Faculty of Engineering
Department of Mechanical and Mechatronics Engineering

LAB 2: MAPPING AND LOCALIZATION

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 describes the process of implementing the mapping and localisation functionality as described in Lab 2. It comprises of two main sections - Theory and Implementation, and Results and Discussion for both parts.

2 Theory and Implementation

2.1 Mapping System

A custom-made gazebo world that was created during Lab 1, was used to implement the mapping algorithm. The map begins as an occupancy gridw ith each cell being initialised to an unknown state, and an equivalent probability of 0.5. The mapping algorithm first reads the 2D laser values obtained from the /scan topic (derived from point cloud data obtained by the XBox Kinect Sensor). This vector of laser values contains the angle of the laser and the distance that the laser measured (with respect to the robot). If an empty distance value is measured then it is disregarded, as no object had been detected. Furthermore, if unrealistic distance values threshold were received the value is also discarded. Each pair of laser range and angle readings are then converted to absolute coordinates. Using the absolute x and y points of the robot origin and obstacle, Bresenham's raytracing algorithm was used to determine which global grid cells to update, developing a vector of x and y coordinates. Since the raytracing algorithm gives a vector of x and y points from the robot's global position up to and including the obstacle's location, all grid cell locations from the resulting vector are given a occupancy probability of 0.4, except the furthest one from the robot, which was given a probability of 0.6.

The log odds formula is used to correctly update the occupancy grid map with multiple measurements. This allows for the addition of the amalgamating probabilities instead of multiplication which is much more computationally-heavy. The algorithm was tested using by manually controlling the turtlebot and mapping the lab area.

Important parameters included the size of the occupancy grid (= 20m x 20m), and the resolution of the occupancy grid (=0.1m) which indicates the size of each cell found in the grid. In our case, this meant that the size of each cell of the occupancy grid was 0.1 x 0.1 meters. The grid cell resolution parameter was tested for values between 0.02 and 0.1 meters. Efficiency of the algorithm was improved by implementing an inverse measurement model which incorporated the Bresenham algorithm.

The next phase of the lab included utilising a live turtlebot. To make the groups mapping algorithm compatible with the live turtlebot, the node subscribed to the the "/indoor_pos" topic instead of the "/gazebo/modelstates" topic. This change in topic resulted in the message type changing from a "ModelState" message to a "PoseWithCovarianceStamped" message. ROSbag was run on the robot in order to allow for additional testing of the algorithm at a remote location. This was crucial for generating plots, graphs, and other information necessary to complete the lab in a realistic time frame.

During the lab, the team overcame many issues. Software included segmentation faults that occurred from incorrectly utilized vectors, g++ compiler incompatibility issues, and CMake library incompatibility. Algorithmic problems included inconsistent ROS topic implementations, incorrect coordinate transformations, incorrect indexing of the occupancy grid, and hanging reference frames. Also, the algorithm initially did not function on the live turtlebot due to inconsistent reference frames that prevented the robots position from updating properly. Additionally, the algorithm did not account for errors in the IPS system.

2.2 Localization

The state estimation of the Turtlebot is achieved using a particle filter. This implementation of a particle filters uses the motion model of the turtlebot and the degraded IPS values in order to estimate the state (x-y coordinates, orientation) of the turtlebot. Particles are distributed uniformly across a previously generated map and move in accordance with the motion model, and the odometry of the turtlebot. The values for the state of the particle are compared with the IPS readings and are then weighed based on their likelihood for being correct. The particles are then re-sampled at the given weights in order to create a new set of particles that are more likely to be the state of the turtlebot. This process is repeated at each instance that the turtlebot moves and a small cloud of particles show the most likely states of the turtlebot at each step.

The particle filter node is subscribed to the "/map" topic, and upon receiving a "nav_msgs::OccupancyGrid" message the message is converted to a map_t struct. This struct holds information about the map's origin, dimensions, and scale. It also contains the information about whether a cell in the grid is occupied or not. The map object is then passed on to the uniform particle distribution generation function.

The initial particle distribution is handled by the function UniformPoseGenerator(). This function generates random pose guesses uniformly across the map. The X and y coordinates are determined by taking the minimum value in that direction, and adding a percentage of the total distance in that direction. The value for the orientation is randomly selected between 0 and π in a similar fashion. The map is then checked to see whether the particle is in an occupied space. If the grid is not currently occupied by an object, then the particle is added to the list of all the particles. This is repeated until all the desired number of particles are generated.

