**Project Overview**

Now it's time to apply what you've learned to Andy's histogram filter. In his code, you will find eight C++ files. Six of these files contain histogram filter functions. The other two are a main.cpp file and a file containing a function for printing out a matrix.

Your challenge is to read through his files, run the program, and then get his code to run faster.

**This project is non-graded**, so we are also providing a solution for you in the section labeled "Solution Code". There is not necessarily a wrong and right answer to this project; the goal is to get the code to run faster. The example solution shows a few possible ways to optimize.

**Project Files**

In the next part of the lesson, you will find an embedded workspace that contains the following files and folders:

* andy\_histogram\_filter folder
* optimized code folder
* instructions.md file

**andy\_histogram\_filter**

The andy\_histogram\_filter folder contains Andy's histogram filter code. There is a main.cpp file that runs and times each function of the histogram filter. The time results are outputted to the terminal.

The other .cpp files contain function definitions for the histogram filter.

**optimized\_code**

The optimized\_code folder contains a copy of Andy's histogram filter code as well. This is where you can tweak the code for optimization. These files have extra comments giving some ideas about what to optimize.

**instructions.md**

This file contains further instructions about how to complete and run the project.

**hints.md**

The optimized\_code files already have ideas about what to optimize. The hints file contains more explicit ideas but does not give the actual code. You are not required to read this file, and we encourage you to try optimizing the code first before reading the hints file.

**Explanation of the Code**

Here is an explanation of how the histogram filter code works.

In main.cpp, you'll find a vector called grid. Grid is a char vector holding the color values of a 2-dimensional square grid.

The **initialize\_beliefs** function takes in the char grid and outputs the initial probabilities for each square on the grid.

Then the **sense** function takes a measurement of the current grid space's color. The measurement is used to update the probabilities of each space on the grid.

Next, the **blur** function passes the grid through a smoothing algorithm.

Then the probabilities are normalized with the **normalize** function.

Finally, the robot moves to a new space on the board, and the probabilities are updated appropriately.

Each function is run ten-thousand times. You can adjust the number of times by changing the value in the iterations variable.

**Order**

As a suggestion, do the project in the following order:

1. Read through the instructions in instructions.md
2. Run the code in the andy\_histogram\_filter folder (see instructions.md for how to run the code)
3. Open main.cpp and look through the code. You do not need to change anything in this file. Note that each function is run the same number of times. And each function is wrapped with a clock.
4. Read through each of the .cpp files to become familiar with the code.
5. Optimize the code in the optimized\_code folder. For each optimization, make sure to run the code to see if the results have in fact improved.

Here is a suggested order for optimizing the files:

* zeros.cpp
* initialize\_beliefs.cpp
* sense.cpp
* blur.cpp
* normalize.cpp
* move.cpp

### **GCC Compiler Optimization**

In the classroom, you have been using a compiler called GCC. You've been compiling code with the following command:

g++ -std=c++11 main.cpp blur.cpp initialize\_beliefs.cpp move.cpp normalize.cpp print.cpp sense.cpp zeros.cpp

By default, gcc will try to lower the time it takes to compile your code; in other words, gcc optimizes for compilation time.

However, gcc can also optimize for execution time to get your code to run faster. The gcc compiler includes three levels of optimization, which you can use by adding the optimization flag to your compilation command: -O1 -O2 -O3

You can read more about what each level does at this link: [**Optimization Flags Link**](https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)

Now that you have optimized your histogram filter code, go back and try compiling the program with the level three flag. See how much the compiler can help you speed up your code.

Here is the command for compiling with level three optimization:

g++ -std=c++11 -O3 main.cpp blur.cpp initialize\_beliefs.cpp move.cpp normalize.cpp print.cpp sense.cpp zeros.cpp

### Project Solution

In the next section, we are providing a potential solution to the code optimization project. Each file contains comments noting what changes have been made to make the code run faster.

We suggest waiting to look at the solution until after you've tried optimizing the code yourself - see if you can do even better!

#### Potential Results

Here are some potential results between Andy's code, another Udacian's optimized code, as well as both using the -O3 flag at compile time.

| **Code** | **Sum of Runtimes** |
| --- | --- |
| Andy's original | ~350 ms |
| Udacian's optimized | ~135 ms |
| Andy's with -O3 | ~70 ms |
| Udacian's with -O3 | ~20 ms |

Optimizing your code, as well as using the appropriate compiler flags, can make a **huge** difference. Note that the most optimized result above takes only 6% of the time to run as Andy's original code.