Signals, Instruments, and Systems – W8

High-Fidelity Simulation of Real-Time Embedded Systems
Outline

- A few words about simulation!
- Webots
  - Generalities
  - Sensors/actuators
  - Webots API
- Programming embedded systems
  - Buffers
  - Timing
  - An example using the camera in Webots
- Quick note about communication
Simulation: why?

- Hardware prototyping is time-consuming and expensive
- Real hardware platforms (e.g., robots) might be expensive
- Quickly change the experimental setup
- Often easier for monitoring experiments (and evaluating specific metrics)
- Sometimes faster than real-time
  - Useful for using numerical optimization schemes (genetic algorithms, particle swarm optimizations, etc.)
  - Enable systematic search of the parameter space
Webots robotic simulator

1. **model**
2. **program**
3. **simulate**
4. **transfer**

In this course, we will focus on steps 2, 3 and 4 only.
Webots features

- Realistic (physics-based) robotics simulator
- Fast prototyping
- Experiment without real hardware
- Can *usually* run faster than real-time
- Available sensors: distance sensors, light sensors, cameras, accelerometers, touch sensors, position sensors, GPSs, receivers, force sensors, etc.
- Available actuators: linear and rotational motors, grippers, LEDs, emitters, etc.
How does it look like (1)?

Nao humanoid robot standing up
How does it look like (2)?

iRobot Create cleaning an appartement
How does it look like (3)?

e-puck mobile robot performing obstacle avoidance and line following
Webots GUI

- Scene tree
- World view
- Editor
- Simulation speed (w.r.t. real-time)
- Console
Modeled sensors

- distance sensor
- light sensor
- camera
- accelerometer
- touch/pressure sensor
- GPS

...and battery sensor, compass, gyroscope, torque sensor, etc.
Modeled actuators

- Motor (rotational / linear)
- Gripper
- Connector (docking systems)
- LED
- Pen

... and propeller, emitter & receiver, etc.
Webots principles

The more robots, the slower the simulation!
(Newtonian) physics-based simulation (ODE)

Rigid bodies simulation:

Each body is defined by its:
- Shape
- Mass (or density)
- Center of mass and Inertia Matrix
- Velocity (angular and linear)

http://www.ode.org/
(Newtonian) physics-based simulation (ODE)

Joints between bodies:

Various type of constraints:
- Ball (3 rotations)
- Hinge (1 rotation)
- Slider (1 translation)
- Etc.

http://www.ode.org/
(Newtonian) physics-based simulation (ODE)

Collision detection:

Configurable friction:

http://www.ode.org/
(Newtonian) physics-based simulation (ODE)

Some basic fluid dynamic simulation allowing to simulate flying robot and floating/underwater robot:

But thermal dissipation, chemical diffusion or any other advanced physical phenomena are NOT captured!

Lily robots in a water pool

http://www.ode.org/
Supervisor

- A **supervisor** is a program that controls the world and its robots.

- Similarly as a robot, supervisor is a node driven by a controller with **extended capabilities** that supervises the whole world.

- The supervisor can read/write any field of any node in the scene tree in order to:
  - move or rotate any object in the scene
  - simulate changing environmental conditions
  - track the position of a robot

- The supervisor can also take a snapshot of the scene or create movies.
The e-puck robot

Main features:

- Cylindrical, Ø 70mm
- dsPIC processor
- Two stepper motors
- Ring of LEDs
- Many sensors:
  - Camera
  - 3 microphones
  - 1 loudspeaker
  - IR proximity
  - 3D accelerometer
- Li-ion accumulator
- Bluetooth wireless communication
- Open source hardware (and software)
Real and simulated e-puck

Real e-puck

Simulated e-puck (Webots)
- sensor- and actuator-based
- noise, nonlinearities of sensors and actuators reproduced
- kinematic (e.g., speed, position) and dynamic (e.g., mass, forces, friction)
Modeling sensors

- Capture **non-linearities** and **noise** of sensors.
- However, **calibration** is often approximative.
- Most often, sensor response is defined by a lookup table (here a proximity sensor):

<table>
<thead>
<tr>
<th>distance</th>
<th>value</th>
<th>noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.1</td>
<td>1000</td>
<td>0.1</td>
</tr>
<tr>
<td>0.2</td>
<td>1000</td>
<td>0.1</td>
</tr>
<tr>
<td>0.3</td>
<td>400</td>
<td>0.1</td>
</tr>
<tr>
<td>0.37</td>
<td>50</td>
<td>0.1</td>
</tr>
</tbody>
</table>

```
lookupTable = [0, 0.1, 0.2, 0.3, 0.37]
```
A typical Webots controller

```c
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/distance_sensor.h>

#define TIME_STEP 32

int main(int argc, const char *argv[]) {
    double speed[2] = {200.0, 200.0};
    double sensor_value;
    WbDeviceTag sensor;

    // initialize Webots
    wb_robot_init();

    // find distance sensors
    sensor = wb_robot_get_device("ps0");
    wb_distance_sensor_enable(sensor, TIME_STEP);

    // main loop
    while (wb_robot_step(TIME_STEP) != -1) {
        sensor_value = wb_distance_sensor_get_value(sensor);
        if (sensor_value > 500.0) {
            speed[0] = 0.0; speed[1] = 0.0;
        }

        // set the motors speed
        wb_differential_wheels_set_speed(speed[0], speed[1]);
    }

    // cleanup Webots resources
    wb_robot_cleanup();
}
```

