



华南理工大学  
South China University of Technology

# 实验报告

## 实验四

课程名称：《数字信号处理实验》

学生姓名：zyh

学生学号：202264691103

学生专业：人工智能

开课学期：2023-2024 年第二学期

提交日期：2024 年 6 月 23 日

# 目录

<b>1 验证性实验</b>	<b>1</b>
1.1 实验目的 . . . . .	1
1.2 实验原理 . . . . .	1
1.2.1 窗函数 . . . . .	1
1.2.2 窗函数设计 FIR 滤波器 . . . . .	2
1.2.3 双线性变换法 . . . . .	3
1.2.4 IIR 滤波器 . . . . .	4
1.2.5 双线性变换法设计 IIR 滤波器 . . . . .	5
1.3 实验内容 . . . . .	6
1.3.1 窗函数法设计 FIR 滤波器 . . . . .	6
1.3.2 双线性变换法设计 IIR 滤波器 . . . . .	9
<b>2 课程设计 1（来源实验题目）：图像信号的卷积处理</b>	<b>12</b>
2.1 课程设计目的 . . . . .	12
2.2 课程设计要求 . . . . .	12
2.3 课程实验原理 . . . . .	12
2.3.1 图像卷积 . . . . .	12
2.3.2 低通滤波器 . . . . .	13
2.3.3 高通滤波器 . . . . .	14
2.3.4 匹配滤波边缘 . . . . .	15
2.3.5 边缘检测 . . . . .	16
2.4 课程设计内容 . . . . .	17
2.4.1 图像进行不同卷积核运算 . . . . .	17
2.4.2 图像基本处理 . . . . .	25
2.4.3 图像特效处理 . . . . .	29
2.4.4 图像变换处理 . . . . .	33
2.4.5 GUI 用户交互页面 . . . . .	36
2.5 课程设计结果 . . . . .	47
2.5.1 GUI 交互界面效果 . . . . .	47
<b>3 课程设计 2（自拟）：fNIRS 信号的处理</b>	<b>50</b>
3.1 选题背景 . . . . .	50
3.1.1 fNIRS 信号 . . . . .	50
3.1.2 fNIRS 测量原理 . . . . .	50
3.1.3 应用场景 . . . . .	50

3.2	课程设计目的	50
3.3	课程设计原理	51
3.3.1	光密度转换	51
3.3.2	血氧浓度转换	51
3.3.3	带通滤波处理	52
3.3.4	基线校准	53
3.4	课程设计内容	54
3.4.1	初始化变量并读取数据	54
3.4.2	处理为 MNE 对象	54
3.4.3	转换光密度与血氧密度	55
3.4.4	制作 MNE 任务表并滤波	55
3.4.5	基线校准 fNIRS 信号	56
3.4.6	fNIRS 信号可视化展示	57
3.5	课程设计结果	58

4	鸣谢	61
---	----	----

# 1 验证性实验

## 1.1 实验目的

- 窗函数法设计 FIR 滤波器
- 双线性变换法设计 IIR 滤波器 [1]

## 1.2 实验原理

### 1.2.1 窗函数

当我们在进行数字信号处理时，窗函数是一个非常重要的概念。下面是窗函数的解释和原理：

- **定义：**在数字信号处理中，窗函数是一种用于加权信号的函数，它将信号限制在一个有限的时间或频率范围内。窗函数通常在信号的端点上逐渐减小，以防止频谱泄漏或降低频谱分辨率。
- **原理：**窗函数的作用是在时域或频域上对信号进行截断或者衰减，以克服信号有限长度或者截断引起的频谱泄漏问题。它通过对信号进行加权来减小信号在截断点处的跃变，使得在频域上表现为光滑的过渡。
- **常见窗函数：**

- **矩形窗 (Rectangular Window)：**矩形窗函数是最简单的窗函数，其形式为：

$$w(n) = \begin{cases} 1, & 0 \leq n \leq N - 1 \\ 0, & \text{其他} \end{cases} \quad (1)$$

其中， $N$  是窗口长度。

- **汉宁窗 (Hanning Window)：**汉宁窗函数在频域上具有良好的性质，能够减小频谱泄漏。其形式为：

$$w(n) = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N - 1}\right) \quad (2)$$

