Lab 1: Image Processing

Table of Contents

1) Preparation

This lab assumes you have Python 3.7 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: `lab1.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).

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` and on the server `test.py` (2 points), and
- a brief "checkoff" conversation with a staff member to share your code, including discussion of the bullets below and a review of your code's clarity/style (1 point).

Both the questions on this page and your code submission are due at 4pm on Friday, 13 Sept.

2) Introduction

Have you ever watched *CSI* or another procedural investigative fantasy show? If so, the words "Enhance this image!" must sound familiar indeed. Reality is not quite so generous, and images cannot simply be enhanced[1]. In the real world, we do have a lot of techniques to analyze images. Photoshop and GIMP are rich toolkits of image-manipulation algorithms that give a user great abilities to evaluate, manipulate, and transform any digital image.

In this lab, you will help our own *CSI* (Computer Science Investigation) lab build up their tools for manipulating images. By the end of the lab, you'll have written code to invert, blur, sharpen, and find the edges in grayscale images. We have also provided suggestions of other image manipulations that build upon this lab (including how to work with color images in subsection 9.3), if you would like to further expand your toolset. Many real-world 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

In order to help our *CSI* lab start *manipulating* images with code, we first need to learn how to *represent* images in code[2].

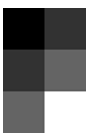
While digital images can be represented in a myriad of ways, the most common has endured the test of time: a rectangular mosaic of *pixels* -- colored dots, which together make up the image. An image, therefore, can be defined by specifying a *width*, a *height*, and an array of *pixels*, each of which 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.

Our *CSI* lab is just getting started, so they want to keep things simple and only focus on grayscale images. Each pixel 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 our lab, we'll represent images as instances of the provided `Image` class, which has three instance variables:

