EECE 5698 Special Topics: Mobile Robotics Fall 2015 Final Project: Autonomous Robot Navigation

Professor: Taskin Padir

Team "Mikey": Matthew Teetshorn, Benjamin Gowaski, Timothy Rupprecht, Jicheng Li, Chander Mohan

December 15th, 2015

Introduction:

This project introduces TurtleBots, Robotics Operating System (ROS), MATLAB Robotic Systems Toolbox, and fundamental mobile robotic principles to achieve a set of goals. The project aims to program a mobile robot, being the TurtleBot, which can navigate in a known global map autonomously while avoiding any unknow obstacles within that environment and updating its local map. To do this we use skills such as map generation using robotic scanning and positioning outputs, as well as path planning algorithms, and obstacle detection and avoidance. All of these features have to be combined to complete a specific set of tasks outlined by the objectives and requirements sections below.

Objectives:

Upon successful completion of this lab, you will be able to:

1. program a mobile robot for waypoint navigation.
2. program a mobile robot for mapping its environment.
3. program a mobile robot for obstacle avoidance.
4. program a mobile robot for path planning in a (dynamic) world map.

Requirements:

1. Robot must operate fully autonomously within the experimental environment setup to be provided.
2. Robot must effectively avoid obstacles and walls within its environment.
3. Robot must generate a local map of its environment.
4. Robot must generate a global map of its workspace.
5. Robot must be capable of planning an admissible trajectory to move within its workspace.
6. Robot must be capable of estimating its position and orientation in world coordinates.
7. Robot must be capable of navigating to a given waypoint. The waypoint will be considered reached when any part of the robot covers the waypoint marker. The waypoints will remain stationary within robot's workspace and will be clearly marked. Robot must navigate at least 4 out of 5 waypoints.
8. Robot must return to the starting point (base) by planning a path that will cover all the open areas in the map generated. The robot must update the world map while returning to the base. Note that the location of the obstacles on the return trip may change and the robot must adjust the map and plan accordingly.

Results & Analysis:

Initial setup and interfacing:

To begin, the TurtleBot base and computer had to be manually powered on. Once connected to the NUwave wireless network, the robot could be accessed through ssh protocol via its unique IP address (Mikey: 10.101.84.35). The Robot Operating System (ROS) was then launched from the linux terminal shell. The MATLAB Robotic Systems Toolbox (RST) was then launched and used as the primary interface between the user and the generated higher level code to the ROS master node running on the Turtlebot. The ROS framework and the accompanying RST framework allow for a robotic control methodology by which robot commands and sensor data are moved around via a message passing interface. Messages from ROS are implemented with a publisher/subscriber model and internode communications are coordinated by the master node. A basic setup script was created in order to launch the RST interface to the assigned Turtlebot whose screenshot is attached ahead:

```
1  function [robot, velmsg, laser] = setup(ROS_IP_ADDRESS, MY_IP_ADDRESS)
2  %[robot, velmsg, laser] = setup('10.101.84.35','10.102.155.225');
3  setenv('ROS_MASTER_URI',strcat('http://',ROS_IP_ADDRESS,':11311'))
4  setenv('ROS_IP',MY_IP_ADDRESS)
5  rosinit
6  rosnode list
7  end
```

Figure 1: setup.m

Manual user level control and initial mapping:

To solve the given problem statement, we first had to generate a global base map of the robot's configuration space in order to conduct effective path planning. A function was abstracted (Figure 2) from the MATLAB teleoperation example tutorial in order to control our robot and display its current position in the room. From this, waypoints could be precisely located and manually mapped. A slight modification, in concert with our scanning code (Figure 3) seen below , was used in order to map the space.

```
function keyboardDrive()
    robot = rospublisher('/mobile_base/commands/velocity');
    odom = rossubscriber('/odom');
    keyObj = ExampleHelperTurtleBotKeyInput();
    disp('Keyboard Control: ');
    disp('i=forward, k=backward, j=left, l=right');
    disp('q=quit');
    disp('Waiting for input: ');
    % Use ASCII key representations for reply, initialize here
    reply = 0;
    while reply~='q'
        forwardV = 0;   % Initialize linear and angular velocities
        turnV = 0;
        reply = getKeystroke(keyObj);
        switch reply
            case 'i'          % i
                forwardV = 0.2;
            case 'k'      % k
                forwardV = -0.2;
            case 'j'      % j
                turnV = 1;
            case 'l'      % l
                turnV = -1;
        end
        velmsg.Linear.X = forwardV
        velmsg.Angular.Z = turnV;
        send(robot, velmsg);
        odom_data = receive(odom);
        disp(odom_data.Pose.Pose.Position);
    end
end
```

Figure 2: keyboardDrive.m

Kinect scanner interfacing:

The scan2map function (Figure 5) accesses the laser on the robot and gets an accurate scan from a certain position and pose that the robot is at. With driving the robot manually and using the scan2map function we were able to create a map of the room with the scanner as we drove it from point to point until the entire room was mapped (Figure 3). We then cleaned this with image editing software and exported it as a portable networks graphic (.png) filetype format seen below (Figure 4). In order to have a sufficient base map for global planning, the gathered data was manually edited in order to create a clean version to be reimported into matlab. This served a similar purpose to something akin to Google Maps, in which a clean global map might be used for global planning, but local scans are used to determine whether actual conditions are driveable.

Figure 3: Map scan data from Turtlebot

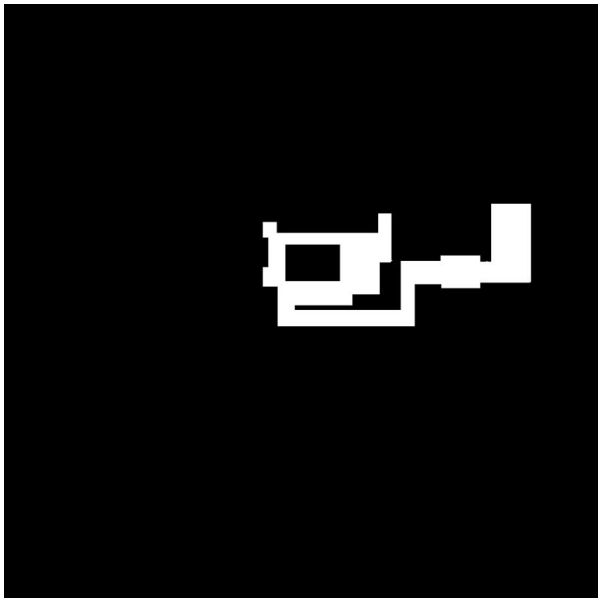

Figure 4: Clean scan data from Turtlebot

```
function y = scan2map(odom, laser, map)
    odom_data = receive(odom);
    scan = receive(laser,1);
    orientation = odom2eul(odom_data.Pose.Pose.Orientation);
    % The z-direction rotation component is the heading of the
    % robot. Use the heading value along with the z-rotation axis to define
    % the rotation matrix
    rotMatrixAlongZ = axang2rotm([0 0 1 orientation(1)]);
    % obtain local XY coordinates for the detected obstacle points
    % in the robot's frame.
    coordsLocal = readCartesian(scan);
    % Convert sensor values (originally in the robot's frame) into the
    % world coordinate frame. This is a 2D coordinate transformation, so
    % only the first two rows and columns are used from the rotation matrix.
    % rotation
    coordsWorld = rotMatrixAlongZ(1:2,1:2) * coordsLocal';
    %translation
    dim = size(coordsWorld);
    coordsWorld = coordsWorld' + repmat([odom_data.Pose.Pose.Position.X odom_data.Pose.Pose.Position.Y], dim(2),
    % Update map based on laser scan data in the world coordinate frame.
    % Populate the map, using the setOccupancy function, by setting the
    % obstacle location on the map as occupied for every sensor reading that
    % detects obstacles
    setOccupancy(map, coordsWorld, 1);

    %count = 1
    show(map)

    y = coordsLocal;
end
```