- **汉明窗 (Hamming Window)：**汉明窗函数也是一种常用的窗函数，其形式为：

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N - 1}\right) \quad (3)$$

- **布莱克曼窗 (Blackman Window):** 布莱克曼窗函数在频域上的波动性较小，具有较高的主瓣宽度。其形式为：

$$w(n) = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right) \quad (4)$$

- **应用：**窗函数广泛应用于信号频谱分析、滤波器设计、谱估计等领域。在频谱分析中，窗函数用于减小频谱泄漏并提高频谱估计的准确性；在滤波器设计中，窗函数用于设计 FIR 滤波器的系数；在谱估计中，窗函数用于对信号进行加权以改善估计的性能。

窗函数在 MATLAB 中的应用非常广泛，可以使用 `rectwin`、`hann`、`hamming` 和 `blackman` 等函数来生成相应的窗函数。

```

1 % 生成矩形窗
2 N = 64; % 窗口长度
3 rect_window = rectwin(N);

4

5 % 生成汉宁窗
6 hann_window = hann(N);

7

8 % 生成汉明窗
9 hamming_window = hamming(N);

10

11 % 生成布莱克曼窗
12 blackman_window = blackman(N);

```

### 1.2.2 窗函数设计 FIR 滤波器

窗函数法的基本思想是将所需的滤波器频率响应形状乘以一个窗函数，以限制滤波器的频率响应在有限的范围内。这样，我们就可以得到一个有限长度的 FIR 滤波器，其频率响应是所需形状和窗函数的乘积。

- **设计步骤：**

1. **确定所需的频率响应：**首先，根据应用的需求确定所需的滤波器的频率响应形状，如低通、高通、带通或带阻等。
2. **选择窗函数：**根据所需的滤波器类型和性能要求，选择适当的窗函数。常用的窗函数有矩形窗、汉宁窗、汉明窗和布莱克曼窗等。

3. 窗函数与理想频率响应相乘：将所选窗函数与理想频率响应相乘，得到所需的滤波器的频率响应。
  4. 反变换：对得到的频率响应进行反变换，得到时域上的滤波器系数。
- **特点与优势：**窗函数法设计 FIR 滤波器的优势在于简单易用、计算方便、设计灵活。它不需要对频率响应进行优化迭代，而是直接根据所选的窗函数得到滤波器系数，因此适用于一些简单的滤波器设计任务。

窗函数法设计 FIR 滤波器在 MATLAB 中的实现也非常简单，只需要选择合适的窗函数和理想频率响应形状，然后将它们相乘即可得到滤波器的系数。

### 1.2.3 双线性变换法

双线性变换法是通过将连续时间系统的模拟滤波器转换为离散时间系统的数字滤波器。它利用了双线性变换的思想，将连续时间系统的频率响应映射到离散时间系统中，从而实现了模拟滤波器到数字滤波器的转换。

- **设计步骤：**
  1. 确定模拟滤波器的频率响应：首先，根据应用的需求选择合适的模拟滤波器，确定其频率响应  $H(s)$ 。
  2. 进行双线性变换：将模拟滤波器的频率响应  $H(s)$  进行双线性变换，得到相应的离散时间系统的数字滤波器的频率响应  $H(z)$ 。双线性变换公式如下：

$$H(z) = H(s) \Big|_{s=\frac{2}{T} \cdot \frac{z-1}{z+1}}$$

其中， $T$  是采样周期， $s$  是连续时间复频率， $z$  是离散时间复频率。

- 3. 频率归一化：对得到的数字滤波器的频率响应  $H(z)$  进行频率归一化，以确保滤波器的频率响应范围在单位圆内，即  $|z| = 1$ 。
- 4. 选择实现结构：根据设计要求和实际应用选择合适的数字滤波器实现结构，如直接型、级联型、二阶级联型等。

- **特点与优势：**双线性变换法设计的数字滤波器具有以下特点：
  - 易于实现：双线性变换法转换简单、计算方便，适用于各种类型的模拟滤波器；
  - 保留模拟滤波器的特性：双线性变换保留了模拟滤波器的幅度响应和相位响应的特性；

