Task 2B

Generate Maze in CoppeliaSim with Remote API

[*Last Updated on:* ***12th November 2020, 22:39 Hrs***]

The aim of this task is as follows:

- This task is based on **maze generation in CoppeliaSim** using the **output of Task 1B**.
- Teams should use **python remote API** to **transmit the encoded maze array** (output of **Task 1B**) to a **Lua script** in CoppeliaSim scene to **generate walls of the maze**.

Make sure you go through the instructions thoroughly and in a sequential order. It will help you understand the tasks better.

1. Given

- A set of **9 maze images** are given in the ***test_cases*** folder. An example image is shown in Figu 1.

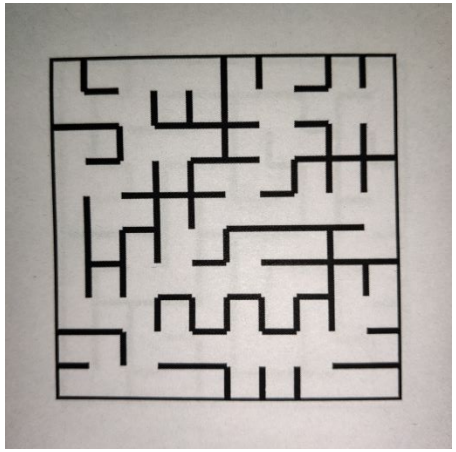

Figure 1: Example Maze Image

2. Problem Statement

- In this task, you will be using the `applyPerspectiveTransform()` and the `detectMaze()` functions that you developed in **Task 1B** to detect the mazes in given images and construct th corresponding maze in the given CoppeliaSim scene.
- The flow of Task 2B is depicted in Figure 2.

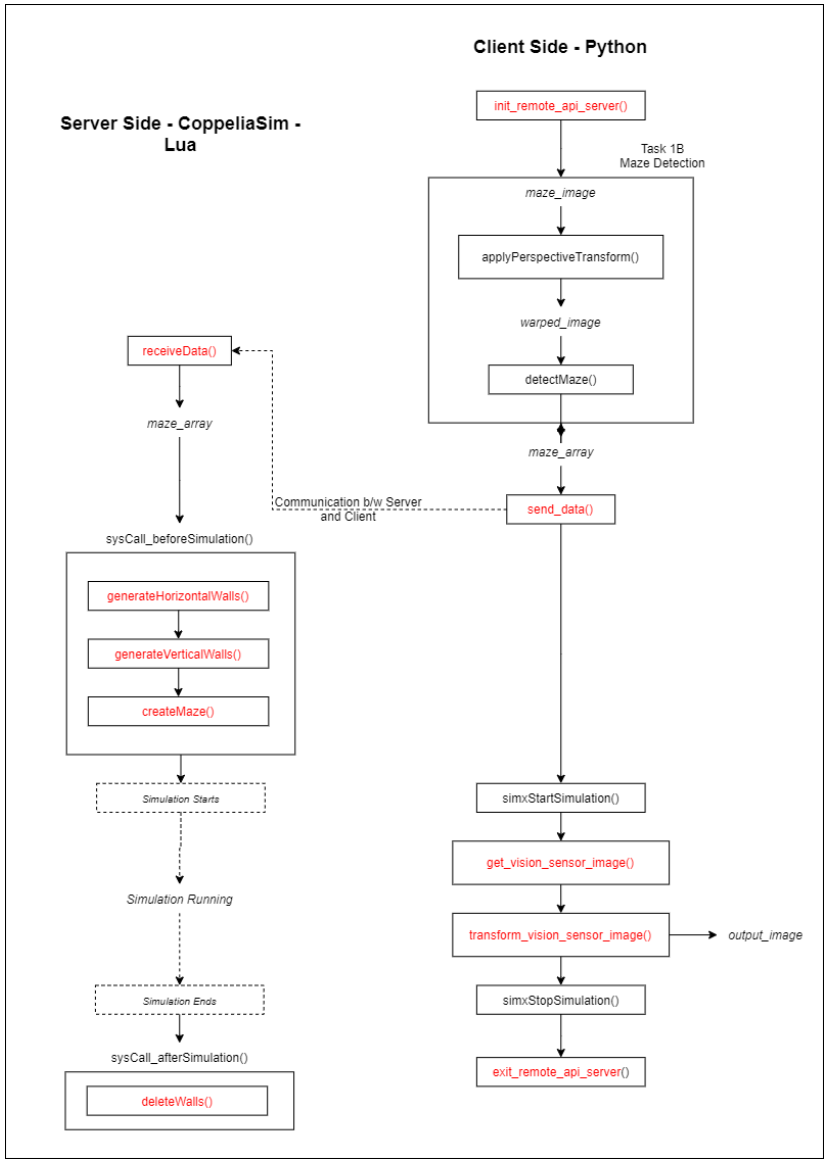

Figure 2: Task 2B Flowchart

- There are **10 pre-written** functions that you ***have to modify***. These are highlighted in ***red*** in Figure 2.
- On Python side, you have to modify **five** functions:
  - `init_remote_api_server` and `exit_remote_api_server`

