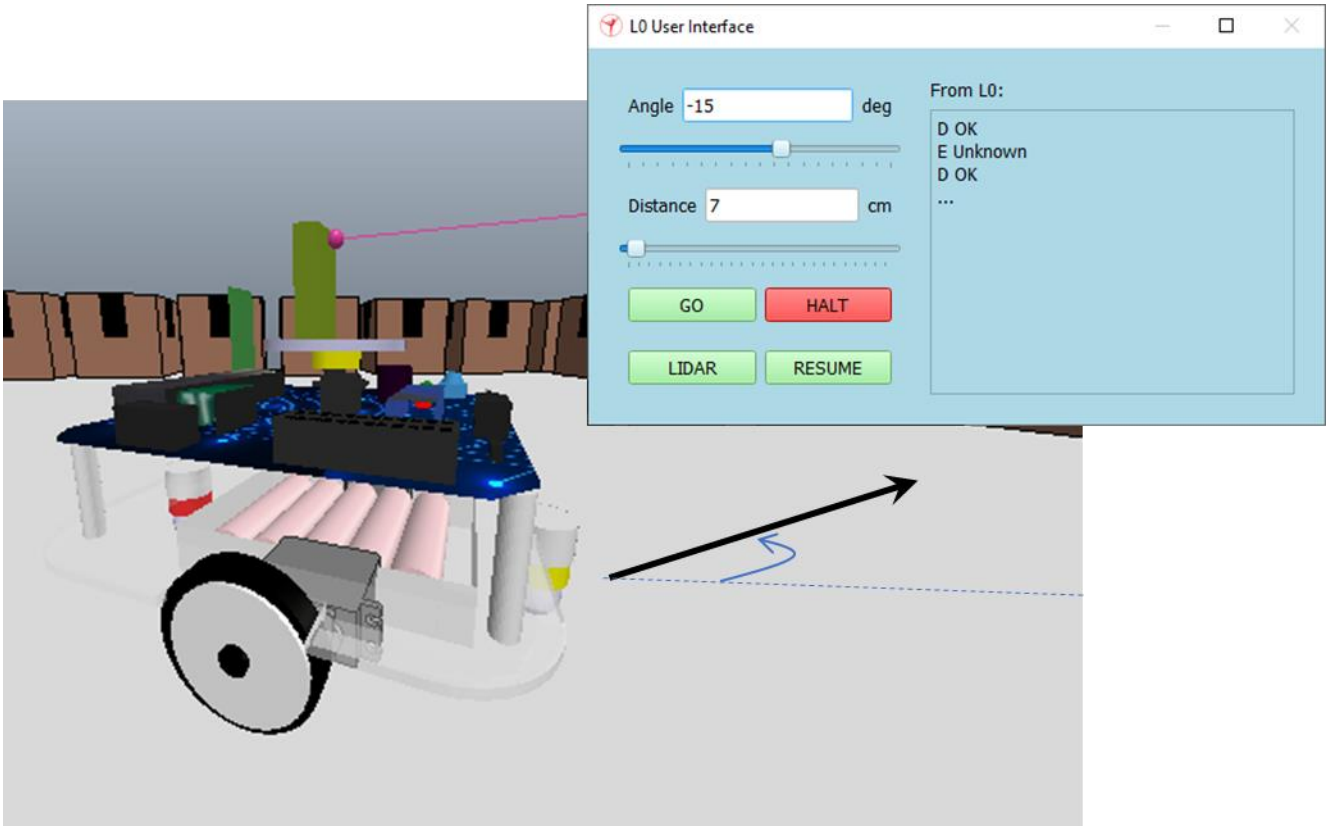
Supervisory control of mobile robots:

