

p2-cv

EECS 280 Project 2: Computer Vision

Project Due Friday, February 7 2020 8pm

Implementing this project provides an opportunity to work with pointers, arrays, structs, strings, and basic I/O operations, as well as C-style “object-based” programming.

This project will be autograded for correctness, comprehensiveness of your test cases, and programming style. See the [style checking tutorial](#) for the criteria and how to check your style automatically on CAEN.

You may work alone or with a partner. Please see the syllabus for partnership rules. As a reminder, you may not share any part of your solution outside of your partnership. This includes both code and test cases.

Table of Contents

- [Project Roadmap](#)
- [Project Introduction](#)
- [Code Structure, Modules, and Interfaces](#)
 - [The Matrix Module \(`Matrix.h` \)](#)
 - [The Image Module \(`Image.h` \)](#)
 - [The `processing` Module \(`processing.h` \)](#)
 - [The Driver Program \(`resize.cpp` \)](#)

- [Code Structure](#)
- [Implement and Test the `Matrix` and `Image` Modules](#)
 - [Implementing `Matrix` \(`Matrix.cpp` \)](#)
 - [Implementing `Image` \(`Image.cpp` \)](#)
 - [Testing `Matrix` and `Image`](#)
- [Implement and Test the Seam Carving Algorithm \(`processing.cpp` \)](#)
 - [Computing the Energy Matrix](#)
 - [Computing the Cost Matrix](#)
 - [Finding the Minimal Vertical Seam](#)
 - [Removing a Vertical Seam](#)
 - [The Seam Carving Algorithm](#)
- [Implement the Driver Program](#)
- [Requirements and Restrictions](#)
- [Appendix A: Working with PPM Files](#)
- [Appendix B: Perf Tutorial](#)
- [Appendix C: Project 2 Coding Practices Checklist](#)

Project Roadmap

This is a big picture view of how to complete this project. Most of the pieces listed here also have a corresponding section later on in the spec that goes into more detail.

This spec is organized so that the interfaces for each module are presented first, but make sure to read the implementation sections as well before starting work.

1. Read the Project Introduction

In this project, you will implement parts of the seam carving algorithm to support content-aware image resizing. See the first section below for an introduction.

2. Download the starter code and set up your visual debugger

Use the [tutorial from project 1](#) to get your visual debugger set up. Use this `wget` link <https://eecs280staff.github.io/p2-cv/starter-files.tar.gz> .

Before setting up your visual debugger, you'll need to rename each `.cpp.starter` file to a `.cpp` file.

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

```
$ mv processing.cpp starter_processing.cpp
```

You'll also need to create these new files and add function stubs.

```
$ touch resize.cpp
```

These are the executables you'll use in this project:

- Matrix_public_test.exe
- Matrix_tests.exe
- Image_public_test.exe
- Image_tests.exe
- processing_public_tests.exe
- resize.exe

If you're working in a partnership, set up [version control for a team](#).

3. Familiarize Yourself with the Code Structure.

Look over the three modules (`Matrix` , `Image` , and `processing`) and their interfaces.

4. Implement and Test the `Matrix` and `Image` Modules.

Implement the functions to match the RME interfaces.

5. Implement and Test the Seam Carving Algorithm and Driver Program

Implement and test each piece of the algorithm separately and all together with the driver.

6. Submit to the Autograder

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


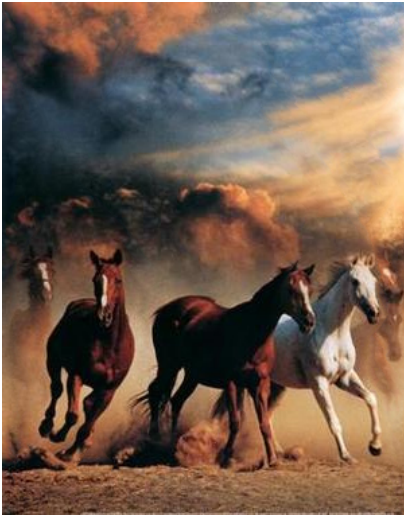

7. Optional Exercises

If you can't get enough of this project, check out the [optional exercises](#) for some reach goals. These are **not** required and **not** graded.

Project Introduction

In this section, we give a brief introduction to the seam carving algorithm. More details and guidelines for implementation can be found later in this project document.

Seam carving is a technique for content-aware resizing of images (sometimes known as “retargeting”). The end result is that we can resize images in a way that changes the aspect ratio but does not distort the image. Here’s an example.

 <p>Original Image: 479x382</p>	 <p>Resized: 300x382</p>	 <p>Resized: 400x250 Pixels</p>
---	---	--

These images aren’t cropped. 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 produced by the authors of the original paper on seam carving.

<https://www.youtube.com/watch?v=6NclJXTlugc>

We will only implement shrinking the dimensions of an image for this project (not enlarging or object removal). If the algorithm seems complicated, don’t worry! We walk through each part in detail in the implementation section further below.

The following table describes each file included in the starter code. Don’t worry about reading through the whole thing on a first reading of the spec, just use it as a reference.

File(s)	Description
Matrix.h	Interface specification for the <code>Matrix</code> module.
Image.h	Interface specification for the <code>Image</code> module.