The next step in the particle filter is the propagation of the particles. Each time a command is sent to the turtlebot to move, the particles also move by the same distance. This is done using the equation $distance = velocity * time$ and transforming from the body frame to the inertial frame. In the simulation the /mobile$_{b}ase/commands/velocitytopicisusedtogetthecommandedvelocityw$

The propagated particles are assigned weights based on their proximity to the measured values obtained from the IPS. This is done using a multivariate normal distribution function, GetProb(), that takes in a particle's values for the position and orientation, the readings of the IPS, and the covariance. This returns a non-normalised weight for the particle. The total weights are summed up, and then each individual particle's weight is divided by the total in order to normalise the weights.

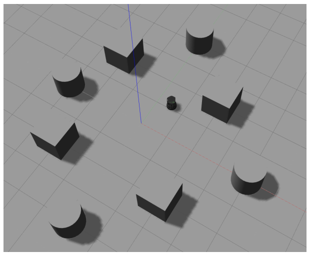
Re-sampling the particles involved generating a cumulative distribution using the weights of each of the particles using an algorithm known as Universal Stochastic Sampling. It was implemented within a function and called after the particles had been weighed. This step of the particle simulates a survival of the fittest by killing off particles that least align with the sensor measurements and duplicating particles that align with the sensor measurements. The amount of duplicate particles created is proportional to the weight of that particle. The regenerated particle are initialised with equal weights.

To publish an estimate of the robot's pose, the average value of the x and y components of the position of each particle was calculated. The estimated yaw was obtained by finding the median of the yaw components of each particle.

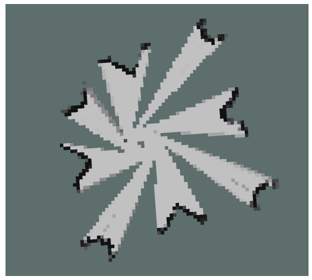
3 Results and Discussion

3.1 Mapping

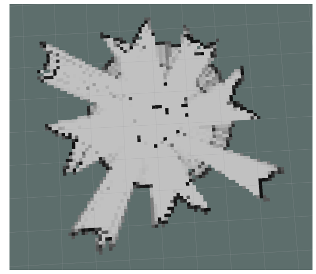
The mapping portion was developed with varying resolution - 10cm, 5cm, and 2cm. This would allow for a study to see which resolution yielded the optimal results given trade-offs between computation and accuracy. For the simulation, however, a verification was only required to see if the mapping algorithm was working as expected, therefore a 10cm resolution was used to limit computation. This is shown below in section 3.1. This map was generated by driving the robot using tele-op commands learned from lab 1.



(a) Simulation Environment



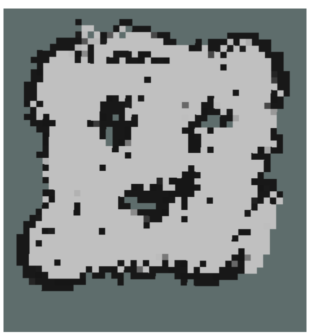
(b) Simulated Mapping Results - First Pass



(c) Simulated Mapping Results - Second Pass

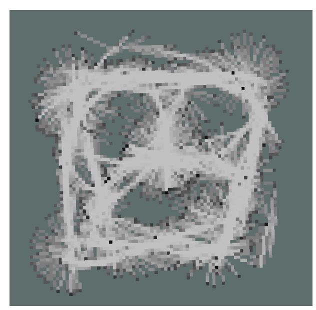
The second pass introduces a more accurate representation of the simulated environment - namely, filling out the unoccupied space. There is, however, still uncertainty regarding occupancy for cells immediately between and beyond the obstacles, since the mapping algorithm is directed to ignore large range values coming from readings between obstacles. Notably, there is still noise or uncertainty regarding the map output even from the simulated environment, since the map shows there are some occupied cells near the centre, where there are no obstacles.

Below is the result of conducting three passes through the physical environment, shown in the three resolutions as mentioned above.



(a) Physical Mapping Results - 10cm Resolution

(b) Physical Mapping Results - 5cm Resolution



(c) Physical Mapping Results - 2cm Resolution

Figure section 3.1 shows that while a 10cm resolution map is most filled, a higher resolution of 2cm shows the most distinction of the obstacle boundaries. A compromise is between these two, with a resolution of 5cm, which seems to be the most accurate of the three maps (most filled out while maintaining the shape of the obstacles).

