# 计算机视觉——第一次作业

Koorye

2024 年 3 月 30 日

## 1 实验内容和目的

图像滤波是图像处理中的一种基本操作，它可以通过不同的滤波器对图像进行处理，以达到去噪、锐化、边缘检测等目的。本实验将通过基本的循环或 numpy 代码来实现卷积、滤波等操作，验证均值滤波、Sobel 滤波、高通滤波等滤波器作用于图像上的效果，并通过高通滤波器和低通滤波器的组合来实现混合图像的效果。

具体来说，本实验将完成以下内容：

1. 实现高斯滤波器卷积核的生成函数。
2. 实现滤波操作的函数。
3. 实现混合图像的函数。

通过上述实验内容，我可以更好地理解图像滤波的原理和实现方法，掌握图像滤波的基本操作。

## 2 实验原理

人眼对图像的感知是通过不同频率的光信号的叠加来实现的。低频信号对应图像的整体结构，而高频信号对应图像的细节。在距离较远时，人眼主要感知到图像的低频信息，而在距离较近时，人眼主要感知到图像的高频信息。因此，通过将低频信息和高频信息分别提取出来，可以实现混合图像的效果。

高斯滤波器是一种常用的低通滤波器，它可以通过卷积操作来实现。高斯滤波器的卷积核可以通过高斯函数来生成。高斯函数是一种钟形曲线，它可以由两个不同维度的高斯分布函数的乘积来表示。具体来说，多元高斯分布可以表示为公式**??**：

$$G(x) = (x-\mu)^T \Sigma^{-1} (x-\mu), \tag{1}$$

其中

$$x = [x1, x2, \ldots, x_n], \mu = [\mu_1, \mu_2, \ldots, \mu_n], \Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \ldots & \sigma_{1n} \\ \sigma_{21} & \sigma_2^2 & \ldots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \ldots & \sigma_n^2 \end{bmatrix}. \tag{2}$$

假设 $\Sigma$ 是对角矩阵，即 $\sigma_{ij} = 0$，则多元高斯分布可以简化为一元高斯分布的乘积，即公式**??**：

$$G(x) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_i} \exp(-\frac{(x_i - \mu_i)^2}{2\sigma_i^2}), \tag{3}$$

其中，$\sigma_i$ 是第 $i$ 个高斯分布的标准差，$\mu_i$ 是第 $i$ 个高斯分布的均值。通过上述公式，可以生成高斯滤波器的卷积核。

此外，还有其他常见的滤波器，如 Sobel 滤波器、均值滤波器等。Sobel 滤波器是一种边缘检测滤波器，它可以通过卷积操作来实现。Sobel 滤波器的卷积核可以通过 Sobel 算子来生成。Sobel 算子是一种差分算子，它可以通过水平方向和垂直方向的差分来实现。而均值滤波器是一种平滑滤波器，可以通过为卷积核中的每个元素赋予相同的权重来实现。

在得到不同滤波器的卷积核后，可以通过卷积操作来实现滤波操作。卷积操作可以通过公式**??**来实现：

$$I'(x,y) = \sum_{i=0}^{k} \sum_{j=0}^{k} I(x+i, y+j) \cdot K(i,j), \tag{4}$$

其中，$I(x,y)$ 是原始图像的像素值，$I'(x,y)$ 是滤波后的图像的像素值，$K(i,j)$ 是卷积核的值，$k$ 是卷积核的大小。

通过上述滤波操作，可以实现对图像的去噪、锐化、边缘检测等操作。此外，通过高通滤波器和低通滤波器的组合，还可以实现混合图像的效果。

具体来说，可以通过高通滤波器提取图像的高频信息，通过低通滤波器提取图像的低频信息，然后将两者叠加在一起，即可实现混合图像的效果。

在实际操作中，可以通过循环或 numpy 代码来实现卷积、滤波等操作。通过调用相应的函数，可以实现高斯滤波器卷积核的生成、滤波操作、混合图像等操作。

# 3 实验环境

本实验基于以下环境：

- 操作系统：Windows 11
- 编程语言：Python 3.12.1
- 编程工具：Jupyter Notebook
- Python 库：numpy 1.24.4、matplotlib 3.7.5、torch 2.2.2

# 4 实验步骤

本章节将详细介绍实验的步骤，包括环境配置、高斯滤波器的生成、滤波操作的实现、混合图像的实现、测试程序的运行等。

## 4.1 环境配置

首先，需要安装 Python 环境，这里我通过 miniconda3 来安装 Python 环境。具体来说，可以通过以下命令来安装 miniconda3：

```
scoop install miniconda3
conda create -n cv python=3.12.1
conda activate cv
```

