Lab + Hwk 3: Introduction to the Webots and Sensor Modeling

This laboratory requires the following software (installed on the DISAL virtual machine (Ubuntu Linux) in GR B0 01):

- Webots simulator
- C development tools (gcc, make, etc.)

The laboratory duration is about 2 hours. The report for this laboratory and homework will be due on the second Wednesday after **your lab session, at 12 noon**. Please format your solution as a **PDF file** with the name **[name]_lab[#].pdf**, where **[name]** is your account user name and **[#]** is the number of the lab+homework assignment, and upload it as the Report submission onto Moodle. If you have additional material (movies, code, and any other relevant files), upload a **[name]_lab[#].tar.gz** archive instead with all files, including the report (hint: `tar cvzf [name]_lab[#].tar.gz [directory or files]`).

Keep in mind that no late solutions will be accepted unless supported by health reasons (and a note from the doctor). If there are extreme circumstances which will prevent you from attending a particular lab session, it may be possible to make arrangements with the TAs ***before*** the lab in question takes place. The more advanced notice you can give in these situations, the more likely it is that we will be able to work something out.

1.1 Office hours

TAs can help you and answer your questions during their office hours:

- Monday, 14:00 - 16:00 (GR A1 445),
- Tuesday, 10:00 - 12:00 (GR A1 424) and
- Friday, 14:00 - 16:00 (GR A1 435)

1.2 Grading

In the following text you will find several exercises and questions.

1. The notation $\mathbf{S_x}$ means that the question can be solved using only additional simulation; **you are not required to write or produce anything for these questions**.
2. The notation $\mathbf{Q_x}$ means that the question can be answered theoretically, without any simulation; **your answers to these questions must be submitted in your report**. The length of answers should be approximately **two sentences** unless otherwise noted.
3. The notation $\mathbf{I_x}$ means that the problem has to be solved by implementing a piece of code and performing a simulation; **the code written for these questions must be submitted in your Additional Material**.

Please use this notation in your answer file!

1.3 The e-puck

The **e-puck** is a miniature mobile robot developed to perform “desktop” experiments for educational purposes.
Figure 1-1 shows a close-up of the **e-puck** robot. More information about the e-puck project is available at http://www.e-puck.org/.

The **e-puck's** most distinguishing characteristic is its small size (7 cm in diameter). Other basic features are: significant processing power (dsPIC 30F6014 from Microchip running at 30 MHz), programmable using the standard gcc compilation tools, energetic autonomy of about 2-3 hours of intensive use (with no additional turrets), an extension bus, a belt of eight light and proximity sensors, a 3-axis accelerometer, three microphones, a speaker, a color camera with a resolution of 640x480 pixels, 8 red LEDs placed around the robot and a bluetooth interface to communicate with a host computer. The wheels are controlled by two miniature stepper motors, and can rotate in both directions.

Figure 1-1: Close-up of an **e-puck** robot.

The simple geometrical shape along with the positioning of the motors allows the **e-puck** to negotiate any kind of obstacle or corner.

Modularity is another characteristic of the **e-puck** robot. Each robot can be extended by adding a variety of modules. A follow-up lab on the real **e-puck** (next week) will allow you to further understand the **e-puck** hardware platform.

1.4 Webots

Webots is a fast prototyping and simulation software developed by Cyberbotics Ltd., a spin-off company from EPFL.

Webots allows the experimenter to design, program and simulate virtual robots which act and sense in a 3D environment. Then, the robot controllers can be transferred to real robots (see Figure 1-2).

