Lab 0: Image Processing

The questions below are due on Friday February 07, 2020; 04:00:00 PM.

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

A Python Error Occurred:

```
Error on line 6 of Python tag (line 7 of source):
    from PIL import Image

ModuleNotFoundError: No module named 'PIL'
```

Table of Contents

1) Preparation

This lab assumes you have Python 3.6 or later installed on your machine.

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: `lab0.zip`

Most of your changes 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).

You can also see and participate in online discussion about this lab in the "Lab 0" Category in the forum.

This lab is worth a total of 4 points. Your score for the lab is based on:

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

All questions on this page (including your code submission) are due at 4pm on Friday, 7 Feb. Checkoffs are due at 10pm on Wednesday, 12 Feb.

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.

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[1]. 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 grayscale 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 2×3 image (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_image` and `save_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.

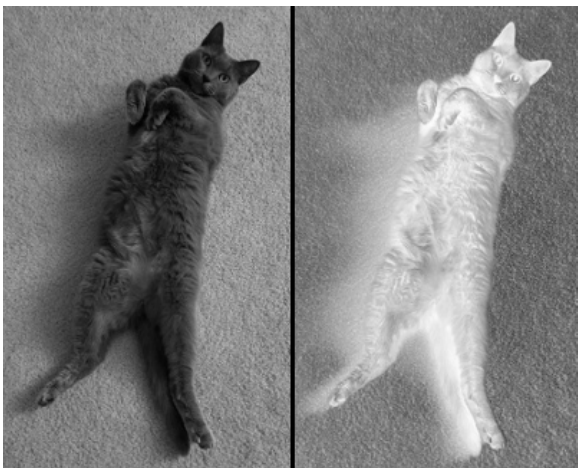
There are several example images in the `test_images` directory inside the lab's code distribution, or you are welcome to use images of your own to test, as well.

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 gray value (`0` black becomes `255` white and vice versa). For example, here is a photograph of Adam's cat, Stronger. On the left side is the original image, and on the right is an inverted version.



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 simple test case so that we can test whether our code is working.

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

- height: `1`
- width: `4`
- pixels: `[18, 69, 142, 216]`

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

This question is due on Friday February 07, 2020 at 04:00:00 PM.

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 6.009.

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 Python's `unittest` module for unit testing.

Each class in this file serves as a collection of test cases, and each method within a class represents a particular test case.

Running `test.py` 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[2] (not including the dollar sign):

```
$ python3 test.py TestImage
```

This will run all the tests under the `TestImage` class. If you run this command, you should get a brief report indicating that the lone test case passed.

If you want to be even more specific, you can tell `unittest` to run only a single test case, for example:

```
$ python3 test.py TestInverted.test_inverted_1
```

(Note that this test case should fail right now because the given implementation of the inversion filter has bugs!)

To add a test case, you can add a new method to one of the classes. Importantly, for it to be recognized as a test case, its name must begin with the word `test`[3].

In the skeleton you were given, there is a trivial test case called `test_inverted_2` in the `TestInverted` group. Modify this method so that it implements the test from above (inverting the small 4×1 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` method of the original image. To compare two images, you may use our `compare_images` function, which is implemented in the `Lab0Test` class (you can look at some of the other test cases to get a sense of how to use it).

For now, we should also expect this test case to fail, 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.

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.

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.

For instance, in the provided code you will see:

```
lambda c: 256-c
```

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

```
def subtract_from_256(c):
    return 256-c
```

(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 all the tests in the `TestInverted` test group (including, especially, the one you just created), 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

This question is due on Friday February 07, 2020 at 04:00:00 PM.

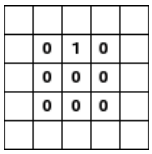
4) Image Filtering via Correlation

Next, we'll explore some slightly more advanced image processing techniques involving correlation of images with various kernels.

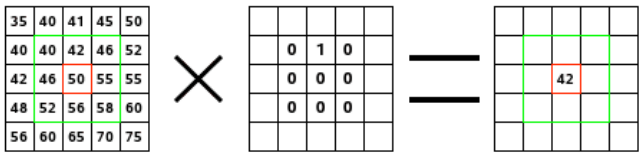
Given an input image I and a kernel k, applying k to I yields a new image O (perhaps with non-integer, out-of-range pixels), equal in height and width to I, the pixels of which are calculated according to the rules described by k.