Figure 5: scan2map.m

Reading the base map:

Using the previously generated base map, a function (Figure 6) was created to read it into a binary occupancy grid using standard MATLAB RST constructors. The image is first read in and loaded, converted to grayscale, then formatted to the ROS specific binary occupancy grid with a 40 meter axis scale (Figure 8 - before PRM nodes are added). Going forward with this map, we can use it in any path planning method that the ROS toolkit can use.

```
% run like the following:
% map = genMap('mapLounge.png');
function map = genMap(image)
    imageMat = imread(image);
    imageMat = rgb2gray(imageMat);
    imageMat = imageMat < 0.5;
    map = robotics.BinaryOccupancyGrid(imageMat, 20);
    map.GridLocationInWorld = [-20 -20];
end
```

Figure 6: genMap.m

Generating paths via Probabilistic Roadmaps:

With the newly generated map, a probabilistic roadmap (PRM) between two points in unoccupied territory on the map can be calculated. To ensure that our robot does not run into walls, we inflate the map with a robot radius of 0.15 meters. The actual robot radius is approximately 0.2 meters, but since we decided not to use the bumper sensors and only the laser scanner, we were able to inflate the map less to allow for more clear paths while still avoiding collisions. The robotics toolkit function to generate the PRM is then applied to the inflated map. The PRM function algorithmically inserts nodes into a connected graph structure based upon available white space in the supplied occupancy grid. The node-fill pattern allows for paths to plan around nearly all available open space, and the final map, post PRM, can be seen below (Figure 8). Parameters such as number of nodes and path connectedness can be tuned in order to generate a suitable selection of available paths while still having control of the processing overhead and later graph traversal.

```
1   function [ prm, map ] = getPRM( image )
2   %Gets the PRM of the map
3       map = genMap(image);
4       show(map)
5       robotRadius = 0.15;
6       mapInflated = copy(map);
7       inflate(mapInflated,robotRadius);
8       prm = robotics.PRM(mapInflated);
9       prm.NumNodes = 500;
10      prm.ConnectionDistance = 10;
11  end
```