    - These are used for connecting and disconnecting with the CoppeliaSim Remote AP server. You have already implemented these functions in Task 2A.
  - `get_vision_sensor_image` and `transform_vision_sensor_image`
    - These are used for interacting with Vision Sensor in CoppeliaSim scene. You have implemented these functions as well in Task 2A.
  - `send_data`
    - This function takes the encoded maze array as input that is computed by the `detectMaze()` function and sends it to the CoppeliaSim Remote API server.
- On Lua side, you have to modify **five** functions:
  - `receiveData()`
    - This function receives the encoded maze array from the Python Remote API client.
  - `generateHorizontalWalls()`, `generateVerticalWalls()` and `createMaze()`
    - These functions are used for creating the maze in CoppeliaSim scene.
  - `deleteWalls()`
    - This function is for deleting the maze after the CoppeliaSim simulation ends.

---

3. Implementing Solution

(A) Setting up the Workspace

- Download the following zip file containing the above mentioned files. **Right-click** on the hyperlink and select **Save Link As...** option to download.
  - **Windows OS Users**:
    - **task_2b_generate_maze_windows.zip**
  - **Ubuntu OS Users**:
    - **task_2b_generate_maze_ubuntu.zip**

- **Macintosh OS Users**:
  - **task_2b_generate_maze_macintosh.zip**

NOTE: The browser might warn that the file can harm your PC, but it will not and you can safely download it.

- You will find the following files/folders in the zip file:
  - ***test_cases*** folder - it contains all the maze images you will use to test your solution
  - ***task_2b_scene.ttt*** file - CoppeliaSim scene file in which you will be generating the maze
  - ***task_2b.py*** file - you will implement your solution in this file
  - ***test_task_2b.exe*** OR ***test_task_2b*** file - you will test your solution using this executable fi
  - ***test_task_2b.csv*** file - this file will help in testing the solution
- **Extract** the downloaded zip file to a new directory.
- Copy `task_1b.py` files made by your team in **Task 1B** and paste in the extracted folder.

NOTE:

- The teams are supposed to edit `task_1b.py` made by team in **Task 1B**.
- These Python scripts are **not** included in the **zip file of Task 2B**.
- These Python scripts are already imported or called in `test_task_2b.exe` / `test_task_2b` .
- Do **NOT** change the names of these files.

- Make sure you add following files in the same directory where all the Python scripts of this **Task 2A** are present.
  - `sim.py`
  - `simConst.py`
  - `remoteApi.dll` (***for Windows***) OR `remoteApi.so` (***for Linux***) OR `remoteApi.dylib` (***for Macintosh***)
- Refer **Task 0 Test Setup** for further details.
- **After completing the above steps**, your folder should have the following files and folder:
  - Files:
    - `task_2b_scene.ttt`
    - `test_task_2b.exe` / `test_task_2b`
    - `test_task_2b.csv`

- `task_2b.py`
- `task_1b.py`
- `sim.py`
- `simConst.py`
- `remoteApi.dll` (***for Windows***) OR `remoteApi.so` (***for Linux***) OR `remoteApi.dylib` (***for Macintosh***)

- Folder:
  - `test_cases`

(B) Examine the CoppeliaSim Scene

- Open **task_2b_scene.ttt** in CoppeliaSim. The CoppeliaSim scene will resemble Figure 3.