之后，安装所需的库：

```
pip install numpy==1.24.4 matplotlib==3.7.5 torch==2.2.2
```

之后，配置 Jupyter Notebook 环境并启动：

```
pip install jupyter
jupyter notebook
```

启动成功后，可以看到 Jupyter Notebook 的界面，如图??所示，证明环境配置成功。

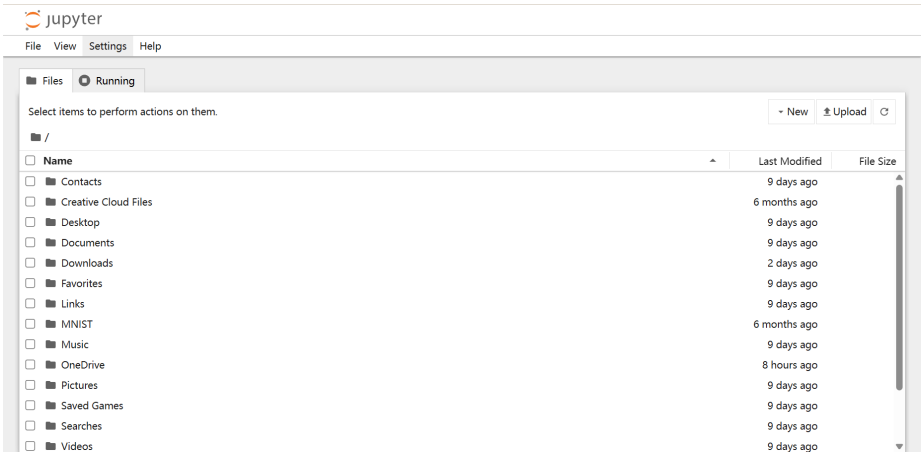



图 1: Jupyter Notebook 界面

## 4.2 高斯滤波器的生成

如上文所述，高斯滤波器的卷积核可以通过多个一元高斯分布的乘积来生成。具体来说，可以通过以下代码来生成高斯滤波器的卷积核：

```
def create_Gaussian_kernel(cutoff_frequency):
    # define 1d gaussion function
    def gaussion(x, mean, std):
        return 1 / (std * np.sqrt(2 * np.pi)) * np.exp(-0.5 * ((x - mean) /
            std) ** 2)

    def create_gaussian_naive(cutoff_frequency):
        """ Naive implementation of Gaussian kernel. """
        # define kernel size
        k = cutoff_frequency * 4 + 1
        kernel = np.zeros((k, k))

        # define mean and std
```

```
13          mean = k // 2
14          std = cutoff_frequency
15
16          # calculate kernel for each pixel
17          for i in  range(k):
18              for j in  range(k):
19                  kernel[i, j] = gaussion(i, mean, std) * gaussion(j, mean,
                        std)
20
21          # normalize the kernel
22          kernel /= np. sum(kernel)
23          return kernel
24
25      def create_gaussian_numpy(cutoff_frequency):
26          """ Numpy implementation of Gaussian kernel. """
27          ...
28          # create 1d gaussion function
29          x = np.arange(k)
30          kernel_x = gaussion(x, mean, std)
31
32          # create 2d gaussion function
33          kernel = np.outer(kernel_x, kernel_x)
34          ...
35          return kernel
36
37      # return create_gaussian_naive(cutoff_frequency)
38      return create_gaussian_numpy(cutoff_frequency)
```

考虑到篇幅限制，部分重复代码被用省略号代替，完整代码可以在附录 A 中找到。上述代码中，首先定义了一维高斯函数 `gaussion`，然后提出了 2 种实现方式：

1. `create_gaussian_naive`：通过循环来实现高斯滤波器的卷积核生成。具体来说，首先定义了卷积核的大小以及高斯分布的均值和标准差，然后遍历每个像素，通过 2 个不同维度的一元高斯分布的乘积得到结果，最后对卷积核进行归一化。

