

Week 9 material posted!

X

Midterm Project

CAUTION

You are expected to work in a team of two.

Due: Wednesday October 28th at 11pm EDT (Baltimore time).

General Requirements

This is a programming assignment, meaning you will write and submit the source code for a program.

DANGER

All programs **must** compile (and run) on the ugrad machines using our standard compiler options.

This is a team assignment of two students. You are to work with your partner on this project, and should take a "paired programming" approach, where you frequently work together at a single screen and code together, in particular when you are testing and debugging your program. Remember, you are all responsible for all the code you submit.

WORKING REMOTELY

To work remotely together, you may consider granting access control to your partner during a Zoom session so that both of you can work on a shared screen to code, test and debug the program together.

As before, you are required to use the `git` version control system and include a *log file* with your project submission.

CAUTION

We will provision you a separate repo for your midterm project under `jhu-ip` organization and both team members along with course staff will be invited to have admin role on the repo.

This project is larger and more complex than previous programming assignments. Additionally, since you have now been exposed to most of what C can do, there are no restrictions on what language features you can use

(other than the conventions of good style, of course; you still shouldn't use global variables, for example).

TIP

From this point forward in the semester, your programs will need to not only compile and run cleanly (i.e. with **no warnings, errors, or crashes**), but also not have any **memory leaks** (as reported by valgrind).

INFO

For this project, you will be working with images. This will mean that you will want to view images, which will that if you want to work remotely on the ugrad machines, you will need to set up X-tunnelling using Xming on Windows or XQuartz on Mac (Reach out to us if you need help to set this up).

Also, PPM images can be quite large; be aware of file sizes, and try not to fill up your hard drive (or your disk quota on ugrad) with thousands of cat pictures...

Program Description

This program will be an image processing program, in the vein of Photoshop. Yours will have a command-line-based user interface (UI), so there will be no graphical interface, and the range of operations will be limited, but the algorithms you will use are the similar ones used in programs like Photoshop or GIMP.

At a basic level, your program will be able to read image files from disk, perform one of a variety of image processing tasks, and then write the result back to disk as a new image file. Since your program will not have a GUI, you will use external programs to view the images. If you are on ugrad (either locally, or remotely with X-tunnelling), you can use the program `feh`.

INFO

`feh` is a very simple command-line image viewer. It is available on ugrad machines and you can simply run the program with the name of an image file as a command-line argument, and it will display the image on your screen. e.g.

```
$ feh myimage.ppm
```

Note: Before running `feh`, make sure either Xming or Xquartz is running and x-tunneling is enabled (i.e. add `-Y` on mac when you `ssh` in to ugrad. On Putty, enable x-11 forwarding option)

If you are using a different platform, you are welcome to use an image viewer of your choice; `feh` is easy to install using most linux package managers, but there are other open source image viewing programs, as well as alternatives for Windows and OSX.

While there are many formats for storing image files, your program will **only** need to read and write one, the **PPM** format. This is essentially the simplest and easiest format to read and write, which is why it was chosen; its main drawback is that it does not use any kind of compression, so images stored in this format tend to be on the large side when compared to formats like JPEG or PNG or GIF. An implementation to read and write PPM file is provided for you. (See `ppm_io.h` and `ppm_io.c` in the starter code.)

CAUTION

Make sure to do a `git pull` on the public repo before starting to work to get the starter files for this project. You must work with the starter files!

Your program will be a command line tool, always run with the name of the executable file `project` followed by the name of an input PPM file, the name of a desired output PPM file, and the specific lower-case name of an image processing operation, as listed below. Some operations require additional arguments, which will also be supplied at the command line by the user, at the end of the line. There is no input entered by the user interactively.

INFO

Regardless of the desired operation, the first two arguments after the executable name `project` are always interpreted as the input file name followed by the output file name. The next argument is always interpreted as the operation name, and the operation's arguments (if any) come after that.

The operations your program will be able to recognize and perform are all of the following (the bolded words are the operation names to be entered at the command line by the user):

1. **grayscale** - covert the input image to grayscale (i.e. a full-color image becomes shades of gray)
2. **binarize** - convert the input image to black and white by thresholding
3. **crop** - crop the input image given corner pixel locations
4. **transpose** - transpose the input image (i.e. flip it along the diagonal)
5. **gradient** - compute the gradient of an input image and save its magnitude as an image
6. **seam** - a simplified version of **seam carving**

For example, at the command prompt, a user of your program might type:

```
$ ./project building.ppm building_crop.ppm crop 50 50 500 500
```

to crop the input image `building.ppm` (in PPM format) and output the cropped image to `building_crop.ppm`, where `(50, 50)` and `(500, 500)` specify the top-left and bottom-right pixel locations to crop. Or, if the desire is to seam carve the input image by half in both dimension of the input image, then the user could run:

```
$ ./project trees.ppm trees_seam.ppm seam 0.5 0.5
```

to generate the output file named `trees_seam.ppm` , which is a seam carved version of `tress.ppm` .

