

HW 2 - Content-Aware Image Resizing

Due Wednesday, February 1 at 8pm Eastern

Optional checkpoint due Wednesday, January 25

How to Read this Document

1. Read the “Introduction” and “High-Level Steps for Completing this Assignment”
2. Read the “Setup” section and follow the instructions in it
3. Read the “Submission and Grading” section
4. Refer to the specific sections on Matrix, Image, Processing, and Resize one at a time as you implement each part of the assignment.

Table of Contents

[HW 2 - Content-Aware Image Resizing](#)

[Introduction](#)

[High-Level Steps for Completing this Assignment](#)

[Submission and Grading](#)

[Unit Test Grading](#)

[Style Requirements](#)

[Setup](#)

[Group Registration](#)

[Starter Files](#)

[Create Stubs and Compile Your Code](#)

[Matrix Class Implementation Notes](#)

[Image Class Implementation Notes & PPM Format](#)

[PPM Format](#)

[Processing Module Implementation](#)

[Energy Matrix](#)

[Cost Matrix](#)

[Minimal Vertical Seam](#)

[Removing a Vertical Seam](#)

[Seam Carving Algorithm](#)

[Testing](#)

[Resize Program](#)

[Error Checking](#)

[Implementation](#)

[Acknowledgements](#)

Introduction

Build an image resizing program using a seam-carving algorithm.

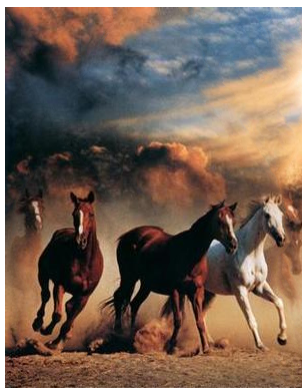
The learning of this project include Testing, Debugging, Structs & Classes, Strings, Streams & IO, translating algorithm descriptions into C++ code.

When you're done, you'll have a program that uses seam carving for content-aware resizing of images. The algorithm works by finding and removing "seams" in the image that pass through the least important pixels. For a quick introduction, check out this video: <https://www.youtube.com/watch?v=6NclJXTlugc>

Original Image: 479x382



Resized: 300x382



Resized: 400x250



High-Level Steps for Completing this Assignment

1. Download the starter files from Canvas & set up your git repository.
2. Read the assignment-specific style & code quality requirements.

3. Implement the Matrix member functions and tests for them. Submit Matrix.cpp and Matrix_tests.cpp to the autograder for feedback.
4. Implement the Image member functions and tests for them. Submit Image.cpp and Image_tests.cpp to the autograder for feedback.
5. Implement the processing functions (declarations in processing.hpp, implement them in processing.cpp). Include processing.cpp with the other files and submit to the autograder.
6. Implement the main function in resize.cpp. Include resize.cpp with the other files and submit to the autograder.

Submission and Grading

Submit the following files to the autograder:

- Matrix.cpp
- Matrix_tests.cpp
- Image.cpp
- Image_tests.cpp
- processing.cpp
- resize.cpp

For the optional checkpoint, submit Matrix.cpp, Matrix_tests.cpp, Image.cpp, and Image_tests.cpp.

This assignment will be autograded for correctness, comprehensiveness of your Matrix and Image test cases, and some aspects of style and code quality. This assignment will be hand graded for aspects of code quality that are not checked automatically.

Unit Test Grading

We will autograde your Matrix and Image unit tests.

Your unit tests must use the [unit test framework](#).

A test suite must complete less than 5 seconds and contain 50 or fewer TEST() items. One test suite is one _tests.cpp file. That is, you can submit one suite of Matrix tests in Matrix_tests.cpp and one suite of Image tests in Image_tests.cpp.

When testing the Matrix and Image functions that use streams (e.g., Matrix::print or Image::read_ppm), your unit tests must not depend on the presence of any input or output files in the filesystem, nor any input sent to cin or cout. Use stringstream when testing these functions instead.

To grade your unit tests, we use a set of intentionally buggy instructor solutions. You get points for catching the bugs.

1. We compile and run your unit tests with a correct solution.
 - Tests that pass are **valid**.
 - Tests that fail are **invalid**, they falsely report a bug.
2. We compile and run all of your **valid** tests against each **buggy solution**.
 - If any of your **valid** tests fail, you caught the bug.
 - You earn points for each bug that you catch.

Style/Code Quality Requirements

These requirements will be evaluated using a combination of automatic tools and manual inspection. We will post a separate document to Canvas with instructions on running the automatic tools on your own computer and will post an announcement when that document is available.

