Assignment 3

Junyi Liao 20307110289

October 23, 2022

1 Spatial Filter

1.0 Filter in image processing

Given a kernel w of size $(2a+1, 2b+1)$, we do spatial filtering by moving it on a given image f along ergodic paths, and do algebraic operation over w and the covered pixels of f, say the subimage.
For a linear filter, sum of elementwise product over w and subimage in f is computed by spatial correlation:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x+s, y+t),$$

With a central symmetric kernel, it has the same form as convolution:

$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t);$$

Thus we get a filtered image g by hitting each pixel in f. To deal with the boundary of f, where the kernel doesn't fully match with pixels in f, we do padding on f by adding zeros outside.
Also some non-linear filters are commonly used in image processing, such as median filter, which is often used in smoothing. It can be mathematically expressed:

$$g(x,y) = \text{Median}\{f(x+s, y+t) \,|\, -a \le s \le a,\ -b \le t \le b\},$$

which is an order statistic of a subimage. In this case only pixels covered by the kernel are taken into account while dealing with the boundary of f.

The code of two kinds of filters are shown below. To visualize the filtered image we may need postprocessing, which is given as normalization in our code.

```
import numpy as np

# Pad the bound of an image with pixel 0 so that the filter can deal with
    pixels on the bound.
def padding(img, pad):
    h, w = img.shape
    img_pad = np.zeros((h + 2 * pad, w + 2 * pad))
    img_pad[pad:h+pad, pad:w+pad] = img
```

```
    return img_pad

# Compute the inner product of filter and sub-image.
def inner_prod(x, y):
    if not x.shape == y.shape:
        raise ValueError
    return np.sum(x * y)

# Linear filter, in form of spatial-correlation.
def lin_filter(img, kernel):
    h, w = img.shape
    img_filter = np.zeros_like(img)
    edge = kernel.shape[0]
    img_pad = padding(img, edge // 2)
    # Process each pixel.
    for i in range(h):
        for j in range(w):
            img_filter[i, j] = inner_prod(img_pad[i:i+edge, j:j+edge], kernel)
    return img_filter

# Median filter.
def median_filter(img, s):
    h, w = img.shape
    img_filter = np.zeros_like(img)
    for i in range(h):
        for j in range(w):
            # determine the bound of sub-image covered by the filter.
            a, b, l, r = max(0, i-s), min(h, i+s+1), max(0, j-s), min(w, j+s+1)
            img_filter[i, j] = np.median(img[a:b, l:r].reshape((b-a) * (r-l)))
    return img_filter

# Normalize the filtered image onto [0, nbins-1].
def normalize(img, nbins=256):
    # In case some pixel values of processed image overflow or underflow on [0,
    nbins - 1],
    # normalize it to the interval [0, nbins - 1].
    lb = np.min(img)
    ub = np.max(img)
    img_normalized = (img - lb) / (ub - lb) * (nbins - 1)
    return img_normalized.astype(int)
```

1.1 Smoothing

In this part, we implement some smoothing filters on several images.

Algorithm For linear filter, mean kernel and gaussian kernel, which can be seen as a weighted mean, are often used in smoothing an image. Of size $(2a+1) \times (2b+1)$, they can be mathematically expressed as

$$w_{\text{mean}}(s,t) = \frac{1}{(2a+1)(2b+1)}, \quad s \in [-a,a],\ t \in [-b,b];$$

$$w_{\text{gaussian}}(s,t) = K \exp\left\{-\frac{s^2+t^2}{2\sigma^2}\right\}, \quad s \in [-a,a],\ t \in [-b,b],$$

where σ^2 is the variance and K is a normalization factor; note thatthe origin $(0,0)$ represents the center of kernel.

Code Here we implemented Gaussian filter for smoothing, and we also implemented median filter in our experiments. The code is shown below.

```
# Smooth the image by gaussian filter.
def smooth_gaussian(img, sigma_sq, s):
    # Compute the gaussian filter.
    gaussian_kernel = np.zeros((2 * s + 1, 2 * s + 1))
    for i in range(2 * s + 1):
        for j in range(2 * s + 1):
            gaussian_kernel[i, j] = math.exp(-((i-s)**2+(j-s)**2)/(2*sigma_sq))
    gaussian_kernel = gaussian_kernel / gaussian_kernel.sum()  # Normalization.
    img_smoothed_gaussian = lin_filter(img.astype(float), gaussian_kernel)
    return img_smoothed_gaussian.astype(int)

# Smooth the image by median filter.
def smooth_median(img, s):
    return median_filter(img, s).astype(int)
```

Result Following shows some result.