- 频率归一化：通过频率归一化，确保数字滤波器的频率响应范围在单位圆内，简化了设计和分析过程。

通过双线性变换法设计的数字滤波器具有较好的频率响应特性，能够满足各种实际应用的需求。在 MATLAB 中，可以利用 `bilinear` 函数进行双线性变换。

```

1 % 定义模拟滤波器的频率响应 H(s)，例如一个二阶低通滤波器
2 % H(s) = 1 / (s^2 + sqrt(2)*s + 1)
3 [num, den] = butter(2, 0.5, 'low', 's');
4
5 % 设计数字滤波器的采样频率和频率归一化
6 Fs = 1000; % 采样频率
7 wp = 0.4; % 截止频率
8 ws = 0.6; % 阻止频率
9 wpn = 2 * tan(wp * pi / 2); % 频率归一化
10 wsn = 2 * tan(ws * pi / 2); % 频率归一化
11
12 % 利用双线性变换进行频率响应的转换
13 [num_d, den_d] = bilinear(num, den, Fs, 'low');
14
15 % 输出数字滤波器的系数
16 disp('数字滤波器的分子系数：'); disp(num_d);
17 disp('数字滤波器的分母系数：'); disp(den_d);

```

这段代码首先定义了一个模拟滤波器的频率响应  $H(s)$ ，然后利用 MATLAB 中的 `bilinear` 函数进行双线性变换，将模拟滤波器转换为数字滤波器。最后输出了数字滤波器的系数。

#### 1.2.4 IIR 滤波器

IIR (Infinite Impulse Response, 无限脉冲响应) 滤波器是一种数字滤波器，其输出的当前样本值不仅依赖于输入信号的当前样本值，还依赖于之前的输出样本值。这种依赖关系导致了系统的反馈环路，使得系统具有无限脉冲响应。

IIR (Infinite Impulse Response, 无限脉冲响应) 滤波器采用了反馈结构，使得输出与输入之间存在递归关系。这种反馈结构赋予了 IIR 滤波器无限脉冲响应的特性，因此能够实现对信号的长期记忆和更为复杂的频率响应特性。通过这种特性，IIR 滤波器能够实现更为灵活的频率选择性，包括低通、高通、带通、带阻等各种类型的滤波器，而且在频域上能够实现更陡的过渡带宽。其设计也更具灵活性，可以通过调整滤波器的结构和参数来实现不同的频率响应特性，常见的设

计方法包括脉冲响应不变法和双线性变换法等。此外，由于其反馈结构，IIR 滤波器通常具有较低的延迟，适用于实时信号处理。但与之相对应的是，IIR 滤波器的设计和实现相对于 FIR 滤波器来说更加复杂。

IIR 滤波器在数字信号处理中具有重要的地位，它不仅可以实现对信号的频率选择性处理，还能够满足一些特殊应用的需求，如音频处理、通信系统等。在 MATLAB 中，可以使用 `butter`、`cheby1`、`ellip` 等函数来设计和实现 IIR 滤波器。

### 1.2.5 双线性变换法设计 IIR 滤波器

双线性变换法是一种常用于设计 IIR (Infinite Impulse Response, 无限脉冲响应) 数字滤波器的方法。它通过将模拟滤波器的频率响应映射到离散时间系统中来实现。以下是关于双线性变换法设计 IIR 滤波器的原理：

双线性变换法的基本原理是将连续时间系统的模拟滤波器转换为离散时间系统的数字滤波器。它利用了双线性变换的思想，将连续时间系统的频率响应映射到离散时间系统中，从而实现了模拟滤波器到数字滤波器的转换。

设计步骤如下：

- **确定模拟滤波器的频率响应：**首先，根据应用的需求选择合适的模拟滤波器，并确定其频率响应  $H(s)$ 。
- **进行双线性变换：**将模拟滤波器的频率响应  $H(s)$  进行双线性变换，得到相应的离散时间系统的数字滤波器的频率响应  $H(z)$ 。双线性变换的公式如下：

$$H(z) = H(s) \Big|_{s=\frac{2}{T} \cdot \frac{z-1}{z+1}}$$

其中， $T$  是采样周期， $s$  是连续时间复频率， $z$  是离散时间复频率。

- **频率归一化：**对得到的数字滤波器的频率响应  $H(z)$  进行频率归一化，以确保滤波器的频率响应范围在单位圆内，即  $|z| = 1$ 。
- **选择实现结构：**根据设计要求和实际应用选择合适的数字滤波器实现结构，如直接型、级联型、二阶级联型等。