In addition to the requirements listed in the Coding Standards for the course, your code must meet the following requirements:

- You may not modify any of the provided .h or .hpp files.
- You may not use any libraries or language features outside of those available in Standard C++17
- Do not `#include` any header files or libraries in .hpp/.cpp files that do not use those headers.
- You may not use C-style strings, C-style arrays, pointers, or C-style I/O libraries. When processing command line arguments, convert values to C++ strings before doing anything else with them. Use only C++ I/O facilities.
- Use `assert` (defined in the `<cassert>` library) to guard against programmer invalid use of `Matrix`, `Image`, and processing functions.
- Wrap any non-member helper functions you add in the unnamed namespace. Do not use the “static” keyword for this purpose.
- While we will not grade your code for performance (other than some reasonably-high time limits), you must avoid “egregious inefficiency” in your code. “Egregious inefficiency” when code performs clearly-inefficient operations (such as making unnecessary copies of large objects) that do not provide any benefit to the understandability of the source code.
- You may not explicitly allocate dynamic memory in this assignment (e.g., no `new/delete`).
- When possible, use member initializer lists instead of assignments in the constructor body.

Setup

Create a project folder and git repository for this assignment like you did for HW1. You do not need to submit a screenshot.

Group Registration

Please register your partnership (or working alone) on the autograder. You will be prompted to register your partnership or declare you are working alone the first time you visit the project page on the autograder. An announcement will be posted on Canvas when the autograder for the assignment is open for submissions.

Starter Files

Download the starter files from Canvas, and unzip them in your HW 2 directory.

```
$ pwd
/home/jameslp/cs3520hw2
$ unzip starter-files.zip
$ ls
starter-files  starter-files.zip
```

Move the starter files from `starter-files/` to your present working directory (`.`), then you can delete the old `starter-files/` directory the zipfile.

```
$ mv starter-files/* .
$ rm -rf starter-files/ starter-files.zip
$ ls
```

Image.hpp	dog.ppm
Image_public_test.cpp	dog_4x5.correct.ppm
Image_test_helpers.cpp	dog_cost_correct.txt
Image_test_helpers.h	dog_energy_correct.txt
	dog_left.correct.ppm
	dog_removed.correct.ppm
Makefile	dog_right.correct.ppm
Matrix.hpp	horses.ppm
Matrix_public_test.cpp	horses_300x382.correct.ppm
Matrix_test_helpers.cpp	horses_400x250.correct.ppm
Matrix_test_helpers.h	horses_cost_correct.txt
	horses_energy_correct.txt
crabster.ppm	horses_left.correct.ppm
crabster_50x45.correct.ppm	horses_removed.correct.ppm
crabster_70x35.correct.ppm	horses_right.correct.ppm
crabster_cost_correct.txt	horses_seam_correct.txt
crabster_energy_correct.txt	processing.cpp.starter
crabster_left.correct.ppm	processing.hpp

```
crabster_removed.correct.ppm  processing_public_tests.cpp
crabster_right.correct.ppm    unit_test_framework.h
crabster_seam.correct.txt
```

Create files named Matrix.cpp and Image.cpp

```
$ touch Matrix.cpp Image.cpp
```

Rename each .cpp.starter file to a .cpp file.

```
$ mv Image_tests.cpp.starter Image_tests.cpp
$ mv Matrix_tests.cpp.starter Matrix_tests.cpp
$ mv processing.cpp.starter processing.cpp
```

Create the file resize.cpp:

```
$ touch resize.cpp
```

Your project directory should look like this.

```
$ ls
Image.cpp                dog.ppm
Image.hpp                dog_4x5.correct.ppm
Image_public_test.cpp    dog_cost_correct.txt
Image_test_helpers.cpp   dog_energy_correct.txt
Image_test_helpers.h     dog_left.correct.ppm
Image_tests.cpp          dog_removed.correct.ppm
Makefile                 dog_right.correct.ppm
Matrix.cpp               dog_seam_correct.txt
Matrix.hpp               horses.ppm
Matrix_public_test.cpp   horses_300x382.correct.ppm
Matrix_test_helpers.cpp  horses_400x250.correct.ppm
Matrix_test_helpers.h    horses_cost_correct.txt
Matrix_tests.cpp         horses_energy_correct.txt
crabster.ppm             horses_left.correct.ppm
crabster_50x45.correct.ppm horses_removed.correct.ppm
crabster_70x35.correct.ppm horses_right.correct.ppm
crabster_cost_correct.txt horses_seam_correct.txt
crabster_energy_correct.txt processing.cpp
crabster_left.correct.ppm processing.hpp
crabster_removed.correct.ppm processing_public_tests.cpp
crabster_right.correct.ppm resize.cpp
crabster_seam.correct.txt unit_test_framework.h
```

Here's a short description of each starter file.

