1 Lab 2: Introduction to the Webots Robotics Simulator

This laboratory requires the following software (installed in GR B0 01 and GR C0 02):

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

The laboratory duration is about three hours. Although this laboratory is not graded, the graded homework following this laboratory will include concepts from the exercises below. We encourage you to take notes during the course of this laboratory to aid in the completion of the homework. A solution sketch will be posted after the lab session.

• Office hours

Additional assistance outside the lab period (office hours) can be requested using the dis-ta@groupes.epfl.ch mailing list.

• Information

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.
2. The notation $\mathbf{Q_x}$ means that the question can be answered theoretically, without any simulation; if you decide to write a report, your answers to these questions should 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.
4. The notation $\mathbf{B_x}$ means that the question is optional and should be answered if you have enough time at your disposal.

To prepare yourself for the homeworks and to allow you for better time planning during the exercise session, we show an indicative number of points for each exercise between parentheses. The combined total number of points for the laboratory or homework exercises is 100.

1.1 The e-puck

The **e-puck** is a miniature mobile robot developed to perform “desktop” experiments for educational purposes.
Figure 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 (although this decreases significantly as the battery ages), 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: Close-up of an **e-puck** robot.

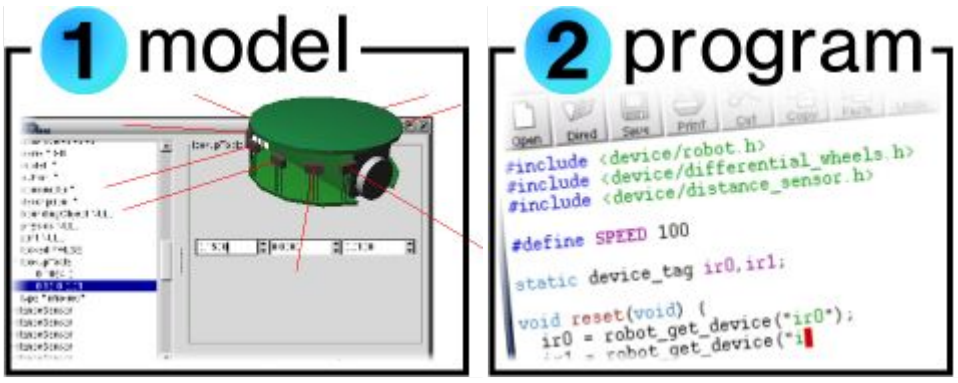
The simple geometrical shape along with the positioning of the motors allows the **e-puck** to negotiate any kind of obstacle or corner.

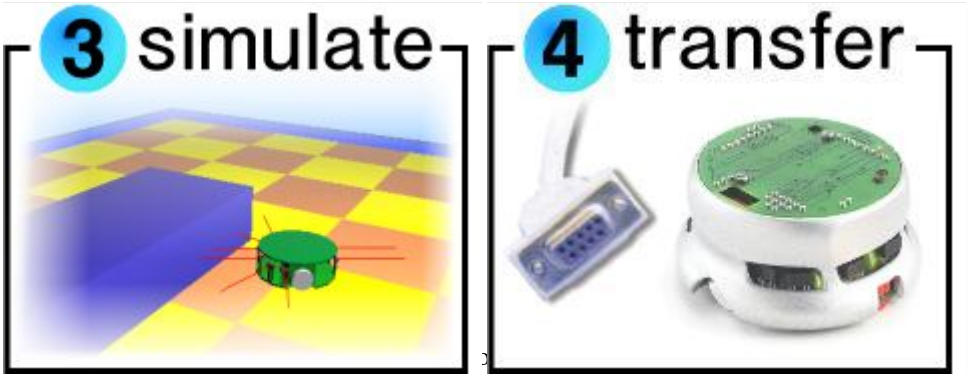
Modularity is another characteristic of the **e-puck** robot. Each robot can be extended by adding a variety of modules. For instance, another series of lab exercises makes use of a dedicated ZigBee radio communication module. A follow-up lab on the real **e-puck** (next week) will allow you to further understand the **e-puck** hardware platform.

1.2 Webots

Webots is a fast prototyping and simulation software developed by Cyberbotics Ltd., a spin-off company of 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 2).





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 can be 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, including wheeled, legged, flying and swimming robots. Some interesting examples can be found in the *Webots Guided Tour* (menu: *Help->Webots Guided Tour)*.