UABotet's GO basics

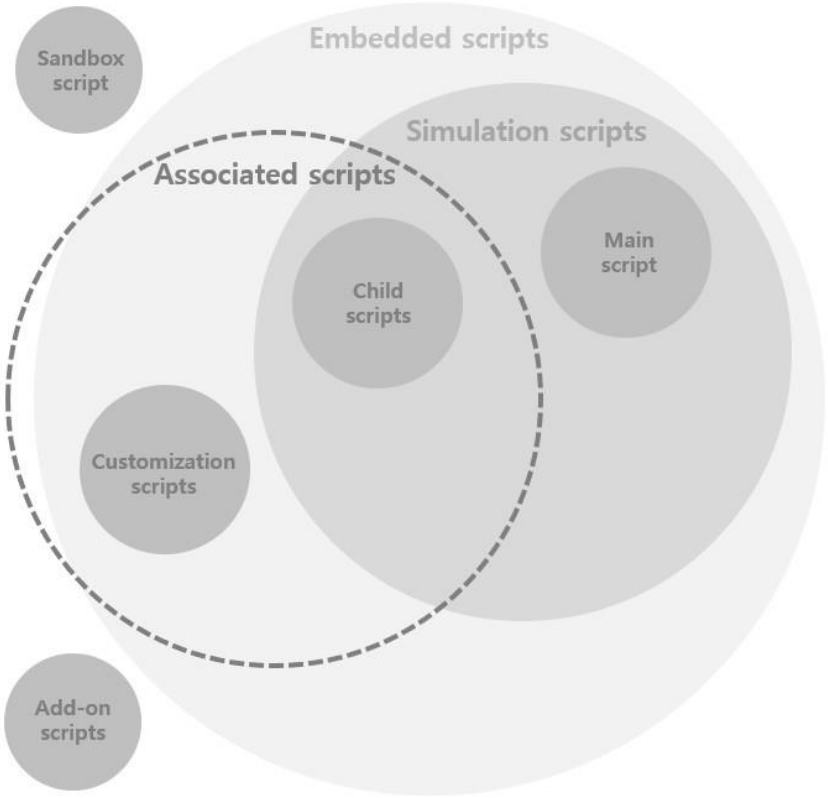


Part 1. CoppeliaSim + UABotet framework

In this laboratory, you'll learn how to program the UABotet virtual robot on CoppeliaSim.

CoppeliaSim[1]

CoppeliaSim runs a physical engine to simulate real-world physics and object interactions that change their poses (position and orientation) and shapes (only in specific physical engines) in the simulated scene. Additionally, it runs code associated with different elements of the simulator and, particularly, with scene objects.



Scripts are invoked via callback functions by CoppeliaSim and follow a specific execution order. We shall use the ones that are used at the beginning of a simulation, to get information from the scene, to update it, and before the simulation stops:

```
function sysCall_init()
    -- do some initialization here, run at the simulation start
end

function sysCall_actuation()
    -- put your actuation code here, to update the scene
end
```

[1] Part of the text and figures are taken from the CoppeliaSim User Manual [https://coppeliarobotics.com/helpFiles/index.html]

```
function sysCall_sensing()
    -- code to get data from the scene
end

function sysCall_cleanup()
    -- do some clean-up here, just before the simulation stops
end
```