File(s)	Description
Matrix.hpp	Interface specification for the <code>Matrix</code> class.
Image.hpp	Interface specification for the <code>Image</code> class.
processing.hpp	Specification of image processing functions that are pieces of the seam carving algorithm.
processing.cpp.starter	Starter code for the <code>processing</code> module.
Matrix_tests.cpp.starter	Starter code for unit testing the <code>Matrix</code> module.
Image_tests.cpp.starter	Starter code for unit testing the <code>Image</code> module.
Matrix_public_test.cpp	Public tests for the <code>Matrix</code> module.
Image_public_test.cpp	Public tests for the <code>Image</code> module.
processing_public_tests.cpp	Tests for the <code>processing</code> module and seam carving algorithm.
Matrix_test_helpers.h Matrix_test_helpers.cpp Image_test_helpers.h Image_test_helpers.cpp	Several helper functions you may use in your tests (e.g. <code>Matrix_equal</code> , <code>Image_equal</code>). Do not use these outside of your test code.

<code>unit_test_framework.h</code>	A simple unit-testing framework you will use in programming assignments. You MUST write your test cases using this framework. A tutorial from EECS 280 (I co-wrote this framework for that course) is available. We'll spend some time during lecture going over how to use the framework.
<code>dog.ppm</code> , <code>crabster.ppm</code> , <code>horses.ppm</code>	Sample input image files.
Several <code>_correct.txt</code> files. Several <code>.correct.ppm</code> files.	Sample (correct) output files used by the <code>processing_public_tests</code> program.
<code>Makefile</code>	Contains various <code>make</code> targets (compile command shortcuts) for convenience.

Create Stubs and Compile Your Code

Create stubs for every member function declared in the `Matrix` and `Image` classes (in files called `Matrix.cpp` and `Image.cpp`, respectively) and every function declared in `processing.hpp`. A “stub” is an empty function definition that we write so that our code will compile before we’ve implemented the function. Note: The stubs should go in the appropriate `.cpp` file for each corresponding `.hpp` file.

Add a simple main function to `resize.cpp` that prints a simple message like in HW 1.

Once you’ve written these stubs, you can use the commands below to compile different parts of your code. Each of these commands creates the file ending in “.exe” named in the command (you can run those .exe files, but they will fail at this point since all our functions are empty).

Note: The shortcut commands below may only work on Mac/Linux/WSL. We strongly recommend testing your code in an environment where you can use those commands. The `make` commands below use the following g++ flags: `--std=c++17 -Wall -Wextra -pedantic -g`. Your code must compile warning-free to receive full credit.

Command (Mac/Linux/WSL 2)	Description
<code>make Matrix_public_test.exe</code>	Compiles the public test for the Matrix class.
<code>make Matrix_tests.exe</code>	Compiles your Matrix test cases.
<code>make Image_public_test.exe</code>	Compiles the public test for the Image class.
<code>make Image_tests.exe</code>	Compiles your Image test cases.
<code>make processing_public_tests.exe</code>	Compiles the processing module public tests.
<code>make resize.exe</code>	Compiles the full image-resizing program.
<code>make test</code>	Compiles and runs the public Matrix, Image, processing, and resize tests provided for you.

Matrix Class Implementation Notes










The Matrix class contains three private member variables: width (number of columns), height (number of rows), and a 1D vector of integers. This class stores a 2D grid of integers in that 1D vector.

Interface	Implementation
<pre> column 0 1 2 3 4 0 [1 3 7 1 3] 1 [1 4 0 13 4] row 2 [3 -6 2 10 0] </pre>	<pre> width [5] height [3] data [1 3 7 1 3 1 4 0 13 4 3 -6 2 10 0] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 </pre>

Image Class Implementation Notes & PPM Format

An Image similar to a 2D Matrix, but it contains Pixels instead of integers. Each Pixel includes three integers, which represent red, green, and blue (RGB) color components.

