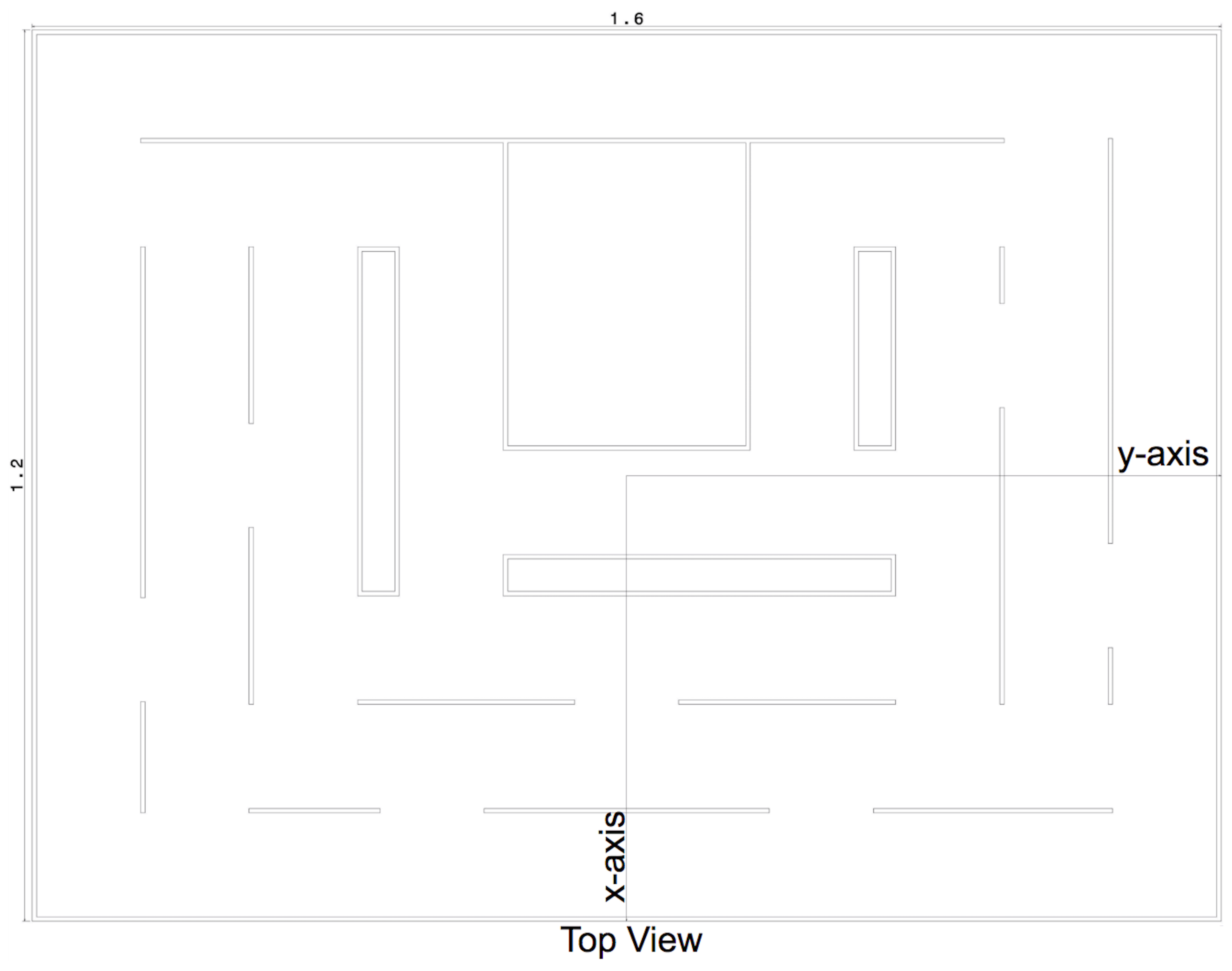
Project Specification

Typically, path planning algorithms are applied on *known* maps. However, in this project, you'll have to search almost blindly, which makes for a challenging problem. Here's what we know about the maze:



Only its width, and length!
In addition to the maze dimensions (1.2m x 1.6m), the *start and goal states* are provided as inputs to your algorithm. The *path toward the goal state* is your output. Inside a launch file, you'll edit the start pose and the goal pose of the object.

