Behaviour-based Map Building Robotic System

CSCK505: Robotics
Charbel Abou Khalil, Muhammad Ali & Tuhin Banerjee

This report details the development of a 2D occupancy grid map for an environment containing obstacles using both inverse sensor model and wandering behaviour. The primary goal is to minimise map construction time. The code of all experiments carried out to achieve the primary goal will be analysed and explained and overall performance of designed robotic systems discussed. Furthermore, possible improvements will also be highlighted to provide a comprehensive critique.

1. Introduction

In recent years, substantial progress has been made in robot navigation in dynamic environments; robots are being preferred in industry over use of AGVs (Automated Guided Vehicles) due their flexible and efficient ability to navigate in changing environments (Kruusmaa & Svensson, 1998). Both (Hernandez, et al., 2009) and (Kaufman, et al., 2016) argue that inverse sensor model is widely used for mapping applications. The inverse sensor model is defined as the probability of occupancy for a grid cell based on range measurements and the configuration of the mobile robot as it plays a crucial role in occupancy grid mapping (Kaufman, et al., 2016). (Kaufman, et al., 2016) highlights that exact solution to the inverse sensor model has not been utilised in occupancy grid mapping due to cumbersome computational requirements.

(Peng, et al., 2015) argues that obstacle avoidance ability is crucial for ground mobile robots such as Pioneer p3dx used in this project. A simple obstacle avoidance algorithm is employed to enable the wandering behaviour in p3dx; the robot will check for obstacles in the front and move if no obstacles are detected, similarly the robot will check for obstacles at the right and move or stop according to presence or absence of obstacles. Obstacle avoidance algorithms are divided in to two categories: model-based and sensor-based; authors of this report employ a sensor-based algorithm, obstacles are represented by cells on a map where the value in the cell determines whether there exists an obstacle or not (Peng, et al., 2015).

2. Software Configuration

In this project, CoppeliaSim (https://www.coppeliarobotics.com/downloads) and MATLAB (https://www.mathworks.com/products/matlab.html) are employed for environment construction and simulation. CoppeliaSim's MATLAB API is used to establish a remote link for simulation. The project brief instructs on constructing a 5m x 10m environment with wall boundaries and obstacles as shown in figure 1.

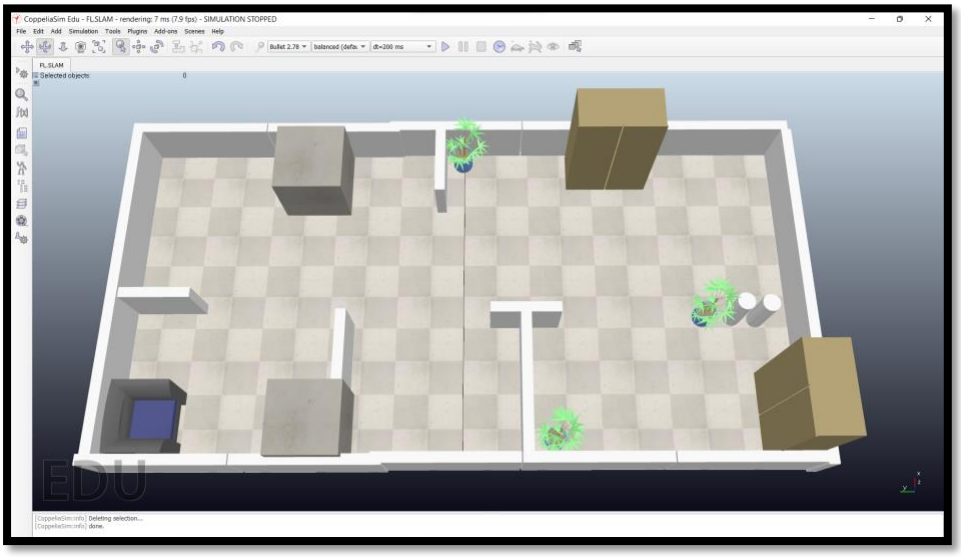

Figure 1: Scene containing wall and obstacles in CoppeliaSim.

Once CoppeliaSim is launched, under Model browser tab, Infrastructure-> Floors is selected. Under the list of floors available, resizable floor 5-25 meters.ttm is chosen and placed on the scene; for obstacles and walls, Model Browser catalog is searched to create boundary walls with obstacles like blocks, plants, walls and placed on the scene (figure 1).