2. `create_gaussian_numpy`：通过 numpy 来实现高斯滤波器的卷积核生成。具体来说，首先定义了卷积核的大小以及高斯分布的均值和标准

差，然后通过 numpy 的 `outer` 函数来实现 2 个不同维度的一元高斯分布的乘积，最后对卷积核进行归一化。

通过上述函数，在指定截止频率的情况下，可以生成高斯滤波器的卷积核。

## 4.3 滤波操作的实现

如上文所述，可以通过卷积操作来实现滤波操作。具体来说，可以通过以下代码来实现滤波操作：

```
1  def my_imfilter(image, filter):
2      # naive implementation
3      def my_imfilter_naive(image, filter):
4          """ Naive implementation of image filter, use for loop. """
5
6          def apply_filter(patch, filter):
7              pixel = 0
8              # for each pixel
9              for k in range(K):
10                 for j in range(J):
11                     # multiply the pixel value with the filter value
12                     pixel += patch[k, j] * filter[k, j]
13             # the final pixel value is the sum of all the pixel value
                   multiplied by the filter value
14             return pixel
15
16         M, N, C = image.shape
17         K, J = filter.shape
18
19         # padding image
20         # (M, N, C) -> (M + 2 * P, N + 2 * P, C)
21         pad_M, pad_N = K // 2, J // 2
22         image_pad = np.zeros((M + 2 * pad_M, N + 2 * pad_N, C))
23         image_pad[pad_M: M + pad_M, pad_N: N + pad_N, :] = image
24
25         image_new = np.zeros((M, N, C))
26
27         # for each pixel, ignore the boundary
```

```
28          for m in  range(M):
29              for n in  range(N):
30                  # for each channel
31                  for c in  range(C):
32                      # get image patch by using the current pixel (i, j) as
                            the center
33                      patch = image_pad[m: m + K, n: n + J, c]
34                      image_new[m, n, c] = apply_filter(patch,  filter)
35
36          return image_new
37
38      # ================================================
39      # numpy implementation, using matrix multiplication to accelerate the
            process
40      def my_imfilter_numpy(image,  filter):
41          ...
42          image_pad = np.pad(image, ((pad_M, pad_M), (pad_N, pad_N), (0, 0)),
                'constant')
43
44          # image to column
45          # (M, N, C) -> (M * N * C, K * j)
46          image_col = np.zeros((M * N * C, K * J))
47          col_idx = 0
48          for m in  range(M):
49              for n in  range(N):
50                  for c in  range(C):
51                      patch = image_pad[m: m + K, n: n + J, c]
52                      image_col[col_idx] = patch.flatten()
53                      col_idx += 1
54
55          # apply filter
56          # (M * N * C, K * J) @ (K * J, 1) -> (M * N * C, 1)
57          filter_col =  filter.flatten()
58          image_new_col = np.dot(image_col, filter_col)
59
60          # reshape the image
61          image_new = image_new_col.reshape((M, N, C))
62          return image_new
63
```

```
64      # ================================================
65
66      # numpy implementation, using stride trick to accelerate image to column
67      def my_imfilter_numpy_stride_trick(image,  filter):
68          """ Numpy implementation of image filter, use matrix multiplication.
                """
69          ...
70          image_col = np.lib.stride_tricks.sliding_window_view(image_pad, (K,
                J, 1)).reshape((M * N * C, K * J))
71          ...
72          return image_new
73
74      # ================================================
75
76      # fft implementation, faster but may have some error
77      def my_imfilter_fft(image,  filter):
78          """ FFT implementation of image filter. """
79          freq_filter = np.fft.fft2( filter, s=image.shape[:2])
80          for c in  range(image.shape[2]):
81              freq_per_channel = np.fft.fft2(image[:, :, c])
82              image[:, :, c] = np.fft.ifft2(freq_per_channel * freq_filter).
                    real
83          return image
84
85      # return my_imfilter_naive(image, filter)
86      # return my_imfilter_numpy(image, filter)
87      return my_imfilter_numpy_stride_trick(image,  filter)
88      # return my_imfilter_fft(image, filter)
```

考虑到篇幅限制，部分重复代码被用省略号代替，完整代码可以在附录 A 中找到。上述代码中，提出了 4 种滤波操作的实现方式：

1. `my_imfilter_naive`：通过循环来实现滤波操作。具体来说，首先对图像进行填充，然后遍历每个像素，通过应用卷积操作得到结果。卷积操作同样通过循环来实现，最后得到滤波后的图像。

2. `my_imfilter_numpy`：通过 numpy 来实现滤波操作。具体来说，首先对图像进行填充，然后通过循环操作来实现图像到列的转换，最后通过 numpy 的矩阵乘法来实现卷积操作，最后还原原图像的尺寸。

3. `my_imfilter_numpy_stride_trick`：通过 numpy 的 `sliding_window_view` 函数来实现滤波操作。具体来说，首先对图像进行填充，然后通过 numpy 的 `sliding_window_view` 函数来实现卷积操作，最后得到滤波后的图像。

4. `my_imfilter_fft`：通过 FFT 来实现滤波操作。具体来说，首先对滤波器进行傅里叶变换，然后对图像的每个通道进行傅里叶变换，最后通过频域的乘法来实现卷积操作，最后通过逆傅里叶变换得到滤波后的图像。

通过上述函数，可以在指定图像和滤波器的情况下，实现滤波操作。

## 4.4 混合图像的实现

如上文所述，可以通过高通滤波器和低通滤波器的组合来实现混合图像的效果。具体来说，可以通过以下代码来实现混合图像的效果：

```python
def create_hybrid_image(image1, image2,  filter):
    if image1.shape != image2.shape:
        # resize the larger image to the size of the smaller image using
            numpy
        image1_size = image1.shape[0] * image1.shape[1]
        image2_size = image2.shape[0] * image2.shape[1]
        if image1_size < image2_size:
            np.resize(image2, image1.shape)
        else:
            np.resize(image1, image2.shape)

    # get low pass by using the filter
    low_pass_image1 = my_imfilter(image1,  filter)
    # get high pass by subtracting the low pass from the original image
    high_pass_image2 = image2 - my_imfilter(image2,  filter)
    # hybrid image is the sum of low pass and high pass
    hybrid_image = low_pass_image1 + high_pass_image2

    # clip the pixel values
    hybrid_image = np.clip(hybrid_image, 0.0, 1.0)

    return low_pass_image1, high_pass_image2, hybrid_image
```

