



---

# 计算机视觉与模式识别

---

第一次实验报告



王易博

人工智能 82  
2186113742

目录

- 一、实验内容.....2
- 二、实验原理.....2
  - 2.1 图像的高斯滤波与 Padding.....2
  - 2.2 高斯核与高斯核卷积实验 .....3
  - 2.3 图像锐化滤波器.....3
  - 2.4 双边滤波.....4
- 三、实验结果与分析.....4
  - 3.1 图像的高斯滤波与 Padding.....4
  - 3.2 高斯核与高斯核卷积实验 .....8
  - 3.3 图像锐化.....9
  - 3.4 双边滤波..... 10
  - 3.5 图像的傅里叶变换..... 11
  - 3.6 探索：频率水印..... 12
- 四、结论与讨论 ..... 13
  - 4.1 图像的高斯滤波与 Padding..... 13
  - 4.2 高斯核与高斯核的卷积实验 ..... 13
  - 4.3 图像锐化..... 13
  - 4.4 双边滤波..... 13
  - 4.5 图像的傅里叶变换..... 13
- 五、代码..... 14
  - 5.1 图像的高斯滤波与 Padding..... 14
  - 5.2 高斯核与高斯核的卷积实验 ..... 16
  - 5.3 图像锐化..... 16
  - 5.4 双边滤波..... 16
  - 5.5 图像的傅里叶变换..... 17
  - 5.6 探索：频率水印..... 17
  - 5.7 作业所有代码及图片地址 ..... 17

# 一、实验内容

- 1、2D 高斯模板的设计（给定方差生成滤波核）；图像的高斯滤波；在高斯滤波中不同边界的处理方法实验
- 2、高斯核与高斯核的卷积实验；利用两个相同方差的一维行列高斯核卷积生成 2D 高斯核，利用一维行列高斯对图像进行滤波；不同方差高斯核之差对图像进行滤波；
- 3、利用两个高斯核设计图像锐化滤波器核；
- 4、图像的双边滤波实验；
- 5、图像的 Fourier 变换，显示幅度谱与相位谱；利用高斯滤波器进行图像的频域滤波；

# 二、实验原理

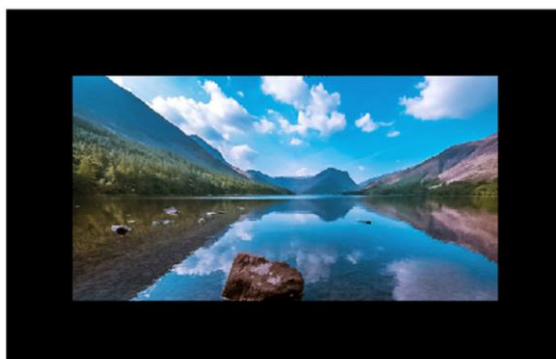
## 2.1 图像的高斯滤波与 Padding

高斯核设计原理：设二维高斯分布为两个独立的零均值，方差相同的一维高斯分布组成。以滤波核中心为原点，像素之间距离为 1，将高斯核每个点转成坐标形式，带入高斯函数中，算出具体值后，进行归一化处理，使得加和为 1。

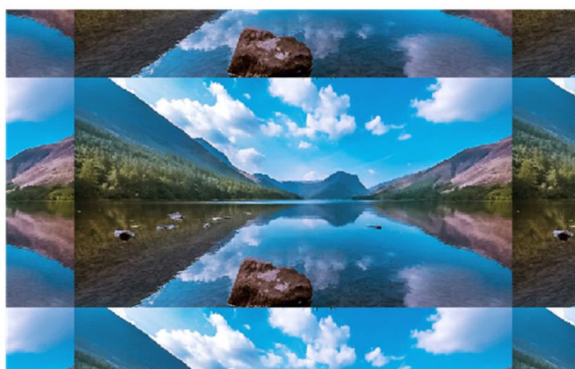
图像的高斯滤波：将高斯核与图像进行卷积

Padding：具体使用 Python 的切片操作实现

*Clip filter (black). — Zero Padding*



*Wrap around*



*Copy edge*



*Reflect across edge*



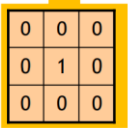
## 2.2 高斯核与高斯核卷积实验

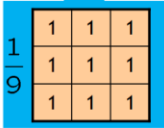
与生成二维高斯核类似，将一维卷积核转换成坐标形式代入一维高斯分布并归一化，生成一维高斯核。将上述一维高斯核转置，生成第二个一维高斯核。利用卷积的交换律，使用两个一维高斯核依次对图像进行卷积。生成两个不同方差的二维卷积核进行相减，得到的新卷积核即可完成不同方差卷积核之差对图像滤波。

## 2.3 图像锐化滤波器

锐化原理为给图像加上 $\alpha$ 倍的图像与高斯滤波后的图像之差，具体操作如下图所示：

$$\begin{aligned}
 f_{sharp} &= f + \alpha(f - f_{blur}) \\
 &= (1 + \alpha)f - \alpha f_{blur} \\
 &= (1 + \alpha)(w * f) - \alpha(v * f)
 \end{aligned}$$





$$= ((1 + \alpha)w - \alpha v) * f$$

## 2.4 双边滤波

由于高斯滤波会破坏边界信息，所以在卷积操作时，加入图像信息作为权重，即将中心点与进行卷积计算点的差值作为高斯函数的自变量，求得高斯函数值作为权重吗，即下图所示：

$$I_{\mathbf{p}}^{\text{bf}} = \underbrace{\frac{1}{W_{\mathbf{p}}^{\text{bf}}}}_{\text{normalization}} \sum_{\mathbf{q} \in \mathcal{S}} \underbrace{G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)}_{\text{space}} \underbrace{G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)}_{\text{range}} I_{\mathbf{q}}$$

## 2.5 图像的傅里叶变换

将图片像素值作为信号幅值，进行傅里叶变换。由于图片是二维的离散信号，所以使用 (M, N) 点的二维快速傅里叶变换。将频谱加上 Mask，使其只保留高频成分，完成滤波。