(a) suomi.jpg (Original)

(b) Filtered by 7×7 Gaussian kernel, $\sigma^2 = 1$

(c) Filtered by 7×7 Gaussian kernel, $\sigma^2 = 9$

(d) Filtered by 7×7 Gaussian kernel, $\sigma^2 = 81$

From the result above, it can be seen that a gaussian filter can smooth the image and get a blur effect. If we increase the variance σ^2, the processed image will be more blurred (we tested it on $\sigma^2 = 1, 9, 81$). This can be interpreted as that increasing the variance will decrease the weight of center pixel in blurred image, thus achieve a more obvious blur effect.

suomi.jpg (Original)

(e) Filtered by 5×5 Median kernel

(f) Filtered by 15×15 Median kernel

To compare with gaussian filters, we also tested the median filter. It shows that the median filter will achieve a significantly different effect – while gaussian filter blurs the original image, median filter smooth the edges and angles of some subjects in the image. In our example, the angles of letters, arrows, triangle and octagon become round, and the boundary between bricks vanished. Also, a larger kernel yields a more smoothed image, and reduces more the sharp characteristics in the original image. (We test this by a 5×5 kernel and a 15×15.)

1.2 Sharpening

In this part, we implement some sharpening algorithms obased on spatial filter.

Algorithm In image processing, the derivative of a pixel indicates the shifting rate of color or grayscale at its place on a certain direction. In implement, it can be estimated as

$$\frac{\partial f}{\partial x} = f(x+1,y) - f(x,y), \quad \frac{\partial f}{\partial y} = f(x,y+1) - f(x,y);$$

Also we can derive the estimate of 2nd-order derivative

$$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y), \quad \frac{\partial^2 f}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y);$$

Consider the Laplacian operator,

$$\Delta f = (\nabla \cdot \nabla) f = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) f = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y);$$

Thus we derive the Laplacian kernel:

$$w_{\text{laplacian}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

In implement we prefer another form of Laplcian kernel

$$\tilde{w}_{\text{laplacian}} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

which performs better in extracting the sharpeness of an image. Intuitively, the kernel measures the difference between the center pixel and its surrounding pixels.
To sharpen an image, we might want to enhance the difference, measured by the derivative, between a pixel and its surrounding pixels, which can be realized by adding the laplacian filter onto the original image:

$$f_{\text{sharpen}} = f + w\nabla^2 f,$$

where w should be a negative constant, whose absolute value indicates the weight of laplacian filter.

Another method called **Highboost** is also used in image sharpening. Suppose we get a blurred image $\bar{f}$ by mean filter, define

$$g_{\text{mask}}(x,y) = f(x,y) - \bar{f}(x,y),$$

then add the mask back to the original image with a weighted portion k (In highboost filtering $k > 1$):

$$f_{\text{sharpen}} = f + kg_{\text{mask}}.$$

Code The code is shown below.

```
def sharpen_laplacian(img, w, nbins=256):
    # return a sharpened image f + w * D2f.
    laplacian_kernel = np.array([[1.0, 1.0, 1.0],
```

```
                                    [1.0, -8.0, 1.0],
                                    [1.0, 1.0, 1.0]]
                                   )
    laplacian_filter = lin_filter(img.astype(float), laplacian_kernel)
    img_sharpened_laplacian = img.astype(float)
    # The boundary of filtered image tends to have small values.
    # We remain the boundary intact to avoid aberrant bright boundary.
    img_sharpened_laplacian[1:-1, 1:-1] -= w * laplacian_filter[1:-1, 1:-1]
    img_sharpened_laplacian[img_sharpened_laplacian < 0] = 0
    img_sharpened_laplacian[img_sharpened_laplacian > nbins - 1] = nbins - 1
    return normalize(laplacian_filter), img_sharpened_laplacian.astype(int)

def highboost(img, k, nbins=256):
    # return a sharpened image f + k * (f - f_bar).
    img_mean = smooth_gaussian(img, sigma_sq=9.0, s=3)
    mask = (img - img_mean)
    img_hboosted = img + k * mask
    img_hboosted[img_hboosted < 0] = 0
    img_hboosted[img_hboosted > nbins - 1] = nbins - 1
    return normalize(mask), img_hboosted.astype(int)
```

