# Course 5 - Mapping and SLAM

## Introduction to Mapping and SLAM

### Introduction

Video 1.1

### Mapping

Video 1.2

### SLAM

Video 1.3

## Occupancy Grid Mapping

### Introduction

Video 2.1

**Localization:**

* *Assumption*: Known Map
* *Estimation*: Robot's Trajectory or poses

**Mapping:**

* *Assumption*: Known Robot's Trajectory or poses
* *Estimation*: Map

In this lesson, you'll learn how to **map** an environment with the **Occupancy Grid Mapping** algorithm!

### Importance of Mapping

Video 2.2

A question with a person pointing at the map

AI-generated content may be incorrect.

### Challenges and Difficulties

Video 2.3

While listing the different challenges and difficulties in mapping, I mentioned the words **discrete** and **continuous**. Here's what these words actually mean:

* **Discrete Data:** You obtain this data by counting it. This data has finite values. *Example*: Number of robots in a room
* **Continuous Data:** You obtain this data by measuring it. This data has an infinite number of steps, which form a continuum. *Example*: Weight of a robot

A screenshot of a quiz

AI-generated content may be incorrect.

### Mapping with Known Poses

Video 2.4

A screenshot of a question

AI-generated content may be incorrect.

### Posterior Probability

A diagram of a diagram of a diagram

AI-generated content may be incorrect.

**Posterior Probability**

Going back to the graphical model of mapping with known poses, our goal is to implement a mapping algorithm and estimate the map given noisy measurements and assuming known poses.

The Mapping with Known Poses problem can be represented with the function



With this function, we can compute the posterior over the map given all the measurements up to time **t** and all the poses up to time **t** represented by the robot trajectory.

In estimating the map, we’ll exclude the controls **u** since the robot path is provided to us from SLAM. However, keep in mind that the robot controls will be included later in SLAM to estimate the robot’s trajectory.

**2D Maps**

For now, we will only estimate the posterior for two-dimensional maps. In the real world, a mobile robot with a two-dimensional laser rangefinder sensor is generally deployed on a flat surface to capture a slice of the 3D world. Those two-dimensional slices will be merged at each instant and partitioned into grid cells to estimate the posterior through the occupancy grid mapping algorithm. Three-dimensional maps can also be estimated through the occupancy grid algorithm, but at much higher computational memory because of the large number of noisy three-dimensional measurements that need to be filtered out.

A screenshot of a quiz

AI-generated content may be incorrect.

### Grid Cells

Video 2.6

A screenshot of a quiz

AI-generated content may be incorrect.

### Computing the Posterior

A diagram of a mathematical equation

AI-generated content may be incorrect.

A text on a white background

AI-generated content may be incorrect.

A screenshot of a quiz

AI-generated content may be incorrect.

### Filtering

Video 2.8

**Forward vs. Inverse Measurement Model**

**Forward Measurement Model** - *P(z1:t| x)*: Estimating a posterior over the measurement given the system state.

**Inverse Measurement Model** - *P(x | z1:t)*: Estimating a posterior over the system state given the measurement.

The inverse measurement model is generally used when measurements are more complex than the system's state.

A screenshot of a test

AI-generated content may be incorrect.

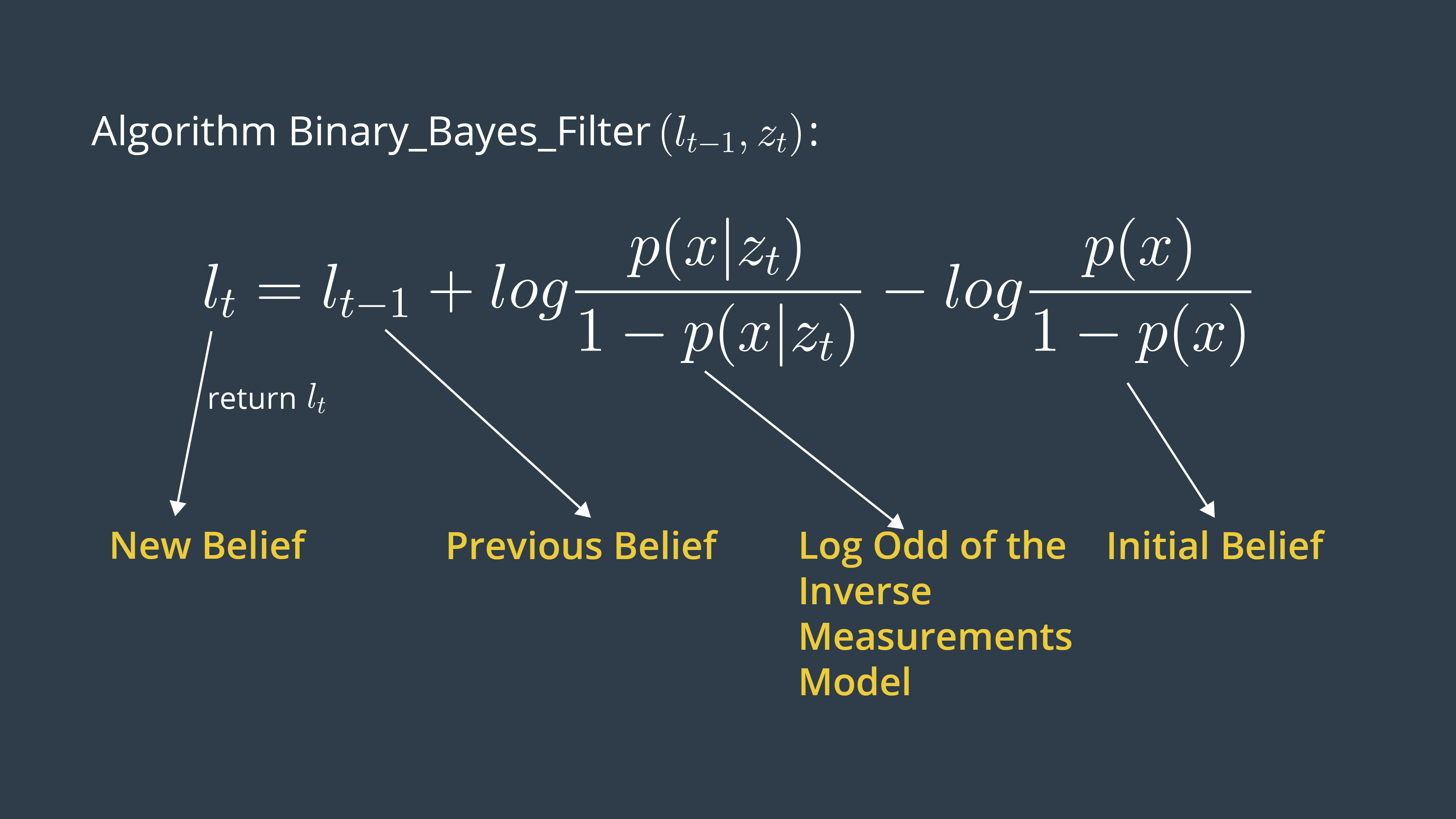
A screenshot of a question

AI-generated content may be incorrect.

The advantage of using a log odds ratio representation is to avoid probability instabilities near 0 or 1. Another advantage relates to system speed, accuracy, and simplicity. Check out these two sources for more information on log probability and numerical stability:

1. [**Log Probability**](https://en.wikipedia.org/wiki/Log_probability)
2. [**Numerical Stability**](https://en.wikipedia.org/wiki/Numerical_stability)

### Binary Bayes Filter Algorithm



**Input**

The binary Bayes filter algorithm computes the log odds of the posterior belief denoted by **lt**. Initially, the filter takes the previous log odds ratio of the belief **t-1** and the measurements **zt** as parameters.

**Computation**

Then, the filter computes the new posterior belief of the system **lt** by adding the previous belief **lt-1** to the log odds ratio of the inverse measurement model *log*(*p*(*x*∣*zt*​) / 1 - *p*(*x*∣*zt*​)​ ) and subtracting the prior probability state also known by initial belief  *log*(*p*(*x*)/ 1- *p*(*x*) )​. The initial belief represents the initial state of the system before taking any sensor measurements into consideration.

**Output**

Finally, the algorithm returns the posterior belief of the system **lt**, and a new iteration cycle begins.

### Occupancy Grid Mapping Algorithm

Video 2.10

Now that you've learned the Occupancy Grid Mapping algorithm, you will get a chance to code it in C++!

In this quiz, a robot equipped with **eight sonar rangefinder sensors** circulates in an environment to map it. This robot is provided with its exact poses at each timestamp. The code structure is as follows:

**Data Files**

1. measurement.txt: The measurements from the sonar rangefinder sensors attached to the robot at each time stamp recorded over a period of 413 seconds. (timestamp, measurement 1:8).
2. poses.txt: The exact robot poses at each timestamp recorded over a period of 413 seconds. (timestamp, x, y, ϴ).

**Global Functions**

1. inverseSensorModel(): You'll code this function as part of your second quiz after learning the inverse sensor model for sonar rangefinder sensors.
2. occupancyGridMapping(): You'll code this function as part of your first quiz.

**Main Function**

1. File Scan: Scanning both the measurement and poses files to retrieve the values. At each time stamp, the values are passed to the occupancy grid mapping function.
2. Display Map: After processing all the measurements and poses, the map is displayed.

