**Project Overview**

The goal of this project is to take the **python** code you wrote when you implemented the two dimensional histogram filter and translate it into **C++**.

After you review the Python code you will download some starter C++ code. Your job is to complete the functions marked with TODO comments until the histogram filter works as expected!

You can evaluate the correctness of your code by compiling tests.cpp and then running the output. Do this by running the following from the command line.

g++ tests.cpp

./a.out

**Evaluation**

Once you have completed your project, use the [Project Rubric](https://review.udacity.com/#!/rubrics/1146/view) to review the project. If you have covered all of the points in the rubric, then you are ready to submit! If you see room for improvement in any category in which you do not meet specifications, keep working!

Your project will be evaluated by a Udacity reviewer according to the same Project Rubric. Your project must "meet specifications" in each category in order for your submission to pass.

# Python Code Reminder

You'll probably want to refer to the 2d Histogram Filter code as you write the C++ version. You can [find your Python code here](https://classroom.udacity.com/nanodegrees/nd113/parts/67914806-9163-4bef-85d4-5fa4a7409fbd/modules/bb316d89-3a59-4093-b62d-f36d7822e530/lessons/919d89ca-c492-4d34-9792-8a23d410517f/concepts/fdb1a3fc-8b15-4336-a8a4-b5090c32c7ea). Make sure you have that link open in another tab before you continue.

# Python to C++

**Note:** We have recently added a project workspace to the classroom that you can use for coding this project, which you can find in a few pages from now. The workspace already has the code downloaded in it if you choose to use that option, but you should still read the below instructions.

1. First, [click here to download the C++ starter code](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/iSDC/iSDC-P2-Translate-Python-Cpp.zip). [Note: This was recently refactored to include header files, as well as updated to remove an out-of-bounds issue with the test\_sense() function].
2. Open the code in your favorite editor. You'll probably want to have the corresponding Python code around to consult as well.
3. Fill out the functions in localizer.cpp and helpers.cpp

**NOTE** - when compiling your code, make sure you use C++11. You can do this from the command line with the following:

g++ -std=c++11 tests.cpp

### Are you having trouble getting the code?

Students in China have not been able to download the project starter code. If you were able to download the starter code you can ignore everything below.

As a temporary solution, we are including the code as text below. You will need to copy and paste the text into files with the correct names. If you have any problems please let us know in the Student Hub channel and we will help you ASAP.

**Directory Structure**

You should create a new project folder/directory to put the project code into. Inside that you should also create a maps directory.

Then you should create empty files within those directories to match the following image:

A picture containing bird

Description automatically generated

### File text (for .cpp and .txt files)

#### debugging\_helpers.cpp

*/\*\**

*debugging\_helpers.cpp*

*Purpose: helper functions for debugging when working*

*with grids of floats and chars.*

*\*/*

**#include <vector>**

**using** **namespace** std;

*/\*\**

*Displays a grid of beliefs. Does not return.*

*@param grid - a two dimensional grid (vector of*

*vectors of floats) which will usually*

*represent a robot's beliefs.*

*\*/*

**void** **show\_grid**(vector < vector <**float**> > grid) {

**int** i, j;

**float** p;

vector<**float**> row;

**for** (i = 0; i < grid.size(); i++)

{

row = grid[i];

**for** (j=0; j< row.size(); j++)

{

p = row[j];

cout << p << ' ';

}

cout << endl;

}

}

*/\*\**

*Displays a grid map of the world*

*\*/*

**void** **show\_grid**(vector < vector <**char**> > map) {

**int** i, j;

**char** p;

vector<**char**> row;

**for** (i = 0; i < map.size(); i++)

{

row = map[i];

**for** (j=0; j< row.size(); j++)

{

p = row[j];

cout << p << ' ';

}

cout << endl;

}

}

#### helpers.cpp

*/\*\**

*helpers.cpp*

*Purpose: helper functions which are useful when*

*implementing a 2-dimensional histogram filter.*

*This file is incomplete! Your job is to make the*

*normalize and blur functions work. Feel free to*

*look at helper.py for working implementations*

*which are written in python.*

*\*/*

**#include <vector>**

**#include <iostream>**

**#include <cmath>**

**#include <string>**

**#include <fstream>**

*// #include "debugging\_helpers.cpp"*

**using** **namespace** std;

*/\*\**

*TODO - implement this function*

*Normalizes a grid of numbers.*

*@param grid - a two dimensional grid (vector of vectors of floats)*

*where each entry represents the unnormalized probability*

*associated with that grid cell.*

*@return - a new normalized two dimensional grid where the sum of*

*all probabilities is equal to one.*

*\*/*

vector< vector<**float**> > normalize(vector< vector <**float**> > grid) {

vector< vector<**float**> > newGrid;