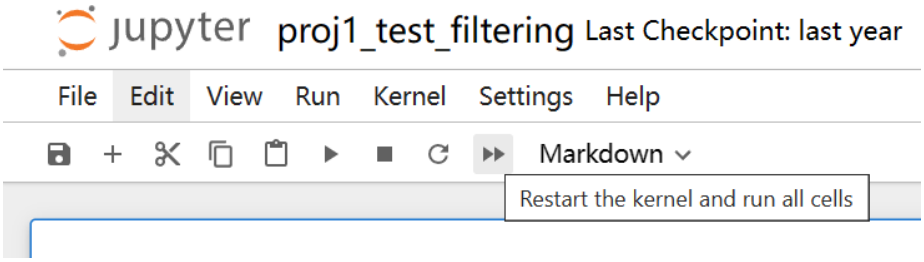



图 2: Jupyter Notebook 运行所有单元格

考虑到篇幅限制，部分重复代码被用省略号代替，完整代码可以在附录 A 中找到。函数 `create_hybrid_image` 接收两个图像和一个滤波器作为输入，然后通过低通滤波器提取第一个图像的低频信息，通过原图与低通滤波后的图像相减提取第二个图像的高频信息，最后将两者叠加在一起，即可得到混合图像。为了防止像素值超出范围，还需要对混合图像进行裁剪。通过上述函数，可以实现混合图像的效果。

### 4.5 测试程序的运行

在实现了上述函数后，可以在 Jupyter Notebook 中调用这些函数来测试程序的运行。如图**??**所示，在 Jupyter Notebook 中打开 `part1_test_filtering.ipynb` 和 `part1.ipynb` 文件，然后点击”Restart the kernel and run all cells” 按钮，即可运行测试程序。

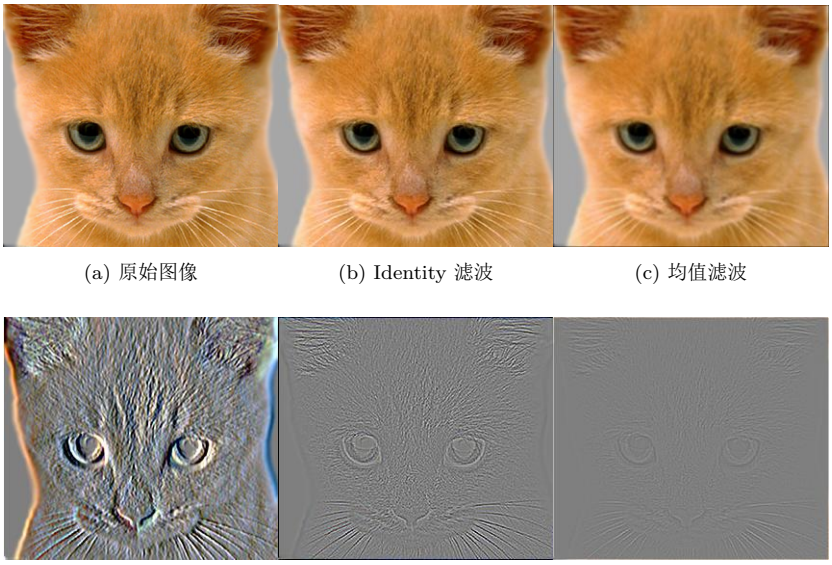
运行成功后，可以看到测试程序的输出结果。

## 5 实验结果与分析

### 5.1 图像滤波的结果

图**??**展示了 `part1_test_filtering.ipynb` 的运行结果，图中包含了原始图像、均值滤波器、Sobel 滤波器、拉普拉斯滤波器处理后的图像，可以通过这些图像来观察滤波器作用于图像上的效果。具体来说：

1. 子图 (a) 是原始图像。
2. 子图 (b) 是 Identity 滤波器处理后的图像，由于滤波器被设置为中心为 1，周围为 0，因此图像不会发生变化。



(a) 原始图像 (b) Identity 滤波 (c) 均值滤波

(d) Sobel 滤波 (e) 拉普拉斯滤波 (f) 高通滤波

图 3: 测试程序的输出结果

3. 子图 (c) 是均值滤波器处理后的图像，由于滤波器的每个元素被设置为和为 1 的均值，因此图像中每个像素会变成当前像素与周围像素的均值，即图像会变得模糊。

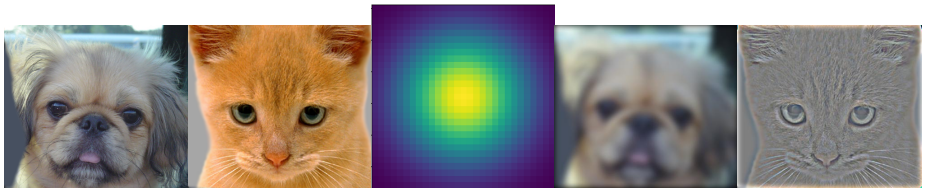
4. 子图 (d) 是 Sobel 滤波器处理后的图像，由于滤波器左边的元素被设为负值，右边的元素被设为正值，因此图像中的每个像素会变成左边像素与右边像素的差值，突出垂直边缘信息。