Now, code the **occupancyGridMapping()** function:

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Use ***measurement.txt*** and ***poses.txt*** files

// main.cpp

#include <iostream>

#include <math.h>

#include <vector>

using namespace std;

// Sensor characteristic: Min and Max ranges of the beams

double Zmax = 5000, Zmin = 170;

// Defining free cells(lfree), occupied cells(locc), unknown cells(l0) log odds values

double l0 = 0, locc = 0.4, lfree = -0.4;

// Grid dimensions

double gridWidth = 100, gridHeight = 100;

// Map dimensions

double mapWidth = 30000, mapHeight = 15000;

// Robot size with respect to the map

double robotXOffset = mapWidth / 5, robotYOffset = mapHeight / 3;

// Defining an l vector to store the log odds values of each cell

vector< vector<double> > l(mapWidth/gridWidth, vector<double>(mapHeight/gridHeight));

double inverseSensorModel(double x, double y, double theta, double xi, double yi, double sensorData[])

{

    // You will be coding this section in the upcoming concept!

    return 0.4;

}

void occupancyGridMapping(double Robotx, double Roboty, double Robottheta, double sensorData[])

{

    //1 - TODO: Generate a grid (size 300x150) and then loop through all the cells

            //2- TODO: Compute the center of mass of each cell xi and yi

            //double xi = x \* gridWidth + gridWidth / 2 - robotXOffset;

            //double yi = -(y \* gridHeight + gridHeight / 2) + robotYOffset;

            //3- TODO: Check if each cell falls under the perceptual field of the measurements

}

int main()

{

    double timeStamp;

    double measurementData[8];

    double robotX, robotY, robotTheta;

    FILE\* posesFile = fopen("poses.txt", "r");

    FILE\* measurementFile = fopen("measurement.txt", "r");

    // Scanning the files and retrieving measurement and poses at each timestamp

    while (fscanf(posesFile, "%lf %lf %lf %lf", &timeStamp, &robotX, &robotY, &robotTheta) != EOF) {

        fscanf(measurementFile, "%lf", &timeStamp);

        for (int i = 0; i < 8; i++) {

            fscanf(measurementFile, "%lf", &measurementData[i]);

        }

        occupancyGridMapping(robotX, robotY, (robotTheta / 10) \* (M\_PI / 180), measurementData);

    }

    // Displaying the map

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            cout << l[x][y] << " ";

        }

    }

    return 0;

}

// solution.cpp

#include <iostream>

#include <math.h>

#include <vector>

using namespace std;

// Sensor characteristic: Min and Max ranges of the beams

double Zmax = 5000, Zmin = 170;

// Defining free cells(lfree), occupied cells(locc), unknown cells(l0) log odds values

double l0 = 0, locc = 0.4, lfree = -0.4;

// Grid dimensions

double gridWidth = 100, gridHeight = 100;

// Map dimensions

double mapWidth = 30000, mapHeight = 15000;

// Robot size with respect to the map

double robotXOffset = mapWidth / 5, robotYOffset = mapHeight / 3;

// Defining an l vector to store the log odds values of each cell

vector< vector<double> > l(mapWidth/gridWidth, vector<double>(mapHeight/gridHeight));

double inverseSensorModel(double x, double y, double theta, double xi, double yi, double sensorData[])

{

    // You will be coding this section in the upcoming concept!

    return 0.4;

}

void occupancyGridMapping(double Robotx, double Roboty, double Robottheta, double sensorData[])

{

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Code the Occupancy Grid Mapping Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            double xi = x \* gridWidth + gridWidth / 2 - robotXOffset;

            double yi = -(y \* gridHeight + gridHeight / 2) + robotYOffset;

            if (sqrt(pow(xi - Robotx, 2) + pow(yi - Roboty, 2)) <= Zmax) {

                l[x][y] = l[x][y] + inverseSensorModel(Robotx, Roboty, Robottheta, xi, yi, sensorData) - l0;

            }

        }

    }

}

int main()

{

    double timeStamp;

    double measurementData[8];

    double robotX, robotY, robotTheta;

    FILE\* posesFile = fopen("poses.txt", "r");

    FILE\* measurementFile = fopen("measurement.txt", "r");

    // Scanning the files and retrieving measurement and poses at each timestamp

    while (fscanf(posesFile, "%lf %lf %lf %lf", &timeStamp, &robotX, &robotY, &robotTheta) != EOF) {

        fscanf(measurementFile, "%lf", &timeStamp);

        for (int i = 0; i < 8; i++) {

            fscanf(measurementFile, "%lf", &measurementData[i]);

        }

        occupancyGridMapping(robotX, robotY, (robotTheta / 10) \* (M\_PI / 180), measurementData);

    }

    // Displaying the map

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            cout << l[x][y] << " ";

        }

    }

    return 0;

}

### Inverse Sensor Model

Video 2.11

**Summary of notations for the sonar rangefinder inverse sensor model:**

* *$$m\_{i}$$:* Map at instant i or current cell that is being processed
* *$$x\_{i},y\_{i}$$:* Center of mass of the current cell mi
* r: Range of the center of mass computed with respect to robot pose and center of mass
* k: The sonar rangefinder cone that best aligns with the cell being considered computed with respect to the robot pose (x,y,θ*θ*), center of mass (xi,yi), and sensor angle.
* *$$\beta$$:* Opening angle of the conical region formed out of the measurement beams.
* *$$\alpha$$:* Width of obstacles which is almost equal to the size of a cell. Please not that alpha is not the width of the conical region as the video mention but instead it's the width of a cell.

A screen shot of a computer screen

AI-generated content may be incorrect.

In this quiz, you'll code the **inverseSensorModel()** function which has two separate tasks:

1. Compute r and phi
2. Evaluate the three different cases of the algorithm

#### main.cpp

#include <iostream>

#include <math.h>

#include <vector>

using namespace std;

// Sensor characteristic: Min and Max ranges of the beams

double Zmax = 5000, Zmin = 170;

// Defining free cells(lfree), occupied cells(locc), unknown cells(l0) log odds values

double l0 = 0, locc = 0.4, lfree = -0.4;

// Grid dimensions

double gridWidth = 100, gridHeight = 100;

// Map dimensions

double mapWidth = 30000, mapHeight = 15000;

// Robot size with respect to the map

double robotXOffset = mapWidth / 5, robotYOffset = mapHeight / 3;

// Defining an l vector to store the log odds values of each cell

vector< vector<double> > l(mapWidth/gridWidth, vector<double>(mapHeight/gridHeight));

double inverseSensorModel(double x, double y, double theta, double xi, double yi, double sensorData[])

{

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Code the Inverse Sensor Model Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    // Defining Sensor Characteristics

    double Zk, thetaK, sensorTheta;

    double minDelta = -1;

    double alpha = 200, beta = 20;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TODO: Compute r and phi\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    //Scaling Measurement to [-90 -37.5 -22.5 -7.5 7.5 22.5 37.5 90]

    for (int i = 0; i < 8; i++) {

        if (i == 0) {

            sensorTheta = -90 \* (M\_PI / 180);

        }

        else if (i == 1) {

            sensorTheta = -37.5 \* (M\_PI / 180);

        }

        else if (i == 6) {

            sensorTheta = 37.5 \* (M\_PI / 180);

        }

        else if (i == 7) {

            sensorTheta = 90 \* (M\_PI / 180);

        }

        else {

            sensorTheta = (-37.5 + (i - 1) \* 15) \* (M\_PI / 180);

        }

        if (fabs(phi - sensorTheta) < minDelta || minDelta == -1) {

            Zk = sensorData[i];

            thetaK = sensorTheta;

            minDelta = fabs(phi - sensorTheta);

        }

    }

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TODO: Evaluate the three cases\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    // You also have to consider the cells with Zk > Zmax or Zk < Zmin as unkown states

}

void occupancyGridMapping(double Robotx, double Roboty, double Robottheta, double sensorData[])

{

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            double xi = x \* gridWidth + gridWidth / 2 - robotXOffset;

            double yi = -(y \* gridHeight + gridHeight / 2) + robotYOffset;

            if (sqrt(pow(xi - Robotx, 2) + pow(yi - Roboty, 2)) <= Zmax) {

                l[x][y] = l[x][y] + inverseSensorModel(Robotx, Roboty, Robottheta, xi, yi, sensorData) - l0;

            }

        }

    }

}

int main()

{

    double timeStamp;

    double measurementData[8];

    double robotX, robotY, robotTheta;

    FILE\* posesFile = fopen("poses.txt", "r");

    FILE\* measurementFile = fopen("measurement.txt", "r");

    // Scanning the files and retrieving measurement and poses at each timestamp

    while (fscanf(posesFile, "%lf %lf %lf %lf", &timeStamp, &robotX, &robotY, &robotTheta) != EOF) {

        fscanf(measurementFile, "%lf", &timeStamp);

        for (int i = 0; i < 8; i++) {

            fscanf(measurementFile, "%lf", &measurementData[i]);

        }

        occupancyGridMapping(robotX, robotY, (robotTheta / 10) \* (M\_PI / 180), measurementData);

    }

    // Displaying the map

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            cout << l[x][y] << " ";

        }

    }

    return 0;

}

#### solution.cpp

#include <iostream>

#include <math.h>

#include <vector>

using namespace std;

// Sensor characteristic: Min and Max ranges of the beams

double Zmax = 5000, Zmin = 170;

// Defining free cells(lfree), occupied cells(locc), unknown cells(l0) log odds values

double l0 = 0, locc = 0.4, lfree = -0.4;

// Grid dimensions

double gridWidth = 100, gridHeight = 100;

// Map dimensions

double mapWidth = 30000, mapHeight = 15000;

// Robot size with respect to the map

double robotXOffset = mapWidth / 5, robotYOffset = mapHeight / 3;

// Defining an l vector to store the log odds values of each cell

vector< vector<double> > l(mapWidth/gridWidth, vector<double>(mapHeight/gridHeight));

double inverseSensorModel(double x, double y, double theta, double xi, double yi, double sensorData[])

{

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Code the Inverse Sensor Model Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    // Defining Sensor Characteristics

    double Zk, thetaK, sensorTheta;

    double minDelta = -1;

    double alpha = 200, beta = 20;

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Compute r and phi\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    double r = sqrt(pow(xi - x, 2) + pow(yi - y, 2));

    double phi = atan2(yi - y, xi - x) - theta;

    //Scaling Measurement to [-90 -37.5 -22.5 -7.5 7.5 22.5 37.5 90]

    for (int i = 0; i < 8; i++) {

        if (i == 0) {

            sensorTheta = -90 \* (M\_PI / 180);

        }

        else if (i == 1) {

            sensorTheta = -37.5 \* (M\_PI / 180);

        }

        else if (i == 6) {

            sensorTheta = 37.5 \* (M\_PI / 180);

        }

        else if (i == 7) {

            sensorTheta = 90 \* (M\_PI / 180);

        }

        else {

            sensorTheta = (-37.5 + (i - 1) \* 15) \* (M\_PI / 180);

