Image Processing HW3

tags: **2019imageprocess**

github url: https://github.com/linnil1/2019ImageProcessing/tree/master/hw3 (https://github.com/linnil1/2019ImageProcessing/tree/master/hw3)

hackmd url: https://hackmd.io/pjj-D3fhRr6dNH-HbhsLYQ (https://hackmd.io/pjj-D3fhRr6dNH-HbhsLYQ)

Part1

3.22

> (A)
>
> $$v = [\,1 \quad 2 \quad 1\,]^T$$
> $$w^T = [\,2 \quad 1 \quad 1 \quad 3\,]$$
> is vw^T separable.

Yes, trivial.

> (B) Find w_1 and w_2 that formed
>
> $$w^T = \begin{bmatrix} 1 & 3 & 1 \\ 2 & 6 & 2 \end{bmatrix}$$

Still, trivial

$$w_1 = [\,1 \quad 2\,]^T$$
$$w_2 = [\,1 \quad 3 \quad 1\,]$$

3.27

> An image is filtered four times using a Gaussian kernel of size 3 × 3 with a standard deviation of 1.0. What is the equivalent results can be obtained using a single Gaussian kernel formed by convolving the individual kernels.
>
> (a) What is the size of the single Gaussian kernel?

Every time when covolute with 3x3 the filter, we get 2 more row and colunms. So $3 + 2 * (4 - 1) = 9$, so the result filter size is 9x9.

> (b) What is its standard deviation?

The convolution is multiplication in Frequency domain.

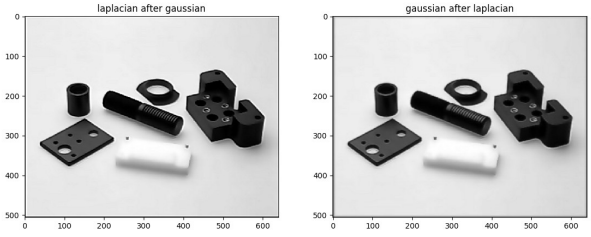
Note the Fourier transform of gaussion is $F(\exp(ax^2)) = \sqrt{\pi/a}\,\exp(-\pi^2 x^2/a)$

After four times multiplication in frequency domain the result is $\sqrt{\pi/a}\,\exp(-4\pi^2 x^2/a)$, and it's inverse is $1/2 * \exp(ax^2/4)$. Correspond to normal distribution $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}}$, we found σ is twice as before.

3.38

> Applying smoothing kernel to input images to reduce noise, then a Laplacian kernel is applied to enhance fine details.
>
> Would the result be the same if the order of these operations is reversed?

The noise become significant when Laplacian goes first. In the other hand, the noise be smoothing when gaussian apply first, so the edge is the only one be detected by laplacian filter.



4.3

> What is the convolution of two, 1-D impulses:
>
> - $\delta(t)$ and $\delta(t - t_0)$

$\delta(t) \bigstar \delta(t - t_0) = \int_{-\infty}^{\infty} \delta(\tau)\delta(t - t_0 - \tau)d\tau$ Two impulse have value when $\tau = 0, t - t0$ simultaneously. Thus, the result is $\delta(t - t_0)$.

> - $\delta(t - t_0)$ and $\delta(t + t_0)$?

Same as above. Two impulse have value when $\tau = -t_0, t - t0$ simultaneously. Thus the result is $\delta(t)$.

4.32

> For arrays of even length, use arrays of 0's ten elements long. For arrays of odd lengths, use arrays of 0's nine elements long. Provided the centers coincide to keep the symmetry of the original array.
>
> - (a) {a, b, c, c, b}
> - (b) {0, – b, – c, 0, c, b}
> - (c) {a, b, c, d, c, b}
> - (d) {0, – b, – c, c, b}

- (a) Can't
- (b) {0, 0, 0, -b, -c, 0, c, b, 0, 0} is odd
- (c) Can't
- (d) {0, 0, 0, -b, -c, c, b, 0, 0} is odd

Part 4:

> Implement an order-statistic filter function which is flexible for users to select: (1) median filter, (2) max filter, (3) min filter. The function should also be flexible in choosing different filter sizes. Test your program function with Figure 3.43(a), and compare the result with Gaussian filter. Discuss the effect of filter size on the image processing result.

Naively implement:

```
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        new_img[i, j] = func(data[i:i+size[0], j:j+size[1]])
```

It takes `10.1714s` on my computer to apply median filter on image 4-1. So, optimization is needed.

Using more space to make it faster, I aggregate all the items need to be multiply in the kernel size to new axis, then calculate median along third axis.

```
for i in range(size[0]):
    for j in range(size[1]):
        new_img[:, :, i * size[1] + j] = data[i:i+img.shape[0],
                                              j:j+img.shape[1]]
return np.median(new_img, axis=2)
```

It takes `1.723s` in the same condition. However, if the kernal size is large enough, the program will raise error for the insufficient space.

