Image Processing, Part 2

Table of Contents

1) Preparation

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

As with last week's lab, we will again use the `pillow` library for loading and saving images.

The following file contains code and other resources as a starting point for this lab:
`image_processing_2.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).

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).

Note in order to receive credit for the lab, you will need to come (in-person) to any open lab time after the lab deadline has passed and add yourself to the queue asking for a checkoff.

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.

> **Academic Integrity Policy**
>
> Please also review the academic integrity policies before continuing.

2) Reminders about Representations From Lab 1

In Lab 1, we introduced a representation for greyscale images, and we implemented some neat filters to operate on images of that form. In that lab, we limited our attention to greyscale images, where we represented brightness values as integers between 0 (the deepest black) and 255 (the brightest white).

Our filters were represented as Python functions that operated on images (i.e., functions that took images as input and produced related images as output).

Before we get into the core of this week's lab, answer the following question about the small piece of code below, which implements a small piece of functionality using our representations from last week.

```
def make_box(color):
    def create_image(h, w):
        return {
            "height": h,
            "width": w,
            "pixels": [color for _ in range(h * w)],
        }

    return create_image

maker = make_box(40)
im = maker(20, 30)
```

Which of the following are true statements about the code above?

- [ ] This code has a syntax error.
- [ ] This code does not have a syntax error, but Python will raise an exception when this code is run.
- [ ] This code will run without error.
- [ ] `make_box` is a function of a single argument.
- [ ] `make_box` is a function of two arguments.
- [ ] `make_box` is a dictionary.
- [ ] `maker` is a function of a single argument.
- [ ] `maker` is a function of two arguments.
- [ ] `maker` is a dictionary.
- [ ] `im` is a function of a single argument.
- [ ] `im` is a function of two arguments.
- [ ] `im` is a dictionary.

3) Introduction

In this lab, we'll continue building on the work from last week's lab, expanding our little image-processing library to include support for color images, as well as a few additional interesting filters and features.

At the end of the lab, you'll have an opportunity to implement one or more additional features of your choosing.

3.1) Representing Color

Although there are many ways we could choose to represent color images, we will use a tried-and-true representation, the RGB color model, which is used in a lot of common image formats. In this representation, rather than a single integer, we will represent the color of each pixel as a tuple of three integers, representing the amount of red, green, and blue in the pixel, respectively. By combining red, green, and blue, we can represent a wide range of colors.

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 values, each represented as a tuple of three integers, `(r, g, b)`, where `r`, `g`, and `b` are all in the range $[0, 255]$. As with last week's lab, these values are 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 (3 rows, 2 columns) image, enlarged here for clarity:

This image would be encoded as the following dictionary:

```
i = {
    "height": 3,
    "width": 2,
    "pixels": [(255, 0, 0), (39, 143, 230),
               (255, 191, 0), (0, 200, 0),
               (100, 100, 100), (179, 0, 199)],
}
```

3.2) Code Distribution

Like last week, we have provided helper functions for loading and saving images (this time, `load_color_image` and `save_color_image`, respectively), as well as some test images in the `test_images` directory (though, as before, you are welcome to use images of your own as well!).