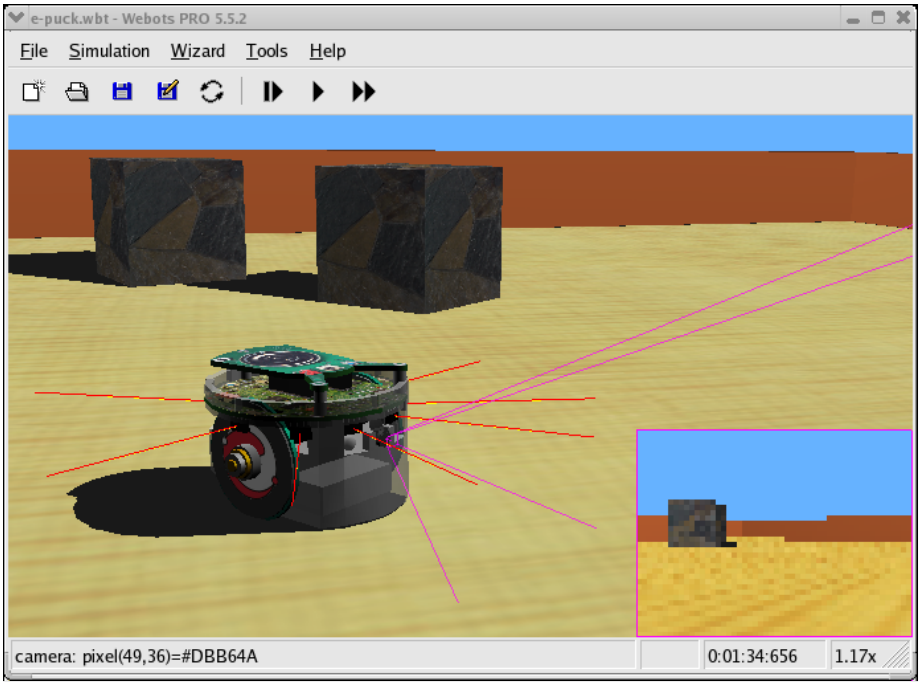


Figure 3:. Webots simulation of the e-puck robot.

A simulated model of the **e-puck** robot is provided with Webots (see Figure 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.

A very useful feature of the Webots simulator, in particular for multi-robot experiments, is the way supervisor modules are implemented. A supervisor is a program similar to a robot

controller; however, such a program is not associated with a given robot but rather with the whole simulation. The supervisor has access to information from robots about their internal (control flags, sensor data, etc.) and external (position, orientation, etc.) variables, and also from the other objects in the world. For example, supervisors can be used for:

- Monitoring of experimental data (e.g., recording the robot trajectories);
- Controlling experimental parameters, (e.g., resetting the robot positions);
- Defining new virtual sensors.

Controller and supervisor modules are programmed in C, C++, or Java. However, for this lab we will use C exclusively.

2 Webots mini-tutorial

At this point,we propose that you go through a mini-tutorial that introduces the Webots user interface. First you need to download the lab package from the course webpage on Moodle, following these steps:

- Download the *webots-lab2.tgz* file from the WEEK 3 section of the course's Moodle site, to the directory of your choice.
- Extract the file with this command:

```
tar xvzf webots-lab2.tgz
```

2.1 Starting Webots

1. Launch Webots from a terminal by entering this command:
   ```
   webots &
   ```
2. If you're opening Webots for the first time, the *About Webots* and *Guided Tour* windows will pop up: you can dismiss the *About Webots* and have a look at the *Guided Tour* if you want.
3. From the menu, select *File->Open World...*, and choose the *e-puck.wbt* file from the *webots-lab2/worlds* directory structure that was just created.
4. At this point the **e-puck** model should appear in Webots main window.
5. Now you can hit the *Run* button and the **e-puck** robot will start moving.
6. The rays of the proximity sensors can be displayed in Webots. Please switch on this feature from the menu: *View->Optional Rendering->Show distance sensors rays*.

You can *Stop*, *Run* and *Revert* the simulation with the corresponding buttons in the Webots toolbar. Please do 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.

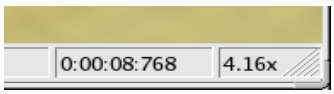


Figure 4: Elapsed time indicator and speedometer

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.

2.2 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.

3 Webots lab

This lab consists of several modules. We will start by programming a simple obstacle avoidance behavior for a single robot.

3.1 Simple obstacle avoidance

The current **e-puck** behavior is controlled by a *controller* program written in C. The program source code can be edited like this:

1. Press *Ctrl-T* to open *Webots Scene Tree* (if not already opened).
2. In the Scene Tree : expand the `DEF E_PUCK DifferentialWheels` node.
3. Scroll down and select the `controller "e-puck"` line.
4. Hit the *Edit* button: this opens the controller's code (*e-puck.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 a bidirectional pipe communication will be initiated between the two processes.