5. 子图 (e) 是拉普拉斯滤波器处理后的图像，由于滤波器中心为-4，周围为 1，因此图像中的每个像素会变成当前像素与周围像素的差值，突出图像的边缘信息。

6. 子图 (f) 是高通图像，是原始图像减去均值滤波器处理后的图像，由于均值滤波会模糊图像，因此原始图像与模糊图像的差值会突出图像中显著的细节信息。

## 5.2 图像混合的结果

图??展示了 `part1.ipynb` 的运行结果，首先程序返回的验证结果均为 True，证明程序编写无误。接下来，运行结果中包含了原始图像、高斯滤波器、滤波后的图像和混合图像，可以通过这些图像来观察滤波器作用于图像上的效果。具体来说：



(a) 原始图像 (b) 原始图像 (c) 高斯滤波器 (d) 低通图像 (e) 高通图像

(f) 混合图像

图 4: 测试程序的输出结果

1. 子图 (a)、(b) 是原始图像。
2. 子图 (c) 是高斯滤波器，高斯滤波器被设置为中间大，周围小的二维高斯分布，因此可以起到平滑图像的作用。
3. 子图 (d) 是低通图像，是原始图像与高斯滤波器处理后的图像的卷积结果，可以看到图像中的整体信息被保留。
4. 子图 (e) 是高通图像，是原始图像减去低通图像，可以看到图像中的细节信息被突出。

5. 子图 (f) 是混合图像，是低通图像和高通图像的相加，可以看到图像中的整体信息和细节信息被合并在一起。将图像按不同尺寸显示，可以看到图像越大，越容易看到高频信息（猫）；图像越小，越容易看到低频信息（狗）。

需要注意的是，基于 FFT 的卷积方式得到的结果与其他方法不同，并不能实现有效的图像混合。如图**??**所示，通过 FFT 方式得到的混合图像与其他方式得到的混合图像有明显的差异，这是由于 FFT 是基于频域的角度来实现卷积，而空域到频域的转换会导致一些信息的丢失，如一些细节信息，因此在实际应用中需要根据具体情况选择合适的实现方式。基于 FFT 得到的完整结果可以在附录 B 中找到。

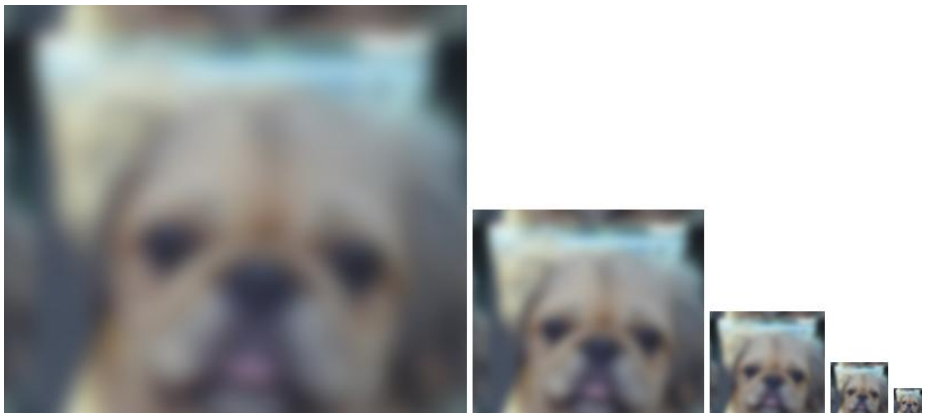


图 5: FFT 得到的混合图像

## 5.3 运行效率的比较

`part1.ipynb` 中展示了混合操作的运行时间，以及与 torch 卷积操作的运行时间的比较。如表**??**所示，可以看出不同实现方式的运行时间有较大差异，其中 Naive 实现方式的运行时间最长，FFT 实现方式的运行时间最短，但是结果存在误差。而 Numpy 和 Numpy Kernel Trick 的实现方式在运行时间和结果正确性上都有较好的表现，不过与 Pytorch 相比，仍然有一定的差距。

其中，高斯核生成操作的实现方式影响不大，而卷积操作的实现方式对运行时间有较大影响。这是因为卷积操作是整个混合图像生成过程中最耗

时的操作，涉及 5 层循环：遍历图像的每个像素（行和列）、遍历图像的每个通道、遍历卷积核的每个像素（行和列）。因此，通过优化卷积操作的实现方式，可以显著提高运行效率。考虑到估计时间的误差，实际运行效率可能会有所不同，但是大致可以看出不同实现方式的运行效率的差异。