Since the map is unknown and the object is not equipped with any sensor, you might be wondering... how do we detect obstacles and move the cuboid object with the robotic arm?

We've included two services that you can call in your python script; `Move` and `CheckPath`:

- `Move`: Command the robot by sending a 2D pose. The robot will move the object on a linear path to this pose. Your moving point should be at least **5mm** away from the starting one. And if you are to rotate the arm, then you need to make sure you are rotating by at least **0.35** radians.
- `CheckPath`: Verify if a linear path between two 2D poses is valid.

Note: Even if you gain familiarity with the maze through repetition or by examining videos and logs, your algorithm must **not** be permitted to make use of this knowledge by hard-coding the map or learning from previous episodes. Each time the robot begins its maze problem, it should be completely ignorant of where the walls are or what the map of the maze looks like!

To summarize, you will have two main files to edit in this project: a launch file where you can specify the start/goal poses, and a python script to code your path planning algorithm.

There are 4 launch files in this project which you'll launch later to start the Gazebo and RVIZ simulation, and all the other nodes:

1. **moveit_planning_execution.launch**(rll_planning_project package): Starts the setup simulation in Gazebo and Rviz.
2. **planning_iface.launch**(rll_planning_project package): Starts the planning interface.
3. **path_planner.launch**(rll_planning_project package): Starts your path planning algorithm code.
4. **run_project.launch**(rll_project_runner package): Starts a single planning and path execution.

You'll have a chance to launch each of these files separately in different terminals or use a provided shell script file that will launch all the nodes in separate instances of xterm terminals. The first method is preferable for debugging since you can easily identify the errors.

NEXT

Getting Started

To get started, navigate to the project workspace concept. Under `home/workspace/catkin-ws/src`, you will see the project SDK(**rll_sdk**) and planning(**rll_planning_project**) packages. You will be mainly working with the planning package where you'll edit a launch file to change the object start/goal pose, and you'll edit a python script to code the path planning algorithm.

Python script: `path_planner.py`

Let's first take a look at the Python path planning script. Navigate to `/home/workspace/catkin_ws/src/rll_planning_project/scripts` and open the `path_planner.py` script. Here, you will find a sample solution where we first retrieve the input values, print them, call the `CheckPath` service and check if it's possible to move directly from the start pose toward the goal pose. If it's valid, the `Move` service is called to move the object linearly toward the goal pose.

Launch file: `planning_iface.launch`

Moving on, let's take a look at the launch file you'll need to modify. Navigate to `/home/workspace/catkin_ws/src/rll_planning_project/launch` and open `planning_iface.launch`. There's a long list of parameters needed for the project. You'll only need to edit the start 2D pose and goal 2D pose parameters.

Run the Sample Solution

It's time to run the project with the sample solution. **Enable the GPU, GO TO DESKTOP**, and follow these steps for a successful run:

1. Update/upgrade the system

```
apt-get update -y
apt-get upgrade -y
```

2. Install all the packages dependencies

```
cd /home/workspace/catkin_ws/
rosdep install --from-paths src --ignore-src -r -y
```

3. Build and source your `catkin_ws`

```
cd /home/workspace/catkin_ws/
catkin_make
source devel/setup.bash
```

4. Launch all the project nodes

```
# We've combined all the launch files in a `start_project.sh` script.
# Note: This script will only run if you are in the visual desktop, don't try to run it ins
cd /home/workspace/catkin_ws/src/rll_planning_project/scripts/
./start_project.sh
```

You'll need to repeat these steps after each reboot!

Stopping a Run

Hit Enter key in the main terminal to instantly kill the project execution and all the ROS nodes. You need to wait at least 30 seconds before starting a new run because the nodes will take some time to fully terminate. If you encountered an error, don't worry about it - just relaunch the nodes either manually or with the shell script file provided.

Video Review

Look closely at the video and try to identify the 4 cycles mentioned earlier:

1. First, the KUKA gripper will move to the object start position, orient itself with respect to the object, grab the object, and lift it up.

```
# We've combined all the launch files in a `start_project.sh` script.
# Note: This script will only run if you are in the visual desktop, don't try to run it ins
cd /home/workspace/catkin_ws/src/rll_planning_project/scripts/
./start_project.sh
```

You'll need to repeat these steps after each reboot!