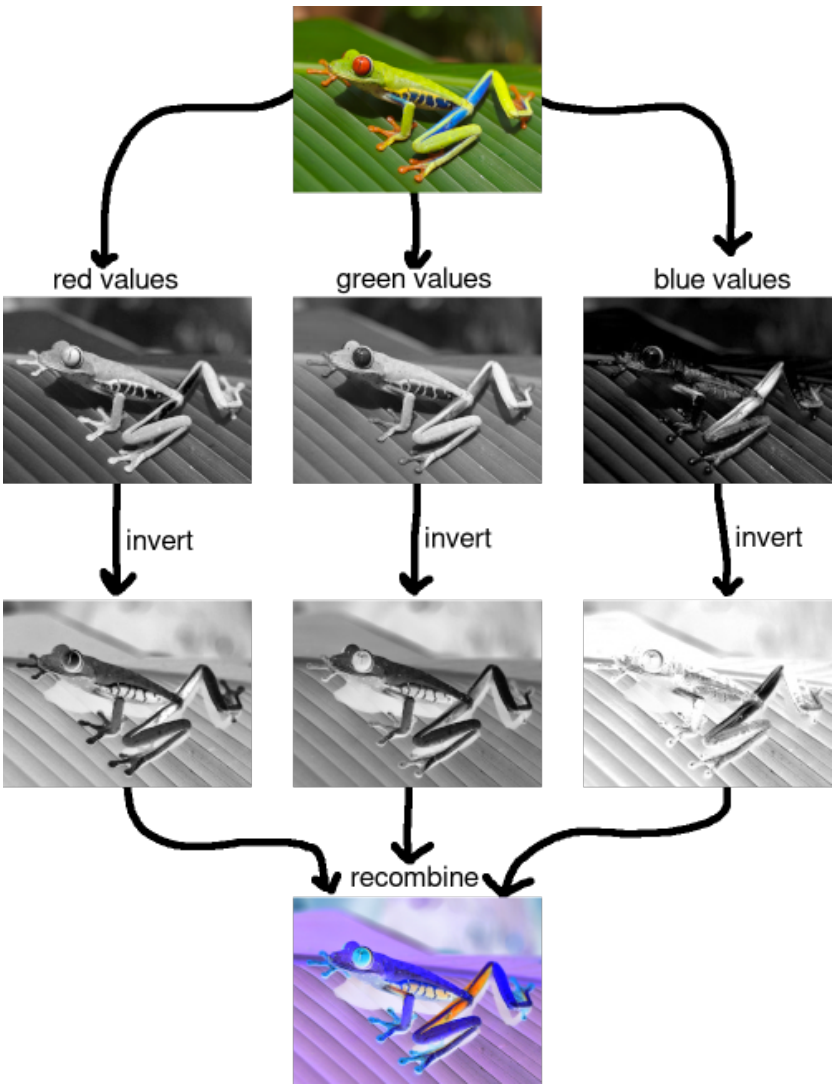
You should also copy over your functions from your work from last week's lab, as some of the things we'll implement here will build on the filters from Lab 1 directly. You do not need to copy the `load_greyscale_image` and `save_greyscale_image` functions, as we have provided additional helper functions for loading both color and greyscale images in this week's code distribution.

4) Filters on Color Images

In last week's lab, we used Python function objects to represent filters: we represented a filter as a function that took an image as input and produced a related image as output. We will continue this representation this week: a filter for color images is a Python function that takes a color image as its input and produces a related color image as its output.

While there are certainly other things we can do with color images, it turns out that we can think about implementing color versions of many of the filters from last week by separating our color image into three separate greyscale images (one for each channel: red, green, and blue), applying the same 'greyscale' filter from last week to each one, and then recombining the results together into a color image.

For example, we can think about a color version of our inversion filter as working in the way illustrated in the diagram below:



We can think about blurring or sharpening in similar ways: separate an image into its three color components, apply a filter to each, and recombine them together.

Given this common structure, it will be useful to have a common way to create these kinds of color image filters from their greyscale counterparts. As such, your first task for this lab is to implement the `color_filter_from_greyscale_filter` function in `lab.py`. This function should take as input *a Python function* representing a greyscale filter (i.e., a function that takes a greyscale image as input and produces a greyscale image as its output). Its return value should be a *new function* representing a color image filter (i.e., a function that takes a color image as input and produces a color image as its output).

This new filter should follow the pattern above: it should split the given color image into its three components, apply the greyscale filter to each, and recombine them into a new color image.

An example of its usage is given below:

```
# if we have a greyscale filter called inverted that inverts a greyscale
# image...
inverted_grey_frog = inverted(load_greyscale_image('grey_frog.png'))

# then the following will create a color version of that filter
color_inverted = color_filter_from_greyscale_filter(inverted)
```

```
# that can then be applied to color images to invert them (note that this
# should make a new color image, rather than mutating its input)
inverted_color_frog = color_inverted(load_color_image('color_frog.png'))
```

Implement the `color_filter_from_greyscale_filter` function in `lab.py`. You may find it helpful to define helper functions for splitting the color image into three separate greyscale images (one for each color component) and for recombining three greyscale images into a single new color image.