*// todo - your code here*

**return** newGrid;

}

*/\*\**

*TODO - implement this function.*

*Blurs (and normalizes) a grid of probabilities by spreading*

*probability from each cell over a 3x3 "window" of cells. This*

*function assumes a cyclic world where probability "spills*

*over" from the right edge to the left and bottom to top.*

*EXAMPLE - After blurring (with blurring=0.12) a localized*

*distribution like this:*

*0.00 0.00 0.00*

*0.00 1.00 0.00*

*0.00 0.00 0.00*

*would look like this:*

*0.01 0.02 0.01*

*0.02 0.88 0.02*

*0.01 0.02 0.01*

*@param grid - a two dimensional grid (vector of vectors of floats)*

*where each entry represents the unnormalized probability*

*associated with that grid cell.*

*@param blurring - a floating point number between 0.0 and 1.0*

*which represents how much probability from one cell*

*"spills over" to it's neighbors. If it's 0.0, then no*

*blurring occurs.*

*@return - a new normalized two dimensional grid where probability*

*has been blurred.*

*\*/*

vector < vector <**float**> > blur(vector < vector < **float**> > grid, **float** blurring) {

vector < vector <**float**> > newGrid;

*// your code here*

**return** normalize(newGrid);

}

*/\*\* -----------------------------------------------*

*#*

*#*

*# You do not need to modify any code below here.*

*#*

*#*

*# ------------------------------------------------- \*/*

*/\*\**

*Determines when two grids of floating point numbers*

*are "close enough" that they should be considered*

*equal. Useful for battling "floating point errors".*

*@param g1 - a grid of floats*

*@param g2 - a grid of floats*

*@return - A boolean (True or False) indicating whether*

*these grids are (True) or are not (False) equal.*

*\*/*

**bool** **close\_enough**(vector < vector <**float**> > g1, vector < vector <**float**> > g2) {

**int** i, j;

**float** v1, v2;

**if** (g1.size() != g2.size()) {

**return** false;

}

**if** (g1[0].size() != g2[0].size()) {

**return** false;

}

**for** (i=0; i<g1.size(); i++) {

**for** (j=0; j<g1[0].size(); j++) {

v1 = g1[i][j];

v2 = g2[i][j];

**if** (abs(v2-v1) > 0.0001 ) {

**return** false;

}

}

}

**return** true;

}

**bool** **close\_enough**(**float** v1, **float** v2) {

**if** (abs(v2-v1) > 0.0001 ) {

**return** false;

}

**return** true;

}

*/\*\**

*Helper function for reading in map data*

*@param s - a string representing one line of map data.*

*@return - A row of chars, each of which represents the*

*color of a cell in a grid world.*

*\*/*

vector <**char**> read\_line(string s) {

vector <**char**> row;

**size\_t** pos = 0;

string token;

string delimiter = " ";

**char** cell;

**while** ((pos = s.find(delimiter)) != std::string::npos) {

token = s.substr(0, pos);

s.erase(0, pos + delimiter.length());

cell = token.at(0);

row.push\_back(cell);

}

**return** row;

}

*/\*\**

*Helper function for reading in map data*

*@param file\_name - The filename where the map is stored.*

*@return - A grid of chars representing a map.*

*\*/*

vector < vector <**char**> > read\_map(string file\_name) {

ifstream **infile**(file\_name);

vector < vector <**char**> > map;

**if** (infile.is\_open()) {

**char** color;

vector <**char**> row;

string line;

**while** (std::getline(infile, line)) {

row = read\_line(line);

map.push\_back(row);

}

}

**return** map;

}

*/\*\**

*Creates a grid of zeros*

*For example:*

*zeros(2, 3) would return*

*0.0 0.0 0.0*

*0.0 0.0 0.0*

*@param height - the height of the desired grid*

*@param width - the width of the desired grid.*

*@return a grid of zeros (floats)*

*\*/*

vector < vector <**float**> > zeros(**int** height, **int** width) {

**int** i, j;

vector < vector <**float**> > newGrid;

vector <**float**> newRow;

**for** (i=0; i<height; i++) {

newRow.clear();

**for** (j=0; j<width; j++) {

newRow.push\_back(0.0);

