Faculty of Engineering
Department of Mechanical and Mechatronics Engineering

LAB 3: PATH PLANNING

Prepared by
Oluwatoni Olorunfunmi Ogunmade
Lalit Lal
Mohamed El Shatshat
Michael Kaca

20574959
20572296
20576631
20564560

ooogunma@edu.uwaterloo.ca
l2lal@edu.uwaterloo.ca
mrelshat@edu.uwaterloo.ca
mskaca@edu.uwaterloo.ca

4A Mechatronics Engineering
6 July 2019

1 Introduction

This report outlines the process of implementing the Probabilistic Roadmap (PRM) algorithm for a turtlebot. A PRM is a series of straight line paths linking milestones through the allowed space of the turtlebot to go from the start point to the goal point. This report comprises of two main sections - Theory and Implementation, and Results and Discussion for both the path planning and the tracking.

2 Theory and Implementation

2.1 Path Planning Algorithm

2.1.1 Probabilisitc Roadmap

A PRM is a probabilistic roadmap in the free configuration space of the robot by generating and interconnecting a large finite number of configurations of the robot. Milestones were generated in the free space using bridge sampling. Additionally, the map was inflated by the width of the robot to ensure that any paths generated can be physically met. The PRM implemented is a multi-query PRM where each of the goals specified were added to the graph and more milestones were generated until the goal could be reached. This type of PRM is more suitable for online use as the path can be quickly computed.

2.1.2 Shortest Path

Once the goal point is found by the PRM, the shortest path is determined. Possible methods for finding the shortest path that are discussed in the course are Wavefront, Potential Fields, Dijkstra's, and A*. The method used in this report is A*, an algorithm that builds on Dijkstra's algorithm for shortest path finding by adding a heuristic bias. This bias increases the speed of the algorithm by adding value to nodes that advance advancement towards the goal position at each node.

2.2 Path Tracking

The path tracking algorithm chosen in this implementation is Pure Pursuit. The pure pursuit approach is a method of geometrically determining the curvature that will drive the vehicle to a chosen path point, termed the goal point. This goal point is a point on the path that is one look ahead distance from the current vehicle position. The turtlebot's steering angle can be determined using the goal point and the angle between the turtlebot's heading and the look ahead vector.

The implementation of the pure pursuit algorithm is broken down into four steps. The first step is to use the current point and the next point in the path to create the curvature that the turtlebot will follow. The derivations for the equations are implemented in the pure pursuit ros node [1]. The next step is to turn the robot to the correct starting orientation. A timer is added in this step in case the robot fails to turn to the correct position. The third step is to set the linear and angular velocities of the turtlebot. The linear velocity is selected first, and the angular velocity will be the linear velocity divided by the radius of the circle determined by the curvature in step 1. The fourth and final step is to check if the turtlebot has reached the goal point. Once the turtlebot arrives at the goal point the process returns to step 1 to calculate the new required curvature between the current and next point.

3 Results and Discussion

3.1 Path Planning

The PRM algorithm produced the path shown in figure 3-1

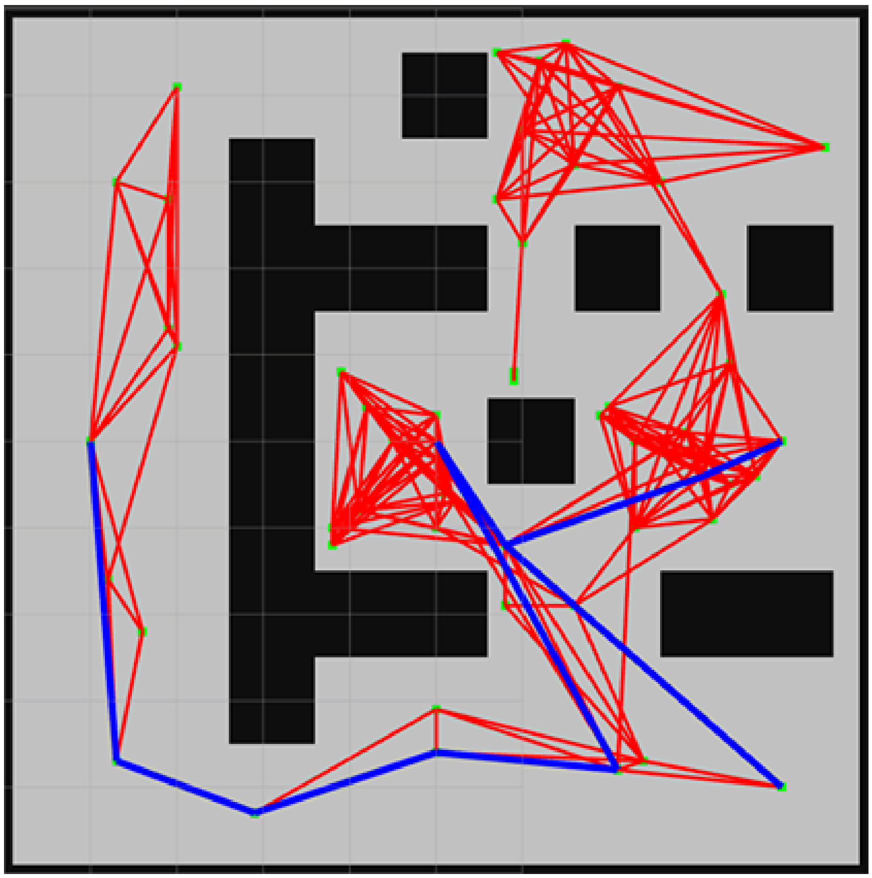
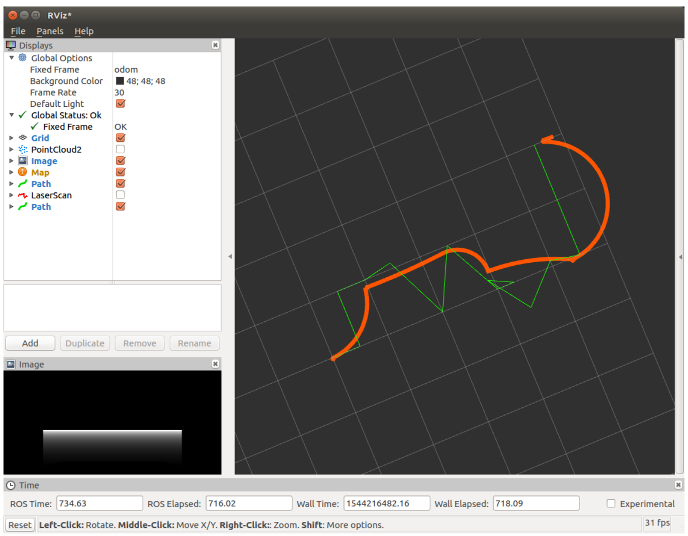


Figure 3-1. Generated milestones in green, Path in blue and edges in green.