        }

        if (fabs(phi - sensorTheta) < minDelta || minDelta == -1) {

            Zk = sensorData[i];

            thetaK = sensorTheta;

            minDelta = fabs(phi - sensorTheta);

        }

    }

    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Evaluate the three cases\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

    if (r > min((double)Zmax, Zk + alpha / 2) || fabs(phi - thetaK) > beta / 2 || Zk > Zmax || Zk < Zmin) {

        return l0;

    }

    else if (Zk < Zmax && fabs(r - Zk) < alpha / 2) {

        return locc;

    }

    else if (r <= Zk) {

        return lfree;

    }

}

void occupancyGridMapping(double Robotx, double Roboty, double Robottheta, double sensorData[])

{

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            double xi = x \* gridWidth + gridWidth / 2 - robotXOffset;

            double yi = -(y \* gridHeight + gridHeight / 2) + robotYOffset;

            if (sqrt(pow(xi - Robotx, 2) + pow(yi - Roboty, 2)) <= Zmax) {

                l[x][y] = l[x][y] + inverseSensorModel(Robotx, Roboty, Robottheta, xi, yi, sensorData) - l0;

            }

        }

    }

}

int main()

{

    double timeStamp;

    double measurementData[8];

    double robotX, robotY, robotTheta;

    FILE\* posesFile = fopen("poses.txt", "r");

    FILE\* measurementFile = fopen("measurement.txt", "r");

    // Scanning the files and retrieving measurement and poses at each timestamp

    while (fscanf(posesFile, "%lf %lf %lf %lf", &timeStamp, &robotX, &robotY, &robotTheta) != EOF) {

        fscanf(measurementFile, "%lf", &timeStamp);

        for (int i = 0; i < 8; i++) {

            fscanf(measurementFile, "%lf", &measurementData[i]);

        }

        occupancyGridMapping(robotX, robotY, (robotTheta / 10) \* (M\_PI / 180), measurementData);

    }

    // Displaying the map

    for (int x = 0; x < mapWidth / gridWidth; x++) {

        for (int y = 0; y < mapHeight / gridHeight; y++) {

            cout << l[x][y] << " ";

        }

    }

    return 0;

}

### Generate the Map

**Mapping**

So far, you’ve coded the Occupancy Grid Mapping algorithm in C++ and generated an occupancy grid map 2D vector. Now, you'll code a visualization function that will loop through each cell. Then, you'll differentiate between occupied, free, and unknown cells depending on their log odds value. And, finally, you'll plot each cell on a graph to generate the map.

**Udacity Workspace**

For this quiz, follow these instructions:

A screenshot of a computer

AI-generated content may be incorrect.

Here are some helpful commands you can use to generate plots with the matplotlib library:

* *Set Title*: plt::title("Your Title");
* *Set Limits*: plt::xlim(x-axis lower limit, x-axis upper limit );
* *Plot Data*:plt::plot({ x-value }, { y-value }, "Color and Shape");
* *Save Plot*: plt::save("File name and directory");
* *Close Plot*: plt::clf();

Check out this [**link**](https://github.com/lava/matplotlib-cpp) for more information on the matplotlib C++ library. For information regarding the plot color and shape refer to the LineSpec and LineColor section of the [**MATLAB**](https://www.mathworks.com/help/matlab/ref/plot.html?requestedDomain=true) documentation.

A screenshot of a computer program

AI-generated content may be incorrect.

A map of a plane

AI-generated content may be incorrect.

A close up of words

AI-generated content may be incorrect.

### Udacity Workspace

* To follow along with the exercise's instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this exercise!

### Multi Sensor Fusion

Video 2.14

Give these two maps **m1** and **m2**:

map1       map2  
0.9 0.6      0.3 0.4  
0.1 0.5      0.4 0.3

Apply sensor fusion to combine the measurements of \*\*m1\*\* and \*\*m2\*\* in a resulting map. Scroll down to the C++ quiz section and code the `sensorFusion` function. ```C++ void sensorFusion(double m1[][mapWidth], double m2[][mapWidth]) { } ```

// main.cpp

#include <iostream>

#include <math.h>

using namespace std;

const int mapWidth =  2;

const int mapHeight = 2;

void sensorFusion(double m1[][mapWidth], double m2[][mapWidth])

{

    //\*#############TODO: Code the Sensor Fusion Function############\*//

    // Fuse the measurments of the two maps and print the resulting

    //map in a matrix form:

    //a  b

    //c  d

}

int main()

{

    double m1[mapHeight][mapWidth] = { { 0.9, 0.6 }, { 0.1, 0.5 } };

    double m2[mapHeight][mapWidth] = { { 0.3, 0.4 }, { 0.4, 0.3 } };

    sensorFusion(m1, m2);

    return 0;

}

// solution.cpp

#include <iostream>

#include <math.h>

using namespace std;

const int mapWidth =  2;

const int mapHeight = 2;

void sensorFusion(double m1[][mapWidth], double m2[][mapWidth])

{

    for (int x = 0; x < mapHeight; x++) {

        for (int y = 0; y < mapWidth; y++) {

            double p = 1 - (1 - m1[x][y]) \* (1 - m2[x][y]);

            cout << p << " ";

        }

        cout << endl;

    }

}

int main()

{

    double m1[mapHeight][mapWidth] = { { 0.9, 0.6 }, { 0.1, 0.5 } };

    double m2[mapHeight][mapWidth] = { { 0.3, 0.4 }, { 0.4, 0.3 } };

    sensorFusion(m1, m2);

    return 0;

}

### Introduction to 3D Mapping

So far, you’ve heard about two dimensional maps, describing a slice of the 3D world. In resource constrained systems, it can be very computationally expensive to build and maintain these maps. 3D representations are even more costly. That being said, robots live in the 3D world, and we want to represent that world and the 3D structures within it as accurately and reliably as possible. 3D mapping would give us the most reliable collision avoidance, and motion and path planning, especially for flying robots or mobile robots with manipulators.

First, let’s talk briefly about how we collect this 3D data, then we will move on to how it is represented. To create 3D maps, robots sense the environment by taking 3D range measurements. This can be done using numerous technologies.

3D lidar can be used, which is a single sensor with an array of laser beams stacked horizontally. Alternatively, a 2D lidar can be tilted (horizontally moving up and down) or rotated (360 degrees) to obtain 3D coverage.

An RGBD camera is a single visual camera combined with a laser rangefinder or infrared depth sensor, and allows for the determination of the depth of the image, and ultimately the distance from an object. A stereo camera is a pair of offset cameras, and can be used to directly infer the distance of close objects, in the same way as humans do with their two eyes.

A single camera system is cheaper and smaller, but the software algorithms needed for monocular SLAM are much more complex. Depth cannot be directly inferred from the sensor data of a single image from a single camera. Instead, it is calculated by analysing data from a sequence of frames in a video.

### 3D Data Representations

Video 2.16

Some of the desired characteristics of an optimal representation:

* Probabilistic data representations can be used to accommodate for sensor noise and dynamic environments.
* It is important to be able to distinguish data that represents an area that is free space versus an area that is unknown or not yet mapped. This will enable the robot to plan an unobstructed path and build a complete map.
* Memory on a mobile robot is typically a limited resource, so memory efficiency is very important. The map should also be accessible in the robot’s main memory, while mapping a large area over a long period of time. To accomplish this, we need a data representation that is compact and allows for efficient updates and queries.

**2.5D maps**, also known as height maps, store the surface of the entire environment as the maximum height measured at every point. They are memory efficient, with constant access time. This type of mapping is not very useful if you have terrain with trees or overhang structures, where the robot could move underneath. Also, height maps are non-probabilistic. Similar to point clouds, there is also no distinction between free and unknown space.

**Elevation maps** are 2D grids that store an estimated height, or elevation, for each cell. A Kalman filter is used to estimate the height, and can also incorporate the uncertainty of the measurement process itself, which typically increases with the measured distance. One problem with elevation maps is the vertical wall - you can tell there is a vertical object but don’t know exactly how tall it is.

**Extended elevation maps** store a set of estimated heights for every cell, and include cells that contain gaps. You can check whether the variance of the height of all data points within each cell is large. If so, you can investigate whether the corresponding set of points contains a gap exceeding the height of the robot (known as a “gap cell”), and ultimately use gap cells to determine traversability.

In **multi-level surface (MLS)** map representations, each 2D cell stores “patches”, of which there can be multiple per cell. Each patch contains 3 key pieces of information - the height mean, the height variance, and the depth value. The height mean is the estimated height of the individual vertical area, also referred to as an interval. The uncertainty of the height is stored as the height variance, with the assumption that the error is represented by a Gaussian distribution. The depth value is defined by the difference between height of the surface patch and the height of the lowest measurement that is considered as belonging to that vertical object (ex the depth of the floor would be 0). Individual surfaces can be directly calculated, allowing the robot to deal with vertical and overhanging objects. This method also works very well with multi-level traversable surfaces, such as a bridge that you could travel over top of, or underneath, or a structure like a parking garage. An MLS map isn’t a volumetric representation, but a discretization in the vertical dimension. Unknown areas are not represented, and localization for this method is not straightforward.

### Octomap

Video 2.17

[**Octomap GitHub Documentation**](https://octomap.github.io/)

[**Octomap on ROS Wiki**](http://wiki.ros.org/octomap)

### Outro

Video 2.18

## Grid-based FastSLAM

### Introduction

Video 3.1

A screenshot of a question

AI-generated content may be incorrect.

### Online SLAM

Video 3.2

**Online SLAM Problem**

* At time **t-1**, the robot will estimate its current pose **xt-1** and the map **m** given its current measurements **zt-1** and controls **ut-1**.
* At time **t**, the robot will estimate its new pose **xt** and the map **m** given only its current measurements **zt** and controls **ut**.
* At time **t+1**, the robot will estimate its current pose **xt+1** and the map **m** given the measurements **zt+1** and controls **ut+1**.

This problem can be modeled with the probability equation **p(xt , m | z1:t , u1:t)** where we solve the posterior represented by the instantaneous pose **xt** and the map **m** given the measurements **z1:t** and controls **u1:t**. Thus, with online SLAM we estimate variables that occur at time **t** only.

A screenshot of a question

AI-generated content may be incorrect.