# 三、实验结果与分析

## 3.1 图像的高斯滤波与 Padding

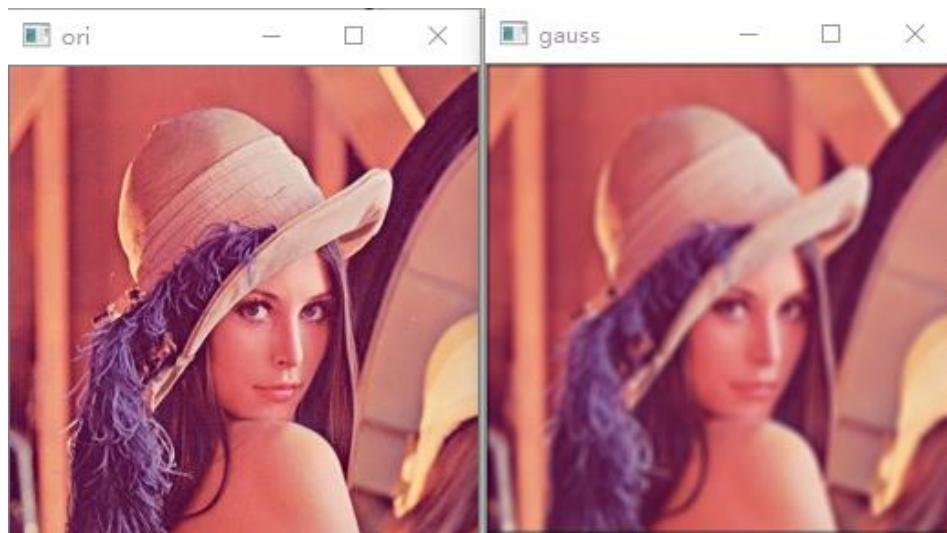
设置 size = 5, sigma = 3, 生成卷积核如下所示：

```

[[0.0317564  0.03751576 0.03965895 0.03751576 0.0317564 ]
 [0.03751576 0.04431963 0.04685151 0.04431963 0.03751576]
 [0.03965895 0.04685151 0.04952803 0.04685151 0.03965895]
 [0.03751576 0.04431963 0.04685151 0.04431963 0.03751576]
 [0.0317564  0.03751576 0.03965895 0.03751576 0.0317564 ]]

```

使用该核对图像滤波，卷积模式为 same，padding 为 zero padding：

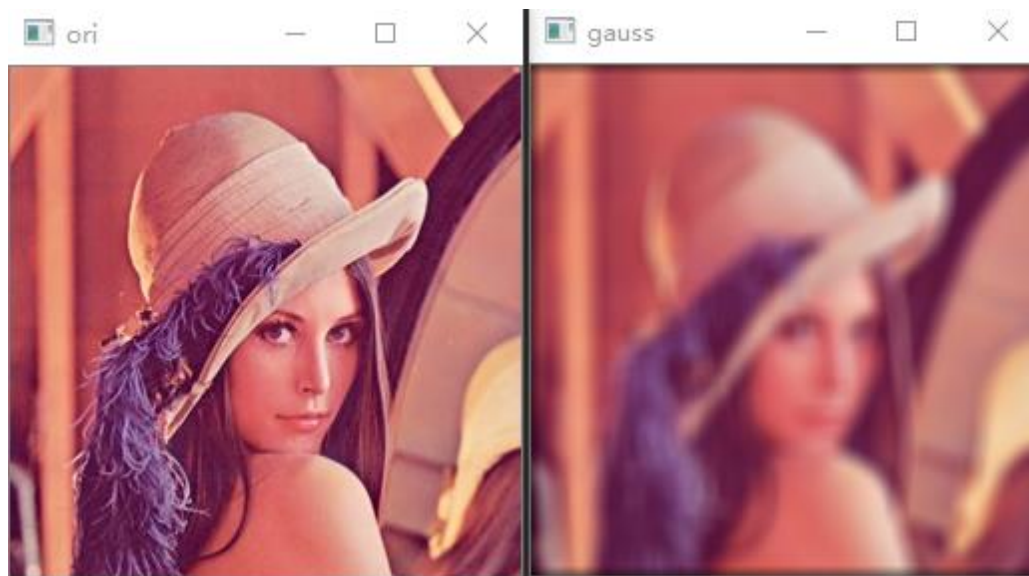


设置 size = 11, sigma = 10, 生成卷积核如下所示

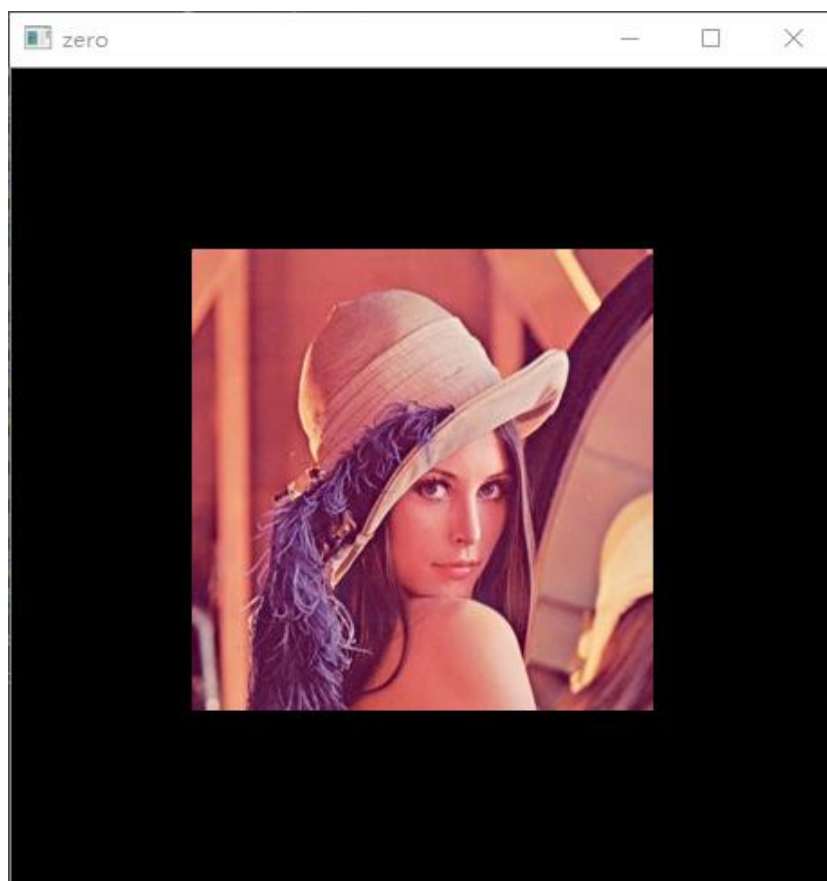
```
[0.00871624 0.0090465 0.009312 0.00950639 0.00962498 0.00966483
 0.00962498 0.00950639 0.009312 0.0090465 ]
[0.0090465 0.00938927 0.00966483 0.00986659 0.00998967 0.01003103
 0.00998967 0.00986659 0.00966483 0.00938927]
[0.009312 0.00966483 0.00994847 0.01015616 0.01028285 0.01032542
 0.01028285 0.01015616 0.00994847 0.00966483]
[0.00950639 0.00986659 0.01015616 0.01036818 0.01049751 0.01054098
 0.01049751 0.01036818 0.01015616 0.00986659]
[0.00962498 0.00998967 0.01028285 0.01049751 0.01062846 0.01067247
 0.01062846 0.01049751 0.01028285 0.00998967]
[0.00966483 0.01003103 0.01032542 0.01054098 0.01067247 0.01071666
 0.01067247 0.01054098 0.01032542 0.01003103]
[0.00962498 0.00998967 0.01028285 0.01049751 0.01062846 0.01067247
 0.01062846 0.01049751 0.01028285 0.00998967]
[0.00950639 0.00986659 0.01015616 0.01036818 0.01049751 0.01054098
 0.01049751 0.01036818 0.01015616 0.00986659]
[0.009312 0.00966483 0.00994847 0.01015616 0.01028285 0.01032542
 0.01028285 0.01015616 0.00994847 0.00966483]
[0.0090465 0.00938927 0.00966483 0.00986659 0.00998967 0.01003103
 0.00998967 0.00986659 0.00966483 0.00938927]]
```

