Image Processing

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

This lab will also use the `pillow` library, which we'll use for loading and saving images. See this page for instructions for installing pillow (note that, depending on your setup, you may need to run `pip3` instead of `pip`). If you have any trouble installing, just ask, and we'll be happy to help you get set up.

The following file contains code and other resources as a starting point for this lab: `image_processing.zip`

Your changes for this lab should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file, nor should you use the `pillow` module for anything other than loading and saving images (which are already implemented for you).

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- correctly answering the questions throughout this page (2 points),
- passing the tests in `test.py` (2 points), and
- a brief "checkoff" conversation with a staff member about your code (1 point).

All of the questions on this page, including your code submission, are due at 5:00pm on Friday, 16 February. Checkoffs are due at 10:00pm on Wednesday, 21 February. **It is a good idea to submit your lab early and often so that if something goes wrong near the deadline, we have a record of the work you completed before then.** We recommend submitting your lab to the server after finishing each substantial portion of the lab.

2) Introduction

In this lab, you will build a few tools for manipulating digital images, akin to those found in image-manipulation toolkits like Photoshop and GIMP. Interestingly, many classic image filters are implemented using the same ideas we'll develop over the course of this lab.

For this week, we will focus on greyscale images only.

2.1) Digital Image Representation and Color Encoding

Before we can get to *manipulating* images, we first need a way to *represent* images in Python. While digital images can be represented in myriad ways, the most common has endured the test of time: a rectangular mosaic of *pixels* -- colored dots, which together make up the image. An image, therefore, can be defined by specifying a *width*, a *height*, and an array of *pixels*, each of which is a color value. This representation emerged from the early days of analog television and has survived many technology changes. While individual file formats employ different encodings, compression, and other tricks, the pixel-array representation remains central to most digital images.

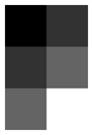
For this lab, we will simplify things a bit by focusing on greyscale images. Each pixel's brightness is encoded as a single integer in the range `[0,255]` (1 byte could contain 256 different values), `0` being the deepest black, and `255` being the brightest white we can represent. The full range is shown below:

For this lab, we'll represent an image using a Python dictionary with three keys:

- `"width"`: the width of the image (in pixels),

- `"height"`: the height of the image (in pixels), and
- `"pixels"`: a Python list of pixel brightnesses stored in row-major order (listing the top row left-to-right, then the next row, and so on)

For example, consider this 3×2 image (3 rows, 2 columns) below (enlarged here for clarity):



This image would be encoded as the following instance:

```
i = {
    "height": 3,
    "width": 2,
    "pixels": [0, 50, 50, 100, 100, 255],
}
```

2.2) Loading, Saving, and Displaying Images

We have provided two helper functions in `lab.py` which may be helpful for debugging: `load_greyscale_image` and `save_greyscale_image`. Each of these functions is explained via a docstring.

You do not need to dig deeply into the actual code in those functions, but it is worth taking a look at those docstrings and trying to use the functions to:

- load an image, and
- save it under a different filename.

You can then use an image viewer on your computer to open the new image to make sure it was saved correctly.

There are several example images in the `test_images` directory inside the lab's code distribution, or you are welcome to use images of your own to test, as well.