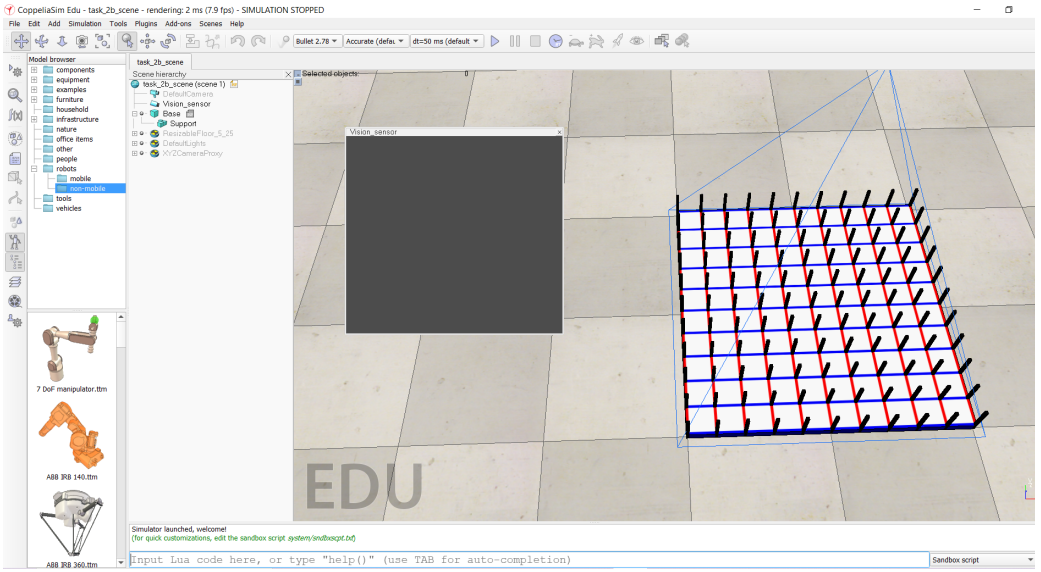

Figure 3: Task 2B CoppeliaSim scene (task_2b_scene.ttt)

- The scene has the following major elements:
  - **Base**
    - Platform on which maze is to be generated.
    - The dimensions are ***101cm x 101cm x 3cm***.
    - This scene object is configured as static
    - A CoppeliaSim customisation script is attached to the Base.
  - **Support**

- Acts as support structures between walls.
- There are total 121 structures in number
- Dimensions of each structure is ***1cm x 1cm x 10cm***.

- **Vision_sensor**
  - Captures the image of the maze
- **Floating View**
  - Displays the Vision Sensor output

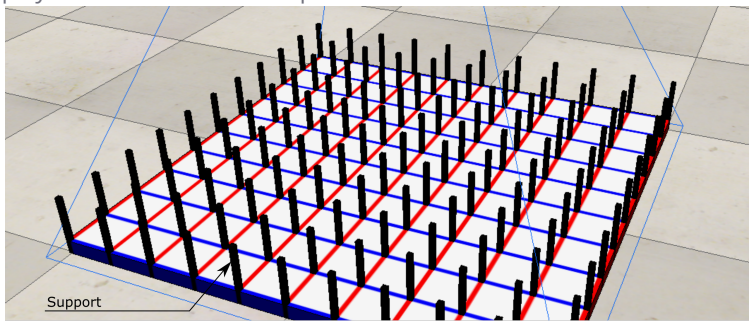

Figure 4: Base and Support

(C) Working with Customization Scripts in CoppeliaSim

- Customization scripts are embedded scripts that can be used to customize a simulation scene to a great extent. They are attached to (or associated with) scene objects in the scene hierarchy.
- The basic difference between customization scripts and child scripts is that customization scripts can be configured to execute when simulation is not running, while child scripts can be executed only when simulation is running.
- In this theme, we will be working with customization scripts mainly for setting up the arena in the starting.
- Open the Customization script attached to **Base** object in the scene.
- In this script, there are **three** major functions in which most of the processing takes place.
  - `sysCall_init()`
    - This function can be used to do some initialization.

- It is executed **once** after the customization script is reset.
- **You are NOT allowed to modify this function**.

```
function sysCall_init()

    if pcall(saveTexture) then
        print("Successfully saved texture")
    else
        print("Texture does not exist. Importing texture from file..")
        retrieveTexture()
        reapplyTexture()
    end
end
```

- `sysCall_beforeSimulation()`
  - This is executed **just before the simulation starts**.
  - In this task, we will be calling the functions associated with **creating the maze** that are called in this function.
  - **You are NOT allowed to modify this function. You can however pass your own arguments to `createMaze()` function**.

```
function sysCall_beforeSimulation()

    sim.setShapeTexture(baseHandle, -1, sim.texturemap_plane, 0, {1.01, 1

    generateHorizontalWalls()
    generateVerticalWalls()
    createMaze()
end
```

- `sysCall_afterSimulation()`
  - This is executed **after the simulation has ended**.
- In this task, we will be using this to **delete all the walls** of the maze after the simulation has ended.
  - **You are NOT allowed to modify this function**.

```
function sysCall_afterSimulation()

    deleteWalls()
    reapplyTexture()
end
```

(D) Generating the walls of the Maze

- We have incorporated a texture on the **Base** scene object to act as a guide for you to generate walls of the Maze (depicted in Figure 5).
- The ***blue-colored bands*** denote the place to generate the **horizontal walls** of the maze.
- The ***red-colored bands*** denote the place to generate the **vertical walls** of the maze.