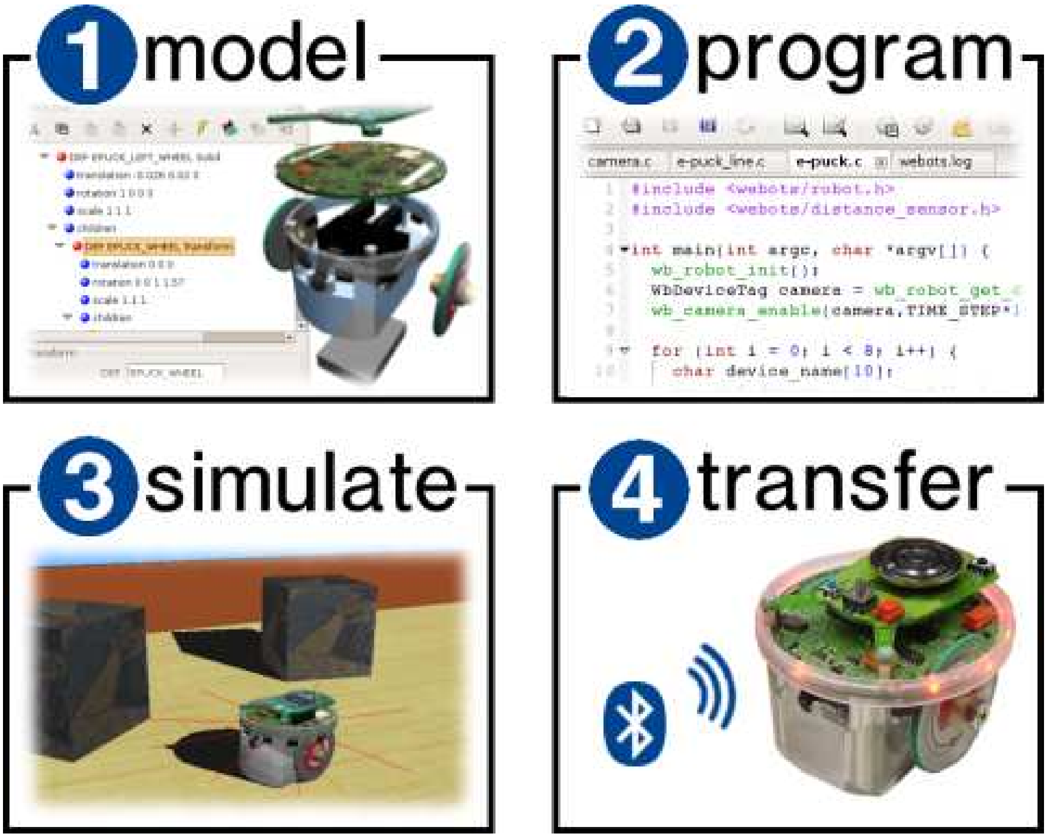


Figure 1-2: Overview of Webots principles.

Webots can either simulate the physics of the world and the robots (nonlinear friction, slipping, mass distribution, etc.) or simply the kinematic laws. The choice of the level of simulation is a trade-off between simulation speed and simulation realism. However, all sensors and actuators are affected by a realistic amount of noise so that the transfer from simulation to the real robot is usually quite smooth.

Many types of robots can be simulated with Webots: and this includes wheeled, legged, flying and swimming robots. Some interesting examples can be found in the *Webots Guided Tour* (menu: *Help->Webots Guided Tour).*