4.1) Check Your Results

Use this function to create a color version of your inversion filter from lab 1 and apply it to the `cat.png` image from the `test_images` directory. Save your result as a color PNG image and upload your image below to be checked:

Inverted `cat.png`:

Select File No file selected

4.2) Other Kinds of Filters

Note that the structure of the `color_filter_from_greyscale_filter` function assumes a particular form for our greyscale filters: specifically, it expects that the filters we pass to it as inputs are functions of *a single argument*. However, some of our filters from last week did not follow this pattern. In particular, our `blurred` and `sharpened` filters took both an image and an additional parameter. So how can we implement these functions in such a way that they can work together with `color_filter_from_greyscale_filter`?

One strategy is to define, for example, a function `make_blur_filter` that takes the parameter `kernel_size` and returns a blur filter (which takes a single image as argument). In this way, we can make a blur filter that is consistent with the form expected by `color_filter_from_greyscale_filter`, for example:

```
# last week, we could create a blurred greyscale image like follows:
blurry = blurred(load_greyscale_image('cat.png'), 3)

# we would like to make a function make_blur_filter that takes a single
# parameter and returns a filter.  the use of this function is
illustrated
# below:
blur_filter = make_blur_filter(3)
blurry2 = blur_filter(load_greyscale_image('cat.png'))

# note that make_blur_filter(3), for example, produces a filter of the
```

```
# appropriate form for use with color_filter_from_greyscale_filter, but
# that blurry and blurry2 are equivalent images.

# it is also not necessary to store the output of make_blur_filter in a
# variable before calling it.  for example, the following also produces
an
# equivalent image:
blurry3 = make_blur_filter(3)(load_greyscale_image('cat.png'))
```

Implement `make_blur_filter` and `make_sharpen_filter` in your `lab.py`. Each one of these functions should take a single argument `kernel_size` and should return a function appropriate for use with `color_filter_from_greyscale_filter`.

For the sake of avoiding repetitious code, you should try to make use of `blurred` and `sharpened` within your code, rather than reimplementing that behavior.

4.3) Check Your Results

Use these functions in conjunction with `color_filter_from_greyscale_filter` to produce the images described below and upload them to be checked:

Blurred `python.png`, with `kernel_size=9`:
Select File No file selected

Sharpened `sparrowchick.png`, with `kernel_size=7`:
Select File No file selected

5) Cascade of Filters

Next, we'll add support for chaining filters together in a cascade (so that the output of one filter is used as the input to another, and so on). You should implement a function called `filter_cascade` in your lab file to this end. This function should take a list of filters (Python functions) as input, and its output should be a function that, when called on an input image, produces the equivalent of applying each of the given filters, in turn, to the input image.

For example, if we have three filters `f1`, `f2`, and `f3`, then the following two images should be equivalent:

```
out1 = f3(f2(f1(image)))

c = filter_cascade([f1, f2, f3])
out2 = c(image)
```

Check Yourself:

Does it matter if `filters` contains color or greyscale filters? Can `filters` contain both color and greyscale filters?

Implement the `filter_cascade` function in your `lab.py`.

5.1) Check Your Results

When you are confident that your function is working as expected, apply the following filter to `test_images/frog.png`, save the result as a PNG image, and upload it below for checking:

```
filter1 = color_filter_from_greyscale_filter(edges)
filter2 = color_filter_from_greyscale_filter(make_blur_filter(5))
filt = filter_cascade([filter1, filter1, filter2, filter1])
```

`frog.png` filtered by the filter given above:
Select File No file selected

6) Seam Carving

You may wish to read all of the subsections in this part before implementing any code, so that you can make a more complete plan before diving in.

Next, we'll implement a really neat kind of effect, often referred to as "content-aware resizing" or "retargeting." The goal of this technique is to scale down an image while preserving the perceptually important parts (e.g., 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[1], which is gradually scaled down from 300 pixels wide to 150 pixels wide. Cropping is shown on the first row, naive scaling is shown on the second row, and a form of content-aware resizing called *seam carving* (which we'll implement in this section) is shown 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 (defined below) 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.