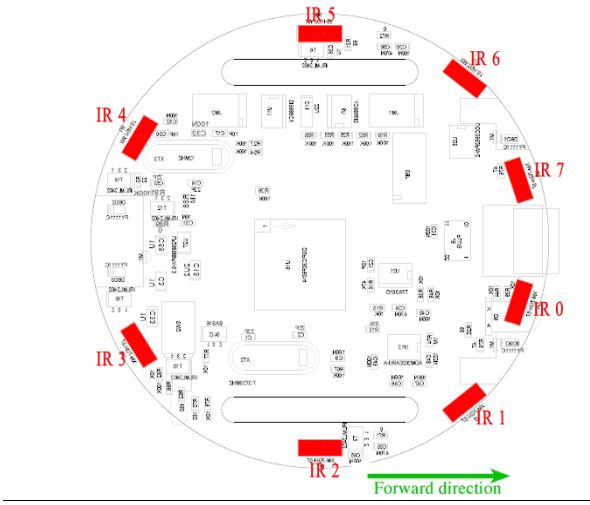


Figure 5: E-puck proximity sensors

Examine the *e-puck.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 `distance_sensor_enable()` initializes an e-puck sensor. The function `distance_sensor_get_value()` reads the sensor value while `differential_wheels_set_speed()` sends commands to the wheel motors (actuation). The user-defined *run()* function is called repeatedly; this function implements one step of simulation. The duration of a step corresponds to the value returned by the *run()* function, 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*.

In this exercise you will have to implement your own controller as a replacement for the current controller (named "e-puck"). Your controller will be named "obstacle", it can be created by copying the "e-puck" controller code. In a terminal proceed like this:

```
$ cd webots-lab2/controllers
$ mkdir obstacle
$ cp e-puck/e-puck.c obstacle/obstacle.c
$ cp e-puck/Makefile obstacle/Makefile
```

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-zizag.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*. From now on, each time we ask you to save your controller we will implicitly assume that you also create the corresponding directory and copy the *Makefile* as explained above.

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

1. Open the *obstacle.c* file in Webots editor

2. Hit the *Clean* button first and then the *Build* button to compile *obstacle.c*.
   (At this point *obstacle.c* should compile without a problem because it is just a copy of the original *e-puck.c*. If compilation fails ask for assistance.)

Now Webots must be told to use the newly created "*obstacle"* controller:

3. If the simulation is running stop it by pushing the *Stop* button
4. Push the *Revert* button
5. If a dialog box asking whether to save the world opens, hit *No*.
6. Open the *Scene Tree* window (if closed) with Ctrl-T.
7. In the scene tree: expand the `DEF E_PUCK DifferentialWheels` node.
8. Scroll down a bit and select the field: `controller "e-puck"`
9. Hit the `[...]` button and choose the `obstacle` controller from the list:
   You should now see: `controller "obstacle"`
10. Now, save your world as *obstacle.wbt,* using: *File->Save World As ...*

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 new *"obstacle"* controller. Of course, at this point the *"obstacle"* controller should behave exactly like the *"e-puck"* controller. Now you can start answering the lab questions:

I_1 (15): As you may have noticed, the original *e-puck.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 all 8 of the available proximity sensors. Test the performance of your obstacle avoidance algorithm. Try to keep your solution simple and reactive. Make sure you save your new controller code in *obstacle.c* and your world in *obstacle.wbt.*

S: Open the *u_obstacle.wbt* world which contains a long, narrow corridor with a dead end. In the Scene Tree as before, change to your *"obstacle"* controller, then save and run the world. When inside the "U", the robot detects both walls simultaneously.

Q_2 (5): What happens when the robot tries to avoid obstacles here? If obstacle avoidance still works, what types of controllers might have trouble here. If it doesn't work, how could the controller be modified to work properly?

I_3 (10): In the same way that you created the *obstacle.c* controller, now create a new controller named *u_obstacle.c* that allows the robot to escape from a U-obstacle (if it could already, you can keep the same code). Be sure you save your new program in *u_obstacle.c*.

S: Test the robustness of your program again by painting all the walls of the U-obstacle maze (*u_obstacle.wbt*) first black, and then white. Hint: you must remove the texture and change the color of 3 objects in the Scene Tree:

- To remove the texture, select: DEF U_{...}_SOLID -> children -> Shape -> Appearance -> ImageTexture, and then hit the *Reset to default* button.

- To change the color: modify DEF U_{...}_SOLID -> children -> Shape -> Appearance-> Material -> diffuseColor

Q_4 (5): What happens? How does the color affect the obstacle avoidance behavior of the robot?