Figure 7: genMap.m

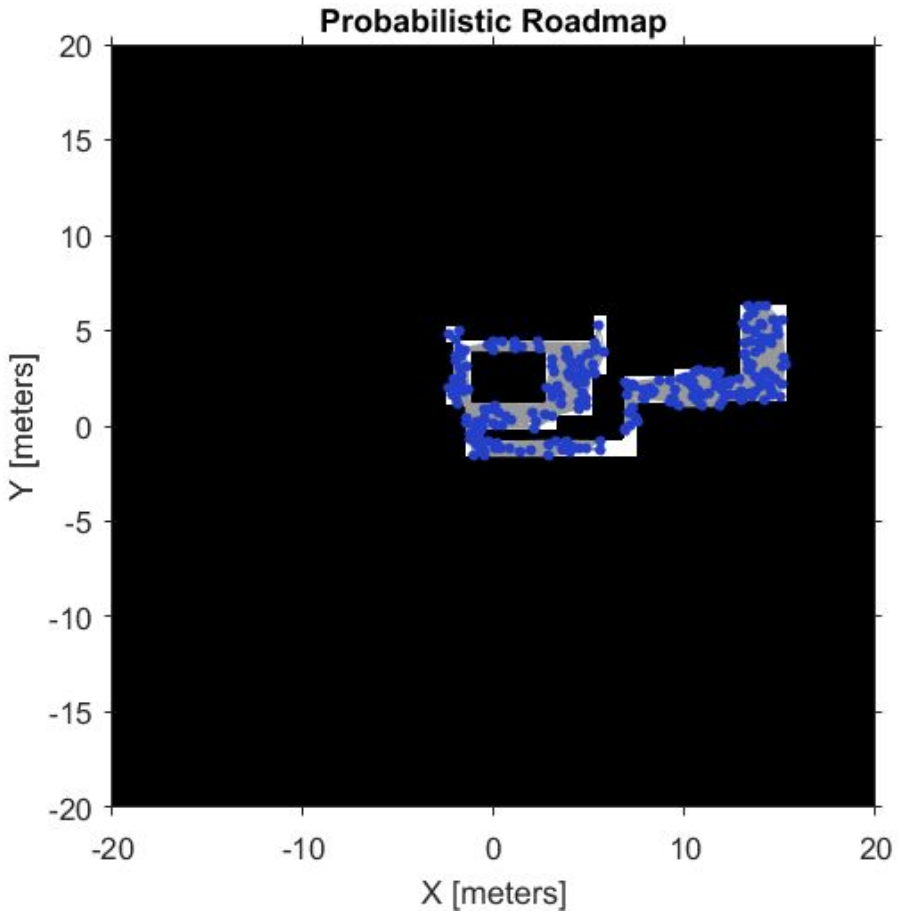

Figure 8: Binary Occupancy Grid with PRM

Unit conversion:

In order to control the robot on its path, a minor conversion from quaternion odometry points to euler angles had to be made (Figure 9). This is due to the fact that robot odometry is stored in the quaternion format which has a combined set of vectors for 3

axis telemetry. However, the primary inputs to the differential drive Turtlebot rely on single axis (Z) rotation velocities in order to function properly.

```
% Must send this odom_data.Pose.Pose.Orientation as input
% i.e.: orientation = odom2eul(odom_data.Pose.Pose.Orientation);
function orientationEul = odom2eul(orientationQuat)
    % Define the Quaternion in a vector form
    q = [orientationQuat.W orientationQuat.X orientationQuat.Y orientationQuat.Z];
    % Convert the Quaternion into Euler angles
    orientationEul = quat2eul(q, 'ZYX');
end
```

Figure 9: odom2eul

Acquiring robot pose data:

To find the current pose of the robot, the currentPose.m function (Figure 10) takes in (or receives) the local odometry by subscribing to the '/odom' topic. Necessary unit conversions are made and the result is returned in standard [X Y Theta] format.

```
function [ pose ] = currentPose( odom )
%CURRENTPOSE Summary of this function goes here
%   Detailed explanation goes here

odom_data = receive(odom);

x = odom_data.Pose.Pose.Position.X;
y = odom_data.Pose.Pose.Position.Y;
orientation = odom2eul(odom_data.Pose.Pose.Orientation);
z = orientation(1);

pose = [x y z];

end
```

Figure 10: currentPose.m

Map registration:

In order to use the same map of the room properly, the robot must be manually placed at the (0,0) point which was marked on the floor for reference to the binary occupancy grid. Once placed, the odometry could be reset and zeroed out by using the resetOdom.m function (Figure 11). This function operates by publishing a standard empty ROS message to the '/reset_odometry' topic in the mobile base as per the ROS specification. This process ensures that the robot local and global coordinate frames are accurately registered to the coordinate system being used on the map and by the PRM functions.

```
function resetOdom()
    emptyMsg = rosmessage('std_msgs/Empty');
    odom_reset = rospublisher('/mobile_base/commands/reset_odometry');
    send(odom_reset, emptyMsg);
end
```

Figure 11: resetOdom.m

Primary Control Loop:

This is the main loop of our program (Figures 12 and 13) which allows for the robot to autonomously follow the given waypoints and avoid colliding with obstacles in its path via the implementation of a simple state machine algorithm. First,initial setup is performed, which includes tasks such as resetting the odometry, initializing the required publishers and subscribers for sending/receiving data, and generating a new PRM from the room map. Next, the controller method is established so that we can access the Pure Pursuit controller which generates robot inputs in the form of robot velocities. Tuneable parameters such as lookahead distance and desired linear velocity are set in order to ensure smooth trajectories without sacrificing obstacle avoidance. The goal radius was adjusted to ensure accurate waypoint following without unnecessary restriction on robot goal pose.