### Full SLAM

Video 3.3

**Full SLAM Problem**

* At time **t-1**, the robot will estimate the robot pose **xt-1** and map **m**, given the measurements **zt-1** and controls **ut-1**.
* At time **t**, the robot will estimate the entire path **xt-1:t** and map **m**, given all the measurements **zt-1:t** and controls **ut-1:t**.
* At time **t+1**, the robot will estimate the entire path **xt-1:t+1** and map **m**, given all the measurements **zt-1:t+1** and controls **ut-1:t+1**.

This problem can be modeled with the probability equation **p(x1:t , m | z1:t u1:t)**, where we solve the posterior represented by the robot's trajectory **x1:t** and the map **m** given all the measurements **z1:t** and controls **u1:t**. Thus, with full SLAM problem we estimate all the variables that occur throughout the robot travel time.

A screenshot of a computer

AI-generated content may be incorrect.

### Nature of SLAM

A diagram of a diagram

AI-generated content may be incorrect.

**Forms**

You’ve just learned the first key feature of the SLAM problem which has to do with its two forms. The online SLAM problem computes a posterior over the current pose along with the map and the full SLAM problem computes a posterior over the entire path along with the map.

**Nature**

Now, the second key feature of the SLAM problem relates to its nature. SLAM problems generally have a continuous and a discrete element.

**Nature - Continuous**

Let’s start with the continuous component of the SLAM problem. During SLAM, a robot continuously collects odometry information to estimate the robot poses and continuously senses the environment to estimate the location of the object or landmark. Thus, both robots poses and object location are continuous aspects of the SLAM problem.

**Nature - Discrete**

Now, moving to the second component of the SLAM problem. As I mentioned earlier, robots continuously sense the environment to estimate the location of the objects, when doing so SLAM algorithms have to identify if a relation exists between any newly detected objects and previously detected ones. This helps the robot understand if it has been in this same location before. At each moment, the robot has to answer the question, “Have I been here before?”. The answer to this question is binary - either yes or no - and that’s what makes the relation between objects a discrete component of the SLAM problem.This discrete relation between objects is known by correspondence.

A screenshot of a question

AI-generated content may be incorrect.

### Correspondence

Video 3.5

Now that you've learned the two **key features** of the SLAM problem, let's summarize them:

**1-Forms**

* **Online SLAM**: Robot estimates its current pose and the map using current measurements and controls.
* **Full SLAM**: Robot estimates its entire trajectory and the map using all the measurements and controls.

**2- Nature**

* **Continuous**: Robot continuously senses its pose and the location of the objects.
* **Discrete**: Robot has to identify if a relation exists between any newly detected and previously detected objects.

A screenshot of a quiz

AI-generated content may be incorrect.

### SLAM Challenges

A diagram of a diagram of a full system

AI-generated content may be incorrect.

**SLAM Challenges**

Computing the full posterior composed of the robot pose, the map and the correspondence under SLAM poses a big challenge in robotics mainly due to the **continuous** and **discrete** portion.

A diagram of a diagram

AI-generated content may be incorrect.

**Continuous**

The continuous parameter space composed of the robot poses and the location of the objects is highly dimensional. While mapping the environment and localizing itself, the robot will encounter many objects and have to keep track of each one of them. Thus, the number of variables will increase with time, and this makes the problem highly dimensional and challenging to compute the posterior.

**Discrete**

Next, the discrete parameter space is composed out of the correspondence values, and is also highly dimensional due to the large number of correspondence variables. Not only that, the correspondence values increase exponentially over time since the robot will keep sensing the environment and relating the newly detected objects to the previously detected ones. Even if you assume known correspondence values, the posterior over maps is still highly dimensional as we saw in the mapping lessons.

You can now see why it’s infeasible to compute the posterior under unknown correspondence. Thus, SLAM algorithms will have to rely on approximation while estimating a posterior in order to conserve computational memory.

A screenshot of a question

AI-generated content may be incorrect.

### Particle Filter Approach to SLAM

Video 3.7

We just saw that adding another dimension to a particle so that it will hold the robot pose, weight, and the map and then solving through MCL in its current form will fail. This is because the map is modeled with many variables resulting in high dimensionality. Thus, the particle filter approach to SLAM in this current form will scale exponentially and is doomed to fail.

Next, you'll be introduced to **FastSLAM** and you'll learn how SLAM problems can be solved under a custom particle filter approach.

### Introduction to FastSLAM

Video 3.8

The **FastSLAM** algorithm solves the Full SLAM problem with known correspondences.

* **Estimating the Trajectory:** FastSLAM estimates a posterior over the trajectory using a particle filter approach. This will give an advantage to SLAM to solve the problem of mapping with known poses.
* **Estimating the Map:** FastSLAM uses a low dimensional Extended Kalman Filter to solve independent features of the map which are modeled with local Gaussian.

The custom approach of representing the posterior with particle filter and Gaussian is known by the **Rao-Blackwellized** particle filter approach.

A screenshot of a question

AI-generated content may be incorrect.

### FastSLAM Instances

A diagram of a diagram of a full slam and online

AI-generated content may be incorrect.

We’ve seen that the FastSLAM algorithm can solve the full SLAM problem with known correspondences. Since FastSLAM uses a particle filter approach to solve SLAM problems, some roboticists consider it a powerful algorithm capable of solving both the **Full SLAM** and **Online SLAM** problems.

* FastSLAM estimates the full robot path, and hence it solves the **Full SLAM** problem.
* On the other hand, each particle in FastSLAM estimates instantaneous poses, and thus FastSLAM also solves the **Online SLAM** problem.

A diagram of fast-based fast-slam

AI-generated content may be incorrect.

Now, three different instances of the FastSLAM algorithm exist.

**FastSLAM 1.0**

The FastSLAM 1.0 algorithm is simple and easy to implement, but this algorithm is known to be inefficient since particle filters generate sample inefficiency.

**FastSLAM 2.0**

The FastSLAM 2.0 algorithm overcomes the inefficiency of FastSLAM 1.0 by imposing a different distribution, which results in a low number of particles. Keep in mind that both of the FastSLAM 1.0 and 2.0 algorithms use a low dimensional Extended Kalman filter to estimate the posterior over the map features.

**Grid-based FastSLAM**

The third instance of FastSLAM is really an extension to FastSLAM known as the grid-based FastSLAM algorithm, which adapts FastSLAM to grid maps. In this lesson, you will learn grid-based FastSLAM. For more information on the fastSLAM 1.0 and 2.0 algorithms refer to the probabilistic robotics book.

A screenshot of a question

AI-generated content may be incorrect.

### Adapting FastSLAM to Grid Maps

A diagram of a fast slam algorithm

AI-generated content may be incorrect.

**FastSLAM 1.0 & 2.0**

The main advantage of the FastSLAM algorithm is that it uses a particle filter approach to solve the SLAM problem. Each particle will hold a guess of the robot trajectory, and by doing so, the SLAM problem is reduced to mapping with known poses. But, in fact, this algorithm presents a big disadvantage since it must always assume that there are known landmark positions, and thus with FastSLAM we are not able to model an arbitrary environment. Now, what if landmark positions are unavailable to us? Are we still able to solve the SLAM problem?

**Grid-based FastSLAM**

Yes, with the grid mapping algorithm you can model the environment using grid maps without predefining any landmark position. So by extending the FastSLAM algorithm to occupancy grid maps, you can now solve the SLAM problem in an arbitrary environment. While mapping a real-world environment, you will mostly be using mobile robots equipped with range sensors. You’ll then extend the FastSLAM algorithm and solve the SLAM problem in term of grid maps.

A diagram of a robot trajectory and math equations

AI-generated content may be incorrect.**Robot Trajectory**

Just as in the FastSLAM algorithm, with the grid-based FastSLAM each particle holds a guess of the robot trajectory.

**Map**

In addition, each particle maintains its own map. The grid-based FastSLAM algorithm will update each particle by solving the mapping with known poses problem using the occupancy grid mapping algorithm.

A screenshot of a question

AI-generated content may be incorrect.

### Grid-based FastSLAM Techniques

Video 3.11

A white sheet with black text and black text

AI-generated content may be incorrect.

A screenshot of a question

AI-generated content may be incorrect.

### The Grid-based FastSLAM Algorithm

Video 3.12

A screenshot of a computer

AI-generated content may be incorrect.

### gmapping ROS Package

Video 3.13

Now that you’ve learned the Grid-based FastSLAM algorithm, let’s test its power by solving the SLAM problem in simulation. In this lab, you’ll implement a ***gmapping*** ROS package which is based on the Grid-based FastSLAM algorithm to map an environment.

A diagram of a diagram

AI-generated content may be incorrect.

***gmapping*** provides laser based SLAM. Meaning that you can feed its node with the robot laser measurements and odometry values and expect it to provide you with a 2D occupancy grid map of the environment. The map will be updated as the robot moves and collect sensory information using its laser range finder sensor.

**gmapping package documentation**

Access this [**link**](http://wiki.ros.org/gmapping) and go over the documentation of the gmapping ROS package

**Deploying a Turltebot in a Willow Garage environment**

You will be deploying a turtlebot in a willow garage environment inside the [**Udacity Workspace**](https://classroom.udacity.com/nanodegrees/nd209/parts/dad7b7cc-9cce-4be4-876e-30935216c8fa/modules/451b7eed-6813-422a-a4d0-ce5db5ee1bca/lessons/411e2410-8f65-4764-a02a-e219ac36c776/concepts/fc59506b-6059-45a2-9d4d-204f7343988a?contentVersion=1.0.0&contentLocale=en-us). Thus, navigate to the next concept, enable GPU, and GO TO DESKTOP.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Udacity Workspace

* To follow along with the exercise's instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this exercise!