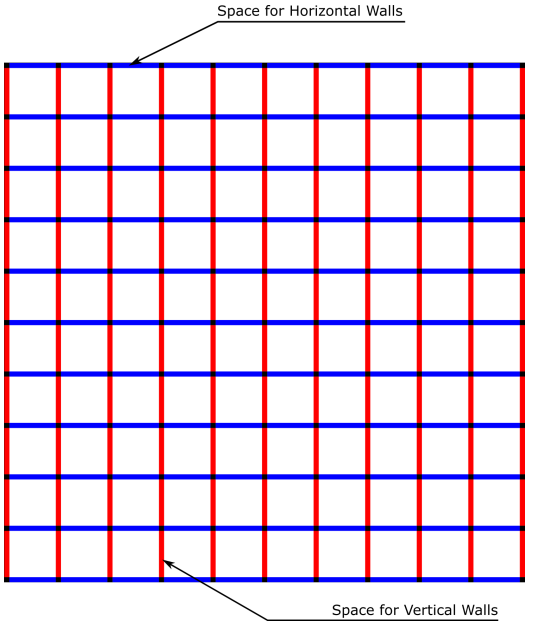

Figure 5: Base scene object Texture

- There are **two** functions pre-written in the customization script which ***you have to modify***.

1. `generateHorizontalWalls()`

Function Name	`generateHorizontalWalls()`

Function Name	`generateHorizontalWalls()`
Purpose	Generates all the Horizontal Walls in the scene.
Input Argument	None
Output Argument	None
Example Call	`generateHorizontalWalls()`

Table 1: generateHorizontalWalls()

- The output of this function is depicted in Figure 6.

Figure 6: Output of generateHorizontalWalls()

2. `generateVerticalWalls()`

Function Name	`generateVerticalWalls`
Purpose	Generates all the Vertical Walls in the scene.
Input Argument	None
Output Argument	None
Example Call	`generateVerticalWalls()`

Table 2: generateVerticalWalls()

- The output of this function is depicted in Figure 7.

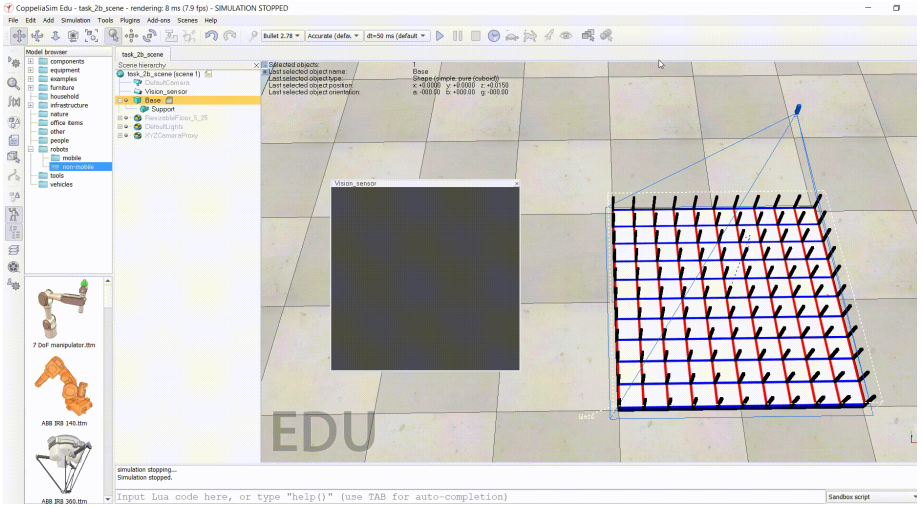

Figure 7: Output of generateVerticalWalls()

Note 1:

- We have included a `createWall()` helper function in the script which you can make use of while writing the above two functions.
- The `createWall()` can be used to create a single black colored wall.
- This function can be called iteratively to generate all the horizontal and vertical walls in the scene.

Note 2:

- The Horizontal and Vertical walls will need to be generated with respect to the position of the Base, hence they should be made child of Base when generated.

(E) Deleting the walls of the Maze

- There is **one** pre-written function `deleteWalls()` which ***you have to modify***.
- After simulation is stopped, the `deleteWalls()` function is called to delete all the walls preser in the scene (output depicted in Figure 6 and 7).