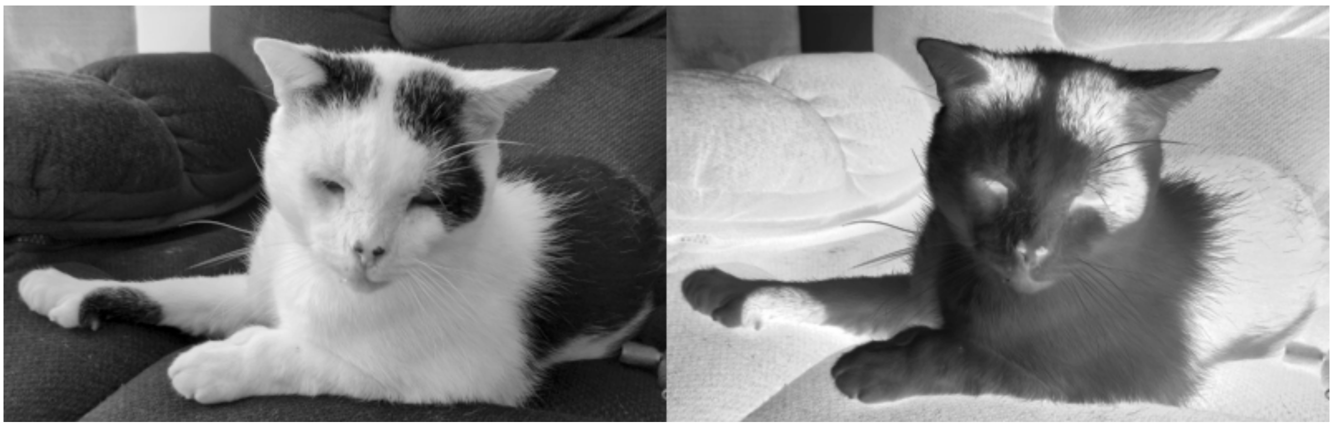
As you implement the various filters described below, these functions can provide a way to visualize your output, which can help with debugging, and they also make it easy to show off your cool results to friends and family.

Note that you can add code for loading, manipulating, and saving images under the `if __name__ == '__main__':` block in `lab.py`, which will be executed when you run `lab.py` directly (but not when it is imported by the test suite). As such, it is a good idea to get into the habit of writing code containing test cases of your own design within that block, rather than in the main body of the `lab.py` file.

3) Image Filtering via Per-Pixel Transformations

As our first task in manipulating images, we will look at an *inversion* filter, which reflects pixels about the middle grey value (`0` black becomes `255` white, `1` becomes `254`, and so on). For example, consider the following three images, each showing an adorable cat. In each, the left side is the original image, while the right is an inverted version.

Notice how, in each example, the darkest spots in the input become the brightest spots in the output and *vice versa*, while grey spots in the input result in similar (but not necessarily identical) grey spots in the output.



Most of the implementation of the inversion filter has been completed for you (it is invoked by calling the function called `inverted` on an image), but some pieces have not been implemented correctly. Your task in this part of the lab is to fix the implementation of the inversion filter.

Before you do that, however, let's add a small test case to `test.py`. We'll start by figuring out (by hand) what output we should expect in response to a given input and writing a corresponding test case.

Eventually, when we have working code for our inversion filter, this test case should pass!

Let's start with a 1×4 image that is defined with the following parameters:

- height: `1`
- width: `4`
- pixels: `[30, 87, 131, 223]`

If we were to run this image through a working inversion filter, what would the expected output be? Compute this result by hand. Then, in the box below, enter a Python list representing the expected value associated with the `pixels` key in the resulting image:

3.1) Adding a Test Case

While we could just add some code using `print` statements to check that things are working, we'll also use this as an opportunity to familiarize ourselves a bit with the testing framework we are using in 6.101.

Let's try adding this test case to the lab's regular tests so that it is run when we execute `test.py`. If you open `test.py` in a text editor, you will see that it is a Python file that makes use of the Python package `pytest` for unit testing.

Each function that has a name starting with `test` in `test.py` represents a particular test case.

Running `test.py` through `pytest` will cause Python to run and report on *all* of the tests in the file.

However, you can make Python run only a subset of the tests by running, for example, the following command from a terminal[1] (not including the dollar sign):

```
$ pytest test.py -k load
```

This will run any test with "load" in its name (in this case, the long test defined in the `test_load` function). If you run this command, you should get a brief report indicating that the lone test case passed. By modifying that last argument (`load` in the example above), you can run different subsets of tests.

You can add a test case by adding a new function to the `test.py` file. Importantly, for it to be recognized as a test case, its name must begin with the word `test`[2].

In the skeleton you were given, there is a trivial test case called `test_inverted_2`. Modify this function so that it implements the test from above (inverting the small 1×4 image). Within that test case, you can define the expected result as an instance of the hard-coded dictionary, and you can compare it against the result from calling the `inverted` function on the original image. To compare two images, you may use our `compare_images` function (you can look at some of the other test cases to get a sense of how to use it).