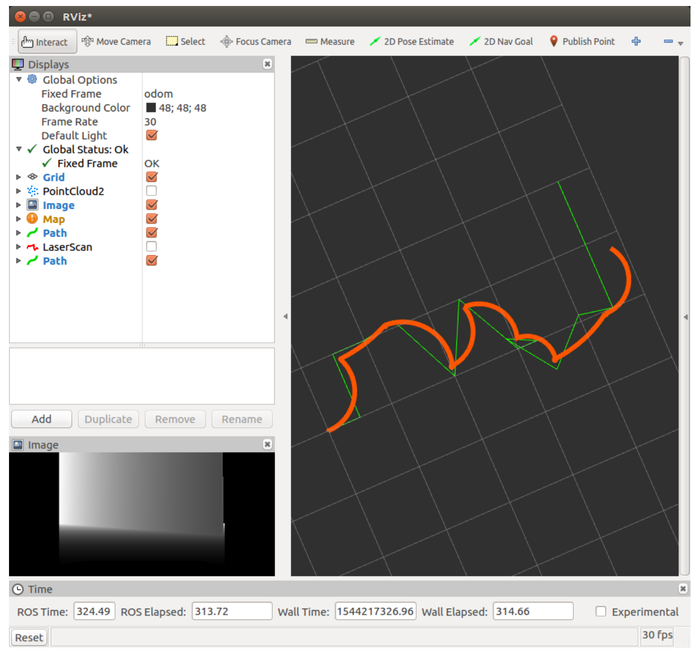
The algorithm was able to produce a path over three runs in 0.1363 seconds. To improve the performance of the algorithm a graph searching algorithm that tracks the graphs from previous searches to reduce the time required to search the graph again. This class of algorithms are called incremental heuristic search algorithms and include algorithms like life long planning A*, D* and D* lite. The path produced by the algorithm is quite straight and tends to be on the outside of obstacles. This was expected of PRM algorithms designed in this way. The straightness is due to the few number of milestones generate, 51 samples were generate to meet all 3 goals. This often means that the path would have sharp turns and the path generated might not be the optimal path. Because the algorithm is probabilistically complete and will sample until it finds a solution, in the absence of a time bound, if a path connecting the start position to the goals exist, the algorithm will find it. To speed up the process, in obstacle rich environments, sampling closer to obstacles would increase the chances of getting around obstacles and through narrow passages.

3.2 Path Tracking

As previously mentioned, the chosen Path Tracking algorithm for this lab is the pure pursuit algorithm. The experimental results were not obtained for this lab. The simulation results can be seen below for an arbitrary user-created path (required for developing the pure pursuit algorithm in parallel with the development of the PRM algorithm). Also, the results obtained from following the path given by the PRM (discussed above) can be seen below. The look-ahead distance indicates the total amalgamated linear distance (traversed in between target points) that the robot is willing to travel. In other words, this look-ahead distance determines how far away the robot is willing to travel for each iteration.



(a) Test Path: look-ahead distance = 2m



(b) Test Path: look-ahead distance = 1m. Note how the path isn't finished due to RVIZ continuously crashing before the completion of the path

The results show how vital the look-ahead distance for this path-following algorithm. Ideally, the look-ahead distance should be small enough so that the robot does not miss any crucial target points, in order to avoid obstacles, and it should also be large enough so that the robot does not spend an excessive amount of time circling around each redundant point. If the look-ahead distance is smaller than the distance between any two sequential points along the given path, then the robot is likely to get stuck. This is one of the limitations in the group's applied path follower. However, this can be mitigated by having a dynamic look-ahead distance that depends on user-defined parameters. The designer can also define the look-ahead point to follow the index number of the target points instead of the actual distance (in meters). Since pure pursuit uses a circular path to traverse from point A to point B, it tends to be more adaptable to following sharp paths. However, this can also lead to cutting (rounding) corners. the biggest limitation of pure pursuit is that can be ineffective in very narrow paths where the follower arc extends beyond the physically available path. This is where more complex path following algorithms are required. Figure 3-3 demonstrates the limitations of the pure pursuit algorithm, especially when applying it to target points that are spaced far away. This causes the radius of the circle (that the robot follows) to be quite large, thus malfunctioning in tight spaces.

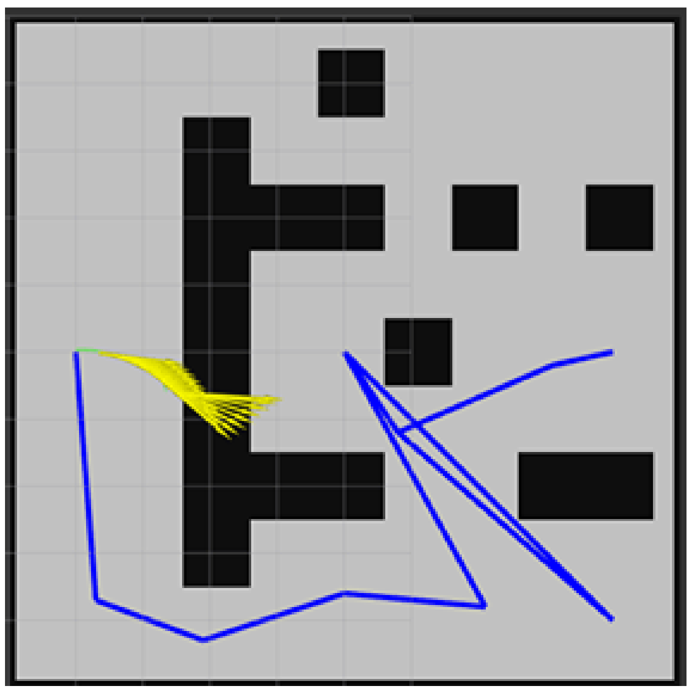


Figure 3-3. PRM Path: look-ahead distance = 1m, The path of the robot is shown in yellow, the robot collides with the wall of the maze

References

[1] Martin Lundgren. ‘‘Path Tracking for a Miniature Robot’’. In: (Dec. 2018).