Function Name	`deleteWalls()`

Function Name	`deleteWalls()`
Purpose	Deletes all the walls in the scene.
Input Argument	None
Output Argument	None
Example Call	`deleteWalls()`

Table 3: deleteWalls()

(F) Combined Output

- The functions `generateHorizontalWalls()` and `generateVerticalWalls()` are called in `sysCall_beforeSimulation()` in the given customization script. This way, all the walls are generated in the scene before the simulation starts.
- `deleteWalls()` function is called in `sysCall_afterSimulation()` in the given customization script. This way, all the walls present in the scene are deleted after the simulation ends.
- The combined grid is depicted in Figure 8.

Figure 8 : Combined Grid

(G) Starting and Ending Remote API connection

- You have already implemented the `init_remote_api_server()` and `exit_remote_api_server()` functions in Task 2A.
- You can simply copy those functions in `task_2b.py` .

(H) Implementing Communication b/w CoppeliaSim server and Python Remote API client

- `send_data()` is written in Python file `task_2b.py`. It sends the encoded maze array to CoppeliaSim Remote API server.

Function	`send_data()`
Purpose	Sends data from Python client
Input Argument	`maze_array` : [***nested list***] Encoded Maze Array
Output Argument	`return_code` : [***integer***] the return code generated from the remote API
Example Call	`send_data()`

Table 4: send_data()

- `receiveData()` is written in Lua script. It receives the encoded maze array from Python Remote API client.

Function	`receiveData()`
Purpose	Sends data to CoppeliaSim server
Input Argument	`inInts` : [***table of Ints***] `inFloats` : [***table of Floats***] `inStrings` : [***table of Strings***] `inBuffer` : [***String***]
Output Argument	`inInts` : [***table of Ints***] `inFloats` : [***table of Floats***] `inStrings` : [***table of Strings***] `inBuffer` : [***String***]
Example Call	`receiveData()`

Table 5: receiveData()

- This **link** will help you implement the communication between server and client using `simxCallScriptFunction()`.

(I) Create Maze

- Till now, by implementing the functions for generating horizontal and vertical walls, we have been able to generate a **Grid** (***Remember grids in Task 1B ??***)

- Using the `createMaze()` function, you have to carve the **Maze** out of the **Grid** by deleting the walls of the grid wherever necessary.
- `createMaze()` function is pre-written in the Lua script and ***you have to modify it***.
- Unlike all the other functions discussed till now, you are allowed to change the input and output parameters for this function as you see fit.

Function	`createMaze`
Purpose	Carves the maze out of the Grid by deleting relevant walls
Input Argument	None, but you can define your input arguments for this function
Output Argument	None, but you can define your output arguments for this function
Example Call	`createMaze()`

Table 6: createMaze()

- The output of `createMaze()` is shown in Figure 9.

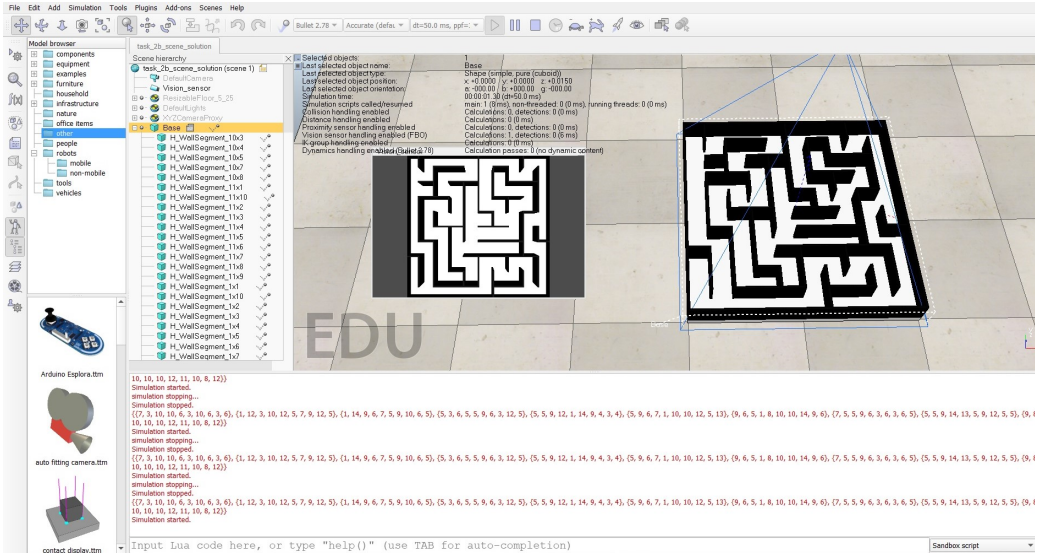

Figure 9: createMaze() Output

(J) Implementing Vision Sensor

- You have already implemented the `get_vision_sensor_image()` and `transform_vision_sensor_image()` functions in Task 2A.
- You can simply copy those functions in `task_2b.py`.