表 1: 运行时间的比较

| 高斯核生成操作 | 卷积操作 | 结果是否正确 | 耗时（秒） |
|---|---|---|---|
| Naive | Naive | √ | 550.127 |
| | Numpy | √ | 8.726 |
| | Numpy Kernel Trick | √ | 6.578 |
| | Numpy FFT | × | 0.271 |
| Numpy | Naive | √ | 549.535 |
| | Numpy | √ | 7.483 |
| | Numpy Kernel Trick | √ | 5.109 |
| | Numpy FFT | × | 0.277 |
| Pytorch | Pytorch | √ | 2.595 |

## 5.4 总结

总的来说，本实验通过实现高斯滤波器的卷积核生成、滤波操作、混合图像等操作，验证了均值滤波、Sobel 滤波、高通滤波等滤波器作用于图像上的效果，并通过高通滤波器和低通滤波器的组合来实现混合图像的效果。通过实验得出了如下结论：

1. 滤波器是一种简单而有效的图像处理操作，通过简单定义不同的滤波器，就可以实现多样的图像处理效果，例如均值滤波器可以模糊图像，Sobel 滤波器可以突出图像的边缘信息，拉普拉斯滤波器可以突出图像的边缘信息。

2. 通过高通滤波器和低通滤波器的组合，可以实现混合图像的效果，即将图像的低频信息和高频信息分别提取出来，然后叠加在一起。混合图像可以实现有趣的视觉效果，例如随着观察距离的变化，图像的低频和高频信息会以不同的形式呈现。图像越近，越容易看到高频信息；图像越远，越容易看到低频信息。

3. 卷积的实现方式有多种，包括循环、矩阵相乘、利用 numpy 内置函数、傅立叶快速卷积等方式。不同的实现方式有不同的运行效率。其中循环的运行效率非常低下，傅立叶快速卷积虽然运行效率较高，但可能会有一些误差，因此在实际应用中需要根据具体情况选择合适的实现方式。

# 6 实验结论

本实验讲述了滤波器和卷积操作的原理，以不同方式实现了滤波器生成、卷积、混合图像等操作，验证了均值滤波、Sobel 滤波、高通滤波等滤波器作用于图像上的效果，并通过高通图像和低通图像的组合来实现混合图像的效果，还对不同实现方式的运行效率进行比较。通过本次实验，我对图像滤波的原理和实现方法有了更深入的了解，掌握了图像滤波的基本操作。

对于本实验，我认为还有以下几点可以改进的地方：

1. 在实现滤波器生成时，可以尝试更多的滤波器的生成方式，例如通过方差不相等的高斯分布函数的乘积来生成高斯滤波器的卷积核，或者协方差不为 0 的高斯分布函数的乘积来生成多维高斯滤波器的卷积核，并观察不同滤波器对图像的影响。