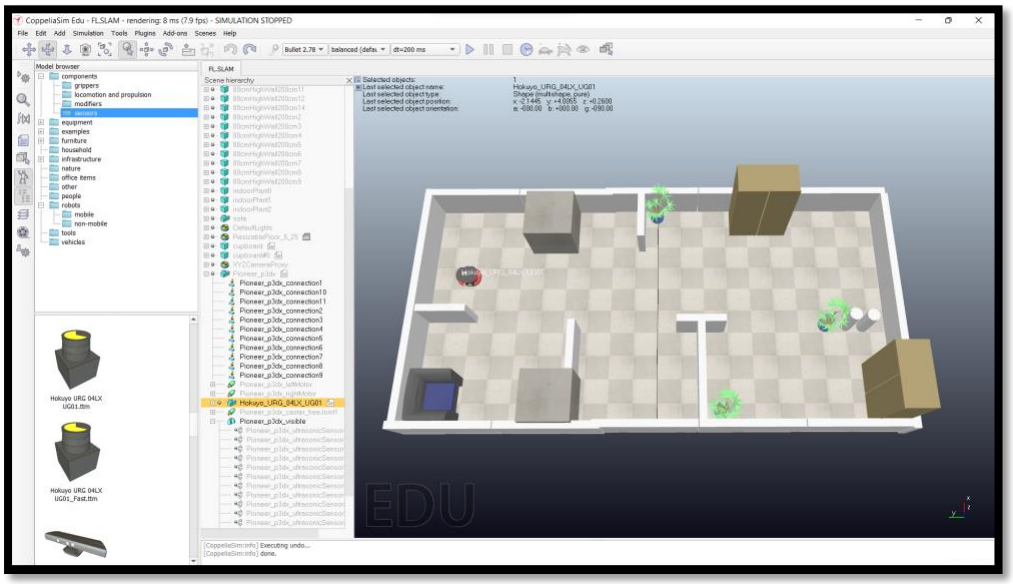

Figure 2: Pioneer p3dx robot coupled with Hokuyo_URG_UG01 sensor.

Under the Model browser tab, robots->mobile subtab is selected to access the Pioneer p3dx robot; to select and attach a 2D Lidar sensor, sensor subtab under Select components tab is accessed to place Hokuyo_URG_04LX_UG01 over the p3dx robot as shown in figure 2.

```
if #points>0 then
    -- sim.tubeWrite(commTube,sim.packFloatTable(points))
    -- laser point cloud data
    pointsData=sim.packFloatTable(points)
    sim.setStringSignal("laserRangeFinderData",pointsData)
    -- laser origin data
    laserOriginData=sim.packFloatTable(laserOrigin)
    sim.setStringSignal("laserOriginData",laserOriginData)
    -- laser orientation data (euler angles)
    laserOrientationData=sim.packFloatTable(laserOrientation)
    sim.setStringSignal("laserOrientationData",laserOrientationData)
end
```

Figure 3: Laserscan child script.

Hokuyo_URG_04LX_UG01 child script is edited to access the laserscan data from CoppeliaSim. The snippet of codes (figure 3) is included at the end of the *function sysCall_sensing()* to send the points acquired from the laser scanner to matlab as a string signal.

As per the project specifications, two experiments are carried out; one for mapping using teleoperation and one for autonomous mapping. The instructions to execute both programs are as follows: For mapping using teleoperation, open **Teleop_Map.m** file and run it using f5 or select run icon under editor menu. For Autonomous mapping, open **wandermap.m** file and run it using f5 or select run icon. Furthermore, use arrow keys for moving robot in Teleop_Map.m file and long press the key “m” to move the robot autonomously in wandermap.m file.

Experiment 1: Analysis of mapping using teleoperation.

Initially the current workspace is cleared including local and global variables then floor boundary is defined along with creation of binary occupancy map object with required floor dimensions (figure 4).

```
X_BOUND = 5;
Y_BOUND = 10;
currOccMap = binaryOccupancyMap(5,10,16);
occMapFigure = figure;
```

Figure 4: Environment parameters.