}

newGrid.push\_back(newRow);

}

**return** newGrid;

}

*// int main() {*

*// vector < vector < char > > map = read\_map("maps/m1.txt");*

*// show\_grid(map);*

*// return 0;*

*// }*

#### localizer.cpp

*/\*\**

*localizer.cpp*

*Purpose: implements a 2-dimensional histogram filter*

*for a robot living on a colored cyclical grid by*

*correctly implementing the "initialize\_beliefs",*

*"sense", and "move" functions.*

*This file is incomplete! Your job is to make these*

*functions work. Feel free to look at localizer.py*

*for working implementations which are written in python.*

*\*/*

**#include "helpers.cpp"**

**#include <stdlib.h>**

**#include "debugging\_helpers.cpp"**

**using** **namespace** std;

*/\*\**

*TODO - implement this function*

*Initializes a grid of beliefs to a uniform distribution.*

*@param grid - a two dimensional grid map (vector of vectors*

*of chars) representing the robot's world. For example:*

*g g g*

*g r g*

*g g g*

*would be a 3x3 world where every cell is green except*

*for the center, which is red.*

*@return - a normalized two dimensional grid of floats. For*

*a 2x2 grid, for example, this would be:*

*0.25 0.25*

*0.25 0.25*

*\*/*

vector< vector <**float**> > initialize\_beliefs(vector< vector <**char**> > grid) {

vector< vector <**float**> > newGrid;

*// your code here*

**return** newGrid;

}

*/\*\**

*TODO - implement this function*

*Implements robot sensing by updating beliefs based on the*

*color of a sensor measurement*

*@param color - the color the robot has sensed at its location*

*@param grid - the current map of the world, stored as a grid*

*(vector of vectors of chars) where each char represents a*

*color. For example:*

*g g g*

*g r g*

*g g g*

*@param beliefs - a two dimensional grid of floats representing*

*the robot's beliefs for each cell before sensing. For*

*example, a robot which has almost certainly localized*

*itself in a 2D world might have the following beliefs:*

*0.01 0.98*

*0.00 0.01*

*@param p\_hit - the RELATIVE probability that any "sense" is*

*correct. The ratio of p\_hit / p\_miss indicates how many*

*times MORE likely it is to have a correct "sense" than*

*an incorrect one.*

*@param p\_miss - the RELATIVE probability that any "sense" is*

*incorrect. The ratio of p\_hit / p\_miss indicates how many*

*times MORE likely it is to have a correct "sense" than*

*an incorrect one.*

*@return - a normalized two dimensional grid of floats*

*representing the updated beliefs for the robot.*

*\*/*

vector< vector <**float**> > sense(**char** color,

vector< vector <**char**> > grid,

vector< vector <**float**> > beliefs,

**float** p\_hit,

**float** p\_miss)

{

vector< vector <**float**> > newGrid;

*// your code here*

**return** normalize(newGrid);

}

*/\*\**

*TODO - implement this function*

*Implements robot motion by updating beliefs based on the*

*intended dx and dy of the robot.*

*For example, if a localized robot with the following beliefs*

*0.00 0.00 0.00*

*0.00 1.00 0.00*

*0.00 0.00 0.00*

*and dx and dy are both 1 and blurring is 0 (noiseless motion),*

*than after calling this function the returned beliefs would be*

*0.00 0.00 0.00*

*0.00 0.00 0.00*

*0.00 0.00 1.00*

*@param dy - the intended change in y position of the robot*

*@param dx - the intended change in x position of the robot*

*@param beliefs - a two dimensional grid of floats representing*

*the robot's beliefs for each cell before sensing. For*

*example, a robot which has almost certainly localized*

*itself in a 2D world might have the following beliefs:*

*0.01 0.98*

*0.00 0.01*

*@param blurring - A number representing how noisy robot motion*

*is. If blurring = 0.0 then motion is noiseless.*

*@return - a normalized two dimensional grid of floats*

*representing the updated beliefs for the robot.*

*\*/*

vector< vector <**float**> > move(**int** dy, **int** dx,

vector < vector <**float**> > beliefs,

**float** blurring)

{

vector < vector <**float**> > newGrid;

*// your code here*

**return** blur(newGrid, blurring);

}

#### simulate.cpp

*/\*\**

*simulate.cpp*

*Purpose: implements a Simulation class which*

*simulates a robot living in a 2D world. Relies*

*on localization code from localizer.py*

*\*/*

**#include "localizer.cpp"**

**#include <algorithm>**

*// #include "helpers.cpp"*

**class** Simulation {

**private**:

vector <**char**> get\_colors() {

vector <**char**> all\_colors;