Each component takes on an intensity value between 0 and 255. The Pixel type is considered "Plain Old Data" (POD), which means it doesn't have a separate interface. We just use its member variables directly. Here is the Pixel struct and some examples:

<pre>struct Pixel { int r; // red int g; // green int b; // blue }</pre>	 (255,0,0)	 (0,255,0)	 (0,0,255)
	 (0,0,0)	 (255,255,255)	 (100,100,100)
	 (101,151,183)	 (124,63,63)	 (163,73,164)

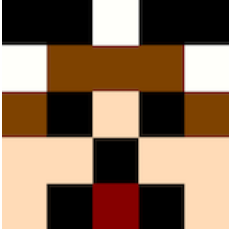
Below is a 5x5 image and its conceptual representation as a grid of pixels.



		column				
		0	1	2	3	4
row	0	(0,0,0)	(0,0,0)	(255,255,250)	(0,0,0)	(0,0,0)
	1	(255,255,250)	(126,66,0)	(126,66,0)	(126,66,0)	(255,255,250)
	2	(126,66,0)	(0,0,0)	(255,219,183)	(0,0,0)	(126,66,0)
	3	(255,219,183)	(255,219,183)	(0,0,0)	(255,219,183)	(255,219,183)
	4	(255,219,183)	(0,0,0)	(134,0,0)	(0,0,0)	(255,219,183)

PPM Format

The `Image` class also provides functions to read and write Images from/to the PPM image format. Here's an example of an `Image` and its representation in PPM.

Image	Image Representation in PPM
	<pre>P3 5 5 255 0 0 0 0 0 0 255 255 250 0 0 0 0 0 0 255 255 250 126 66 0 126 66 0 126 66 0 255 255 250 126 66 0 0 0 0 255 219 183 0 0 0 126 66 0 255 219 183 255 219 183 0 0 0 255 219 183 255 219 183 255 219 183 0 0 0 134 0 0 0 0 0 255 219 183</pre>

The PPM format begins with these elements, each separated by whitespace:

- `P3` (Indicates it's a "Plain PPM file".)
- `WIDTH HEIGHT` (Image width and height, separated by whitespace.)
- `255` (Max value for RGB intensities. We'll always use 255.)

This is followed by the pixels in the image, listed with each row on a separate line. A pixel is written as three integers for its RGB components in that order, separated by whitespace.

To write an image to PPM format, use the `Image::print` function that takes in a `std::ostream`. This can be used in conjunction with file I/O to write an image to a PPM file. The `Image::print` function must produce a PPM using whitespace in a very specific way so that we can use `diff` to compare your output PPM file against a correct PPM file. See the RME for the full details.

To create an image by reading from PPM format, use the `Image::read_ppm` "named constructor" static function that takes in a `std::istream`. This can be used in conjunction with file I/O to read an image from a PPM file. Because we may be reading in images generated from programs that don't use whitespace in the same way that we do, the `Image::read_ppm` function must accommodate any kind of whitespace used to separate elements of the PPM format (if you use C++ style I/O with `>>`, this should be no problem). Other than variance in whitespace (not all PPM files put each row on its own line, for example), you may assume any input to this function is in valid PPM format. (Some PPM files may contain "comments", but you do not have to account for these.)

Processing Module Implementation

The `processing` module contains several functions that perform image processing operations. Some of these provide an interface for content-aware resizing of images, while others correspond to individual steps in the seam carving algorithm.

The main interface for using content-aware resizing is through the `seam_carve`, `seam_carve_width` and `seam_carve_height` functions. These functions use the seam carving algorithm to shrink either an image's width or height in a context-aware fashion. The `seam_carve` function adjusts both width and height, but width is always done first. For this project, we only support shrinking an image, so the requested width and height will always be less than or equal to the original values.

Energy Matrix

`compute_energy_matrix`: The seam carving algorithm works by removing seams that pass through the least important pixels in an image. We use a pixel's energy as a measure of its importance.

To compute a pixel's energy, we look at its neighbors. We'll call them N (north), S (south), E (east), and W (west) based on their direction from the pixel in question (we'll call it X).

	0	1	2	3	4
0					
1			N		
2		W	X	E	
3			S		
4					

Neighbor Pixels

	0	1	2	3	4
0	1470	1470	1470	1470	1470
1	1470	1148	57	1148	1470
2	1470	1470	202	1470	1470
3	1470	1464	960	1464	1470
4	1470	1470	1470	1470	1470

Energy Matrix

The energy of X is the sum of the squared differences between its N/S and E/W neighbors:

$$\text{energy}(X) = \text{squared_difference}(N, S) + \text{squared_difference}(W, E)$$

The static function `squared_difference` is provided as part of the starter code. Do not change the implementation of the `squared_difference` function.