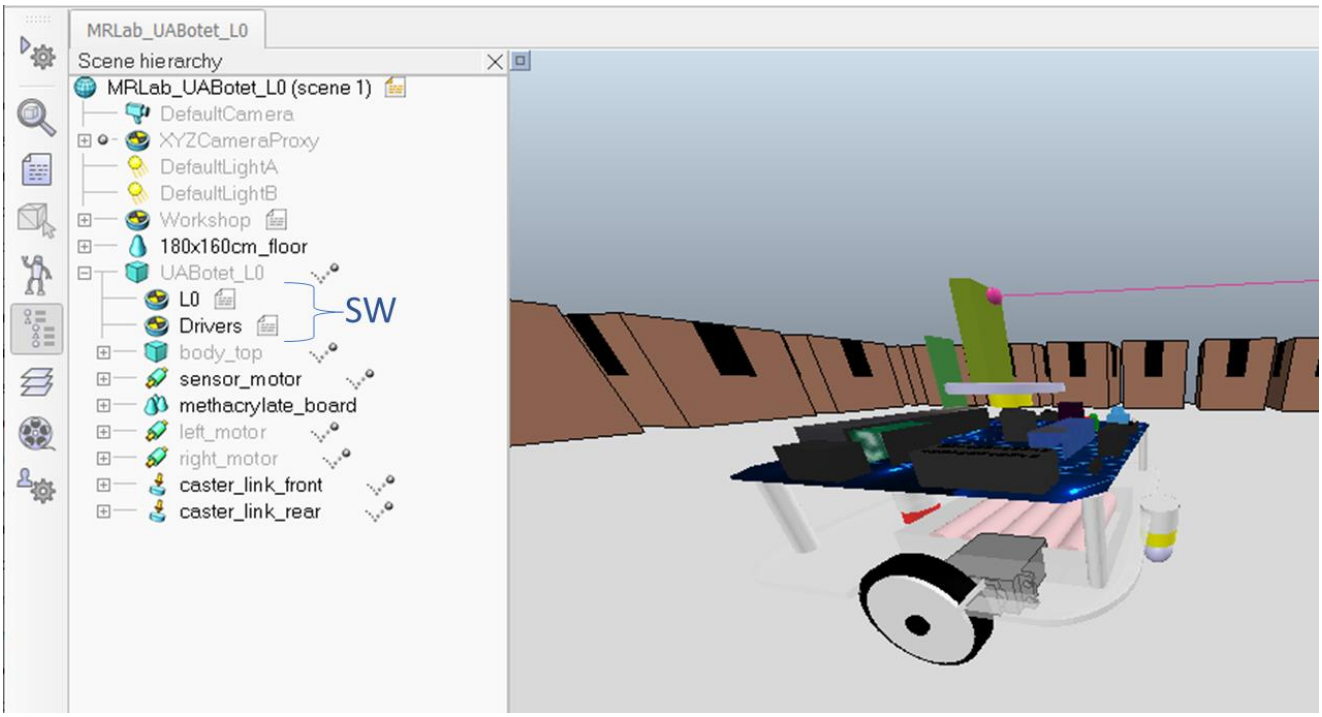
Only the **non-threaded scripts** execute in synchrony with the simulation loop. (The threaded scripts execute in parallel with the simulation loop at regular intervals and might be used as, e.g., interfaces with other software components of a system.)

Scripts are run from the lowest level in the hierarchy of objects to the upper levels and, then the global ones from the main script to the sandbox one, though we shall not use them.

The scripts are written in the **Lua** (Python will not be used) programming language, which can use the simulator's API in a straightforward way: all constants and functions are members of the `sim` table.

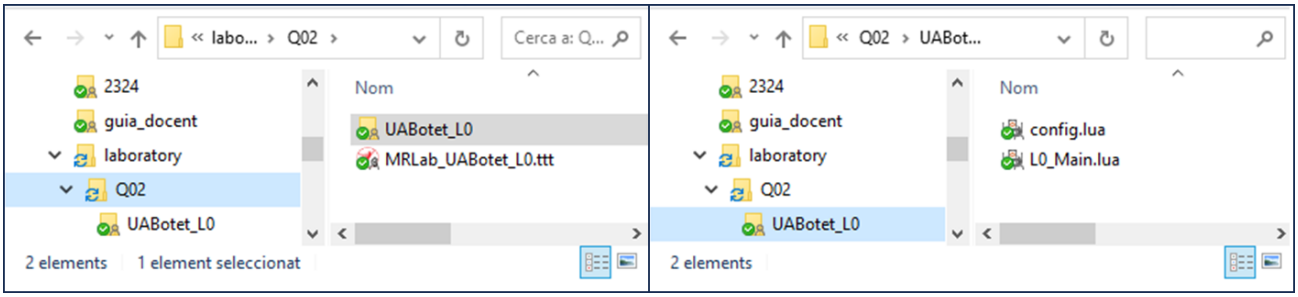
UABotet's embedded software

The model of the robot is composed of a hierarchy of objects that include two dummies: Drivers and L0.