(K) Running your Solution

- To test and run your solution, do the following:

1. Open Anaconda Prompt or Terminal and navigate to the directory / folder on your system.
2. **Activate** the **Conda environment**.
3. Run the command: `python task_2b.py` to execute your solution. Your should get an output similar to Figure 10 below.

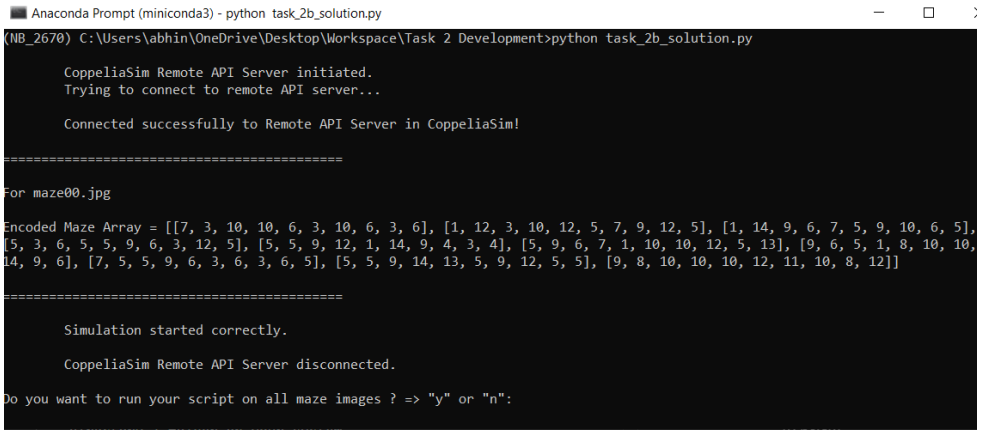

Figure 10: Output of task_2b.py

4. You can choose to run your script for rest **8 maze images** from **test_cases** folder by providing "**y**" as an input.
5. The output images (captured from Vision Sensor) will be saved in the ***generated_images*** folder. You may verify your solution by comparing the test cases with the result images.
6. Once you are satisfied with the solution, you may proceed with testing your solution.

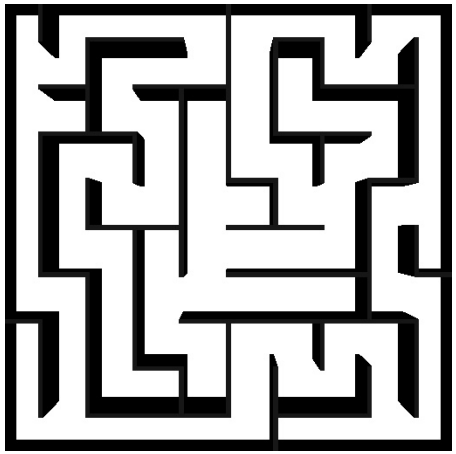

Figure 11 : Example Result Image

---

4. Testing the Solution

- You have to test your solution using the file `test_task_2b.exe` OR `test_task_2b` .
- This executable file will require `task_1b.py` , `task_2b.py` and `test_task_2b.csv` files to present in the same directory.
- In your workspace, activate the Conda environment.
    - **For Windows**:
        - Type the following command:
            - `test_task_2b.exe`
        - The code will start to run and it will sequentially generate maze in your CoppeliaSim scene. The output terminal should resemble Figure 12.