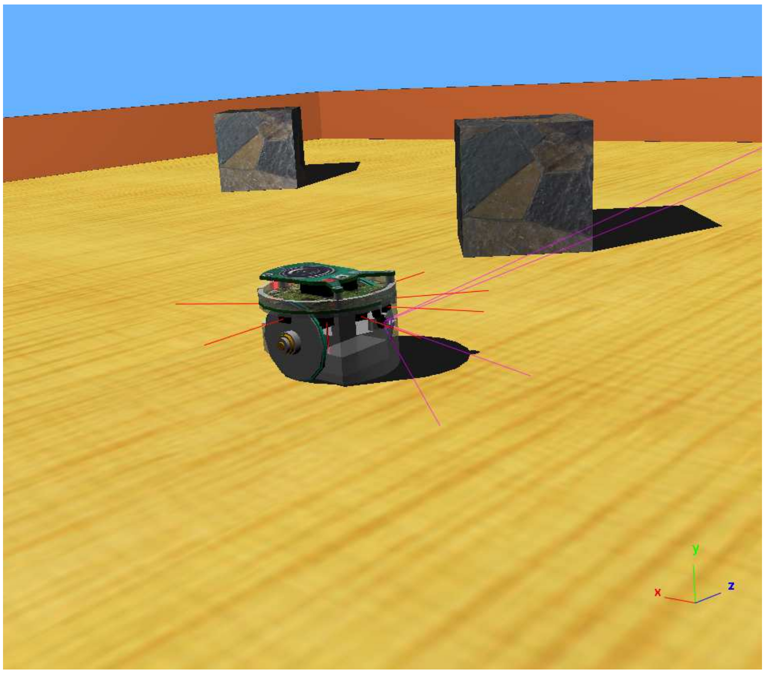


Figure 1-3:. Webots simulation of the e-puck robot.

A simulated model of the **e-puck** robot is provided with Webots (see Figure 1-3). The current version of the e-puck model simulates the differential-drive system with physics (including friction, collision detection, etc.), the distance sensors, the light sensors and the camera. In the future, this model will be further refined to simulate more of the **e-puck**'s functionality. During this laboratory we will perform experiments exclusively with simulated **e-puck** models.

Webots mini-tutorial

Webots is pre-installed in GR B0 01 on the DISAL virtual machine. First, start VirtualBox (Start/Programs/DISAL/Start VM Ubuntu-Hardy DISAL). Once Ubuntu started, a dialog box should appear to order to mount your personal directory under Linux:

Select your "Section" and type in your username and password. Your personal folder should open in a new window. It is important that you do not confuse your linux username (which is user) with your EPFL username and your linux home directory (which is /home/user/) with your EPFL personal folder (which is mounted on Linux in your home directory under *mydocs*: /home/user/mydocs/). **Always use your EPFL personal folder to save your work**.

Then, you can download the lab archive from Moodle using your favorite browser and save it in your personal folder (either from Windows or Ubuntu).

Now, we are going to extract the lab archive, to do so, start a *Terminal*. In the terminal, navigate to your personal folder (remember, you can use the cd, pwd commands). Once you are at the correct location (i.e. in the folder where lab_hwk03.tar.gz is), you can type: **tar xvfz lab_hwk03.tar.gz**. This command will create a new folder, *lab_hwk03*, in which the material for this lab is present. Now navigate in the newly created folder: cd lab_hwk03, and leave the Terminal open at this location. All the files, that you create, need to be saved in the *lab_hwk03* folder.

Now we propose you to go through a mini-tutorial that introduces the Webots user interface.

1.5 Starting Webots

1. Launch Webots from a terminal by entering this command:
   ```
   webots &
   ```
2. If you're opening Webots for the first time, the *Welcome* screen will show up: you can have a look at the *Guided Tour* if you want. Otherwise, tick the message *Don't show this welcome dialog again* and click on *Your project.*
3. From the menu, select *File->Open World*, and choose the *e-puck.wbt* file from the *webots-lab3/worlds* directory structure that was just created.
4. At this point the **e-puck** model should appear in Webots main window.
5. The rays of the proximity sensors can be displayed in Webots. Please switch on this feature from the menu: *Tools->Preferences->Rendering->Display sensor rays*.
6. Now you can hit the *Run* button and the simulation will start.

You can *Stop*, *Run* and *Revert* the simulation with the corresponding buttons in the Webots toolbar. Please try pressing all these buttons to see what they do:

- *Revert*: Reloads the world (*.wbt* file) and restarts the simulation from the beginning
- *Step*: Executes one simulation step
- *Stop*: Stops at the current simulation step
- *Run*: Runs the simulation
- *Fast*: Runs the simulation at the maximal CPU speed (OpenGL rendering is disabled for better performance)

At the bottom of the simulator's main window you will see two numerical indicators (see Figure 4): The left indicator (*0:00:08:768*) shows the simulation elapsed time as *Hours:Minutes:Seconds:Milliseconds*. Note that this is simulated time (rather than the wall clock time) emulating faithfully the potential real time progress that would be expected if the experiment was carried out in reality. It stops when the simulation is stopped.

Figure 4: Elapsed time indicator and speedometer

The right indicator (*4.16x*) is the speedometer which indicates how fast the simulation is currently running with respect to real time (wall clock time). See how the elapsed time and speedometer are affected by the *Run* and *Fast* buttons.