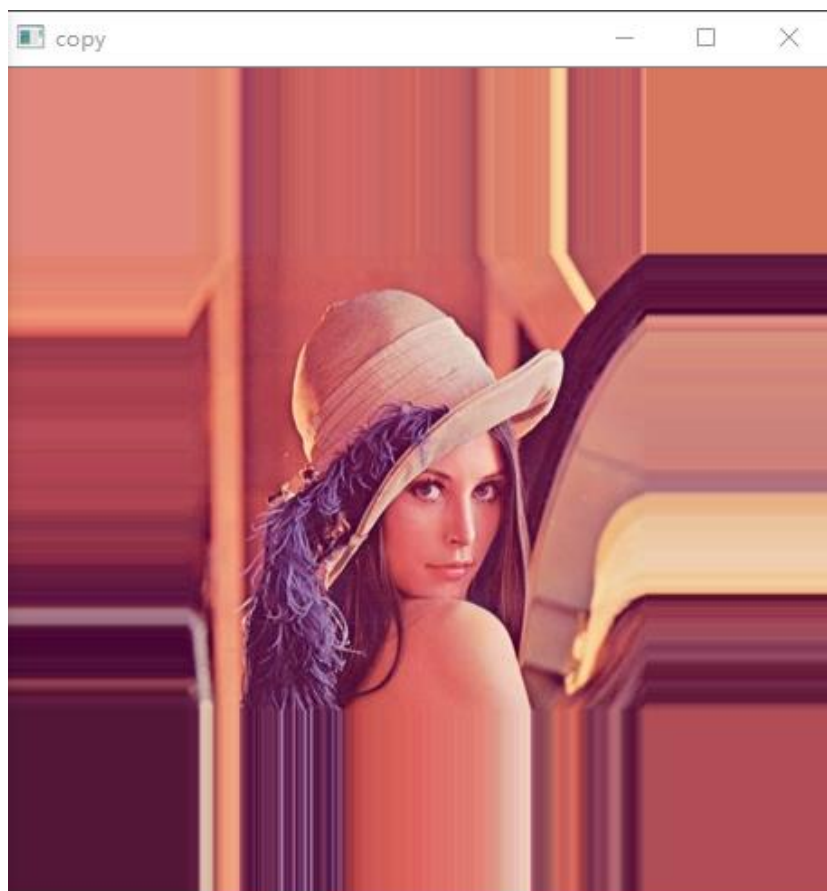
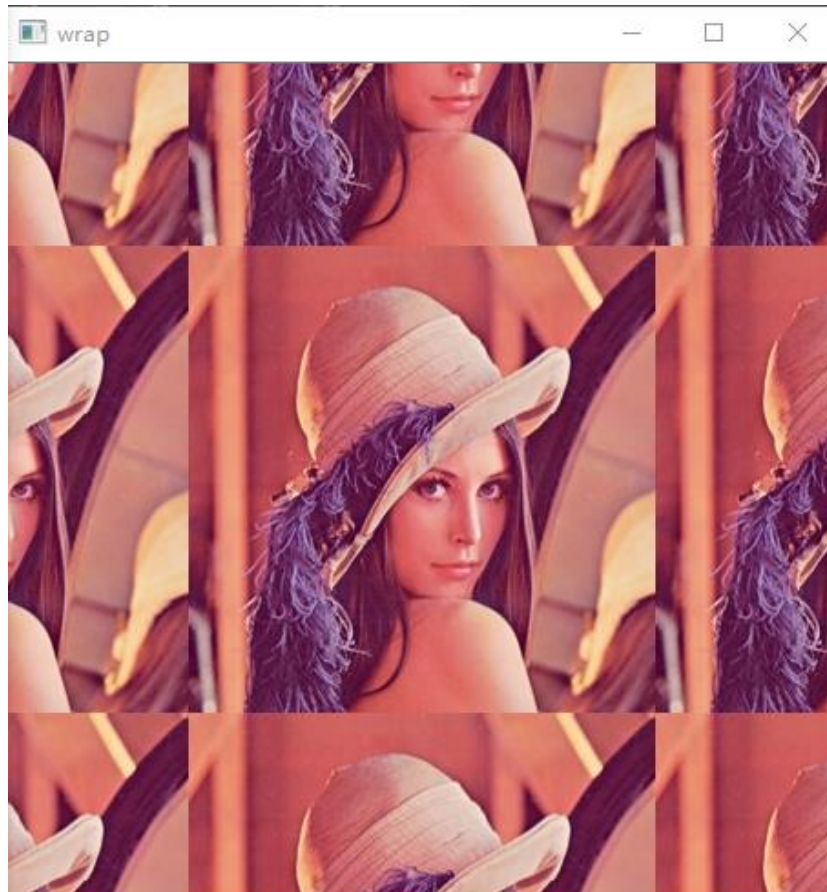
使用该核对图像进行滤波，卷积模式为 same, padding 为 zero padding:



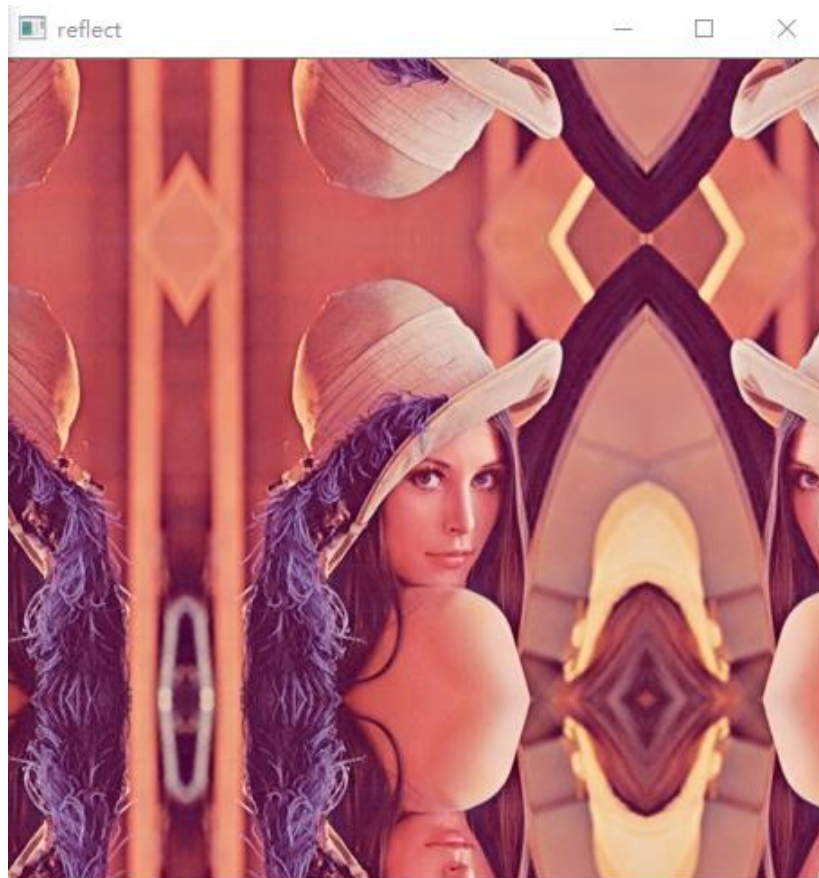


可以看到，由于使用的是 zero padding，卷积核和 sigma 的越大，边缘越黑  
Padding 结果如下：









## 3.2 高斯核与高斯核卷积实验

生成  $\text{size} = 5$ ,  $\text{sigma} = 3$  的一维卷积核如下所示

```
[[0.17820326]
 [0.21052227]
 [0.22254894]
 [0.21052227]
 [0.17820326]]
```

将其和它的转置进行 full 卷积，结果如下：

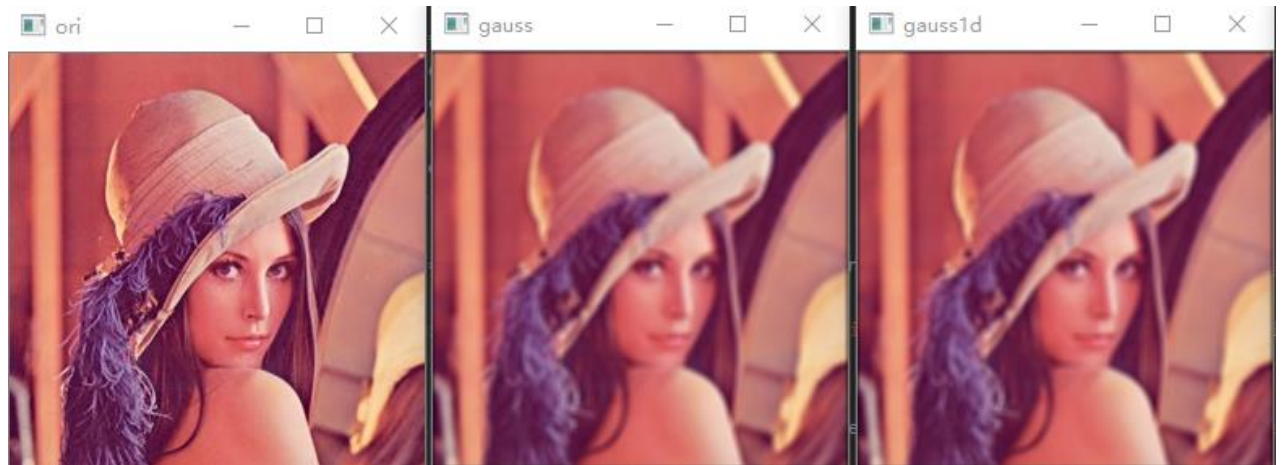
```
[[0.0317564  0.03751576 0.03965895 0.03751576 0.0317564 ]
 [0.03751576 0.04431963 0.04685151 0.04431963 0.03751576]
 [0.03965895 0.04685151 0.04952803 0.04685151 0.03965895]
 [0.03751576 0.04431963 0.04685151 0.04431963 0.03751576]
 [0.0317564  0.03751576 0.03965895 0.03751576 0.0317564 ]]
```

与 3.1 中生成的二维高斯卷积一致

将这两个卷积核分别对图像进行滤波：

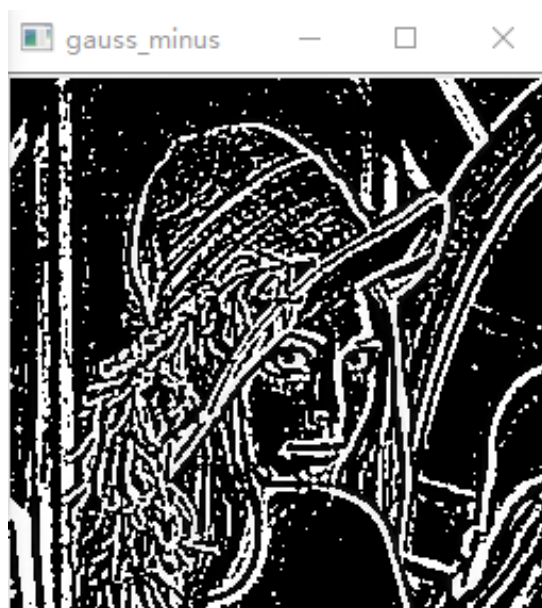
```
for i in range(3):
    img_gauss1d[:, :, i] = conv(image=img[:, :, i], kernel=sep_filter_0, mode_conv=mode_conv, mode_padding=mode_padding)
for i in range(3):
    img_gauss1d[:, :, i] = conv(image=img_gauss1d[:, :, i], kernel=sep_filter_1, mode_conv=mode_conv, mode_padding=mode_padding)
```