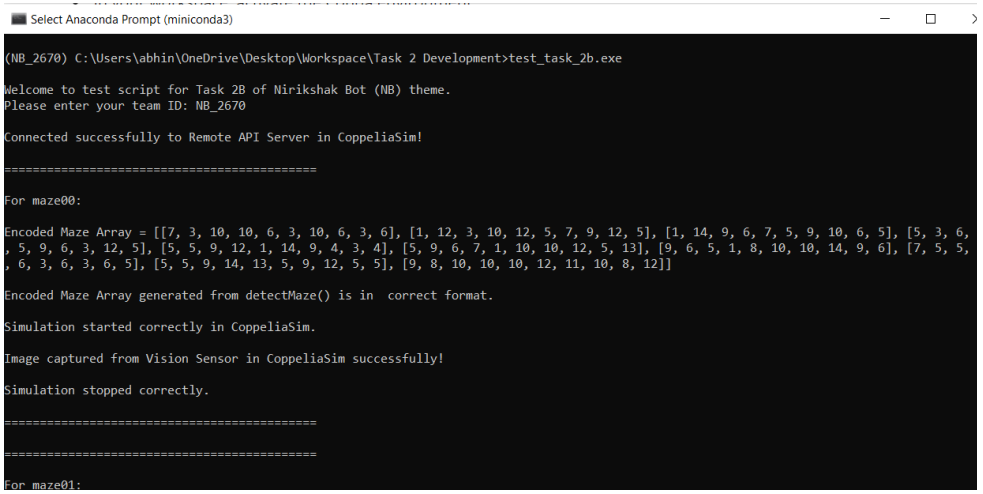

Figure 12 : **Output on Windows**

- For **Ubuntu**:
  - Type the following command:
    - `./test_task_2b`
  - The code will start to run and it will sequentially generate maze in your CoppeliaSim scene. The output terminal should resemble Figure 13.

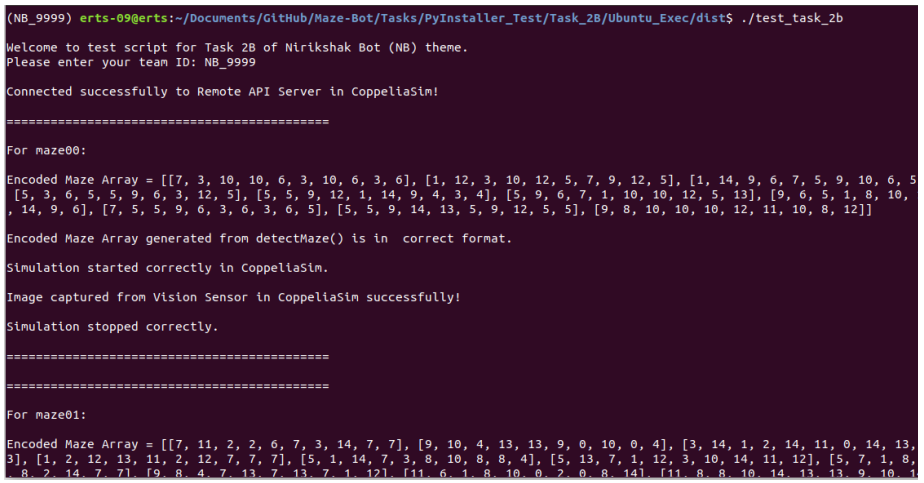

Figure 12 : **Output on Ubuntu**

- If the code ran successfully without any errors, then you will see `task_2b_output.csv` generated in your workspace.

Submission Instructions

For **Task_2B submission** you have to upload a **.zip** file. To create the appropriate file please follow instructions given below:

1. Create a new folder named `NB_<Team-ID>_Task_2B`.
    - For example: if your team ID is **9999** then you need to create a folder named `NB_9999_Task_2B`.
2. Create a zip file of / Compress just the `task_2b_output.csv` file and name it as `task_2b_output.zip`.

   NOTE: There should be **no separate folder** but **only ONE CSV file** i.e., `task_2b_output.csv` inside the zip file `task_2b_output.zip`.
3. Now copy and paste following files into this folder:
    - `task_1b.py` (modified in **Task 1B**)
    - `task_2b.py` (modified in **Task 2B**)
    - `task_2b_scene.ttt` (scene file you have modified in **Task 2B**)
    - `task_2b_output.zip` (zip file containing only one CSV file `task_2b_output.csv` generated after running `test_task_2b.exe` OR `test_task_2b`)
        - The `task_2b_output.zip` file has **only one file** `task_2b_output.csv` and **nothing else**.
    - `task_2b_script.lua` (**copy the CoppeliaSim customization script in a new file and save it as .lua extension**)

      The folder `NB_9999_Task_2B` and the zip file `task_2b_output.zip` should look similar to the Figure 13 and 14 respectively.