1.6 Manipulating objects

Learn how to navigate in the 3D view using the mouse: try to drag the mouse while pressing all possible mouse button (and the wheel) combinations.

Various objects can be manipulated with the mouse: This allows you to change the initial configuration, or to interact with a running simulation. In order to move an object: select the object, hold the *Shift* key and:

- Drag the mouse while pressing the left mouse button to shift an object in the xz-plane (parallel to the ground)
- Drag the mouse while pressing the right mouse button to rotate an object around its axis
- Drag the mouse up and down, while pressing simultaneously the left and right mouse buttons (or the middle button), to lift and lower the object (alternatively you can also use the mouse wheel)

Now, if you want, you may try all of the above manipulations with the **e-puck** and the obstacles both while the simulation is stopped or running. The mini-tutorial is finished.

Webots lab

This lab consists of several modules. We will start by programming a simple obstacle avoidance behavior for a single robot. We will then explore further features of Webots that can help you to get insights for solving the homework problems.

1.7 Proximity and light sensors

A small control window for the e-puck is built in Webots. This window provides proximity sensor values (in blue, outside the body), light measurements (in grey, inside the body), and motor speeds (in red). Normally, the control window should be visible right away, but you can open it by double-clicking on the e-puck in the world view. Now you can start answering the lab questions:

Q_1 (3): You may have noticed that the sensor measurements are changing all the time? Why? Is it the same on a real e-puck?

Q_2 (5): By moving objects around the e-puck, sketch the response of the proximity sensor as a function of the distance to an obstacle. *Hint:* an e-puck has a diameter of 7 cm.

Q_3 (3): What happens if an obstacle and the robot are interpenetrating? What is the proximity sensor measurement in this case?

2.1 Simple obstacle avoidance

In this exercise you will have to implement your own controller as a replacement for the current controller (named "e-puck"). Your controller and your world will be named "obstacle". We created them for you; just select *File->Open World* from the menu, and choose the *obstacle.wbt* file from the *webots-lab3/worlds* directory.

Note that with Webots, each controller must be placed in its own directory and in addition, the directory name and the source file name must match (e. g. the *xyz-zigzag.c* file must be located in a directory named *xyz-zigzag*). Furthermore, each controller directory must also contain a copy of the standard controller *Makefile*.

The current **e-puck** behavior is controlled by a *controller* program written in C. The program source code can be edited using the editor that is provided by Webots. Normally, the source code of the controller should be already open in the editor. **If it is not the case**, follow the following procedure:

1. Press *Ctrl+E* to open *Webots editor* (if not already opened).
2. Press *Ctrl+T* to open *Webots Scene Tree* (if not already opened).
3. In the Scene Tree: expand the `DEF E_PUCK DifferentialWheels` node.
4. Scroll down and select the `controller "e-puck"` line.
5. Hit the *Edit* button: this opens the controller's code (*obstacle.c*) in Webots's integrated editor.

Remark: Such a controller program, once compiled and linked, cannot be run from a shell like a regular UNIX program. It has to be launched by Webots, because bidirectional pipe communication will be initiated between the two processes.

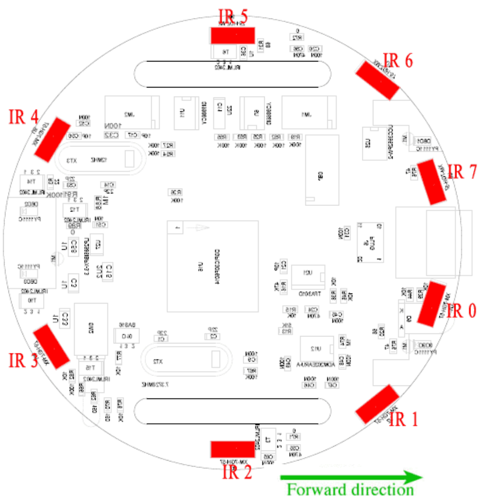


Figure 5: E-puck proximity sensors

Examine the *obstacle.c* code carefully: The distance sensors in the file correspond to those indicated in Figure 5. For example, `ps0` in the code corresponds to *IR0* in Figure 5, `ps1` in the code corresponds to *IR1* in the figure, etc. Note that sensor numbers increment going clockwise; `ps0` and `ps7` are at the front of the robot.