While navigating, the robot exists in 3 effective states: path generation, navigation and collision avoidance By keeping track of a flag variable, the robot can be told to generate a path from its current position to the next desired waypoint. This state is implemented after startup, after arriving within the goal radius of the designated target waypoint and after exiting the collision avoidance state. Navigation is the standard state of operation while the robot is driving between waypoints. While the robot is in this state, it generates single step controls via the Pure Pursuit controller to be used as the next set of velocity inputs and then executes them. If the robot encounters an obstacle it enters the collision avoidance state. The current collision avoidance state is described in detail below (Figure 14) as a basic, open-loop avoidance algorithm.

```
function path_follow(odom, prm)
    %'10.101.84.35' - MIKEY IP
    %resetOdom;
    odom = rossubscriber('/odom');
    robot = rospublisher('/mobile_base/commands/velocity');
    velmsg = rosmessage(robot);
    laser = rossubscriber('/scan');
    % prm = getPRM('mapLounge.png')

    needPath = true;
    visited = 0;

    hold on
    show(prm, 'Map', 'off', 'RoadMap', 'off');
    hold off

    controller = robotics.PurePursuit
    controller.DesiredLinearVelocity = 0.2;
    controller.MaxAngularVelocity = 1;
    controller.LookaheadDistance = 0.3;
    release(controller)
    goalRadius = 0.15;
    Locations = [2.5 0.4; 4.25 2.3; 2.15 4.18; -1.46 3.53; 0.0 0.0];
    numWaypoints = size(Locations);
    numWaypoints = numWaypoints(1);
    disp(numWaypoints)
    lastDist = 10000;
    approachingObjectCounter = 0;
    nonapproachingObjectCounter = 0;
    approachingObjectFlag = false;

    while visited < numWaypoints
        if checkClear(laser)
            if needPath
                release(controller)
                robotCurrentPose = currentPose(odom);
                startLocation = robotCurrentPose(1:2);
                endLocation = Locations((visited+1),:);
                path = findpath(prm, startLocation, endLocation);
                controller.Waypoints = path;
                robotGoal = path(end,:);
                distanceToGoal = norm(startLocation - robotGoal);
                needPath = false;
            end
```

Figure 12: path_follow.m (segment 1, lines 1-44)

```
45 -            if( distanceToGoal > goalRadius )
46
47 -                robotCurrentPose = currentPose(odom);
48                  % Compute the controller outputs, i.e., the inputs to the robot
49                  %[v, omega] = step(controller, robot.CurrentPose);
50 -                [v, omega] = step(controller, robotCurrentPose);
51
52                  % Simulate the robot using the controller outputs
53                  %drive(robot, v, omega)
54 -                velmsg.Linear.X = v;
55 -                velmsg.Angular.Z = omega;
56 -                send(robot, velmsg);
57
58 -                robotCurrentPose = currentPose(odom);
59                  % Re-compute the distance to the goal
60 -                distanceToGoal = norm(robotCurrentPose(1:2) - robotGoal);
61 -            else
62 -                distanceToGoal = 100;
63 -                needPath = true;
64 -                visited = visited + 1;
65 -            end
66 -        else
67
68 -            [approachingObjectFlag, lastDist] = checkApproaching(laser, lastDist);
69 -            if(approachingObjectFlag)
70 -                ++approachingObjectCounter;
71 -            else
72 -                ++nonapproachingObjectCounter;
73 -            end
75 -            if (approachingObjectCounter == 20)
76 -                retreat(robot, velmsg);
77 -                nonapproachingObjectCounter = 0;
78 -                approachingObjectCounter = 0;
79 -                lastDist = 10000;
80 -            elseif (nonapproachingObjectCounter == 20)
81 -                ObstacleAvoidance(robot, velmsg, laser);
82 -                needPath = true;
83
84 -                nonapproachingObjectCounter = 0;
85 -                approachingObjectCounter = 0;
86 -                lastDist = 10000;
87 -            end
88 -        end
89 -    end
90
91 - end
```

Figure 13: path_follow.m (segment 2, lines 45 - 91)

Collision Avoidance State:

Obstacle avoidance is a crucial part of the robot's final design. In its current state, the robot operates on a defined open-loop collision avoidance algorithm (Figure 14) that encompasses a series of velocity and trajectory adjustments designed to move it to a suitable location for follow-on path planning and return to the navigation state.