<code>processing.h</code>	Specification of image processing functions that are pieces of the seam carving algorithm.
<code>Matrix.cpp.starter</code>	Starter code for the <code>Matrix</code> module. Rename to <code>Matrix.cpp</code> when you start.
<code>Image.cpp.starter</code>	Starter code for the <code>Image</code> module. Rename to <code>Image.cpp</code> when you start.
<code>processing.cpp.starter</code>	Starter code for the <code>processing</code> module. Rename to <code>processing.cpp</code> when you start.
<code>Matrix_tests.cpp.starter</code>	Starter code for unit testing the <code>Matrix</code> module. Rename to <code>Matrix_tests.cpp</code> when you start.
<code>Image_tests.cpp.starter</code>	Starter code for unit testing the <code>Image</code> module. Rename to <code>Image_tests.cpp</code> when you start.
<code>Matrix_public_test.cpp</code>	The public test case for the <code>Matrix</code> module.
<code>Image_public_test.cpp</code>	The public test case for the <code>Image</code> module.
<code>processing_public_tests.cpp</code>	Contains incremental and end-to-end tests for the <code>processing</code> module and seam carving algorithm.
<code>Matrix_test_helpers.h</code> <code>Matrix_test_helpers.cpp</code> <code>Image_test_helpers.h</code> <code>Image_test_helpers.cpp</code>	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>	The unit test framework from Lab 2. You MUST write your test cases using this 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 for convenience.

Authors

This project was written by James Juett, Winter 2016 at the University of Michigan. It was inspired by Josh Hug's "Nifty Assignment" at SIGCSE 2015.

Code Structure, Modules, and Interfaces

The following sections describe the modules and interfaces used throughout the project, which are specified in `Matrix.h`, `Image.h`, and `processing.h`. You will write implementations for these in the corresponding `.cpp` files as part of the project, but first we want to give you an idea of what they do. Do not modify the `.h` files!

The Matrix Module (`Matrix.h`)

A matrix is a two-dimensional grid of elements. For this project, matrices store integer elements and we will refer to locations by row/column. For example, here's a 5x3 matrix.

		column				
		0	1	2	3	4
row	0	1	3	7	1	3
	1	1	4	0	13	4
	2	3	-6	2	10	0

The `Matrix.h` file defines a `Matrix` struct to represent matrices and specifies the interface for functions to operate on them. The maximum size for a `Matrix` is given by constants in `Matrix.h`. Dimensions of 0 are not allowed.

To create a `Matrix`, first allocate storage and then use an initializer function.

```
Matrix* m = new Matrix; // allocate storage in dynamic memory
Matrix_init(m, 100, 100); // initialize it as a 100x100 matrix
```

The `new` operator allocates an object in dynamic memory, giving us a pointer to the newly created object. We will use dynamic memory to store `Matrix` and `Image` objects, since they are too large to be created as local variables on the stack.

Once a `Matrix` is initialized, it is considered valid. Now we can use any of the functions declared in `Matrix.h` to operate on it.

```
Matrix_fill(m, 0); // fill with zeros

// fill first row with ones
for (int c = 0; c < Matrix_width(m); ++c) {
    *Matrix_at(m, 0, c) = 1; // see description below
}

Matrix_print(m, cout); // print matrix to cout
```

Access to individual elements in a `Matrix` is provided through a pointer to their location, which can be retrieved through a call to `Matrix_at`. To read or write the element, you just dereference

The RMEs in `Matrix.h` give a full specification of the interface for each `Matrix` function.










When a `Matrix` object is no longer needed, its storage should be deallocated with the `delete` operator.

```
delete m;
```

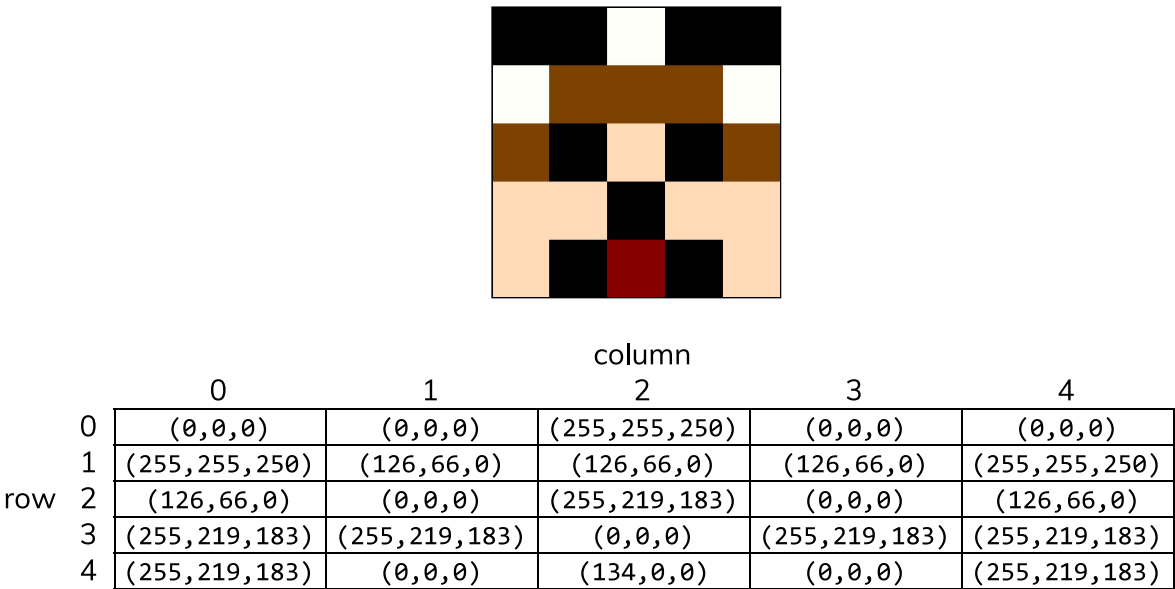
The Image Module (`Image.h`)

An `Image` is similar to a `Matrix`, but contains `Pixel`s 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:

```
struct Pixel {
    int r; // red
    int g; // green
    int b; // blue
}
```

		
(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.