The figures above show that the Xbox Kinect sensor and IPS are affected by much more noise that is not accounted for in simulation. The edges of the map and obstacles are soft and unclear. Also, noise is evident as there isn't clear distinction of obstacle edges due to reflection, noise, and other robots in the vicinity yielding false information to the mapping node. Simulation results differ because simulated lasers are more accurate and precise, and the gazebo world did not have other turtlebots and people moving through. The noise from the laser and IPS sensors impact the final results since rather than suggesting definite cells that are occupied, the laser readings bounce around rather than giving a steady value, and the IPS measurements does not give a static value if the robot was static. The main way to improve the results given the hardware and setup was to do multiple passes of the map to increase the probability of occupied cells and remove false positives. The main limitation was the update speed of the IPS sensor and the nature of the noise coming from the Kinect scanner and IPS sensor. These two hardware limitations made it difficult to properly develop a map with clean distinctions between occupied and unoccupied areas. Finally, an uncontrollable part of this mapping part of the lab was the nature of it - the map was naturally dynamic due to students and robots moving around the area of interest, making data collection inaccurate and unrepresentative of the static map.

The gazebo pose updates at 20hz which created a pose mismatch with laser calculations when the algorithm switched to updating from the IPS messages which update at 1hz. Consequently, the mapping frequency was changed to operate at 1hz, the slower frequency, to prevent improper matching of the laser scan's with the robot's pose. Another improvement was tightening the maximum and minimum and maximum range values on which the algorithm operates, to reduce disturbance and false positives. Finally, to improve the computational speed of the mapping algorithm, a subset of the points from the Bresenham raytracing algorithm were updated in the occupancy grid rather than the entire set of ray points - particularly, only the points close to the obstacle. Some suggested improvements involve noise filtering (either hardware or software) and mapping in a static environment without other people or other objects traversing during the mapping procedure. Finally, if the noise filtering and improved hardware is implemented, a more accurate map could be generated by increasing the resolution to 1 centimetre per grid cell. Given the immediate circumstances of a faulty IPS sensor and noisy scanner, an Extended Kalman Filter can be applied since the motion model and control inputs of the robot are known - this can help develop a better estimation of the robot's position, given the IPS measurements.

3.2 Localization

To test the localization within the simulation environment, the robot was driven in a one meter square and the path moved through by the robot was visualised. The mean squared error of the position and orientation for different number of particles is given below.

Number of Particles	Position Error	Orientation Error
50	1.1313	6.3029
500	0.0052	0.0092
5000	0.0026	0.0054

It was noticed that as the number of particles was increased the accuracy of the particle filter improved. Through further testing it was determined that the amount of improvements did not reduce as significantly after increasing the particles beyond 300 particles. A plot of the path is provided for the case with 5000 particles.

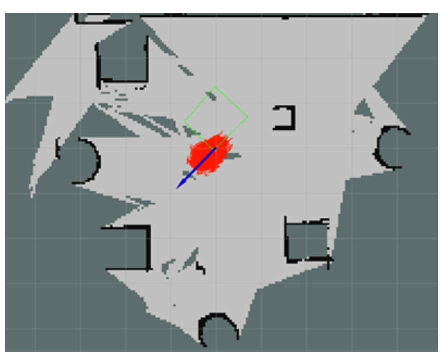


Figure 3-3. Simulated Particle Filter Path

When running the particle filter on bagged data no true ground truth data was available due to the fact that the IPS sensor has noise. The path obtained from the bagged data tracked the real robot very closely. To improve the particle filter it is recommended that the re-sampling step be changed to an implementation of the Stochastic Universal Sampling. This algorithm runs in O(1) in comparison to the current re-sampling algorithm which runs in O(n).