Using sliding window and image histogram proposed by Hang (Hang. et al., 1979), I can reach 0.0428s with c++ codes. The sliding windows eliminate the time complexity to $O(r)$, where r is the kernel radius. The concept of histogram is same as bucket, we can get median in $O(1)$. The min and max filter are similar too. Finding the value of the first or the last one in histogram.

You can see more in the c++ code in `hw3/utils.cpp`

reference

- T. Huang, G. Yang, and G. Tang, "A Fast Two-Dimensional MedianFiltering Algorithm,"IEEE Trans. Acoust., Speech, Signal Processing,vol. 27, no. 1, pp. 13–18, 1979

Part 2:

> Design a computer program for spatial filtering operations using various types of masks. Test your program with several images and report your results. Discuss the effect of mask size on the processed images and the computation time.
>
> You should design a mask operation function that is flexible for adjusting mask size and setting coefficients in the mask.

Convolution still a costly function if you write in numpy. As the aggregation method mentioned before, we get the fastest speed by numpy with this code.

```
new_img = np.zeros([*img.shape, size[0] * size[1]])
for i in range(size[0]):
    for j in range(size[1]):
        new_img[:, :, i * size[1] + j] = data[i:i+img.shape[0],
                                              j:j+img.shape[1]]
return new_img.dot(kernal.flatten())
```

The complexity is still $O(N_k M_k)$. It takes 0.838s to complete, not to say applying larger kernel. So, we need to use FFT to make it faster.

A simple, 1D fourier transform:

```
data = [1, 2, 4, 4]
n = len(data)
j, i = np.meshgrid(np.arange(n), np.arange(n))
ftdata = np.sum(data * np.exp(-np.pi * i * j * 2j / n), axis=1)
invftdata = np.real(np.sum(ftdata * np.exp(np.pi * i * j * 2j / n), axis=1) / n)
```

Because the complexity of fourier transform above is $O(n^2)$, so we implement $O(log(n))$ fft for faster calculation.

```
def fft(data):
    if len(data) == 1:
        return data

    even = fft(data[::2])
    odd  = fft(data[1::2])
    n_fft = len(data)
    i = np.arange(n_fft / 2)
    w = np.exp(-np.pi * 2j * i / n)
    fftdata = np.concatenate([even + odd * w,
                              even - odd * w])
    return fftdata
```

The next step is to apply it on 2D. The trick is same. The original formula of fourier transform for 2D is

$$F(u,v) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} f(n,m) \exp(-2\pi i \frac{mu}{M} \frac{nv}{N})$$

Then, separate n and m

$$F(u,v) = \sum_{u=0}^{m-1} [\sum_{v=0}^{n-1} f(n,m) \exp(-2\pi i \frac{nv}{N})] \exp(-2\pi i \frac{mu}{M})$$

With this formula, we can use two time fourier transform to complete. The complexity is $O(log(max(N,M))$

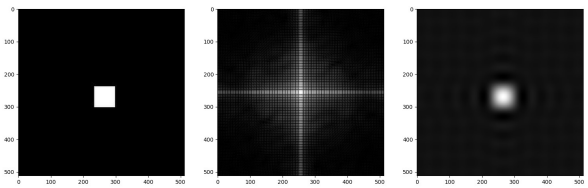
```
def fft2d(data):
    data_fft_row = np.stack([fft(i) for i in data])
    data_fft_col = np.stack([fft(i) for i in data_fft_row.T]).T
```

My code takes .281s, which is much slower than `numpy.fft.fft2` with 0.004s. After I rewrite into c++ and optimized by g++ O3, we got 0.005s, almost the same as official.

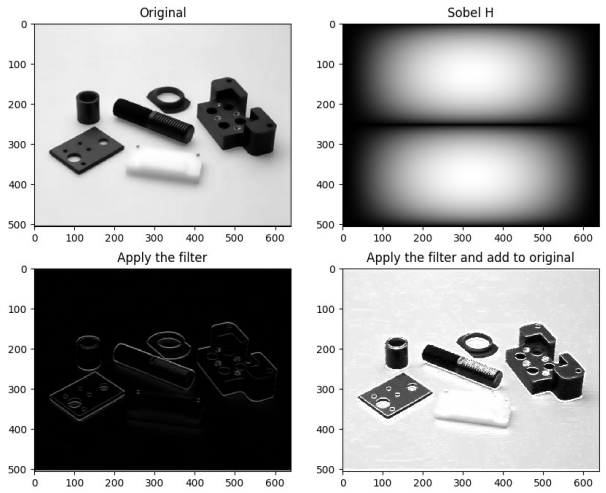
After proper padding, translation and log transform, we can reproduce the result in textbook.