To construct the energy `Matrix` for the whole image, your function should do the following:

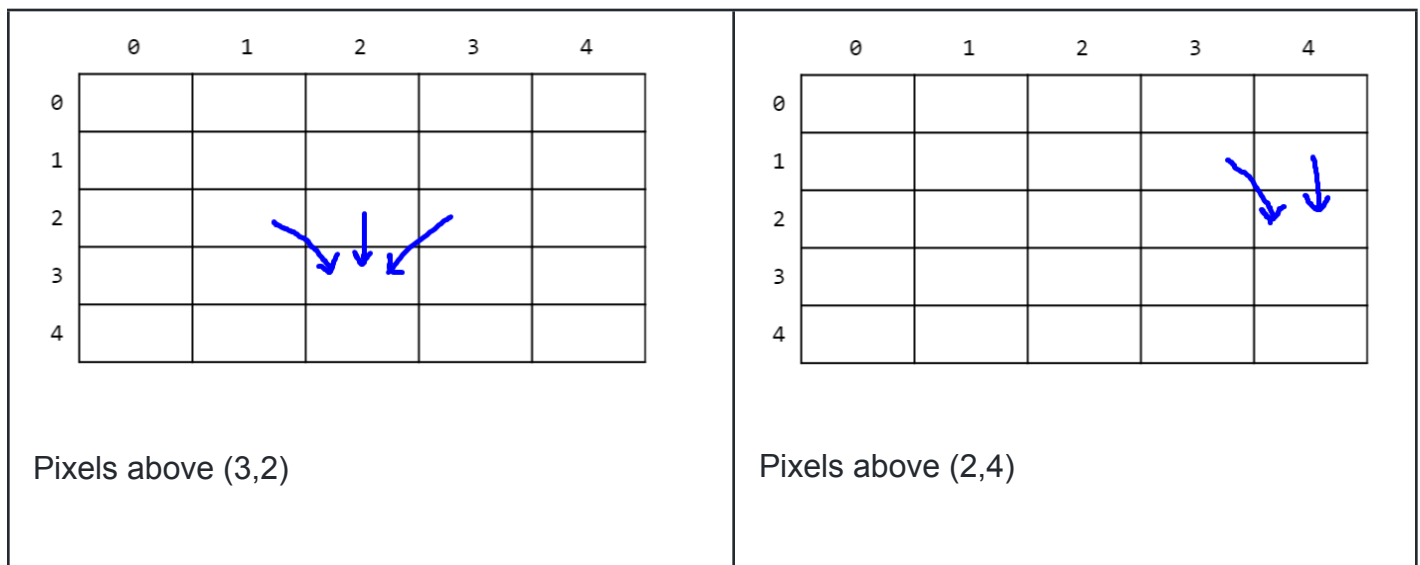
1. Initialize the energy `Matrix` with the same size as the `Image` and fill it with zeros.
2. Compute the energy for each non-border pixel, using the formula above.
3. Find the maximum energy so far, and use it to fill in the border pixels.

Cost Matrix

`compute_vertical_cost_matrix`: Once the energy matrix has been computed, the next step is to find the path from top to bottom (i.e. a vertical seam) that passes through the pixels with the lowest total energy (this is the seam that we would like to remove).

We will begin by answering a related question - given a particular pixel, what is the minimum energy we must move through to get to that pixel via any possible path? We will refer to this as the cost of that pixel. Our goal for this stage of the algorithm will be to compute a matrix whose entries correspond to the cost of each pixel in the image.

Now, to get to any pixel we have to come from one of the three pixels above it.



We would want to choose the least costly from those pixels, which means the minimum cost to get to a pixel is its own energy plus the minimum cost for any pixel above it. This is a recurrence relation. For a pixel with row r and column c , the cost is:

