Computational Photography
Assignment #4: Gradients & Edges

Ram Subramanian
Fall 2016

Part 1: normalizeImage

```
img = src.astype(np.float_)

img = img - img.min()

img = img * 255.0 / img.max()

return img.astype(src.dtype)
```

To normalize an image we want to scale all the intensities linearly such that:

1. The lowest intensity equals 0.
2. The highest intensity equals 255.

For (1), we simply subtract the lowest value from all values. The range now is [0, max-min].

For (2), we divide all intensities by the highest value. The range now is [0.0, 1.0].

All that's left to do is multiply all intensities by 255.0 and convert to the needed dtype.

Part 1: gradientX & gradientY (with test images)

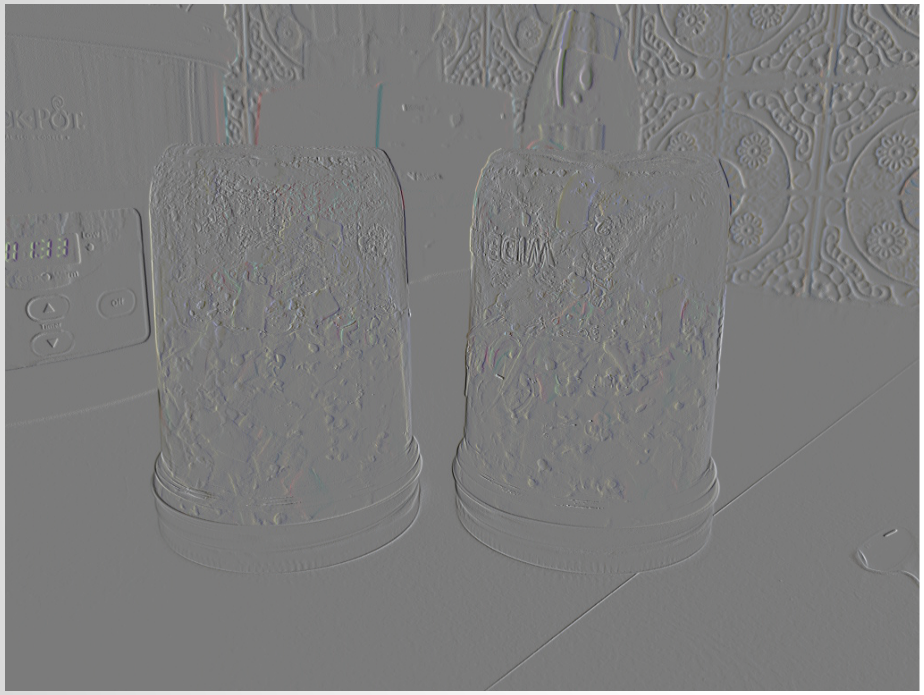
```
img = img[:,1:] - img[:,:-1]  ## gradient X (img - signed dtype, atleast 16 bits wide)

img = img[1:] - img[:-1]      ## gradient Y (img - signed dtype, atleast 16 bits wide)
```

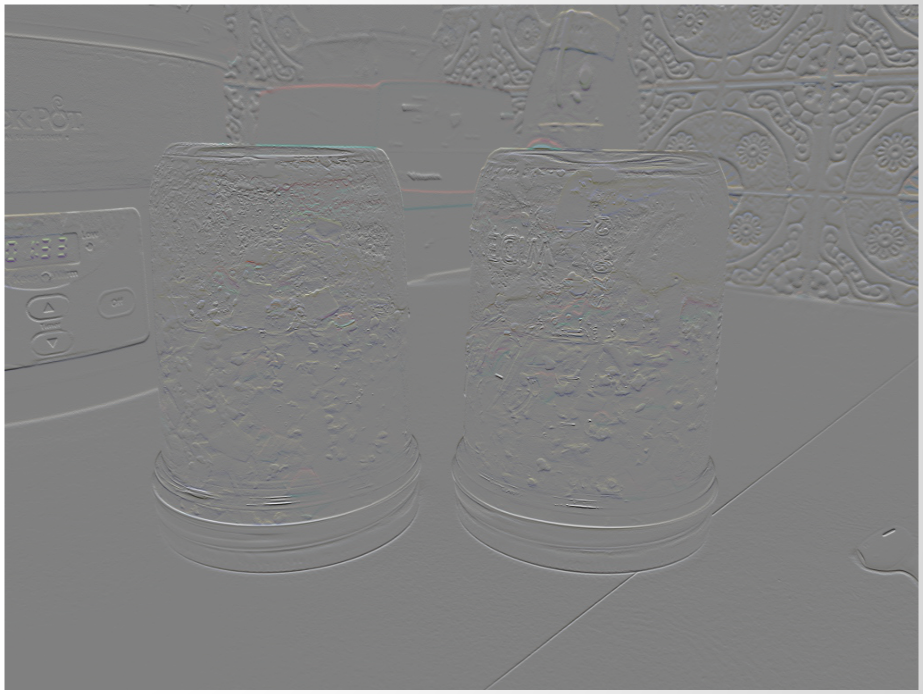
To calculate the gradient in the X direction, we use array splicing to extract all but the first column (`img[:,1:]`), all but the last column (`img[:,:-1]`) and subtract the latter from the former. Since they both have one less column than the original image, they will of the same dimensions thus allowing any arithmetic operations with each other.