Appendix A

Appendix A.1 Mapping Code

```
//  ////////////////////////////////////////////////////////////
//
// turtlebot_example_node_lab2_mapping.cpp
// This file is the mapping node used for developing occupancy grid.
// It is used in Lab 2 of MTE 544
//
//
// Author: Lalit Lal & Michael 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 <sensor_msgs/LaserScan.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
//#include <eigen3/Eigen/Core>

//using namespace Eigen;

ros::Publisher pose_publisher;
ros::Publisher marker_pub;

//IPS Data
double ips_x;
double ips_y;
double ips_yaw;

//Laser Data
float angle_min;        // start angle of the scan [rad]
float angle_max;        // end angle of the scan [rad]
float angle_increment;  // angular distance between measurements [rad]
float scan_time; //time between scans (we can check for this to modify the rate of updates)
float range_min;
float range_max;

//laser rangedata:
std::vector <float> r;

//laser constants
//float alpha = 1;
//float beta = 0.05;

//Map Properties
static const int RES_X = 20; //20m x direction map
static const int RES_Y = 20; //20m y direction map
static const float cell_size_in_m = 0.1; //10 cm per cell
static float cell_not_occ = 0.4;
static float cell_occ = 0.6;

static const int M = RES_X/cell_size_in_m;
static const int N = RES_Y/cell_size_in_m;

double m[M][N] = {50.0};
double L0[M][N] = {0};

double L[M][N] = {0};
```

```
//store position of robot
std::vector <double> robot_pose_x;
std::vector <double> robot_pose_y;
std::vector <double> robot_pose_yaw;

std::vector<int8_t> final_array;


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 inverse_meas_bres(float theta, float r,
 std::vector <int> &vector_x, std::vector <int> &vector_y, std::vector <float> &vector_prob)
{
    //Steps:
    // Get Range, bearing
    // convert to global coordinates
    //Run occupancy grid bayes filter
    if(std::isnan(r)) return;
    int x0 = 0;
```

```
    int y0 = 0;
    int x1 = 0;
    int y1 = 0;

    //int r_max = 40/cell_size_in_m;
    //int r_min = 0/cell_size_in_m;
    r = r/cell_size_in_m;

    x0 = ips_x/cell_size_in_m + (M/2);
    y0 = ips_y/cell_size_in_m + (N/2);

    if(x0 > M-1 || x0 < 0)
    {
        //ROS_INFO("X out of bounds: %d", x0);
        return;
    }
    if(y0 > N-1 || y0 < 0)
    {
        //ROS_INFO("Y out of bounds: %d", y0);
        return;
    }

    if(r > (range_max/cell_size_in_m) || (r < range_min/cell_size_in_m))
    {
        //ROS_INFO("Range out of bounds: %f", r);
        return;
    }

    float rel_x = r * cos(theta);
    float rel_y = r * sin(theta);

    float endpt_x = (rel_x * cos(ips_yaw)) - (rel_y * sin(ips_yaw)) + x0;
    float endpt_y = (rel_x * sin(ips_yaw)) + (rel_y * cos(ips_yaw)) + y0;

    x1 = endpt_x;// + M/2;
    y1 = endpt_y;// + N/2;

    if(x1 > M-1 || x1 < 0)
    {
        ROS_INFO("x1 out of bounds: %d", x1);
        return;
    }
    if(y1 > N-1 || y1 < 0)
    {
        ROS_INFO("y1 out of bounds: %d", y1);
        return;
    }


    bresenham(x0, y0, x1, y1, vector_x, vector_y);
    //ROS_INFO("VALID DATA");

    for(std::vector<int>::size_type i = 0; i < vector_x.size(); i++)
    {
        if(i == (vector_x.size() - 1) && r > (range_min/cell_size_in_m) && r < (range_max/cell_size_in_m))
        {
            //ROS_INFO("Occupied Cell!");
            vector_prob.push_back(cell_occ);
        }
        else
        {
            vector_prob.push_back(cell_not_occ);
        }
    }

}
void mapping()
{
```

```
    //GO THROUGH EACH CELL
    // CHECK IF CELL's (thorugh relative measurements) are in bearing field
    int measurement_size = (angle_max - angle_min)/angle_increment;
    float measL[M][N] = {0};

    std::vector <int> vector_x;
    std::vector <int> vector_y;
    std::vector <float> vector_prob;
    // find which cells to update

    for(int i = 0; i < measurement_size; i++)
    {
        vector_x.clear();
        vector_y.clear();
        vector_prob.clear();
        //ROS_INFO("GOT HERE 1");
        float theta = i*angle_increment;
        if(!std::isnan(r[i]))
        {
            inverse_meas_bres(theta,r[i], vector_x, vector_y, vector_prob);  // returns points AND their probabilities (0-1)
            //ROS_INFO("Measurement size: %ld %ld %ld", vector_x.size(), vector_y.size(), vector_prob.size());
            int start = vector_x.size() > 50 ? vector_x.size()-25 : 0;
            for(int j = start; j < vector_x.size(); j++)
            {
                int ix = vector_x[j];
                int iy = vector_y[j];
                float il = vector_prob[j];

                //update log odds
                L[ix][iy] = L[ix][iy] + log(il/(1-il)) - L0[ix][iy];
                double new_prob = exp(L[ix][iy])/(1 + exp(L[ix][iy]))*100;

                m[ix][iy] = new_prob;
                if(new_prob > 100) new_prob = 100;
                if(new_prob < 0) new_prob = 0;

                int insert_index = (ix * N) + iy;
                final_array[insert_index] = new_prob;
                //ROS_INFO("Assigned Prob: %f", m[ix][iy]);
                //ROS_INFO("x: %d y: %d prob: %f", ix, iy, m[ix][iy]);
            }

        }

    }

    //clear data from previous inputs
    vector_x.clear();
    vector_y.clear();
    vector_prob.clear();
    r.clear();

}

void init_final_occupancy_grid(nav_msgs::OccupancyGrid &msg)
{
    msg.header.frame_id = "odom";
    msg.info.resolution  = cell_size_in_m; // each cell is 0.1m [m/cell]
    msg.info.width = M; // in cells
    msg.info.height = N; // in cells
    geometry_msgs::Pose pose;
    pose.position.x = -RES_X/2;
    pose.position.y = -RES_Y/2;
    pose.position.z = 0;
    msg.info.origin = pose; // [m, m, rad]
}
```

```
void update_final_grid(nav_msgs::OccupancyGrid &msg)
{
    msg.data = final_array;
}

void laser_callback(const sensor_msgs::LaserScan &msg)
{
    size_t RANGE_SIZE = sizeof(msg.ranges)/sizeof(msg.ranges[0]);

    angle_min = msg.angle_min;
    angle_max = msg.angle_max;
    angle_increment = msg.angle_increment;
    range_max = msg.range_max;
    range_min = msg.range_min;
    //ROS_INFO("increment: %f", angle_increment);

    for(std::vector<int>::size_type i = 0; i < RANGE_SIZE; i++)
    {
        //ROS_INFO("Getting range: %f", msg.ranges[i]);
        //ROS_INFO("min r: %f, max r: %f", range_min, range_max);
        r.push_back(msg.ranges[i]);
    }

    //ROS_INFO("Range size: %ld\n", RANGE_SIZE);
    //ROS_INFO("Min Angle: %f\n", angle_min);
    //ROS_INFO("Max Angle: %f\n", angle_max);

    mapping();

}

//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; //cell_size_in_m;
    ips_y = msg.pose[i].position.y; //cell_size_in_m;
    ips_yaw = tf::getYaw(msg.pose[i].orientation);

    //ROS_INFO("X: %lf\n", ips_x);
    //ROS_INFO("Y: %lf\n", ips_y);
    //ROS_INFO("Theta: %lf\n", ips_yaw);

    //store robot poses for plotting path
    //robot_pose_x.push_back(ips_x);
    //robot_pose_y.push_back(ips_y);
    //robot_pose_yaw.push_back(ips_yaw);
}*/

//Callback function for the Position topic (LIVE)
void pose_callback(const geometry_msgs::PoseWithCovarianceStamped &msg)
{

    ips_x = msg.pose.pose.position.x; // Robot X psotition
    ips_y = msg.pose.pose.position.y; // Robot Y psotition
    ips_yaw = tf::getYaw(msg.pose.pose.orientation); // Robot Yaw
    ROS_DEBUG("pose_callback X: %f Y: %f Yaw: %f", ips_x, ips_y, ips_yaw);
}

//Callback function for the map
void map_callback(const nav_msgs::OccupancyGrid &msg) {
    //This function is called when a new map is received