Includes for accessing Webots API

Define simulation step in milliseconds

Define data structures

Initialize Webots

Get and enable different devices (sensors and actuators)

Perform one simulation step

Read sensor values

Controller lifetime loop (robot behavior must be coded here)

Update actuators

Cleanup Webots resources
How does it look like?
Webots controllers

- A Webots controller is a **C program** (Webots also supports C++, Java, Python, and even Matlab).
- Therefore, everything you can do in conventional C programs, you can do in a Webots controller.
- You must just keep in mind that your controller **will eventually run on a robot**.
- Webots **does not** simulate the microcontroller of your robot:
  - Your controller will run **much slower** on the real robot than it does in Webots.
  - Your memory will be **much more limited** on the real robot than in Webots.
- In the first case, the behavior of the real robot will be very different from that of the simulated robot. In the second case, you will not be able to compile your controller!
Webots API

- **Definition:** an application programming interface (API) is a set of functions, procedures, methods or classes that an operating system, library or service provides to support requests made by computer programs.
- Webots provides a lot of such functions that allow you to interact with the different devices of your robot:

```c
#include <webots/distance_sensor.h>

void wb_distance_sensor_enable(WbDeviceTag tag, int ms);
void wb_distance_sensor_disable(WbDeviceTag tag);
double wb_distance_sensor_get_value(WbDeviceTag tag);
```

- The principle of the API is that you must always enable a sensor before using them (pretty much like on a real robot)!
- **The Webots API is NOT available on the real robot.** This means that you will need to modify your controller before transferring it to the real robot. In the specific case of the e-puck, cross-compilation is available, though.
Webots API: camera

- If you want to use a camera, you need to include the following header file:

```c
#include <webots/camera.h>
```

- The first thing to do next is to get the device by using the standard function:

```c
WbDeviceTag camera = wb_robot_get_device("cam");
```

- Then, you need to enable it before using it (and possibly disable it if you no longer need it):

```c
void wb_camera_enable(WbDeviceTag tag, int ms);
void wb_camera_disable(WbDeviceTag tag);
```

- You also have a number of auxiliary functions that might come in handy at some point:

```c
int wb_camera_get_width(WbDeviceTag tag);
int wb_camera_get_height(WbDeviceTag tag);
```
Webots API: camera

- Then, you can of course get an image using the following functions:

```c
unsigned char *wb_camera_get_image(WbDeviceTag tag);
unsigned char wb_camera_image_get_red(const unsigned char *image, int width, int x, int y);
unsigned char wb_camera_image_get_green(const unsigned char *image, int width, int x, int y);
unsigned char wb_camera_image_get_blue(const unsigned char *image, int width, int x, int y);
unsigned char wb_camera_image_get_grey(const unsigned char *image, int width, int x, int y);
```

- Here is an example of usage of these functions:

```c
const unsigned char *image = wb_camera_get_image(camera);
for (int x = 0; x < image_width; x++) {
    for (int y = 0; y < image_height; y++) {
        int r = wb_camera_image_get_red(image, image_width, x, y);
        int g = wb_camera_image_get_green(image, image_width, x, y);
        int b = wb_camera_image_get_blue(image, image_width, x, y);
        printf("red=%d, green=%d, blue=%d", r, g, b);
    }
}
```
The concept of buffer

- A **buffer** is a region of memory used to temporarily hold data while it is being moved from one place to another.

- Typically, the data is stored in a buffer as it is retrieved from an input device (such as a sensor) or just before it is sent to an output device (such as an actuator).

- The device writes/reads in the buffer independently of the controller. Therefore, to read the device, you just need to read the buffer, using a **pointer**:

```c
const unsigned char *image = wb_camera_get_image(camera);
for (int x = 0; x < image_width; x++) {
    for (int y = 0; y < image_height; y++) {
        int r = wb_camera_image_get_red(image, image_width, x, y);
        int g = wb_camera_image_get_green(image, image_width, x, y);
        int b = wb_camera_image_get_blue(image, image_width, x, y);
        printf("red=%d, green=%d, blue=%d", r, g, b);
    }
}
```
const unsigned char *image = wb_camera_get_image(camera);
for (int x = 0; x < image_width; x++) {
    for (int y = 0; y < image_height; y++) {
        int r = wb_camera_image_get_red(image, image_width, x, y);
        int g = wb_camera_image_get_green(image, image_width, x, y);
        int b = wb_camera_image_get_blue(image, image_width, x, y);
        printf("red=%d, green=%d, blue=%d", r, g, b);
    }
}

