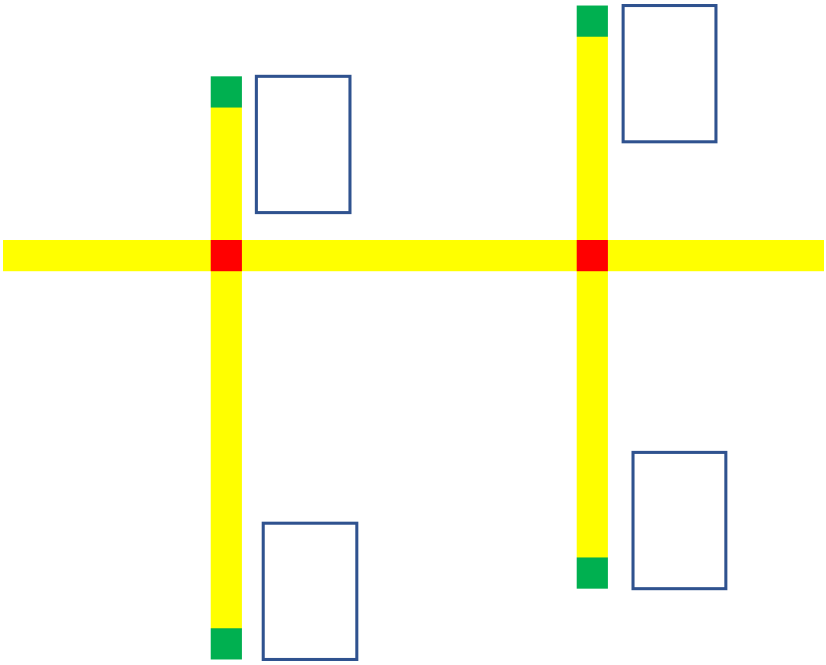
Suggested Design-Build-Test Mini-projects

[CBPrice 19-11-20]

1. Line-Following Warehouse Picker

Robot is given a shopping list and has to collect a series of items from racks in a warehouse. It follows a yellow line where junctions are labelled red and end-points near racks green:



First get the robot to navigate to any specified rack. You will probably use a finite state machine

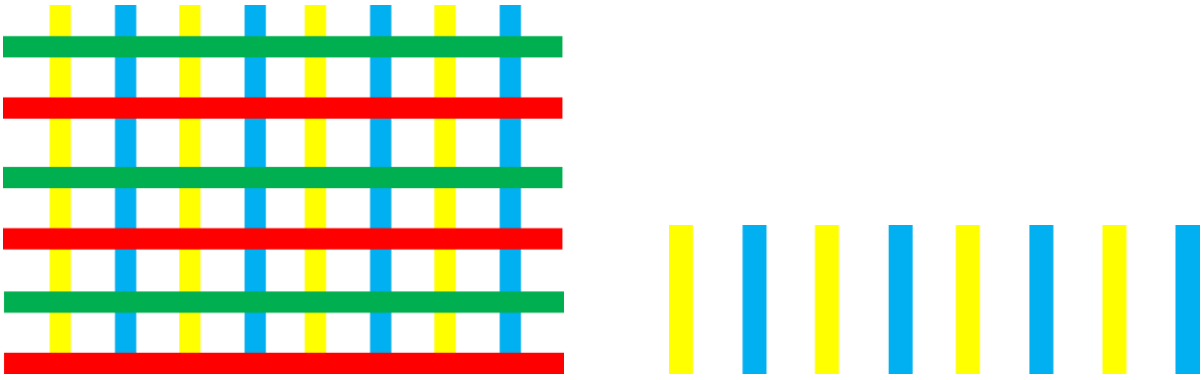
Then get it to collect items on the list. Think about the order. Could you achieve an optimal (shortest) journey?

Controller: CBP_ePuck_Camera_6.c **World: CBP_ePuck_Camera_6.wbt**

Controller: CBP_ePuck_LinearCamera_1.c **World: CBP_ePuck_LinearCamera_1.wbt**

2. Localization using Pairs of Colored Lines

Look at the grid below containing two repeated colored lines in the vertical and in the horizontal direction. It's possible to write an algorithm so the robot know which square it is in. Your task here is to write that algorithms then implement it in Webots. Start by solving the 1D problem on the right



Controller: CBP_ePuck_Camera_6.c **World: CBP_ePuck_Camera_6.wbt**

3. Maze-Following Algorithm

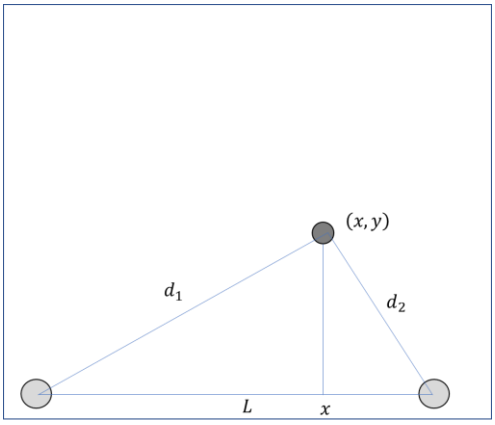
Develop or complete the code for the follow the left wall algorithm. Or, if already done, implement the follow-right algorithm or any other algorithm of your choice. You could design your own maze (the one provided is modular, so just copy and paste some of the walls, which are arranged on a grid of size 0.112 metres.