The maximum size for an `Image` is given by constants in `Image.h`. Dimensions of 0 are not allowed. To create an `Image`, first allocate storage and then use an initializer function. There are several initializer functions, but for now we'll just use the basic one.

```
Image* img = new Image; // allocate storage in dynamic memory
Image_init(img, 5, 5); // initialize it as a 5x5 image
```

Once an `Image` is initialized, it is considered valid. Now we can use any of the functions declared in `Image.h` to operate on it.

```
Pixel um_blue = { 0, 46, 98 };
Pixel um_maize = { 251, 206, 51 };
Image_fill(img, um_blue); // fill with blue

// fill every other column with maize to make stripes
for (int c = 0; c < Image_width(img); ++c) {
    if (c % 2 == 0) { // only even columns
        for (int r = 0; r < Image_height(img); ++r) {
            Image_set_pixel(img, r, c, um_maize);
        }
    }
}
```

To read and write individual `Pixel`s in an `Image`, use the `Image_get_pixel` and `Image_set_pixel` functions, respectively.


The RMEs in `Image.h` give a full specification of the interface for each `Image` function.

When an `Image` object is no longer needed, its storage should be deallocated with the `delete` operator.

```
delete img;
```

Reading and Writing Images in PPM Format

The `Image` module also provides functions to read and write `Image`s 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 0 255 255 250 0 0 0 0 0 255 255 250 126 66 0 126 66 0 126 66 0 255 255 250 126 66 0 0 0 255 219 183 0 0 0 126 66 0</pre>



```
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
```

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_init` 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_init` 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.)

See [appendix A](#) for more information on working with PPM files and programs that can be used to view or create them on various platforms.

The `processing` Module (`processing.h`)

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.

Some of the functions involved in the seam carving algorithm should not necessarily be a part of

the interface for the processing module (e.g. `find_minimal_vertical_seam`) because it wouldn't make sense to call them individually "from the outside". We have nevertheless included them in `processing.h` to help you structure the code and so that individual pieces of the algorithm can be unit tested. This will make finding bugs throughout the algorithm much more manageable and also allows us to autograde and issue partial credit for individual functions that are correct.

See the RMEs for the individual functions for more details.

The Driver Program (`resize.cpp`)

The driver 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.

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 on this page. However, you must use precisely the error messages given here in order to receive credit.

Code Structure

The code structure in this project is modeled after C-style “object-based” programming. We will work with “manual” approaches to encapsulation, information hiding, and modular design. The C++ language provides additional features for “object-oriented” programming - we will cover these later in the course and you will have an opportunity to use them in project 3. Our hope is that you gain an appreciation for different approaches to the same goals in both C-style and C++ style programming.

Respect the Interfaces!

Our goal is to use several modules that work together through well-defined interfaces, and to keep the implementations separate from those interfaces. The interfaces consist of the functions we provide in the `.h` file for the module, but NOT the member variables of the struct. The member variables are part of the implementation, not the interface! (We didn’t even mention member variables when we were discussing the module interfaces above.)

This means you may access member variables directly (i.e. using `.` or `->`) when you’re writing code within their module, but never from the outside world! For example, let’s consider the `Image` module. If I’m writing the implementation of a function inside the module, like `Image_print`, it’s fine to use the member variable `height` directly:

```
void Image_print(const Image *img, std::ostream& os) {
    ...
    // loop through all the rows
    for (int r = 0; r < img->height; ++r) {
        // do something
    }
    ...
}
```

This is fine, because we assume the person who implements the module is fully aware of all the

details of how to use `height` correctly. However, if I'm working from the outside, then using member variables directly is very dangerous. This code won't work right:

```
int main() {
    // Make a 400x300 image (sort of)
    Image* img = new Image;
    img->width = 400;
    img->height = 300;
    // do something with img but it doesn't work :(
    ...
    delete img;
}
```

The problem is that we “forgot” about initializing the width and height of the `Matrix` structs that make up each color channel in the image. Instead, we should have used the `Image_init` function from the outside, which takes care of everything for us.

Here's the big idea - we don't want the “outside world” to have to worry about the details of the implementation, or even to know them at all. We want to support substitutability, so that the implementation can change without breaking outside code (as long as it still conforms to the interface). Using member variables directly from the outside messes this all up. Don't do it! It could break your code and this will be tested on the autograder!

An exception to this rule is the `Pixel` struct. It's considered to be a “Plain Old Data” (POD) type. In this case, the interface and the implementation are the same thing. It's just an aggregate of three ints to represent an RGB pixel - nothing more, nothing less.

We should note there are patterns used in C-style programming that hide away the definition of a struct's members and prevent us from accidentally accessing them outside the correct module. Unfortunately, this causes complications that we don't have all the tools to deal with yet (namely dynamic memory management). We'll also see that C++ adds some built-in language mechanisms to control member accessibility. For now, you'll just have to be careful!

Copying `Matrix` and `Image` Structs

In many cases you will find it useful to copy `Matrix` and `Image` structs in your code. This is supported by the interface, so feel free to use it wherever useful. **However, try to avoid making unnecessary copies, as this can slow down your code.** [Appendix B](#) gives a tutorial on a tool called Perf that can be helpful if your code is too slow! This usually shows up on the larger test cases such as “horses.”

As an example, let's say you wanted to add a border to a `Matrix` and print it without changing the original. You could write this:

...

```
// Assume we have a variable mat that points to a Matrix

// Make a copy of mat and add the border. original remains unchanged
Matrix* mat_border = new Matrix(*mat); // need to dereference mat to copy it
Matrix_fill_border(mat_border, 0);

// print the bordered version
Matrix_print(mat_border, os);
...
```

For posterity, we will note that the main reason it is fine to use the built-in copying behavior for the `Matrix` and `Image` structs is that neither holds an object or resource indirectly through a pointer. Thus, a straightforward member-by-member copy of the structs is sufficient and the

built-in copy is fine. If this doesn't seem relevant right now, just wait until we get to the "Big Three" in lecture :).