```

```
    //you probably want to save the map into a form which is easy to work with

}

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

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

    //Setup topics to Publish from this node
    ros::Publisher velocity_publisher = n.advertise<geometry_msgs::Twist>("/cmd_vel_mux/input/navi", 1);
    //pose_publisher = n.advertise<geometry_msgs::PoseStamped>("/pose", 1, true);
    //marker_pub = n.advertise<visualization_msgs::Marker>("visualization_marker", 1, true);
    ros::Publisher map_publisher = n.advertise<nav_msgs::OccupancyGrid>("/map", 1, true);

    //Velocity control variable
    geometry_msgs::Twist vel;
    nav_msgs::OccupancyGrid occ_grid;
    init_final_occupancy_grid(occ_grid);
    //m = {50};

    //Set the loop rate
    ros::Rate loop_rate(1); //20Hz update rate
    for(int a = 0; a < M; a++)
    {
        for(int b = 0; b < N; b++)
        {
            final_array.push_back(-1);
        }
    }

    while (ros::ok()) {

        //Main loop code goes here:
        //vel.linear.x = 0.2;  // set linear speed
        //vel.angular.z = 0.5; // set angular speed

=         // update map
        update_final_grid(occ_grid); // generate occupancy grid message

        //velocity_publisher.publish(vel); // Publish the command velocity
        map_publisher.publish(occ_grid);  //publish grid to /map topic
        loop_rate.sleep(); //Maintain the loop rate
        ros::spinOnce();   //Check for new messages
    }

    return 0;
}
```