**char** color;

**int** i,j;

**for** (i=0; i<height; i++) {

**for** (j=0; j<width; j++) {

color = grid[i][j];

**if**(std::find(all\_colors.begin(), all\_colors.end(), color) != all\_colors.end()) {

*/\* v contains x \*/*

} **else** {

all\_colors.push\_back(color);

cout << "adding color " << color << endl;

*/\* v does not contain x \*/*

}

}

}

colors = all\_colors;

num\_colors = colors.size();

**return** colors;

}

**public**:

vector < vector <**char**> > grid;

vector < vector <**float**> > beliefs;

**float** blur, p\_hit, p\_miss, incorrect\_sense\_prob;

**int** height, width, num\_colors;

std::vector<**int**> true\_pose;

std::vector<**int**> prev\_pose;

vector <**char**> colors;

Simulation(vector < vector<**char**> >, **float**, **float**, vector <**int**>);

};

*/\*\**

*Constructor for the Simulation class.*

*\*/*

Simulation::Simulation(vector < vector <**char**> > map,

**float** blurring,

**float** hit\_prob,

std::vector<**int**> start\_pos

)

{

grid = map;

blur = blurring;

p\_hit = hit\_prob;

p\_miss = 1.0;

beliefs = initialize\_beliefs(map);

incorrect\_sense\_prob = p\_miss / (p\_hit + p\_miss);

true\_pose = start\_pos;

prev\_pose = true\_pose;

}

*/\*\**

*You can test your code by running this function.*

*Do that by first compiling this file and then*

*running the output.*

*\*/*

*// int main() {*

*// vector < vector <char> > map;*

*// vector <char> mapRow;*

*// int i, j, randInt;*

*// char color;*

*// std::vector<int> pose(2);*

*// for (i = 0; i < 4; i++)*

*// {*

*// mapRow.clear();*

*// for (j=0; j< 4; j++)*

*// {*

*// randInt = rand() % 2;*

*// if (randInt == 0 ) {*

*// color = 'r';*

*// }*

*// else {*

*// color = 'g';*

*// }*

*// mapRow.push\_back(color);*

*// }*

*// map.push\_back(mapRow);*

*// }*

*// cout << "map is\n";*

*// Simulation simulation (map, 0.1, 0.9, pose);*

*// // simulation = Simulation(map, 0.1, 0.9, pose);*

*// cout << "initialization success!\n";*

*// show\_grid(map);*

*// cout << "x, y = (" << simulation.true\_pose[0] << ", " << simulation.true\_pose[1] << ")" << endl;*

*// return 0;*

*// }*

#### tests.cpp

**#include <iostream>**

**#include "simulate.cpp"**

**bool** **test\_normalize**() {

vector < vector <**float**> > unnormalized, normalized, result;

unnormalized = zeros(2, 2);

normalized = zeros(2,2);

**int** i,j;

**for** (i=0; i<2; i++) {

**for**(j=0; j<2; j++) {

unnormalized[i][j] = 1.0;

normalized[i][j] = 0.25;

}

}

result = normalize(unnormalized);

**bool** correct;

correct = close\_enough(normalized, result);

**if** (correct) {

cout << "! - normalize function worked correctly!\n";

}

**else** {

cout << "X - normalize function did not work correctly.\n";

cout << "For the following input:\n\n";

show\_grid(unnormalized);

cout << "\nYour code returned the following:\n\n";

show\_grid(result);

cout << "\nWhen it should have returned the following:\n";

show\_grid(normalized);

}

**return** correct;

}

**bool** **test\_blur**() {

vector < vector <**float**> > in, correct, out;

in = zeros(3, 3);

correct = zeros(3,3);

in[1][1] = 1.0;

**float** corner = 0.01;

**float** side = 0.02;

**float** center = 0.88;

correct[0][0] = corner;

correct[0][1] = side;

correct[0][2] = corner;

correct[1][0] = side;

correct[1][1] = center;

correct[1][2] = side;

correct[2][0] = corner;

correct[2][1] = side;

correct[2][2] = corner;

out = blur(in, 0.12);

**bool** right;

right = close\_enough(correct, out);

**if** (right) {

cout << "! - blur function worked correctly!\n";