In the code, the function `wb_distance_sensor_enable()` initializes an e-puck sensor. The function `wb_distance_sensor_get_value()` reads the sensor value while `wb_differential_wheels_set_speed()` sends commands to the wheel motors (actuation). The function `wb_robot_step(int ms)` asks Webots to perform one simulation step. The parameter `ms` determines the duration of a step, specified in milliseconds (in this example 64 or 1280 milliseconds). You can find more information on these functions in *Webots Reference Manual* which can be opened using the menu: *Help->Reference Manual*.

Now try to compile the *obstacle.c* controller:

1. Open the *obstacle.c* file in Webots editor, if not already.
2. Hit the *Build* button of the editor to compile *obstacle.c*.
   (At this point *obstacle.c* should compile without problem. If compilation fails ask for assistance.)

Remark: The *save* button, is used to save the **current** state of the world. But when the simulation is run the current world state changes constantly; for that reason it is very important to always *Stop* and *Revert* the simulator just before modifying the world, otherwise an altered world state would be saved.

Now, you can push the *Run* button to execute the simulation with the *"obstacle"* controller.

I_4 (10): As you may have noticed, the original *obstacle.c* controller does not allow the robot to avoid obstacles very well; the robot bumps into obstacles and quickly gets stuck. Modify *obstacle.c* so that your robot becomes capable of avoiding any type of obstacle in any maze. *Hint*: implement an appropriate perception-to-action loop by starting with the 2 central proximity sensors on the front and then exploiting more of the available proximity sensors. Test the performance of your obstacle avoidance algorithm. Try to keep your solution simple and reactive.

2.2 Braitenberg vehicles

Valentino Braitenberg presented in his book "Vehicles: Experiments in Synthetic Psychology" (The MIT Press, 1984) several interesting ideas for developing simple, reactive control architectures and obtaining several different behaviors. These types of architectures are also called "proximal" because they tightly couple sensors to actuators at the lowest possible level. Conversely, "distal" architectures imply the presence of at least one additional layer between sensors and actuators, a layer which could consist for instance of basic behaviors, based on a priori "knowledge" which the programmer wants to give the robot.

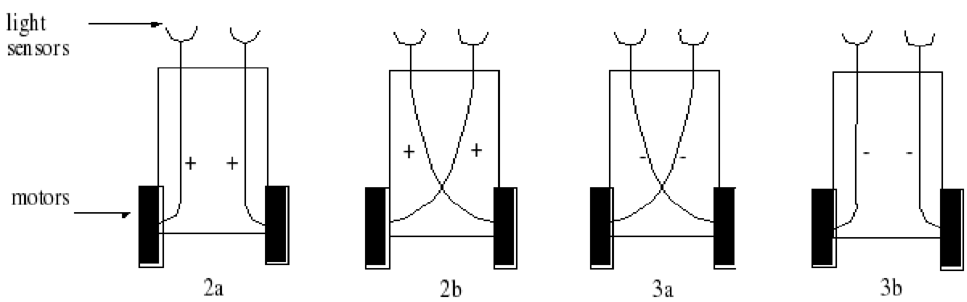


Figure 2- 6: Four Braitenberg vehicles. Plus signs correspond to excitatory connections, minus signs to inhibitory ones. Vehicle 2a avoids light by accelerating away from it. Vehicle 2b exhibits a light approaching and following behavior, accelerating more and more as

approaches the source. Vehicle 3a avoids light by braking and accelerating away toward darker areas. Finally, vehicle 3b approaches light but brakes and stops in front of it.

In this lab, we want to see what we can achieve if robots are programmed as Braitenberg vehicles. *Figure 2- 6* shows four different Braitenberg vehicles.