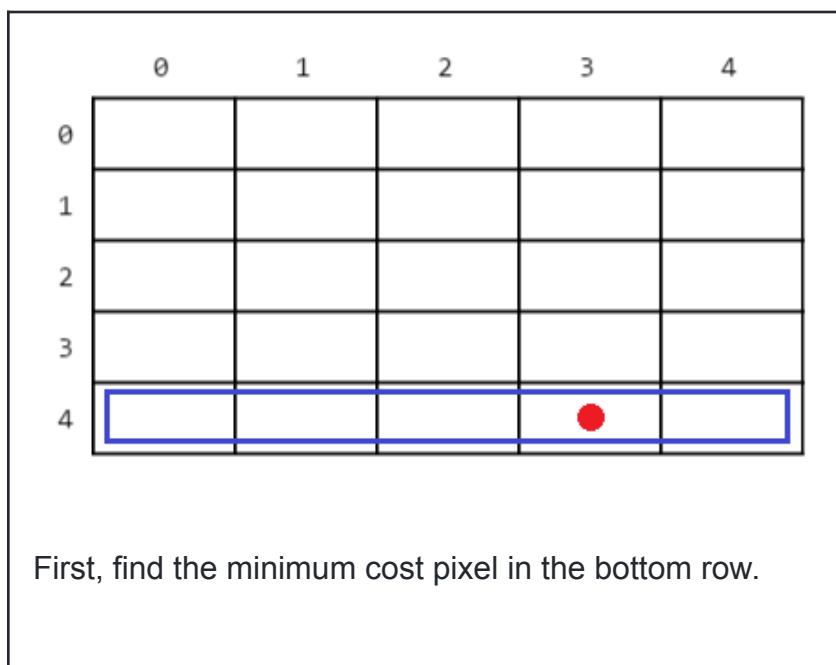
$$\text{cost}(r, c) = \text{energy}(r, c) + \min(\text{cost}(r-1, c-1), \text{cost}(r-1, c), \text{cost}(r-1, c+1))$$

We could compute costs recursively, with pixels in the first row as our base case, but this would involve a lot of repeated work since our subproblems will end up overlapping. Instead, let's take the opposite approach...

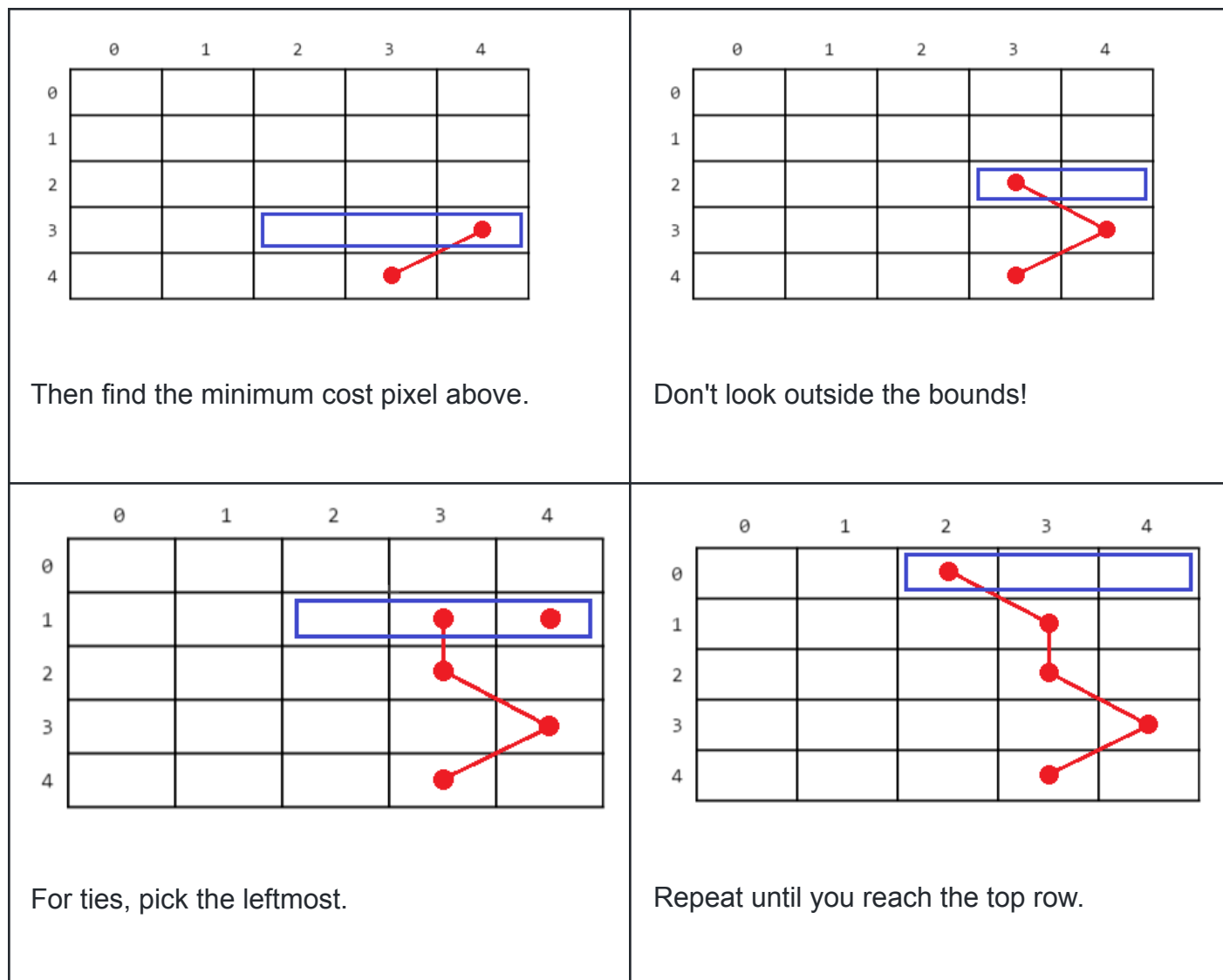
1. Initialize the cost `Matrix` with the same size as the energy `Matrix`.
2. Fill in costs for the first row (index 0). The cost for these pixels is just the energy.
3. Loop through the rest of the pixels in the `Matrix`, row by row, starting with the second row (index 1). Use the recurrence above to compute each cost. Because a pixel's cost only depends on other costs in an earlier row, they will have already been computed and can just be looked up in the `Matrix`.

