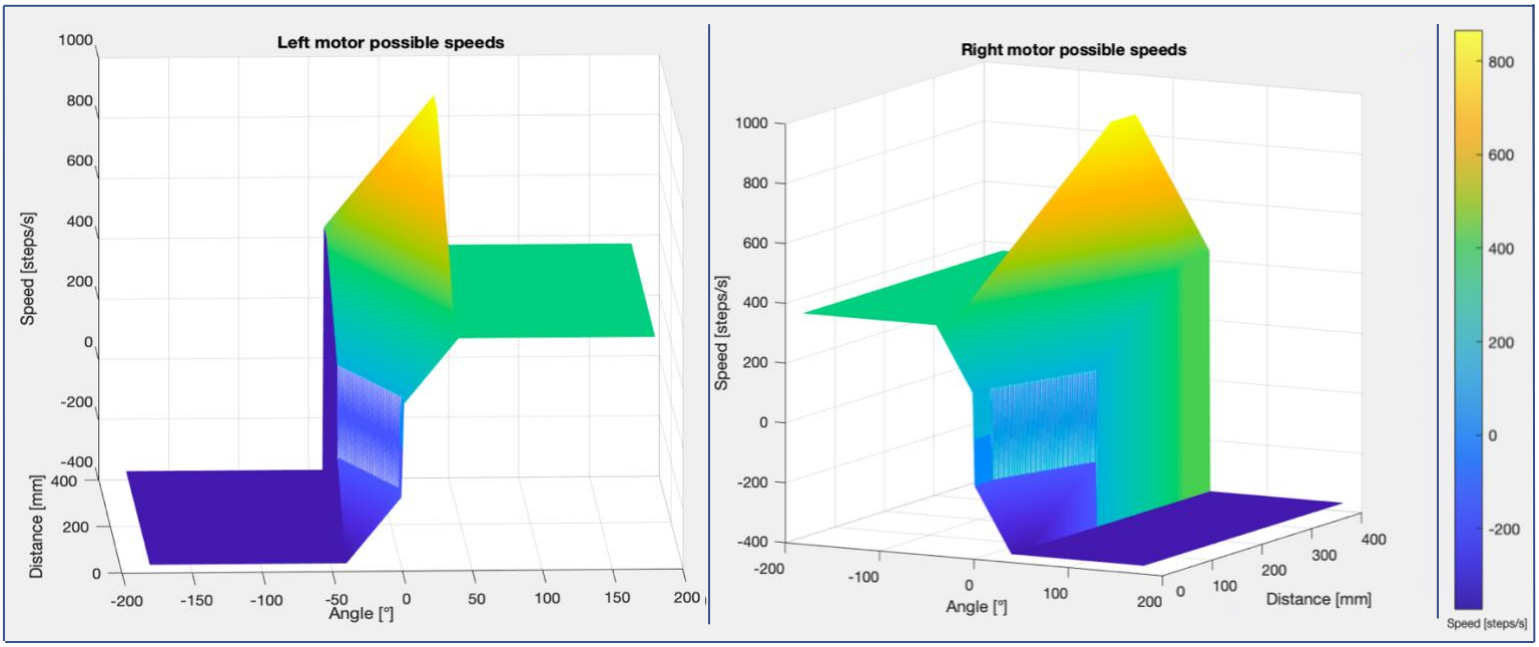


Figure 13 - Left and right motor possible speeds for angle between -180° and 180°, and distances from 0mm to 4000mm.

3. RESULTS

3.1. Robot performance

Throughout the development of our code, we evaluated the response times, the calculations cost, that we incurred on the ePuck. This enabled us to avoid any bottlenecks, and we had enough resources for all our needs. We evaluate for each part of the code what optimizations are made, in each dedicated section of this report.

3.2. Remote team work analysis

As we set out for our penguin project, we thought we would be able to work together and test our code on the robot, whenever we needed. It turned out otherwise due to the COVID19 crisis. As EPFL closed, we happened to have the robot at home, as one of us had taken it for the weekend, fortunately. We had initially set out to split the code in two, one of us doing the audio processing, one of us the motor control and communication. When we started to work remotely, we had to change our approach as part of the code could not be tested while it was being developed. As we were already using git and sharing our code through GitHub, we could both work on the same project, but the one of us without the robot could not test his code bit by bit.

