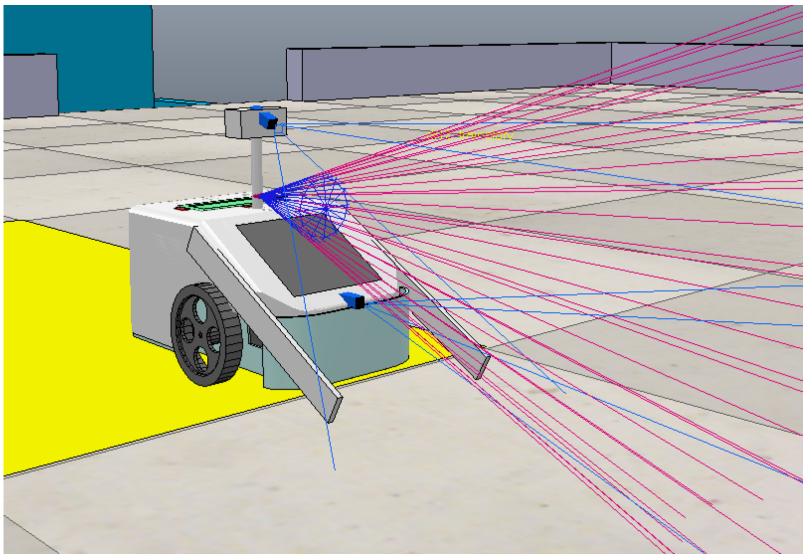
# BKI 115A INTRODUCTION ROBOTICS

PRACTICAL ASSIGNMENT



## INTRODUCTION

Welcome to the practical assignment for Introduction Robotics! This document gives you all the details you need to get started during the first practical. Primarily, it documents the assignment you have to complete.

As you will see throughout the document, you will write robot controllers to solve three different challenges. Each challenge requires the use of a **PID controller**, a **subsumption architecture**, or both. The robots are instantiated in CoppeliaSim, a state-of-the-art robotics platform, and controlled from external Python scripts. We will provide you with the necessary code for this; this allows you to focus on the design of your robot behaviours without having to worry too much about how exactly Python and CoppeliaSim communicate with each other.

The practicals are **group work**, and it is expected that every member of a group contributes to it. You are free to work on this whenever you wish; however practical sessions are organised during which TAs will be present throughout to help you if you have any questions.

Good luck!

## COPPELIASIM AND PYTHON

You will need to install CoppeliaSim and Python, if you have not already done so. CoppeliaSim can be downloaded here:

https://www.coppeliarobotics.com/downloads

Be sure to use the **EDU** version – it is fully featured and also free.

Python has a number of different implementations and IDEs. You can use your favourite, but be careful with installing multiple versions of Python.

We will provide you with skeleton Python code that allows you to connect to CoppeliaSim. There will be variables from which you can read sensor data, and variables that you can write to to send motor commands to the robot. You do not need to worry about the details of this interface, but it helps to read the documentation:

https://www.coppeliarobotics.com/helpFiles/en/legacyRemoteApiOverview.htm

Do note that, in addition to the provided skeleton code, you will need three files that are provided with CoppeliaSim. These are called *sim.py*, *simConst.py*, and *remoteAPI.XXX* (where *XXX* depends on your operating system). See the documentation for their location on your operating system:

https://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm

Copy these files into the folder where you keep the controllers you develop in these practicals.

**Always use the files that came with your own installation of CoppeliaSim, and that are suitable for your own operating system.** The files change between OS and CoppeliaSim versions and using the wrong kind can have unintended consequences.

For each of the challenges below, we will provide you with two kinds of files:

- A skeleton controller in Python (with a .py ending). This contains all the code necessary to connect to the robot in CoppeliaSim, read out the relevant sensors, and send commands to the motors. It does not implement any behaviours; that is for you to do.
- One or more CoppeliaSim scene files (with a .ttt ending). These contain the environments for the challenges, including the robot and the code for communicating with the robot.