Stopping a Run

Hit Enter key in the main terminal to instantly kill the project execution and all the ROS nodes. You need to wait at least 30 seconds before starting a new run because the nodes will take some time to fully terminate. If you encountered an error, don't worry about it - just relaunch the nodes either manually or with the shell script file provided.

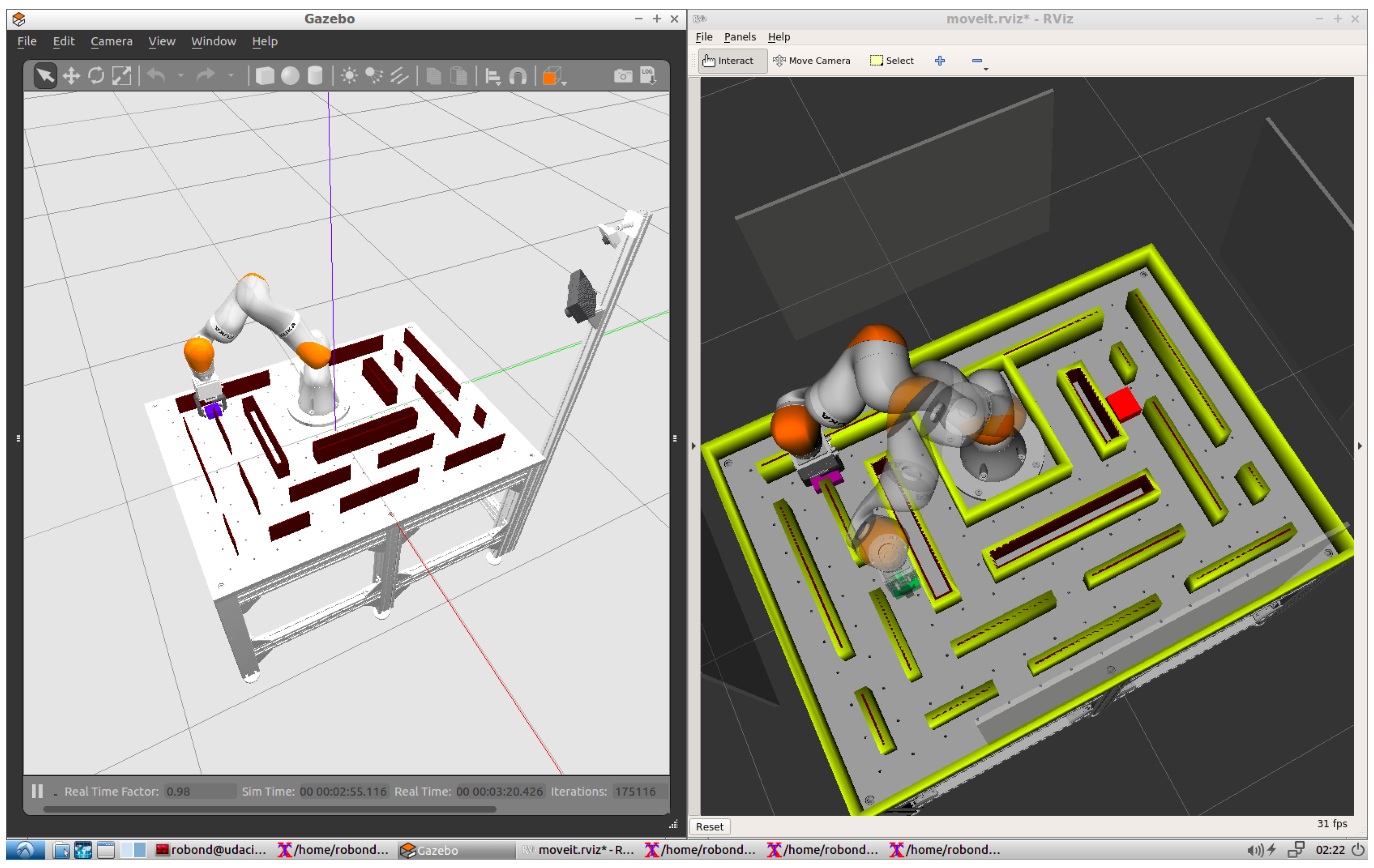
Video Review

Look closely at the video and try to identify the 4 cycles mentioned earlier:

1. First, the KUKA gripper will move to the object start position, orient itself with respect to the object, grab the object, and lift it up.