The drivers are the controllers of the actuators and sensors of the robots and provide the L0 controller with an interface to interact with them. This way, L0 controller has not to deal with actual hardware implementation details.

While the code of the drivers is entirely included in the associated script, the code of the L0 controller is stored in an independent file in a folder with the same name than the main name for the UABotet model in the scene, i.e., "UABotet_L0". This folder must be in the same folder than the CoppeliaSim scene file, which is MRLab_UABotet_L0.ttt, in this case.



The child script of dummy object L0 within UABotet_L0 tells the Lua interpreter to read file "..\UABotet_0\L0_Main" at the simulation initialization and, then, it calls `write_outputs()` in the actuation phase and `read_inputs()`, `step()` and `forward()` in the sensing phase of each simulation cycle:

```
function sysCall_init()
    local me = sim.getObject("..")
    local myname = sim.getObjectAlias(me, -1)
    require(myname.."\\L0_Main")
    init()
    forward()
end -- function sysCall_init()

function sysCall_actuation()
    write_outputs()
end -- function sysCall_actuation()

function sysCall_sensing()
    read_inputs()
    step()
    forward()
end -- function sysCall_sensing()
```

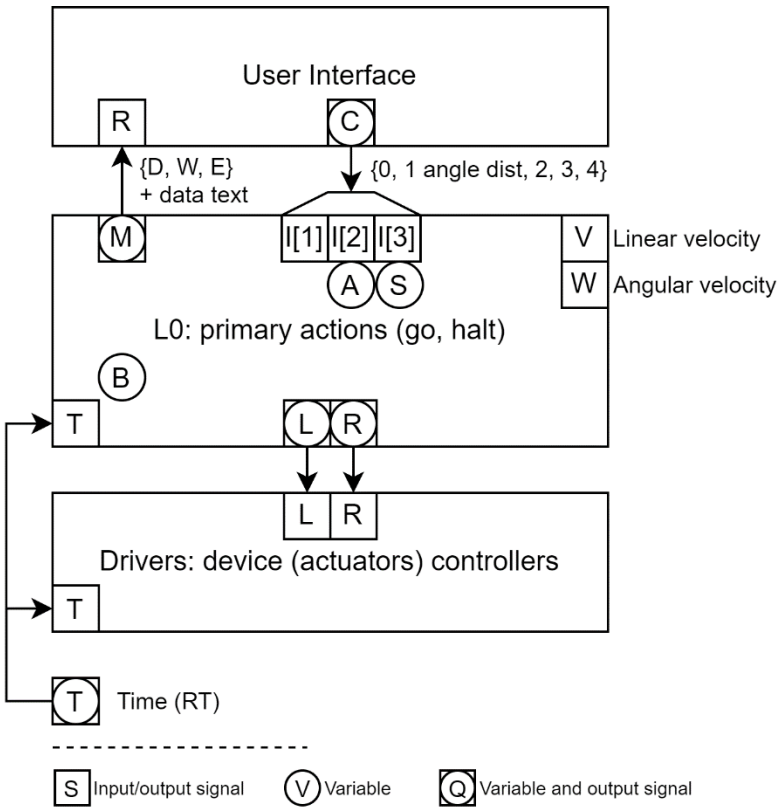
Therefore, we can edit the L0 controller's code in any IDE of our choice, e.g. ZeroBrane. As the script executes in the same program environment than the files required by it, global variables and namespace is the same.

UABotet's open-loop L0 controller

The lowest level of the supervisory control is L0, which is devoted to make the robot perform elementary operations such as to go to some point in polar coordinates, i.e. GO α *r*.