RemoteAPI plugin is employed to establish connection between CoppeliaSim as seen in figure 5.

```
csim = remApi('remoteApi');
csim.simxFinish(-1); % just in case, close all opened connections
clientID = csim.simxStart('192.168.0.122',19999,true,true,5000,5);
```

Figure 5: RemoteAPI parameters and functions.

Once the remote connection configuration is finalised, sensor and data type are defined as shown in figure 6 where CsimLidarSensor is the file where datatype structure is defined.

```
sensor = CsimLidarSensor;
sensor.MaxRange = 4.095; % in metres
sensor.MinRange = 0.06; % in metres
sensor.FieldOfView = 240 * pi/180; % 240 degrees in radians
sensor.PointCloudDataPointer = 'laserRangeFinderData';
sensor.LaserOriginDataPointer = 'laserOriginData';
sensor.LaserOrientationDataPointer = 'laserOrientationData';
```

Figure 6: Parameters of sensor.

Another imperative step at this stage is to specify the wheel speed and check the remote connection via the ClientID function. Furthermore, by using simGetObjectHandle, the attributes of left and right motor can be obtained. The initial speed of left and right motor is set by employing simSetJointTargetVelocity function (figure 7).

```
[~,leftMotor]=csim.simxGetObjectHandle(clientID,'Pioneer_p3dx_leftMotor',csim.
simx_opmode_blocking);
[~,rightMotor]=csim.simxGetObjectHandle(clientID,'Pioneer_p3dx_rightMotor',csi
m.simx_opmode_blocking);
opMode = csim.simx_opmode_blocking;
csim.simxSetJointTargetVelocity(clientID,leftMotor,0,opMode);
csim.simxSetJointTargetVelocity(clientID,rightMotor,0,opMode);
```

Figure 7: Robot left and right motors parameters.

Initially userInput is set to 0 to check whether the robot is in motion; if its moving then, lidarFillAccMap function (figure 8) is called to get the data from the lidar sensor for map file generation.

```
userInput = 0;
while userInput ~=27
    currOccMap = lidarFillOccMap(currOccMap,csim,clientID,sensor, ...
        X_BOUND, Y_BOUND, X_BIAS, Y_BIAS);
    figure(occMapFigure);
    show(currOccMap);
    userInput =
moveRobotFromUserInput(csim,clientID,leftMotor,rightMotor,WHEEL_SPEED);
    disp(userInput);
end
```

Figure 8: Details of lidarFillAccMap function.

To understand the occupancy grid construction and update process along with incorporation of latest sensor data, it's imperative to highlight lidarFillOccMap function in detail.

```
function currOccMap = lidarFillOccMap(currOccMap, csim, clientID, sensor, maxXPos, maxYPos,
xAxisCorrectionFactor, yAxisCorrectionFactor)
  arguments
    currOccMap % binaryOccupancyMap obj
    csim % CoppeliaSim remote api obj
    clientID % client id of the established MATLAB -> CSIM connection
    sensor CsimLidarSensor
    maxXPos double
    maxYPos double
    xAxisCorrectionFactor double
    yAxisCorrectionFactor double
  end
```

Figure 9: Details of function updating occupancy grid.

In figure 9, currOccMap function takes old occupancy grid map as input and updates it with latest sensor data.

```
  if ~sensor.Initialised
    % CoppeliaSim recommends first call to be simx_opmode_streaming and
    % subsequent calls as simx_opmode_buffer for simxGetStringSignal remote
    % api
    [~,~] =
csim.simxGetStringSignal(clientID,sensor.PointCloudDataPointer,csim.simx_opmode_streaming);
    [~,~] =
csim.simxGetStringSignal(clientID,sensor.LaserOriginDataPointer,csim.simx_opmode_streaming);
    [~,~] =
csim.simxGetStringSignal(clientID,sensor.LaserOrientationDataPointer,csim.simx_opmode_streaming);

    sensor.Initialised = 1;
  end
```

Figure 10: Initialising sensor signal streaming.

Once the sensor signalling is initialised, pointcloud data, sensor origin and orientation is collected from Coppeliasim; the data received is a vector of 1 x 2052 whereas the datapoints count is 684 each with 3 values for x,y,z. Figure 11 shows conversion of 2052 vector in to a 3 x684 matrix.

```
if(csim.simx_return_ok == respCodePointCloud && ...
      csim.simx_return_ok == respCodeOrigin && ...
      csim.simx_return_ok == respCodeOrient)

    pointCloudVec = double(csim.simxUnpackFloats(pointCloud));

    hokuyoOrigin = double(csim.simxUnpackFloats(laserSensorOrigin));

    hokuyoOrient = double(csim.simxUnpackFloats(laserSensorOrientation));

    cartesianMat = reshape(pointCloudVec,3,[]);
```

Figure 11: Vector conversion in to required matrix.