2. At this stage, your path planning code will be executed. Your code will search for a path and navigate the robot through the maze by commanding 2D positions, and an orientation angle, in order to get around corners in the maze
3. You will have a total time of 8 minutes to search for a path and move toward the goal pose. Once reached, your cuboid object will be placed in the goal pose.
4. Finally, the robot will lift the object from the goal pose, return it back to the start pose, drop it, and get ready for another run.

Gazebo and Rviz



Gazebo: Visual maze(red color) which is a replica of the real maze located at the KIT Lab in Germany.

Rviz: Collision maze(yellow color) which is the same as the visual maze but with thicker walls and some extra ones. Inside the collision maze, you can still see the visual maze that has thinner walls. The red block indicates the goal pose of the object.

As a safety measure, we added a tolerance value to the visual maze to prevent any collision between the object or robot and the maze walls in the real world.

NEXT

Scoring Criteria

SEND FEEDBACK

Scoring

After launching all the nodes in this project, the robot will go through the different cycles and then generate a score. Your score is simply the total time it took the robot to plan and execute a path from start to goal. The clock starts right after the object is lifted from the start pose and stops whenever your robot reaches the goal pose. On the hardware system, your code will be run three times, with the median time selected as your reported score.

Things to always keep in mind while planning and executing a path:

- Do not exceed 8 minutes for planning plus execution - this will result in a failed run.
- Do not send any invalid pose - this will result in a failure and your code will be immediately aborted!
- Always call the `Move` service before moving to a new pose.

Knowledge
Search project Q&A

Student Hub
Chat with peers and mentors

NEXT

Path Planning

Earlier, you launched the sample solution which executed a path from start to goal. Now, let's make this problem more challenging by changing the start and end pose values of the object.

Navigate to the `/home/workspace/catkin_ws/src/rll_planning_project/launch` directory, open the `planning_iface.launch` file, and change the start and 2D pose of the object. Try your code for different configurations of start and goal poses.

The `Contest Maze` is over and now we are back to the `Practice Phase Maze 1`.

`Practice Phase Maze 1` Start/Goal configuration:

```
<!-- Start 2D pose  -->
<arg name="start_pos_x" default="0.38" />
<arg name="start_pos_y" default="0.0" />
<arg name="start_pos_theta" default="0.0" />
<!-- Goal 2D pose  -->
<arg name="goal_pos_x" default="-0.37" />
<arg name="goal_pos_y" default="0.5" />
<arg name="goal_pos_theta" default="0.0" />
```

Now, code a path planning algorithm to search for a path and move the object from its new start pose to its new goal pose. At each launch, note your score down and try to improve your planner to be as fast as possible.

Pre-Submitting

Once you finish coding the path planning algorithm and succesfully testing it in simulation, you'll want to jump in and submit your code to the hardware. Before you do that, you need to test your code more than once! Actually, **three times** since your code will run **three times** on the hardware, with the median time selected as your reported time. To test your code three times in simulation, change the `run_three_times` argument value in the `planning_iface.launch` file to true:

```
<!-- call the path planner three times and take the median as duration -->
<arg name="run_three_times" default="true"/>
```

Then, run your code and check if your able to succesfully move the object from start to goal three times continuously. The reason why we're asking you to do that is to make sure that you accounted for multiple runs by resetting your global variables. After that, feel free to move on and submit your code.

NEXT

Submitting your project during the PRACTICE PHASE

Once you've gotten your project working in the simulator, you can submit it to run on the real KUKA arm! Once submitted, your "job" will be put in a queue at the KIT Robotics Learning Lab (RLL). Your job will first be run in a simulator, to make sure it can run successfully on the hardware.

There are only a few steps for your submission:

1. Manually fetch your user code and install it in your workspace
2. Enter `submit` on the command line in a terminal window
3. To check on your jobs, enter `check_jobs` on the command line in a terminal window.

Install your user code - the `jwt` file

The `jwt` file is your user code and must be available at `/home/workspace/JWT/jwt` in order to submit a project. You can download your user code by clicking here.
You may need to **navigate to the link a second time** after logging in for the file to automatically download. Once you've downloaded your `jwt`, copy the contents into: `/home/workspace/JWT/jwt`.

Submit your job

Your python script must reside at `/home/workspace/catkin_ws/rll_planning_project/scripts/path_planner.py`. This is the only file that will be uploaded. If the submission succeeded, you'll see the "Submission Complete" feedback. Note that this does not indicate any status of the code itself, only that the job was successfully submitted to the queue. You may only have one job at a time in the queue.