### SLAM with ROS

Video 3.15

**SLAM with ROS**

Here's the [**LAB solution**](https://github.com/udacity/RoboND-SLAMLAb)! While scrolling through the instructions, you will notice that some statements are a bit different than the one presented in the videos. The reason behind that is the different environment used. In the videos, a **Virtual Machine** booted with LUbuntu is used. Here in the description, you are presented with instructions on how to replicate the steps in the **Udacity Workspace**. The only difference between what you see and what is listed is the directory of the catkin\_ws. In VM, you can store it anywhere you want, preferably in the root ~ directory. Whereas, in the workspace, you will have to store it under /home/workspace/ so it won't get deleted after a reboot.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

With the **[map\_server](http://wiki.ros.org/map_server" \t "_blank)** you can load and save maps. Running map\_server will generate the **map.pgm** and the **map.yaml** files:

A drawing of a building

AI-generated content may be incorrect.

**map.pgm** of the Willow Garage environment

1- **map.pgm**: Picture of the map in occupancy grid representation

* *White pixels*: Free cells
* *Black pixels*: Occupied cells
* *Gray pixels*: Unknown state

A close-up of a sign

AI-generated content may be incorrect.

**map.yaml** of the Willow Garage environment

2- **map.yaml**: The map metadata

* *image*: Map name
* *resolution*: Resolution of the map (meters/pixel)
* *origin*: Pose of the lower-left pixel in the map (x, y, Θ)
* *Occupied\_thresh*: Cell is considered occupied if its probability is greater than this threshold.
* *free\_thresh*: Cell is considered unoccupied or free if its probability is less than this threshold.
* *negate*: This value will check whether the notation of black colored cell=occupied and white colored cell = free should be preserved

**Wondering why you got a bad quality map?**

That’s because the gmapping parameters values used were the default values. In general, it’s essential to tune them in order to get a 100% accurate map. These parameters are all listed under the gmapping documentation, where you can look at them yourself. If you experiment with some of these parameter values, you should be able to get better maps.

For example, you might try, reducing the angularUpdate and linearUpdate values so the map gets updated for smaller ranges of movements, reducing the x and y limits, which represent the initial map size, increasing the number of particles. You can try tweaking these parameters and/or any other parameter you think should be changed.

### Outro

Video 3.16

## GraphSLAM

### Introduction

Video 4.1

### Graphs

Video 4.2

Note: At timestamp 1:45 when Julia says the poses will be labeled x1 and x2, she means to say x0 and x1 as in the diagram.

**Summary of Notation**

* **Poses** are represented with triangles.
* **Features** from the environment are represented with stars.
* **Motion constraints** tie together two poses, and are represented by a solid line.
* **Measurement constraints** tie together a feature and a pose, and are represented by a dashed line.

### Constraints

Video 4.3

### Front-End vs Back-End

Video 4.4

### Maximum Likelihood Estimation

At the core of GraphSLAM is graph optimization - the process of minimizing the error present in all of the constraints in the graph. Let’s take a look at what these constraints look like, and learn to apply a principle called *maximum likelihood estimation* (MLE) to structure and solve the optimization problem for the graph.

**Likelihood**

Likelihood is a complementary principle to probability. While probability tries to estimate the outcome given the parameters, likelihood tries to estimate the parameters that best explain the outcome. For example,

**Probability:** What is the probability of rolling a 2 on a 6-sided die?

(Answer: 1/6)

**Likelihood:** I’ve rolled a die 100 times, and a 2 was rolled 10% of the time, how many sides does my die have?

(Answer: 10 sides)

When applied to SLAM, likelihood tries to estimate the most likely configuration of state and feature locations given the motion and measurement observations.

**Probability & Likelihood Quiz**

A screenshot of a computer

AI-generated content may be incorrect.

**Feature Measurement Example**

Let’s look at a very simple example - one where our robot is taking repeated measurements of a feature in the environment. This example will walk you through the steps required to solve it, which can then be applied to more complicated problems.

The robot’s initial pose has a variance of 0 - simply because this is its start location. Recall that wherever the start location may be - we call it location 0 in our relative map. Every action pose and measurement hereafter will be uncertain. In GraphSLAM, we will continue to make the assumption that motion and measurement data has Gaussian noise.

The robot takes a measurement of a feature, m1*m*1​, and it returns a distance of 1.8 metres.

If we return to our spring analogy - 1.8m is the spring’s resting length. This is the spring’s most desirable length; however, it is possible for the spring to be compressed or elongated to accommodate other forces (constraints) that are acting on the system.

This probability distribution for this measurement can be defined as so,

A black and white math equation

AI-generated content may be incorrect.

In simpler terms, the probability distribution is highest when z1 and x0 are 1.8 meters apart.

A math formula and a star

AI-generated content may be incorrect.

However, since the location of the first pose, x0*x*0​ is set to 0, this term can simply be removed from the equation.

A math equation with numbers and symbols

AI-generated content may be incorrect.

Next, the robot takes another measurement of the same feature in the environment. This time, the data reads 2.2m. With two conflicting measurements, this is now an overdetermined system - as there are more equations than unknowns!

A math equations and formulas

AI-generated content may be incorrect.

With two measurements, the most probable location of the feature can be represented by the product of the two probabilities.

A math problem with numbers

AI-generated content may be incorrect.

In this trivial example, it is probably quite clear to you that the most likely location of the feature is at the 2.0 meter mark. However, it is valuable to go through the maximum likelihood estimation process to understand the steps entailed, to then be able to apply it to more complicated systems.

To solve this problem analytically, a few steps can be taken to reduce the equations into a simpler form.

**Remove Scaling Factors**

The value of m that maximizes the equation does not depend on the constants in front of each of the exponentials. These are scaling factors, however in SLAM we are not usually interested in the absolute value of the probabilities, but finding the maximum likelihood estimate. For this reason, the factors can simply be removed.

A red line with black numbers and a red line with black numbers

AI-generated content may be incorrect.

A page of a math problem

AI-generated content may be incorrect.

A graph of a function

AI-generated content may be incorrect.

One thing to note when working with logs of likelihoods, is that they are always negative. This is because probabilities assume values between 0 and 1, and the log of any value between 0 and 1 is negative. This can be seen in the graph above. For this reason, when working with log-likelihoods, optimization entails *minimizing* the *negative* log-likelihood; whereas in the past, we were trying to maximize the likelihood.

Lastly, as was done before, the constants in front of the equation can be removed without consequence. As well, for the purpose of this example, we will assume that the same sensor was used in obtaining both measurements - and will thus ignore the variance in the equation.

A math equations with numbers

AI-generated content may be incorrect.

A white paper with black text and numbers

AI-generated content may be incorrect.

A graph of a function

AI-generated content may be incorrect.

In more complex examples, the curve may be multimodal, or exist over a greater number of dimensions. If the curve is multimodal, it may be unclear whether the locations discovered by the first derivative are in fact troughs, or peaks. In such a case, the second derivative of the function can be taken - which should clarify whether the local feature is a local minimum or maximum.

**Overview**

The procedure that you executed here is the *analytical* solution to an MLE problem. The steps included,

* Removing inconsequential constants,
* Converting the equation from one of *likelihood estimation* to one of *negative log-likelihood estimation*, and
* Calculating the first derivative of the function and setting it equal to zero to find the extrema.

In GraphSLAM, the first two steps can be applied to *every* constraint. Thus, any measurement or motion constraint can simply be labelled with its negative log-likelihood error. For a measurement constraint, this would resemble the following,

A math equation with numbers and symbols

AI-generated content may be incorrect.

And for a motion constraint, the following,

A math equation with numbers and symbols

AI-generated content may be incorrect.

Thus, from now on, constraints will be labelled with their negative log-likelihood error,

A math equations and a star

AI-generated content may be incorrect.

with the estimation function trying to minimize the sum of all constraints,

A math equation with numbers

AI-generated content may be incorrect.

In the next section, you will work through a more complicated estimation example to better understand maximum likelihood estimation, since it really is the basis of GraphSLAM.

### MLE Example

In the previous example you looked at a robot taking repeated measurements of the same feature in the environment. This example demonstrated the fundamentals of maximum likelihood estimation, but was very limited since it was only estimating one parameter - z1*z*1​.

In this example, you will have the opportunity to get hands-on with a more complicated 1-dimensional estimation problem.

**Motion and Measurement Example**

The robot starts at an arbitrary location that will be labeled 0, and then proceeds to measure a feature in front of it - the sensor reads that the feature is 7 meters way. The resultant graph is shown in the image below.

A math equation with a star

AI-generated content may be incorrect.

After taking its first measurement, the following Gaussian distribution describes the robot’s most likely location. The distribution is highest when the two poses are 3 metres apart.

A math equation with numbers and symbols

AI-generated content may be incorrect.

Recall that since we constrained the robot’s initial location to 0, x0*x*0​ can actually be removed from the equation.

Next, the robot moves forward by what it records to be 10 meters, and takes another measurement of the same feature. This time, the feature is read to be 4 meters behind the robot. The resultant graph looks like so,

A math equation with arrows and a yellow star

AI-generated content may be incorrect.

Now it’s up to you to determine what the two new constraints look like!

**Constraints Quizzes**

A screenshot of a math problem

AI-generated content may be incorrect.

A screenshot of a math problem

AI-generated content may be incorrect.

**Sum of Constraints**

The completed graph, with all of its labelled constraints can be seen below.

A math equations and numbers

AI-generated content may be incorrect.

Now, the task at hand is to minimize the sum of all constraints:



To do this, you will need to take the first derivative of the function and set it to equal zero. Seems easy, but wait - there are two variables! You’ll have to take the derivative with respect to each, and then solve the system of equations to calculate the values of the variables.

For this calculation, assume that the measurements and motion have equal variance.

See if you can work through this yourself to find the values of the variables, but if you’re finding this task challenging and would like a hint, skip ahead to the solution to the quiz where I will step you through the process.

**Optimization Quiz**

A screenshot of a math problem

AI-generated content may be incorrect.

If you’ve gotten this far, you’ve figured out that in the above example you needed to take the derivative of the error equation with respect to two different variables - z1*z*1​ and x1*x*1​ - and then perform variable elimination to calculate the most likely values for z1*z*1​ and x1*x*1​. This process will only get more complicated and tedious as the graph grows.

**Optimization with Non-Trivial Variances**

To make matters a little bit more complicated, let’s actually take into consideration the variances of each measurement and motion. Turns out that our robot has the fanciest wheels on the market - they’re solid rubber (they won’t deflate at different rates) - with the most expensive encoders. But, it looks like the funds ran dry after the purchase of the wheels - the sensor is of very poor quality.