Appendix A

Appendix A.1 Path Planning

```

prm.h

#ifndef PRM_H
#define PRM_H

#include <ros/ros.h>
#include <vector>
#include "particle_filter.h"
#include <visualization_msgs/Marker.h>

typedef struct {
  //const int NUM_NODES = 500;
  //NX2 array that contains milestones (nodes) for potential shortest path
  std::vector<double> milestones_x;
  std::vector<double> milestones_y;

  //int milestones[NUM_NODES][2]; //initially milestones were just an array but it's dynamic, so let's
    create a vectr. Wish I could do dynamic vector
} milestone_t;

class PRM {
  private:
    ros::Publisher prm_path_;
    ros::Publisher path_milestones_;
    ros::Publisher marker_pub_;

    ros::Subscriber odom_sub_;
    ros::Subscriber command_sub_;
    ros::Subscriber sim_ips_sub_;
    ros::Subscriber ips_sub_;
    ros::Subscriber map_sub_;

    nav_msgs::Path robot_path_;
    visualization_msgs::Marker points;
    visualization_msgs::Marker line_list;
    visualization_msgs::Marker path_list;

    double ips_x_;
    double ips_y_;
    double ips_yaw_;
    bool measurement_ready_;
    bool map_received_;
    map_t map_;
    std::queue<std::array<double, 3> > pose_deltas_;
    double last_command_time_;
    double last_measurement_time_;
    Eigen::Vector3d measurements_, measurements_raw_;
    double dt_;
    std::default_random_engine generator_;
    geometry_msgs::Twist vel_;
    bool plan_created_;

    int ec_; // initialize edge count
    std::vector< std::vector<int> > edge_matrix_; //edge matrix
    milestone_t stones_;

  public:
    ros::NodeHandle nh_;
    PRM();
    void ConvertMap(const nav_msgs::OccupancyGrid &map_msg);
```

```
    map_t InflateMap(map_t mapi, int inflation_val);

    //void CommandCb(const geometry_msgs::Twist::ConstPtr &msg);
    void MapCb(const nav_msgs::OccupancyGrid &msg);
    void PoseCb(const geometry_msgs::PoseWithCovarianceStamped &msg);
    void SimIPSCb(const gazebo_msgs::ModelStates::ConstPtr &msg);
    void OdomCb(const nav_msgs::Odometry::ConstPtr& msg);
    void DrawCurve(int k);
    void Plan();

    //Core PRM tingz
    template <typename T> std::vector<size_t> SortIndexes(const std::vector<T> &v);
    bool GaussianSample(int (&sample)[2], int M, int N);
    void BridgeSample(int (&sample)[2], int M, int N);
    void MilestoneGen(map_t map);
    void CollisionCheck(int num_neighbors);
    void FindShortestPath(std::vector<double> milestones_x,
                          std::vector<double> milestones_y,
                          std::vector <std::vector<int> > edges,
                          int start_index, int end_index,
                          std::vector <int> &path, int &path_length);
    void Publish_Points(float x, float y);
    void Plot_Lines(float x1, float y1, float x2, float y2);
    void Plot_path(float x1, float y1, float x2, float y2);

};
#endif //PRM_H_


prm.cpp

////////////////////////////////////////////////////////////////
//
// turtlebot_example.cpp
// This file contains example code for use with ME 597 lab 2
// It outlines the basic setup of a ros node and the various
// inputs and outputs needed for this lab
//
// Author: James Servos
//
// ////////////////////////////////////////////////////////////

#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Twist.h>
#include <tf/transform_datatypes.h>
#include <gazebo_msgs/ModelStates.h>
#include <visualization_msgs/Marker.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include "prm.h"
#include <numeric>

#define TAGID 0

// Mapping helper functions.
inline int GetMapIndex(map_t map, int x, int y) {
  return ((x) + (y) * map.size_x);
}

short sgn(int x) { return x >= 0 ? 1 : -1; }

//Bresenham line algorithm (pass empty vectors)
// Usage: (x0, y0) is the first point and (x1, y1) is the second point. The calculated
//        points (x, y) are stored in the x and y vector. x and y should be empty
//    vectors of integers and shold be defined where this function is called from.
void bresenham(int x0, int y0, int x1, int y1, std::vector<int> &x, std::vector<int> &y) {
    int dx = abs(x1 - x0);
```

```
    int dy = abs(y1 - y0);
    int dx2 = x1 - x0;
    int dy2 = y1 - y0;

    const bool s = abs(dy) > abs(dx);

    if (s) {
        int dx2 = dx;
        dx = dy;
        dy = dx2;
    }

    int inc1 = 2 * dy;
    int d = inc1 - dx;
    int inc2 = d - dx;

    x.push_back(x0);
    y.push_back(y0);

    while (x0 != x1 || y0 != y1) {
      if (s)
          y0 += sgn(dy2);
      else
          x0 += sgn(dx2);
      if (d < 0)
          d += inc1;
      else {
          d += inc2;
          if (s)
              x0 += sgn(dx2);
          else
              y0 += sgn(dy2);
      }

      //Add point to vector
      x.push_back(x0);
      y.push_back(y0);
    }
}

void PRM::OdomCb(const nav_msgs::Odometry::ConstPtr& msg) {
  double now = ros::Time::now().toSec();
  dt_ = now - last_command_time_;
  std::array<double,3> pose_delta;

  if (last_command_time_ == -1){
    last_command_time_ = now;
    return;
  }

  if (map_received_) {
    pose_delta[0] = msg->twist.twist.linear.x * dt_;
    pose_delta[1] = msg->twist.twist.linear.y * dt_;
    pose_delta[2] = msg->twist.twist.angular.z * dt_;
    pose_deltas_.push(pose_delta);
  }
  last_command_time_ = now;
  ROS_INFO("odom");
}

//Callback function for the Position topic (LIVE)
void PRM::PoseCb(
  const geometry_msgs::PoseWithCovarianceStamped &msg) {
  measurements_ << msg.pose.pose.position.x, msg.pose.pose.position.y,
                   tf::getYaw(msg.pose.pose.orientation);
  measurements_raw_ << msg.pose.pose.position.x, msg.pose.pose.position.y,
                   tf::getYaw(msg.pose.pose.orientation);
  ROS_INFO("pose");
```

```
  measurement_ready_ = true;
  //ROS_DEBUG("pose_callback X: %f Y: %f Yaw: %f", measurements_(0),
  //          measurements_(1), measurements_(2));
}

map_t PRM::InflateMap(map_t map, int inflation_val){
  map_t out;
  out.size_x = map.size_x;
  out.size_y = map.size_y;
  out.scale = map.scale;
  out.origin_x = map.origin_x;
  out.origin_y = map.origin_y;
  out.cells = (cells_t*)calloc(map.size_x*map.size_y, sizeof(cells_t));

  for(int i = 0; i < out.size_x; i++) {
    for(int j = 0; j < out.size_y; j++) {
      ///*
      if (map.cells[GetMapIndex(map, i, j)].occupied) {
        out.cells[GetMapIndex(out, i, j)].occupied = true;
        int x = i;
        int y = j;
        x--;
        while(x >= 0 && abs(x-i) <= inflation_val) {
          out.cells[GetMapIndex(out, x, j)].occupied = true;
          x--;
        }
        x = i;
        x++;
        while(x < out.size_x && abs(x-i) <= inflation_val) {
          out.cells[GetMapIndex(out, x, j)].occupied = true;
          x++;
        }

        y--;
        while(y >= 0 && abs(y-j) <= inflation_val) {
          out.cells[GetMapIndex(out, i, y)].occupied = true;
          y--;
        }
        y = j;
        y++;
        while(y < out.size_y && abs(y-j) <= inflation_val) {
          out.cells[GetMapIndex(out, i, y)].occupied = true;
          y++;
        }
      }
      //*/
    }
  }
  return out;
}

void PRM::ConvertMap(const nav_msgs::OccupancyGrid &map_msg) {
  map_.size_x = map_msg.info.width;
  map_.size_y = map_msg.info.height;
  map_.scale = map_msg.info.resolution;
  map_.origin_x = map_msg.info.origin.position.x;
  map_.origin_y = map_msg.info.origin.position.y;

  // Convert to player format
  map_.cells = (cells_t*)malloc(sizeof(cells_t)*map_.size_x*map_.size_y);
  for(int i = 0; i < map_.size_x * map_.size_y; i++) {