通过双线性变换法设计的 IIR 数字滤波器具有较好的频率响应特性，能够满足各种实际应用的需求。在 MATLAB 中，可以使用 `bilinear` 函数来进行双线性变换。

```
1 % 定义模拟滤波器的传递函数 H(s)，例如一个二阶低通滤波器  
2 [num, den] = butter(2, 0.5, 'low', 's');
```

```

3
4 % 设计数字滤波器的采样频率和频率归一化
5 Fs = 1000; % 采样频率
6 wp = 0.4; % 截止频率
7 ws = 0.6; % 阻止频率
8 wpn = 2 * tan(wp * pi / 2); % 频率归一化
9 wsn = 2 * tan(ws * pi / 2); % 频率归一化
10
11 % 利用双线性变换进行模拟滤波器到数字滤波器的转换
12 [num_d, den_d] = bilinear(num, den, Fs);
13
14 % 输出数字滤波器的系数
15 disp('数字滤波器的分子系数: '); disp(num_d);
16 disp('数字滤波器的分母系数: '); disp(den_d);

```

## 1.3 实验内容

### 1.3.1 窗函数法设计 FIR 滤波器

利用窗函数法设计 FIR 滤波器的过程包括以下几个步骤：

- **定义变量：**通过 `wp1`、`wp2`、`ws1` 和 `ws2` 定义了通带和阻带的边界频率，`As` 定义了阻带衰减，`tr_width` 计算了过渡带宽，`M` 是滤波器的阶数，`n` 是时域序列。
- **生成理想冲激响应：**利用 `ideal_lp` 函数生成了频率响应为 `wc1` 和 `wc2` 的理想低通滤波器的冲激响应 `hd`。
- **生成窗函数：**使用 `blackman` 函数生成了长度为 `M` 的黑曼窗口 `w_bla`。
- **生成实际冲激响应：**将理想冲激响应 `hd` 与窗函数相乘，得到了实际冲激响应 `h`。
- **计算频率响应：**利用 `freqz_m` 函数计算了滤波器的频率响应，并保存了幅度响应到 `db` 中。
- **绘图：**使用 `subplot` 函数绘制了四个子图，分别展示了理想冲激响应、窗函数、实际冲激响应和幅度响应。

```

1 clc;
2 clear all;
3
4 wp1 = 0.45 * pi;
5 wp2 = 0.65 * pi;
6 ws1 = 0.3 * pi;
7 ws2 = 0.75 * pi;
8 As = 40;
9 tr_width = min((wp1 - ws1), (ws2 - wp2));
10 M = ceil(7 * pi / tr_width) + 1;
11 n = [0:1:M-1];
12 wc1 = (ws1 + wp1) / 2;
13 wc2 = (wp2 + ws2) / 2;
14 hd = ideal_lp(wc2, M) - ideal_lp(wc1, M);
15 w_bla = (blackman(M))';
16 h = hd .* w_bla;
17 [db, mag, pha, grd, w] = freqz_m(h, [1]);
18
19 subplot(2,2,1);
20 stem(n, hd);
21 axis([0 M-1 -0.4 0.5]);
22 title('理想冲激响应');
23 xlabel('n');
24 ylabel('hd(n)');
25
26 subplot(2,2,2);
27 stem(n, w_bla);
28 axis([0 M-1 0 1.1]);
29 title('Blackman 窗口');
30 xlabel('n');
31 ylabel('w(n)');
32
33 subplot(2,2,3);
34 stem(n, h);
35 axis([0 M-1 -0.4 0.5]);
36 title('实际冲激响应');
37 xlabel('n');

```

```

38 ylabel('h(n)');
39
40 subplot(2,2,4);
41 plot(w/pi, db);
42 axis([0 1 -150 10]);
43 grid;
44 title('幅度响应 (单位: dB)',);
45 xlabel('频率 (单位: pi)',);
46 ylabel('分贝',);

```

其中 `ideal_lp` 函数如下:

```

1 function hd = ideal_lp(wc,M)
2 % 理想低通滤波器计算
3
4 % -----
5 % [hd] = ideal_lp(wc,M)
6 % hd = 0 to M-1之间的理想脉冲响应
7 % wc = 截止频率(弧度)
8 % M = 理想滤波器的长度
9 %
10 alpha = (M-1)/2;
11 n = [0:1:(M-1)];
12 m = n - alpha + eps;
13 hd = sin(wc*m) ./ (pi*m);