Obstacle avoidance starts when the robot detects an obstacle in its immediate path. The robot starts off backing up at a relatively slow rate for one move command. After backing up to make room for further maneuvers, the robot enters a loop where the robot will start by turning left by 90 degrees. It will move parallel along the side of the obstacle for approximately one robot length. It then turns back towards its originally heading and scans to ensure if it has cleared the obstacle. If the path is clear, it returns to the path generation state before continuing its navigation. If the path is not clear, it returns to the beginning of the collision avoidance state.

Given the time available, we were satisfied with the performance of the open loop collision avoidance algorithm as it created a suitable response in almost all circumstances. Although limitations were that the robot did not implement any state update or object detection code during the execution of the collision avoidance code and there is no intelligent path planning involved in the maneuver. A proposed solution is to have the robot dynamically generate an updated occupancy grid based upon discovery of the new obstacle. As the unexpected obstacle might not be permanent, this would be done independently of the global map and a new PRM would be generated. Once the PRM was complete, a new path could be developed that would navigate around the obstacle, towards the desired waypoint. This would have the benefit of leveraging existing code structures and would readily fit into the state machine model that the robot currently operates in. This would handle single step, state based update for improved intra-maneuver collision avoidance along with intelligent path planning and would automatically recover in the event a collision avoidance took the robot towards an area from which the waypoint was unreachable. Once the proposed collision avoidance maneuver was complete, the local map could be discarded or used in a more complex system for updating the global maps.

```
function ObstacleAvoidance( robot, velmsg, laser)
%OBSTACLEAVOIDANCE Summary of this function goes here
go = true;
velmsg.Linear.X = -0.2;
velmsg.Angular.Z = 0;
send(robot, velmsg);

while(go)

    velmsg.Linear.X = 0;
    velmsg.Angular.Z = pi/2;
    tic;
    while(toc < 1);
        send(robot, velmsg);
    end
    velmsg.Linear.X = 0.2;
    velmsg.Angular.Z = 0;
    tic;
    while(toc < 2);
        send(robot, velmsg);
    end
    velmsg.Linear.X = 0;
    velmsg.Angular.Z = -pi/3;
    tic;
    while(toc < 1);
        send(robot, velmsg);
    end

    go = ~checkClear(laser);
end
```

Figure 14: obstacleAvoidance.m

Collision detection scanning:

As seen from obstacleAvoidance.m (Figure 14) and path_follow.m (Figures 12 and 13), checkClear.m (Figure 15) is called when the robot is determining what is in front of it. The crux of how the robot determines whether or not the path is clear is by using the Xbox Kinect scan data. The Xbox Kinect scan picks up objects that are in a predefined arc area in front of the robot. The returned sensor data is transformed to return an object that simulates the return of a 2 dimensional lidar sensor for range detection. This system provides information in the form of polar coordinates representing the angle of the lidar scanner and the distance at which the scan impacted the nearest object on that vector. These coordinates are then converted from polar to cartesian coordinates and basic Euclidean geometry is used in order to determine the minimum distance of all detected objects in the scan area. If the minimum distance to an object is underneath a predefined tolerable threshold, 0.6 meters in our case, the function returns false and the

robot enters collision avoidance state. The code and a plot of typical single scan can be seen below.

```
function [ clear ] = checkClear(laser )
%CHECKCLEAR Summary of this function goes here
% Collect information from laser scan
  scan = receive(laser);
  plot(scan);
  data = readCartesian(scan);
  x = data(:,1);
  y = data(:,2);
  % Compute distance of the closest obstacle
  dist = sqrt(x.^2 + y.^2);
  minDist = min(dist);
  % Command robot action
  if minDist < 0.6
      clear = false;
  else
      clear = true;
  end

end
```