The algorithm presented in class uses odometry to locate the robot and to rotate it. As you know, there are errors in odometry, and these accumulate in time, as the robot moves. So another approach would be to investigate how to correct these errors. You could start with a simple scenario, such as a single long corridor and produce error correction to get the robot to move along the centre of this corridor.

Controller: CBP_ePuck_Maze_5.c **World: CBP_ePuck_Maze_3b.wbt**

4. Localization using Beacons

Set up a world with two beacons as shown below. Take range measurements of the beacons and so deduce the robots (x,y) location. Compare accuracy with GPS values.



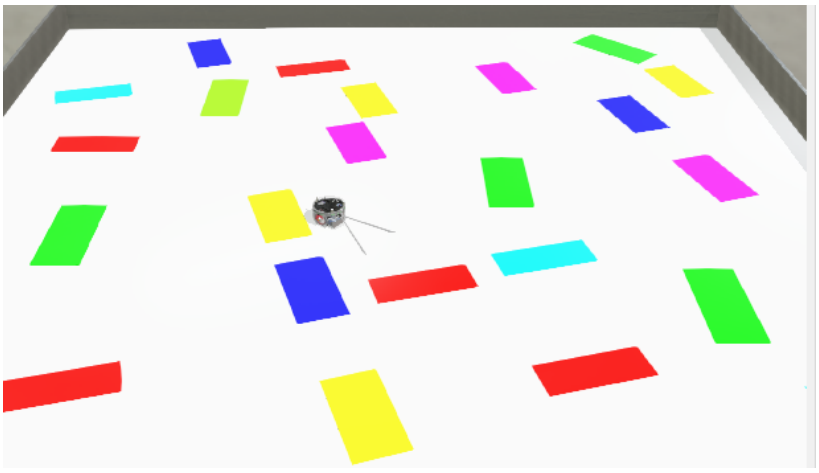
The origin is at the left beacon. The distance between the two beacons is *L*. The distances measured by the sensors (or computed from their readings) are d_1 and d_2. Use Pythagoras theorem applied to the two triangles to find an expression for x in terms of d_1, d_2 and *L*. Ask for help if you like.

What to use for the beacons? Here are some possibilities (some better than others)
1) Two light sources – use light sensors to measure the distance
2) Colored cylinders – use camera to identify and tine-of-flight to measure the distance

World: CBP_ePuck_objDet_1.wbt and associated controller

5. Robot Behaviour directed by Colored Blobs

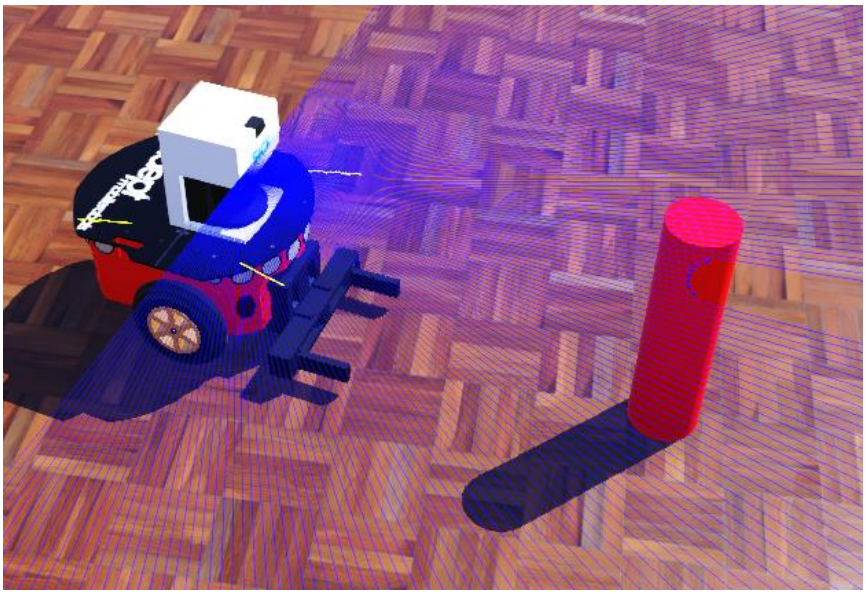
Think how to use the camera and colored blobs to produce a meaningful robot behaviour.