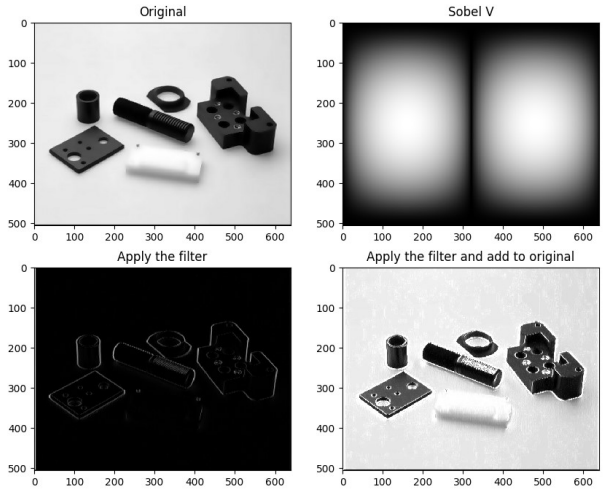
I also found that centerized is not only to get the better visualization, but easier in calculation because we don't need to deal with boundary conditions. Such as setting the ideal cutoff functionin the below code.

```
cx = data2d.shape[0] // 2
cy = data2d.shape[1] // 2
j, i = np.meshgrid(np.arange(data2d.shape[0]) - cx, np.arange(data2d.shape[1]) - cy)
fftdata[i ** 2 + j ** 2 > 10000] = 0
```



Finally, convolution in spatial is equivalent to multiplication in frequency domain. Thus, we multiply the value of kernel and the original image pixelwisely, both after fourier transform. Here are some results of spectrum of some kernels, the others can be found in `hw3/data` .





Part 3:

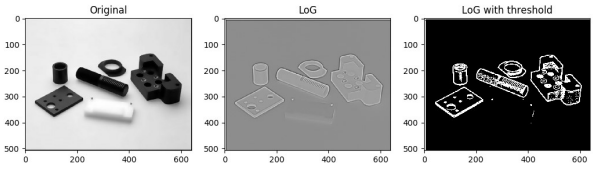
> Laplacian of Gaussian (LoG) is a second derivative of a Gaussian filter. The LoG can be broken up into two steps. First, smooth the image with a Gaussian filter, and second, convolve the image with a Laplacian mask. Read the Section 10.2 of our textbook for a detailed theory and procedure of this edge detection method.
>
> Implement a computer program for edge detection using the Marr-Hildreth edge detector.

The equation look like this:

$$LoG(x,y) = \frac{x^2 + y^2 - 2\sigma^2}{\sigma^4} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

```
def LoG(img, sig):
    n = int(sig * 6)
    n += (n % 2 == 0)
    j, i = np.meshgrid(np.arange(n) - n // 2,
                       np.arange(n) - n // 2)
    krn = (j ** 2 + i ** 2 - 2 * sig ** 2) / (sig ** 4) * \
          np.exp(-(j ** 2 + i ** 2) / (2 * sig ** 2))
    return spatialConv(img, krn)
```

Then apply the threshold > 0, we can get the edges with the property of isotropic.

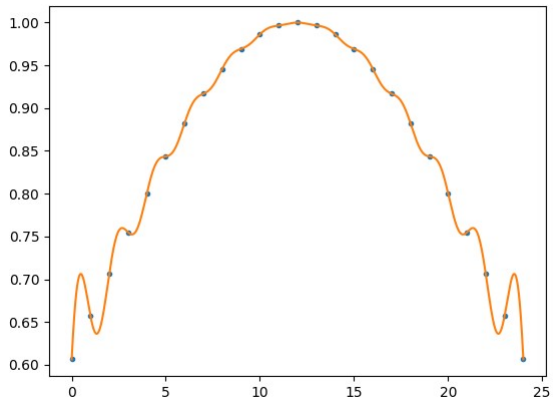


Interpolation by FFT

We use sinc func to interpolation when we see the data as a contiuous function.

Here is the result

```
n = 25
kn = 1000
kx = np.arange(kn) / (kn - 1) * (n - 1)
arr = np.sinc(np.repeat([kx], n, axis=0).T - np.arange(n)).dot(y)
```



Program usage

Install on local machine

```
./setup.sh
```

Install by Docker

```
docker build -t linnil1/2019imageprocess .
docker run -ti --rm \
    --user $(id -u) \
    -v $PWD:/app \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -e DISPLAY=$DISPLAY \
    linnil1/2019imageprocess python3 qt.py
```

Without QT

You can use command line as a postfix image processor.(Same as HW2)

For example

- `python3 hw3_np.py --read data/Image\ 3-2.JPG --graya --kernel "0 1 0; 1 -4 1; 0 1 0" --show`
- `python3 hw3_np.py --read data/Image\ 3-2.JPG --graya --laplacian8 1 --show`

You can use `python3 hw3_np.py --help` to see more.

QT

`python3 qt.py` or `./run.sh`

Note: all filter should run on gray scale image.