- `width`: the width of the image (in pixels),
- `height`: the height of the image (in pixels), and
- `pixels`: a Python list of pixels 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 = Image(2, 3, [0, 50, 50, 100, 100, 255])
```

2.2) Loading, Saving, and Displaying Images

We have provided three methods near the bottom of the `Image` class that may be helpful for debugging: `load`, `save`, and `show`. Each of these functions is explained via a docstring.

You do not need to dig deeply into the actual code in those methods, but take a look at those docstrings and try using the methods to:

- load an image,
- save it under a different filename, and
- display it using `show`.

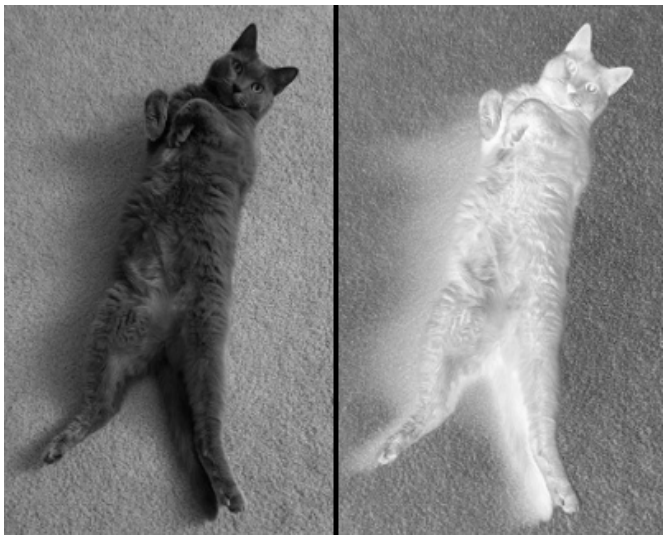
There are several example images in the `test_images` directory inside the lab's code distribution, or you are welcome to use images of your own to test, as well.

As you implement the various filters, 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).

3) Image Filtering via Per-Pixel Transformations

Our lab has decided that the first tool which they would like built is 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 Hartz's cat. On the left side is the original image, and on the right is an inverted version.



Unfortunately, the *CSI* programmers didn't wait for the MIT expert (you) and instead tried to complete the work themselves. Most of the implementation of the inversion filter has been completed for you (it is invoked by calling the method called `inverted`), but some pieces have not been implemented correctly. Your first task is to fix their buggy 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.

3.1) Adding a Test Case

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

- height: `1`
- width: `4`
- pixels: `[0, 70, 144, 218]`

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 `pixels` key in the resulting image:

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

Note that you are allowed to submit your answers to these concept questions as many times as you like, with no penalty!

Let's also add 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[3] (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`[4].

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 `Image` class, and you can compare it against the result from calling the `inverted` method of the original image.

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.

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 have seen previously. One such feature is the `lambda` function. In Python, `lambda` is a convenient shorthand for defining small nameless functions. Everything to the left of the colon is considered an argument to the function, and everything to the right is the result returned from the function.

For instance, in the provided code you will see:

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

This defines a function that takes in a value `c` and returns the value `256-c`.

In the provided code, this created function is then passed into `apply_per_pixel`, and used like any other function. When we have a function like `apply_per_pixel` that takes another function as an argument, this is called a "higher-order function."

Fun fact: It turns out that the concept of functions as inputs/outputs is so powerful that a Turing-complete language can be formed using only higher-order functions. If you're curious, you can read more here.

3.3) Debugging

Now it's time to work through the process of finding and fixing the errors in the provided code for the inversion filter. Happy debugging!

When you are done and your code passes all the tests in the `TestInverted` test group (specifically including 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

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

4) Image Filtering via Correlation

The folks at our *CSI* lab have been impressed! They have also learned their lesson and will not be attempting to code any more of their own image manipulation tools. *Whew!*

The lab has now requested a slightly more advanced feature, involving correlation of images with various kernels. Below is a partial transcript of their instructions:

4.1) Correlation Instructions

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 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 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:

	0	1	0	
	0	0	0	
	0	0	0	

When we apply this kernel to an image I, each output pixel $O_{x,y}$ is a linear combination 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:

35	40	41	45	50
40	40	42	46	52
42	46	50	55	55
48	52	56	58	60
56	60	65	70	75

$\times$

	0	1	0	
	0	0	0	
	0	0	0	

$=$

		42		

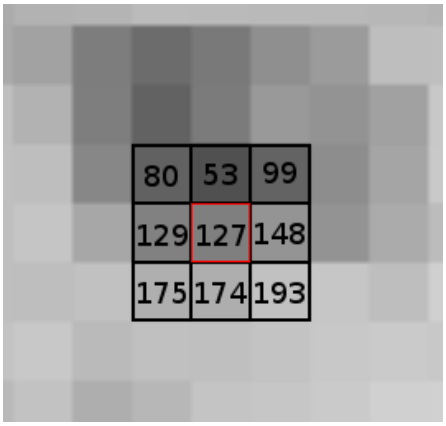
In particular, for a 3×3 kernel k, we have:

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

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 at this point we have not yet rounded or clipped the value.

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

4.2) 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[5] (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. Your correlation code, then, will not have to worry itself about whether pixels are in-bounds or not.

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.

Note 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).

So the final step in every image-processing function is to *clip* negative pixel values to 0 and values > 255 to 255, and to ensure that all values in the image are integers. *Hint:* you may wish to implement a function to perform this operation on an image.

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.

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) Correlation

If you have not done so already, you will want at this point to write a function to handle these correlations in a general sense (for an arbitrary image and an arbitrary kernel). Note that your function should not modify the input image during correlation, but rather create a new one. Regarding the kernel, it is up to you to choose how to represent it within your code.

(To help with debugging, you may wish to write a few test cases correlating `test_images/centered_pixel.png` or another simple image with a few kernels.)

You can use the above kernels to help test that your code produces the expected results. You may wish to first write your code specifically for 3×3 kernels, and then to generalize to kernels of arbitrary size (however, you can assume that kernels will always be square and that they will have an odd number of rows and columns).

When you have implemented your code, 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

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