    if(map_msg.data[i] == 0)
      map_.cells[i].occupied = false;
    else
      map_.cells[i].occupied = true;
  }
  map_ = InflateMap(map_, 2);
}
```

```
//Example of drawing a curve
void PRM::DrawCurve(int k) {
    // Curves are drawn as a series of stright lines
    // Simply sample your curves into a series of points

    double x = 0;
    double y = 0;
    double steps = 50;

    visualization_msgs::Marker lines;
    lines.header.frame_id = "/map";
    lines.id = k; //each curve must have a unique id or you will overwrite an old ones
    lines.type = visualization_msgs::Marker::LINE_STRIP;
    lines.action = visualization_msgs::Marker::ADD;
    lines.ns = "curves";
    lines.scale.x = 0.1;
    lines.color.r = 1.0;
    lines.color.b = 0.2*k;
    lines.color.a = 1.0;

    //generate curve points
    for(int i = 0; i < steps; i++) {
        geometry_msgs::Point p;
        p.x = x;
        p.y = y;
        p.z = 0; //not used
        lines.points.push_back(p);

        //curve model
        x = x+0.1;
        y = sin(0.1*i*k);
    }

    //publish new curve
    marker_pub_.publish(lines);

}

//Callback function for the map
void PRM::MapCb(const nav_msgs::OccupancyGrid &msg) {
  //This function is called when a new map is received
  ConvertMap(msg);
  ROS_INFO("Made the map!");
  map_received_ = true;
}

void PRM::SimIPSCb(const gazebo_msgs::ModelStates::ConstPtr &msg) {
  double now = ros::Time::now().toSec();
  double yaw = tf::getYaw(msg->pose[8].orientation); // Robot Yaw
  measurements_raw_ << msg->pose[8].position.x, msg->pose[8].position.y, yaw;

  if (last_measurement_time_ == -1 || now - last_measurement_time_ > 1) {
    std::normal_distribution<double> x_noise(0, 0.1);
    std::normal_distribution<double> theta_noise(0, 0.01);
    //TODO find which model has the right index for the lookup below
    measurements_ << msg->pose[8].position.x + x_noise(generator_),
    msg->pose[8].position.y + x_noise(generator_), yaw + theta_noise(generator_);

    //ROS_INFO("%s %f %f %f", msg->name[8].c_str(), msg->pose[8].position.x,
    //          msg->pose[8].position.y, yaw);
    //ROS_INFO("%lf %lf %lf ", measurements_(0), measurements_(1), measurements_(2));
    last_measurement_time_ = now;
    measurement_ready_ = true;
  }
}

template <typename T> std::vector<size_t> PRM::SortIndexes(const std::vector<T> &v) {
```

```
  // initialize original index locations
  std::vector<size_t> idx(v.size());
  iota(idx.begin(), idx.end(), 0);

  // sort indexes based on comparing values in v
  sort(idx.begin(), idx.end(),
       [&v](size_t i1, size_t i2) {return v[i1] < v[i2];});

  return idx;
}

bool PRM::GaussianSample(int (&sample)[2], int M, int N) {
  int sample_q[2];
  int sample_qnot[2];

  // generate uniform random number between 0 and M and 0 and N
  std::uniform_int_distribution<int> uniform_x_distribution(0,M);
  std::uniform_int_distribution<int> uniform_y_distribution(0,N);

  // generate uniform random sample
  sample_q[0] = uniform_x_distribution(generator_);
  sample_q[1] = uniform_y_distribution(generator_);
  while(sample_q[0] > M || sample_q[1] > N) {
    sample_q[0] = uniform_x_distribution(generator_);
    sample_q[1] = uniform_y_distribution(generator_);
  }

  // generate random points with gaussian dist around original random point above
  std::normal_distribution<double> norm_x_distribution(sample_q[0],2.0);
  std::normal_distribution<double> norm_y_distribution(sample_q[1],2.0);

  sample_qnot[0] = norm_x_distribution(generator_);
  sample_qnot[1] = norm_y_distribution(generator_);
  while((sample_qnot[0] > M || sample_qnot[1] > N) || sample_qnot[0]<0 || sample_qnot[1]<0) {
    // while out of bounds, keep sampling
    sample_qnot[0] = norm_x_distribution(generator_);
    sample_qnot[1] = norm_y_distribution(generator_);
  }

  if (map_.cells[GetMapIndex(map_, sample_q[0], sample_q[1])].occupied && !map_.cells[GetMapIndex(map_, sample_qnot[0], sample_qnot[1])].occupied) {
    // if sample_q is occupied and sample_qnot is free, take sample_q because we are bridge sampling (want occupied points)
    sample[0] = sample_q[0];
    sample[1] = sample_q[1];
    return true;
  } else if (!map_.cells[GetMapIndex(map_, sample_q[0], sample_q[1])].occupied && map_.cells[GetMapIndex(map_, sample_qnot[0], sample_qnot[1])].occupied) {
    // if sample_q is free and sample_qnot is occupied, take sample_qnot because we are bridge sampling (want occupied points)
    sample[0] = sample_qnot[0];
    sample[1] = sample_qnot[1];
    return true;
  } else {
    return false;
  }
}

void PRM::BridgeSample(int (&sample)[2], int M, int N) {
  int point1[2];
  int point2[2];
  int keep = 0;

  // keep sampling until you get a valid miletstone
  while(!keep) {
    while(!GaussianSample(point1, M, N)) {} //wait to get a successful point for point one
    while(!GaussianSample(point2, M, N)) {} //wait for another successful point for point 2
//    ROS_INFO("point1 x:%d y:%d", point1[0], point1[1]);
```

```
//    ROS_INFO("point2 x:%d y:%d", point2[0], point2[1]);

    // got a successful pair of points, run bridge sample
    int x_bridge = (point1[0] + point2[0])/2;
    int y_bridge = (point1[1] + point2[1])/2;
//    ROS_INFO("bridge x:%d y:%d", x_bridge, y_bridge);

    if(!map_.cells[GetMapIndex(map_, x_bridge, y_bridge)].occupied) //if bridge point in open space, get the sample, break out of loop
    {
      if(std::find(stones_.milestones_x.begin(), stones_.milestones_x.end(), sample[0]) == stones_.milestones_x.end() &&
        std::find(stones_.milestones_y.begin(), stones_.milestones_y.end(), sample[1]) == stones_.milestones_y.end())
      {
        /* new samples */
        sample[0] = x_bridge;
        sample[1] = y_bridge;

        keep = 1;
      }
    }
  }
}

void PRM::MilestoneGen(map_t map) {
  int M = map.size_x;
  int N = map.size_y;

  //generate TWO random points using LAVALLE Gaussian sampling (x_sam, y_sam) that is within your map
  //p1 = uniform random sample in FREE space (map(x1,y1 != 1))
  //p2 = sample at random with gaussian distribution
  //GET P1

  //repeat for p3, p4 to get P2
  // BRIDGE SAMPLE
  // run bridge sample on P2 and P2

  int sample[2] = {0};
  BridgeSample(sample, M, N); // blocking call, waits for the sample to be generated.
  //ROS_INFO("sample x:%d y:%d", sample[0], sample[1]);
  //ROS_INFO("bounds x:%d y:%d", map_.size_x, map_.size_y);

  /* new samples */
  stones_.milestones_x.push_back(sample[0]);
  stones_.milestones_y.push_back(sample[1]);
  Publish_Points((float)sample[0], (float)sample[1]);
}

// input: number of neighbors to check for each node
void PRM::CollisionCheck(int num_neighbors) {
  //sort submatrix [0....n-2] in order or distance from recently added node
  std::vector<int> d;
  std::vector<int> d2;
  std::vector<size_t> ind;

  if(stones_.milestones_x.size() <= 2)
  {
    return;
  }
  // Attempt to add closest num_neighbors edges to a set to check, labelled as ind
  int cur = stones_.milestones_x.size()-1;
  for (int i = 0; i < cur; i++) {
    float x_diff = abs(stones_.milestones_x[cur] - stones_.milestones_x[i]);
    float y_diff = abs(stones_.milestones_y[cur] - stones_.milestones_y[i]);
    float norm_val = sqrt((pow(x_diff, 2) + pow(y_diff, 2)));
    // need to check what norm does in matlab and implement the C++ version of it
    d.push_back(int(norm_val));
```

```
  }

  // for(int i = 0; i < d.size(); i++)
  // {
  //   ROS_INFO("D val of %d is %d", i, d[i]);
  // }
