#YesFilter Image Processing

Submission to website: Monday, February 13, 10pm

Checkoff by LA/TA: Thurdsay, February 16, 10pm (or three days after submission, if submitted late).

This lab assumes you have Python 3.5 or later installed on your machine. Please use the Chrome web browser.

Your final score is determined from the number of tests your code passes when running 'test.py'. In addition, the staff will award coding points during checkoff for code that is well organized -- read "Organizing your code" below before starting to code. This lab has 10 tests and 4 coding points.

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 (although there is some work to do the next best thing, some of it done right here at MIT). 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.

This week, you will build three such tools: *Inversion*, *Gaussian blur*, and a *Sobel edge detector*, which are together enough to reveal a license plate on a blurry, grainy photo. In a fascinating side note, many if not most classic image filters are implemented using a 2D image convolution kernel, the very same code you will write for this lab.

Digital Image Representation and Color Encoding

The representation of digital images and colors as data is a rich and interesting topic. (Have you ever noticed that your photos of purple flowers are unable to capture the vibrant purple you see in real life? This is the reason why.)

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 has a *width*, a *height*, and an array of *pixels*, each of which is a color value (the RGB encoding is a common representation for color values). This representation emerges 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.

In 6.009, we keep things simple: our images are grayscale (no color, only luminosity, or brightness), and each pixel is encoded as a single integer in the range `[0,255]` (256 values are representable by 1 byte), `0` being the deepest black, and `255` being the brightest white we can represent. The full range is shown below:

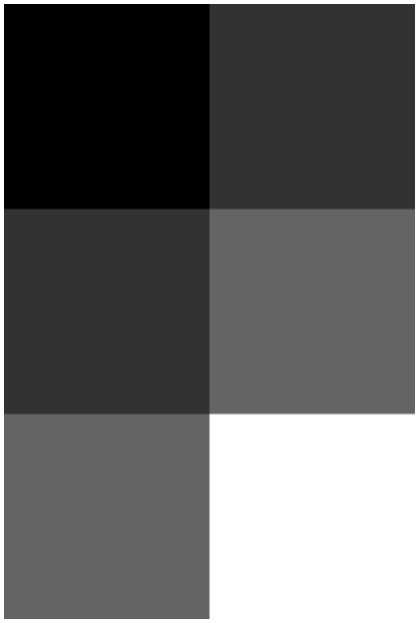
`0`  `255`

A digital image is an array of pixels stored in a linear array in row-major order listing the top row left-to-right, then the next row, and so on. So `pixels[0]` is the brightness at the top left of the image, and `pixels[x+width*y]` the brightness at pixel `(x,y)`.

A 6.009 image is represented by a dictionary, as shown below:

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

Here is the 2x3 image this encodes (figure below):



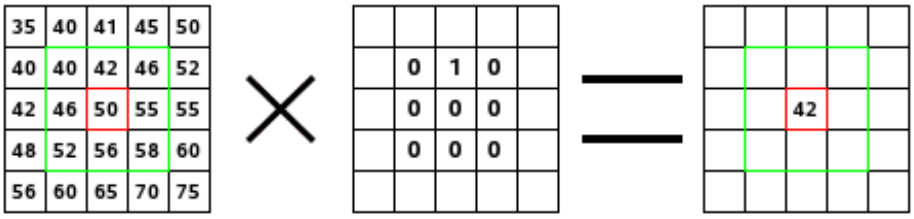
A 2D Convolution: Image Kernels

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`.

2D convolution kernels stem from a sophisticated foundation in math but in this case are *2D matrices*. Each is a rectangular array of numbers, specifying how to combine pixels from the source image into each pixel of the output. In this lab, we will concern ourselves with 3x3 kernels only, meaning each pixel of `o` is computed from `3*3=9` pixels of `i`, multiplied by the appropriate values in `k`.

To apply `k` (to convolve it with `i`), each output pixel (x,y) is given by a linear combination of the 9 pixels nearest to (x,y) in `i`, multiplied by the corresponding values in `k`, as shown in the diagram below:



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 (0,0).:

```
o[0,0] = i[0,0]*k[1,1] + i[1,0]*k[2,1] + i[0,1]*k[1,2] + i[1,1]*k[2,2]
```

NOTE: this is *not* valid Python! Arrays cannot be indexed with `x` and `y` coordinates!

(`i[-1,-1]`, `i[-1,0]`, `i[-1,1]`, `i[-1,0]`, `i[1,-1]` are all out of bounds of `i` and are assumed to have a value of 0 (black). This is correct and may cause dark edges in `o`.) This process is repeated for each pixel (x,y) of `o`, with the kernel centered over (x,y). By extension,

```
o[x,y] = i[x-1,y-1]*k[0,0] + i[x,y-1]*k[1,0] + i[x+1,y-1]*k[2,0] +
         i[x-1,y]*k[0,1] + i[x,y]*k[1,1] + i[x+1,y]*k[2,1] +
         i[x-1,y+1]*k[0,2] + i[x,y+1]*k[1,2] + i[x+1,y+1]*k[2,2]
```

Take care to exclude terms that fall outside of the dimensions of `i`.

Some additional examples:

```
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.

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

The above kernel shifts the input image one pixel to the*right*, discards the rightmost column of pixels, and blacks out the leftmost column.

```
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.

There is a world of interesting operations that can be expressed as image kernels (many scientific programs also use this pattern), so feel free to experiment.

Note that the output of a convolution need **not** be a legal 6.009 image (pixels may be outside the `[0,255]` range). So the final step in every image-processing function is *clip* negative pixel values to 0 and values > 255 to 255.