2. 在实现混合图像时，可以尝试更多的滤波器组合，例如通过不同的高通滤波器和低通滤波器的组合，来实现更多样的混合图像效果。

# A 完整代码

```python
#!/usr/bin/python3

import numpy as np


def create_Gaussian_kernel(cutoff_frequency):
    """
        Returns a 2D Gaussian kernel using the specified filter size standard
        deviation and cutoff frequency.

        The kernel should have:
        - shape (k, k) where k = cutoff_frequency * 4 + 1
        - mean = floor(k / 2)
        - standard deviation = cutoff_frequency
        - values that sum to 1

        Args:
        - cutoff_frequency: an int controlling how much low frequency to leave in
          the image.
        Returns:
        - kernel: numpy nd-array of shape (k, k)

        HINT:
        - The 2D Gaussian kernel here can be calculated as the outer product of two
          vectors with values populated from evaluating the 1D Gaussian PDF at each
          corrdinate.
    """
    # define 1d gaussion function
    def gaussion(x, mean, std):
        return 1 / (std * np.sqrt(2 * np.pi)) * np.exp(-0.5 * ((x - mean) / std) ** 2)

    def create_gaussian_naive(cutoff_frequency):
```

```
33          """ Naive implementation of Gaussian kernel. """
34          # define kernel size
35          k = cutoff_frequency * 4 + 1
36          kernel = np.zeros((k, k))
37
38          # define mean and std
39          mean = k // 2
40          std = cutoff_frequency
41
42          # calculate kernel for each pixel
43          # each pixel is multiplied by the product of 1d gaussion function of
                X and 1d gaussion function of Y
44          # the product of two 1d gaussion function is the 2d gaussion
                function
45          for i in  range(k):
46              for j in  range(k):
47                  kernel[i, j] = gaussion(i, mean, std) * gaussion(j, mean,
                        std)
48
49          # normalize the kernel
50          kernel /= np. sum(kernel)
51          return kernel
52
53      def create_gaussian_numpy(cutoff_frequency):
54          """ Numpy implementation of Gaussian kernel. """
55          # define kernel size
56          k = cutoff_frequency * 4 + 1
57          kernel = np.zeros((k, k))
58
59          # define mean and std
60          mean = k // 2
61          std = cutoff_frequency
62
63          # create 1d gaussion function
64          x = np.arange(k)
65          kernel_x = gaussion(x, mean, std)
66
67          # create 2d gaussion function
68          kernel = np.outer(kernel_x, kernel_x)
```

```
69
70          # normalize the kernel
71          kernel /= np. sum(kernel)
72          return kernel
73
74      # return create_gaussian_naive(cutoff_frequency)
75      return create_gaussian_numpy(cutoff_frequency)
76
77
78  def my_imfilter(image,  filter):
79      """
80      Apply a filter to an image. Return the filtered image.
81
82      Args
83      - image: numpy nd-array of shape (m, n, c)
84      - filter: numpy nd-array of shape (k, j)
85      Returns
86      - filtered_image: numpy nd-array of shape (m, n, c)
87
88      HINTS:
89      - You may not use any libraries that do the work for you. Using numpy to
            work
90      with matrices is fine and encouraged. Using OpenCV or similar to do the
91      filtering for you is not allowed.
92      - I encourage you to try implementing this naively first, just be aware
            that
93      it may take an absurdly long time to run. You will need to get a
            function
94      that takes a reasonable amount of time to run so that the TAs can verify
95      your code works.
96      """
97
98      # naive implementation
99      def my_imfilter_naive(image,  filter):
100         """ Naive implementation of image filter, use for loop. """
101
102         def apply_filter(patch,  filter):
103             """
104             Apply filter to a patch of image.
```

```
105             Params:
106             - patch: numpy nd-array of shape (K, J)
107             - filter: numpy nd-array of shape (K, J)
108             Returns:
109             - pixel: float
110             """
111             pixel = 0
112             # for each pixel
113             for k in  range(K):
114                 for j in  range(J):
115                     # multiply the pixel value with the filter value
116                     pixel += patch[k, j] *  filter[k, j]
117             # the final pixel value is the sum of all the pixel value
                    multiplied by the filter value
118             return pixel
119
120         M, N, C = image.shape
121         K, J =  filter.shape
122
123         # padding image
124         # (M, N, C) -> (M + 2 * P, N + 2 * P, C)
125         pad_M, pad_N = K // 2, J // 2
126         image_pad = np.zeros((M + 2 * pad_M, N + 2 * pad_N, C))
127         image_pad[pad_M: M + pad_M, pad_N: N + pad_N, :] = image
128
129         image_new = np.zeros((M, N, C))
130
131         # for each pixel, ignore the boundary
132         for m in  range(M):
133             for n in  range(N):
134                 # for each channel
135                 for c in  range(C):
136                     # get image patch by using the current pixel (i, j) as
                            the center
137                     patch = image_pad[m: m + K, n: n + J, c]
138                     image_new[m, n, c] = apply_filter(patch,  filter)
139
140         return image_new
141
```

```
142     # ==================================
143
144     # numpy implementation, using matrix multiplication to accelerate the
            process
145     def my_imfilter_numpy(image,  filter):
146         """ Numpy implementation of image filter, use matrix multiplication.
                """
147
148         M, N, C = image.shape
149         K, J =  filter.shape
150
151         # padding image
152         # (M, N, C) -> (M + 2 * P, N + 2 * P, C)
153         pad_M, pad_N = K // 2, J // 2
154         image_pad = np.pad(image, ((pad_M, pad_M), (pad_N, pad_N), (0, 0)),
                'constant')
155
156         # image to column
157         # (M, N, C) -> (M * N * C, K * j)
158         image_col = np.zeros((M * N * C, K * J))
159         col_idx = 0
160         for m in  range(M):
161             for n in  range(N):
162                 for c in  range(C):
163                     patch = image_pad[m: m + K, n: n + J, c]
164                     image_col[col_idx] = patch.flatten()
165                     col_idx += 1
166
167         # apply filter
168         # (M * N * C, K * J) @ (K * J, 1) -> (M * N * C, 1)
169         filter_col =  filter.flatten()
170         image_new_col = np.dot(image_col, filter_col)
171
172         # reshape the image
173         image_new = image_new_col.reshape((M, N, C))
174         return image_new
175
176     # ==================================
177
```