Once you have implemented your function, paste it into the box below (you do not need to include the other functions/code from `test.py`). When you submit the box below, we will check your test case against several implementations of `inverted`, some of which work and some of which don't.

```python
def test_inverted_2():
    assert False

```

Now that you've tested this function, copy it back into your `test.py` so that it is run every time we run `test.py`. Even once we've implemented this function, we should also expect the test case to fail when run on your own code (after all, we haven't finished fixing the `inverted` filter yet!), but we can expect it to pass once all the bugs in the inversion filter have been fixed. In that way, it can serve as a useful means of guiding our bug-finding process (i.e., as long as it isn't passing, there are still more bugs to find!).

Throughout the lab, you are welcome to (and may find it useful to) add your own test cases for other parts of the code as you are debugging `lab.py` and any extensions or utility functions you write. Any function that starts with the word `test_` will be interpreted by `pytest` as a test case.

3.2) `lambda` and Higher-Order Functions

As you read through the provided code, you will encounter some features of Python that you may not be familiar with. One such feature is the `lambda` keyword.

In Python, `lambda` is a convenient shorthand for defining small, nameless functions.[3]

For instance, in the provided code you will see:

```python
lambda color: 256-color
```

This expression creates a function object that takes a single argument (`color`) and returns the value `256-color`. It is worth noting that we could have instead used `def` to create a similar function object, using code like the following:

```python
def subtract_from_256(color):
    return 256-color
```

(the main difference being that `def` will both create a function object *and* bind that object to a name, whereas `lambda` will only create a function object)

If we had defined our function this way (with `def`), we could still provide that function as an argument to `apply_per_pixel`, but we would have to refer to the function by name: `apply_per_pixel(image, subtract_from_256)`

3.3) Debugging

Now, work through the process of finding and fixing the errors in the code for the inversion filter. Happy debugging!

When you are done and your code passes the `test_inverted_1` and `test_inverted_2` test cases, run your inversion filter on the `test_images/bluegill.png` image, save the result as a PNG image, and upload it below (choose the appropriate file and click "Submit"). If your image is correct, you will see a green check mark; if not, you will see a red X.

Inverted `bluegill.png`:
Select File `No file selected`

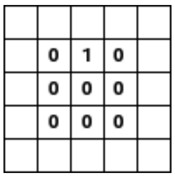
4) Image Filtering via Correlation

Next, we'll explore some slightly more advanced image-processing techniques involving an operation called *correlation*.

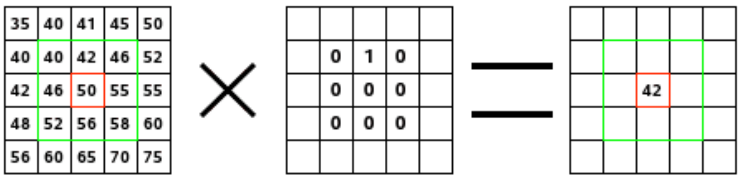
Given an input image I and a kernel k, applying k to I yields a new image O (perhaps with non-integer, out-of-range pixels), equal in height and width to I, the pixels of which are calculated according to the rules described by k.

The process of applying a kernel k to an image I is performed as a *correlation*: the brightness of the pixel at position (r, c) in the output image, which we'll denote as $O_{r,c}$ (with $O_{0,0}$ being the upper-left corner), is expressed as a linear combination of the brightnesses of the pixels around position (r, c) in the input image, where the weights are given by the kernel k.

As an example, let's start by considering a 3×3 kernel:



When we apply this kernel to an image I, the brightness of each output pixel $O_{r,c}$ is a linear combination of the brightnesses of the 9 pixels nearest to (r, c) in I, where each input pixel's value is multiplied by the associated value in the kernel:



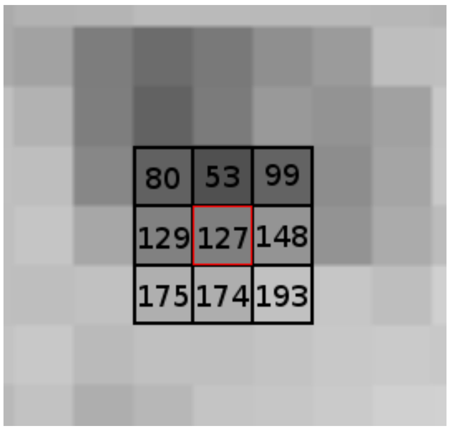
In particular, for a 3×3 kernel k, we have:

$$\begin{aligned} O_{r,c} = & I_{r-1,c-1} \times k_{0,0} + I_{r-1,c} \times k_{0,1} + I_{r-1,c+1} \times k_{0,2} + \\ & I_{r,c-1} \times k_{1,0} + I_{r,c} \times k_{1,1} + I_{r,c+1} \times k_{1,2} + \\ & I_{r+1,c-1} \times k_{2,0} + I_{r+1,c} \times k_{2,1} + I_{r+1,c+1} \times k_{2,2} \end{aligned}$$

Consider one step of correlating an image with the following kernel:

```
  0.00  -0.07   0.00
 -0.45   1.20  -0.25
  0.00  -0.12   0.00
```

Here is a portion of a sample image, with the specific luminosities for some pixels given:



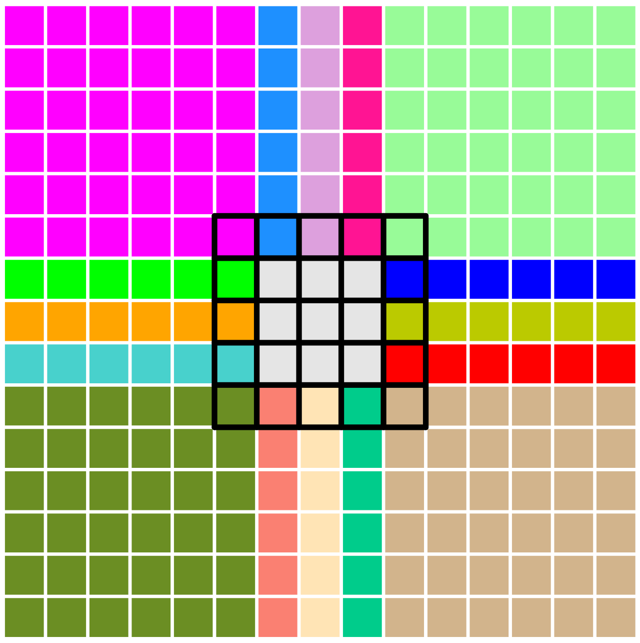
What will be the value of the pixel in the output image at the location indicated by the red highlight? Enter a single number in the box below. Note that, although our input brightnesses were all integers in the range $[0, 255]$, this value will be a decimal number.

4.1) Edge Effects

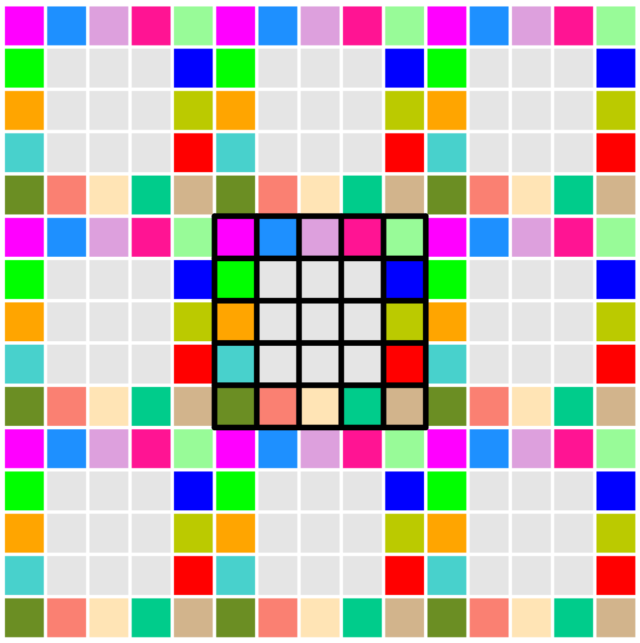
When computing the pixels at the perimeter of O, fewer than 9 input pixels are available. For a specific example, consider the top left pixel at position $(0, 0)$. In this case, all of the pixels to the top and left of $(0, 0)$ are out-of-bounds. We have several options for dealing with these edge effects:

- One option is to treat every out-of-bounds pixel as having a value of 0.
- Another is to *extend* the input image beyond its boundaries, explained in the next paragraph.
- Yet another is to *wrap* the input image at its edges, explained in the paragraph after that.