```

其中 `freqz_m` 函数如下:

```

1 function [db,mag,pha,grd,w]=freqz_m(b,a)
2 [H,w]=freqz(b,a,1000,'whole');
3 H=(H(1:501));
4 w=(w(1:501));
5 mag=abs(H);
6 db=20*log10((mag+eps)/max(mag));
7 pha=angle(H);
8 grd=grpdelay(b,a,w);

```

可以得到以下结果:

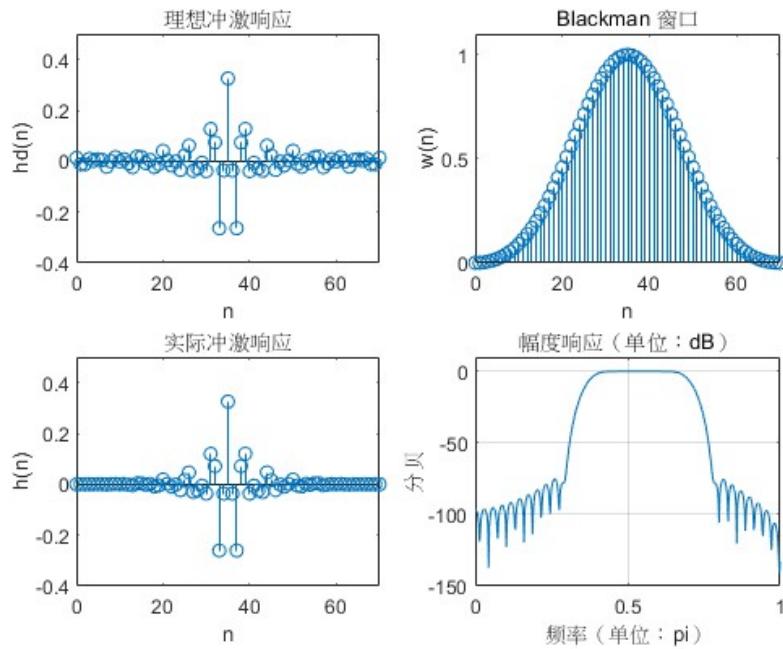


图 1：窗函数法设计 FIR 滤波器结果

在这个 FIR 滤波器设计的分析中，有以下几个关键变量和步骤：

- $h_d(n)$ : 展示了 FIR 带阻滤波器的理想冲激响应。理想冲激响应是通过相减两个理想低通滤波器的响应得到的，即

$$\text{ideal\_lp}(\text{wc2}, M) - \text{ideal\_lp}(\text{wc1}, M)$$

其中  $\text{wc1}$  和  $\text{wc2}$  是带通滤波器截止频率的中点。

- $w(n)$ : 理想响应通过与 Blackman 窗函数相乘，得到实际的冲激响应。Blackman 窗是用于减少旁瓣并改善主瓣平滑度的窗函数。
- $h(n)$ : 实际冲激响应，通过将理想冲激响应和 Blackman 窗函数相乘得到。
- $\text{db}$ : 显示了滤波器的频率响应，以分贝 (dB) 表示。图示显示在阻带频率处衰减显著，达到设计要求的 40 dB 衰减。

这种设计方法通过结合理想滤波器响应和窗函数的特性，成功地实现了一个性能优良的 FIR 带阻滤波器。

### 1.3.2 双线性变换法设计 IIR 滤波器

代码主要思路如下：

- 初始化和参数设置：

- `clear all; clc; close all;`: 清除 MATLAB 环境中的所有变量，清空命令窗口，并关闭所有图形窗口。
- `T = 2; fs = 1/T;`: 设置采样周期  $T$  为 2 秒，计算相应的采样频率  $fs$ 。

- 滤波器设计参数：

- `wp = [0.45*pi, 0.65*pi]; ws = [0.3*pi, 0.8*pi];`: 定义通带和阻带的边界频率（以  $\pi$  为单位的归一化频率）。
- `Wp = (2/T)*tan(wp/2); Ws = (2/T)*tan(ws/2);`: 使用预扭转公式将这些归一化频率转换为模拟频率，以适应模拟滤波器设计。
- `Ap = 1; As = 40;`: 分别设置通带最大衰减（1dB）和阻带最小衰减（40dB）。