Redo your math with the following new information,

* Motion variance: 0.02,
* Measurement variance: 0.1.
* Optimization Quiz 2

A screenshot of a math problem

AI-generated content may be incorrect.

That seemed to be a fair bit more work than the first example! At this point, we just have three constraints - imagine how difficult this process would be if we had collected measurement and motion data over a period of half-an hour, as may happen when mapping a real-life environment. The calculations would be tedious - even for a computer!

Solving the system analytically has the advantage of finding *the* correct answer. However, doing so can be very computationally intensive - especially as you move into multi-dimensional problems with complex probability distributions. In this example, the steps were easy to perform, but it only takes a short stretch of the imagination to think of how complicated these steps can become in complex multidimensional problems.

Well, *what is the alternative?* you may ask. Finding the maximum value can be done in two ways - *analytically* and *numerically*. Solving the problem numerically allows for a solution to be found rather quickly, however its accuracy may be sub-optimal. Next, you will look at how to solve complicated MLE problems numerically.

### Numerical Solution to MLE

The method that you applied in the previous two examples was very effective at finding a solution quickly - but that is not always the case. In more complicated problems, finding the analytical solution may involve lengthy computations.

Luckily there is an alternative - numerical solutions to maximum likelihood problems can be found in a fraction of the time. We will explore what a numerical solution to the previous example would look like.

**Numerical solution**

The graph of the error function from the previous example is seen below. In this example, it is very easy to see where the global minimum is located. However, in more complicated examples with multiple dimensions this is not as trivial.

A graph of a curve

AI-generated content may be incorrect.

This MLE can be solved numerically by applying an optimization algorithm. The goal of an optimization algorithm is to *speedily* find the optimal solution - in this case, the local minimum. There are several different algorithms that can tackle this problem; in SLAM, the [**gradient descent**](https://en.wikipedia.org/wiki/Gradient_descent), [**Levenberg-Marquardt**](https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm), and [**conjugate gradient**](https://en.wikipedia.org/wiki/Conjugate_gradient_method) algorithms are quite common. A brief summary of gradient descent.

**Quick Refresher on Gradient Descent**

Recall that the gradient of a function is a vector that points in the direction of the greatest rate of change; or in the case of an extrema, is equal to zero.

In gradient descent - you make an initial guess, and then adjust it incrementally in the direction *opposite* the gradient. Eventually, you should reach a minimum of the function.

This algorithm does have a shortcoming - in complex distributions, the initial guess can change the end result significantly. Depending on the initial guess, the algorithm converges on two different local minima. The algorithm has no way to determine where the global minimum is - it very naively moves down the steepest slope, and when it reaches a local minima, it considers its task complete. One solution to this problem is to use stochastic gradient descent (SGD), an iterative method of gradient descent using subsamples of data.

### Mid-Lesson Overview

Video 4.8

### 1-D to n-D

A maths formula on a white background

AI-generated content may be incorrect.

### Information Matrix and Vector

Video 4.10

Summary

* A **motion constraint** ties together two poses,
* A **measurement constraint** ties together the feature and the pose from which is was measured,
* Each operation updates 4 cells in the information matrix and 2 cells in the information vector,
* All other cells remain 0. Matrix is called ‘sparse’ due to large number of zero elements,
* **Sparsity** is a very helpful property for solving the system of equations.

### Inference

Once the information matrix and information vector have been populated, the path and map can be recovered by the following operation,



The result is a vector, *μ* defined over all poses and features, containing the best estimate for each. This operation is *very* similar to what you encountered before in the simple one-dimensional case, with a bit of added structure. Just as before, all constraints are considered when computing the solution.

Completing the above operation requires solving a system of equations. In small systems, this is an easily realizable task, but as the size of the graph and matrix grows - efficiency becomes a concern.

The efficiency of this operation, specifically the matrix inversion, depends greatly on the topology of the system.

**Linear Graph**

If the robot moves through the environment once, without ever returning to a previously visited location, then the topology is linear. Such a graph will produce a rather sparse matrix that, with some effort, can be reordered to move all non-zero elements to near the diagonal. This will allow the above equation to be completed in linear time.

**Cyclical Graph**

A more common topology is cyclical, in which a robot revisits a location that it has been to before, after some time has passed. In such a case, features in the environment will be linked to multiple poses - ones that are not consecutive, but spaced far apart. The further apart in time that these poses are - the more problematic, as such a matrix cannot be reordered to move non-zero cells closer to the diagonal. The result is a matrix that is more computationally challenging to recover.

However, all hope is not lost - a variable elimination algorithm can be used to simplify the matrix, allowing for the inversion and product to be computed quicker.

**Variable Elimination**

Variable elimination can be applied iteratively to remove all cyclical constraints. Just like it sounds, variable elimination entails removing a variable (ex. feature) entirely from the graph and matrix. This can be done by adjusting existing links or adding new links to accommodate for those links that will be removed.

If you recall the spring analogy, variable elimination removes features, but keeps the *net* forces in the springs unaltered by adjusting the tension on other springs or adding new springs where needed.

This process is demonstrated in the following two images. The first image shows the graph, matrix, and vector as they were presented in the previous video.

A diagram of a triangle and a diagram of a triangle

AI-generated content may be incorrect.

The second image shows the elimination of m1*m*1​. You can see that in this process the matrix will have five cells reset to zero (indicated in red), and four cells will have their values adjusted (indicated in green) to accommodate the variable elimination. Similarly, the information vector will have one cell removed and two adjusted.

A screen shot of a diagram

AI-generated content may be incorrect.

This process is repeated for all of the features, and in the end the matrix is defined over all robot poses. At this point, the same procedure as before can be applied,  *μ*=Ω−1*ξ*.

Performing variable elimination on the information matrix/vector prior to performing inference is less computationally intense than attempting to solve the inference problem directly on the unaltered data.

In practice, the analytical inference method described above is seldom applied, as numerical methods are able to converge on a sufficiently accurate estimate in a fraction of the time. More will be said on this topic later, but first it is important to explore how nonlinear constraints are handled in GraphSLAM.

### Nonlinear Constraints

In the Localization lesson, you were introduced to nonlinear motion and measurement models. The idea that a robot only moves in a linear fashion is quite limiting, and so it became important to understand how to work with nonlinear models. In localization, nonlinear models couldn’t be applied directly, as they would have turned the Gaussian distribution into a much more complicated distribution that could not be computed in closed form (analytically, in a finite number of steps). The same is true of nonlinear models in SLAM - most motion and measurement constraints are nonlinear, and must be linearized before they can be added to the information matrix and information vector. Otherwise, it would be impractical, if not impossible, to solve the system of equations analytically.

Luckily, you will be able to apply the same procedure that you learned in the [**EKF lesson**](https://classroom.udacity.com/nanodegrees/nd209/parts/dad7b7cc-9cce-4be4-876e-30935216c8fa/modules/f5048868-4bd8-4e8d-8c6b-69bd559ed9db/lessons/f002d591-94af-4c70-aeac-ac2ed6f7b527/concepts/bb377b96-eed0-49e8-bad9-5d416ee80cd1) of Localization to linearize nonlinear constraints for SLAM.

If you recall, a Taylor Series approximates a function using the sum of an infinite number of terms. A linear approximation can be computed by using only the first two terms and ignoring all higher order terms. In multi-dimensional models, the first derivative is replaced by a Jacobian - a matrix of partial derivatives.

**Linearizing Constraints**

A linearization of the measurement and motion constraints is the following,

A math equations with a white background

AI-generated content may be incorrect.

To linearize each constraint, you need a value for μt−1*μt*−1​ or *μt*​ to linearize about. This value is quite important since the linearization of a nonlinear function can change significantly depending on which value you choose to do so about. So, what *μt*−1​ or *μt*​ is a reasonable estimate for each constraint?

Well, when presented with a completed graph of nonlinear constraints, you can apply only the motion constraints to create a pose estimate, [*x*0​…*xt*​]*T*, and use this primitive estimate in place of *μ* to linearize all of the constraints. Then, once all of the constraints are linearized and added to the matrix and vector, a solution can be computed as before, using *μ*=Ω−1*ξ*.

This solution is unlikely to be an accurate solution. The pose vector used for linearization will be erroneous, since applying just the motion constraints will lead to a graph with a lot of drift, as errors accumulate with every motion. Errors in this initial pose vector will propagate through the calculations and affect the accuracy of the end result. This is especially so because the errors may increase in magnitude significantly during a poorly positioned linearization (where the estimated *μt*​ is far from reality, or the estimated *μt*​ lies on a curve where a small step in either direction will make a big difference).

To reduce this error, we can repeat the linearization process several times, each time using a better and better estimate to linearize the constraints about.

**Iterative Optimization**

The first iteration will see the constraints linearized about the pose estimate created using solely motion constraints. Then, the system of equations will be solved to produce a solution, *μ*.

The next iteration will use this solution, *μ*, as the estimate used to linearize about. The thought is that this estimate would be a little bit better than the previous; after all, it takes into account the measurement constraints too.

This process continues, with all consequent iterations using the previous solution as the vector of poses to linearize the constraints about. Each solution incrementally improves on the previous, and after some number of iterations the solution converges.

**Summary**

Nonlinear constraints can be linearized using Taylor Series, but this inevitably introduces some error. To reduce this error, the linearization of every constraint must occur as close as possible to the true location of the pose or measurement relating to the constraint. To accomplish this, an iterative solution is used, where the point of linearization is improved with every iteration. After several iterations, the result, *μ*, becomes a much more reasonable estimate for the true locations of all robot poses and features.

The workflow for GraphSLAM with nonlinear constraints is summarized below:

* Collect data, create graph of constraints,
* Until convergence:
  + Linearize all constraints about an estimate, μ*μ*, and add linearized constraints to the information matrix & vector,
  + Solve system of equations using *μ*=Ω−1*ξ*.