When you first begin this project, we suggest that you checkout the provided demo program `checkerboard` . You can compile the demo program by running `make checkerboard` . An executable `checkerboard` should be generated. The program demonstrates how to use `ppm_io` to read and write PPM formatted files, and also shows how the `struct Pixel` and `struct Image` are used.

TIP

In the starter code, we also provide a helper executable `img_cmp` , which you could run on ugrad machine to compare if two PPM files are the same up to a tolerance. It's usage is:

```
$ ./img_cmp <PPM file1> <PPM file2> [tolerance = 0]
```

The program takes two PPM files with the same dimension and compare them pixel by pixel. It counts how many pairs of pixels are within the given tolerance. A pair of pixel (with same row and col indices) of two images is said within the tolerance if each absolute difference of their three channel values is less than or equal to the tolerance. For example, if you run:

```
$ ./img_cmp checkerboard1.ppm checkerboard2.ppm 5
```

it should tell you how many pair of pixels in `checkerboard1.ppm` and `checkerboard2.ppm` have an absolute difference of more than 5 (i.e. intensity difference).

After you have checked the `checkerboard.c` , `ppm_io.c` , and `ppm_io.h` files, as an initial test to be sure you're on the right track, try to read in a PPM file and write it out unchanged. Use the `img_cmp` program to verify the two files are the same. Once this works well, begin successively working through the other commands as listed.

Development Plan

For this project, the development plan is largely in your hands. You are required to create (or modify the provided one) a Makefile and submit it for this project, and the executable it generates should be named `project` . You should also rigorously test your code, but you are not required to hand in specific tests with your submission. Beyond that, it is largely up to you how to structure your program, and what order to implement features in, but you **must take a modular approach**.

We recommend that you break your code into several files, and have as little code as possible in `main()`. Here is a suggested (but not required) breakdown of features into files:

- `Makefile` - you **must** include a `Makefile` that can build your program; it's how the graders will compile

your submission. You are required to build a target whose name is `project`.

- `project.c` - the main program. The `main()` function should be extremely simple; it might literally call a single function, then return 0. A `project.c` with an empty `main()` is provided in the starter code.
- `ppm_io.c` - contains implementations of functions for reading, writing, creating, destroying, copying, etc. images (using the PPM format). In the starter code, reading and writing PPM files are provided for you. You are recommended to add other image *memory*-related processing (destroying, copying, etc.) in this file.
- `ppm_io.h` - the header file for PPM I/O stuff (struct and function declarations).
- `img_processing.c` - contains implementations of all image processing operations.
- `img_processing.h` - the header file for the image processing stuff (struct and function declarations).

INFO

The list above is not a list of ALL files that should be included in your submission. It is only a listing of possible source code files.

We recommend that before you start coding, you make a **development plan**. This means that you sit down and plan out (on paper) how you will break the program down into modules (e.g. functions or groups of functions), what each module will do, and how they will interact.

TIP

Aim to write small, clean helper functions for better readability and easier testing, as well as greater reusability. Then, make a plan for what order you will implement the modules in. You will also want to test your modules; it's a good idea to use test-driven design, which means that you will design tests for your functions before you actually start trying to write the functions themselves.

For each module, it's important to think about precisely what it should do, and also how you can test it to be sure it's doing what you want. There are lots of ways of testing your code; for this project, a lot of your tests will likely involve the visual inspection of output images to see if they look the way they're supposed to. Still, having an idea of how you'll test each piece before you start writing it (and then testing/fixing it before you move on to the next one) will make your life a lot easier.

TIP

Some results images are provided in the public repo. If you like, you can use the `img_cmp` program to compare your results with them.

Scaffolding Folder

The scaffolding (i.e. starter code) folder for this project (available in the public repository) provides you with `ppm_io.c`, `ppm_io.h`, `checkerboard.c`, `project.c`, `img_cmp`, and a `Makefile` for the project. It also contains some testing PPM files in a folder named `data` and some expected results in a subfolder named

results , which is in the PPM format.

INFO

Note that the results displaying in this page is the png version of it. You should use the PPM ones for comparison.