After the map is generated, simulation is cancelled by calling simFinish function as seen in figure 12.

```
csim.simxFinish(clientID);
else
    disp('Error connecting to CoppeliaSim');
end
```

Figure 12: Simulation termination function.

Experiment 2: Analysis of Autonomous mapping.

Wandermap.m file contains the code for autonomous mapping implementation; there are a number of functions which are similar to the previous experiment. In this code review we will focus on the new functions pertinent to autonomous control such as moveRobotFromProximity instead of moveRobotFromUserInput. The former function is utilised to get the data from all four proximity sensors in order to navigate around obstacles. Initially the attributes of left and right motor are obtained by using simGetObjectHandle function from the CoppeliaSim then speed of left and right motor is initialised as 0 (figure 13).

```
[~,leftMotor]=csim.simxGetObjectHandle(clientID,'Pioneer_p3dx_leftMotor',csim.
simx_opmode_blocking);
[~,rightMotor]=csim.simxGetObjectHandle(clientID,'Pioneer_p3dx_rightMotor',csi
m.simx_opmode_blocking);
[res,objs]=sim.simxGetObjects(clientID,sim.sim_handle_all,sim.simx_opmode_bloc
king);
[res, leftMotor] =
sim.simxGetObjectHandle(clientID,'Pioneer_p3dx_leftMotor',sim.simx_opmode_bloc
king);
[res, rightMotor] =
sim.simxGetObjectHandle(clientID,'Pioneer_p3dx_rightMotor',sim.simx_opmode_blo
cking);
sim.simxSetJointTargetVelocity(clientID,leftMotor,lSpeed,sim.simx_opmode_block
ing);
sim.simxSetJointTargetVelocity(clientID,rightMotor,rSpeed,sim.simx_opmode_bloc
king);
```

Figure 13: Attributes of left and right motor.

Figure 14 shows the code for obtaining proximity sensor data from CoppeliaSim, errors are also effectively handled.

```
[errorCode, fLeft] = sim.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor4',sim.simx_opmode_oneshot_wait);
[errorCode, fRight] = sim.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor5',sim.simx_opmode_oneshot_wait);
[errorCode, sLeft] = sim.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor2',sim.simx_opmode_oneshot_wait);
[errorCode, sRight] = sim.simxGetObjectHandle(clientID,
'Pioneer_p3dx_ultrasonicSensor7',sim.simx_opmode_oneshot_wait);
```

Figure 14: Fetching data from Pioneer's proximity sensors.

The obstacles detected by proximity sensors are stored as coordinates in detectionPoints variable as seen in figure 15.

```
[errorCode, detectionState1, detectedPoint1, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, fLeft,
sim.simx_opmode_streaming);
[errorCode, detectionState1, detectedPoint2, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, fRight,
sim.simx_opmode_streaming);
```

Figure 15: Obstacle handling in detail.

In moveRobotFromProximity function, obstacle position is obtained via the proximity sensors and every time this function is called, the old position is replaced by the new one thus, allowing dynamic fetching of obstacle position.

```
function userInput = moveRobotFromProximity(sim,clientID,...
    leftMotor,rightMotor,speed,fLeft,fRight,sLeft,sRight)
    [errorCode, detectionState1, detectedPoint1, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, fLeft,
sim.simx_opmode_streaming);
    [errorCode, detectionState1, detectedPoint2, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, fRight,
sim.simx_opmode_streaming);
    [errorCode, detectionState1, detectedPoint3, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, sLeft,
sim.simx_opmode_streaming);
    [errorCode, detectionState1, detectedPoint4, detectedObjectHandle,
detectedSurfaceNormalVector] = sim.simxReadProximitySensor(clientID, sRight,
sim.simx_opmode_streaming);
```

Figure 16: Dynamically handling obstacle position.