Implement and Test the `Matrix` and `Image` Modules

This section provides guidance for writing implementations of the functions in the `Matrix` and `Image` modules. Ultimately, correct implementations must follow the RME clauses specified in the header files, and those are the authoritative source for what you need to do from a correctness standpoint, but the information here should be useful to you when writing your implementations.

Implementing `Matrix` (`Matrix.cpp`)

Create a `Matrix.cpp` file and write your implementations for `Matrix` functions there.

The `Matrix` struct looks like this:

```
struct Matrix {
    int width;
    int height;
    int data[MAX_MATRIX_WIDTH * MAX_MATRIX_HEIGHT];
};
```

`Matrix` stores a 2D grid of numbers in an underlying one-dimensional array.

Interface	Implementation
<div>column</div> <div><div>01234</div></div>	<div>width<div>5</div></div> <div>height<div>3</div></div>

row	0	1	3	7	1	3									
	1	1	4	0	13	4									
	2	3	-6	2	10	0									

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

Each of the functions in the `Matrix` module takes a pointer to the `Matrix` that is supposed to be operated on. In your implementations of these functions, you should access the `width`, `height`, and `data` members of that `Matrix`, but this is the only place you may do so. To all other code, the individual members are an implementation detail that should be hidden behind the provided interfaces for the `Matrix` module.

Your `Matrix_at` functions will need to perform the appropriate arithmetic to convert from a (row,column) pair to an index in the array, and your `Matrix_row` and `Matrix_column` functions

will need to go in the other direction. **None of these functions require a loop, and you'll find your implementation will be very slow if you use a loop.**

There are two versions of the `Matrix_at` function to support element access for both const and non-const matrices. The constness of the pointer returned corresponds to the `Matrix` passed in. The implementations for these will be identical.

Remember that you may call any of the functions in a module as part of the implementation of another, and in fact you should do this if it reduces code duplication. **In particular, no function other than `Matrix_row`, `Matrix_column`, and the two versions of `Matrix_at` should access the `data` member directly – call one of these functions instead.** (However, you will not be able to call one version of `Matrix_at` from the other due to the differences in constness.)

Be Assertive!

Use assertions to check the conditions in the `REQUIRES` clause for each function whenever possible. This will alert you with a nice, clean error right away if a function was called with bad input, which is desirable. How is producing an error and crashing the program a good thing? If some other part of your code is breaking the interface for a module, that's a bug and you want to know right away!

Here's an example of assertions you could use in the `Matrix_at` function:

```
// REQUIRES: mat points to a valid Matrix
//           0 <= row && row < Matrix_height(mat)
//           0 <= column && column < Matrix_width(mat)
// EFFECTS: Returns a pointer to the element in
//           the Matrix at the given row and column.
int* Matrix_at(Matrix* mat, int row, int column) {
    assert(0 <= row && row < mat->height);
    assert(0 <= column && column < mat->width);

    // Implementation
    //
```

```
// ...
}
```

If we were to call `Matrix_at` with parameters that are outside the bounds of the `Matrix`, we get a failed `assert` instead of undefined behavior caused by potentially going outside the bounds of our data array. The failed `assert` is easier to debug, trust us.

Of course, you can't assert every single thing in the `REQUIRES` clause. Some of them can't possibly be checked. For example, you can't check that a pointer points to a valid `Matrix` (i.e. one that has been initialized). You also can't check that the size of an array parameter is large enough, since it's actually turned into a pointer parameter by the compiler and doesn't know its size.

Implementing `Image` (`Image.cpp`)

Create an `Image.cpp` file and write your implementations for `Image` functions there.

The `Image` struct looks like this:

```
struct Image {
    int width;
    int height;
    Matrix red_channel;
    Matrix green_channel;
    Matrix blue_channel;
};
```

The interface for `Image` makes it seem like we have a grid of `Pixel`s, but the `Image` struct actually stores the information for the image in three separate `Matrix` structs, one for each of the RGB color channels. There are no `Pixel`s in the underlying representation, so your `Image_get_pixel` function must pack the RGB values from each color `Matrix` into a `Pixel` to be returned. Likewise, `Image_set_pixel` must unpack RGB values from an input `Pixel` and store them into each `Matrix`.

Each of the functions in the `Image` module takes a pointer to the `Image` that is supposed to be operated on. When you are writing implementations for these functions, you may be tempted to access members of the `Matrix` struct directly (e.g. `img->red_channel.width`, `img->green_channel.data[x]`). Don't do it! They aren't part of the interface for `Matrix`, and you should not use them from the outside. Instead, use the `Matrix` functions that are part of the interface (e.g. `Matrix_width(&img->red_channel)`, `Matrix_at(&img->green_channel, r, c)`).

In your implementation of the `Image_init` functions, space for the `Matrix` members will have already been allocated as part of the `Image`. However, you still need to initialize these with a call to `Matrix_init` to ensure they are the right size!

The `Image` struct contains `width` and `height` members. These are technically redundant, since each of the `Matrix` members also keeps track of a width and height, but having them around should make the implementations for your functions easier to read.

Reading from PPM Format

As mentioned above in the section on the `Image` interface, the `Image_init` function that takes in a `std::istream` must support the use of any kind of whitespace to separate elements in the PPM format. The best way to ensure this is to use the `>>` (extraction) operator to read from the input stream. Because `>>` treats all whitespace equivalently, you shouldn't need to do anything special to ensure your function can read any PPM files (assuming they don't have comments, which you do not need to handle).