We decided to spend a lot of time working together using desktop sharing and remote control, to develop together instead of separately. We still did a lot of work separately for code that required logic and not ePuck functionality, or to review each other's code and comment or restructure things. But for most of the development process, we had to test together the new code we were developing, as we went along, through remote desktop sharing and control. In the end, this worked out well for us, even if it meant more time spend as a whole because we often had to wait for the other to code something, just watching until the next test to be done. Towards the end of the project, as we were tuning the robot more finely, adjusting control constants, or adding small improvements, which needed a lot of tests to make sure everything kept working as previously and improved how we intended, without unnoticed regressions, we switched to a new way of splitting the work. One of us finished the written report, while the other who had the robot did all the final tests and minor changes. We then reviewed each other's work and after a few iterations this lead to a satisfactory result.

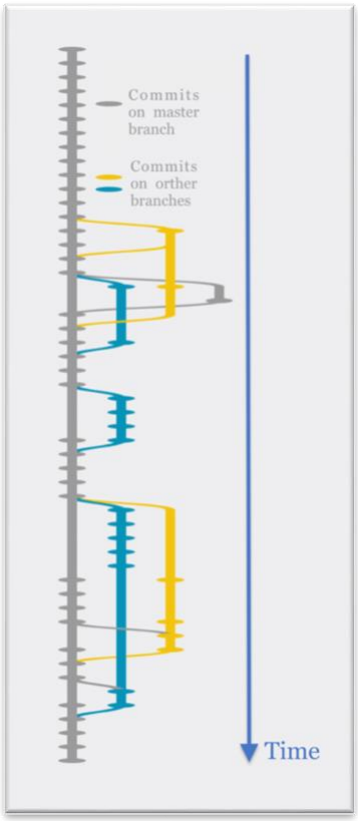


Figure 14 - GitHub repository network graph. Master branch in gray, other branches for specific implementations in other colors. Each node represents a commit, each time two branches join represents a merge.

You can see Figure 14 our GitHub repository network graph, where all the commits on the different branches over time is revealed. The git tool and GitHub hosting platform enabled us to work alongside one another, while being able to go back to see past versions when we encountered problems in new changes.

4. CONCLUSION

During this project development, we have found that penguins were very skilled, but that the ePuck2 robot is also very capable. We were able to utilize some of its many sensors to detect multiple disguised penguin chicks, but also run from any disguised killer whale that presented itself !
We simplified the sound signature of each animal to its most simple form, so the real penguins still win this round. Also, we chose to implement sound detection and analysis in a semi synchronous manner. The sound acquisition was done in a specific thread so in an asynchronous manner, but the analysis was synchronous and made the main function wait for the results. This means, our little penguin robot was not always trying to detect changes in the sound frequencies and angles, or worse, not always checking for the killer whale ! Therefore, in a real environment, our ePuck2 might have been quickly eaten. We decided not to use a specific thread because the rest of the code was not very big, and because we didn't want to always be calculating frequencies and angles when not needed. But after seeing how the robot performed, we know there was a lot of computing headroom. So maybe our robot penguin could have better mimicked real penguins.
For the number of sources that can be detected, we set a fixed maximum, as we knew we wouldn't have more. But as penguin colonies consist of hundreds of individuals, with a much larger number of sound sources, we could have dynamically allocated space for new sound sources that we found. In our western European, there were not many chances of finding more penguins than anticipated… ! But if our ePuck2 needed to perform as well, say, near Antarctica: we would definitively have needed to dynamically allocate memory!

THANK YOU FOR YOUR ATTENTION

We hope this document was pleasant to read. We are available for any more questions and details, via our mails nicolaj.schmid@epfl.ch and theophane.mayaud@epfl.ch . Feel free to ask !

Sources
---

[i] First page image | Composition of ePuck2 personal photo and emperor penguins | Penguins courtesy of Shutterstock, acquired rights through paid account | 25.4.2020
[ii] Image - Emperor penguins and its chicks | Courtesy of Shutterstock, acquired rights through paid account | 25.4.2020
[iii] Tuning Fork | Free android app | https://play.google.com/store/apps/details?id=org.utsoft.android.tuningfork&hl=de | 25.4.2020