Figure 15: checkClear.m

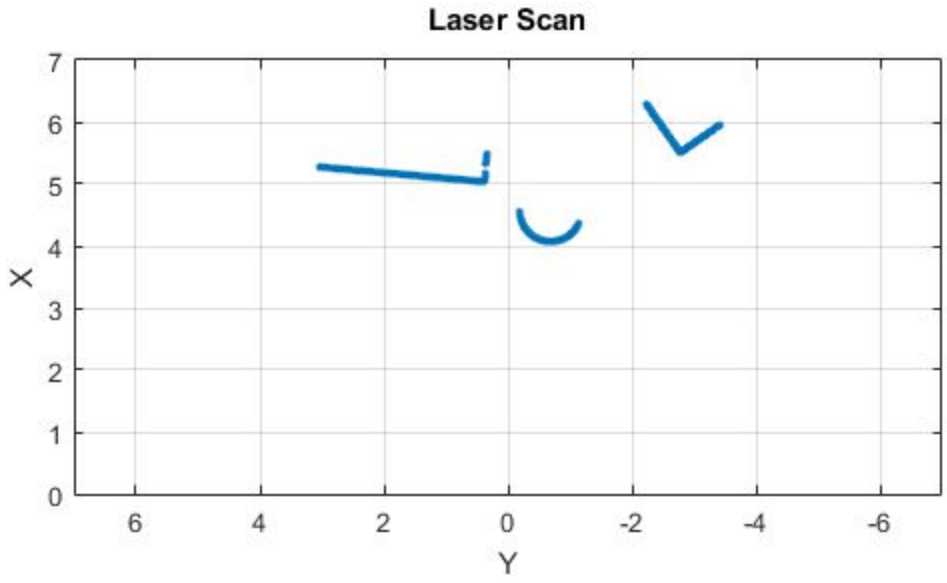

Figure 16: laser scan from TurtleBot scanner

General Practices and Conclusions

One thing that was not unique to an individual script was how we passed all of our robot control objects by reference. At first we were creating new ROS subscriptions or ROS publishers for each function. This certainly reduces the amount of local memory and temporary variables needed for any given function and we certainly saw a performance increase after switching to pass by reference. The team feels that the effects of this would be more greatly felt and realized by a system that uses our work to continue on to do more complex things.

As stated before our group used a Probabilistic Roadmap (PRM) algorithm for path planning. Creating a global map with strategic placement of nodes requires repeated experimentation. The nodes are placed randomly so the roadmap needs to be generated ahead of time and a map should be saved for later if/when it yields a path that creates smooth transitions between waypoints. PRM struggles with trading off between the path planning processing efficiency and the accuracy and smoothness of the path generated. As we can see in genMap.m (Figure 7) line 9, we choose to use 500 nodes for our PRM algorithm. This created often smoother lines that would more directly lead to waypoints however we definitely saw a performance trade off during path route generation. When each way point was reached you could see the robot pause briefly to plan its next path before continuing onward. Members of the team postulated that a proper trade off could be achieved between path smoothness and planning efficiency if the placement of nodes on the map was more closely tied to where the waypoints were. You could theoretically decrease the amount of total nodes, while increasing the density of nodes in the areas between waypoints by placing a circumference of nodes around each waypoint and not placing any other nodes between them and the way point. This allows for the robot to enter the waypoint from any direction and not need to worry about reaching a node that is ‘closest’ to the waypoint but in an awkward position relative to the robot and the waypoint itself.

As a group, we felt that each requirement of the project has been met successfully and all have been done so efficiently. However successful we may have been, that success is largely credited to our experimentation with our environment. The ability of our robot to work well inside a test area might not translate to any use case the robot may find itself in, since some design decisions were tailored to the environment we knew the robot would be exploring. Going forward, we are confident that we can adapt these solutions and lessons learned to other mobile robotics challenges that we are faced with in the future.