During the submission process, you must positively "opt in" if you wish to see your time and rank on the leaderboard.

Note: In case the submission process fails, refresh the workspace and try again.

Check on your job(s)

Once your job is submitted, it will be given a job number and placed in a queue at the KIT RLL. To check on your job, enter `check_jobs` on the command line. You'll receive a status update on all the jobs you've entered. Here is an example of two failed jobs and another sitting in the queue partially done:

```
{
    "5b61d7d3f7971f68ce977d43": {
        "job_data": null,
        "job_result": "sim failure",
        "job_status": "finished",
        "position": -1
    },
    "5b631f18f7971f14d02942c6": {
        "job_data": null,
        "job_result": "sim failure",
        "job_status": "finished",
        "position": -1
    },
    "5b634ca8f7971f14d02942c7": {
        "job_data": {
            "duration": 144.0544441403444594
        },
        "job_result": "sim success",
        "job_status": "waiting for real",
        "position": 2
    }
}
```

Note that the last job has a "duration" time associated with it. However, the job has not run on the "real" yet, so this is just the simulation run time. If the job runs successfully on the "real" arm, you should see an entry something like this:

```
 {   "5b634ca8f7971f14d0222222": {
        "job_data": {
            "duration": 99.123451234
        },
        "job_result": "real success",
        "job_status": "finished",
        "position": -1
    }
}
```

At this point, the score (time) will be entered in the leaderboard database. If you have opted in, your time and rank will be visible on the leaderboard under whatever name you've chosen to post with.

Submitting your project during the CONTEST PHASE

The same command, `submit`, should be used to submit your project. The submission system will operate a little differently during this phase:

- During submission, you will be presented with the Terms and Conditions for the contest and must positively agree to them to enter the contest.
- You will not receive any log links or video feedback until the contest is over, though you will receive job status with the `check_jobs` command.
- All jobs that pass the simulator will be added to a queue to run on the real KUKA arm. The real runs for each entry will only take place after the contest closes.

KUKA Project Hints!!!

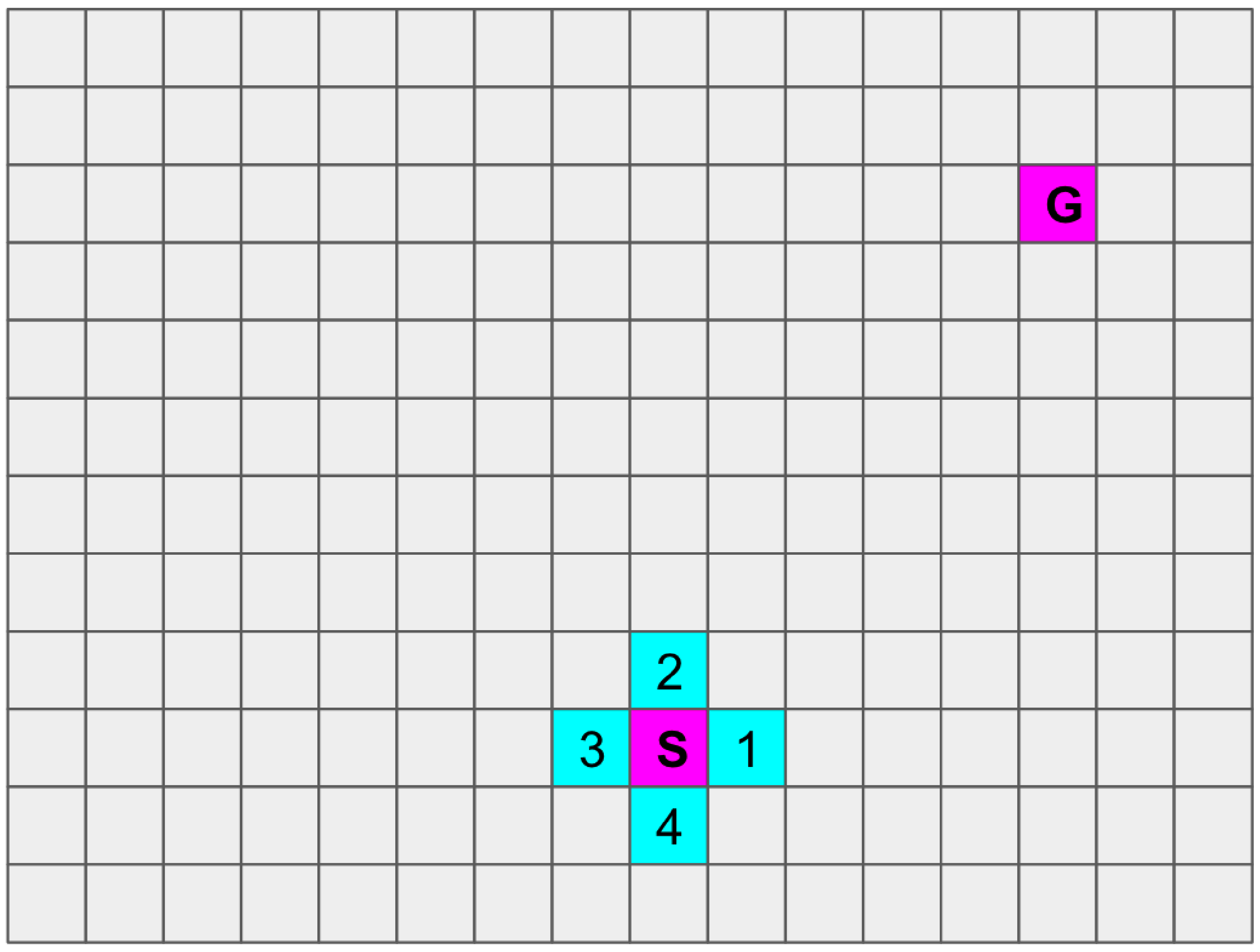
There are two main differences between classic path planning algorithms that you've learned earlier and the path planning algorithm that you'll have to write in the KUKA project