Data from keyboard is fetched via *get(gcf,'CurrentCharacter')* command. Autonomous mapping is enabled once ‘m’ is pressed. The obstacle distance is computed by using data points (obstacle coordinates) obtained from all proximity sensors. Based on obstacle distance, speed of left and right motor is adjusted accordingly to achieve desired motion (figure 17).

```
if (rDist < 0.00000001)
    if(detectedPoint2(2)<  0.00000001)
        lSpeed = -1;
        rSpeed = 1;
    else
        lSpeed = 0;
        rSpeed = 1;
    end
end
if (lDist < 0.00000001)
    if(detectedPoint1(2)<  0.00000001)
        lSpeed = 1;
        rSpeed = -1;
    else
        lSpeed = 1;
        rSpeed = 0;
    end
end
```

Figure 17: Code to enable turning or rotation based on obstacle position and proximity sensors.

Similarly, linear speed is computed once ideal value is assigned using simSetJointVelocity function as seen in figure 18.

```
if (fDist > 0.00000001)
    lSpeed = 2;
    rSpeed = 2;
end
if ( (fDist < 0.00000001)& (rDist < 0.00000001) & (lDist < 0.00000001))
    lSpeed = -2;
    rSpeed = 2;
end
```

Figure 18: Computation of linear speed of Pioneer.

If the “s” key is pressed, the figure is saved and simulation gets terminated with the aid of sim.simxStopSimulation() function as seen in figure 19.

```
    elseif (val == 's')

            saveas(gcf,'map_w.bmp');
             disp('Saving Map');
            pause(0.5);

            disp('Closing....!!!!!!!!!!!!');
            sim.simxStopSimulation(clientID,sim.simx_opmode_oneshot);
            clc;
            close();
```

Figure 19: Simulation termination code.

3. Future Work

During the literature review, many simple and complex obstacle avoidance algorithms were used in various projects which seem to significantly alter overall performance and efficiency of the mobile robot (Susnea, et al., 2010). The authors propose an extension of this project by experimenting with Potential Field, Vector Field Histogram algorithms along with Bubble Band techniques to compare time taken for map generation and observe obstacle avoidance efficiency of each algorithm. A comparative study of environments with varying degree of obstacles versus overall efficiency of previously mentioned algorithms is also proposed. Furthermore, integration of ROS (Robot Operating System) with Lidar (GenerationRobots, 2018) can serve as an alternative configuration for future experiments.

4. Conclusion

The autonomous mapping was found to relatively perform better in terms of covering maximum area in less time than mapping through teleoperation. Real life use cases of this project might involve map creation in difficult terrains where sending in humans is both dangerous and uneconomical. Furthermore, real time mapping can aid police and first responders in approaching volatile areas prone to crime and post disaster/accident zones respectively.

References:

Hernandez, E., Ridao, P., Mallios, A. & Carreras, M., 2009. *Occupancy Grid Mapping in anUnderwater Structured Environment.* Guaruja, Proceedings of the 8th IFAC International Conference onManoeuvring and Control of Marine Craft.

Kaufman, E., Lee, T., Ai, Z. & Moskowitz, I. S., 2016. *Bayesian Occupancy Grid Mapping via an Exact Inverse Sensor Model.* Boston, 2016 American Control Conference (ACC).

Kruusmaa, M. & Svensson, B., 1998. *Combined Map-Based and Case-Based Path Planning for Mobile Robot Navigation.* Castellón, Spain, Tasks and Methods in Applied Artificial Intelligence, 11th International Conference on Industrial and Engineering Applications of Artificial In telligence and Expert Systems.

Norris, D. J., 2017. Behavior-Based Robotics. In: *Beginning artificial intelligence with the Raspberry Pi.* s.l.:Apress.

Peng, Y. et al., 2015. *The Obstacle Detection and Obstacle Avoidance Algorithm Based on 2-D Lidar.* Lijiang, China, Proceeding of the 2015 IEEE International Conference on Information and Automation.

Susnea, I. et al., 2010. *The Bubble Rebound Obstacle Avoidance Algorithm for Mobile Robots.* Xiamen, 2010 8th IEEE International Conference on Control and Automation.

GenerationRobots, 2018. *LiDAR integration with ROS: quickstart guide and projects ideas.* [Online]
Available at: https://www.generationrobots.com/blog/en/lidar-integration-with-ros-quickstart-guide-and-projects-ideas/
[Accessed 20 December 2021].