Testing Matrix and Image

Write unit tests for the `Matrix` and `Image` modules and turn them in with your submission.

With unit testing, the idea is that you test individual pieces (i.e. units) of the code to ensure that your implementations conform to specification in the RME. This does not always mean testing one function in isolation - for example, you can't test `Matrix_fill` without using another function like `Matrix_at` to check the elements.

Respect the interfaces for the modules you are testing. Do not access member variables of the structs directly. Do not test inputs that break the `REQUIRES` clause for a function.

Make sure to create `Matrix` and `Image` objects with the `new` operator, and then destroy them with `delete` when you are done. Do not declare a `Matrix` or `Image` on the stack, as they are too large to be placed in a stack frame. (Declaring a `Matrix*` or `Image*` is fine, since a pointer is small.)

You may use stringstream to simulate file input and/or output for your unit tests. You may also use the image files `dog.ppm`, `crabster.ppm`, and `horses.ppm`, but no others.

You should heed the Small Scope Hypothesis for these tests. There is no need to work with particularly large `Matrix` / `Image` structs. (Other than an as edge case for max size.) Think about what makes tests **meaningfully different** for the function at hand.

Writing the Tests

IMPORTANT - Your `Matrix` and `Image` test cases defined in `Matrix_tests.cpp` and `Image_tests.cpp` must be written using the unit test framework introduced in Lab 2. Read the [tutorial on the unit test framework](#) before writing your test cases.

You must write and submit tests for the `Matrix` and `Image` classes. Your test cases **MUST** use the unit test framework, otherwise the autograder will not be able to evaluate them. Since unit tests should be small and you quickly run into the 50,000 character limit, you should write your tests in a separate file and use a subdirectory for your tests.

tests should be small and run quickly, you are limited to **50** `TEST()` items per file and your whole test suite must finish running in less than 5 seconds. Please bear in mind that you **DO NOT** need 50 unit tests to catch all the bugs. Writing targeted test cases and avoiding redundant tests can help catch more bugs in fewer tests.

Rename the provided `Matrix_tests.cpp.starter` and `Image_tests.cpp.starter` files to `Matrix_tests.cpp` and `Image_tests.cpp`, respectively. Write unit tests in these files.

In your `Matrix` tests, you may use the functions provided in `Matrix_test_helpers.h`. In your `Image` tests, you may use the functions provided in `Matrix_test_helpers.h` and `Image_test_helpers.h` in your tests. **DO NOT use `Image_test_helpers.h` in your `Matrix` tests (the autograder will not like this).**

Use the Makefile to compile the tests with one of these commands:

```
$ make Matrix_tests.exe
$ make Image_tests.exe
```

Then you can run them with:

```
$ ./Matrix_tests.exe
$ ./Image_tests.exe
```

How We Grade Your Tests

We will autograde your `Matrix` and `Image` unit tests by running them against a number of implementations of those modules. If a test of yours fails for one of those implementations, that is considered a report of a bug in that implementation.

We grade your tests by the following procedure:

1. We compile and run your test cases with a **correct** solution. Test cases that pass are considered **valid**. Tests that fail (i.e. falsely report a bug in the solution) are invalid. The autograder gives you feedback about which test cases are valid/invalid. Since unit tests should be small and run quickly, your whole test suite must finish running in **less than 5 seconds**.
2. We have a set of intentionally **incorrect** implementations that contain bugs. You get points for each of these "buggy" implementations that your **valid** tests can catch.
3. How do you catch the bugs? We compile and run all of your **valid** test cases against each buggy implementation. If any of these test cases fail (i.e. report a bug), we consider that you have caught the bug and you earn the points for that bug.

Using Valgrind to Check for Undefined Behavior and Memory Leaks

If your code produces different results on different machine, the likely source is undefined

behavior. Refer to the [Valgrind tutorial](#) for how to use Valgrind to check for undefined behavior. You can also use Valgrind to make sure that you are using the `delete` operator on every object created by `new`.

Implement and Test the Seam Carving Algorithm (`processing.cpp`)

This section provides a detailed walkthrough of each step in the seam carving algorithm and guidance for writing implementations of the functions in the `processing` module.

This is a complex algorithm, but the abstractions provided by the `Matrix` and `Image` modules make it manageable. We hope you will appreciate the strength of ADTs here!

Start by renaming the provided `processing.cpp.starter` file to `processing.cpp`. Write your implementations for processing functions in this file.

Computing the 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.