Q₅ (3): Which vehicle depicted in *Figure 2- 6* do you expect to be most effective at avoiding obstacles if light sensors were to be replaced by proximity sensors, and why?

In the remainder of this laboratory, we are going to implement the controller requested in step using Braitenberg principle on distance sensors (instead of light sensors) and exclusively using a linear perception-to action map (i. e., essentially a 8x2 coefficient matrix).

S_6 : Open the world *braitenberg.wbt* and the controller *braitenberg.c*. Observe the structure of the controller and identify the part of the code to be completed or modified.

The principle of a Braitenberg controller is to directly compute the wheel speeds from the sensor values using a simple linear combination of parameters and sensor values:

$$\text{speed}_{\text{left}} = \sum_{i=0}^{n} \alpha_{\text{left},i} \left(1 - \frac{\text{ps\_value}_i}{\text{ps\_range}}\right)$$

$$\text{speed}_{\text{right}} = \sum_{i=0}^{n} \alpha_{\text{right},i} \left(1 - \frac{\text{ps\_value}_i}{\text{ps\_range}}\right)$$

where ps_va is the value of the i-th proximity sensor, ps_ra is the acceptable range of sensor values, and α and α_{ri} are two parameters.

Q_7 (4): What is the influence of the parameter, say, on the speed of the left wheel? *Hint:* explain what happens if is large or small when an obstacle is detected by the proximity sensor `ps0`.

I_8 (10): Using two nested `for` loops, implement the computation of the wheel speeds from the sensor values using the above formulas.

2.3 The role of noise

It is possible to configure the noise on sensor readings and motor outputs in the Webots simulator in order to model what happens in the real world. Real-world noise can cause poor performance on many algorithms which perform very well theoretically. However, if treated correctly, noise can also be used to positive effect in some systems. In some cases, noise can even become an important ingredient of the algorithm. Hereafter, you will investigate the role of noise for escaping deadlock in proximal approaches to obstacle avoidance.

Q_9(3): Open and run the *no_noise.wbt* world; in this example the noise of all the proximity sensors is set to 0 and the robot must avoid the V-shaped obstacle without remaining stuck. Run this simulation **several times**. Report the number of runs and the success rate of the robot.

Q_{10} (3): What happens? Why?

Q_{11} (3): Now, open and run the *medium_noise.wbt* world, which is exactly the same as *no_noise.wbt*, except for the noise of the proximity sensors of the e-puck. Run this simulation as many times as the previous experiment without noise. Report the number of runs and the success rate of the robot. Discuss briefly the differences with the previous experiment without noise.

Q_{12} (3): What is the noise on the proximity sensors of the e-puck in the *medium_noise.wbt* world?

Q_{13} (3): Now, open and run the *huge_noise.wbt* world. Run this simulation as many times as the previous experiment without noise. Report the number of runs and the success rate of the robot.

Q_{14} (5): What happens now? What does it tell us about noise in robotics experiments?

Hwk: More insight into Webots

2.4 Understanding distance sensor modeling in Webots

In Webots, the behavior of distance sensor is modeled by computing the intersection between a ray cast from the center of the sensor and directed according to the orientation of the sensor (`rotation` field of the `DistanceSensor` node). For each sensor a *lookup table* is used to determine the answer of the sensor according to the distance to the obstacle. This also defines implicitly the range of the sensor. This lookup table includes the return values as well as the noise level for a number of distances entries. A linear interpolation is performed on this lookup table to compute the answer (value and noise) for a given measurement. You will find more information about the modeling of the `DistanceSensor` in Webots Reference Manual (Chapter 3, Section 16). Be sure you understood the principle of the distance sensor before going further:

Q_{15} (5): As you may have noticed, a special case is made for the “infra-red” distance sensor, since their behavior depends on the color of the objects. How can one notice that a distance sensor is an infra-red sensor in Webots? How do object colors influence the response of such infra-red sensors?

Q_{16} (3): Look at the `DistanceSensor` nodes used to simulate the sensors of the **e-puck** robot (found under the `children` node of `DEF E_PUCK DifferentialWheels` in the Scene Tree). How many entries are contained in the lookup table? What is the maximum (operational) range of the sensors?