与二维卷积、原图对比如下：



可见，与二维卷积核的滤波结果没有区别

设置 size=5，sigma 分别为 3 和 1 的两个卷积核相减得到新卷积核，对图像进行滤波，结果如下：



可见，使用高斯核之差进行滤波，起到的边缘提取的作用

### 3.3 图像锐化

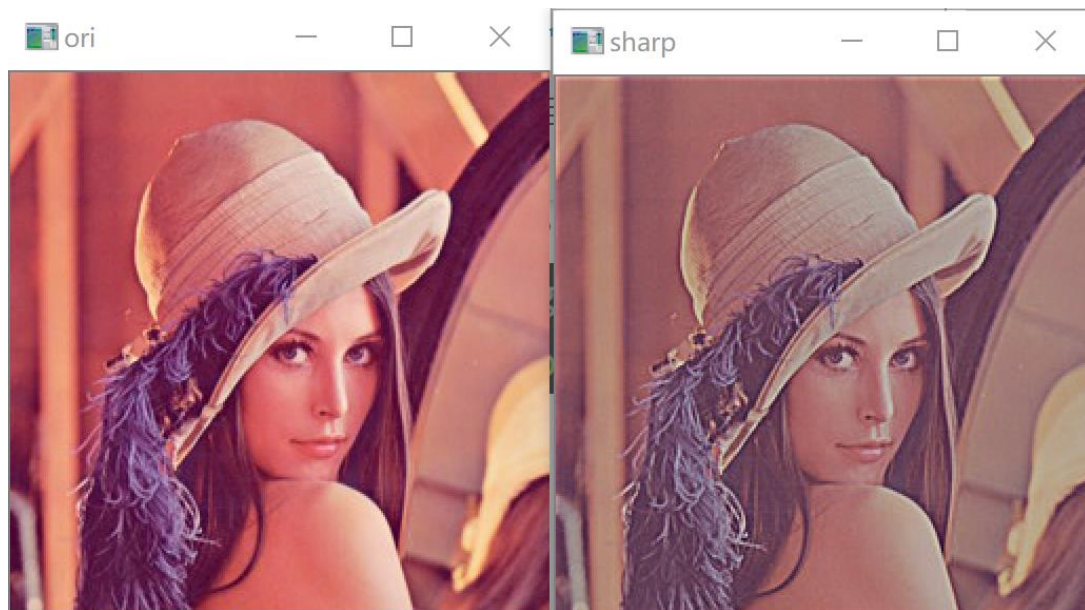
设置 size = 5，sigma = 3 的高斯核，设置 $\alpha=1$ ，形成新的锐化卷积核：

```
alpha = 1
w = np.zeros((size, size))
w[size // 2, size // 2] = 1
kernel_sharp = (1+alpha)*w - alpha*kernel
for i in range(3):
    img_sharp[:, :, i] = conv(image=img[:, :, i], kernel=kernel_sharp, mode_conv=mode_conv, mode_padding=mode_padding)
```

由于相加后会产生大于 255 和小于 0 的值，所以将图像标准化后输出：

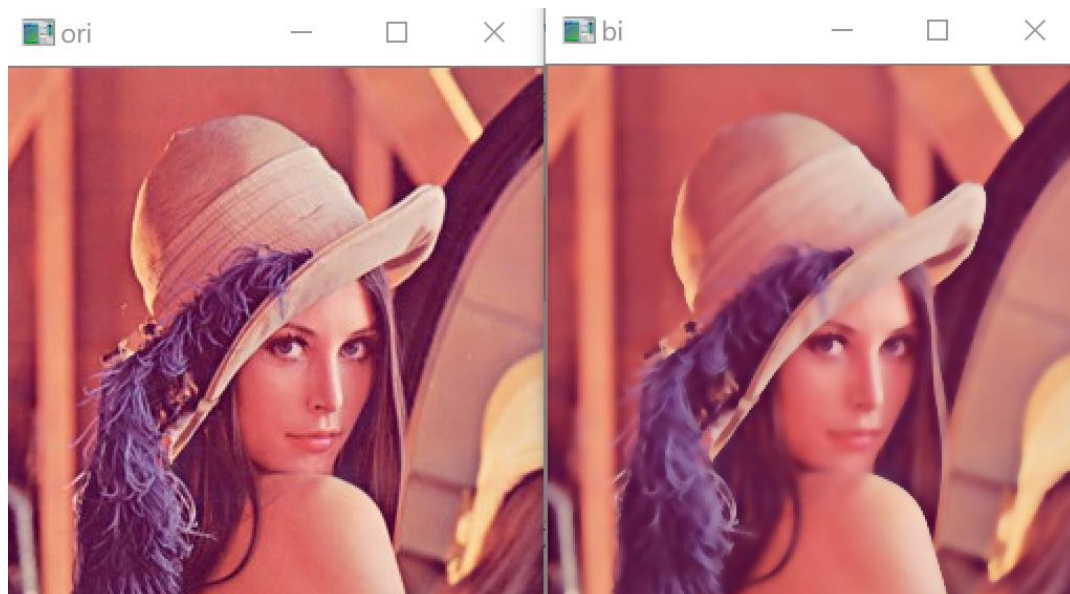
```
cv2.normalize(img_sharp, img_sharp, 0, 255, cv2.NORM_MINMAX)
cv2.imshow("sharp", img_sharp.astype("uint8"))
```