1. Classic path planning algorithms assume a known map where the position of obstacles and free spaces are known. However, in the KUKA project the maze is unknown! The only information you have is the map dimensions, start pose, and goal pose.
2. Classic path planning algorithms applied earlier assume a change in position only. The orientation angle of the robot is kept fixed. With the KUKA project, however, you need to change both the position of the gripper and its orientation to move anywhere inside the maze.

Applying what you learned earlier about classic path planning algorithms to the KUKA project

1. Since the maze is unknown, you have to implement "a search as you go" technique. Using the map dimensions you can represent the maze in a matrix form. Begin searching from the start pose, identifying if a cell is free or occupied, and as you're searching update the matrix until you reach the goal pose.
2. Define a set of angles that enables you to move anywhere inside the maze. To update the status (occupied or free) of a cell in the matrix, you can change the orientation angle of your gripper and check if you're able to move with each of the predefined angles towards the cell. The cell will be updated as free if any of the predefined angles returns a valid response. On the contrary, the cell will be updated as occupied if all the predefined angles return an invalid response.

Visual Example



This is an example of how you can represent `Maze 1` practice phase in a matrix form. As a reminder, this is `Maze 1` start and goal configuration:

```
<!-- Start 2D pose  -->
  <arg name="start_pos_x" default="0.38" />
  <arg name="start_pos_y" default="0.0" />
  <arg name="start_pos_theta" default="0.0" />
  <!-- Goal 2D pose  -->
  <arg name="goal_pos_x" default="-0.37" />
  <arg name="goal_pos_y" default="0.5" />
  <arg name="goal_pos_theta" default="0.0" />
```

- Transforming the maze **(1.2mx1.6m)** into a matrix size **(12x16)**. I am using a very small resolution here for demonstration. You need to test which resolution works best for you.
- Transforming `Maze 1` start and goal poses to matrix coordinates:
  - Start pose **(0.38, 0, 0)** to matrix coordinate **(9, 8, 0)**
  - Goal pose **(-0.37, 0.5, 0)** to matrix coordinate **(2, 13, 0)**