The process of applying a kernel k to an image I is performed as a *correlation*: the brightness of the pixel at position (x, y) in the output image, which we'll denote as $O_{x,y}$ (with $O_{0,0}$ being the upper-left corner), is expressed as a linear combination of the brightnesses of the pixels around position (x, y) 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 $O_{x,y}$ is a linear combination of the brightnesses of the 9 pixels nearest to (x, y) in I, where each input pixel's value is multiplied by the associated value in the kernel:



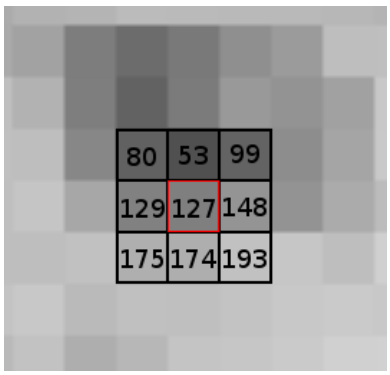
In particular, for a 3×3 kernel k, we have:

Could not parse math: `\begin{align} O_{x,y} = & I_{x-1, y-1} \times k_{0,0} + I_{x,y-1} \times k_{1,0} + I_{x+1,y-1} \times k_{2, 0} +\\ & I_{x-1, y} \times k_{0,1} + I_{x,y} \times k_{1,1} + I_{x+1,y} \times k_{2, 1} +\\ & I_{x-1, y+1} \times k_{0,2} + I_{x,y+1} \times k_{1,2} + I_{x+1,y+1} \times k_{2, 2} \end{align}`

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.

This question is due on Friday February 07, 2020 at 04:00:00 PM.

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. One option we have for dealing with these edge effects is to treat every out-of-bounds pixel as having a value of 0. However, this can lead to unfortunate artifacts on the edges of our images.

When implementing correlation, we will instead 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 1, values to the top of the image should be considered to have the values from row 1, etc., as illustrated in the following diagram[4] (note, however, that the image should be extended in all four directions, not just to the upper-left):

To accomplish this, you may wish to implement an alternative to `get_pixel`, which returns pixel values from within the image normally, but which handles out-of-bounds pixels by returning appropriate values 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, all of which involve 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 6.009 image (pixels may be outside the `[0,255]` range or may be floats). 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 re-implementing 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 your checkoff. You can assume that kernels will always be square and that they 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 `test_images/centered_pixel.png` or another simple image with a few different kernels. You can use the kernels in subsection 4.3, for example, 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

There is a world of interesting operations that 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 two columns.

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 9×9 kernel:

```
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
```

Save the result as a PNG image and upload it below:

Result:

Select File No file selected

This question is due on Friday February 07, 2020 at 04:00:00 PM.

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. 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 under the `TestFilters` class by filling in the definitions of `TestFilters.test_blurred_black_image` and `TestFilters.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 5, save the result as a PNG image, and upload it below to be checked for correctness:

Blurred `cat.png`:

Select File No file selected

This question is due on Friday February 07, 2020 at 04:00:00 PM.

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:

$$S_{x,y} = 2I_{x,y} - B_{x,y}$$

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).

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(i, n)`, `i` 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 6.009 image, which you can do by making use of your helper function from earlier in the lab.

Note also that, unlike the functions above, we have not provided a skeleton for this function inside of `lab.py`; you will need to implement it yourself.

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 box blur 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

This question is due on Friday February 07, 2020 at 04:00:00 PM.

6) Edge Detection

Although we will continue working with images in next week's lab, our last task for this week 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 Kx and Ky, which are shown below:

Kx:

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

Ky:

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

After computing Ox and Oy by correlating the input with Kx and Ky respectively, each pixel of the output O is the square root of the sum of squares of corresponding pixels in Ox and Oy:

$$O_{x,y} = \text{round}\left(\sqrt{Ox_{x,y}^2 + Oy_{x,y}^2}\right)$$

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 Ox and Oy. If you clip the intermediate results, the combining calculation will be incorrect.

> **Check Yourself:**
>
> What does each of the above kernels, on its own, do? 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(i)`, 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; there is no skeleton provided.

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
>
> **This question is due on Friday February 07, 2020 at 04:00:00 PM.**

7) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` by clicking on "Choose File", then clicking the `Submit` button to send the file to the 6.009 server. The server will run the tests and report back the results below.

Because of this time delay, it is a good idea to implement some tests on your own machine so that you can locally verify that these behaviors are working correctly.

> Select File No file selected
>
> **This question is due on Friday February 07, 2020 at 04:00:00 PM.**

8) Checkoff

Once you are finished with the code, you will need to come to a tutorial, lab session, or office hour 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 and test cases 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). In particular, be prepared to discuss:

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

8.1) Grade

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

Footnotes

[1] It turns out that the representation of digital images and colors as data is a rich and interesting topic. Have you ever noticed, for example, that your photos of purple flowers are unable to capture the vibrant purple you see in real life? This is the reason why.

[2] Note that you won't be able to run this command from within Python; rather, it should be run from a terminal. We don't necessarily expect that you have had experience with using the terminal in the past. If you want to try it out and you are having trouble, we're happy to help in office hours!

[3] You can also create new test groups by creating a new class whose name begins with `Test`.

[4] This image, which is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License, was created by Michael Plotke, and was obtained via Wikimedia Commons.

[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.