6.1) Outline and Description of Steps

The images above were generated using a technique called *seam carving*, which we'll implement in this section. And, although the images above were the result of applying seam carving to greyscale images, your code should work for color images.

The idea behind seam carving is that, when we remove a pixel from each row to decrease the size of an

image, instead of removing a straight vertical line, 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," removing the pixels contained therein. To shrink the image further, we can apply this process repeatedly.

We will define the "energy" based on the edge-detection algorithm from last week's lab, applied to a greyscale version of the image: 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 and removing the minimum-energy path is accomplished by an algorithm which is outlined below[2].

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

1. **Make a greyscale copy** of the current image, for using in computing the energy map. You can compute a color pixel's equivalent greyscale value v using the following formula, where r, g, and b represent the pixel's red, green, and blue values, respectively:

   $v = \text{round}\left(.299 \times r + .587 \times g + .114 \times b\right)$

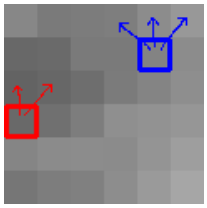
2. **Compute energy map**. Because we are using our edge detector as our "energy" function, this should simply involve calling the `edges` function on the greyscale image.

3. **Compute a "cumulative energy map"**, where the value at each position is the total energy of the lowest-energy path from the top of the image to that pixel.

   For the top row, this will be equal to the top row from the energy map in step 2. Subsequent rows can be computed using the following algorithm (in pseudocode):

   ```
    For each row of the "cumulative energy map":
        For each pixel in the row:
            Set this value in the "cumulative energy map" to be:
              the value of that location in the energy map, added to the
              minimum of the cumulative energies from the "adjacent"
   pixels in the row
              above
   ```

   For our purposes, we'll consider pixels in the row above to be "adjacent" to the pixel we're considering if they are either directly above, or one pixel to the right or to the left of, the pixel in question. For example, the pixel marked in blue below has three "adjacent" pixels in the row above (indicated by the arrows), and the pixel marked in red (on the edge of the image) has two "adjacent" pixels in the row above:



(After completing this process, the value at each pixel in this "cumulative energy map" will be the total energy of the lowest-energy path from the top of the image to that location.)

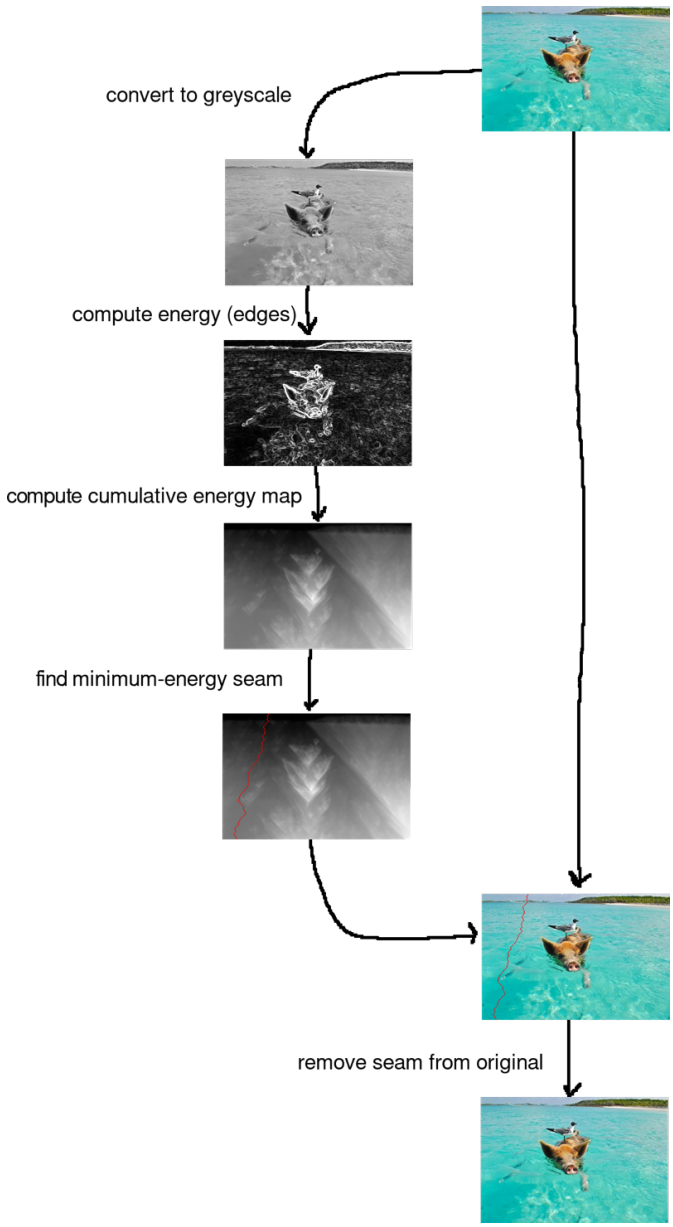
4. **Find the minimum-energy seam.** The minimum seam can then be found by backtracing from the bottom to the top of the cumulative energy map. First, the minimum value pixel in the bottom row of the cumulative energy map is located. This is the bottom pixel of the minimum seam. The seam is then traced back up to the top row of the cumulative energy map by following the adjacent pixels with the smallest cumulative energies.