Similarly, to calculate the gradient in the Y direction, we extract all but the first row (`img[1:]`), all but the last row (`img[:-1]`) and subtract the latter from the former.

Since we are not expected to normalize it, we simply return the results in each case.

Part 1: gradientX & gradientY - Test Images


gradientX


gradientY

We need to normalize the gradient image to be in the range of [0.0, 255.0] because that’s the range of values cv2.imwrite() supports and expects. Since normalization is linear, no data is lost (except when converting to a lesser datatype viz. float to int)

Part 1: padReflectBorder

```
img = np.hstack((image[:,1:N+1][:,::-1], image, image[:,-N-1:-1][:,::-1]))

img = np.vstack((img[1:N+1][::-1], img, img[-N-1:-1][::-1]))
```

I tackle this problem in 2 parts. The first pads the left and right of the image, the second the top and bottom. An additional benefit of this method is that corners are handled correctly and automatically.

1st part: I use array splicing to extract the first N columns (while skipping the first, `[:,1:N+1]`) and the last N columns (while skipping the last, `[:,-N-1:-1]`), reverse them (`[:,::-1]`) and stack them on either side of the original image (`np.hstack`).

2nd part: Similarly, I extract the first N rows (while skipping the first, `[1:N+1]`) and the last N rows (while skipping the last, `[-N-1:-1]`), reverse them (`[::-1]`) and stack them on top and below the original image.

Comparing my function to cv2.copyMakeBorder(), it takes around the same time to complete. One difference I could make out was when N >= min(height, width) – since it's reflect_101, when N equals or exceeds the smaller dimension, undefined behaviour results (i.e. the image may not be padded by the same amount on all four sides), whereas copyMakeBorder seems to handle this case very well.

```
>>> cProfile.run('cv2.copyMakeBorder(img, 767, 767, 767, 767, cv2.BORDER_REFLECT_101)')

4 function calls in 0.004 seconds

>>> cProfile.run('padReflectBorder(img, 767)')

49 function calls in 0.004 seconds
```

Part 1: crossCorrelation2D (with own images)

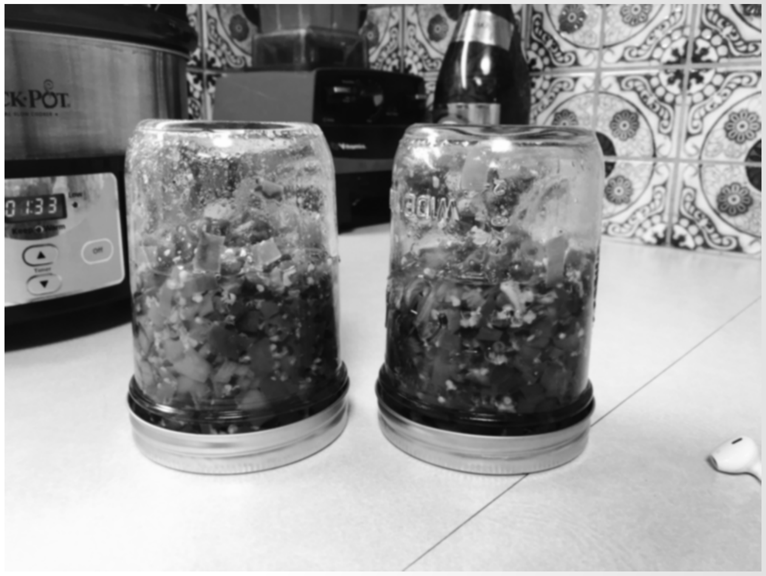
To apply a kernel/filter to an image, I simply “run” the kernel over the image at every pixel. What I mean by “run” a kernel is, say this is the image patch on which we wish to apply the kernel:

1 2 3	1 1 1		1 2 3		
4 5 6	1 1 1	=>	4 5 6	=>	1+2+3+4+5+6+7+8+9 = 45 (new pixel value)
7 8 9	1 1 1		7 8 9		
Image	kernel				

We simply multiply each element from the image to the corresponding element in the kernel and sum all the resulting values together. This will be the new filtered value stored at the center of the image patch. Since kernels are usually odd-dimensioned, there will always be an exact center to place the calculated value at.

The resulting filtered image may need to be normalized by the sum of the entries in the kernel.

Part 1: crossCorrelation2D - Image Results

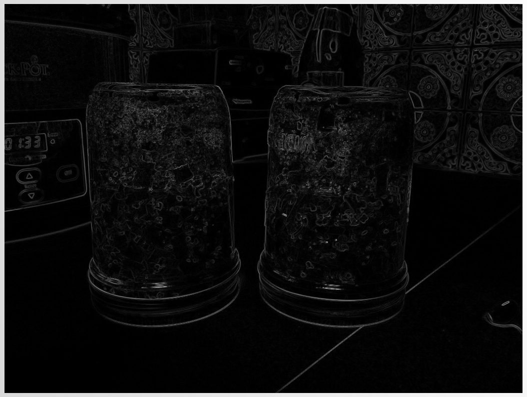


Note: When performing the cross correlation I had to return the result as np.int64, rather than the np.int16 that the autograder expects.
I also had to divide the result of pyFilter2D by the sum of the GAUSSIAN_KERNEL to normalize the image.

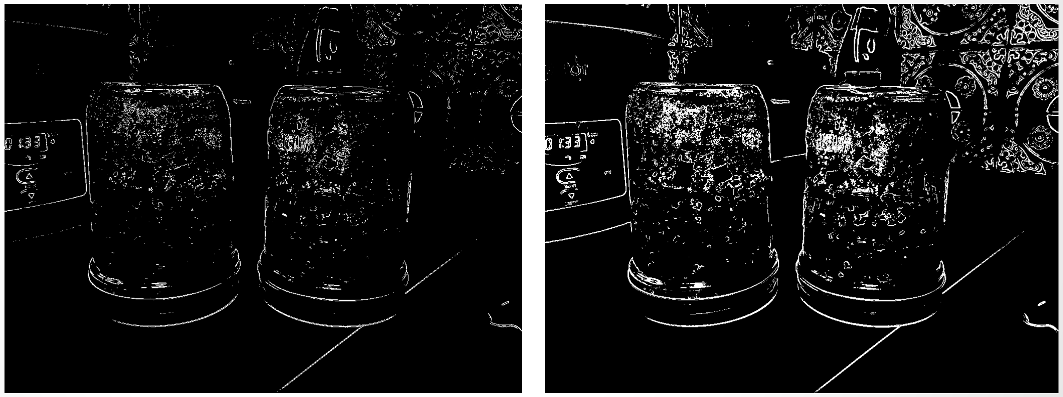
Part 2: Edge Detection

First method I tried was to compute the magnitude of gradients at each pixel. This can be done by finding the partial gradients in the X and Y direction and combining them together. Already the edge image looked quite good (shown below). All that needed to be done was to threshold and smooth to get rid of any noise – but this proved to be the most difficult. Selecting the parameters for the thresholding and smoothing was a constant juggle between getting rid of noise and keeping good edges.

I also tried other well known operators for finding gradients, operators such as Sobel and Laplacian. However, since the trouble area for my algorithm was after finding the gradient, these did not yield much better results.

