1 Summary

A farmer was planting a tree in their back yard when the ground caved in uncovering an entry way to an ancient underground maze. A research team is assembled. They send a robot into the maze. The robot must map the maze while taking pictures of all encountered objects. The maze may include cliffs and traps. The robot must protect itself while in the maze. Scientists are excited and have software (payload) they want to download to the robot. This payload will execute analytics on data received. It may be buggy, but it needs to be executed in the maze because limited bandwidth prevents the robot from reporting all data outside. The robot should complete the mission as quickly as possible.

2 Problem Description

Write a C++ program that drives iRobot by closely following the walls, i.e. the contour of a maze. You must use the wall signal, and try to drive parallel to the wall edges. In practice, it will be hard to drive completely parallel, so you are allowed occasional bumps into the wall to navigate the robot properly. However, excessive bumps are prohibited. While navigating the maze, it will plot its contour as a polygon. At the same time, the robot will also identify known objects inside the maze. You will be given a database of pictures of known objects. The robot will use the camera to identify printed instances of those pictures placed inside the maze. At the same time, it will run scientific tasks given to you as binaries. There can be arbitrary number of tasks, starting at arbitrary times, and taking arbitrary amount of time to finish (or not finish at all). The robot must ensure its own safety, and complete navigating the maze as fast as it can.
2.1 Following Wall

Initially the robot will have an arbitrary direction towards a wall inside the maze. From there, it will start driving until it reaches very near to the wall, or bumps it. Use the wall sensor and the bump sensor to detect this event. After this event the robot should rotate to align its direction parallel to the wall. The *robot will be parallel to wall when the wall signal reaches a local maxima while rotating*. This rotation must be counterclockwise. After alignment, the robot should drive close to the wall. If it's too near the wall, it will bump. If it's far from the wall, it will lose the wall signal. Both are unexpected events that you have to minimize.

Once you reach a corner, you will likely bump another wall, in which case you need to turn left. If the wall ends in your direction, you need to turn right. Note that the robot always drives on the right relative to its direction, *i.e.* right side of the robot should always be near the wall. This is because the wall sensor is placed on that side. For the same reason, left turns will always be counterclockwise, and right turns will always be clockwise. Figure 1 shows an example maze traversal.

2.2 Robot Safety

Inside the maze there are dangers, and the robot needs to keep itself safe *at all times*. Therefore it should continuously monitor the four Cliff Signals, Wheel Drop Sensor, and Wheel Overcurrent. Note that we are interested in the analog cliff signal values, not the binary cliff detectors. Cliff signals come from optical sensors located at the bottom of iRobot. They give you information on whether the robot is about to fall off a cliff. The wheel drop sensor gives information about whether the robot has been picked up, or is in a position where the wheels are not touching a surface. The wheel overcurrent sensor gives information about whether the wheels are stuck. When the robot is trying to drive, but is
stuck somewhere, the wheels draw excessive current. The wheel overcurrent sensor detect that. The robot should stop its motors, and play a song while any such unexpected event happens. It should resume operation, when the situation is fixed.

2.3 Plotting Contour

While the robot is navigating it will plot the contour using OpenCV. Because we do not have any location service on the robot, we will be using coordinates relative to the robot. At the beginning, wait until the robot reaches the first wall and aligns itself parallel to the wall. Now consider the present location of the robot as $(0,0)$. Note that the movements of the robot can be simplified as translations or rotations. Track those operations, and accordingly generate waypoints for the maze traversal. Once the robot has completed the traversal, plot the waypoints, and connect those using straight lines. Store the figure as a file. A tutorial on basic 2D drawing using OpenCV is available here http://docs.opencv.org/3.1.0/d3/d96/tutorial_basic_geometric_drawing.html

Suppose the robot is currently at $(10\,\text{mm},\,20\,\text{mm})$, and it is driving for $200\,\text{ms}$ in the direction of negative $y$-axis at a speed of $300\,\text{mms}^{-1}$, the new location is $(10\,\text{mm},\,-40\,\text{mm})$. Although the corner angles in the test cases are not necessarily right angles, for the sake of plotting the contour, you are allowed to assume that. To make the computation easier, you are encouraged to maintain a matrix $M$ of transformations, and use homogenous matrix forms for translations and rotations. If $M$ is your present matrix, and you apply a rotation of angle $\theta$, represented by matrix $R_\theta$, you can update $M \leftarrow M \cdot R_\theta$. Similarly, if $T_x$ is the matrix for translation, you can apply $M \leftarrow M \cdot T_x$. Details on homogeneous coordinates https://en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations. Details on matrix forms for 2D rotations can be found at https://en.wikipedia.org/wiki/Transformation_matrix#Examples_in_2D_computer_graphics.