   Ties should always be broken by preferring the left-most of the tied columns.

5. **Remove the computed path.** Finally, we need to remove the pixels in the computed seam from our original color image (and we should use this smaller result when we start back at the top of these steps). Note, though, that we would like for this process to work without modifying the original image.

6.1.1) Graphical Depiction

The process described above (for removing one seam) is shown in the following diagram:



6.2) Suggested Helper Functions and Structure

This is a pretty complicated process, so it will be helpful to break it down into smaller pieces.

While you are welcome to structure this code as you see fit, we have provided skeletons for a number of helper functions in `lab.py`, which we think can be a useful way to organize your code. We strongly encourage you to fill these functions in and use them to implement your overall seam-carving function, but you are welcome to ignore these helper functions (or modify their structure) if you wish; they will not be tested as part of your submission.

We have also provided test cases for these helper functions in the `test_seam_carving_helpers.py` file (which you can make use of in the same way as the normal `test.py`). These tests will *not* be checked on the server or included in your grade for Lab 2 (so you are welcome to comment them out or delete them if you prefer to use a different structure for implementing seam carving), but they can be helpful during debugging if you do choose to use the structure outlined below.

The helper functions we recommend include:

- `greyscale_image_from_color_image(image)`: given a color image, return a greyscale image

according to the formula above.
- `compute_energy(grey)`: given a greyscale image resulting from the step above, compute the energy map (using your `edges` function from last lab).
- `cumulative_energy_map(energy)`: given an energy map resulting from the above step, return the "cumulative energy map" as described above.
- `minimum_energy_seam(cem)`: given a "cumulative energy map," return a list of the indices into `im['pixels']` associated with the seam that should be removed
- `image_without_seam(image, seam)`: given an image and a list of indices, return a new image with the associated pixels removed.

The optional test cases in `test_seam_carving_helpers.py` assume that each of these returns a structure like a greyscale image (a dictionary with `'height'`, `'width'`, and `'pixels'` keys) but that these values are not necessarily valid images (i.e., the values in the `'pixels'` array may be outside the $[0, 255]$ range), except for `minimum_energy_seam`, which returns a list of integers.

6.3) Example with Intermediate Results

As you debug, it may also be helpful to have some concrete examples readily available. This section includes some intermediate results from removing one seam from `pattern.png`. While the exact structure of these results may be different (if you choose not to use the helper functions described above), the *values* contained in each should be the same regardless of your method.