Data can be lost! Only one frame out of three is actually processed here!
Buffer mechanism

write at 0x1
at 30 Hz

Robot memory

0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7

image = 0x1

buffer

read at image
at 10 Hz

time

read and process

100ms

write

33ms

WARNING: TIMING!
Asynchronous vs synchronous

- Each type of robot (DifferentialWheels, Robot or Supervisor) may be **synchronous** or **asynchronous**.
- Webots waits for the requests of **synchronous** robots before performing the next simulation step.
- It does not wait for **asynchronous** robots.
- Hence, an **asynchronous** robot may be **late** (if the controller is computationally expensive, or runs on a remote computer with a slow network connection).
- Obviously, in reality, all robots are **asynchronous** (with respect to real time).
- In practice, we use synchronous robots in simulation because Webots (like most simulation packages) **does not** simulate the microcontroller anyway.
Real time?

- A robot (like any computing entity) has **limited computational resources**. Therefore, any controller has a **computational cost**, which can be expressed as the time required to perform one iteration of the main loop.

  - For instance, the total duration of one single iteration of the controller depicted above is 415\(\mu\)s (~0.5 ms).
  - Therefore, the maximal execution speed of the loop (which is also the update rate of the actuators) is 2 kHz.
Perception-to-Action Loop
(closed-loop system)

Delays are everywhere!

sensors

Perception

analog-digital conversion time

processing time

Computation

actuator delay

Environment

sampling rate

propagation time

actuators
Real time!

- The perception-to-action delay defines the responsiveness of the robot.
- If the perception-to-action loop is too slow, the robot (and, in general, the embedded system) might miss important events!
- **Obstacle avoidance example:**
  - What is the maximal perception-to-action loop delay (in ms) required to prevent collisions?

  **Theoretical answer:** at most 250ms (at least 4 Hz)!

  **In practice, we need much faster responses!**
Robot control – Braitenberg

Excitatory connections

Inhibitory connections
Braitenberg applied to e-puck

- 2 actuators
- 8 proximity sensors
- Motor speed is a linear combination:

\[
\begin{bmatrix}
\nu_L \\
\nu_R
\end{bmatrix} =
\begin{bmatrix}
\alpha_{L0} & \alpha_{L1} & \cdots & \alpha_{L7} \\
\alpha_{R0} & \alpha_{R1} & \cdots & \alpha_{R7}
\end{bmatrix}
\cdot
\begin{bmatrix}
d_{IR0} \\
\vdots \\
d_{IR7}
\end{bmatrix}
\]
Communication in Webots

- Webots does not simulate the communication delays:
  - Communication packets have always a delay of a time-step.
  - Which may be longer or shorter than reality.

```c
static WbDeviceTag emitter = wb_robot_get_device("emitter");
...
for (i = 0; i < 5; i++)
    wb_emitter_send(emitter, "Hello!", 7);

static WbDeviceTag receiver = wb_robot_get_device("receiver");
wb_receiver_enable(receiver, TIME_STEP);
...
while (wb_receiver_get_queue_length(receiver) > 0) {
    const char *message = wb_receiver_get_data(receiver);
    wb_receiver_next_packet(receiver);
}
```
Communication in Reality

- Everything is asynchronous and subject to variable delays.
- Packets may be lost.

- One cannot always expect that packets will arrive.
- Hence, the controller needs to account for this.
- Wireless communication is an advanced physical phenomena.
How to detect that e-puck is falling using accelerometer
The controller

- Default empty Webots controller
- Move the e-puck
- Get and enable the accelerometer
- Read accelerometer

- Check if falling
Using Webots at Home

→ Available for Windows, Linux and Mac OS X
  → Support for the lab is **only guaranteed** for the Ubuntu machines in GR B0 01

→ Download Webots 2018 installation package from
  → [http://www.cyberbotics.com](http://www.cyberbotics.com)

→ Use EPFL license:
  → Username: epfl@cyberbotics.com
  → Password: JXG01UqOFWX8u6+f7WsJgcY5k0Y=

→ Make sure to have an EPFL IP (epfl Ethernet/wifi or VPN)

→ If you need you can download locally the license for a few days (tools->Licenses Manager)
Conclusion
Take-home messages

- Programming embedded systems can be a difficult task: simulation is here to help!
- Simulation allows one to achieve rapid prototyping and experimentation **without using actual hardware.**
- However, simulation is an abstraction of reality (even though Webots, for instance, tries to be as realistic as possible); simulation does not account for all details of the targeted system (some of which can be important)
- Be especially careful about **timing issues** (i.e., perception-to-action delay): Webots does not account for these aspects (synchronous robots)
- Real-time aspects are a key ingredient of embedded systems: more about that later
Reading and acknowledgements

- If you need help about a specific function and/or device, refer to the Webots Reference Manual: http://www.cyberbotics.com/reference/
- In case of problems with the e-puck robot, refer to the official website www.e-puck.org.
- Thanks to Yvan Bourquin, ex-CTO of Cyberbotics, for his slides about Webots!