锐化结果如下图所示：



### 3.4 双边滤波

设置 space 权重的 sigma 为 3，range 权重的 sigma 为 50，双边滤波结果如下：



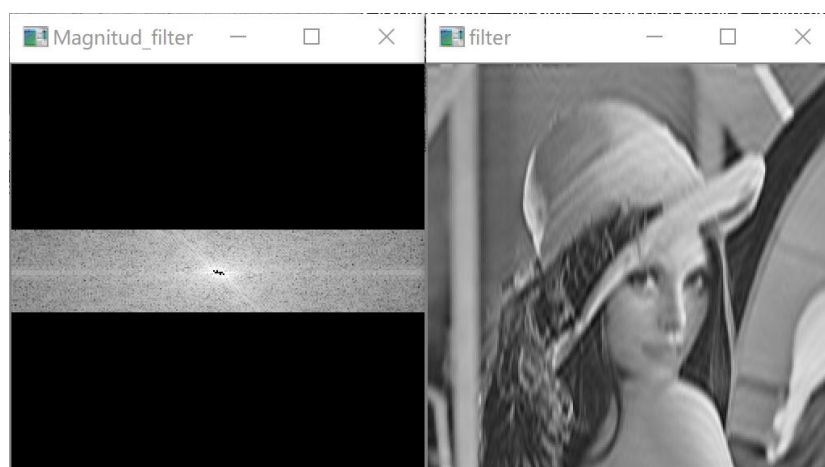


### 3.5 图像的傅里叶变换

由于自己手写傅里叶速度过慢，遂使用 numpy 的 2 维 fft 函数，变换后，为了方便观察，使用 shift 函数将低频部分移动至中心。但幅频特性输出后与实际不符，输出图像像素发现 fft 后的值大部分远超 255，查阅资料得知，需要求幅频的分贝数进行输出，即给幅频加  $20\log$  结果如下图所示：

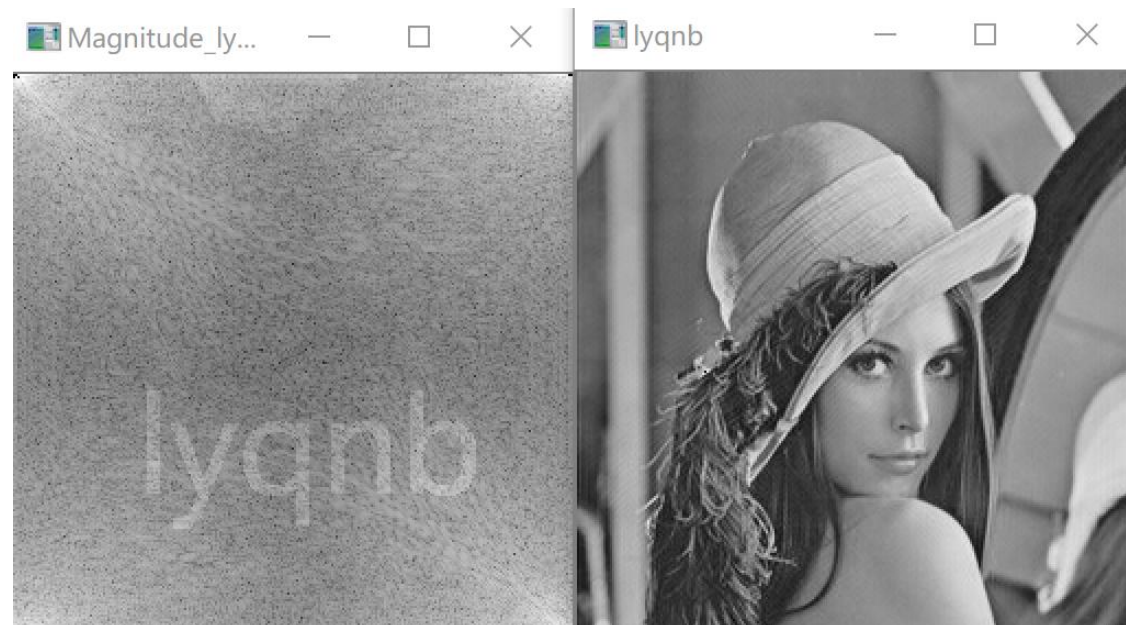


利用幅频进行滤波即给幅频添加一个 Mask，使其舍弃高频部分，然后再通过傅里叶逆变换还原图像，结果如下图所示：



### 3.6 探索：频率水印

将频域加上水印后，再进行傅里叶反变换，即可得到带有频域水印的图片，下图为频域水印和逆变换后的图像



可见，在频域加入水印对原图并不会产生很大的影响，但当把该图片用傅里叶转成频域时，发生了有趣的事：



水印居然便成上下对称的了，分析得知，时信号为偶对称，由于频域转换为时域的过程中丢失了虚部，使得水印的对称分布，并且能量平分。

## 四、结论与讨论

### 4.1 图像的高斯滤波与 Padding

在高斯滤波中，经过选取不同的 size 与 sigma 发现，sigma 越大，滤波后的图像越模糊，但 sigma 对图像的影响程度与 size 有关，size 越大，影响程度越大，这是因为大的 size 会让图像去加权更多周围的点，使得 sigma 的作用大大加强。Padding 与 3 种卷积方式的结合可以让边缘的处理更佳完善，除了 zero padding 外，其他三种 padding 都有很好的表现。

### 4.2 高斯核与高斯核的卷积实验

经过实验验证，两个一维高斯核的卷积结果是具有相同 sigma 的二维高斯核，分步对图像进行卷积的结果也与用二维高斯核一致。在时间测试时发现，可分离卷积的计算速度并没有二维卷积快，与理论推导不符。经过多次改变参数，发现当卷积核的 size 很大时，可分离卷积的速度要远快与二维卷积，在较小时，例如 size=3，并没有二维卷积快的原因是一维卷积需要做两次，而二维卷积只需要一次，较小的卷积核的对时间的影响没有计算次数的影响大。时间复杂度的推导会忽略掉常数倍数，所以导致在 size 较小时，速度与理论“不符”。在实验中，当 size 为 3 时，二维卷积的耗时为 1s 左右，可分离卷积的耗时为 2s 左右，当 size 为 99 时，二维卷积的耗时为 5s 左右，而可分离卷积的耗时仍为 2s 左右。

### 4.3 图像锐化

在实验中，alpha 的选择常常会使图像像素值超过 0~255 的范围，此时将图像输出，便会有肉眼可见的失真，所以要将图像标准化——给 RGB 每层都减自己的最小值，使得下界为 0，然后除最大值并乘 255，这样便实现了标准化，但这样操作后，图像的色彩与原图会出现差异。

### 4.4 双边滤波

在实验中，首先使用了 PPT 中所给的 sigma，但是滤完后发现图像和之前并没有区别，于是去掉 range 权重后再滤波，发现和高斯滤波结果相同，分析得可能是 range 权重的 sigma 取的过小，导致像素值稍有一点差异都会使得权重非常低，于是加大 sigma 到 10, 50, 100 后，发现双边滤波的效果与 PPT 一致，遂选择 sigma = 50 作为第三部分的演示图。