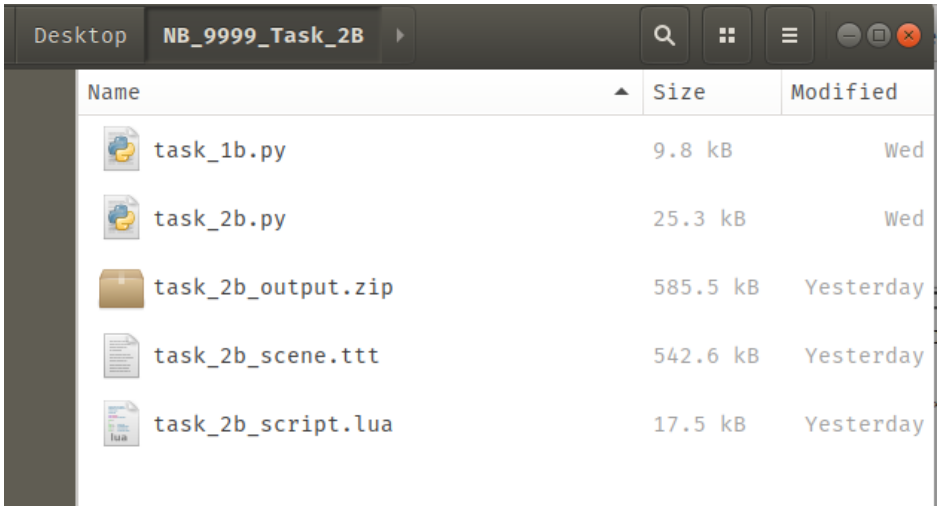

Figure 13: Contents of **NB_9999_Task_2B** folder

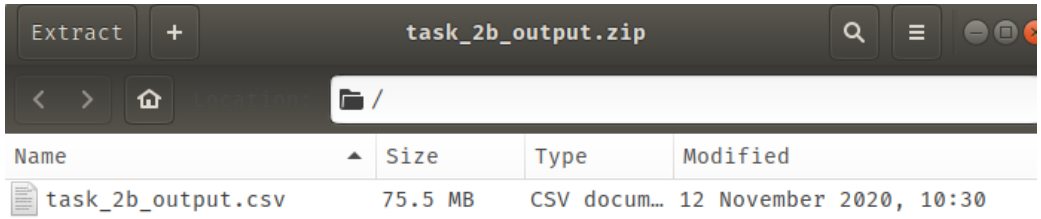

Figure 14: Contents of **task_2b_output.zip** file

4. Compress this folder `NB_9999_Task_2B` into a `NB_9999_Task_2B.zip` file.

5. Now go to the eYRC Portal and follow the instructions to upload this **.zip** file for **Task_2B** as shown in Figure 15.

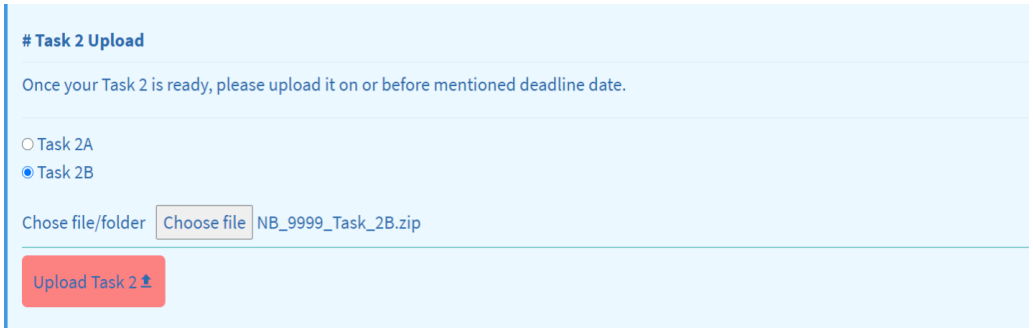

Figure 15: Submission of **NB_9999_Task_2B.zip** file on eYRC portal

> **NOTE:** File names mentioned are case sensitive. Verify all the file names before creating the zip file.

6. After you have **successfully submitted Task 2A and Task 2B files**, you can verify the zip file uploaded from the **'Verify Task 2 Upload'** section on the eYRC portal. The same is shown is Figure 16.

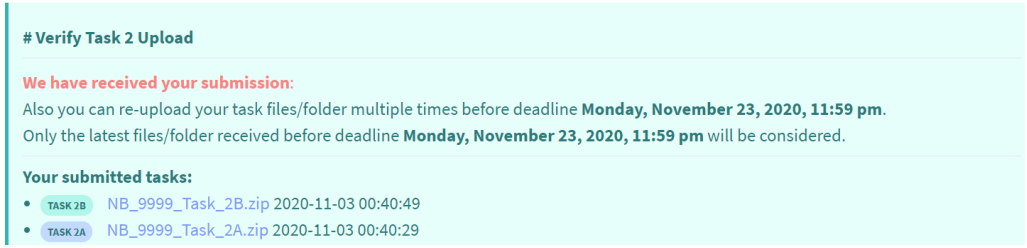

Figure 16: Successful submission of **Task 2A and Task 2B** on eYRC portal.

ALL THE BEST!!