TIP

If you want to work on your own images, you can use convert them on ugrad using `convert` . e.g. to convert `selfie.jpg` to PPM, you could run:

```
$ convert selfie.jpg selfie.ppm
```

We encourage you to store the provided PPM images and all created images in a subfolder of your own repository named `data`, to keep your images separate from your source code files. You don't need to submit any PPM files to us; keeping them in a separate folder will help you avoid accidentally including them.

TIP

If you're using the `data` subfolder, we suggest you execute your code from within the `data` folder by typing `../project` , so you can refer to input filenames while the program is executing directly as `building.ppm` , rather than `data/building.ppm` , saving yourself a lot of typing while testing.

Error Reporting

The approach your program will take for error reporting is to have your `main()` method return a `0` value indicating success or a positive value indicating failure. Which positive value your program should return is indicated in the table below. If more than one error condition occurs, your program should return the error code listed earliest in the table below.

In addition to returning the specified value, your program should also output an informative error message to `stderr`. (The text of the error messages will not be specified; we'll leave the exact wording up to you.)

Return value	Error condition it signifies
0	No errors detected
1	Wrong usage (i.e. mandatory arguments are not provided)
2	Input file I/O error

3	Output file I/O error
4	The Input file cannot be read as a PPM file
5	Unsupported image processing operations
6	Incorrect number of arguments for the specified operation
7	Invalid arguments for the specified operation
8	Other errors not specified above

Implementation Details

This section contains detailed descriptions of the image processing operations and the like that will be necessary for this assignment. We use the following sample image for all the all examples/operations illustrated below.



The original kitten image

Grayscale

This is a fairly simple operation, but it does require a little math. Basically, for each pixel, you will calculate a single grayscale value based on the three color values (red, green and blue), and then assign that same value

to all three color channels (if all three color channels have the same value, you know the pixel will show up as some shade of gray). For this program, we will use the NTSC standard conversion formula:

$$\text{grayscale} = 0.3 * \text{red} + 0.59 * \text{green} + 0.11 * \text{blue}$$

If you look around online, you will discover that there are actually several different formulas that can be used, which result in slightly different results; please use the NTSC version for this assignment (see Wikipedia on [grayscale conversion](#)).

TIP

Once the value is calculated, you will need to cast it back to an `unsigned char` before you assign it to each channel.

If you apply the grayscale transform to the `kitten.ppm` image, the result should look like:



The grayscale kitten image

Binarize

To binarize an image, we use the thresholding. You will first convert the input image to a grayscale version. Then, you will calculate a single binary value by comparing the grayscale value with a `threshold` value. Therefore, this operation will take an additional input parameter as a `threshold`, which is expected to be an integer and in the range between 0 and 255 inclusively. In your program, you should check if there is exactly one parameter is provided for the binarize operation. Otherwise, you should report an error. You do not need to check if the input `threshold` is an integer or not, but you do need to check it is a valid number and is between 0 and 255. If not,

you should report an error. In case the input `threshold` is not an integer, you can handle it by casting it to an integer (i.e. truncate the floating point value).

The binary value is set to 0 if the grayscale value is smaller than the threshold. Otherwise, it is set to 255. For each pixel, assign the same `$binary%` value to the three color channels of your output image. For example, if you run the below command:

```
$ ./project kitten.ppm kitten-binarize-127.ppm binarize 127
```

The result should look like:

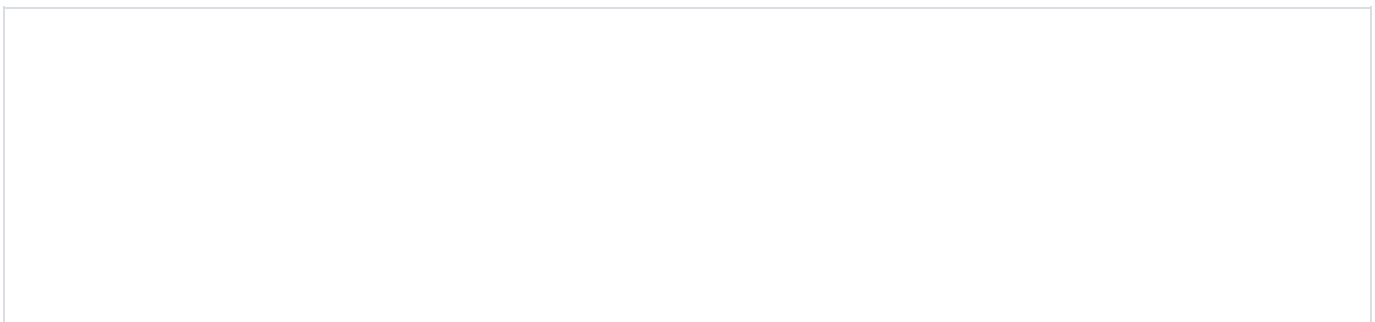


The binarized kitten image with threshold 127

If you run:

```
$ ./project kitten.ppm kitten-binarize-200.ppm binarize 200
```

The result should look like:





The binarized kitten image with threshold 200

Crop

Cropping an image is pretty simple; you just need to specify the two corners of the section you want to crop. That will mean 4 integer values: the column and then row of the upper-left corner, and the column and then row of the lower-right corner. By looking at the differences between those values, you can calculate the size of the new image; this will let you allocate the correct amount of space for the pixel array. Once you've done that, you can just use a loop to go through the pixels of the specified region in the original image, and copy each component of each pixel to the new image. You should check if it is exactly 4 additional arguments provided for the cropping operation, and check if the specified corners are senseless or not. You should report appropriate errors.

If you crop the `kitten.ppm` image from (top col=200, top row=200) to (bottom col=300, bottom row=300), the result should look like:



The kitten image cropped with 200 200 300 300

Transpose

Transposing an image is straightforward; you just need to allocate a new image with a flipped dimension in width and height of the input image. Then, you use a loop to go through the pixels to assign the new pixel value using a column-row flipped index, i.e. $\text{New}(x, y) = \text{Old}(y, x)$. If you transpose the `kitten.ppm` image, you should see the following result image:


[CS220 Syllabus](#)
[Course Staff](#)
[Course Material](#)
[CA Office Hours](#)
[Exercises](#)
[Homeworks](#)
[Projects](#)
[Exam Resources](#)


The transposed kitten image

Gradient

[Image gradient](#) is useful in many image processing operations, such as [Canny edge detector](#) and [Seam carving](#). Let $I(x, y)$ be our input image. You will compute it's gradient in x- and y-direction using the below finite difference formula:

$$\nabla_x I(x, y) = \frac{I(x+1, y) - I(x-1, y)}{2}$$

$$\nabla_y I(x, y) = \frac{I(x, y+1) - I(x, y-1)}{2}$$

Then, you will compute the magnitude of the gradient at each pixel by taking their absolute sum, i.e.

$$G(x, y) = |\nabla_x I(x, y)| + |\nabla_y I(x, y)|$$

Note that, you will only compute the gradient for each interior pixel (i.e. pixels that are not on the boundary). For those on the boundary, you can set its gradient magnitude to 0. For each pixel, assign the same gradient magnitude $G(x, y)$ to the three color channels of your output image. For example, if you run

```
$ ./project kitten.ppm kitten-gradient.ppm gradient
```

[Copy](#)

The result should look like:



The kitten gradient image

Seam Carving

Sometimes when we rescale an image, we don't want to change the aspect ratio. We can do it by cropping an image, but this requires a manual input and sometimes what we want to crop is something in the middle. [Seam carving](#) is a rescaling technique serves this purpose. You are going to implement a simplified version of seam carving.

You first determine the rescaled image size by multiplying the first scale factor to the input image column size and

the second scale factor to the input image row size (*round it down* if the new size is not an integer). If the rescaled column size or row size is less than 2 pixel, we cap it at 2 pixel. i.e. the minimal output image size should be 2x2.

INFO

We cap it to 2x2 because we need to use the gradient image, which has no values at the boundary.

You should always perform the seam carving by **columns first, then rows**. From now, we only consider in one direction, say columns (i.e. x- or horizontal direction). Given a scale and input image number of columns, and a computed the output image number of columns. The difference in columns is the number of seams, d , that you are going to remove from the input image.

You will remove seams from the input image iteratively for d times. At each iteration, you will only remove one seam. You will remove the one having the least **gradient energy**. To do that, at each iteration, you first compute the gradient image of the current seam carved image (say, if you are at the i th iteration, the current seam carved image should have i columns less than the input image).

You will create seams starting at row 0 and going down row by row. To start, you have each column at row 0 as a start pixel of a seam. i.e. if the current seam carved image has $c\#$ columns, you will have $c\#$ seams created. For each seam, you go down row by row by connecting it with the next row neighbor pixel which has the least gradient magnitude (which you can check using the gradient image), until you hit the second last row.

INFO

Each pixel should usually have three neighbors in the next row, except when you are at the boundary, where you only have two neighbors. Furthermore, since we don't have gradient magnitude at the boundary, when you look for the neighbor which has the least gradient magnitude, **you should skip checking the boundary pixels.**

Because we don't have gradient magnitude at the last row (boundary pixels), when you are at the second last row, you could not look for the next row neighbor which has the least gradient magnitude (their magnitudes are all 0). In this case, you will complete the seam by connecting to its immediate next row neighbor (i.e. the same column index). At this point, you should have created $c\#$ seams and you need to determine which one to be removed.