To connect CoppeliaSim and Python, proceed as follows:

1. Start CoppeliaSim and open one of the .ttt scene files
2. Run the simulation by clicking on the corresponding button
3. Run the Python script containing the controller corresponding to the .ttt file running in CoppeliaSim. You should see messages indicating that the connection is established. If the controller does something, you should see this in the CoppeliaSim simulation.

To get started, try the following in your Python code:

- Explore the values read from the sensors (even if the robot is stationary). Manually move the robot to different locations by dragging it around in CoppeliaSim and see what happens to the sensor readings
- Send various constant commands to the motors and observe their effects in CoppeliaSim

## THE CHALLENGES

You will need to program robots to solve three challenges:

- The line following challenge
- The object sorting challenge
- The wall maze challenge

The challenges themselves are described below. First, some important general points:

- Each challenge has multiple maps. You will be given some, but we may also use a new map in the evaluation of the submission. **It is therefore not advised to hardcode solutions that will only work on one map.**
- Each challenge includes a degree of noise and randomness. This can be sensory or motor noise, or it can affect events in the environment (for example the degree to which your battery drains in the second and third challenge, or the spawn rate of objects in the second challenge). Each time you run a challenge, these aspects will vary slightly. Again, it is not advised to hardcode solutions. For example, you cannot rely on the shade of a green box always being exactly the same.
- Scoring partial points is possible on all challenges even if the primary goal is not fully met

## LINE FOLLOWING CHALLENGE

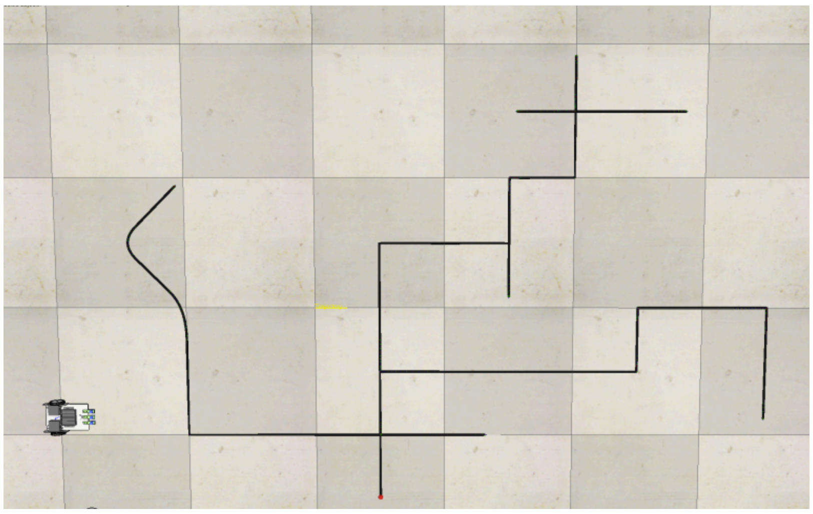


Figure 1: Example line following maze

Your robot is in an environment that has a line maze drawn on the floor. The robot has a camera pointed to the ground, so it can see what's on the floor. The robot begins its journey in a starting area, and is pointed towards the line maze. Its first task is to find the line, which it can do by moving forward (we won't trick the robot by messing with the start position). Once it has found the line, it should follow it until it has found the goal location.

For this task, you will need to implement a PID controller. You do not need to consider a subsumption architecture, but it may help in some decision making while navigating the maze. To solve this task, you will need to think about the following:

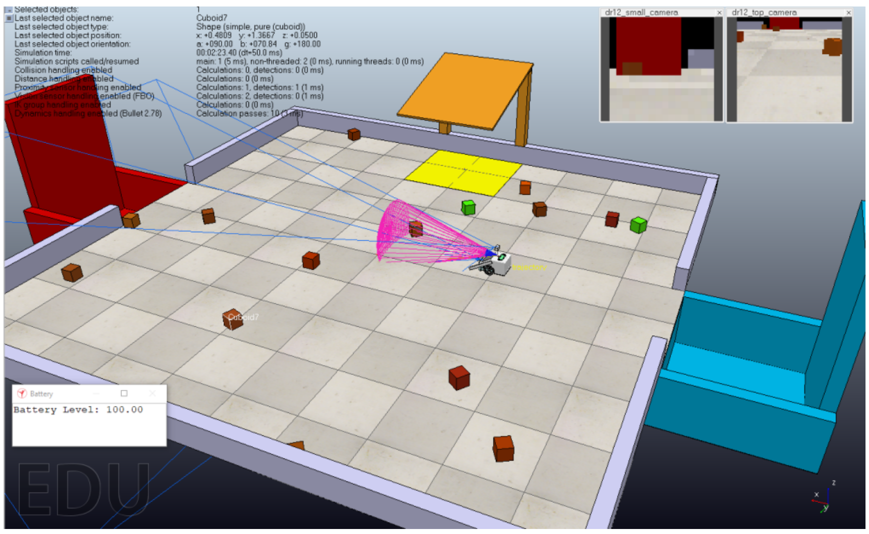
- What is the error signal? It needs to be something you can evaluate from the camera image you obtain, and it should contain information about whether the heading of the robot is currently too far to the left or the right
- Some changes in the lines are abrupt; for example, 90 degree turns. Your robot is likely to lose track of the line every now and then. How will it find it back?
- The maze contains some crossings. How do you explore where the different paths lead to?

### PRIMARY GOAL

Your objective is to navigate the maze until you find the goal location. If you achieve this, the task is considered fully completed.

### FAILURE

You fail the task if your robot is unable to follow the lines or you do not use a PID controller for this.



```
Selected objects:                       1
Last selected object name:              Cuboid7
Last selected object type:              Shape (simple, pure (cuboid))
Last selected object position:          x: +0.4809   y: +1.3667   z: +0.0500
Last selected object orientation:       a: +090.00   b: +070.84   g: +180.00
Simulation time:                        00:02:23.40 (dt=50.0 ms)
Simulation scripts called/resumed       main: 1 (5 ms), non-threaded: 2 (0 ms), running threads: 0 (0 ms)
Collision handling enabled              Calculations: 0, detections: 0 (0 ms)
Distance handling enabled               Calculations: 0 (0 ms)
Proximity sensor handling enabled       Calculations: 1, detections: 1 (1 ms)
Vision sensor handling enabled (FBO)    Calculations: 2, detections: 0 (1 ms)
IK group handling enabled               Calculations: 0 (0 ms)
Dynamics handling enabled (Bullet 2.78) Calculation passes: 10 (3 ms)
```

```
Battery Level: 100.00
```

Figure 2: example arena. Note that different maps may also contain walls inside the arena

Your robot is spawned inside an arena that features two large containers on either side and a charging pad. Small objects spawn over time; these are either plants (green cubes) or thrash (brownish cubes). Plant cubes can be pushed around. Thrash cubes will be compacted by the robot: approaching them from the front will allow you to compact them, resulting in a smaller cube being produced at the back of the robot.

In addition, your robot will slowly run out of battery, and the rate at which the battery decays depends on its interactions with objects. You can recharge at any time by entering the charging area.

For this task, you should develop a subsumption architecture. You do not need a PID controller.

## PRIMARY GOAL

Your objective is to sort all plant boxes into the blue container, compact all thrash boxes, and sort the compacted cubes into the red container. You should achieve this as quickly as possible.

## FAILURE

You can only fail this task if you fail to sort any box into its correct container.