Minimal Vertical Seam

`find_minimal_vertical_seam`: The pixels in the bottom row of the image correspond to the possible endpoints for any seam, so we start with the one of those that is lowest in the cost matrix.



Now, we work our way up, considering where we would have come from in the row above. In the pictures below, the blue box represents the "pixels above" in each step.



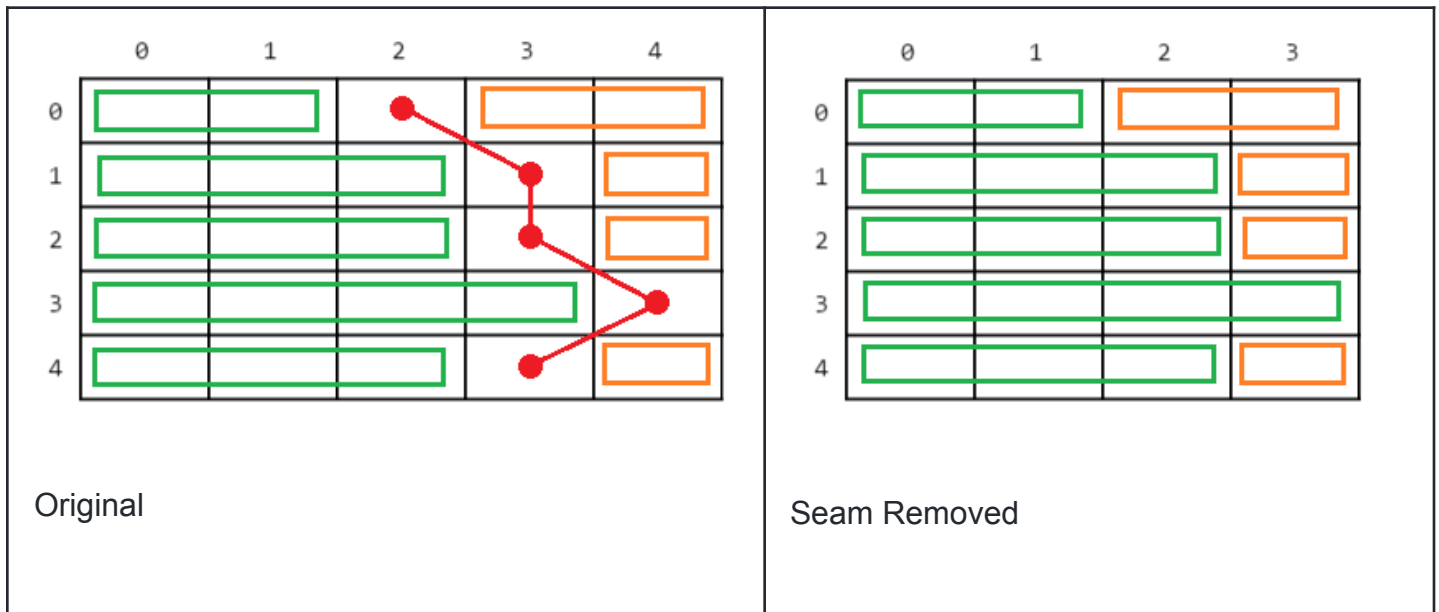
You will find the `Matrix::get_row_slice` function useful here. Each time you process a row, put the column number of the best pixel in the seam array, working your way from the back to front. (i.e. The last element corresponds to the bottom row.)

seam	2	3	3	4	3
	0	1	2	3	4

Removing a Vertical Seam

`remove_vertical_seam`: The seam vector passed into this function contains the column numbers of the pixels that should be removed in each row, in order from the top to

bottom rows. To remove the seam, copy the image one row at a time, first copying the part of the row before the seam (green), skipping that pixel, and then copying the rest (orange).