S: Test the effectiveness of your *u_obstacle.c* controller in the presence of other robots. Open the *multi_obstacle.wbt* world which contains five **e-pucks,** change the five controllers to "u_obstacle", then save and run the world.

Q_5 (5): What happens? What are the additional difficulties involved in avoiding other robots? (list at least two)

Q_6 (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 at http://www.e-puck.org/.)

3.2 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.

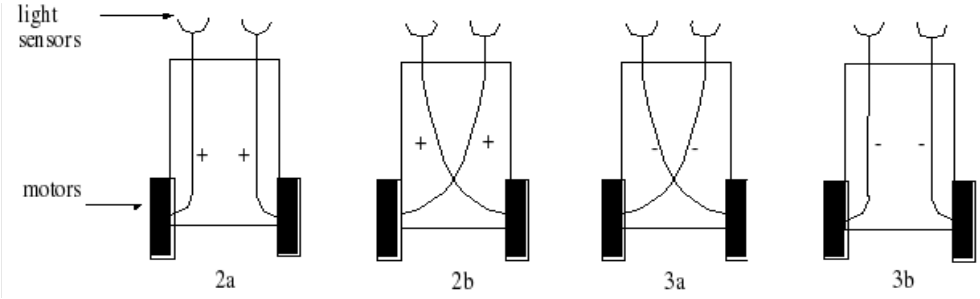


Figure 6: Four Braitenberg vehicles. Plus signs correspond to excitatory connections, minus signs to inhibitory ones. Vehicle 2a avoids light by accelerating away from it. Vehicle 2b exhibits a light approaching and following behavior, accelerating more and more as approaches the source. Vehicle 3a avoids 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 6 shows four different Braitenberg vehicles.

Q_7 (5): Which vehicle do you expect to be most effective at avoiding obstacles, and why?

I_8 (15): Implement the controller requested in step I_1 using Braitenberg principles on proximity sensors (instead of light sensors) and exclusively using a linear perception-to action map (i. e., essentially a 8x2 coefficient matrix). Create a new controller and save it as *braiten.c*. Test your controller in the original world file *e-puck.wbt*.

Q_9 (10): What are the difficulties of designing such a controller? What are the differences (if any) between one a Braitenberg vehicle and the controller you implemented in I_1 (*obstacle.c)* and in I_3 (*u_obstacle.c*)?

3.3 Distal Architectures

As said before, "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.

In the example described in the world file *epuck_and_beacon.wbt*, the e-puck is located in a bounded arena, populated with several obstacles and one light beacon that moves randomly in the arena. The robot will have to be programmed for two different tasks: to avoid the light beacon (*photophobia*) or to follow the light beacon (*phototaxis*). During both these tasks the robot will still need to avoid obstacles and the walls of the arena. To combine these basic behaviors you will need to use a distal architecture.

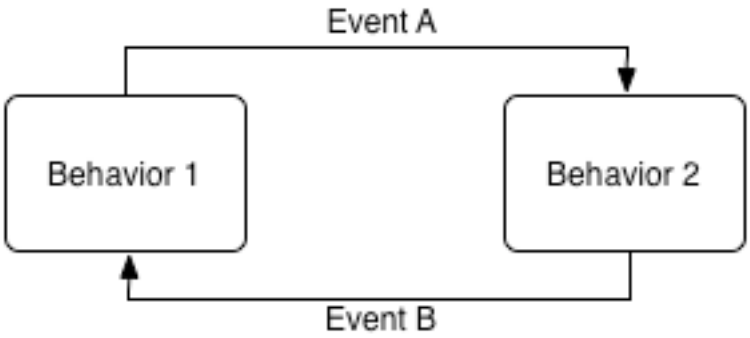


Fig. 7: Two State Finite State Machine Blueprint.

I_{10} (10): Implement behaviors *photophobia* and *phototaxis* using the Braitenberg architecture seen in the previous section. You will need to use the light sensors of the e-puck instead of the proximity sensors (check the Webots reference manual for the appropriate functions, light sensors already initialized in *reset()* function). Use the world file *epuck_and_beacon_no_obstacles.wbt*.

Q_{11} (5): Using the blueprint of a two-state Finite State Machine (FSM), design two different controllers (distal architecture) which accomplish the tasks defined above and avoid obstacles at the same time (one FSM for photophobia and obstacle avoidance, and one FSM for phototaxis and obstacle avoidance).

I_{12} (10): Implement the FSMs designed in the previous question. Use a state variable to define in which state the robot is at each time step. A skeleton for how a state might be implement is present in *epuck_controller.c*. Use the world file *epuck_and_beacon.wbt*.