5) Blurring and Sharpening

5.1) Blurring

CSI is once again impressed. They're excited to work with the new tool that you've created and want to include you in the first applications of it: box blur!

The box blur filter can be implemented by feeding an appropriate kernel to our newly created correlation function.

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.

Implement the method `blurred` in the `Image` class, which takes in a number `n` and returns a new *Image* with height and width equal to the dimensions of the input image that has been blurred with an appropriately sized box blur kernel. Take care to output *integer* brightness values (`round(value)`, for example) in range `[0, 255]`. That is, in many cases you will need to clip integer values explicitly to this range.

When you are done and your code passes all the blur-related tests, 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:

Blurred `cat.png`:

Select File No file selected

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

5.2) Sharpening

Now of course, no *CSI* lab would be complete without an "*Enhance!*" operation. Our lab is still content to start small and so requests that you do this by implementing a *sharpen* filter.

The "sharpen" operation often goes by another name that 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:

$$S_{x,y} = \text{round}\left(2I_{x,y} - B_{x,y}\right)$$

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.

However, it is actually possible to perform this operation with a single correlation (with an appropriate 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 the method `sharpened` in the `Image` class, where `n` denotes the size of the blur kernel that should be used to generate the blurred copy of the image. This method should return a new, sharpened 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).

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

Sharpened `python.png`:

Select File `No file selected`

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

6) Edge Detection

The lab has its first case! They can't tell us what, something about "national security", but they apparently need an edge-detecting tool, and fast!

For this part of the lab, we will implement a Sobel operator, which is useful for detecting edges in images.

This edge detector is more complex than a simple image kernel but is a combination of two image kernels Kx and Ky.

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 is the square root of the sum of squares of corresponding pixels in Ox and Oy:

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

Again, take care to ensure 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 `show` on the results of those intermediate correlations to get a sense of what is happening here.

Implement the edge detector as the method `edges` within the `Image` class. The method should return a new instance of `Image` resulting from the above operations.

When you are done and your code passes the edge detection tests, 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

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 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 "Select 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.

On the server, we will run 4 additional tests:

- we will test your inversion filter on `centered_pixel.png`, `pattern.png`, and `blob.png`
- we will test your blur filter with a 3×3 kernel on `centered_pixel.png`, `pattern.png`, and `blob.png`
- we will test your sharpen filter with a 3×3 kernel on `centered_pixel.png`, `pattern.png`, and `blob.png`
- we will test your edges filter on `centered_pixel.png` and `pattern.png`

You are welcome to submit your code as often as you like, but you will only be able to see the results of the server-only test cases once per 30 minutes.

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

Submit

You have submitted this assignment 0 times.

This question is due on Friday September 13, 2019 at 04:00:00 PM.

8) Checkoff

Once you are finished with the code, please come to a lab session or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.** Since the clarity of your code will be evaluated as part of the checkoff, you may wish to take some time to comment your code, use good variable names, avoid repetitive code (create helper methods), etc.

Be prepared to discuss:

- Your test case for inversion.
- Your implementation of correlation.
- Your implementation of the unsharp mask.

8.1) Grade

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

9) Optional Additional Behaviors

If you have finished early, there are a lot of interesting image filters to tackle, and we encourage you to explore (sorry, no additional points, though!). Some ideas you may wish to consider:

9.1) Additional Filters

- Implement a *normalization* filter that will scale pixel values linearly such that the darkest pixel in the output has a value of `0`, and the brightest has a value of `255`.
- Add capability for a Gaussian blur with arbitrary σ value.
- Create a more complex filter with position-dependent effects (such as ripple or vignette). If you've studied linear algebra (and remember it!), try rotating the image.

9.2) Content-Aware Rescaling

As an additional optional exercise for this lab, we can walk through implementing a version of content-aware resizing of images (often referred to as *retargeting*). The goal of this technique is to scale down an image while preserving the perceptually important parts (i.e., removing background but preserving subjects).

Two common approaches for resizing an image are *cropping* and *scaling*. However, in certain situations, both of these methods can lead to undesirable results. The animation below shows these various kinds of resizing in action on an image of two cats[6], which is gradually scaled down from 300 pixels wide to 150 pixels wide. Cropping is shown on the first row; naive scaling on the second row; and a form of retargeting on the third row.

The first two strategies have shortcomings: cropping causes one of the cats to be almost completely deleted, and scaling distorts both cats. By focusing on removing "low-energy" portions of the image, however, the seam-carving technique manages to keep both cats almost completely intact while removing mostly background.

Here is another example, on a picture of several trees. Notice again that the trees are preserved almost exactly despite decreasing the width of the image by 75 pixels.

These images were generated using a technique called *seam carving*. We'll start off by implementing a simpler variant of this algorithm and then describe a full implementation shortly thereafter.