Note that after adding the filter or mask onto the original image may yields a matrix with some extremely high or low pixel values, which is out of range $[0, 255]$, and if we simply use the linear normalization, it may compress most pixel values into a very narrow range. (For instance, most pixels in the image are in the range $[0, 255]$, but one pixel has an extremely high 511, then after linear normalization the pixels in range $[0, 255]$ would be compressed to $[0, 127]$, which causes a significant loss.) Thus we don't modify the pixels within $[0, 255]$, and set the underflowing pixels to 0, and the overflowing ones 255. The outputs verified the practicability of our normalization method.

Result Following shows some of our result.

(g) suomi.jpg (Original)

(h) Laplacian filter of suomi.jpg

(i) Filtered by Laplacian kernel, $w = 0.5$

(j) Filtered by Laplacian kernel, $w = 1.0$

Laplacian filter The upper-right image is the laplacian filter of the original image, and we can see that it extracts the edges and some sharp characteristics, where the pixel values shift steeply (the derivative is large). After adding the filter onto the original image, we get a sharpened image, in which the edges of subjects are obviously enhanced. In our examples, the edges of letters and shapes as well as the boundaries between bricks. A larger weight of addition yields a stronger enhancement, say the processed image is more sharpened. We test our code on $w = 0.5$ and 1 to show this.

(k) suomi.jpg (Original)

(l) Mask under a gaussian blurring, 7×7, $\sigma^2 = 9$

(m) Highboost with $k = 1.0$

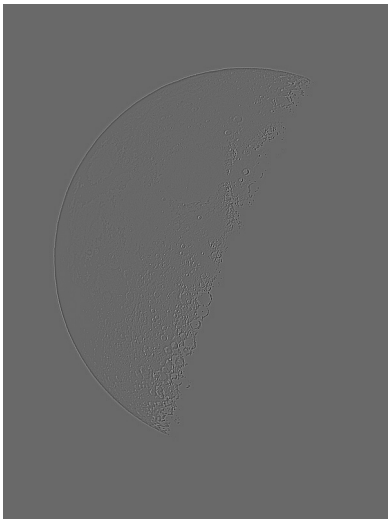
(n) Highboost with $k = 4.0$

Highboost Here we implement highboost based on a 7×7 gaussian filter with $\sigma^2 = 9$. The mask acquired by this filter are shown in upper-right. Like the laplacian filter, it also extracts the sharp charasteristics in the original image. Because the filter (7×7) we implement are larger than the laplacian filter (3×3), one pixel value is influenced by a larger proximal region. Thus the light-and-shade-contrast we see in the mask extends wider.

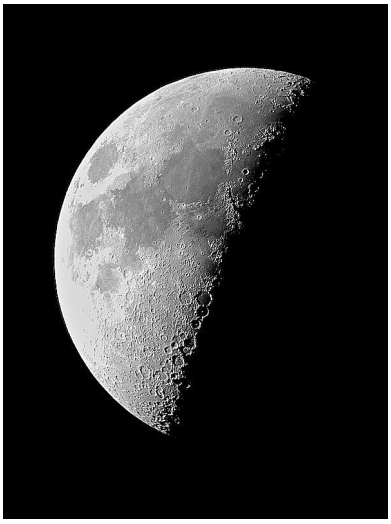
Also, by adding the mask onto the original image can we get a sharpened image, and a larger adding weight yields a more sharpened image. We show this by setting $k = 1.0$ and 4.0. This method enhance the edges of subjects with a light-and-shade-contrast around them.