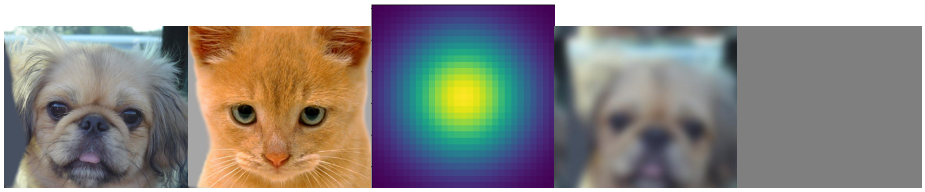
```
178     # numpy implementation, using stride trick to accelerate image to column
179     def my_imfilter_numpy_stride_trick(image,  filter):
180         """ Numpy implementation of image filter, use matrix multiplication.
                """
181 
182         M, N, C = image.shape
183         K, J =  filter.shape
184 
185         # padding image
186         # (M, N, C) -> (M + 2 * P, N + 2 * P, C)
187         pad_M, pad_N = K // 2, J // 2
188         image_pad = np.pad(image, ((pad_M, pad_M), (pad_N, pad_N), (0, 0)),
                'constant')
189         image_col = np.lib.stride_tricks.sliding_window_view(image_pad, (K,
                J, 1)).reshape((M * N * C, K * J))
190 
191         # apply filter
192         # (M * N * C, K * J) @ (K * J, 1) -> (M * N * C, 1)
193         filter_col =  filter.flatten()
194         image_new_col = np.dot(image_col, filter_col)
195 
196         # reshape the image
197         image_new = image_new_col.reshape((M, N, C))
198         return image_new
199 
200     # ==================================
201 
202     # fft implementation, faster but may have some error
203     def my_imfilter_fft(image,  filter):
204         """ FFT implementation of image filter. """
205 
206         freq_filter = np.fft.fft2( filter, s=image.shape[:2])
207 
208         for c in  range(image.shape[2]):
209             freq_per_channel = np.fft.fft2(image[:, :, c])
210             image[:, :, c] = np.fft.ifft2(freq_per_channel * freq_filter).
                    real
211 
212         return image
```

```
213
214     # return my_imfilter_naive(image, filter)
215     # return my_imfilter_numpy(image, filter)
216     return my_imfilter_numpy_stride_trick(image,  filter)
217     # return my_imfilter_fft(image, filter)
218
219
220 def create_hybrid_image(image1, image2,  filter):
221     """
222     Takes two images and a low-pass filter and creates a hybrid image.
            Returns
223     the low frequency content of image1, the high frequency content of image
            2,
224     and the hybrid image.
225
226     Args
227     - image1: numpy nd-array of dim (m, n, c)
228     - image2: numpy nd-array of dim (m, n, c)
229     - filter: numpy nd-array of dim (x, y)
230     Returns
231     - low_frequencies: numpy nd-array of shape (m, n, c)
232     - high_frequencies: numpy nd-array of shape (m, n, c)
233     - hybrid_image: numpy nd-array of shape (m, n, c)
234
235     HINTS:
236     - You will use your my_imfilter function in this function.
237     - You can get just the high frequency content of an image by removing
            its low
238       frequency content. Think about how to do this in mathematical terms.
239     - Don't forget to make sure the pixel values of the hybrid image are
            between
240       0 and 1. This is known as 'clipping'.
241     - If you want to use images with different dimensions, you should resize
            them
242       in the notebook code.
243     """
244     if image1.shape != image2.shape:
245         # resize the larger image to the size of the smaller image using
                numpy
```

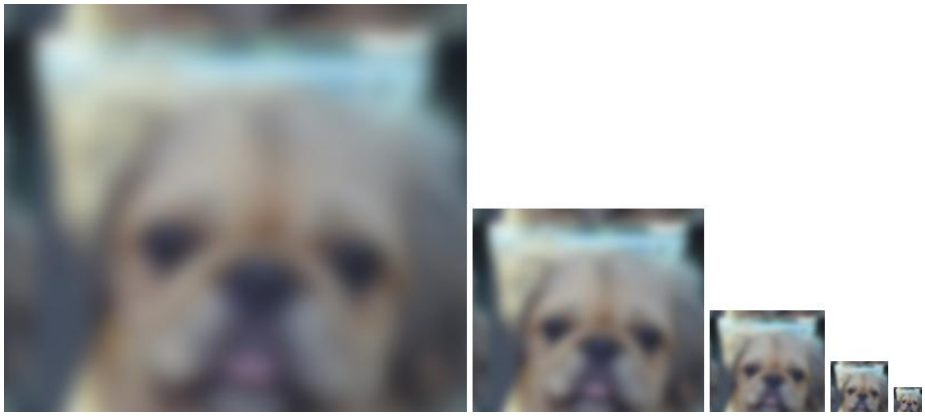
```
246         image1_size = image1.shape[0] * image1.shape[1]
247         image2_size = image2.shape[0] * image2.shape[1]
248         if image1_size < image2_size:
249             np.resize(image2, image1.shape)
250         else:
251             np.resize(image1, image2.shape)
252 
253     # get low pass by using the filter
254     low_pass_image1 = my_imfilter(image1,  filter)
255     # get high pass by subtracting the low pass from the original image
256     high_pass_image2 = image2 - my_imfilter(image2,  filter)
257     # hybrid image is the sum of low pass and high pass
258     hybrid_image = low_pass_image1 + high_pass_image2
259 
260     # clip the pixel values
261     hybrid_image = np.clip(hybrid_image, 0.0, 1.0)
262 
263     return low_pass_image1, high_pass_image2, hybrid_image
```

# B 基于 FFT 的图像混合

图??展示了基于 FFT 的图像混合的输出结果。可以看出，FFT 难以保留图像的细节，导致混合图像的效果不如直接使用滤波器的方法。



(a) 原始图像 (b) 原始图像 (c) 高斯滤波器 (d) 低通图像 (e) 高通图像



(f) 混合图像

图 6: 测试程序的输出结果