Seam Carving Algorithm

We can apply seam carving to the width of an image, the height, or both.

seam_carve_width

To apply seam carving to the width, remove the minimal cost seam until the image has reached the appropriate width.

1. Compute the energy matrix
2. Compute the cost matrix
3. Find the minimal cost seam
4. Remove the minimal cost seam

seam_carve_height

To apply seam carving to the height, just do the following:

1. Rotate the image left by 90 degrees
2. Apply `seam_carve_width`
3. Rotate the image right by 90 degrees

seam_carve

To adjust both dimensions:

1. Apply `seam_carve_width`
2. Apply `seam_carve_height`

Testing

We have provided the `processing_public_tests.cpp` file that contains a test suite for the seam carving algorithm that runs each of the functions in the `processing` module and compares the output to the `"_correct"` files included with the project.

You should write your own tests for the `processing` module, but you do not need to turn them in. You may do this either by creating a copy of `processing_public_tests.cpp` and building onto it, or writing more tests from scratch. Pay attention to edge cases.

Use the Makefile to compile the test with this command:

```
$ make processing_public_tests.exe
```

Then you can run the tests for the `dog`, `crabster`, and `horses` images as follows:

```
$ ./processing_public_tests.exe
```

You can also run the tests on just a single image:

```
$ ./processing_public_tests.exe dog
$ ./processing_public_tests.exe crabster
$ ./processing_public_tests.exe horses
```

When the test program runs, it will also write out image files containing the results from your functions before asserting that they are correct. You may find it useful to look at the results from your own code and visually compare them to the provided correct outputs when debugging the algorithm.

The seam carving tests work sequentially and stop at the first deviation from correct behavior so that you can identify the point at which your code is incorrect.

Resize Program

The main `resize` program supports content-aware resizing of images via a command line interface. For example, to resize the file `horses.ppm` to be 400x250 pixels and store the result in the file `horses_400x250.ppm`, we would use the following command:

```
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250
```

In particular, here's what each of those means:

Argument	Meaning
<code>horses.ppm</code>	The name of the input file from which the image is read.
<code>horses_400x250.ppm</code>	The name of the output file to which the image is written.
<code>400</code>	The desired width for the output image.
<code>250</code>	The desired height for the output image. (Optional)

The program is invoked with three or four arguments. If no height argument is supplied, the original height is kept (i.e. only the width is resized). If your program takes about 30 seconds for large images, that's ok. There's a lot of computation involved.

Error Checking

The program checks that the command line arguments obey the following rules:

- There are 4 or 5 arguments, including the executable name itself (i.e. `argv[0]`).
- The desired width is greater than 0 and less than or equal to the original width of the input image.
- The desired height is greater than 0 and less than or equal to the original height of the input image.

If any of these are violated, use the following lines of code (literally) to print an error message.

```
cout << "Usage: resize.exe IN_FILENAME OUT_FILENAME WIDTH [HEIGHT]\n"
      << "WIDTH and HEIGHT must be less than or equal to original" << endl;
```

Your program should then exit with a non-zero return value from `main`. Do not use the `exit` function in the standard library, as it does not clean up local objects.

If the input or output files cannot be opened, use the following lines of code (literally, except change the variable `filename` to whatever variable you have containing the name of the problematic file) to print an error message, and then return a non-zero value from `main`.

```
cout << "Error opening file: " << filename << endl;
```

You do not need to do any error checking for command line arguments or file I/O other than what is described in this section. However, you must use precisely the error messages given here in order to receive credit.

Implementation

Create a `resize.cpp` file and write your implementations of the driver program there.

Your `main` function should not contain much code. It should just process the command line arguments, check for errors, and then call the appropriate functions from the other modules to perform the desired task.

Use the Makefile to compile the driver with this command:

```
$ make resize.exe
```

Then you can run it with a command like:

```
$ ./resize.exe horses.ppm horses_400x250.ppm 400 250  
$ diff dog_4x5.out.ppm dog_4x5.correct.ppm
```

Acknowledgements

This assignment is based on a project written by James Juett, Winter 2016 at the University of Michigan for the course EECS 280.