- 巴特沃斯模拟滤波器设计：

- 使用 `buttord` 函数计算滤波器的最小阶数  $N$  和截止频率  $Wc$ 。
- 使用 `butter` 函数根据这些参数设计巴特沃斯带阻模拟滤波器，得到其系数  $B$  和  $A$ 。

- 频率响应计算：

- `Hs = freqs(B, A, W);`: 计算模拟滤波器的频率响应。
- `Hz = freqz(Bz, Az, W);`: 使用 `bilinear` 转换将模拟滤波器转换为数字滤波器，并计算其频率响应。

- 绘图：

- 绘制模拟滤波器和数字滤波器的幅值响应，分别为图的左侧和右侧。图表中使用 `grid on` 增加了网格线，便于观察频率响应的变化。

```

1 clear all;
2 clc
3 close all
4 T = 2;
5 fs = 1/T;
6 wp = [0.45*pi, 0.65*pi];
7 ws = [0.3*pi, 0.8*pi];
8 Wp = (2/T)*tan(wp/2);
9 Ws = (2/T)*tan(ws/2);

```

```

10 Ap = 1;
11 As = 40;
12
13 [N ,Wc] = buttord(Wp,Ws,Ap,As,'s');
14 [B ,A] = butter(N,Wc,'stop','s');
15 W = linspace(0,pi,400*pi);
16 Hs = freqs(B,A,W);
17 [Bz,Az] = bilinear(B,A,fs);
18 Hz = freqz(Bz,Az,W);
19
20 figure;
21 subplot(1,2,1);
22 plot(W/pi,abs(Hs));
23 xlabel('频率/Hz');ylabel('幅值');title('巴特沃斯模拟滤波器');
24 grid on;
24 subplot(1,2,2);
25 plot(W/pi,abs(Hz));
26 xlabel('频率/Hz');ylabel('幅值');title('巴特沃斯数字滤波器');
27 grid on;