  //sort the vector of distances in order of smallest first, closest neighbor strategy
  d2 = d;
  std::sort (d2.begin(), d2.end());
  ind = SortIndexes(d); // get the indices of the shortest points in order of smallest distance first
  //Check for collisions with node CUR, update edge database appropriately:
  //ROS_INFO("size of D vector = %d", d.size());
  for(int j = 0; j < std::min((unsigned int)num_neighbors,(unsigned int)ind.size()); j++) {

    //get points for raytrace (bresenham)
    ec_++;
    double x1 = stones_.milestones_x[cur];
    double y1 = stones_.milestones_y[cur];
    double x2 = stones_.milestones_x[ind[j]];
    double y2 = stones_.milestones_y[ind[j]];

    //ROS_INFO("x1 = %lf y1 = %lf x2 = %lf y2 = %lf", x1, y1, x2, y2);

    // create vectors that will be returned from bresenham
    std::vector<int> vector_x;
    std::vector<int> vector_y;
    bresenham(x1,y1,x2,y2, vector_x, vector_y);

    // default is no collision, if any ray trace element has a map value of 1, collision
    bool collision = false;

    for(int k = 0; k < vector_x.size(); k++) {
      if (map_.cells[GetMapIndex(map_, vector_x[k], vector_y[k])].occupied) {
        collision = true;
        break;
      }
    }

    //ROS_INFO("edge matrix size is %d", edge_matrix_.size());
    //ROS_INFO("cur index is %d", cur);
    //prepare edge matrix (resize if necessary)
    if(edge_matrix_.size() <= ind[j]) {
      //ROS_INFO("Resizing");
      edge_matrix_.resize(ind[j]+1);  // resize top level vector
      for(int len = 0; len < edge_matrix_.size(); len++) {
        edge_matrix_[len].resize(ind[j]+1);
      }
    }

    //safe check - incase cur is actually bigger than ind - make array as big as possible
    if(edge_matrix_.size() <= cur) {
      //ROS_INFO("Resizing");
      edge_matrix_.resize(cur+1);  // resize top level vector
      for(int len = 0; len < edge_matrix_.size(); len++) {
        edge_matrix_[len].resize(cur+1);
        //ROS_INFO("STILL IN THIS LOOP MUAHAHAHAH");
      }
    }

    //ROS_INFO("edge matrix size is %d by %d", edge_matrix_.size(), edge_matrix_[0].size());
    //ROS_INFO("Index is %d cur is %d", ind[j], cur);
    //update edge matrix now that it's in the right size
    if(!collision) {
      //ROS_INFO("No Collision");
      // update edges[index] and edges[index_transpose] = 1
      edge_matrix_[ind[j]][cur] = 1;
      edge_matrix_[cur][ind[j]] = 1;
      Plot_Lines(x1,y1,x2,y2);
```

```
    } else {
      //ROS_INFO("Collision");
      //update edges[index] and edges[index_transpose] = 0
      edge_matrix_[ind[j]][cur] = 0;
      edge_matrix_[cur][ind[j]] = 0;
    }
  }
}

void PublishGraph() {
}

inline double FindDist(int start, int end, std::vector<double> milestones_x,
 std::vector<double> milestones_y) {
  return sqrt(pow(milestones_x[end] - milestones_x[start], 2) + pow(milestones_y[end] - milestones_y[start
    ], 2));
}

typedef struct list_entry{
  int index;
  int prev_index;
  double curr_togo;
  double curr;
} entry_t;

void PRM::FindShortestPath(std::vector<double> milestones_x,
  std::vector<double> milestones_y, std::vector <std::vector<int> > edges,
  int start_index, int end_index, std::vector <int> &path, int &path_length) {

  //Implementation A* star algorithm
  std::vector <entry_t> open_list;
  std::vector <entry_t> closed_list;
  int n = milestones_x.size();
  double dist[n][n] = {};
  double max_dist = 0;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (edges[i][j]) {
        dist[i][j] = FindDist(i, j, milestones_x, milestones_y);
        dist[j][i] = dist[i][j];
        if (dist[i][j] > max_dist) {
          max_dist = dist[i][j];
        }
      }
    }
  }

  entry_t temp {start_index, 0, FindDist(start_index, end_index, milestones_x, milestones_y), 0};
  //ROS_INFO("temp %d: %d %d %lf %lf", 0, temp.index, temp.prev_index,
  //          temp.curr_togo, temp.curr);
  open_list.push_back(temp);
  //ROS_INFO("point %d: %lf %lf %lf %lf", 0, open_list[0].index, open_list[0].prev_index,
  //          open_list[0].curr_togo, open_list[0].curr);
  bool done = false;
  while(!done) {
    if (open_list.empty()) {
      path_length = 0;
      path.clear();
      ROS_INFO("The open list is empty!");
      return;
    }
    int min_dist = INFINITY;
    int open_index = -1;
    int graph_index = -1;
    // Find best node in the open set i.e. the node with the smallest cost
    for (int i = 0; i < open_list.size(); i++) {
      if (open_list[i].curr_togo < min_dist) {
```

```
        min_dist = open_list[i].curr_togo;
        graph_index = open_list[i].index;
        open_index = i;
      }
    }
    if (open_index != -1) {
      closed_list.push_back(open_list[open_index]);
    } else {
      ROS_INFO("Open list contents");
      for (int i = 0; i < open_list.size(); i++) {
        //ROS_INFO("point %d: %d %d %d %d", i, open_list[i][0], open_list[i][1],
        //          open_list[i][2], open_list[i][3]);
      }
      ROS_FATAL("SOMETHING IS WRONG WITH THE OPEN LIST!");
    }
    //Check the end condition
    if (graph_index == end_index) {
        done = true;
        ROS_INFO("found a path to the end!");
        continue;
    }
    //Go through all the neighbouring nodes
    std::vector <int> neighbors;
    for (int i = 0; i < edges.size(); i++) {
        if (edges[graph_index][i]){
            neighbors.push_back(i);
        }
    }

    for (int i = 0; i < neighbors.size(); i++) {
      int loop = 0;
      for (int j = 0; j < closed_list.size(); j++) {
          if (closed_list[j].index == neighbors[i]) {
              loop++;
          }
      }
      if (loop) {
          continue;
      }

      double dtogo = FindDist(end_index, neighbors[i], milestones_x, milestones_y);
      double dcur = open_list[open_index].curr + dist[graph_index][neighbors[i]];

      loop = 0;
      int found = -1;
      for (int j = 0; j < open_list.size(); j++) {
          if (open_list[j].index == neighbors[i]) {
              found = j;
          }
      }
      entry_t temp;
      temp.index = neighbors[i];
      temp.prev_index = graph_index;
      temp.curr_togo = dtogo + dcur;
      temp.curr = dcur;

      // if the node is not in the open list add it and if it is update the
      // current dist from the origin
      if (found == -1) {
          open_list.push_back(temp);
      } else {
          if (dcur < open_list[found].curr) {
            open_list[found] = temp;
          }
      }
    }
    open_list.erase(open_list.begin() + open_index);
  }
```

```

  done = false;
  int found;
  int cur = end_index;
  int cur_closed;
  for (int cur_closed = 0; cur_closed < closed_list.size(); cur_closed++) {
      if (closed_list[cur_closed].index == end_index) {
          found = cur_closed;
      }
  }
  path.push_back(cur);
  int prev = closed_list[found].prev_index;
  path_length = 0;
  ROS_INFO("%d, ", cur);

  while (!done) {
    if (prev == start_index) {
        done = 1;
        ROS_INFO("finished backtracking!");
    }
    cur = prev;
    ROS_INFO("%d, ", cur);

    for (int j = 0; j < closed_list.size(); j++) {
      if (closed_list[j].index == cur) {
          found = j;
      }
    }
    path.push_back(cur);
    prev = closed_list[found].prev_index;
  }
  for (int i = 0; i < path.size()-1; i++) {
    path_length += FindDist(path[i], path[i+1], milestones_x, milestones_y);
  }
}

void PRM::Publish_Points(float x, float y) {
  visualization_msgs::Marker line_strip;
  points.header.frame_id = line_strip.header.frame_id = "/odom";