Controller: CBP_ePuck_Camera_6.c **World: CBP_ePuck_Camera_6.wbt**

6. Pick and Place

The Pioneer3 robot shown below has been given a gripper in addition to its LIDAR scanner. The task is to get the robot to look for cylinders and to pick them up and transport them to a place

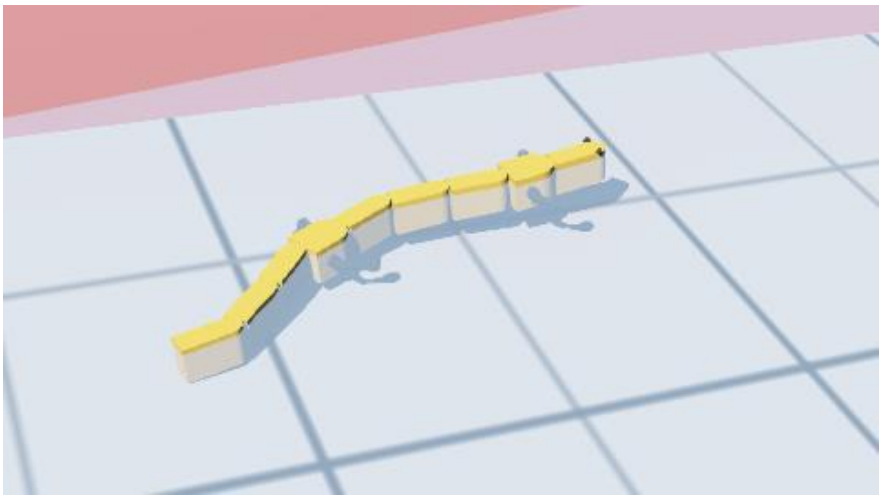


Perhaps the most challenging part of this is to decide how to get the robot to recognize where the 'put-down' place actually is.

Controller: CBP_Pioneer3_Gripper_2.c **World: CBP_Pioneer_Pickup_1.wbt**

7. Salamander

Here you will code a Salamander which can both walk and swim. When it moves from land to water it changes gait from walking to swimming

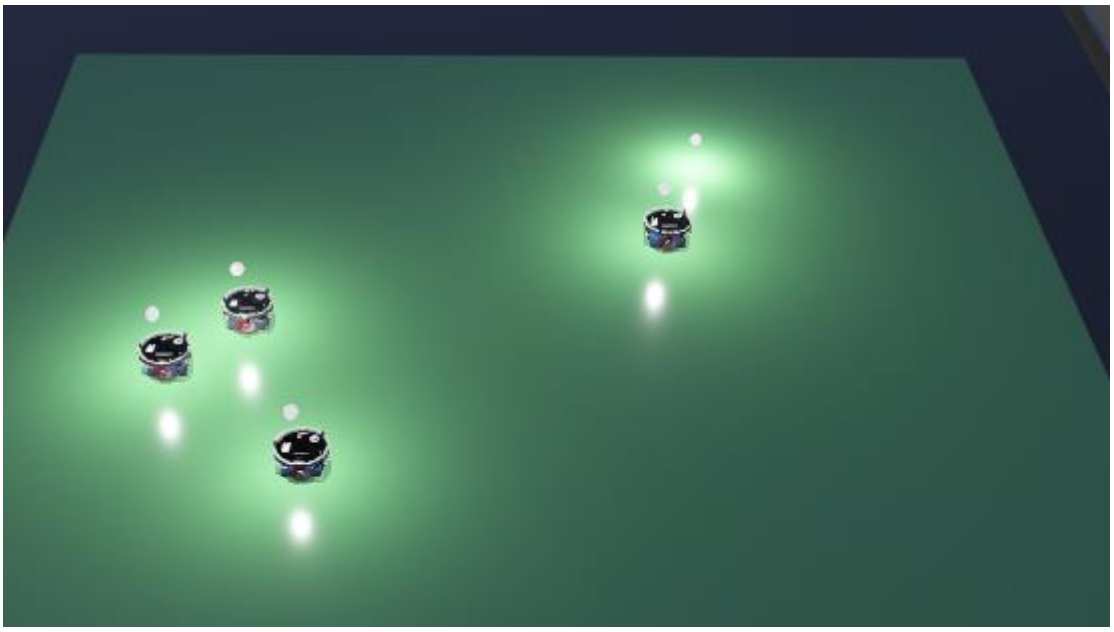


Code: CBP_Salamander_4.c **World: CBP_Salamander_4.wbt** **See additional notes.**

8. Self-Organizing Light-Following Robots

Add a light to the turret slot of an ePuck and code a Braitenberg-like controller. Now replicate the ePuck N times (where you could progressively increase N). See what patterns emerge.