}

**else** {

cout << "X - blur function did not work correctly.\n";

cout << "For the following input:\n\n";

show\_grid(in);

cout << "\nYour code returned the following:\n\n";

show\_grid(out);

cout << "\nWhen it should have returned the following:\n";

show\_grid(correct);

}

**return** right;

}

**bool** **test\_helpers**() {

**bool** correct = true;

**bool** question\_correct;

question\_correct = test\_normalize();

**if** (!question\_correct) {

correct = false;

}

cout << endl;

question\_correct = test\_blur();

**if** (!question\_correct) {

correct = false;

}

**return** correct;

}

**bool** **test\_initialize**() {

vector < vector <**char**> > map;

map = read\_map("maps/m1.txt");

**int** h = map.size();

**if** (h < 1) {

cout << "failed to load map. Make sure there is a maps/ directory in the same directory as this file!\n";

**return** false;

}

vector < vector <**float**> > beliefs, correct;

beliefs = initialize\_beliefs(map);

**int** w, A;

**float** belief;

w = map[0].size();

A = h \* w;

belief = 1.0 / A;

**int** i, j;

vector <**float**> row;

**for** (i=0; i<map.size(); i++) {

row.clear();

**for** (j=0; j<map[0].size(); j++) {

row.push\_back(belief);

}

correct.push\_back(row);

}

**bool** right = close\_enough(correct, beliefs);

**if** (right) {

cout << "! - initialize\_beliefs function worked correctly!\n";

}

**else** {

cout << "X - initialize\_beliefs function did not work correctly.\n";

cout << "For the following input:\n\n";

show\_grid(map);

cout << "\nYour code returned the following:\n\n";

show\_grid(beliefs);

cout << "\nWhen it should have returned the following:\n";

show\_grid(correct);

}

**return** right;

}

**bool** **test\_move**() {

vector < vector <**float**> > in, out, correct;

in = zeros(3,3);

in[2][2] = 1.0;

**int** dx, dy;

dx = 1;

dy = 1;

**float** blurring = 0.0;

correct = zeros(3,3);

correct[0][0] = 1.0;

out = move(dy, dx, in, blurring);

**bool** right = close\_enough(correct, out);

**if** (right) {

cout << "! - move function worked correctly with zero blurring\n";

}

**else** {

cout << "X - move function did not work correctly.\n";

cout << "When dx=1, dy=1, blurring=0.0 and with\nthe following beliefs:\n\n";

show\_grid(in);

cout << "\nYour code returned the following:\n\n";

show\_grid(out);

cout << "\nWhen it should have returned the following:\n";

show\_grid(correct);

}

**return** right;

}

**bool** **test\_sense**() {

vector < vector <**float**> > in, out, correct;

in = zeros(4,2);

in[2][1] = 1.0;

**int** i,j;

**for** (i=0; i<in.size(); i++)

{

**for** (j=0; j<in[0].size(); j++) {

in[i][j] = 1.0/8.0;

}

}

**char** color = 'r';

vector < vector <**char**> > map;

map = read\_map("maps/half\_red.txt");

**float** p\_hit, p\_miss;

p\_hit = 2.0;

p\_miss = 1.0;

out = sense(color, map, in, p\_hit, p\_miss);

**float** total = 0.0;

**for** (i=0; i<out.size(); i++)

{

**for** (j=0; j<out[0].size(); j++) {

total += out[i][j];

}

}

**bool** right = true;

**if** ( (total < 0.99) || (total > 1.01) ) {

right = false;

}

**if** ( (out.size() != in.size()) || out[0].size() != in[0].size()) {

right = false;

cout << "X - sense function not working correctly.\n";

cout << "Your function returned a grid with incorrect dimensions.\n";

**return** right;

}

**float** r\_prob, g\_prob, r\_exp, g\_exp;

r\_prob = out[0][0];

g\_prob = out[0][1];

r\_exp = 1.0 / 6.0;

g\_exp = 1.0 / 12.0;

**if** (close\_enough(r\_prob, r\_exp) && close\_enough(g\_prob, g\_exp)) {

cout << "! - sense function worked correctly\n";

**return** false;

}

**else** {

cout << "X - sense function did not work correctly.\n";

cout << "When p\_hit=2.0, p\_miss=1.0 and with\nthe following beliefs:\n\n";

show\_grid(in);

cout << "\nYour code returned the following:\n\n";

show\_grid(out);

cout << "\nbut this is incorrect.\n";