  points.header.stamp = line_strip.header.stamp = ros::Time::now();
  points.ns = line_strip.ns = "points_and_lines";
  points.action = line_strip.action = visualization_msgs::Marker::ADD;
  points.pose.orientation.w = line_strip.pose.orientation.w = line_list.pose.orientation.w = path_list.pose.orientation.w = 1.0;
  points.id = 0;
  line_strip.id = 1;
  points.type = visualization_msgs::Marker::POINTS;
  line_strip.type = visualization_msgs::Marker::LINE_STRIP;

  points.scale.x = map_.scale; // size of the point
  points.scale.y = map_.scale; // this is the size of the point itself

  //points are green
  points.color.g = 1.0f;
  points.color.a = 1.0;

  geometry_msgs::Point p;
  p.x = x * map_.scale + map_.origin_x;
  p.y = y * map_.scale + map_.origin_y;
  p.z = 0;
  //p.x = map_.size_x*0.05;//this prints 27
  //p.y = map_.size_y*0.05; //this prints 28
  points.points.push_back(p);
  marker_pub_.publish(points);
  //ROS_INFO(" Origins x = %lf y = %lf", map_.origin_x, map_.origin_y);
}

void PRM::Plot_Lines(float x1, float y1, float x2, float y2) {
```

```
  line_list.header.frame_id = "/odom";
  line_list.header.stamp = ros::Time::now();
  line_list.ns = "points_and_lines";
  line_list.action = visualization_msgs::Marker::ADD;
  line_list.pose.orientation.w = 1.0;

  line_list.id = 2;
  line_list.type = visualization_msgs::Marker::LINE_LIST;
  line_list.scale.x = 0.04;

  line_list.color.r = 1.0;
  line_list.color.a = 1.0;


  geometry_msgs::Point p1, p2;
  p1.x = x1 * map_.scale + map_.origin_x;
  p1.y = y1 * map_.scale + map_.origin_y;
  p1.z = 0;

  p2.x = x2 * map_.scale + map_.origin_x;
  p2.y = y2 * map_.scale + map_.origin_y;
  p2.z = 0;

  line_list.points.push_back(p1);
  line_list.points.push_back(p2);

  marker_pub_.publish(line_list);
}

void PRM::Plot_path(float x1, float y1, float x2, float y2) {
  path_list.header.frame_id = "/odom";
  path_list.header.stamp = ros::Time::now();
  path_list.ns = "points_and_lines";
  path_list.action = visualization_msgs::Marker::ADD;
  path_list.pose.orientation.w = 1.0;

  path_list.id = 3;
  path_list.type = visualization_msgs::Marker::LINE_LIST;
  path_list.scale.x = 0.08;

  path_list.color.b = 1.0;
  path_list.color.a = 1.0;


  geometry_msgs::Point p1, p2;
  p1.x = x1 * map_.scale + map_.origin_x;
  p1.y = y1 * map_.scale + map_.origin_y;
  p1.z = 0;

  p2.x = x2 * map_.scale + map_.origin_x;
  p2.y = y2 * map_.scale + map_.origin_y;
  p2.z = 0;

  path_list.points.push_back(p1);
  path_list.points.push_back(p2);

  marker_pub_.publish (path_list);
}

void PRM::Plan() {
  //ROS_INFO("Planning!");
  //variable to check if got a path
  int num_neighbors = 8; //start with closest 8 neighbors, if there are more than 8 neighbors.
  if (map_received_ && !plan_created_) {
    bool done = false;
    ROS_INFO("Creating the plan!");
    //add your starting point
    stones_.milestones_x.push_back(-map_.origin_x/map_.scale);
```

```
    stones_.milestones_y.push_back(-map_.origin_y/map_.scale);
    CollisionCheck(num_neighbors);
    Publish_Points(-map_.origin_x/map_.scale, -map_.origin_y/map_.scale);
    // add your goal point
    stones_.milestones_x.push_back((4 - map_.origin_x)/map_.scale);
    stones_.milestones_y.push_back(-map_.origin_y/map_.scale);
    CollisionCheck(num_neighbors);
    Publish_Points((4 - map_.origin_x)/map_.scale, -map_.origin_y/map_.scale);
    int start_ind = 0;
    int goal_ind = 1;

    while (!done) {
      MilestoneGen(map_);

      //collision check
      CollisionCheck(num_neighbors);
      //ROS_INFO("Checked for collisions!");

      std::vector<int> path;

      //input for shortest path function - milestones, edgematrix, start and goal positions,
      //and finally a shortest path index vector and boolean to see if path detected

      int sd = 0;
      FindShortestPath(stones_.milestones_x, stones_.milestones_y, edge_matrix_, start_ind, goal_ind, path
    , sd);

      if(sd > 0) {
        ROS_INFO("Found a path!");
        double path_points[path.size()][2];
        done = true;
        plan_created_ = true;
        ROS_INFO("path size: %d", path.size());
        for(int i = 0; i < path.size(); i++)
        {
          //plot milestones
          path_points[i][0] = stones_.milestones_x[path[i]];
          path_points[i][1] = stones_.milestones_y[path[i]];
          if (i > 0) {
            Plot_path(path_points[i][0],path_points[i][1],path_points[i-1][0],path_points[i-1][1]);
          }
        }
      }
    }
    // second target
    done = false;
    ROS_INFO("Creating the second plan!");
    // add your goal point
    stones_.milestones_x.push_back((8 - map_.origin_x)/map_.scale);
    stones_.milestones_y.push_back((-4 -map_.origin_y)/map_.scale);
    CollisionCheck(num_neighbors);
    Publish_Points((8 - map_.origin_x)/map_.scale, (-4 -map_.origin_y)/map_.scale);

    start_ind = 1;
    goal_ind = stones_.milestones_x.size()-1;
    while (!done) {
      MilestoneGen(map_);

      //collision check
      CollisionCheck(num_neighbors);
      //ROS_INFO("Checked for collisions!");

      std::vector<int> path;

      //input for shortest path function - milestones, edgematrix, start and goal positions,
      //and finally a shortest path index vector and boolean to see if path detected

      int sd = 0;
```

```
      FindShortestPath(stones_.milestones_x, stones_.milestones_y, edge_matrix_, start_ind, goal_ind, path
    , sd);

      if(sd > 0) {
        ROS_INFO("Found a path!");
        double path_points[path.size()][2];
        done = true;
        plan_created_ = true;
        ROS_INFO("path size: %d", path.size());
        for(int i = 0; i < path.size(); i++)
        {
          //plot milestones
          path_points[i][0] = stones_.milestones_x[path[i]];
          path_points[i][1] = stones_.milestones_y[path[i]];
          if (i > 0) {
            Plot_path(path_points[i][0],path_points[i][1],path_points[i-1][0],path_points[i-1][1]);
          }
        }
      }
    }
    // third target
    done = false;
    ROS_INFO("Creating the third plan!");
    // add your goal point
    stones_.milestones_x.push_back((8 - map_.origin_x)/map_.scale);
    stones_.milestones_y.push_back((-map_.origin_y)/map_.scale);
    CollisionCheck(num_neighbors);
    Publish_Points((8 - map_.origin_x)/map_.scale, (-map_.origin_y)/map_.scale);

    start_ind = 1;
    goal_ind = stones_.milestones_x.size()-1;
    while (!done) {
      MilestoneGen(map_);

      //collision check
      CollisionCheck(num_neighbors);
      //ROS_INFO("Checked for collisions!");

      std::vector<int> path;

      //input for shortest path function - milestones, edgematrix, start and goal positions,
      //and finally a shortest path index vector and boolean to see if path detected

      int sd = 0;
      FindShortestPath(stones_.milestones_x, stones_.milestones_y, edge_matrix_, start_ind, goal_ind, path
    , sd);

      if(sd > 0) {
        ROS_INFO("Found a path!");
        double path_points[path.size()][2];
        done = true;
        plan_created_ = true;
        ROS_INFO("path size: %d", path.size());
        for(int i = 0; i < path.size(); i++)
        {
          //plot milestones
          path_points[i][0] = stones_.milestones_x[path[i]];
          path_points[i][1] = stones_.milestones_y[path[i]];
          if (i > 0) {
            Plot_path(path_points[i][0],path_points[i][1],path_points[i-1][0],path_points[i-1][1]);
          }
        }
      }
    }
  }
  // got a path or timed out, need to do something here now
}