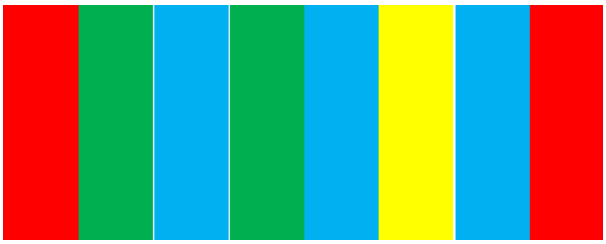
You may want to use all light sensors, or just stick with two.



Controller: CBP_ePuck_Braitenberg_Light.c **World: CBP_ePuck_Braitenberg_Light.wbt**

9. Directing Robot Behaviour using "Bar-Codes"

The robot would encounter 'flats' in the world, which could inform it what to do. The 'flat' would contain a series of colored stripes (which you would design). In the example below, the red stripes are used to identify the edges of the image. The robot could search for some red in the world, then rotate so that there is red at the left and the right of the image.



Then you could scan the image (one row) and detect the colored stripes. So in this example we have GBGBYB. So there are 3 colors and 6 slots for the stripes. This means the number of codes we can use is $6^3 = 216$. That's a lot. But you get the idea. So you may want to use less slots or fewer colors.

Controller: CBP_ePuck_Camera_6.c **World: CBP_ePuck_Camera_6.wbt**

10. Line Following with Obstacles

Detecting the obstacle is quite straightforward, but how do you get the robot back onto the line? What happens if the obstacle is another robot coming the other way?

Perhaps a mix of the two below. Probably use a FSM.

Controller: CBP_ePuck_LinearCamera_1.c **World: CBP_ePuck_LinearCamera_1.wbt**

Controller: CBP_ePuck_ObjDet_1.c **World: CBP_ePuck_ObjDet_1.wbt**

11. Reverse-engineer, augment or repurpose sample worlds or their robots.

1) Have a look at \vehicles\worlds**city.wbt** (vehicles section) where a car drives along the yellow line separating the lanes. This is just daft. You could work out how to get the car to drive in the centre of the lane. Also have several cars which drive autonomously and do not collide.

2) There's a rather daft game of soccer at \samples\demos\worlds**soccer.wbt** but the author has indicated how improvements can be made. This uses an architecture we have not used supervisor-client communications using radio packets.

3) There are tons of robots in the \robots section. There's even \robots\sphero\bb8\worlds**bb-8.wbt** It has a camera. Perhaps you could do something with this, perhaps in another world, perhaps not.

4) One I quite like is \robots\nasa\worlds**sojourner.wbt** With the addition of a camera, this could be turned in a useful AGV. The trick here will be to develop an image processing function to recognize rocks and avoid them. I suggest the best approach is to do some sort of texture analysis. Here is a useful link I just found https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect12.pdf

How to approach a Design-Build-Test mini-project

Well DBT is a classic approach in engineering (mechanical, aeronautical, civil, software). So here are the stages we shall go through.

Design. For your chosen project you will come up with a design. This will start with a choice of software architecture, e.g., to use a Finite State Machine or another approach. Then you will flesh out the details; in the case of a FSM, you will select the states. Alternatively, you will identify some functions you will need to provide sub-goals for your solution. All this could be in your mind, or a mix of mindwork + paper sketches and conversations with other folk.

Build. In this phase you will start to code your design, debug and evaluate. You may find you may need to amend your design; perhaps elements of your design do not work as intended. But the outcome of this phase is working code which you will be happy with.

Test. In an engineering situation, testing is conducted against the original problem specification. Here we take a more liberal view, and understand testing as an *investigation* of your artefact, so we are perhaps more like scientists than engineers. So what should you do? Well here are some suggestions.

1) Increase the 'scale' of the solution. For project (1) you could increase the size of the warehouse and the length of the shopping list. You could measure time to completion based on how you deal with the shopping list. For project (3) you could investigate various mazes, of increasing scale or complexity (whatever that means) and again measure time to completion. For (8) you could investigate pattern formation for an increasing number of robots. Here you will look to classify the geometrical patterns that emerge.

2) Investigate the effects of parameters. For project (7) you could investigate parameters, such as the frequency and amplitude of the swimming motion. You could measure the swimming speed as a function of these. For (8) you could see how parameter changes affect the time to form a stable pattern.

3) Engineering tests. Here you will investigate how well your solution worked. For project (1) did the robot lose its localization on the yellow line, did it fail to interpret the colored markers? In (9) did the bar-code recognizer always work, or were there occasions where it failed? In (7) were there occasions where the robot did not pick up the object, or drop it, or did not release it? In (2) were there occasions where the localization failed? Did you amend your pattern to improve this? *These engineering tests may feed back into the Build phase, where you amend some aspect of your solution.*