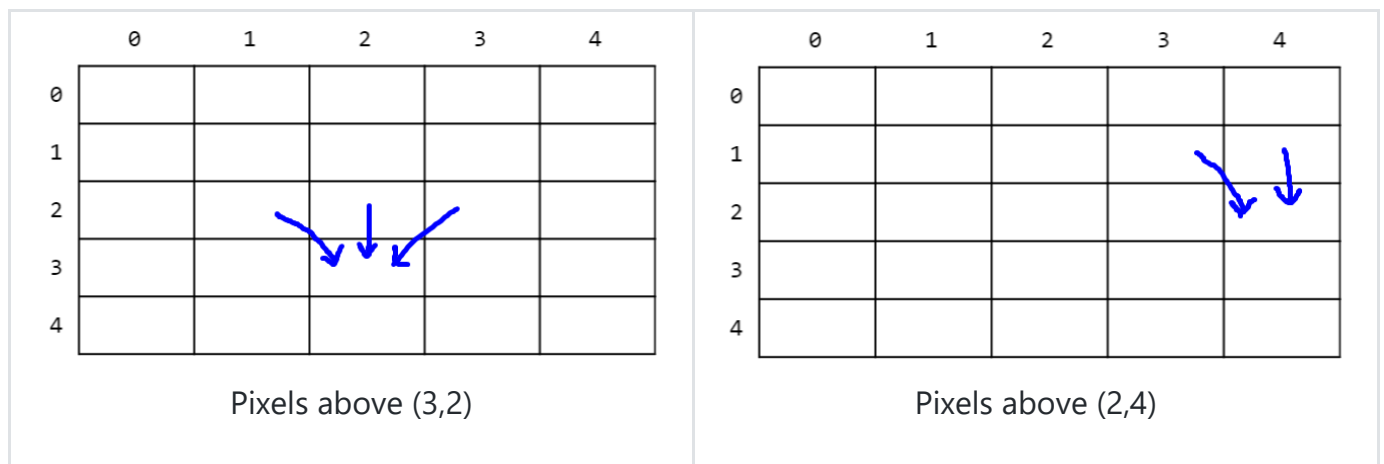
Computing the Cost Matrix

`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:

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

Use the `Matrix_min_value_in_row` function to help with this equation. Of course, you need to be careful not to consider coming from pixels outside the bounds of the `Matrix`.

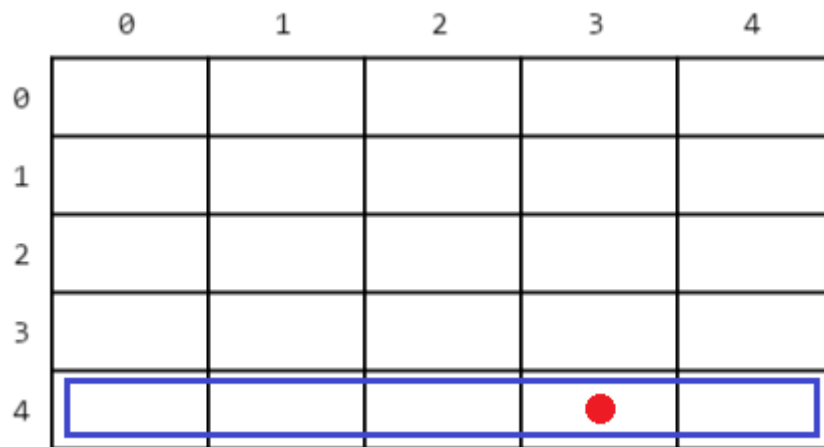
We could compute costs recursively using a top-down approach, with pixels in the first row as our base case, but this would involve a lot of repeated work since our subproblems will end up overlapping. Instead, let's take a bottom up approach...

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`.

Finding the Minimal Vertical Seam

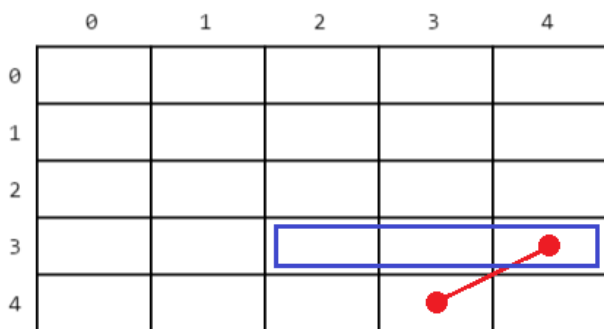
`find_minimal_vertical_seam`

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.

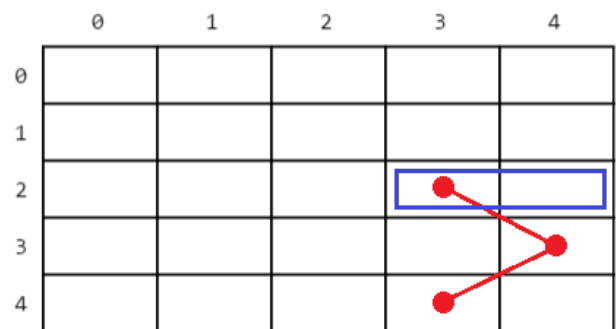


First, find the minimum cost pixel in the bottom row.

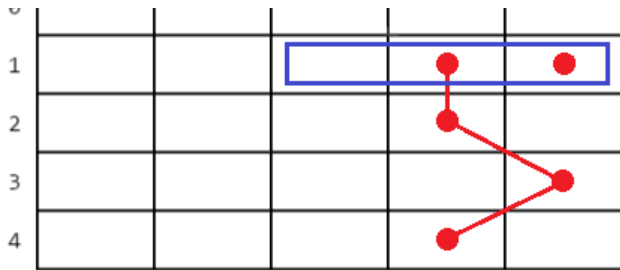
Now, we work our way up, considering where we would have come from in the row above. In the pictures below, the blue box represents the "pixels above" in each step.



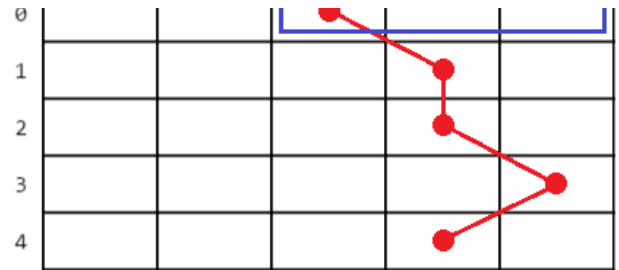
Then find the minimum cost pixel above.



Don't look outside the bounds!



For ties, pick the leftmost.



Repeat until you reach the top row.