```

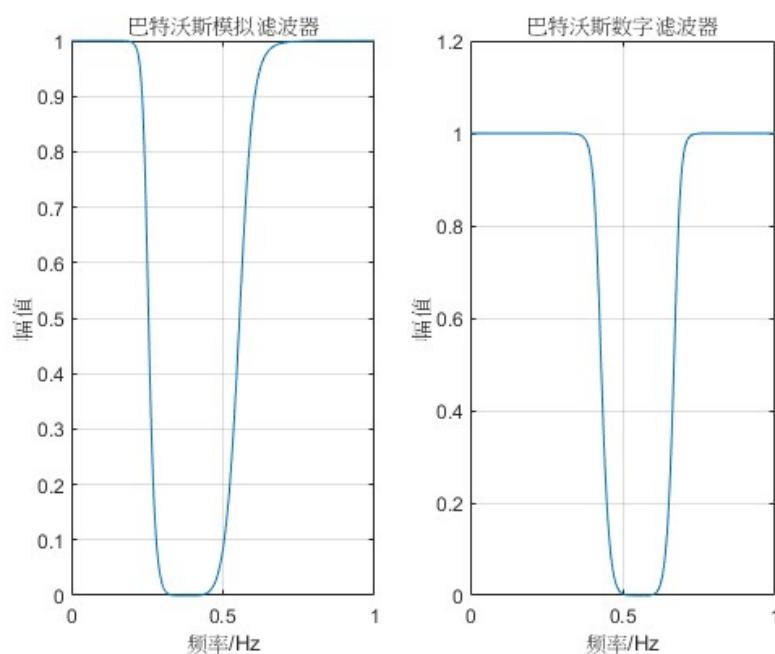


图 2: 双线性变换法设计 IIR 滤波器结果

## 2 课程设计 1（来源实验题目）：图像信号的卷积处理

### 2.1 课程设计目的

- 熟悉图像处理常用函数和方法
- 培养通过查阅文献解决问题的能力。
- 了解图像卷积、卷积核（算子）、常用的卷积核及其用途

### 2.2 课程设计要求

- **实现对图像进行不同卷积核的运算：**可以通过应用不同的卷积核（如锐化、模糊、边缘检测等）来对图像进行滤波处理，改变图像的特征和外观。
- **实现图像的加噪、低通滤波、高通滤波、边缘检测、高斯滤波等：**可以通过添加不同类型的噪声、应用低通、高通、边缘检测和高斯滤波器来改变图像的频域特性和视觉效果。
- **对图像进行特效处理：**可以实现各种图像特效处理，如模糊、马赛克、光照效果、色彩调整等，以增强图像的视觉吸引力和艺术效果。
- **对图像进行任意倍数放大和缩小、任意角度旋转：**可以实现对图像进行缩放和旋转的操作，以改变图像的尺寸和方向，满足不同应用场景下的需求。
- **为程序设计 UI 图形交互界面**

### 2.3 课程实验原理

#### 2.3.1 图像卷积

在图像数字信号处理中，卷积是一种常用的操作，用于处理图像以实现各种功能，如图像滤波、特征提取等。卷积操作可以看作是将两个函数（在图像处理中通常是一个图像和一个卷积核）进行加权求和的过程，其中卷积核是一个小的矩阵，用于对图像进行局部区域的加权处理。

给定一个图像  $I$  和一个卷积核  $K$ ，卷积操作可以用以下公式表示：

$$(I * K)(m, n) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(i, j) \cdot K(m - i, n - j) \quad (5)$$

其中， $(m, n)$  表示图像的像素位置， $I(i, j)$  表示图像在位置  $(i, j)$  处的像素值， $K(m - i, n - j)$  表示卷积核在位置  $(m - i, n - j)$  处的权重。

卷积操作的过程可以简单描述为：将卷积核与图像的每个像素位置进行对齐，然后将卷积核的每个元素与与之对应的图像像素进行乘法运算，最后将所有乘积结果相加得到输出图像的像素值。

在实际应用中，卷积操作常用于图像滤波，通过选择不同的卷积核可以实现不同的滤波效果，如平滑、锐化、边缘检测等。卷积操作也被广泛用于深度学习中的卷积神经网络（CNN），用于图像分类、目标检测等任务。

### 2.3.2 低通滤波器

低通滤波器是一种常用的图像滤波器，用于去除图像中高频部分，保留图像中的低频信息，从而实现图像的平滑和去噪。低通滤波器通过减小图像中像素值之间的变化来实现平滑效果，常用于图像预处理、去除噪声以及图像恢复等应用场景。

以下是一些常见的低通滤波器及其卷积核的示例：

- 均值滤波器：均值滤波器使用一个固定大小的卷积核，在卷积核的区域内取平均值作为输出像素的值。常见的均值滤波器包括 3x3 和 5x5 的卷积核，示例如下：

$$3 \times 3 \text{ 均值滤波器: } \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$5 \times 5 \text{ 均值滤波器: } \frac{1}{25} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

均值滤波器的效果是平滑图像，去除图像中的细节和噪声，但可能会导致图像变得模糊。

- 高斯滤波器：高斯滤波器使用高斯函数作为卷积核，以加权平均的方式对图像进行滤波。高斯滤波器在滤波过程中考虑了像素距离中心像素的距离，使得距离中心更近的像素具有更大的权重。常见的高斯滤波器示例如下：

$$3 \times 3 \text{ 高斯滤波器: } \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

$$5 \times 5 \text{ 高斯滤波器: } \frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

高斯滤波器具有平滑效果，同时保留了图像的细节，不会造成明显的模糊。

### 2.3.3 高通滤波器

高通滤波器是一种图像处理滤波器，用于增强图像中的高频部分，抑制图像中的低频部分，从而突出图像的边缘和细节特征。高通滤波器通常用于图像锐化、边缘检测等应用场景。

以下是一些常见的高通滤波器及其卷积核的示例：

- 拉普拉斯滤波器：拉普拉斯滤波器对图像进行二阶微分，可以提取图像中的边缘和细节信息。常见的拉普拉斯滤波器示例如下：

– 3x3 拉普拉斯滤波器：

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

– 5x5 拉普拉斯滤波器：

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 \\ -1 & -2 & 16 & -2 & -1 \\ 0 & -1 & -2 & -1 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