## WALL MAZE CHALLENGE

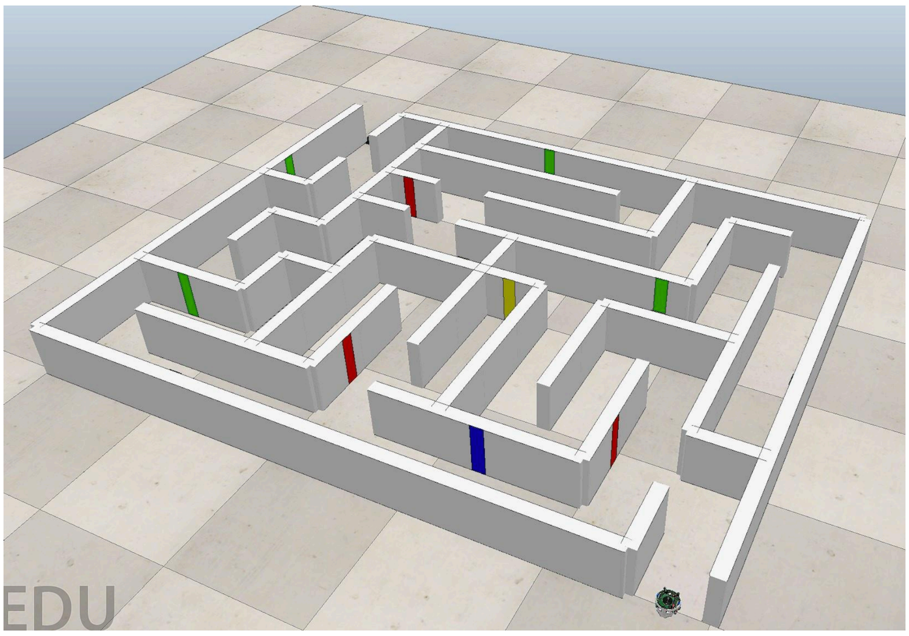


Figure 3: An example maze, with the colours for active (green) and inactive (red) charge points as well as goal locations that have not been found (yellow) and those that have been found (blue).

Your robot is at the entrance of a maze. Its overall objective is to find the exit at the other end. However, it also needs to visit a number of goal locations before it can exit the maze. Lastly, the robot's battery also drains over time, so that it needs to charge every now and then.

The goal locations and charge locations are indicated through colours on the wall. They behave as follows:

- Charge stations, when active, will fully charge your battery and then become inactive for some time, after which they become available again. The colour of a charge station changes depending on whether or not it is active
- Goal locations will permanently change colour once visited.

You need to solve the task using a subsumption architecture. A PID controller is optional but can be useful if you want to follow walls at a certain distance.

### PRIMARY GOAL

Your objective in this task is to find all goal locations (the number may vary between mazes, but will be known to you), recharging as needed and then exit the maze. You should achieve this as quickly as possible.

### FAILURE

You can only fail this task entirely if you somehow fail to enter the maze or fail to find a single goal location.

## SUBMISSION AND EVALUATION

At the end, you will submit

- The code you have developed
- A report documenting the code (see below)

For each challenge, we will run your code on both a map that you have seen during the practicals, and on a new map. For each challenge, your score is determined by:

1. The quality of your implemented solution and its documentation in the report **independently of how well it performs during the evaluation**. An idea that is sound in principle but does not fully solve a challenge will still score well.
2. The performance of your controllers on the known and unknown maps. A controller that reliably solves a task will score full marks, while partial points are awarded for controllers that solve at least some parts of the challenges some of the time.

## REPORT

In addition to the code, you are asked to produce a report. It documents your solutions for each of the challenges. You should describe the controller you implemented, focussing on the design of the mandatory aspects (either a PID controller or a subsumption architecture depending on the task) and any evaluations you may have carried out. If you decided to design additional maps, you can also document them here. You can include videos, for example to document a successful run, but please keep the file size small. **It is important that the reader of this part understands how you designed your controller (e.g. how you tuned your PID controller or how you decided what the individual components of your subsumption architecture were going to be) and how you implemented it.** The discussion in the report should match the code you submitted.

In addition, you will be asked to fill out a peer review form. This is a standard from used in group work exercises, where you can give individual feedback on both yourself and the other members of the group, and the process of working together in these practicals.  This can be used to modulate marks, for example, when it becomes evident that one group member did all the work or a group member did not contribute at all.

**All members of a group submit the same group report but individual peer assessment forms.**

## GRADING SUMMARY

- Challenge 1: **10p total**, broken down into:
    - Documentation in the report: 5p
    - Performance on known map: 1,5p
    - Performance on unknown map: 3,5p
- Challenge 2: **30p total**, broken down into:
    - Documentation in the report: 15p
    - Performance on known map: 4,5p
    - Performance on unknown map: 10,5p
- Challenge 3: **20p total**, broken down into:
    - Documentation in the report: 10p
    - Performance on known map: 3p
    - Performance on unknown map: 7p

Total maximum: **60p** (30p conceptualisation and documentation of solutions and 30p performance on challenges).