Appendix A.2   Localization Code

```
#ifndef PARTICLE_FILTER_H
#define PARTICLE_FILTER_H

#include <algorithm>
#include <Eigen/Core>
#include <Eigen/Dense>
#include <Eigen/Eigenvalues> // header file
#include <gazebo_msgs/ModelStates.h>
#include <geometry_msgs/PoseArray.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <geometry_msgs/Twist.h>
#include <math.h>
#include <nav_msgs/OccupancyGrid.h>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/Path.h>
#include <random>
#include <ros/ros.h>
#include <stdlib.h>
#include "tf/transform_datatypes.h"
#include <tf2/LinearMath/Quaternion.h>
#include <queue>

typedef struct {
  double v[3];
  double weight;
} particle_t;

// Description for a single map cell.
typedef struct {
  // Occupancy state (true = unknown or occupied, false = empty)
  bool occupied;
} cells_t;

// Description for a map
typedef struct {
  // Map origin; the map is a viewport onto a conceptual larger map.
  double origin_x, origin_y;
  // Map scale (m/cell)
  double scale;
  // Map dimensions (number of cells)
  int size_x, size_y;
  // The map data, stored as a grid
  cells_t *cells;
} map_t;

class ParticleFilter {
  private:
    ros::Publisher pf_cloud_publisher_;
    ros::Publisher pf_pose_publisher_;
    ros::Publisher path_publisher_;
    ros::Subscriber odom_sub_;
    ros::Subscriber sim_ips_sub_;
    ros::Subscriber map_sub_;

    nav_msgs::Path robot_path_;
    double ips_x_;
    double ips_y_;
    double ips_yaw_;
    bool map_received_;
    map_t map_;
    std::queue<std::array<double, 3> > pose_deltas_;
    double last_command_time_;
    std::vector<particle_t> particles_;
    geometry_msgs::PoseArray particles_msg_;
```

```
    int num_of_particles_;
    Eigen::Vector3d measurements_;
    double dt_;
    std::default_random_engine generator_;
  public:
    ros::NodeHandle nh_;
    ParticleFilter();
    void ConvertMap(const nav_msgs::OccupancyGrid &map_msg);
    void UniformPoseGenerator();
    bool PropagateParticles();
    bool WeighParticles();
    void ResampleParticles();

    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 PublishPoseArray();
    void PublishPosePath();
    void Estimate();
};
#endif

#include "particle_filter.h"

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

double GetProb(const Eigen::Vector3d& x, const Eigen::Vector3d& mean,
               const Eigen::Matrix3d& covar) {
  // Multivariate Gaussian distribution
  using namespace Eigen;
  //std::cout << x << std::endl <<  mean << std::endl << covar << std::endl;
  VectorXd quadform = (x - mean).transpose() * covar.inverse() * (x - mean);
  double quad = quadform(0);
  return std::exp(-0.5 * quad);
}

void ParticleFilter::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_.size_x / 2) * map_.scale;
  map_.origin_y = map_msg.info.origin.position.y
                  + (map_.size_y / 2) * map_.scale;

  // 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;
  }
}

// Generates random pose guesses uniformly across map
void ParticleFilter::UniformPoseGenerator() {
  double min_x, max_x, min_y, max_y;

  min_x = -(map_.size_x * map_.scale)/2.0 + map_.origin_x;
  max_x = (map_.size_x * map_.scale)/2.0 + map_.origin_x;

  min_y = -(map_.size_y * map_.scale)/2.0 + map_.origin_y;
```

```
  max_y = (map_.size_y * map_.scale)/2.0 + map_.origin_y;

  particle_t p;
  p.weight = 1.0/(float)num_of_particles_;
  int n = 0;
  ROS_INFO("Generating new uniform particles");
  while(n < num_of_particles_) {
    p.v[0] = min_x + ((double)rand() / (double)RAND_MAX) * (max_x - min_x);
    p.v[1] = min_y + ((double)rand() / (double)RAND_MAX) * (max_y - min_y);
    p.v[2] = ((double)rand() / (double)RAND_MAX) * 2 * M_PI - M_PI;
    // Check that it's a free cell
    int x_pos,y_pos;
    x_pos = (floor((p.v[0] - map_.origin_x) / map_.scale + 0.5) + map_.size_x / 2);
    y_pos = (floor((p.v[1] - map_.origin_y) / map_.scale + 0.5) + map_.size_y / 2);
    if(map_.cells[GetMapIndex(map_, x_pos,y_pos)].occupied == false) {
