Robotics: Capstone
Week 1: Using the Rover Simulator

1 Introduction

Before implementing and testing on the actual rover, we will have you implement your algorithms on a simulator. In this document, we will show you how to use both the simulator class `RobotSim` and the interface class `RobotControl`.

2 Simulation Class

The simulation class is fully impemented, and thus you will not have to change anything in order to be able to use it. The class is also fully documented, so, for a more detailed overview of the class, refer to the documentation in the code. Here we will give a general overview of how the class works.

The class uses basic numerical integration techniques to simulate the robot moving. It accepts a map of the world in two parts, AprilTag markers and obstacles. Unlike the real world, obstacles will not obstruct your vision or motion, but if the robot hits an obstacle you should consider it a failure. As you have full access to the simulation class, you can technically access all the ground truth parameters when estimating position or moving the robot. This however will not be to your advantage when you move your code to the real robot, so it is suggested you only use the ground truth values if you need to debug your code.

2.1 Parameter Reading

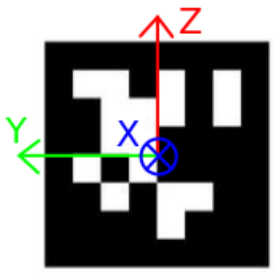
To get the parameters of the simulation and/or environment, you should use the `params.yaml` file. To load the parameters, we will be using the `yaml` package (in python code, `import yaml`). The code to load parameters is already implemented in the `RobotControl` class so you only need to change it if you want to make extra things. Here is some sample code to use the `yaml` package if you wish to modify anything:

```
f = open("params.yaml",'r')
params_raw = f.read()
f.close()
params = yaml.load(params_raw)
# Pseudocode for loading a value
val = params['val_name']
}
```

The parameters you need to load are:

1. `occupancy_map`: The `N` by `M` array of boolean values determining if that position has an obstacle. To convert a position `i,j` on this map to metric coordinates use `x_spacing` and `y_spacing`
2. `max_omega`: Cap on the maximum angular velocity for the robot
3. `max_vel`: Cap on the maximum linear velocity for the robot
4. `pos_init`: The 3 by 1 array for the starting position of the robot, as `(x,y,theta)`
5. `pos_goal`: The 3 by 1 array of the goal position for the robot, as `(x,y,theta)`

6. `world_map`: A `K` by 4 array of `(x,y,theta,id)` for the AprilTags in the environment, the x,y being the position in the world, the theta being the orientation in the world (angle between the world x axis and the tag x axis in radians) and id being the unique identifier. Note: The ids must be in order, as in 0,1,2,... not 1,3,... or 1,2,3,... etc. **Please note** that the x-axis points away from the AprilTag, as below.



7. `x_spacing`: The distance in meters between each grid in the occupancy map in the x direction
8. `y_spacing`: The distance in meters between each grid in the occupancy map in the y direction

If you wish to change the parameters of the simulation just change the `params.yaml` file.

2.2 Interface

There are many functions in the simulator class, but only a few you will actually use. The main functions you will have to interface with are:

1. `get_measurements()`: This takes in no arguments, and returns a 5 by 1 numpy array `(x,y,theta,id,time)`. The `x`, `y`, and `theta` give you the position and orientation of the robot with respect to the marker. The `id` field gives the unique ID of the April Tag seen. The `time` field gives the time the measurement was taken. If the camera does not have the marker in its field of view, it returns `None`.
2. `get_imu()`: This takes in no arguments and returns a 5 by 1 numpy vector `(acc_x,acc_y,acc_z,omega,time)`. The first three show the 3D acceleration of the robot, while the fourth is the `omega` to get the IMU's noisy estimate of angular velocity and `time` to determine when the measurement was taken. If no IMU measurement has come in yet, this returns a value of `None`.
3. `command_velocity(vx,wz)`: This takes in two arguments, `vx` and `wz`, and returns nothing. Once sent, it tells the robot to move forward at velocity `vx` meters per second and rotate at speed `wz` radians per second (although there will be some noise here also).
4. `set_est_state(est_state)`: This takes a 3 by 1 numpy vector (in the usual format `(x,y,theta)`) of your estimated state. This is for debugging purposes when implementing the Kalman Filter. It will visualize your estimated state in white next to the true state to see how good your filtered estimate is compared to the true value

The first three are the same functions you will use when you transition to the actual robot. The simulation will add noise to the system so your algorithms will need to be somewhat robust even in simulation.

3 Robot Control Class

The `RobotControl` class will be the main interface to control the robot. Here we outline the interface to the class and how to use it.

3.1 Interface

There are only two functions you must implement:

1. `__init__`: this function has many inputs which are simply the loaded parameters of the YAML file. We have already written here the code to start the simulator. This is also where you will set up things like the Kalman Filter class or the Differential Drive controller class.

2. `process_measurements()`: This is the main loop of the rover both in simulation and on the actual robot. Here you should check for measurements and use them appropriately.

3.2 Running the Simulation

To test the simulator, To run the simulation use the following command:

```
python RobotControl.py
```

(Note that the command for your operating system might differ, please refer to this document: `http://pythoncentral.io/execute-python-script-file-shell/`). This should start the simulation and run anything you have in the `process_measurements` function. By default, the robot will move forwards at 0.3 m/s. If the program crashes, please check that you have installed all the libraries and are interfacing with the functions correctly. In particular, you will need:

- matplotlib
- numpy
- yaml