```

```
PRM::PRM() {
  //Subscribe to the desired topics and assign callbacks
  ros::Subscriber pose_sub = nh_.subscribe("/indoor_pos", 1, &PRM::PoseCb, this);

  //Setup topics to Publish from this node
  ros::Publisher velocity_publisher =
    nh_.advertise<geometry_msgs::Twist>("/cmd_vel_mux/input/navi", 1);
  marker_pub_ =
    nh_.advertise<visualization_msgs::Marker>("visualization_marker", 1, true);

  //Velocity control variable
  //odom_sub_ = nh_.subscribe("/odom", 1, &PRM::OdomCb, this);
  //command_sub_ = nh_.subscribe("/mobile_base/commands/velocity", 1,
  //                                          &PRM::CommandCb, this);
  sim_ips_sub_ = nh_.subscribe("/gazebo/model_states", 1,
                                             &PRM::SimIPSCb, this);
  map_sub_ = nh_.subscribe("/map", 1, &PRM::MapCb,
                                          this);

  robot_path_.header.frame_id = "/odom";
  map_received_ = false;
  last_command_time_ = -1;
  last_measurement_time_ = -1;
  measurement_ready_ = false;
  plan_created_ = false;

  ec_ = 0; // initialize edge count
  //create a 1 by 1 matrix to start
  edge_matrix_.resize(1);  // resize top level vector
  for (int i = 0; i < 1; i++) {
    edge_matrix_[i].resize(1);  // resize each of the contained vectors
  }

  //Draw Curves
  float x = 0;// + map_.origin_x;
  float y = 0;// + map_.origin_y;
  //ROS_INFO("Input coords x = %f y = %f", x, y);
  //Publish_Points(x,y);
  //DrawCurve(1);
  //DrawCurve(2);
  //DrawCurve(4);
}
```