      particles_.push_back(p);
      n++;
    }
  }
  ROS_INFO("Generated new uniform particles");
}

void ParticleFilter::PublishPoseArray() {
    for (int i = 0; i < num_of_particles_; i++) {
        particles_msg_.poses[i].position.x = particles_[i].v[0];
        particles_msg_.poses[i].position.y = particles_[i].v[1];
        particles_msg_.poses[i].position.z = 0;
        tf2::Quaternion q;
        q.setRPY(0, 0, particles_[i].v[2]);
        particles_msg_.poses[i].orientation.w = q.getW();
        particles_msg_.poses[i].orientation.x = q.getX();
        particles_msg_.poses[i].orientation.y = q.getY();
        particles_msg_.poses[i].orientation.z = q.getZ();
        // ROS_INFO("particles: %lf %lf %lf ",poses[i].v[0], poses[i].v[1], poses[i].v[2]);
    }
    pf_cloud_publisher_.publish(particles_msg_);
}

bool ParticleFilter::PropagateParticles() {
  std::normal_distribution<double> x_noise(0, 1);
  std::normal_distribution<double> theta_noise(0, 0.6);
  double pose_delta[3] = {};

  if (pose_deltas_.empty()) {
    return false;
  }

  while (!pose_deltas_.empty()) {
    pose_delta[0] = pose_deltas_.front()[0];
    pose_delta[1] = pose_deltas_.front()[1];
    pose_delta[2] = pose_deltas_.front()[2];
    pose_deltas_.pop();
    for (int i = 0; i < particles_.size(); i++) {
      double v  = pose_delta[0] + x_noise(generator_) * dt_;
      particles_[i].v[0] += v * cos(particles_[i].v[2]);
      particles_[i].v[1] += v * sin(particles_[i].v[2]);
      particles_[i].v[2] += pose_delta[2];
      particles_[i].v[2] += theta_noise(generator_) * dt_;
      if (particles_[i].v[2] > M_PI) {
        particles_[i].v[2] -= 2*M_PI;
      }
      else if (particles_[i].v[2] < -M_PI) {
        particles_[i].v[2] += 2*M_PI;
      }
    }
  }
  return true;
}
```

```
void ParticleFilter::PublishPosePath() {
  geometry_msgs::PoseStamped average;
  average.header.frame_id = "/odom";
  std::vector<double> angles;

  for (int i = 0; i < particles_.size(); i++) {
    average.pose.position.x += particles_[i].v[0] * particles_[i].weight;
    average.pose.position.y += particles_[i].v[1] * particles_[i].weight;
    int num = particles_[i].weight * 10000;
    if (num > 1) {
      while(num-- > 0) {
        angles.push_back(particles_[i].v[2]);
      }
    }
  }
  if(angles.size() > 2) {
    std::sort(angles.begin(), angles.end());
    tf2::Quaternion q;
    q.setRPY(0, 0, angles[angles.size()/2]);
    average.pose.orientation.w = q.getW();
    average.pose.orientation.x = q.getX();
    average.pose.orientation.y = q.getY();
    average.pose.orientation.z = q.getZ();
  }
  robot_path_.poses.push_back(average);
  pf_pose_publisher_.publish(average);
  path_publisher_.publish(robot_path_);
}

bool ParticleFilter::WeighParticles() {
  double total  = 0.0;
  double weight_total  = 0.0;

  if (fabs(measurements_(2)) <= 0.00001) {
    ROS_INFO("Jumped ship");
    return false;
  }
  Eigen::Matrix3d cov;
  cov << 0.1, 0, 0,
         0, 0.1, 0,
         0, 0, 0.1;
  for (int i = 0; i < particles_.size(); i++) {
    Eigen::Vector3d mean;
    mean << particles_[i].v[0], particles_[i].v[1], particles_[i].v[2];
    // ROS_INFO("WEIGHED particles: %lf %lf %lf ",particles[i].v[0],
    //          particles[i].v[1], particles[i].v[2]);
    //ROS_INFO("before: %lf %lf ", measurements_(2), mean(2));
    /*
    double ang_diff = measurements_(2) - mean(2);
    mean(2) = std::min(fabs(ang_diff), 2*M_PI -fabs(ang_diff));
    if (ang_diff < 0) {
      mean(2) *= -1;
    }
    measurements_(2) = 0;
    */
    particles_[i].weight = GetProb(measurements_, mean, cov) * 1000000;
    //ROS_INFO("mean: %lf %lf %lf ", mean(0), mean(1), mean(2));
    //ROS_INFO("mesu: %lf %lf %lf ", measurements_(0), measurements_(1),
    //         measurements_(2));
    total += particles_[i].weight;
  }

  for (int i = 0; i < particles_.size(); i++) {
    particles_[i].weight /= total;
    weight_total += particles_[i].weight;
    //ROS_INFO("Weight of particles = %.17g", particles_[i].weight);
