Project 1 Writeup

Instructions

- Provide an overview about how your project functions.
- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- List any extra credit implementation and result (optional).
- Use as many pages as you need, but err on the short side.
- **Please make this document anonymous.**

Project Overview

This project allows you to filter images by convolution given an image and a kernel and create hybrid images from 2 images.

Implementation Detail

I followed the general steps given on the project website. The first step was to pad the image. To do this, I first got the kernel's dimensions and checked whether either of them were even- if it was, an error is raised. Since the kernel dimensions can be different, I then made 2 variables, `pad_rows` and `pad_cols`, to hold the number of pixels to pad on the sides of the image given by the corresponding kernel dimension divided by 2 (ignoring the remainder).

The next step to pad the image was to support grayscale and color images. Since `numpy.pad`'s `array_like` argument changes depending on the input's shape, I added an `if` statement to set it to the correct padding amount like so:

```
if len(image.shape) == 2:
    padding_dim = ((pad_rows, pad_rows), (pad_cols, pad_cols))
else:
    padding_dim = ((pad_rows, pad_rows), (pad_cols, pad_cols), (0,0))
padded_image = np.pad(image, padding_dim)
```

After this I stored the original image's dimensions in `img_rows` and `img_cols` and rotated the kernel 180 degrees using `numpy.rot90`. Next was to apply the kernel to the image.

I used another `if` statement to handle the grayscale and color images separately. For both cases I used nested for loops to iterate through each pixel in the padded image, excluding padding. I initially tried to find a way to do this without using for loops but was unsuccessful. In the grayscale case I first determined the overlay area where the kernel would fit over the padded image. I then multiplied this with the rotated kernel and summed all the values up. This looked like:

```
for i in range(pad_rows, img_rows + pad_rows):
    for j in range(pad_cols, img_cols + pad_cols):
        overlay_arr = padded_image[i - pad_rows:i + pad_rows + 1, j -
                                   pad_cols:j + pad_cols + 1]
        val = (overlay_arr * rotated_kernel).sum()
        filtered_image[i - pad_rows, j - pad_cols] = val
```

Similarly for the color case, I first determined the overlay area and multiplied it with the rotated kernel. To adjust this to support 3 channels, I stacked three identical rotated kernels to multiply with the overlay. Then I computed the sums separately for each channel and set the corresponding `filtered_image` value to them. This looked like:

```
for i in range(pad_rows, img_rows + pad_rows):
    for j in range(pad_cols, img_cols + pad_cols):
        overlay_arr = padded_image[i - pad_rows:i + pad_rows + 1, j -
                                   pad_cols:j + pad_cols + 1,
                                   :]
        mult_kernels = overlay_arr * np.stack((rotated_kernel,
                                   rotated_kernel,
                                   rotated_kernel), axis = 2)

        sum1 = np.sum(mult_kernels[:,:,0])
        sum2 = np.sum(mult_kernels[:,:,1])
        sum3 = np.sum(mult_kernels[:,:,2])
        filtered_image[i - pad_rows, j - pad_cols] = [sum1, sum2, sum3
                                   ]
```

I originally calculated the sums in one line using `numpy.stack` but after comparing the execution time I settled with the above method since it requires less operations and is therefore faster

```
sums = np.stack((np.sum(mult_kernels[:,:,0]), np.sum(mult_kernels[:,:,
                           1]), np.sum(mult_kernels[:,:,2])),
                           axis=0)
sums = np.expand_dims(sums, axis = (1,0))
filtered_image[i - pad_rows, j - pad_cols] = sums
```

The last step I did was to clip the image so all the values would be between 0 and 255 inclusive.

`gen_hybrid_image` was fairly straightforward to fill out. I just replaced the filler code with calls to `my_imfilter`.