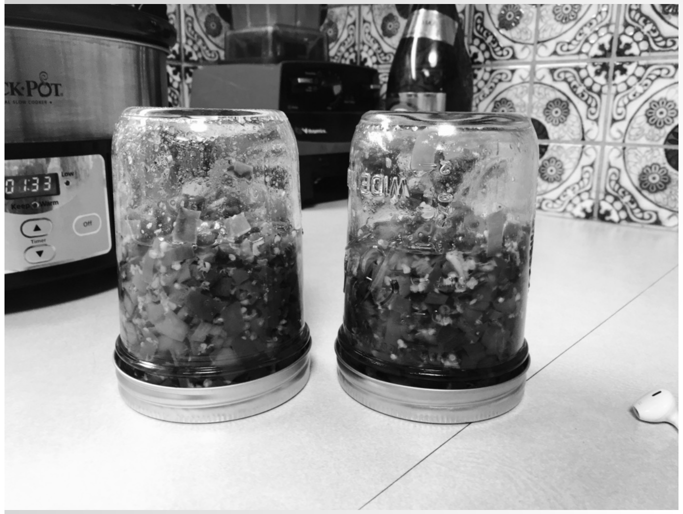


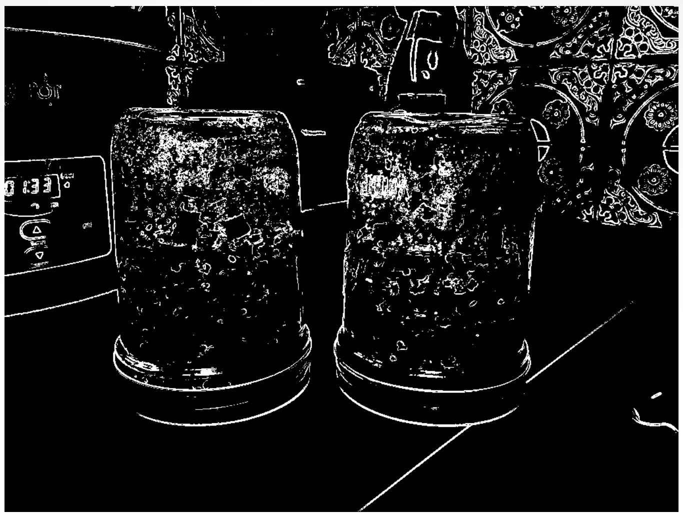
Gradients at each pixel
Intensity = magnitude of gradient

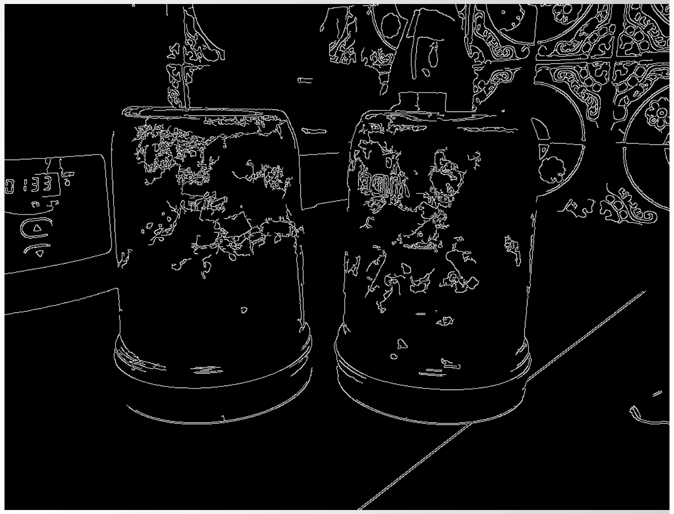


The edges at various threshold values

Part 2: Edge Detection - Image Results


Original


My algorithm


Canny

Since my algorithm doesn't perform edge thinning, the edges are a lot thicker in my image compared to the one produced by canny. There is also a lot more noise in my image, but with more smoothing, I start to lose the good edges as well..

My algorithm also doesn't perform non-maximal suppression unlike Canny – this causes some edges to be broken. My attempts to mimic this functionality were unfortunately not very impressive.

Part 2: Edge Detection - Code

```
def find_edges(image):

    gx = a4.gradientX(image).astype(np.int64)
    gx = np.hstack((gx, np.zeros((image.shape[0], 1))))

    gy = a4.gradientY(image).astype(np.int64)
    gy = np.vstack((gy, np.zeros((1, image.shape[1]))))

    z = np.sqrt(gx**2 + gy**2)
    z = cv2.GaussianBlur(z, (3, 3), 0.5)        ## faster :)

    th = 30
    z[z >= th] = 255
    z[z < th] = 0

    return z
```

Your Observations

I thought that the idea of non-maximal suppression is quite genius as we are bound to lose some good pixels to smoothing and thresholding – like I said before my attempts at this weren’t very good and so I’ll definitely be looking into this more.