`lab.py`

This file is yours to edit in order to complete this lab. You are not expected to read or write any other code provided as part of this lab. In `lab.py`, you will find some highly suspect code implementing the `filter_invert` function. Your first job is to do some detective work to figure out what is wrong with this code and fix it. We also ask you to implement two other functions for different image filters: `gaussian_blur` and `edge_detect`, following the rules that we will present shortly. All three filters must pass their tests to get full credit for the lab.

Your code will be loaded into a small server (`server.py`) and will serve up a visualization website, which we recommend to visually debug your image filters. To use the visualization, run `server.py` and use your web browser to navigate tolocalhost:8000. You will need to restart `server.py` in order to reload your code if you make changes.

You are responsible for the following functions:

Part 1: A broken inversion (`filter_invert`) [tests 1-4]

We, your friendly 6.009 staff, implemented an*inversion* filter. This filter reflects pixels about the middle gray value (`0` black becomes `255` white and vice versa). Our implementation unfortunately has some problems. Please fix this function before proceeding to write your filters in parts 2 and 3!

Create a test image `resources/images/gradient.json` to make debugging this code easy:

```
{
  "height": 1,
  "width":  4,
  "pixels": [ 0, 64, 128, 255 ]
}
```

Furthermore, add a test case to verify your code! To do this, create a pair of files in `cases/`, much like the other test cases (these **must** have integer names; name this test "11.in" and "11.out"). The `.in` file describes which filter to apply and which image in `resources/images` to apply it to.

```
{
  "test": "This test tries to invert a tiny image for manual inspection",
  "function": "filter_invert",
  "image_file": "gradient.json"
}
```

The corresponding `.out` file is quite simply the result of manually applying the filter to the input:

```
{
  "height": 1,
  "width":  4,
  "pixels": [ 255, 191, 127, 0 ]
}
```

Happy debugging!

Part 2: Things get blurry with `filter_gaussian_blur( image )` [tests 5-7]

For this part of the lab, implement aGaussian blur `filter_gaussian_blur(image)` with a 3x3 kernel given below:

```
1.0/16 2.0/16 1.0/16
2.0/16 4.0/16 2.0/16
1.0/16 2.0/16 1.0/16
```

In this case, the Gaussian blur is a simple 2D convolution of the input image with the kernel above. Precisely, the output should be an *image* with height and width equal to the dimensions of the input. Take care to output *integer* brightness values (`int(round(value))` , for example) in range `[0, 255]` . That is, in many cases you will need to clip integer values explicitly to this range.

Create two additional test cases:

- Gaussian blur of a 6-by-5 image consisting entirely of black pixels. The output is trivially identical to the input. Gaussian blur for the `centered_pixel.json` image. You can compute the output image manually with relative ease,

Use the web UI to debug your implementation visually.

Part 3: Things get edgy with `filter_edge_detect( image )` [tests 8-10]

For this part of the lab, implement a Sobel operator, `filter_edge_detect(image)` , as described below:

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 convolving 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) = int(round((Ox(x,y)**2 + Oy(x,y)**2)**0.5))` , for example.

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.

Create a new test case: edge detection on the `centered_pixel.json` image. The correct result is a white border at the perimeter of the image, and a white ring around the center pixel, both 1 pixel wide. How many unique patterns can be created by running edge detection against the "centered dot" test case from part 2? Hint: it's 20, and that's a lot!

How to run your code

Once your code produces *images*, you can visualize the output to help you debug your kernel code. Run `server.py` and open your browser to localhost:8000. You will be able to select the image and filter and see the result.

The web UI loads all images from `resources/images/` in our special 6.009 `.json` format. (You can convert your favorite images with our handy conversion tool: `resources/convert_image.py kittens.jpg > resources/images/kittens.json` . Note that it only works if you have the Python `Image` library installed, while we've kept the rest of the lab compatible with base Python installations.) Furthermore, all functions in `lab.py` named `filter_<anything>` can be used via the web UI.

To verify your code before submitting it to the 6.009 online system, use the `test.py` script, which runs the very same set of test cases against your code as our auto-grader will.

Organizing your code

Coding points will be awarded if you write, test, and use the following functions when implementing your filters (one codign point per function).

`get_pixel(image,x,y,default=0)`

> If `0 <= x < image['width']` and `0 <= y < image['height']`, returns the value of the image pixel at (x,y). If the requested pixel is outside the image, return the `default` value. The plan is to access pixel values by using *only* this function.

`convolve2d(image, kernel3x3)`

> Returns a new image constructed by running the 3x3 kernel over the provided image, where the output is *not* forced into the range [0,255]. If you feel fancy, consider generalizing it to any `N`x`M` kernel.

`combine_images(image1, image2, f)`

Returns a new image where `f` is a **function** of the form `combine_f(pixel1, pixel2)` that describes *how* the two images are to be combined. This is useful for adding and averaging images, to mention just a few.

`legalize_range(image)`

that forces each pixel of `image` into integers in the range `[0,255]`. Returns the updated image as its value.

Extensions

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:

- Implement a 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`. If you call your function `filter_normalize`, or `filter_<anything>`, you will be able to use it via the web UI.
- Extend your kernel code to support kernels larger than 3x3 and create a larger Gaussian blur.
- 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.
- If you wish to put your edge detection through a more rigorous test, create a new test image and corresponding test case: a 60-by-60 checkerboard pattern of black and white tiles. Each tile should be 10-by-10 pixels (we sincerely hope you don't try to do this manually). Write a tiny Python program to print out the desired pixels. If your Python installation includes the `Image` library (this unfortunately isn't standard), you may also generate it via `resources/convert_image.py resources/tile.jpg > resources/images/tile.json`. The `Kx` convolution marks vertical edges, while `Ky` marks horizontal edges. Together, the two convolutions should find all edges in the tile image, creating vertical and horizontal white lines at the tile boundaries.