

# Algorithm for Image Upsampling

Programming assignment 2

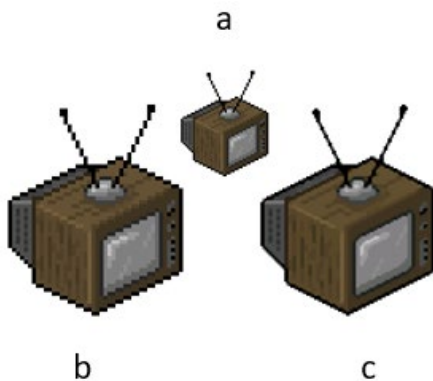
Due Date: November 23<sup>rd</sup>

---

A digital image can be rescaled in size. It could either be resized to a smaller image (down sampling) or into a bigger image (up sampling). Image quality should be considered while rescaling.

Example :

In Figure 1 upsampling (resizing to a larger size) is performed on image “a” with two algorithms  $f(a)$  and  $g(a)$ . The resultant images “b” and “c” are shown respectively.



As you could see both the functions gave output of same size but the image quality for function  $g$  is better than function  $f$ .

In this programming assignment we encourage you to come up with algorithms that output both the preferred size as well as better-quality images.

You will implement a program, `main.cpp`. It will read input matrices of size  $256 \times 256$  of type integers ranging from 0 to 255. The data is stored in row major order. The output of your program will be a  $512 \times 512$  matrix of the same type as input (2x enlargement).

There are many methods to do upsampling. In this homework you will at least implement two such methods and compare them on real data. In your final executable, you must only call the method that you think gives the best-looking output. Here is an example algorithm:

#### Nearest Neighbor

1	2
3	4



1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Input: 2 x 2

Output: 4 x 4

Other methods to do upsampling are bilinear, bicubic, lanczos, sinc, and others. Ideally we would like you to come up with your own algorithm to do upsampling. But you must implement at least one of the known methods.

**Compilation:** Compilation must be done by Makefile.

The total time of execution cannot exceed 3 seconds on Linprog. The algorithm cannot have complexity more than  $O(n \log n)$  where  $n$  is the total number of pixels in the image that is input.

#### Evaluation:

To evaluate your method, you should first crop your favorite image to size 512 x 512, and turn it into grayscale, using an image editor. Let us call this image **O**. Then scale it down using your editor to 256x256 and let's save this image (we will call it **L**). Then use your implementation to convert **L** to a 512x512 image called **H**. How well your algorithm does can be deduced using the L1 distance between **H** and **O** images using the following formula:

$$\text{L1 distance} = \sum_{i,j}^{512} |H_{ij} - O_{ij}|$$

The closer these two are, the better your algorithm is. Grading will be done in 200 points total.

First 100 points: To evaluate your algorithm, we will compute the L1 metric for the test data that you do not have access to and sort all the submissions based on how good they are.

Quality	Grade
Top 10 percentile	100
80-90 percentile rank	95
70-80 percentile	90
60-70	85
50-60	80
40-50	75
Below 50	70

Second 100 points:

- Gitlab commit messages 20 points (you can see them using **git log** command)  
- Note: Development of the project must happen with Gitlab using proper commit messages which will be considered as part of your documentation.
- Documentation 20 points (Function/class/cpp file documentation)
- Code readability 10 points (variable naming, easy to understand/maintain code)
- Execution speed 10 points. Must have at most  $O(n \log n)$  complexity.
- Self-testing 40 points.

**Input:** Sample1\_input\_image.txt

- A text file 256 rows x 256 columns data.
- Every row stored as a new line with comma separated 256 integers pixel values.

**Output:** Output\_image.txt

- A text file 512 rows x 512 columns data.
- Every row stored as a new line with comma separated 512 integers pixel values.

## Self-Testing :

You need to use the GoogleTest framework for UnitTesting the code.

## Installing GoogleTest:

1. cd ~
2. git clone <https://github.com/google/googletest.git>
3. cd googletest # Main directory of the cloned repository.
4. mkdir build # Create a directory to hold the build output.
5. cd build
6. cmake .. # Generate native build scripts for GoogleTest.
7. vim ../CMakeLists.txt
8. Add SET(CMAKE\_CXX\_FLAGS "-std=c++0x") in the begining of the file and save it
9. Now being in build folder itself give “make” command

After the eight step you should be able to see output as

```
Scanning dependencies of target gtest
[ 25%] Building CXX object googletest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
Linking CXX static library ../lib/libgtest.a
[ 25%] Built target gtest
Scanning dependencies of target gmock
[ 50%] Building CXX object googlemock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
Linking CXX static library ../lib/libgmock.a
[ 50%] Built target gmock
Scanning dependencies of target gmock_main
[ 75%] Building CXX object googlemock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
Linking CXX static library ../lib/libgmock_main.a
[ 75%] Built target gmock_main
Scanning dependencies of target gtest_main
[100%] Building CXX object googletest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
Linking CXX static library ../lib/libgtest_main.a
[100%] Built target gtest_main
```

10. Navigate to lib folder under build “cd lib” and perform “ls” you should be able to see libgtest.a and libgtest\_main.a and two other files.