### 4.5 图像的傅里叶变换

图像可以看作是二维的离散信号，可以使用二维离散傅里叶变换，由于手写傅里叶太慢，于是使用 numpy.fft.fft2 实现变换。在对图像频域加 Mask 滤掉高频后，可以发现图像变得扭曲、模糊，这是因为图像精细程度是由高频信息决定的，舍弃后会变得粗糙、模糊。



## 五、代码

### 5.1 图像的高斯滤波与 Padding

高斯核生成函数：

```
def Gauss_fun(sigma, x):
    return np.exp(-(x / sigma) ** 2 / 2) / (sigma * (2 * np.pi) ** 0.5)

def Gauss_gen(sigma, size=3):
    kernel = np.zeros((size, size))
    for i in range(size):
        for j in range(size):
            kernel[i][j] = Gauss_fun(sigma, i - size // 2) * Gauss_fun(sigma, j - size // 2)
    kernel /= np.sum(kernel)
    return kernel
```

Padding:

```
def padding(image, size_edge0, size_edge1, mode_padding):
    img = np.zeros((image.shape[0] + 2 * size_edge0, image.shape[1] + 2 * size_edge1))
    if mode_padding == "zero":
        img[size_edge0:image.shape[0] + size_edge0, size_edge1:image.shape[1] + size_edge1] = image
    if mode_padding == "wrap":
        # center
        img[size_edge0:image.shape[0] + size_edge0, size_edge1:image.shape[1] + size_edge1] = image
        # side
        img[:size_edge0, size_edge1:-size_edge1] = image[image.shape[0] - size_edge0:, :]
        img[image.shape[0] + size_edge0:, size_edge1:-size_edge1] = image[:size_edge0, :]
        img[size_edge0:-size_edge0, :size_edge1] = image[:, image.shape[1] - size_edge0:]
        img[size_edge0:-size_edge0, image.shape[1] + size_edge1:] = image[:, :size_edge1]
        # corner
        img[:size_edge0, :size_edge1] = image[image.shape[0]-size_edge0:, image.shape[1]-size_edge1:]
        img[:size_edge0, image.shape[1] + size_edge1:] = image[image.shape[0] - size_edge0:, :size_edge1]
        img[image.shape[0] + size_edge0:, :size_edge1] = image[:size_edge0, image.shape[1] - size_edge1:]
        img[image.shape[0] + size_edge0:, image.shape[1] + size_edge1:] = image[:size_edge0, :size_edge1]
    if mode_padding == "copy":
        # center
        img[size_edge0:image.shape[0] + size_edge0, size_edge1:image.shape[1] + size_edge1] = image
        # side
        img[:size_edge0, size_edge1:-size_edge1] = image[0, :]
        img[image.shape[0] + size_edge0:, size_edge1:-size_edge1] = image[-1, :]
        img = img.T
        img[:size_edge1, size_edge0:-size_edge0] = image[:, 0]
        img[image.shape[1] + size_edge1:, size_edge0:-size_edge0] = image[:, -1]
        img = img.T
        # corner
        img[:size_edge0, :size_edge1] = image[0, 0]
        img[:size_edge0, image.shape[1] + size_edge1:] = image[0, -1]
        img[image.shape[0] + size_edge0:, :size_edge1] = image[-1, 0]
        img[image.shape[0] + size_edge0:, image.shape[1] + size_edge1:] = image[-1, -1]
```

```

if mode_padding == "reflect":
    # center
    img[size_edge0:image.shape[0] + size_edge0, size_edge1:image.shape[1] + size_edge1] = image
    # side
    img[:size_edge0, size_edge1:-size_edge1] = image[:size_edge0, :][::-1, :]
    img[image.shape[0] + size_edge0:, size_edge1:-size_edge1] = image[image.shape[0] - size_edge0:, :][::-1, :]
    img[size_edge0:-size_edge0, :size_edge1] = image[:, :size_edge1][::-1, :]
    img[size_edge0:-size_edge0, image.shape[1] + size_edge1:] = image[:, image.shape[1] - size_edge1:][::-1, :]
    # corner
    img[:size_edge0, :size_edge1] = image[:size_edge0, :size_edge1][::-1, ::-1]
    img[:size_edge0, image.shape[1] + size_edge1:] = image[:size_edge0, image.shape[1] - size_edge1:][::-1, ::-1]
    img[image.shape[0] + size_edge0:, :size_edge1] = image[image.shape[0] - size_edge0:, :size_edge1][::-1, ::-1]
    img[image.shape[0] + size_edge0:, image.shape[1] + size_edge1:] = image[image.shape[0] - size_edge0:, image.shape[1] - size_edge1:][::-1, ::-1]
return img

```

1/2D 卷积:

```

def conv(image, kernel, mode_conv="same", mode_padding="zero"):
    if mode_conv == "full":
        image_padding = padding(image, kernel.shape[0] - 1, kernel.shape[1] - 1, mode_padding=mode_padding)
        image_conv = np.zeros((image.shape[0] + kernel.shape[0] - 1, image.shape[1] + kernel.shape[1] - 1))
    if mode_conv == "same":
        image_padding = padding(image, kernel.shape[0] // 2, kernel.shape[1] // 2, mode_padding=mode_padding)
        image_conv = np.zeros((image.shape[0], image.shape[1]))
    if mode_conv == "valid":
        image_padding = image
        image_conv = np.zeros((image.shape[0] - kernel.shape[0] + 1, image.shape[1] - kernel.shape[1] + 1))
    length0 = kernel.shape[0] // 2
    length1 = kernel.shape[1] // 2
    kernel = kernel[length0:-length0, length1:-length1]
    for i in range(length0, image_padding.shape[0] - length0):
        for j in range(length1, image_padding.shape[1] - length1):
            image_conv[i - length0, j - length1] = np.sum(image_padding[i - length0:i + length0 + 1, j - length1:j + length1 + 1] * kernel)
    return image_conv