2.4 Object Identification

*Place the camera facing left on the left side of the cargo bay.* Attach it firmly with a sticky tape. Placing the camera on the left side will allow the robot to actually see inside the maze. Continuously take sample frames using the camera, and match the frames against the pictures in http://goo.gl/2rwbzP

To get an image from the camera, we used function RaspiCam::retrieve. This function takes an object of class cv::Mat, which is a OpenCV matrix storing color information for every pixel of the image. We stored it as a file using the function cv::imwrite. For the purpose of identifying objects, you do not need to store the image as a file. Rather, you can directly use the cv::Mat matrix to look for the known pictures. For testing, we will strategically place objects as “printed pictures” inside the maze. If you have a match, store the respective image. You can assume that a particular object from the database will appear
at most once inside the maze. If an object has already been found, you can stop looking for that particular object in future iterations.

2.5 Safety, wall-following, plotting contour requirement

TA might lift the robot during its travel to test safety requirements. TA will not rotate the robot when lifting. To avoid wasting time, it will only be lifted at most once. Failure to stop motor after being lifted will result in losing score.

Wall following allows segmented or slightly curved trajectory, as long as the robot does not go away too far. There is no standard grading criteria, it is TA who decides if the trajectory is satisfactory.

After completing the maze, show TA the plotted trajectory. It is understandable that there might be some inaccuracy compared to the actual trajectory, as long as the angle, shape, and distance are reasonably recorded, it is fine. It is TA who makes the judgment for this requirement.

2.6 Time and object identification requirement

The time to travel through 7 walls is 120 seconds with no grading penalty. For every additional second exceeding 120s, 5% of score is subtracted. That is, if your total time exceeds 140s, you will earn 0 for “Time to complete mission” requirement.

It is allowed to take photos and store them in memory without doing processing and matching during the travel. You can process taken photos after your total travel, it will not count as travel time. For example, TA would place xxx.jpg yyy.jpg printed pictures on the maze, after completing the maze, just match the photos in RPi’s memory and store matched items as xxx.jpg yyy.jpg etc. The grading for this is based on the portion of correctly matched pictures.

2.7 External Tasks

We will provide external tasks to run. These tasks represent scientific analysis. Your program should not run those external tasks in your code. Note that the
task is run at external terminal as binary executable. They can hog CPU resources. You must ensure safety, progress, and performance, while running these external tasks.

The external task will be given at the demo as “./external.cc” which can look like the following (may be different and will not be given now):

```cpp
int main(int argc, char const *argv[]) {
    int array1[10];
    int array2[20];
    int count=0;

    while (1) {
        for (int j = 0; j < 100000; j++) {
            array1[j % 10] = array2[j % 10] * 20 - array2[j % 10];
            array2[(j + 5) % 10] = array1[j % 10] * 20.03 - array2[(j + 100) % 10];
        }
        for (int i = 0; i < 10; i++) {
            array1[i] = 6 * i + i * i - i % 10;
            array2[i] = array1[i] - array1[(i+60) % 10] * 60;
        }
        usleep(10 * 1000);
        count++; cout<< count << endl;
    }

    return 0;
}
```

// Or it can simply be like this:
```cpp
int main() {
    int count = 0;
    while(true) {
        count += 1;
        cout<< count << endl;
    }
}
```

It will be compiled using:

```bash
g++ -std=c++11 -o external external.cc
```

Then when your program is running, TA will open another terminal and run commands like:

```bash
nice -19 ./external
```

Which runs the external task in lowest priority. The output loop-count number will be used as an indicator of how well the task is run. The number should be
sufficiently large for “scientific payload”. MP2 does not set grading criterion for external task loop count, however MP3 does.

### 2.8 Architecture

You have to find an architecture that allows the mission to complete while ensuring the safety, progress, and performance requirements. *You must use multiple threads, and assign appropriate priorities to those.*

### 3 Project Report

Write a report that illustrates your architecture with figures. Explain which thread relates to which requirement, and explain their priority levels. The report should also include the algorithms used for navigating the robot. Explain the major issues encountered while solving the problem, and how you solved those. Report the maximum speed you could run your robot during your tests. Include *three different examples* of contours plotted using your program, and the locations where you tested it.

### 4 Submission and Grading

Submission will be done via UofI Box. *Submit your code, and the project report in pdf format.* Create a folder `mp2-robotpiN` where `N` is your group number, and copy all your files (from RPi) including the project report into it. Upload the `mp2-robotpiN` folder to the box folder shared to your group. You will submit your final code and report by the deadline, and later come for a demo. You need to bring your own equipment for the demo. No code change is allowed after the deadline has passed.

During the demo, TA will download the mp file from box.com. TA will copy the `mp2-robotpiN` file to RPi using scp. TA will compile and run the code, as well as external task. You are supposed to write Makefile. Be prepared to explain your work and answer questions during the demo. The following table shows distribution of points.
<table>
<thead>
<tr>
<th>Requirement</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robot Safety</td>
<td>20%</td>
</tr>
<tr>
<td>Following Wall</td>
<td>30%</td>
</tr>
<tr>
<td>Mapping Contour</td>
<td>10%</td>
</tr>
<tr>
<td>Object Identification</td>
<td>10%</td>
</tr>
<tr>
<td>Time to complete mission</td>
<td>10%</td>
</tr>
<tr>
<td>Demonstration and Q/A</td>
<td>10%</td>
</tr>
<tr>
<td>Project Report</td>
<td>10%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>