  }
```

```
  //ROS_INFO("total: %f", weight_total);
  return true;
}

void ParticleFilter::ResampleParticles() {
  double r;
  double count_inv;
  double c[num_of_particles_] = {};
  std::vector<particle_t> resampled_poses;
  resampled_poses.clear();
  particle_t single;

  c[0] = particles_[0].weight;
  for(int i=1; i < num_of_particles_; i++) {
    c[i] = c[i-1] + particles_[i].weight;
  }
  for(int i = 0; i < num_of_particles_; i++) {
    r = rand() / double(RAND_MAX);

    for (int j = 0; j < num_of_particles_; j++) {
      if (r <= c[j]) {
        single.v[0] = particles_[j].v[0];
        single.v[1] = particles_[j].v[1];
        single.v[2] = particles_[j].v[2];
        single.weight = 1.0 / (double)num_of_particles_;
        resampled_poses.push_back(single);
        //ROS_INFO("select %lf %lf", r, );d
        break;
      }
    }
  }

  particles_.clear();
  for(int l = 0; l < num_of_particles_; l++)
    particles_.push_back(resampled_poses[l]);
}

void ParticleFilter::SimIPSCb(const gazebo_msgs::ModelStates::ConstPtr &msg) {
  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
  double yaw = tf::getYaw(msg->pose[8].orientation); // Robot Yaw
  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));
}

//Callback for the command data
void ParticleFilter::CommandCb(const geometry_msgs::Twist::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->linear.x * dt_;
    pose_delta[1] = msg->linear.y * dt_;
    pose_delta[2] = msg->angular.z * dt_;
    pose_deltas_.push(pose_delta);
  }
  last_command_time_ = now;
```

```
}

//Callback function for the Position topic (LIVE)
void ParticleFilter::PoseCb(
  const geometry_msgs::PoseWithCovarianceStamped &msg) {
  measurements_ << msg.pose.pose.position.x, msg.pose.pose.position.y,
                   tf::getYaw(msg.pose.pose.orientation);
  //ROS_DEBUG("pose_callback X: %f Y: %f Yaw: %f", measurements_(0),
  //          measurements_(1), measurements_(2));
}

//Callback function for the map
void ParticleFilter::MapCb(const nav_msgs::OccupancyGrid &msg) {
  //This function is called when a new map is received
  ConvertMap(msg);
  UniformPoseGenerator();
  particles_msg_.header.frame_id = "/odom";
  particles_msg_.poses.resize(num_of_particles_);
  ROS_INFO("Made the particles!");
  PublishPoseArray();
  map_received_ = true;
}

ParticleFilter::ParticleFilter () {
  //nh_.param<int>("no_of_particles", num_of_particles_, 20);
  //if (!nh_.hasParam("no_of_particles")) {
  //  ROS_INFO("no_of_particles is not defined.");
  //}
  num_of_particles_ = 4000;
  //Subscribe to the desired topics and assign callbacks
  odom_sub_ = nh_.subscribe("/mobile_base/commands/velocity", 1,
                                            &ParticleFilter::CommandCb, this);
  sim_ips_sub_ = nh_.subscribe("/gazebo/model_states", 1,
                                               &ParticleFilter::SimIPSCb, this);
  map_sub_ = nh_.subscribe("/map", 1, &ParticleFilter::MapCb,
                                           this);
  //Setup topics to Publish from this node
  pf_cloud_publisher_ = nh_.advertise<geometry_msgs::PoseArray>("/pf_cloud", 1,
                                                                 true);
  pf_pose_publisher_ = nh_.advertise<geometry_msgs::PoseStamped>("/pf_pose", 1,
                                                                  true);
  path_publisher_ = nh_.advertise<nav_msgs::Path>("/pf_path", 1, true);
  robot_path_.header.frame_id = "/odom";
  map_received_ = false;
  double last_command_time_ = -1;
}

void ParticleFilter::Estimate() {
  /*
  Load map
  Generate Particles
  Propagate the particles using the commanded velocity.
  Weight each of the particles
  Normalize the weights
  Resample using a process called stochastic universal sampling(Low variance resampling)
  */
  if (map_received_) {
    if (PropagateParticles()) {
      if (WeighParticles()) {
        ResampleParticles();
        PublishPosePath();
      }
    }
    PublishPoseArray();
  }
}
```