1.3 Additional Result

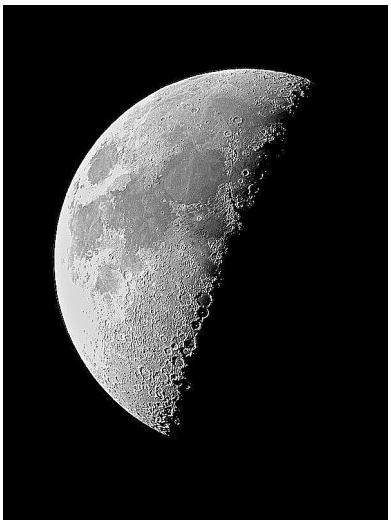
(o) moon.jpg (Original)



(p) Laplacian filter

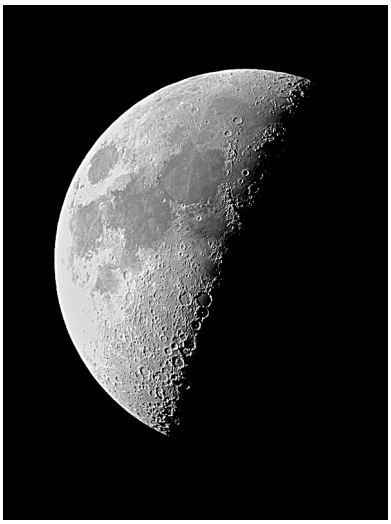


(q) Sharpened $w = 0.5$



(r) Sharpened $w = 1.0$

(s) Mask with 7×7 gaussian filter, $\sigma^2 = 9.0$



(t) Highboost with $k = 2.0$

2 Fourier Transform

2.0 Lemmas

Notation In our subsequent deduction, we use i to denote the imaginary unit in $\mathbb{C}$.

Lemma 1.

$$\int_{-\infty}^{+\infty} \exp\{-i2\pi\mu t\}\mathrm{d}t = \delta(\mu) := \begin{cases} +\infty, \text{ if } \mu = 0, \\ 0, \text{ otherwise;} \end{cases}$$

Pf. If $\mu = 0$,

$$\int_{-\infty}^{+\infty} \exp\{-i2\pi\mu t\}\mathrm{d}t = \int_{-\infty}^{+\infty} \mathrm{d}t = +\infty;$$

otherwise,

$$\begin{aligned} \int_{-\infty}^{+\infty} \exp\{-i2\pi\mu t\}\mathrm{d}t &= \int_{-\infty}^{+\infty} \cos(2\pi\mu t) + i\sin(2\pi\mu t)\,\mathrm{d}t \\ &= \frac{\sin(2\pi\mu t)}{2\pi\mu} - \frac{i\cos(2\pi\mu t)}{2\pi\mu}\Big|_{t=-\infty}^{+\infty} = 0; \end{aligned}$$

Therefore $\int_{-\infty}^{+\infty} \exp\{-i2\pi\mu t\}\mathrm{d}t = \delta(\mu)$. □

Lemma 2. M is a positive integer, and $\forall\, k \in \mathbb{Z}$, if $z \neq 0$, then

$$\sum_{m=0}^{M-1} \exp\left(i\frac{2mk\pi}{M}\right) = 0;$$

Pf. Suppose the unit root $\omega = \exp\left(i\frac{2k\pi}{M}\right)$, then it implies $1 - \omega^M = (1-\omega)\left(1 + \sum_{m=1}^{M-1} \omega^m\right) = 0;$

since $k \neq 0,\ 1 - \omega \neq 0$, and $1 + \sum_{m=1}^{M-1} \omega^m = 0$, which can be rewritten as

$$\sum_{m=0}^{M-1} \exp\left(i\frac{2mk\pi}{M}\right) = 0.$$

□

2.1 FT of Impulse Train

An impulse train $s_{\Delta T}$ of period ΔT is defined as:

$$s_{\Delta T}(t) = \sum_{n=-\infty}^{+\infty} \delta(t - n\Delta T),$$

where δ is a unit impulse:

$$\delta(t) = \begin{cases} +\infty, \text{ if } t = 0, \\ 0, \text{ otherwise;} \end{cases}$$