Appendix A.2 Path Tracking

```
purepursuit.cpp

// ////////////////////////////////////////////////////////
//
// Pure Pursuit Path Following algorithm
// Assumes that robot starting position is the same as in the PRM calculated path plan
//
// Author: Michal Kaca
//
// ///////////////////////////////////////////////////////

#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Twist.h>
#include <tf/transform_datatypes.h>
#include <gazebo_msgs/ModelStates.h>
#include <visualization_msgs/Marker.h>
#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/Path.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <math.h>

ros::Publisher pose_publisher;
ros::Publisher velocity_publisher;
ros::Publisher path_pub;
ros::Publisher given_path_pub;

geometry_msgs::Twist vel;
nav_msgs::Path givenPathMsg;
nav_msgs::Path pathMsg;

// init current index (to start following)
int current_index = 0;

// robot speed
double linearSpeed = 0.05; // m/s
double angleSpeed = 0.2;  // m/s

//init path vectors    ..... MODIFY to obtain vectors from PRM
std::vector<float> pathVectorX = {0.0, 0.3, 0.5, 0.5, 0.8, 1.0,
 1.5, 2.0, 2.5, 3.0, 3.2, 3.3,  2.9, 3.4, 4.0, 4.5, 4.5, 4.5};
std::vector<float> pathVectorY = {0.0, 0.0, 0.0, 1.0, 1.0, 1.0,
 1.1, 0.0, 1,0, 0.0, 0.0, 0.2, -0.5, 0.1, 0.0, 1.0, 2.0, 1.5};
std::vector<geometry_msgs::PoseStamped> givenVectorPath; //given input


// current robot position
float ips_yaw;
float ips_x;
float ips_y;

// Custom lookahead distance
float lookahead = 1;

//Callback function for the Position topic (SIMULATION)
void pose_callback(const gazebo_msgs::ModelStates &msg) {

    int i;
    for (i = 0; i < msg.name.size(); i++)
        if (msg.name[i] == "mobile_base")
            break;

    ips_x = msg.pose[i].position.x;
    ips_y = msg.pose[i].position.y;
    ips_yaw = tf::getYaw(msg.pose[i].orientation);
```

```
    //ROS_INFO_STREAM(" UPDATED POSES");

}

/*void pose_callback(const geometry_msgs::PoseWithCovarianceStamped & msg)
{
    //This function is called when a new position message is received
        geometry_msgs::PoseWithCovarianceStamped curpose;
    curpose.header.stamp = ros::Time::now();
    curpose.header.frame_id="/map";
    curpose.pose.pose.position = msg.pose.pose.position;
    curpose.pose.pose.orientation = msg.pose.pose.orientation;
    //republish pose for rviz
    pose_publisher.publish(curpose);
}*/

//pesudo code
/*
1. Get vector of points to follow
2. Get distances from your point to first point in vector, check if great than lookup, if not, get distance to next point and amalgamate, else set point as target and move there, record index
Small lookahead --moves quickly
Large lookahead --overshoots corners more severly
*/

// Returns linear distance between two points
float getLinearDistance (float x0, float y0, float x1, float y1){
    return pow(pow(y0 - y1, 2) + pow(x0 - x1, 2), 0.5);
}


int main(int argc, char **argv)
{
    //Initialize the ROS framework
    ros::init(argc,argv,"try_moving_bitch");
    ros::NodeHandle n;

    int step = 1;

    ros::Publisher velocity_publisher = n.advertise<geometry_msgs::Twist>("/cmd_vel_mux/input/navi", 1);

    //Subscribe to the desired topics and assign callbacks
    //ros::Subscriber pose_sub = n.subscribe("/indoor_pos", 1, pose_callback);
    ros::Subscriber pose_sub = n.subscribe("/gazebo/model_states", 1, pose_callback);

    //pose_publisher = n.advertise<geometry_msgs::PoseWithCovarianceStamped>("/vis_pos", 1, true);

    // For visualizing path
    path_pub = n.advertise<nav_msgs::Path>("/pathFlex",1,true);
    given_path_pub = n.advertise<nav_msgs::Path>("/givenPath",1,true);


    //// Generate vector path for testing:
    for (int i = 0; i < pathVectorX.size(); i++){
        geometry_msgs::PoseStamped pathPose;
        pathPose.pose.position.x = pathVectorX[i];
        pathPose.pose.position.y = pathVectorY[i];
        givenVectorPath.push_back(pathPose);
    }

    givenPathMsg.header.frame_id = "odom";
    givenPathMsg.poses = givenVectorPath;
    /// Section for testing path ends


    geometry_msgs::PoseStamped pathPose;
    std::vector<geometry_msgs::PoseStamped> vectorPath;
    pathMsg.header.frame_id = "odom";
```

```

    given_path_pub.publish(givenPathMsg); // publishes given position (provided by path planner)

/*
    // set speeds to zero
    vel.linear.x = 0; //m /s
    vel.angular.z = 0;
    velocity_publisher.publish(vel); // Publish the command velocity
*/
    //Set the loop rate
    ros::Rate loop_rate(50);    // update rate in Hz


    while (ros::ok())
    {

        loop_rate.sleep(); //Maintain the loop rate
        ros::spinOnce();   //Check for new messages --> processes callbacks

        // init transitional variables
        float radius;
        float newVehicleDirection;
        int rotateCounter = 0;

        // MAIN LOOP HERE
        switch(step) {
            // Calculates optimal path and point to follow for Robot
            case 1:{
                // Gets total path
                // Basically algo below checks for distance between points, (between point 0 to point 1, point 1 to point 2, etc... and once lookahead distance is reached, we go to that point)
                float totDistance = 0;
                float workingDistance;
                bool first_loop = true;
                while (1){
                    if (first_loop){
                        // ends path follower
                        if (current_index == pathVectorY.size() - 1){
                            ROS_INFO_STREAM("ENDING PROGRAM... Index: " << current_index);
                            return 0;
                        }
                        workingDistance = getLinearDistance(ips_x, ips_y, pathVectorX[current_index], pathVectorY[current_index]);
                        ROS_INFO_STREAM("First Loop working distance:" << workingDistance);
                    }
                    else{
                        workingDistance = getLinearDistance(pathVectorX[current_index-1], pathVectorY[current_index-1], pathVectorX[current_index], pathVectorY[current_index]);
                    }
                    if (totDistance + workingDistance < lookahead) {
                        totDistance = totDistance + workingDistance;
                        current_index++;
                    }
                    else{
                        if (first_loop){
                            ROS_INFO_STREAM("THIS IS SOME BULLLLLLLSHIT: lookahead distance is TOO SMALL");
                        }
                        break;
                    }
                    first_loop = false;
                }

                // ends path follower
                if (current_index == pathVectorY.size()){
                    ROS_INFO_STREAM("ENDING PROGRAM... Index: " << current_index);
                    return 0;
                }
```

```
            // get curvature  (1/radius)
            float linDist = getLinearDistance(ips_x, ips_y, pathVectorX[current_index],pathVectorY[current_index]);
            float curvature = 2*(pathVectorY[current_index] - ips_y)/ (linDist*linDist);
            radius = 1.0/curvature;

            ROS_INFO_STREAM("targetY:" << pathVectorY[current_index]
                << " targetX: " << pathVectorX[current_index]);
            ROS_INFO_STREAM("current Y Pos: " << ips_y << "current X Pos: " << ips_x);

            //ROS_INFO_STREAM("current_index:" << current_index);

            //get circle center
            float xMed = (ips_x + pathVectorX[current_index])/2;
            float yMed = (ips_y + pathVectorY[current_index])/2;
            //ROS_INFO_STREAM("xMed:" << xMed << " yMed: " << yMed);

            // here we obtain the larger circle (so we add medians to trig method)
            float xCir = xMed + std::sqrt(radius*radius - pow(linDist/2,2))*
            (ips_y-pathVectorY[current_index])/linDist;
            float yCir = yMed + std::sqrt(radius*radius - pow(linDist/2,2))*
            (pathVectorX[current_index] - ips_x)/linDist;

            ROS_INFO_STREAM("xCir:" << xCir << " yCir: " << yCir);
            ROS_INFO_STREAM("radius:" << radius);

            //get radius direction (angle)
            float radiusDir =  atan2((yCir - ips_y),(xCir - ips_x));
            float newVehicleDirection = radiusDir - M_PI/2;

            /*// Find starting angle based on cosine law and trigonometry
            // cos law: a^2 = b^2 +c^2 -2abcosA
            float middleAngle = radiusDir - M_PI/2;
            float a = getLinearDistance(xCir, yCir, ips_x, ips_y);
            float b = getLinearDistance(xCir, yCir, pathVectorX[current_index], pathVectorY[current_index]);
            float differenceAngle = acos( (linDist*linDist - a*a - b*b)/(-2.0*a*b) );
            ROS_INFO_STREAM("diff Angle: " << differenceAngle);
            newVehicleDirection = middleAngle - differenceAngle/2.0;*/
            ROS_INFO_STREAM("radiusDir:" << radiusDir);
            ROS_INFO_STREAM("newVehicleDirection:" << newVehicleDirection);
            ROS_INFO_STREAM("currentVehicleOrientation:" << ips_yaw);
            ROS_INFO_STREAM(" ");

            // Set angular direction
            if (ips_yaw > newVehicleDirection){
                angleSpeed = angleSpeed*(-1);
            }
            step = 2;
            break;
        }

        case 2:{
            // set robot in correct starting orientation
            if (std::abs(ips_yaw - newVehicleDirection) > 0.08){
                // Set angular direction
                if (ips_yaw < newVehicleDirection){
                    angleSpeed = angleSpeed*(-1);
                }
                vel.angular.z = angleSpeed; // set angular speed for rotating robot to its target
                ROS_INFO_STREAM("anglespeed IS SET:" << angleSpeed);
            }
            step = 3;
            break;
        }

        case 3:{
            if (std::abs(ips_yaw - newVehicleDirection) > 0.08){
```

```
                    vel.angular.z = angleSpeed;
                }
                else{
                    ROS_INFO_STREAM("ANGLE SET TO: " << ips_yaw);
                    vel.angular.z = 0;
                    rotateCounter = 0;
                    step = 4;
                }
                rotateCounter++;
                if (rotateCounter > 600000)
                {
                    ROS_INFO_STREAM(" COUNTER HIT 60, exiting");
                    return 0;
                }
                break;
            }

            case 4:{
                ROS_INFO_STREAM("Mans stopped spinning. Moving towards target");

                // Set robot to start following circular path for allotted amount of time
                vel.angular.z = linearSpeed/radius; // stop spinning                              MIGHT
    HAVE TO BE NEGATIVE OR ABSOLUTE  ---> NEEDS TESTING
                vel.linear.x = linearSpeed;
                //ROS_INFO_STREAM("PUBLISHED ANGULAR SPEED IS: " << linearSpeed/radius);
                step = 5;
                break;
            }

            case 5:{
                if (getLinearDistance(ips_x, ips_y, pathVectorX[current_index], pathVectorY[current_index
    ]) < 0.15){
                    // stop robot to prepare for next calculation
                vel.angular.z = 0; // stop spinning
                vel.linear.x = 0;
                ROS_INFO_STREAM("STEP 5 COMPLETE");
                step = 1;
                }
                /*else
                    ROS_INFO_STREAM("STEP 5 IS HUSTTTTTTLING");*/
            }
        }
                        ROS_INFO_STREAM(step);

        velocity_publisher.publish(vel); // Publish the command velocity*/


        //ROS_INFO_STREAM("Re-running loop");


        // update pose path for rviz VISUALIZATION
        pathPose.pose.position.x = ips_x;
        pathPose.pose.position.y = ips_y;
        // create quaternion from yaw angle
        geometry_msgs::Quaternion msgQuat;
        msgQuat = tf::createQuaternionMsgFromRollPitchYaw(0, 0, ips_yaw);
        pathPose.pose.orientation = msgQuat;
        vectorPath.push_back(pathPose);
        pathMsg.poses = vectorPath;
        path_pub.publish(pathMsg);
    }

    return 0;
}


 /*
    // QUICK TEST:
```

```
    int supernoodecount = 0;
    geometry_msgs::Twist velTest;
    while (supernoodecount < 50){
        velTest.linear.x = linearSpeed*2;
        velTest.angular.z = angleSpeed*2;
        velocity_publisher.publish(velTest);
        ros::Duration(0.2).sleep();
    }
    return 0;*/
```