```

高斯滤波:

```

img = cv2.imread("Lena.jpg")

conv_setting = ["full", "same", "valid"]
padding_setting = ["zero", "wrap", "copy", "reflect"]
mode_conv = conv_setting[1]
mode_padding = padding_setting[0]

size = 5
sigma = 3
kernel = Gauss_gen(sigma=sigma, size=size)
if mode_conv == "full":
    img_gauss = np.zeros((img.shape[0] + len(kernel) - 1, img.shape[1] + len(kernel) - 1, 3))
if mode_conv == "same":
    img_gauss = np.zeros((img.shape[0], img.shape[1], 3))
if mode_conv == "valid":
    img_gauss = np.zeros((img.shape[0] - len(kernel) + 1, img.shape[1] - len(kernel) + 1, 3))

for i in range(3):
    img_gauss[:, :, i] = conv(image=img[:, :, i], kernel=kernel, mode_conv=mode_conv, mode_padding=mode_padding)

cv2.imshow("ori", img)
cv2.imshow("gauss", img_gauss.astype("uint8"))
cv2.waitKey(0)

```

## 5.2 高斯核与高斯核的卷积实验

```
sep_filter_0 = np.zeros((size, 1))
for i in range(size):
    sep_filter_0[i, 0] = Gauss_fun(sigma, i - size//2)
sep_filter_0 = sep_filter_0 / np.sum(sep_filter_0)
sep_filter_1 = sep_filter_0.T
img_gauss1d = img_gauss.copy()
for i in range(3):
    img_gauss1d[:, :, i] = conv(image=img[:, :, i], kernel=sep_filter_0, mode_conv=mode_conv, mode_padding=mode_padding)
for i in range(3):
    img_gauss1d[:, :, i] = conv(image=img_gauss1d[:, :, i], kernel=sep_filter_1, mode_conv=mode_conv, mode_padding=mode_padding)
```

## 5.3 图像锐化

```
alpha = 1
w = np.zeros((size, size))
w[size // 2, size // 2] = 1
kernel_sharp = (1+alpha)*w - alpha*kernel
for i in range(3):
    img_sharp[:, :, i] = conv(image=img[:, :, i], kernel=kernel_sharp, mode_conv=mode_conv, mode_padding=mode_padding)
cv2.normalize(img_sharp, img_sharp, 0, 255, cv2.NORM_MINMAX)
cv2.imshow("sharp", img_sharp.astype("uint8"))
```

## 5.4 双边滤波

```
def Bilateral_Filter(image, sigma_s, sigma_r, size, mode_conv="same", mode_padding="zero"):
    kernel_s = Gauss_gen(sigma=sigma_s, size=size)
    if mode_conv == "full":
        image_padding = padding(image, kernel_s.shape[0] - 1, kernel_s.shape[1] - 1, mode_padding=mode_padding)
        image_conv = np.zeros((image.shape[0] + kernel_s.shape[0] - 1, image.shape[1] + kernel_s.shape[1] - 1))
    if mode_conv == "same":
        image_padding = padding(image, kernel_s.shape[0] // 2, kernel_s.shape[1] // 2, mode_padding=mode_padding)
        image_conv = np.zeros((image.shape[0], image.shape[1]))
    if mode_conv == "valid":
        image_padding = image
        image_conv = np.zeros((image.shape[0] - kernel_s.shape[0] + 1, image.shape[1] - kernel_s.shape[1] + 1))
    length0 = kernel_s.shape[0] // 2
    length1 = kernel_s.shape[1] // 2
    for i in range(length0, image_padding.shape[0] - length0):
        for j in range(length1, image_padding.shape[1] - length1):
            delta = image_padding[i - length0:i + length0 + 1, j - length1:j + length1 + 1] - image_padding[i, j]
            kernel_r = Gauss_fun(sigma_r, delta)
            kernel = np.multiply(kernel_r, kernel_s)
            kernel /= np.sum(kernel)
            image_conv[i - length0, j - length1] = np.sum(np.multiply(image_padding[i - length0:i + length0 + 1, j - length1:j + length1 + 1], kernel))
    return image_conv
```

## 5.5 图像的傅里叶变换

```
img = cv2.imread("lena.jpg", 0)
img_fourier = np.fft.fft2(img)
img_fourier_shift = np.fft.fftshift(img_fourier)
Magnitude = 20*np.log(np.abs(img_fourier))
Magnitude_shift = 20*np.log(np.abs(img_fourier_shift))
Phase = np.angle(img_fourier)
Phase_shift = np.angle(img_fourier_shift)
cv2.imshow("ori", img)
cv2.imshow("Fourier", Magnitude.astype("uint8"))
cv2.imshow("Phase", Phase.astype("uint8"))
cv2.imshow("Fourier_shift", Magnitude_shift.astype("uint8"))
cv2.imshow("Phase_shift", Phase_shift.astype("uint8"))
```

频域滤波:

```
# filter
rate = 0.1
Magnitude_filter = np.zeros(Magnitude_shift.shape)
Magnitude_filter[int(img.shape[0]*(0.5 - rate)):int(img.shape[0]*(0.5 + rate)), :] = Magnitude_shift[int(img.shape[0]*(0.5 - rate)):int(img.shape[0]*(0.5 + rate)), :]
cv2.imshow("Magnitude_filter", Magnitude_filter.astype("uint8"))
Magnitude_filter = np.fft.ifftshift(Magnitude_filter)
Magnitude_filter = np.exp(Magnitude_filter / 20)
img_filter = np.real(np.fft.ifft2(np.multiply(Magnitude_filter, np.exp(1j*Phase))))
cv2.imshow("filter", img_filter.astype("uint8"))
```

## 5.6 探索：频率水印

```
# lyqnb
img_cxk = cv2.imread("cxk_head.png", 0)
cxk_head = img_cxk[img_cxk.shape[0] // 2 - 100:img_cxk.shape[0] // 2 + 156, img_cxk.shape[1] // 2 - 80:img_cxk.shape[1] // 2 + 176]
Magnitude_lyq = Magnitude + 0.1*cxk_head
cv2.imshow("Magnitude_lyq_ori", Magnitude_lyq.astype("uint8"))
Magnitude_lyq = np.exp(Magnitude_lyq / 20)
lyqnb = np.real(np.fft.ifft2(np.multiply(Magnitude_lyq, np.exp(1j*Phase))))
cv2.imshow("lyqnb", lyqnb.astype("uint8"))
img_fourier_lyq = np.fft.fft2(lyqnb)
Magnitude = 20*np.log(np.abs(img_fourier_lyq))
cv2.imshow("Magnitude_lyqnb", Magnitude.astype("uint8"))
cv2.waitKey(0)
```

## 5.7 作业所有代码及图片地址

<https://github.com/wyb2333/Computer-Vision-and-Pattern-Recognition>