If we want to consider these out-of-bounds pixels in terms of an *extended* version of the input image, values to the left of the image should be considered to have the values from column 0, values to the top of the image should be considered to have the values from row 0, etc., as illustrated in the following diagram, where the bolded pixels in the center represent the original image, and the pixels outside that region represent the logical values in the "extended" version of the image (and note that this process continues infinitely in all directions, even though a finite number of pixels are shown below):



If we want to *wrap* the input image at its edges, that means trying to get values beyond the left edge of the image should return values from the right edge, but from the same row. Similarly, trying to get values beyond the top edge of the image should return values from the bottom edge, but from the same column. We can think of it as the image being 'tiled' in all four directions:



The thick black square above indicates the original image, and the copies are an illustration of the wrapping behavior (but they are not part of the actual image). Note that this only shows one copy of the image in each direction, but we would like the wrapping to extend infinitely in all directions.

To accomplish the goal of switching between these different edge behaviors, you may wish to implement an alternative to `get_pixel` with an additional parameter for the out-of-bounds behavior, which expects one of the strings `"zero"`, `"wrap"`, or `"extend"`. The function would return pixel values from within the image normally but handle out-of-bounds pixels by returning appropriate values accordingly as discussed above rather than raising an exception. If you do this, other pieces of your code can make use of that function and not have to worry themselves about whether pixels are in-bounds or not.

4.2) Correlation

Throughout the remainder of this lab, we'll implement a few different filters, each of which involves computing at least one correlation. As such, we will want to have a nice way to compute correlations in a general sense (for an arbitrary image and an arbitrary kernel).

Note, though, that the output of a correlation need **not** be a legal 6.101 image (pixels may be outside the `[0,255]` range or may be floats [meaning they are not whole numbers]). Since we want our filters all to output valid images, the final step in every one of our image-processing functions will be to *clip* negative pixel values to 0 and values greater than 255 to 255, and to ensure that all values in the image are integers.

Because these two things are common operations, we're going to recommend writing a "helper" function for each (so that our filters can simply invoke those functions rather than reimplementing the underlying behaviors multiple times).

We have provided skeletons for these two helper functions (which we've called `correlate` and `round_and_clip_image`, respectively) in `lab.py`. For now, read through the docstrings associated with these functions.

Note that we have not explicitly told you how to represent the kernel used in correlation. You are welcome to use any representation you like for kernels, but you should document that change by modifying the docstring of the `correlate` function, and you should be prepared to discuss your choice of representation during the checkoff. You can assume that kernels will always be square and that every kernel will have an odd number of rows and columns.

Now that we have a sense of a reasonable structure for this code, it's time to implement these two functions!

To help with debugging, you may wish to write a few test cases comprised of correlating the 11-by-11 `test_images/centered_pixel.png` or another simple image with a few different kernels. You can use the kernels in the next section to help test that your code produces the expected results.

(You may also find it helpful to first implement correlation for 3x3 kernels only and then generalize to account for larger kernels.)

4.3) Example Kernels

Many different interesting operations can be expressed as image kernels (some examples can be seen below), and many scientific programs also use this pattern, so feel free to experiment.

4.3.1) Identity

```
0 0 0
0 1 0
0 0 0
```

The above kernel represents an *identity* transformation: applying it to an image yields the input image, unchanged.