### Graph-SLAM at a Glance

Video 4.13

If you'd like to dive deeper into the mathematics of GraphSLAM, feel free to explore the following resources:

[**A Tutorial on Graph-Based SLAM, Grisetti**](http://www2.informatik.uni-freiburg.de/~stachnis/pdf/grisetti10titsmag.pdf)

[**The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structures, Thrun**](http://robot.cc/papers/thrun.graphslam.pdf)

### Intro to 3D SLAM With RTAB-Map

Video 4.14

### 3D SLAM With RTAB-Map

Video 4.15

A close-up of a building

AI-generated content may be incorrect.

The importance of loop closure is best understood by seeing a map result without it!

When loop closure is disabled, you can see parts of the map output that are repeated, and the resulting map looks a lot more choppy. It is not an accurate representation of the environment. This is caused by the robot not using loop closure to compare new images and locations to ones that are previously viewed, and instead it registers them as new locations. When loop closure is enabled, the map is significantly smoother and is an accurate representation of the room.

For example, on the left, where loop closure is disabled, you'll see highlighted where the door is represented as multiple corners and parts of a door, where on the right, you see a single clearly defined door.

### Visual Bag-of-Words

Video 4.16

### RTAB-Map Memory Management

Video 4.17

RTAB-Map uses a memory management technique to limit the number of locations considered as candidates during loop closure detection. This technique is a key feature of RTAB-Map and allows for loop closure to be done in real time.

The overall strategy is to keep the most recent and frequently observed locations in the robot’s **Working Memory (WM)**, and transfer the others into **Long-Term Memory (LTM)**.

* When a new image is acquired, a new node is created in the **Short Term Memory (STM)**.
* When creating a node, recall that features are extracted and compared to the vocabulary to find all of the words in the image, creating a bag-of-words for this node.
* Nodes are assigned a weight in the STM based on how long the robot spent in the location - where a longer time means a higher weighting.
* The STM has a fixed size of S. When STM reaches S nodes, the oldest node is moved to WM to be considered for loop closure detection.
* Loop closure happens in the WM.
* WM size depends on a fixed time limit T. When the time required to process new data reaches T, some nodes of graph are transferred from WM to LTM - as a result, WM size is kept nearly constant.
* Oldest and less weighted nodes in WM are transferred to LTM before others, so WM is made up of nodes seen for longer periods of time.
* LTM is not used for loop closure detection and graph optimization.
* If loop closure is detected, neighbours in LTM of an old node can be transferred back to the WM (a process called retrieval).

A diagram of a memory

AI-generated content may be incorrect.

### RTAB-Map Optimization and Output

Here we will discuss graph and map optimization, as well as time complexity considerations.

**Graph Optimization**

When a loop closure hypothesis is accepted, a new constraint is added to the map’s graph, then a graph optimizer minimizes the errors in the map. RTAB-Map supports 3 different graph optimizations: Tree-based network optimizer, or TORO, General Graph Optimization, or G2O and GTSAM (Smoothing and Mapping).

All of these optimizations use node poses and link transformations as constraints. When a loop closure is detected, errors introduced by the odometry can be propagated to all links, correcting the map.

Recall that Landmarks are used in the graph optimization process for other methods, whereas RTAB-Map doesn't use them. Only odometry constraints and loop closure constraints are optimized.

You can see the impact of graph optimization in the comparison below.

A screenshot of a map

AI-generated content may be incorrect.

**Map assembly and Output**

The possible outputs of RTAB-Map are a 2d Occupancy grid map, 3d occupancy grid map (3d octomap), or a 3D point cloud.

A screenshot of a computer generated image

AI-generated content may be incorrect.

**Graph SLAM Complexity and the Complexity of RTAB-Map**

A graph on a graph

AI-generated content may be incorrect.

Graph-SLAM complexity is linear, according to the number of nodes, which increases according to the size of the map.

A diagram of a computer

AI-generated content may be incorrect.

By providing constraints associated with how many nodes are processed for loop closure by memory management, the time complexity becomes constant in RTAB-Map.

### Outro

Video 4.19

## Map My World

### Overview

A robot vacuum cleaner in a room

AI-generated content may be incorrect.

**Overview**

Welcome to Project 3: Map My World! In this project you will create a 2D occupancy grid and 3D octomap from a simulated environment using your own robot with the RTAB-Map package.

RTAB-Map (Real-Time Appearance-Based Mapping) is a popular solution for SLAM to develop robots that can map environments in 3D. RTAB-Map has good speed and memory management, and it provides custom developed tools for information analysis. Most importantly, the quality of the documentation on ROS Wiki ([**http://wiki.ros.org/rtabmap\_ros**](http://wiki.ros.org/rtabmap_ros)) is very high. Being able to leverage RTAB-Map with your own robots will lead to a solid foundation for mapping and localization well beyond this Nanodegree program.

For this project we will be using the rtabmap\_ros package, which is a ROS wrapper (API) for interacting with RTAB-Map. Keep this in mind when looking at the relative documentation.

**Project Instructions**

Udacity provides you with a in-classroom Workspace with ROS configured to work on the project. If you need a refresher, check out the [**Workspace tutorials**](https://classroom.udacity.com/nanodegrees/nd209-beta/parts/5762c142-a09c-4379-b0e7-94a6693b47e8/modules/48156d08-abb1-4c03-a18d-9db738a0b92b/lessons/e0c61e8d-7eac-4807-8737-d2bd321ae7a2/concepts/47784838-aea6-4834-9ebb-79fbb3e135af).

The project flow will be as follows:

1. You will develop your own package to interface with the rtabmap\_ros package.
2. You will build upon your localization project to make the necessary changes to interface the robot with RTAB-Map. An example of this is the addition of an RGB-D camera.
3. You will ensure that all files are in the appropriate places, all links are properly connected, naming is properly setup and topics are correctly mapped. Furthermore you will need to generate the appropriate launch files to launch the robot and map its surrounding environment.
4. When your robot is launched you will teleop around the room to generate a proper map of the environment.

We are excited to see you grow with your ROS skills and we can't wait to see what your map looks like!

### Simulation Setup

Similar to the last project, let us setup the simulation environment and the robot in Project 4 Workspace for our next tasks.

Navigate to Workspace by clicking the Project Workspace on the side bar.

Setup your catkin\_ws folder as well as the src folder, then we need to grab the code from last project. Again, you could do that in two ways.

**Method 1: git**

If you have pushed your submission for the localization project to GitHub, go ahead and create a new repository then duplicate your code from last project to it.

A screenshot of a computer code

AI-generated content may be incorrect.

*Source:*[***https://help.github.com/articles/duplicating-a-repository/***](https://help.github.com/articles/duplicating-a-repository/)

**Method 2: Folder Upload**

If you want to upload the package folder to the Project 4 Workspace, go to the project Workspace and click the + button, then select Upload Folder to upload your package folder!

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Project Workspace

* To follow along with the project's instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this project!

### RTAB-Map Package

**RTAB-Map Pacakge**

Although ROS provides you with huge amount of packages, integrating a ROS package requires your understanding of the package itself and how it connects to your project. The best place to start is on the [**RTAB-Map ROS Wiki page**](http://wiki.ros.org/rtabmap_ros).

A screenshot of a computer

AI-generated content may be incorrect.

According to the [**documentation**](http://wiki.ros.org/rtabmap_ros/Tutorials/SetupOnYourRobot), the recommended robot configuration requires:

* A 2D Laser, providing sensor\_msgs/LaserScan messages
* Odometry sensors, providing nav\_msgs/Odometry messages
* 3D Camera, compatible with openni\_launch, openni2\_launch or freenect\_launch ROS packages

It seems that we lack the 3D camera sensor!

### Sensor Upgrade

In the previous projects you have been using an RGB camera; now it's time to give your robot an upgrade. Specifically, we will use a simulated Kinect camera for RTAB-Map.

A screenshot of a computer

AI-generated content may be incorrect.

**Add optical camera link**

For RGB-D camera in URDF file, we need to add an extra link and an extra joint to the camera link in order to align the camera image in Gazebo properly with the robot. Note that the parent link of camera\_optical\_joint should be properly configured to the original camera link.

Add the following joint and link to the robot's .xacro file:

A screenshot of a computer code

AI-generated content may be incorrect.

**Configuring the RGB-D Camera**

To do this we will need to replace the existing camera and its shared object file: libgazebo\_ros\_camera.so to that of the Kinect shared object file: libgazebo\_ros\_openni\_kinect.so. Also, update the <frameName> to be the camera\_link\_optical link you created just now.

On top of this, additional parameters need to be set for the RGB-D camera as well as matching the topics published by the drivers of its real world counterpart. We have provided an example for the camera code below. Substitute that in your robot's .gazebo file!

*Note: You could use the Classroom Workspace editor instead of the VNC Desktop to edit the gazebo file.*

A snippet of camera code is provided below:

<!-- RGBD Camera -->

  <gazebo reference="camera\_link">

    <sensor type="depth" name="camera1">

        <always\_on>1</always\_on>

        <update\_rate>20.0</update\_rate>

        <visualize>true</visualize>

        <camera>

            <horizontal\_fov>1.047</horizontal\_fov>

            <image>

                <width>640</width>

                <height>480</height>

                <format>R8G8B8</format>

            </image>

            <depth\_camera>

            </depth\_camera>

            <clip>

                <near>0.1</near>

                <far>20</far>

            </clip>

        </camera>

         <plugin name="camera\_controller" filename="libgazebo\_ros\_openni\_kinect.so">

            <alwaysOn>true</alwaysOn>

            <updateRate>10.0</updateRate>

            <cameraName>camera</cameraName>

            <frameName>camera\_link\_optical</frameName>

            <imageTopicName>rgb/image\_raw</imageTopicName>

            <depthImageTopicName>depth/image\_raw</depthImageTopicName>

            <pointCloudTopicName>depth/points</pointCloudTopicName>

            <cameraInfoTopicName>rgb/camera\_info</cameraInfoTopicName>

            <depthImageCameraInfoTopicName>depth/camera\_info</depthImageCameraInfoTopicName>

            <pointCloudCutoff>0.4</pointCloudCutoff>

                <hackBaseline>0.07</hackBaseline>

                <distortionK1>0.0</distortionK1>

                <distortionK2>0.0</distortionK2>

                <distortionK3>0.0</distortionK3>

                <distortionT1>0.0</distortionT1>

                <distortionT2>0.0</distortionT2>

            <CxPrime>0.0</CxPrime>

            <Cx>0.0</Cx>

            <Cy>0.0</Cy>

            <focalLength>0.0</focalLength>

            </plugin>

    </sensor>

  </gazebo>

### RTAB-Map Launch File

**Launch File**

We have the launch file to launch Gazebo and Rviz. Now, similar to what we did to create the amcl.launch in last project, we need to add the launch file for RTAB-Map!