Obviously $s_{\Delta T}$ is a periodic function, indicating that it can be expressed as Fourier series:

$$s_{\Delta T}(t) = \sum_{n=-\infty}^{+\infty} c_n \exp\left\{\frac{2i\pi nt}{\Delta T}\right\},$$

where

$$
\begin{aligned}
c_n &= \frac{1}{\Delta T}\int_{-\Delta T/2}^{\Delta T/2} s_{\Delta T}(t)\exp\left\{-\frac{2i\pi nt}{\Delta T}\right\}\mathrm{d}t \\
&= \frac{1}{\Delta T}\sum_{m=-\infty}^{+\infty}\int_{-\Delta T/2}^{+\Delta T/2}\delta(t-m\Delta T)\exp\left\{-\frac{2i\pi nt}{\Delta T}\right\}\mathrm{d}t \\
&= \frac{1}{\Delta T}\int_{-\Delta T/2}^{+\Delta T/2}\delta(t)\exp\left\{-\frac{2i\pi nt}{\Delta T}\right\}\mathrm{d}t \\
&= \frac{1}{\Delta T}\exp(0) = \frac{1}{\Delta T};
\end{aligned}
$$

Then it implies

$$
\begin{aligned}
\mathcal{J}\{s_{\Delta T}\} &= \int_{-\infty}^{+\infty} s_{\Delta T}(t)\exp\{-2i\pi\mu t\}\,\mathrm{d}t \\
&= \sum_{n=-\infty}^{+\infty}\int_{-\infty}^{+\infty} c_n\exp\left\{2i\pi\left(\frac{n}{\Delta T}-\mu\right)t\right\}\mathrm{d}t \\
&= \frac{1}{\Delta T}\sum_{n=-\infty}^{+\infty}\int_{-\infty}^{+\infty}\exp\left\{2i\pi\left(\frac{n}{\Delta T}-\mu\right)t\right\}\mathrm{d}t \\
&= \frac{1}{\Delta T}\sum_{n=-\infty}^{+\infty}\delta\left(\mu-\frac{n}{\Delta T}\right);
\end{aligned}
$$

The last equality is implied by **Lemma1**;

Therefore the Fourier transform of an impulse train $s_{\Delta T}(t)$ is still an impulse train $S(\mu)=\frac{1}{\Delta T}\sum_{n=-\infty}^{+\infty}\delta\left(\mu-\frac{n}{\Delta T}\right)$ with period $\frac{1}{\Delta T}$.

2.2 Conjugate Symmetry of DFT

DFT: on 1D real signal $f(x)$,

$$
\begin{cases}
F_m = \sum_{n=0}^{M-1} f_n\exp(-2i\pi mn/M), \quad m=0,1,\cdots,M-1, \\
\\
f_n = \frac{1}{M}\sum_{m=0}^{M-1} F_m\exp(2i\pi mn/M), \quad n=0,1,\cdots,M-1;
\end{cases}
\qquad (*)
$$

Pf. For $k=0,1,\cdots,M-1$,

$$
\begin{aligned}
\frac{1}{M}\sum_{m=0}^{M-1} F_m\exp(2i\pi mk/M) &= \frac{1}{M}\sum_{m=0}^{M-1}\sum_{n=0}^{M-1} f_n\exp(-2i\pi mn/M)\cdot\exp(2i\pi mk/M) \\
&= \frac{1}{M}\sum_{n=0}^{M-1} f_n\sum_{m=0}^{M-1}\exp\{2i\pi m(k-n)/M\} = \frac{1}{M}\sum_{n=0}^{M-1} f_n\sum_{m=0}^{M-1}\mathbb{1}\{n=k\} = f_k,
\end{aligned}
$$

the third equality is implied by **Lemma2**;

Similarly, for $l=0,1,\cdots,M-1$,

$$
\sum_{n=0}^{M-1} f_n\exp(-2i\pi ln/M) = \frac{1}{M}\sum_{n=0}^{M-1}\sum_{m=0}^{M-1} F_m\exp(2i\pi mn/M)\cdot\exp(-2i\pi ln/M)
$$

$$= \frac{1}{M} \sum_{m=0}^{M-1} F_m \sum_{n=0}^{M-1} \exp\{2i\pi m(m-l)/M\} = \frac{1}{M} \sum_{m=0}^{M-1} F_m \sum_{n=0}^{M-1} \mathbb{1}\{m=l\} = F_l,$$

the third equality is implied by **Lemma2**;

Thus we can derive either formola of $(*)$ from the other, say DFT of $f(x)$ has conjugate symmetry. □

2.3 Convolution in 2D-DFT

Consider the DFT in 2D plane:

$$\mathcal{J}\{f\} = F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp\left\{-2i\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\right\},$$

$$\mathcal{J}\{g\} = G(u,v) = \sum_{w=0}^{M-1} \sum_{z=0}^{N-1} g(w,z) \exp\left\{-2i\pi \left(\frac{uw}{M} + \frac{vz}{N}\right)\right\};$$