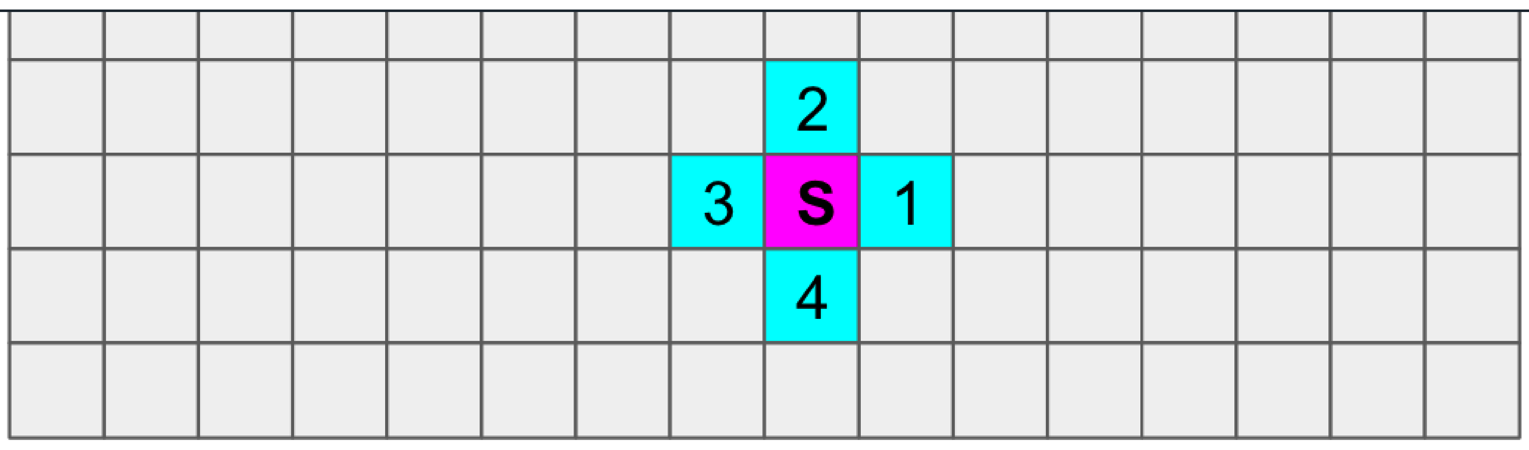
Now that you've transformed world coordinates into matrix ones, implement a **search as you go** technique where you start with the **S** cell and then update the status of adjacent cells.

Updating the status of adjacent **Cell 1**:

Defining a set of angles in radians to navigate the maze: **[0, 0.78, 1.57]**

- **(9, 8,0)** to **(9,9,0)** Valid or Invalid?
- **(9, 8,0)** to **(9,9,0.78)** Valid or Invalid
- **(9, 8,0)** to **(9,9,1.57)** Valid or Invalid?

Cell 1 will be considered as occupied if all these questions return an invalid answer!



This is an example of how you can represent `Maze 1` practice phase in a matrix form. As a reminder, this is `Maze 1` start and goal configuration:

```
<!-- Start 2D pose  -->
  <arg name="start_pos_x" default="0.38" />
  <arg name="start_pos_y" default="0.0" />
  <arg name="start_pos_theta" default="0.0" />
  <!-- Goal 2D pose  -->
  <arg name="goal_pos_x" default="-0.37" />
  <arg name="goal_pos_y" default="0.5" />
  <arg name="goal_pos_theta" default="0.0" />
```

- Transforming the maze **(1.2mx1.6m)** into a matrix size **(12x16)**. I am using a very small resolution here for demonstration. You need to test which resolution works best for you.
- Transforming `Maze 1` start and goal poses to matrix coordinates:
  - Start pose **(0.38, 0, 0)** to matrix coordinate **(9, 8, 0)**
  - Goal pose **(-0.37, 0.5, 0)** to matrix coordinate **(2, 13, 0)**

Now that you've transformed world coordinates into matrix ones, implement a **search as you go** technique where you start with the **S** cell and then update the status of adjacent cells.

Updating the status of adjacent **Cell 1**:

Defining a set of angles in radians to navigate the maze: **[0, 0.78, 1.57]**

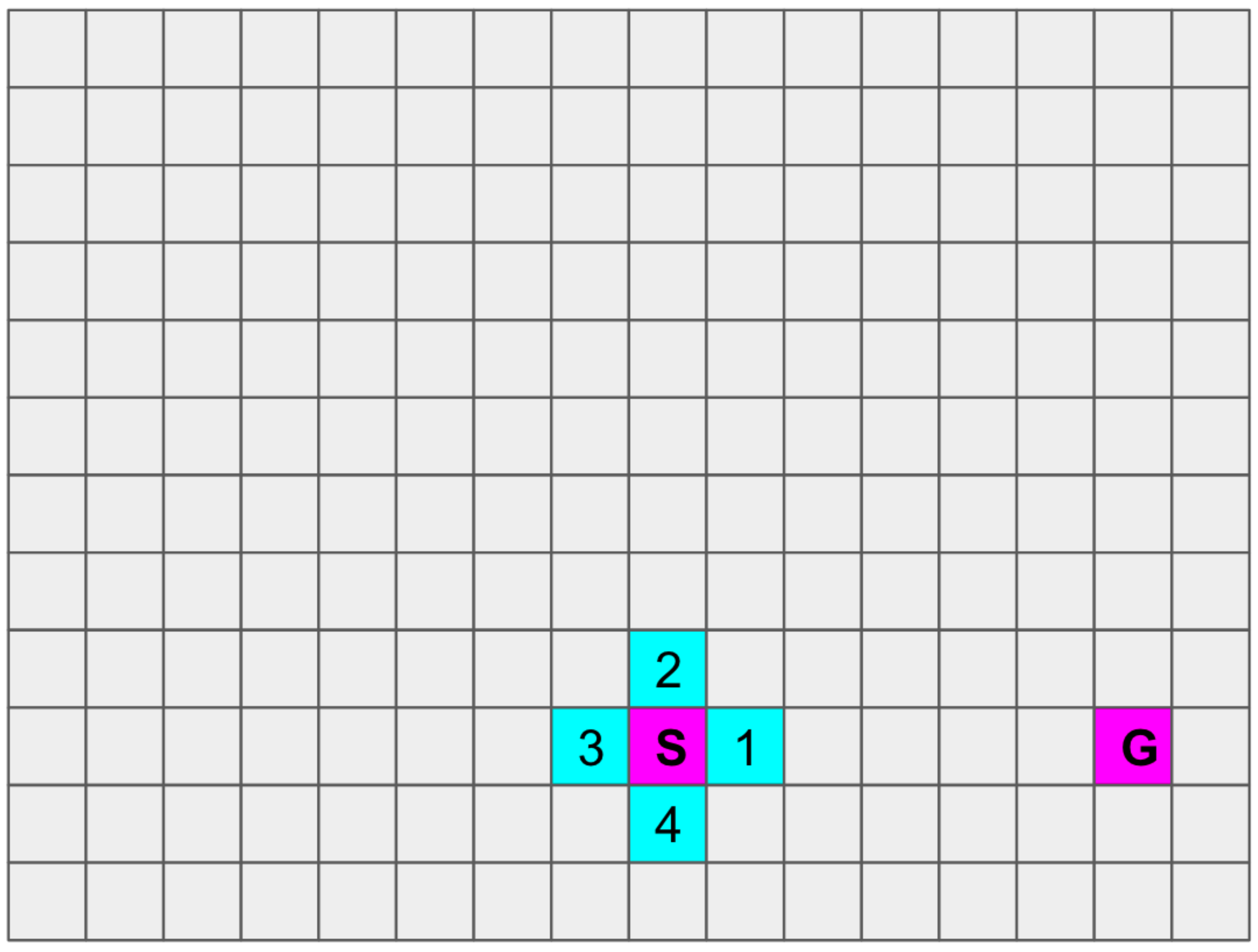
- **(9, 8,0)** to **(9,9,0)** Valid or Invalid?
- **(9, 8,0)** to **(9,9,0.78)** Valid or Invalid
- **(9, 8,0)** to **(9,9,1.57)** Valid or Invalid?

Cell 1 will be considered as occupied if all these questions return an invalid answer!

Now, move on to the second adjacent cell, check for movement with different angles, and update your matrix. Then, move to another undiscovered cell and so on until you build a map and reach the goal **G**.

How to start searching and planning?

As a good start, solve an easy problem where both the start pose and goal pose are on the same line. Search your way toward the goal by keeping the angle of your gripper constant. Once you solve this problem, assign a set of angles to each direction of movement as you've seen in the earlier visual example.



Discovering if cells **1 to 4** are occupied or free by keeping the gripper angle constant:

- Cell 1 - **(9, 8,0)** to **(9,9,0)** Valid or Invalid?
- Cell 2 - **(9, 8,0)** to **(8,8,0)** Valid or Invalid?
- Cell 3 - **(9, 8,0)** to **(9, 7,0)**Valid or Invalid?
- Cell 4 - **(9, 8,0)** to **(10,8,0)** Valid or Invalid?

NEXT