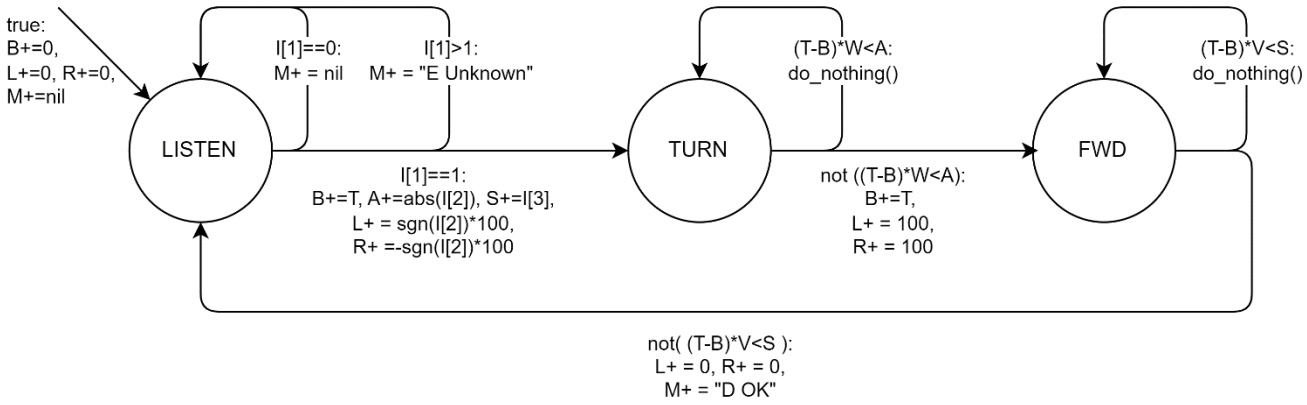
The MRLab_UABotet_L0.ttt scene includes a model of UABotet prepared to develop the L0 controller on top of the drivers (associated script of dummy object Drivers).

The inputs for L0 are the robot model parameters *V* and *W*, time (*T*), and instruction table *I* (capital *i*), which has three elements: the operation code (*I*[1] = 0 for none, 1 for GO), the angle (*I*[2] = α, in degrees) and the distance (*I*[3] = *r*, in cm):



L0 outputs are values to control the rotational velocity of left (*L*) and right (*R*) motors, which must be integers in the [–100, +100] range, and a message to the user interface, *M*.

To help model the corresponding behavior, additional state variables are used on top of the main state: *B* to hold the begin time of an operation, and *A* and *S* to store the absolute value of angle and the distance to move forward:



In the state machine diagram above, *sgn*(*x*) returns –1, 0, or +1 on *x* being <0, =0, or >0.

Task 0. Program the L0 basic controller

Complete the code of the L0 controller in L0_Main.lua and verify that it is working both, standalone and in combination with CoppeliaSim.

This task should be done before laboratory session so to solve any questions that arise when carrying it out at the beginning of it.

Task 1. Implementation of HALT

Modify the EFSM diagram to respond to HALT (operation code = 4) commands: At the LISTEN state, the controller should reply with *M* = "E HALT LISTEN?", and at TURN and FWD, with messages containing information about state and pending number of degrees or cm to go.

For instance, in case FWD is HALTed, a possible reply message would be "D FWD HALTed, 5 cm to go". To create such strings, you might use the following.

```
string.format("D FWD HALTed, %dcm to go", math.floor(val))
```

where `val` is the value of the pending number of cm to traverse.

Modify L0_Main.lua in accordance with the changes you introduced in the EFSM.

Show the result to the teacher before moving forward to the next task.

Task 2. Increase of accuracy

Note that either rotation or forward movement stop when the traversed angle or distance are equal or above the target ones. This condition might lead to movement errors of 0 to the traversed angle/distance per cycle. For instance, if the robot goes at a linear velocity of 4 cm/s and cycle time is 0.05 s, it moves 0.2 cm per cycle, and there can be a last displacement of nearly 0.2 cm. Although it is not very significative, this error can be halved if we make the controller stop rotating/moving when its close enough to the target. In the example, "close enough" would mean 0.2/2 = 0.1 cm. In this case, errors due to stopping conditions would be in the range of ±0.1 cm (same width than before but centered in the target).

Again, modify the EFSM to stop rotation/forward movement when target within half the value of last cycle's variation and change the program accordingly.

Show the result to the teacher before leaving laboratory.

Should not you finish the last task, show anyhow your work to the teacher.