I. Convolution of DFT

$$(F * G)(u,v) = \sum_{s=0}^{M-1} \sum_{t=0}^{N-1} F(s,t)\, G(u-s, v-t)$$

$$= \sum_{s=0}^{M-1} \sum_{t=0}^{N-1} \left[\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \exp\left\{-2i\pi \left(\frac{sx}{M} + \frac{ty}{N}\right)\right\}\right] \left[\sum_{w=0}^{M-1} \sum_{z=0}^{N-1} g(w,z) \exp\left\{-2i\pi \left(\frac{(u-s)w}{M} + \frac{(v-t)z}{N}\right)\right\}\right]$$

$$= \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \sum_{w=0}^{M-1} \sum_{z=0}^{N-1} \left[f(x,y)g(w,z) \exp\left\{-2i\pi \left(\frac{uw}{M} + \frac{vz}{N}\right)\right\} \sum_{s=0}^{M-1} \sum_{t=0}^{N-1} \exp\left\{-2i\pi \left(\frac{s(x-w)}{M} + \frac{t(y-z)}{N}\right)\right\}\right];$$

Consider that

$$\sum_{s=0}^{M-1} \sum_{t=0}^{N-1} \exp\left\{-2i\pi \left(\frac{s(x-w)}{M} + \frac{t(y-z)}{N}\right)\right\}$$

$$= \sum_{s=0}^{M-1} \exp\left\{-2i\pi \frac{s(x-w)}{M}\right\} \sum_{y=0}^{N-1} \exp\left\{-2i\pi \frac{t(y-z)}{N}\right\}$$

$$= \begin{cases} MN, \text{ if } (x,y) = (w,z), \\ 0, \text{ otherwise;} \end{cases} \quad \text{(from } \textbf{Lemma 2}\text{)}$$

So $(F * G)(u,v) = MN \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)g(x,y) \exp\left\{-2i\pi \left(\frac{ux}{M} + \frac{vy}{N}\right)\right\} = MN \cdot \mathcal{J}\{f \cdot g\}(u,v),$

Say $\mathcal{J}\{f \cdot g\} = \frac{1}{MN}(F * G)$.

II. DFT of Convolution

$$(f * g)(x,y) = \sum_{w=0}^{M-1} \sum_{z=0}^{N-1} f(w,z)g(x-w, y-z),$$

$$\mathcal{J}\{f*g\}(u,v) = \sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\sum_{w=0}^{M-1}\sum_{z=0}^{N-1} f(w,z)g(x-w,y-z)\exp\left\{-2i\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right\}$$
$$= \sum_{w=0}^{M-1}\sum_{z=0}^{N-1} f(w,z)\exp\left\{-2i\pi\left(\frac{uw}{M}+\frac{vz}{N}\right)\right\}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} g(x-w,y-z)\exp\left\{-2i\pi\left(\frac{u(x-w)}{M}+\frac{v(y-z)}{N}\right)\right\}$$
$$= \sum_{w=0}^{M-1}\sum_{z=0}^{N-1} f(w,z)\exp\left\{-2i\pi\left(\frac{uw}{M}+\frac{vz}{N}\right)\right\}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1} g(x,y)\exp\left\{-2i\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)\right\}$$
$$= F(u,v)\cdot G(u,v),$$

where the third equality is acquired by the periodicity of g, with the period (M,N) on two components; so $\mathcal{J}\{f*g\} = F*G$.

Thus we derived the convolution formula in 2D-DFT:

$$\begin{cases} f\cdot g \overset{\mathcal{J}}{\mapsto} \dfrac{1}{MN}(F*G), \\ f*g \overset{\mathcal{J}}{\mapsto} F\cdot G. \end{cases}$$

3 Appendix

3.1 Libraries

We use the `numpy` library to support the large-scale computation of matrices which store the grey level information of images.

We use `opencv` library (`cv2`) to read the input images and save the output images.

We use `matplotlib` to show images before and after processing.

We also use `argparse` to read in the arguments for processing, which can be specified in the command line.

3.2 IDE

All our experiments are done on PyCharm Community Edition 2022.1.2.