Q_{17} (5): How are the e-puck proximity sensors implemented from a hardware point of view? *Hint*: if you don't know, check the **e-puck** documentation.

Q_{18} (3): What is the main problem with such an implementation? Does Webots take this factor into account?

Q_{19} (4): Further, you have seen in **Q_3** what happens if an obstacle and the robot are interpenetrating. This may happen due to a simulation artifact, and can be avoided by carefully coding the robot controller. How?

2.5 Braitenberg controllers

Q_{20} (3): What are the difficulties in designing a Braitenberg controller? What are the differences (if any) between one a Braitenberg vehicle and the controller you implemented in **I_4** (*obstacle.c)*?

Q_{21} (3): Look at the weights in the *braitenberg.c* controller source code. You notice that there are asymmetries (the values corresponding to the robot's *left* are not equally weighted to the *right*). Why?

Q_{22} (3): By simply inversing the weights in the *braitenberg.c* controller (corresponding to left and right robot sides), a new robot behavior is created. Without necessarily testing the new controller, what do you assume will be this new behavior?

Q_{23} (5): **Bonus.** A Braitenberg type controller is a reactive controller (proportional to sensor input) - what are the limitations of such a controller? *Hint*: checkout http://en.wikipedia.org/wiki/PID_controller

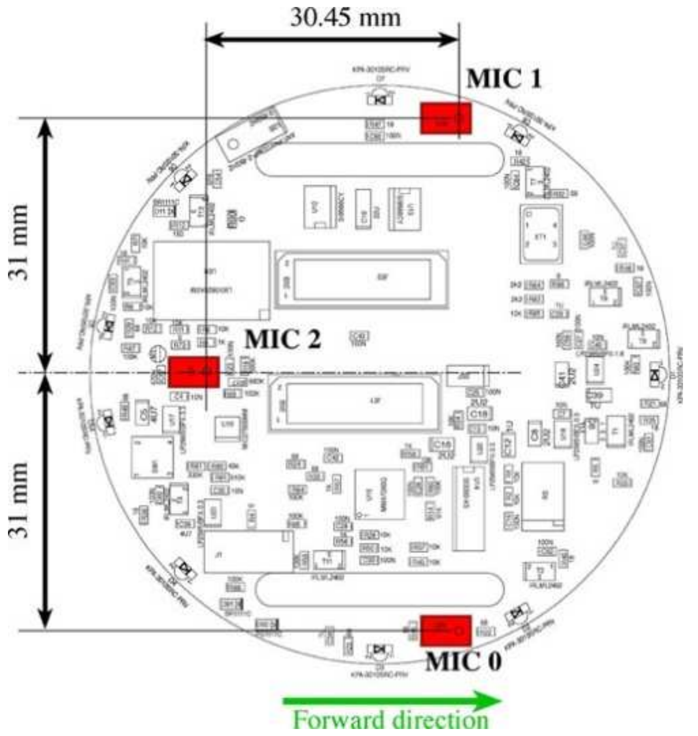
2.6 Microphone

As briefly explained in the lecture, an e-puck robot is able to determine the approximate direction of a sound owing to its three microphones. Hereafter, assume that your robot must monitor a noisy environment with a lot of parasitic sounds of various frequencies and intensities in order to detect and orientate towards a sound source emitting the musical note A (whose pitch is at 440 Hz).

Q_{24} (4): Which signal processing techniques are required to isolate and detect this note? *Hint:* assume that the target signal has a higher intensity than that of a single frequency of the parasitic sounds.

Q_{25} (4): Given the following layout of the robot, determine the phase shift (in rad) between the signals captured by MIC 1 and MIC 2, and between the signals captured by MIC 1 and MIC 0 if the source is located **in front of the robot**. Assume the speed of sound to be 340.29 m/s.

Q_{26} (5): If one of the tasks of the robot is to transmit the data to a base station that is going to achieve a perfect reconstruction of the signal, what is the maximal duration of perception-to-action loop?


30.45 mm
MIC 1
31 mm
MIC 2
31 mm
MIC 0
Forward direction