9.2.1) Algorithm

While normal cropping of images works by removing columns of pixels from the left and right sides of the image indiscriminantly, this approach works by instead removing columns of pixels with low 'energy' (i.e., where not much is changing in the image).

Each time we want to decrease the horizontal size of the image by one pixel, we start by finding the column that has the minimum total energy and removing the pixels contained therein. To shrink the image further, we can apply this process repeatedly.

Here, we will define the 'energy' based on the edge-detection algorithm from earlier: the energy of any given pixel will be the associated value in the output from edge detection; and the energy of a path is the sum total of the energies of the pixels it contains.

This can be implemented by repeating the following three steps until the width of the image reaches the desired size:

1. **Compute energy map**. Because we are using our edge detector as our 'energy' function, this should simply involve calling the `edges` method of the image.
2. **Find the minimum energy column** by summing all the energies in each column and picking the minimum.
3. **Remove the computed path.** Finally, we need to remove the pixels in the chosen column from the image.

If we need to remove more columns, we can repeat all three steps of this process (including recomputing the energy map).

9.2.2) Implementation

Now it's time to implement the retargeting algorithm.

You are welcome to implement this using any name you prefer, and in any location in the file. It might make sense, also, to define some "helper" functions (perhaps corresponding to the steps above) rather than implementing everything in one function (e.g., you may wish to define a helper function to find the lowest-energy column in an image and another to remove a given column number from an image).

Note that this process should involve computing an energy map, removing a single column, then computing the energy map again and removing a second column.

To check yourself, run your code to do the following:

- Remove 75 columns from `tree.png`

- Remove 100 columns from `twocats.png`
- Remove 100 columns from `pigbird.png`

Save the resulting images and compare them to the originals. Note that these tests may take a long time (several minutes) to run.

Note that you may notice some artifacts in these images (particularly in the sky in `tree.png`). If you are interested in improving this, keep reading into our discussion on "seam carving".

9.2.3) Seam Carving

If you take a look at your result of using our original retargeting code to resize `tree.png` down by 75 pixels, you'll notice some artifacts in the skyline. It is possible to produce much-improved results by using a variation of this technique called *seam carving*.

You won't be awarded any additional points for implementing this technique, but it is **so cool**!

This technique is very similar to the one we used earlier, except that instead of removing the lowest-energy *column* in an image, we instead find and remove connected 'paths' of pixels from the top to the bottom of the image, with one pixel in each row. Each time we want to decrease the horizontal size of the image by one pixel, we start by finding the connected path from top to bottom that has the minimum total energy and removing the pixels contained therein. To shrink the image further, we can apply this process repeatedly.

As above, we will define the 'energy' based on the edge-detection algorithm from earlier: the energy of any given pixel will be the associated value in the output from edge detection; and the energy of a path is the sum total of the energies of the pixels it contains.

The goal of finding the minimum-energy path is accomplished by an algorithm which is outlined briefly below and which is described in detail on the Wikipedia page for seam carving.

The algorithm works by repeating the following three steps until the width of the image reaches the desired size:

1. **Compute energy map**. Because we are using our edge detector as our 'energy' function, this should simply involve calling the `edges` method of the image.

2. **Find the minimum-cost path from top to bottom.** We can start by creating a "cumulative energy map", where the value at each position is the total cost of the least-cost path from the top of the image to that pixel.

   For the top row, this will be equal to the energy map's top row. Subsequent rows can be computed using the following algorithm (in pseudocode):

   ```
   For each row of pixels in the image:
       For each pixel of a row:
           Find the minimum-cost path to reach that pixel by taking the
           minimum of the costs to reach each of the three possible
           preceding pixels and adding that to the cost of the current
           pixel. The updated cost of the current pixel now reflects the
           total cost of the minimum-cost path to reach that pixel.
   ```

   The minimum seam is then calculated by backtracing from the bottom to the top edge. First, the minimum value pixel in the bottom row of the accumulated cost map is located. This is the bottom pixel of the minimum seam. The seam is then traced back up to the top row of the accumulated cost map by following the neighboring pixels with the smallest cumulative costs.

3. **Remove the computed path.** Finally, we need to remove the pixels in the computed seam from the image.

Try running your code on `pigbird.png`, `twocats.png`, and `construct.png`, removing 100 seams from each.

9.3) Colors

Extend your code to work with color images. This can be accomplished by loading images from `pillow` in mode `"RGB"` instead of mode `"L"`. In this mode, `pixels` will be a list of `(r, g, b)` tuples.

Most of the filters described here can be implemented for color images by applying the black-and-white versions to the red, green, and blue channels separately and then recombining. The notable exception is that edge detection should be based on luminosity, which can be computed as:

$$L = \frac{299}{1000}R + \frac{587}{1000}G + \frac{114}{1000}B$$

Footnotes

[1] Though there is some work to do the next best thing, some of it done right here at MIT

[2] 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.

[3] 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!

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

[5] 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.

[6] This animation was created by Adam Hartz and is licensed under a CC BY SA 4.0 International License. The original image of two cats, which is licensed under a CC BY 2.0 License, comes from wellflat on Flickr.