**mapping.launch**

Our mapping launch file acts as the main node that interfaces with all the required parts to be able to perform SLAM with RTAB-Map. A labeled template for the mapping.launch file has been provided below. Create mapping.launch file in the launch folder.

Read through the code and the comments to understand what each part is accomplishing and why. Feel free to reach beyond this template with the documentation of RTAB-Map. You task here is to assign the correct topic to be remapped to the topics required by rtabmap.

* scan
* rgb/image
* depth/image
* rgb/camera\_info

You should find the actual topics that your robot is publishing to, in the robot's urdf file. When you find the correct ones, substitute them in the <arg> tags at the beginning of this launch file. Then your mapping node could find all required information to perform RTAB-Mapping!

<?xml version="1.0" encoding="UTF-8"?>

<launch>

  <!-- Arguments for launch file with defaults provided -->

  <arg name="database\_path"     default="rtabmap.db"/>

  <arg name="rgb\_topic"   default="/camera/rgb/image\_raw"/>

  <arg name="depth\_topic" default="/camera/depth/image\_raw"/>

  <arg name="camera\_info\_topic" default="/camera/rgb/camera\_info"/>

  <!-- Mapping Node -->

  <group ns="rtabmap">

    <node name="rtabmap" pkg="rtabmap\_ros" type="rtabmap" output="screen" args="--delete\_db\_on\_start">

      <!-- Basic RTAB-Map Parameters -->

      <param name="database\_path"       type="string" value="$(arg database\_path)"/>

      <param name="frame\_id"            type="string" value="base\_footprint"/>

      <param name="odom\_frame\_id"       type="string" value="odom"/>

      <param name="subscribe\_depth"     type="bool"   value="true"/>

      <param name="subscribe\_scan"      type="bool"   value="true"/>

      <!-- RTAB-Map Inputs -->

      <remap from="scan" to="/scan"/>

      <remap from="rgb/image" to="$(arg rgb\_topic)"/>

      <remap from="depth/image" to="$(arg depth\_topic)"/>

      <remap from="rgb/camera\_info" to="$(arg camera\_info\_topic)"/>

      <!-- RTAB-Map Output -->

      <remap from="grid\_map" to="/map"/>

      <!-- Rate (Hz) at which new nodes are added to map -->

      <param name="Rtabmap/DetectionRate" type="string" value="1"/>

      <!-- 2D SLAM -->

      <param name="Reg/Force3DoF" type="string" value="true"/>

      <!-- Loop Closure Detection -->

      <!-- 0=SURF 1=SIFT 2=ORB 3=FAST/FREAK 4=FAST/BRIEF 5=GFTT/FREAK 6=GFTT/BRIEF 7=BRISK 8=GFTT/ORB 9=KAZE -->

      <param name="Kp/DetectorStrategy" type="string" value="0"/>

      <!-- Maximum visual words per image (bag-of-words) -->

      <param name="Kp/MaxFeatures" type="string" value="400"/>

      <!-- Used to extract more or less SURF features -->

      <param name="SURF/HessianThreshold" type="string" value="100"/>

      <!-- Loop Closure Constraint -->

      <!-- 0=Visual, 1=ICP (1 requires scan)-->

      <param name="Reg/Strategy" type="string" value="0"/>

      <!-- Minimum visual inliers to accept loop closure -->

      <param name="Vis/MinInliers" type="string" value="15"/>

      <!-- Set to false to avoid saving data when robot is not moving -->

      <param name="Mem/NotLinkedNodesKept" type="string" value="false"/>

    </node>

  </group>

</launch>

***Further Resources:***

* [**RTAB-Map Parameter Tutorial**](http://wiki.ros.org/rtabmap_ros/Tutorials/Advanced%20Parameter%20Tuning)
* [**List of RTAB-Map Parameters**](https://github.com/introlab/rtabmap/blob/master/corelib/include/rtabmap/core/Parameters.h)

### RTAB-Map Real Time Visualization

**Real Time Visulization**

Another tool that you can use is rtabmapviz, which is an additional node for real time visualization of feature mapping, loop closures, and more. It’s not recommended to use this tool while mapping in simulation due to the computing overhead. rtabmapviz is great to deploy on a real robot during live mapping to ensure that you are getting the necessary features to complete loop closures.

A computer screen shot of a video game

AI-generated content may be incorrect.

If you would like to enable it for mapping, add this code snippet to the mapping.launch file. This will launch the rtabmapviz GUI and provide you with realtime feature detection, loop closures, and other relevant information to the mapping process.

<!-- visualization with rtabmapviz -->

    <node pkg="rtabmap\_ros" type="rtabmapviz" name="rtabmapviz" args="-d $(find rtabmap\_ros)/launch/config/rgbd\_gui.ini" output="screen">

        <param name="subscribe\_depth"             type="bool" value="true"/>

        <param name="subscribe\_scan"              type="bool" value="true"/>

        <param name="frame\_id"                    type="string" value="base\_footprint"/>

        <remap from="rgb/image"       to="$(arg rgb\_topic)"/>

        <remap from="depth/image"     to="$(arg depth\_topic)"/>

        <remap from="rgb/camera\_info" to="$(arg camera\_info\_topic)"/>

        <remap from="scan"            to="/scan"/>

    </node>

### ROS Teleop Package

You have used teleop node to control your robot by keyboard strokes in previous labs and projects. Here, we also need it so that we could navigate the robot in the environment and perform RTAB-Mapping.

Clone the teleop package to your Workspace src folder and complile! You could find the code here: [**https://github.com/ros-teleop/teleop\_twist\_keyboard**](https://github.com/ros-teleop/teleop_twist_keyboard)

A screenshot of a computer

AI-generated content may be incorrect.

### Mapping: Map My World

**Map My World!**

We have everything ready to go. Launch the ROS nodes and let us start mapping.

A computer generated image of a room

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Navigate your robot in the simulation to create map for the environment! When you are all set, terminal the node and you could find your map db file in the place you specified in the launch file. If you did not modify the argument, it will be located in the /root/.ros/ folder.

**Best Practices**

You could start by lower velocity. Our goal is to create a great map with the least amount of passes as possible. Getting 3 loop closures will be sufficient for mapping the entire environment. You can maximize your loop closures by going over similar paths two or three times. This allows for the maximization of feature detection, facilitating faster loop closures! When you are done mapping, be sure to copy or move your database before moving on to map a new environment. Remember, relaunching the mapping node deletes any database in place on launch start up!

### Mapping: Database

**Database Analysis**

The rtabmap-databaseViewer is a great tool for exploring your database when you are done generating it. It is isolated from ROS and allows for complete analysis of your mapping session.

This is how you will check for loop closures, generate 3D maps for viewing, extract images, check feature mapping rich zones, and much more!

A screenshot of a computer

AI-generated content may be incorrect.

Let’s start by opening our mapping database:

We can do this like so: rtabmap-databaseViewer ~/.ros/rtabmap.db

Once open, we will need to add some windows to get a better view of the relevant information, so:

* Say yes to using the database parameters
* View -> Constraint View
* View -> Graph View

Those options are enough to start, as there are many features built into the database viewer!

Let’s talk about what you are seeing in the above image. On the left, you have your 2D grid map in all of its updated iterations and the path of your robot. In the middle you have different images from the mapping process. Here you can scrub through images to see all of the features from your detection algorithm. These features are in yellow. Then, what is the pink, you may ask? The pink indicates where two images have features in common and this information is being used to create neighboring links and loop closures! Finally, on the right you can see the constraint view. This is where you can identify where and how the neighboring links and loop closures were created.

A screenshot of a computer

AI-generated content may be incorrect.

You can see the number of loop closures in the bottom left. The codes stand for the following: Neighbor, Neighbor Merged, Global Loop closure, Local loop closure by space, Local loop closure by time, User loop closure, and Prior link.

When it comes time to design your own environment, this tool can be a good resource for checking if the environment is feature-rich enough to make global loop closures. A good environment has many features that can be associated in order to achieve loop closures.

### Optional: RTAB-Map Localization

If you desire to perform localization using the map you created, there are only a few changes you need to make. You can start by duplicating your mapping.launch file and renaming the duplicated file tolocalization.launch.

The following changes also need to be made to the localization.launch file:

* Remove the args="--delete\_db\_on\_start" from your node launcher since you will need your database to localize too.
* Remove the Mem/NotLinkedNodesKept parameter
* Add the Mem/IncrementalMemory parameter of type string and set it to false to finalize the changes needed to put the robot into localization mode.

This is another method for localization you can keep in mind when working on your next robotics project!

### Project Rubric

**Basic Requirements**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student submit all required files? | Student submited all required files: ROS Package: robot and RTABMAP Db file generated (could be link to file if oversized) |

**Simulation Setup**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student set up the simulation environment properly? | Student's simulation world and robot could properly load in Gazebo. |
| Is the student's simulation suitable for mapping task? | The student's environment should have clear features and geometric shapes to perform mapping. |

**Mapping Package**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Does the student correctly build all required launch files for RTAB-Mapping? | Student created the following launch files properly: mapping.launch teleop.launch localization.launch The student's program should be able to launch without errors |

**Mapping Accuracy**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Was the student able to generate a 3D map using RTAB-Map? | Student's map should contain at least 3 loop closures and the occupancy grid is identifiable.  The resulting map should be free of noticeable distortions, with features accurately aligned and the environment's geometry properly represented. |
| Does the student's 3D map portray environment characteristics? | Student's 3D map should clearly portray the environment. The student should be able to display the characteristics of the landmark features.  The map should be free of distortions, breaks, or gaps and show a coherent representation of the robot's environment. |

**Suggestions to Make Your Project Stand Out**

Try building different simulation environments and evaluate the performance between them! Also, document your work on SLAM, which will contribute towards the final Home Service Robot project!

### Submit Project