4.3.2) Translation

```
0 0 0 0 0
0 0 0 0 0
1 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

The above kernel shifts the input image two pixels to the *right*, discards the rightmost two columns of pixels, and duplicates the leftmost column twice (when using the "extend" out-of-bounds behavior).

4.3.3) Average

```
0.0 0.2 0.0
0.2 0.2 0.2
0.0 0.2 0.0
```

The above kernel results in an output image, each pixel of which is the average of the 5 nearest pixels of the input.

4.4) Check Your Results

When you have implemented your code and are confident that it is working, try running it on `test_images/pigbird.png` with the following 13×13 kernel (which is all zeros except for a single value):

```
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
```

Run it once with each edge behavior (`'zero'`, `'extend'`, and `'wrap'`) and save the results as PNG images. Take a look at those images. Do they match your expectations? How do the different edge-effect behaviors manifest in the output?

Result with `'zero'` edge behavior:
Select File No file selected

Result with `'extend'` edge behavior:
Select File No file selected

Result with `'wrap'` edge behavior:
Select File No file selected

5) Blurring and Sharpening

5.1) Blurring

For this part of the lab, we will implement a box blur, which can be implemented via correlation using the 'extend' behavior for out-of-bounds pixels[4]. For a box blur, the kernel is an $n \times n$ square of identical values that sum to 1. Because you may be asked to experiment with different values of n, you may wish to define a function that takes a single argument `n` and returns an `n`-by-`n` box-blur kernel.

Notice that we have provided an outline for a function called `blurred` in the lab distribution. Read the docstring and the outline for `blurred` to get a sense for how it should behave.

Before implementing it, create two additional test cases by filling in the definitions of `test_blurred_black_image` and `test_blurred_centered_pixel` with the following tests:

- Box blurs of a 6×5 image consisting entirely of black pixels, with two different kernel sizes. The output is trivially identical to the input.
- Box blurs for the `centered_pixel.png` image with two different kernel sizes. You should be able to compute the output manually.

These test cases can serve as a useful check as we implement the box blur. **Note** that you will be expected to demonstrate and discuss these test cases during a checkoff.

Finally, implement the box blur by filling in the body of the `blurred` function.

5.1.1) Check Your Results

When you are done and your code passes all the blur-related tests (including the ones you just created), run your blur filter on the `test_images/cat.png` image with a box-blur kernel of size 13, save the result as a PNG image, and upload it below to be checked for correctness:

Blurred `cat.png`:
Select File No file selected

Before moving on, though, let's try running this same filter with the other correlation edge effects. Generate an image using the `'zero'` boundary behavior for the correlation in the blur filter, and then generate another using the `'wrap'` behavior. Look at those images; why do they look the way they do? When you're ready, upload them below.

Blurred `cat.png` using `'zero'`:
Select File No file selected

Blurred `cat.png` using `'wrap'`:
Select File No file selected

Check Yourself:

Before moving on, make sure your `blurred` function has been set back to using the `'extend'` edge behavior, which is what the tests (and the other filters we'll implement in this lab) will expect.

5.2) Sharpening

Next, we'll implement the opposite operation, known as a *sharpen* filter. The "sharpen" operation often goes by another name which is more suggestive of what it means: it is often called an *unsharp mask* because it results from subtracting an "unsharp" (blurred) version of the image from a scaled version of the original image.

More specifically, if we have an image (I) and a blurred version of that same image (B), the value of the sharpened image S at a particular location is:

$$S_{r,c} = 2I_{r,c} - B_{r,c}$$

One way we could implement this operation is by computing a blurred version of the image, and then, for each pixel, computing the value given by the equation above.

While you are not required to do so, it is actually possible to perform this operation with a single correlation (with an appropriately chosen kernel and again using the `'extend'` behavior for out-of-bounds pixels).

Check Yourself:

If we want to use a blurred version B that was made with a 3×3 blur kernel, what kernel k could we use to compute the entire sharpened image with a single correlation?

Implement the *unsharp mask* as a function `sharpened(image, n)`, where `image` is an image and `n` denotes the size of the blur kernel that should be used to generate the blurred copy of the image. You are welcome to implement this as a single correlation or using an explicit subtraction, though if you use an explicit subtraction, make sure that you do not do any rounding until the end (the intermediate blurred version should not be rounded or clipped in any way).

Note that after computing the above, we'll still need to make sure that `sharpened` ultimately returns a valid 6.101 image, which you can do by making use of your helper function from earlier in the lab.

Unlike the functions above, we have not provided a skeleton for this function inside of `lab.py`; you will need to implement it yourself. And make sure to include an informative docstring for your new function!

5.2.1) Check Your Results

When you are done and your code passes the tests related to sharpening, run your sharpen filter on the `test_images/python.png` image with a kernel of size 11, save the result as a PNG image, and upload it below to be checked:

Sharpened `python.png`:
Select File No file selected

6) Edge Detection

Our last task for greyscale images will be to implement a really neat filter called a Sobel operator, which is useful for detecting edges in images.

This edge detector is a bit more complicated than the filters above because it involves *two* correlations[5]. In particular, it involves kernels K_1 and K_2, which are shown below:

K_1:

```
-1 -2 -1
 0  0  0
 1  2  1
```

K_2:

```
-1 0 1
-2 0 2
-1 0 1
```

After computing O_1 and O_2 by correlating the input with K_1 and K_2, respectively (using the 'extend' behavior for each correlation), each pixel of the output O is the square root of the sum of squares of corresponding pixels in O_1 and O_2:

$$O_{r,c} = \text{round}\left(\sqrt{O1_{r,c}^2 + O2_{r,c}^2}\right)$$

As always, take care to ensure that the final image is made up of integer pixels in range `[0,255]`. But only clip the output *after* combining O_1 and O_2. If you clip the intermediate results, the combining calculation will be incorrect.

Check Yourself:

What does each of the kernels K_1 and K_2 do, on its own? Try running, saving, and viewing the results of those intermediate correlations to get a sense of what is happening here.

Implement the edge detector as a function `edges(image)`, which takes an image as input and returns a new image resulting from the above operations (where the edges should be emphasized).

Also, create a new test case: edge detection on the `centered_pixel.png` image. The correct result is a white ring around the center pixel that is 1 pixel wide.

As with `sharpened`, you should add this code to `lab.py` yourself (and make sure to include an informative docstring); there is no skeleton provided.

Note also that `math` has been imported for you, and you are welcome to use the `sqrt` function from it (though you can also compute square roots by raising numbers to the $1/2$ power if you want).

6.1) Check Your Results

When you are done and your code passes the edge-detection tests (including the one you just wrote), run your edge detector on the `test_images/construct.png` image, save the result as a PNG image, and upload it below:

Edges of `construct.png`:
Select File No file selected

7) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script. If you haven't already installed it, see the instructions on this page.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a image_processing /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.

8) Checkoff

Once you are finished with the code, you will need to come (in person) to any open lab time and add yourself to the queue asking for a checkoff in order to receive credit for the lab. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions; see the notes on style for more information). In particular, be prepared to discuss:

- Your test case for inversion.
- Your implementation of correlation, including your choice of representation for kernels.
- Your test cases for blurring.
- Your implementation of the unsharp mask.
- Your implementation of edge detection (including `test_edges_centered_pixel` test case).

> *You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.*

9) What Comes Next?

There are a lot of interesting classes at MIT that explore ideas related to this lab. 6.3000 (Signal Processing) was already mentioned in Lab 0's writeup, but it is relevant here as well, as a way to get a different perspective and a deeper understanding of how filters like these behave. 6.8371 (Digital and Computational Photography) is another class that is worth considering, perhaps later in your career, where you get an opportunity to learn about (and implement) a number of other interesting image-processing applications.

Footnotes

[1] Note that you won't be able to run this command from within Python; rather, it should be run from a terminal. Our readings on using the command line should get you started. If you want to try it out and are having trouble, we're happy to help in open lab hours!

[2] `pytest` has many other features, such as grouping tests or creating test classes whose methods form groups of tests. Check out its documentation if you are interested in learning about these features.

[3] Lambda functions in Python correspond more broadly to the idea of anonymous functions in computer programming. Most modern programming languages provide ways of creating anonymous functions.

[4] Although you are welcome to try and see what happens with the other out-of-bounds behaviors!

[5] It is for this reason, and specifically since we want to compute these correlations *without rounding* before combining the results together, that we suggested separating `correlate` and `round_and_clip_image` rather than implementing a single function that performed both operations.