Result

Here are some results from filter_test using the cat image:

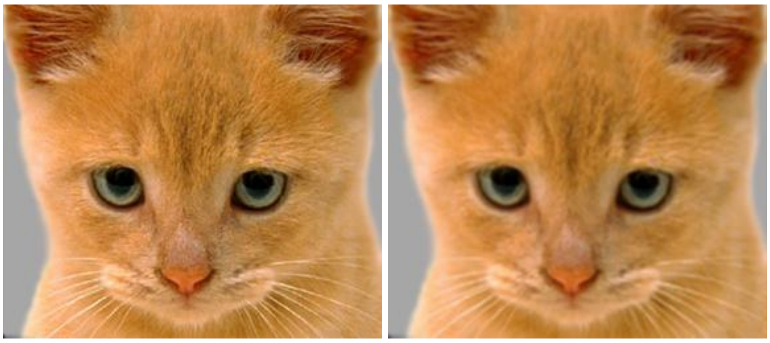

Figure 1: *Left:* Identity image. *Right:* Blur image.

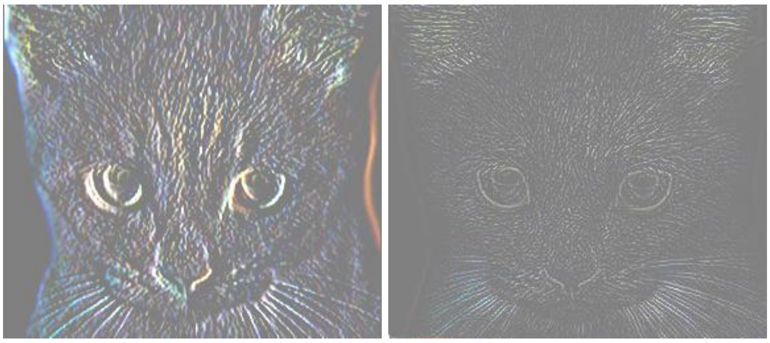

Figure 2: *Left:* Sobel image. *Right:* Laplacian image.

My results came out as I expected. I compared the identity image to the original image and all the pixels are the same.

Filter	Time (seconds)
Identity	1.63849
Small blur	1.53775
Large blur	1.56644
Oriented	1.53352
High pass	1.54334

Table 1: Time to run `my_imfilter` on each filter in `filter_test`.

Extra Credit (Optional)

1. To pad with the reflected image I modified this line:

```
# original pad with zeros:
padded_image = np.pad(image, padding_dim)
# pad with reflected image content:
padded_image = np.pad(image, padding_dim, 'reflect')
```

2. I attempted to create a hybrid image of Benedict Cumberbatch and Abraham Lincoln. Here is the result:

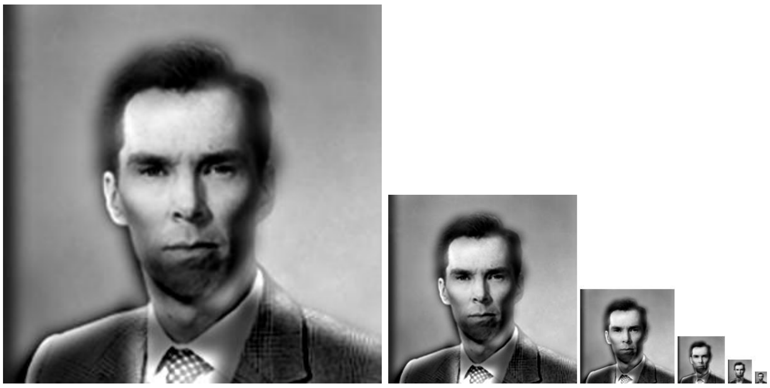

Figure 3: Hybrid image.

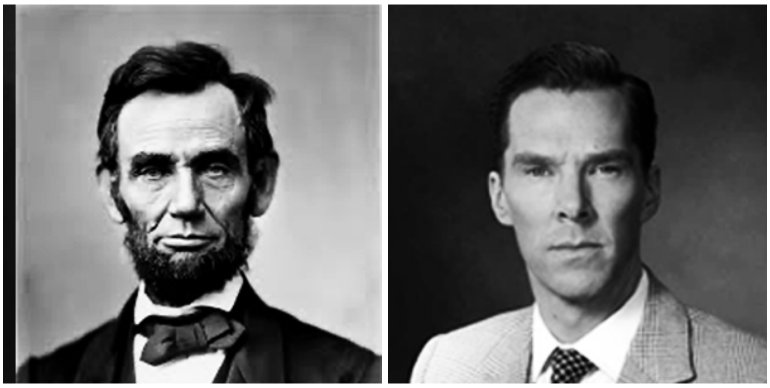

Figure 4: Original Images