This ensures that GoogleTest is installed successfully.

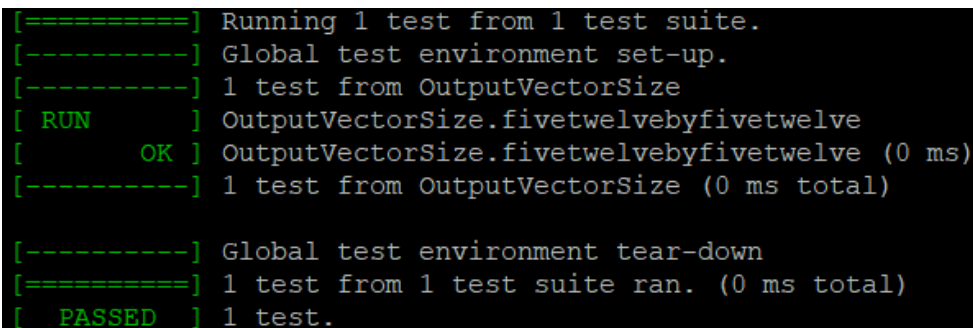
Navigate to project directory and write unit test cases for the source code, include the main.hpp but not main.cpp inside UnitTest.cpp( refer internet for unit testing using Gtest). Your main.cpp file should be as small as possible and all functions/classes and implementation complexity should be handled in main.hpp.

To run the unit tests we should be able to use “make test” with something like the following target in the Makefile:

test:

```
g++ -std=c++11 -I ~/googletest/googletest/include -L
~/googletest/build/lib UnitTest.cpp -o test -lgtest -lpthread
./test
```

The output of “make test” should look like the following:



```
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from OutputVectorSize
[ RUN    ] OutputVectorSize.fivetwelvebyfivetwelve
[       OK ] OutputVectorSize.fivetwelvebyfivetwelve (0 ms)
[-----] 1 test from OutputVectorSize (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[ PASSED ] 1 test.
```

## Code Coverage:

### Installing gcovr:

We use the gcovr for getting the code coverage.

1. cd ~
2. python3 -m pip install git+<https://github.com/gcovr/gcovr.git> --user

By now you can assume both GoogleTest and gcovr are installed.

To run coverage we would invoke “make coverage” with a target that perhaps looks like this:

coverage:

```
g++ -std=c++11 -fprofile-arcs -ftest-coverage -fPIC main.cpp  
-o coverage  
./coverage  
./coverage Sample1_input_image.txt  
~/local/bin/gcovr -r .
```

The output of “make coverage” should look like the following:

GCC Code Coverage Report				
Directory: .				
File	Lines	Exec	Cover	Missing
main.cpp	97	94	96%	31,149-150
TOTAL	97	94	96%	

### Makefile and Execution steps:

Your Makefile should have three targets, “all”, “test” and “coverage”:

1. make – Compiles all your code and creates the ./main executable that when run can upsample a given input file into an output file.
  - a. The usage of ./main is as follows:  
./main inputfile.txt outputfile.txt
2. make test – produce the output of GoogleTest and on terminal.
3. make coverage – produces the coverage output on terminal.

You need to come up with coverage of at least of 50%.

### Project folder:

Should contain source files and document, along with .git directory which tracked your development on git.

Provide documentation for implemented algorithm/s in document.

Zip the project folder and submit it through canvas and also provide access on gitlab. The gitlab URL must be provided in the README.txt file in the project root directory. Please give read permission of your gitlab repository to :

**@shivachaitanyas3**

You are not allowed to have GoogleTest and gcovr in folders in your zip file ( you can assume GoogleTest and gcovr are installed).

---

Additionally, the following should help you to understand and improve the coverage. This is not required to be shared in the zip file.

1. `~/local/bin/gcovr -r . --html > coverage.html`
2. `~/local/bin/gcovr -r . --html --html-details -o coverage-complete.html`

Coverage html:

GCC Code Coverage Report					
Directory: .		Exec		Total	Coverage
Date: 2020-11-11 18:14:08		Lines:	96	96	100.0 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %		Branches:	34	36	94.4 %
File	Lines		Branches		
main.cpp	<div></div>	100.0 %	96 / 96	94.4 %	34 / 36

Generated by: [GCOVR \(Version 4.2\)](#)

Complete Coverage html:

This gives a more readable output:

- No of lines covered.
- No of branches covered.
- No of cases in a branch that are covered.

Sample:

# GCC Code Coverage Report

Directory: .		Exec	Total	Coverage
File: example.cpp	Lines:	6	7	85.7 %
Date: 2018-07-02 23:02:54	Branches:	1	2	50.0 %

Line	Branch	Exec	Source
1			// example.cpp
2			
3		1	int foo(int param)
4			{
5	x✓	1	if (param)
6			{
7			return 1;
8			}
9			else
10			{
11		1	return 0;
12			}
13			}
14			
15		1	int main(int argc, char* argv[])
16			{
17		1	foo(0);
18			
19		1	return 0;
20			}
21			