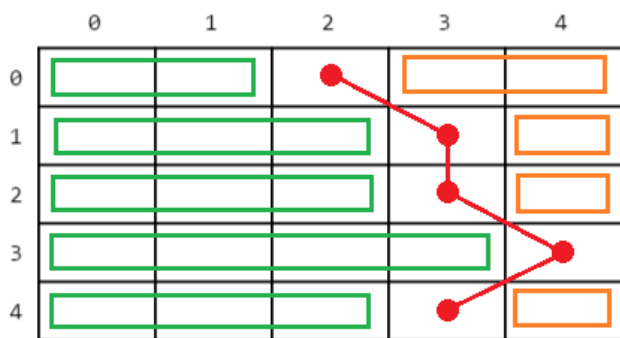
You will find the `Matrix_column_of_min_value_in_row` function useful here. Each time you process a row, put the column number of the best pixel in the seam array, working your way from the back to front. (i.e. The last element corresponds to the bottom row.)

seam	2	3	3	4	3
	0	1	2	3	4

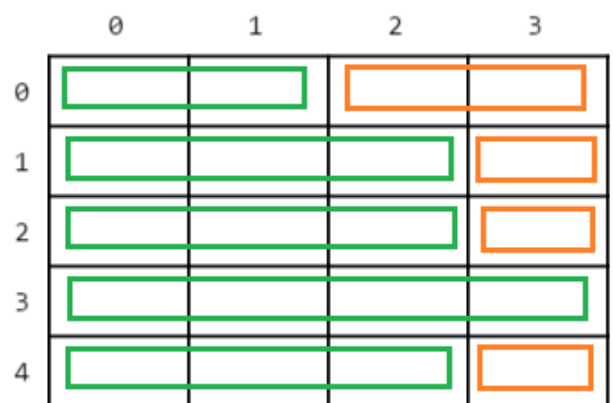
Removing a Vertical Seam

`remove_vertical_seam`

The seam array 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).



Original



Seam Removed

You should copy into a smaller auxiliary `Image` and then back into the original, because there is no way to change the width of an existing image. Do not attempt to use `Image_init` to "resize" the original - it doesn't preserve existing data in an `Image`.

The Seam Carving Algorithm

The Seam Carving Algorithm

seam_carve_width

The seam carving algorithm is just to repeatedly execute the procedure outlined above for removing 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 the Seam Carving Algorithm

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.

Implement the Driver Program

The driver program for this project provides a command-line interface to content-aware resizing through the seam carving algorithm. See the earlier section for more details.

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
```

Requirements and Restrictions

It is our goal for you to gain practice with good C-style object-based programming and proper use of pointers, arrays, and structs. Here are some (mandatory) guidelines.

DO	DO NOT
Use either traversal by pointer or traversal by index, as necessary	
Modify .cpp files	Modify .h files

modify .cpp files	modify .h files
Put any extra helper functions in the .cpp files and declare them static	Modify .h files
	#include a .h file from a module that does not require the code in the .h file (e.g. including Image.h from Matrix.cpp), as this introduces an incorrect dependency between modules
Use these libraries: <iostream> , <fstream> , <cstdlib> , <sstream> , <cassert> , <string>	Use other libraries

DO	DO NOT
#include a library to use its functions	Assume that the compiler will find the library for you (some do, some don't)
Use C-style strings when processing command line arguments	
Use C++ style strings when working with streams and file I/O	
const global variables	Global or static variables that are not const
Pass large structs or classes by pointer	Pass large structs or classes by value
Pass by pointer-to-const when appropriate	"I don't think I'll modify it ..."
Use dynamic memory (new and delete) to create and destroy Matrix and Image objects in processing.cpp , resize.cpp , and test cases	Use dynamic memory anywhere else (Matrix.cpp , Image.cpp)

See the [coding practices checklist](#) for more specific guidelines on writing your code.

Appendix A: Working with PPM Files

Viewing PPM Files

If you're logged into a CAEN computer or working remotely with VNC, you can just double click on PPM files and they will open in Image Viewer. The GIMP is also available on CAEN computers and displays the images more clearly than Image Viewer.

If you're on Windows, a number of programs can view PPM files, but two options are the GIMP (<https://www.gimp.org/>) and IrfanView (<http://download.cnet.com/IrfanView/>).

If you're on a Mac, you can use ToyViewer (<https://itunes.apple.com/us/app/toyviewer/id414298354>).

Creating PPM Files

We've provided you some PPM files to work with, but here's how you can create more. If you're on CAEN or Ubuntu Linux, you can use the ImageMagick `convert` tool to convert files in other image formats to PPM. It might need to be installed at the command line via:

```
$ sudo apt install imagemagick
```

Use it like this:

```
$ convert imageFile.png -compress none imageFile.ppm
```

You can also use `convert` in the other direction:

```
$ convert imageFile.ppm imageFile.png
```

If you're on another distribution of Linux, Windows, or a Mac, you may not have the ImageMagick suite already, but you should be able to install it. The process seems more or less complicated depending on the platform, but if you would like to try, visit <http://www.imagemagick.org/>.

Otherwise, several programs exist that are able to save images into PPM format. One that works across all platforms is the GIMP. You can find it online at <https://www.gimp.org/>. Note that the GIMP may include comments in saved PPM files, which must be removed.

Comments in PPM Files

The images we provide with the project don't have any comments, but some programs that generate PPM images may include comments (e.g. the GIMP), which are not supported by the `Image_init` function (i.e. you don't have to account for them in your implementation). If you're on CAEN or Ubuntu Linux, you can use the `sed` tool to remove all comments from a PPM file with a command like this:

```
$ sed -e 's/#.*$/' -e '/^\$$/d' -i image11e.ppm
```

Appendix B: Perf Tutorial

Perf is a tool used to profile code. Profiling code involves dynamically monitoring code execution to measure time, space, number of function calls, or usage of particular instructions. It is usually used after program completeness and correctness to analyze efficiency. Perf is a tool that is used to do just that. This tutorial will walk through steps to profile code in the space of the image processing project in EECS 280 with the solution to the project. You can use this as a starting point for comparison if your project is taking too long. **Do not profile your code until it gives the correct output.**

Step 1: Compile code for profiling by using `-g`. Our Makefile does this by default.