Our greyscale copy of `pattern.png` should look like the following:

```
{'height': 4, 'width': 9, 'pixels': [200, 160, 160, 160, 153, 160, 160,
160, 200, 200, 160, 160, 160, 153, 160, 160, 160, 200, 0, 153, 160, 160,
160, 160, 160, 153, 0, 0, 153, 153, 160, 160, 160, 153, 153, 0]}
```

Computing the energy map produces:

```
{'width': 9, 'height': 4, 'pixels': [160, 160, 0, 28, 0, 28, 0, 160, 160,
255, 218, 10, 22, 14, 22, 10, 218, 255, 255, 255, 30, 0, 14, 0, 30, 255,
255, 255, 255, 31, 22, 0, 22, 31, 255, 255]}
```

Using this to compute the cumulative energy map yields the following (which looks very similar for this particular image but is actually different):

```
{'width': 9, 'height': 4, 'pixels': [160, 160, 0, 28, 0, 28, 0, 160, 160,
415, 218, 10, 22, 14, 22, 10, 218, 415, 473, 265, 40, 10, 28, 10, 40,
265, 473, 520, 295, 41, 32, 10, 32, 41, 295, 520]}
```

From which we find the minimum-energy seam contains the values indicated in red below:

Represented by the following list of indices (in any order):

```
[2, 11, 21, 31]
```

After removing the seam, the resulting image should look like `test_results/pattern_1seam.png`:

6.4) Write It!

Fill in the body of the `seam_carving` function in your `lab.py` so that it implements seam carving using the method described above. Note that both the input and output should be *color* images, and note that the `seam_carving` function should not modify the image that it is given as input.

6.5) Check Your Results

When you are confident that things are working as expected, try running your code on `twocats.png`, removing 100 seams from it (reducing the width by 100). Save your results (in color), and upload the resulting files below for testing. Note that this may take a **long** time to run (multiple minutes on most

machines)! You may wish to add some print statements (for example, printing how many columns have been removed every time a new column is removed) to help keep track of your program's progress.

Seam Carved `twocats.png`:
Select File No file selected

7) Something of Your Own

Finally, we would like for you to implement something new of your own choosing!

You are welcome to implement whatever you like here, and, unlike other portions of the lab, you are welcome to import the `math` or `random` modules in this section. If you want to use other modules, that might be ok, but please ask us first via `6.101-help@mit.edu`!

We encourage you to implement things that sound interesting to you! If you are having trouble thinking of what to do, here are some ideas that might help you get started, and we're certainly happy to answer questions as you work through things:

- You could implement some drawing primitives so that you can draw shapes on top of images. Maybe `circle(image, (r, g, b), x, y, radius)` draws a circle of the given size at the given location. You could implement multiple drawing primitives and use them to draw a nice new picture (or to modify an existing one!).
- You could implement some other new filters such as a directional emboss. You could even combine them with some of the others using `filter_cascade` to make a new effect.
- You could add the equivalent of "pasting" smaller images on top of larger ones (maybe also including a color that should be treated as transparent for purposes of the copy/paste, so that pixels of that color in the pasted image are not copied over).
- You could implement a different kind of filter that takes spatial information into account, in addition to color information, such as a vignette or a ripple filter.
- You could implement the equivalent of color curves, for example by implementing something like described in this article. Or, you could play around with the color content of images in other ways.
- You could implement the opposite of seam carving: smart resizing to *increase* the size of an image by inserting appropriate rows at low-energy regions in the image.

7.1) Submitting This Part

Regardless of what you choose to implement here, you should have some code that generates an image. Include your **documented** code for this part in `lab.py`, specifically in a **new** function called `custom_feature()`. This function should include all of the code that you needed to create your new custom feature, as well as code to generate an output image, and comments describing the change you implemented (including any additional imports that you used, if any).

Below are two boxes that you can use to upload images. If your custom feature is a *filter* (i.e., it operates on an input image), use the two boxes to upload an example of an input/output pair of images. If your custom feature simply generates an image (rather than operating on an existing image), you may leave the "Input image" box below empty.

Note: A green checkmark in this section only means that we have received your submission. You should be prepared to discuss your custom behavior during your checkoff.

Image before filter:

Input image (if any):
Select File No file selected

Image after filter:

Image generated by your custom feature:
Select File No file selected

8) 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_2 /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.

9) 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 implementation of `color_filter_from_greyscale_filter`.
- Your implementation of `make_blur_filter`.
- Your implementation of `filter_cascade`.
- Your implementation of the seam-carving.
- Your creative extension.

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

Footnotes

[1] This animation 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.

[2] For more information, see the Wikipedia page for seam carving.