}

**return** right;

}

**bool** **test\_localizer**() {

**bool** correct = true;

**bool** question\_correct;

question\_correct = test\_initialize();

**if** (!question\_correct) {

correct = false;

}

**if** (!correct) {

*// map could not be loaded*

**return** false;

}

cout << endl;

question\_correct = test\_move();

**if** (!question\_correct) {

correct = false;

}

cout << endl;

question\_correct = test\_sense();

**if** (!question\_correct) {

correct = false;

}

**return** correct;

}

*// bool test\_simulation() {*

*// // todo*

*// }*

**int** **main**() {

cout << endl;

test\_helpers();

test\_localizer();

cout << endl;

**return** 0;

}

#### maps/half\_red.txt

**r** g

g r

r r

g g

#### maps/m1.txt

**r** r r

r g r

r r r

#### maps/m2.txt

**r** g

r r

g g

NEXT

# Project Tips

### Project Files

There are only two files you need to worry about for this project: helpers.cpp and localizer.cpp. Here's what each file included in the starter code is for:

* You can mostly ignore any .h (header) files, although if you add additional functions to .cpp files you should also define them within the related header file!
* The maps folder just has data of the map data used in the project
* tests.cpp is just for testing - no need to touch it
* debugging\_helpers.cpp is to help you debug; don't need to implement anything
* helpers.cpp - Implement normalize() and blur()
* localizer.cpp - Implement initialize\_beliefs(), sense() and move()
* simulate.cpp - You can uncomment portions of this to further help you develop the project, but this is not needed just to pass. See more tips below if you want to delve deeper here.

### How to approach the TO-DO's

While there is no need to touch tests.cpp, it can help order your implementations, as you can unit test each function to see if it works before moving on further.

Therefore, the best method to approach the project is to write the code for one of these, compile your code (make sure to flag c++11!), then run tests.cpp. If that individual function passes the test without any errors, you can move onto the next one. If not, make sure to debug and fix it first!

### Initializing vectors & matrices in C++11

In older versions of C++, you could initialize the size of the vector or matrix to start, but you had to either replace each index or use .push\_back() to place values within the vector. In C++11, this is made easier as you can place the values to begin with, as shown below (assuming the various some\_vals shown below are already initialized or are a const):

vector < vector <**float**> > our\_matrix {{some\_val1, some\_val4, some\_val7},

{some\_val2, some\_val5, some\_val8},

{some\_val3, some\_val6, some\_val9}};

Note that the values in the above should be floats given how the inner vector is defined; if they are doubles, you will likely get a compilation error due to loss of precision.

### Modulo (%) in C++ and Negative Numbers

If you type -1 % 5 into either Python script or Google, you should get 4 as the answer. However, if you do the same in C++, you'll get 1.

Why? Well, it isn't actually modulo in C++, but the remainder! Check out [this Stack Overflow post](https://stackoverflow.com/questions/11630321/why-does-c-output-negative-numbers-when-using-modulo) if you want to read about why this is.

As you may remember, Sebastian used the modulo operator when writing his code in Python for localization. If you don't account for the difference between these implementations in Python vs. C++, you may end up with a **segmentation fault** when you try to call an index outside of the size of your vector!

If you're stuck on how to deal with this difference between Python and C++, take a look at the top answers [here](https://stackoverflow.com/questions/12276675/modulus-with-negative-numbers-in-c) for some useful tips.

### Segmentation Faults

One potential cause of these is concerning modulo as described above, but there is also another potential cause.

Depending on your implementation, after you've coded blur(), running the out file from tests.cpp may produce a segmentation fault when the tests reach move() (i.e. normalize(), blur() and initialize\_beliefs() will show their test results, and then the fault occurs). Typically this should resolve itself once newGrid within the move() function takes shape.

### Additional Simulation (optional)

While all you need to pass the project is to pass all tests in tests.cpp, you can also visualize a simulation of your localizer with simulate.cpp. In order to do so, the first step is to uncomment all the lines at the bottom of the file around the main() function.

From here, it is up to you how to proceed - the map is initialized for you, and you can then call your functions from localizer.cpp to see what happens.

To see the results of the simulation, you'd run:

g++ -std=c++11 simulate.cpp

Note that this is because you actually have a separate main() function here, so it is a completely separate program from tests.cpp.

There's one last item to note here - if you try to run tests.cpp, it actually uses the Simulation class from this file, and so trying to run tests.cpp while the main() function within simulate.cpp is uncommented will result in an error - make sure to comment it back out when submitting!