```
$ make processing_public_tests.exe
```

Step 2: Use Perf's `record` command with the `-g` flag. Pass in any command line arguments accordingly. The results below may vary. This command should create a file called "perf.data".

```
$ perf record -g ./processing_public_tests.exe crabster
Testing crabster rotate left...PASS
Testing crabster rotate right...PASS
Testing crabster energy...PASS
Testing crabster cost...PASS
Testing crabster find seam...PASS
Testing crabster remove seam...PASS
Testing crabster seam carve 50x45...PASS
Testing crabster seam carve 70x35...PASS
crabster tests PASS
```

```
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.032 MB perf.data (162 samples) ]
```

Step 3: Use Perf's `report` command to generate a call graph of the execution of the code.

```
$ perf report
```

Samples: 268 of event 'cpu-clock:uhM', Event count (approx.): 67000000					
Children	Self	Command	Shared Object	Symbol	
+ 84.70%	0.00%	processing_publ	processing_public_tests.exe	[.] test_all	
+ 84.70%	0.00%	processing_publ	processing_public_tests.exe	[.] main	
+ 84.70%	0.00%	processing_publ	libc-2.17.so	[.] __libc_start_main	
+ 70.15%	0.00%	processing_publ	processing_public_tests.exe	[.] test_seam_carve	
+ 69.78%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve	
+ 66.04%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve_width	
+ 42.16%	42.16%	processing_publ	libc-2.17.so	[.] __memcpy_ssse3_back	
+ 20.90%	0.00%	processing_publ	processing_public_tests.exe	[.] seam_carve_height	
+ 19.03%	12.31%	processing_publ	processing_public_tests.exe	[.] Image_get_pixel	
+ 14.55%	1.12%	processing_publ	processing_public_tests.exe	[.] compute_energy_matrix	
+ 11.57%	0.75%	processing_publ	processing_public_tests.exe	[.] compute_vertical_cost_matrix	
+ 9.70%	9.70%	processing_publ	processing_public_tests.exe	[.] Matrix_at	

```

+ 8.58% 3.36% processing_public_tests.exe [...] Matrix_column_of_min_value_in_row
+ 8.21% 0.75% processing_public_tests.exe [...] Matrix_min_value_in_row
+ 7.84% 0.00% processing_public_tests.exe [...] remove_vertical_seam
+ 5.60% 0.00% processing_public_tests.exe [...] 0xffff80eca0abb340
+ 5.60% 0.00% processing_public_tests.exe [...] std::basic_ofstream<char, std::char_traits<char>
+ 5.60% 0.00% processing_public_tests.exe [...] 0x48087f8d48fb0948
+ 4.48% 0.00% processing_public_tests.exe [...] test_rotate
+ 4.10% 4.10% processing_public_tests.exe [...] std::num_get<char, std::istreambuf_iterator<char>
+ 4.10% 4.10% processing_public_tests.exe [...] Matrix_at
ip: Treat branches as callchains: perf report --branch-history

```

These results show the percentage of execution time for each function. In the above image you can see that execution spends 14.55% of the time in the symbol known as `compute_energy_matrix`. These percentages will vary between runs. The command that generated this report is `processing_public_tests.exe`.

Step 4: Navigate through the call trees of functions with the arrow keys by highlighting a function in question and pressing enter.

```

- 14.55% 1.12% processing_public_tests.exe [...] compute_energy_matrix
- 13.43% compute_energy_matrix
+ 11.19% Image_get_pixel
+ 1.12% squared_difference
+ 0.75% Matrix_max

```

`compute_energy_matrix` is spending the majority of its execution by calling the function `Image_get_pixel`. This may or may not be problematic. Our job now becomes investigating certain functions that could be “bottlenecks” in the execution of our code.

Step 5: Given the output of Perf, determine which functions are possibly taking too much of the execution time.

Compare these results from the solution with yours. If any functions are near the top that aren’t in the solution, that might be a good place to start looking to optimize. Remember that your percentages will likely never be the identical to the solution’s. Things to look for are unnecessary loops, function calls, or objects passed by copy. **Again, don’t use this tool until your code gives the correct output!**

Appendix C: Project 2 Coding Practices Checklist

The following are coding practices you should adhere to when implementing the project. Adhering to these guidelines will make your life easier and improve the staff’s ability to help you in office hours. You **do not** have to submit this checklist.

General code quality:

- ☐ Helper functions used if and where appropriate
- ☐ Lines are not too long
- ☐ Descriptive variable and function names (i.e. `int radius` instead of `int x`)

- ☐ Descriptive variable and function names (i.e. `int radius` instead of `int x`)
- ☐ Effective, consistent, and readable line indentation
- ☐ Code is not too deeply nested in loops and conditionals
- ☐ Main function is reasonably short

Test case quality:

- ☐ Test cases are small and test one behavior each.
- ☐ Test case names are descriptive, or test cases are commented with a short description of what they test.
- ☐ Test cases are present for every public function.
- ☐ Test cases use the unit test framework from Lab 2.

Project-specific quality:

- ☐ The `Matrix` and `Image` interfaces are respected.
- ☐ `Matrix_at` is used by all other functions in the `Matrix` module to access individual elements.
- ☐ `Image_get_pixel` and `Image_set_pixel` are used by all other functions in the `Image` module to access individual pixels.
- ☐ `assert` is used to verify `REQUIRES` clauses where possible.