For each seam, you will compute the total gradient magnitude by summing each pixel's of the seam, which defines the gradient energy of the seam. Then, you will remove the seam that has the least gradient energy. Once you have determined which seam you are going to remove, you can allocate a new image with columns size minus one, then use a loop to copy the pixel values from the current seam carved image by skipping this seam. Set this new image as your current seam carved image and perform the next iteration until $c\#$ seams have been removed. After you have removed seams from columns, you will do the same to remove seams from rows.

TIP

To help you understand better, below is a brief pseudocode for removing seams (carving columns). Note that condition checking is ignored in this pseudocode.

Compute number of seams (d) to remove

Iterate from 1 to d. At each iteration, given an image (I) with c# columns,

1. compute the gradient image (G) of I
2. create c# seams (e.g. store in a 2D array). To create a seam,
 - A. start at the c'th col and 1'st row, where c is the c'th seam to create
 - B. connect the seam to the next row as follows:
 - a. Given the seam currently be at the i'th col and j'th row, compare the gradient image values of its next row three neighbors i.e. compare $G(i-1, j)$, $G(i, j)$, and $G(i+1, j)$
 - b. connect the seam to the pixel having the least $G(.,.)$ value i.e. connect to (m,n) where $G(m,n) = \min(G(i-1, j), G(i, j), G(i+1, j))$
 - c. repeat a-b. until reaching the second last row
 - C. close the seam by connecting the next row using the same column index. i.e. if currently the seam is at (m, n), connect it to (m, n+1)
3. for each seam, compute the "gradient energy" by summing up the $G(.,.)$ value at all pixels that the seam connects to / passes by
4. find the seam that has the least "gradient energy"
5. remove this seam (all its pixels) from I to produce an output image
6. use this output image as I for the next iteration

Return the final output image

To operate on the rows, you can **transpose** the image, operate on its columns, then **transpose** it back.

INFO

As you may imagine, the seam carving performs several iterations and each iteration contains several loops. Expect this operation to be significantly slower than the others.

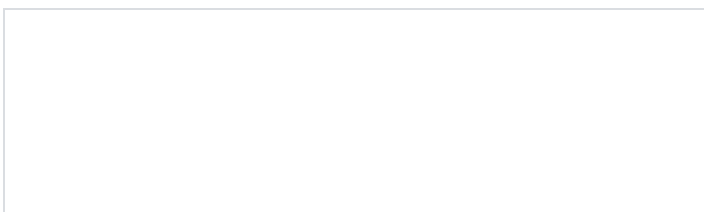
TIP

The above-given pseudocode is the suggested approach to seam carve an image, but it does not mean you have to follow it. As long as your implementation produces correct outputs, it is fine!

If you seam carve `kitten.ppm` by the following command:

```
$ ./project kitten.ppm kitten-seam_0.5_0.5.ppm seam 0.5 0.5
```

The result should look like:





The seam carved kitten image by 0.5 0.5

And if you run:

```
$ ./project kitten.ppm kitten-seam_0.2_0.9.ppm seam 0.2 0.9
```

You should see:



The seam carved kitten image by 0.2 0.9

Look how the kitten is preserved after rescaling using seam carving.

Submission

DANGER

Remember that programs which do not compile (with standard compiler flags on the ugrad machines) will

not receive credit. Additionally, points will be deducted for any compiler warnings. Points will also be deducted for any warnings, errors, or memory leaks reported by valgrind. All executables should be buildable using a `Makefile`, and should build and run cleanly.

Submit your project via Gradescope. Your submission should contain all source code and files necessary to compile your program (including a `Makefile`) as well as a `README` file (which includes both partner names and JHEDs) and a `git log` file from your midterm project repo. The `log` should indicate that all team members were contributing code and pushing their contributions to the repository. Your submission **should not** contain any compiled binaries (executables or object files), or any testing-related files (in particular, please do not submit any image files).

The requirements for your `git log` are the same as in previous assignments, except note that we expect all members of your team to be contributing commits to your shared midterm project repo.

Only one team member should submit the project on Gradescope. The same team member should submit all versions of the project in his/her account. **Be sure that source file headers include the names and JHEDs of all team members**, so that each student gets credit for this work.

 [Edit this page](#)

General Requirements

Program Description

Development Plan

Scaffolding Folder

Error Reporting

Implementation Details

Grayscale

Binarize

Crop

Transpose

Gradient

[Seam Carving](#)

Submission