**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

# 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?

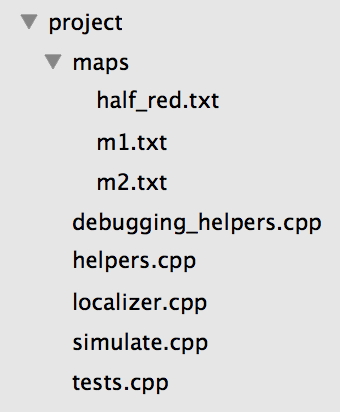
Students in China have not been able to download the project starter code. If you were able to download the starter code you can ignore everything below.

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:

**[[](https://classroom.udacity.com/nanodegrees/nd113/parts/8a7dde4d-a66a-4666-8ddb-783ed670263b/modules/c7c475c8-da86-4f23-93b1-639baaec3e4e/lessons/269ac8b0-5f8d-43a8-bb05-426c277496c0/concepts/5db92d72-4d35-40c0-995f-e14759515808)](https://classroom.udacity.com/nanodegrees/nd113/parts/8a7dde4d-a66a-4666-8ddb-783ed670263b/modules/c7c475c8-da86-4f23-93b1-639baaec3e4e/lessons/269ac8b0-5f8d-43a8-bb05-426c277496c0/concepts/5db92d72-4d35-40c0-995f-e14759515808)**

# 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!