拉普拉斯滤波器能够增强图像中的边缘和细节，但同时也会增加图像中的噪声。

- Sobel 滤波器：Sobel 滤波器是一种常用的边缘检测滤波器，通过计算图像的梯度来识别图像中的边缘。Sobel 滤波器有水平方向和垂直方向两种形式，分别用于检测图像中的水平和垂直边缘。常见的 Sobel 滤波器示例如下：

– 水平方向 Sobel 滤波器:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

– 垂直方向 Sobel 滤波器:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Sobel 滤波器能够有效地检测图像中的边缘，提取出图像中的细节信息。

#### 2.3.4 匹配滤波边缘

匹配滤波边缘是一种常用于图像处理中的边缘检测方法。其基本思想是利用模板或滤波器与图像进行卷积，通过与预先定义的边缘模式或特征模式进行匹配，从而识别图像中的边缘或特征。

常见的滤波器包括以下几种：

- **Prewitt 滤波器:**

– 水平方向 Prewitt 滤波器:

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

– 垂直方向 Prewitt 滤波器:

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

- **Roberts 滤波器:**

– Roberts 交叉梯度滤波器:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

– Roberts 十字梯度滤波器:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

匹配滤波边缘方法适用于各种图像处理任务，如目标检测、图像分割、特征提取等。通过调整不同的模板或滤波器，可以实现对不同形状、大小和方向的边缘进行有效的检测和提取。

### 2.3.5 边缘检测

边缘检测是图像处理中常用的技术，用于检测图像中的边缘和轮廓。边缘通常是图像中像素灰度值变化较大的地方，表示图像中物体的边界或区域之间的转换。

常见的边缘检测算法基于卷积操作，通过在图像上应用特定的卷积核来检测边缘。以下是一些常见的边缘检测卷积核及其效果：

- **Sobel 滤波器：**

- 水平方向 Sobel 滤波器：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- 垂直方向 Sobel 滤波器：

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Prewitt 滤波器：**

- 水平方向 Prewitt 滤波器：

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

- 垂直方向 Prewitt 滤波器：

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

- **Roberts 滤波器：**

– Roberts 十字梯度滤波器:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

– Roberts 交叉梯度滤波器:

$$\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

## 2.4 课程设计内容

### 2.4.1 图像进行不同卷积核运算

根据实验要求，常用卷积核（算子、模板）如下图所示：

#### 1. 低通滤波器

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \frac{1}{9}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \frac{1}{10}$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * \frac{1}{16}$$

⋮

#### 2. 高通滤波器

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -2 & 1 \\ -2 & 5 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

#### 3. 平移和差分边缘检测

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

#### 4. 匹配滤波边缘检测

$$\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

#### 5. 边缘检测

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

⋮

#### 6. 梯度方向边缘检测

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ 1 & -2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{bmatrix}$$

⋮

图 3: 常用卷积核 (算子、模板)

处理的图片如下图：

按照上述模板，可以编写出以下代码：

```
1 import cv2
2 import numpy as np
3
4 # 例如，这是一个使用卷积核1的函数
5 def kernel_lowpass_1(image):
6     kernel = np.array([[1, 1, 1],
7                         [1, 1, 1],
8                         [1, 1, 1]]) / 9
9     return cv2.filter2D(image, -1, kernel)
10
11 def kernel_lowpass_2(image):
12     kernel = np.array([[1, 1, 1],
13                         [1, 2, 1],
14                         [1, 1, 1]]) / 10
15     return cv2.filter2D(image, -1, kernel)
16
17 def kernel_lowpass_3(image):
18     kernel = np.array([[1, 2, 1],
19                         [2, 4, 2],
20                         [1, 2, 1]]) / 16
21     return cv2.filter2D(image, -1, kernel)
22
23 def kernel_highpass_1(image):
24     kernel = np.array([[0, -1, 0],
25                         [-1, 5, -1],
26                         [0, -1, 0]])
27     return cv2.filter2D(image, -1, kernel)
28
29 def kernel_highpass_2(image):
30     kernel = np.array([[-1, -1, -1],
31                         [-1, 9, -1],
32                         [-1, -1, -1]])
33     return cv2.filter2D(image, -1, kernel)
34
35 def kernel_highpass_3(image):
```

```
36     kernel = np.array([[1, -2, 1],
37                         [-2, 5, -2],
38                         [1, -2, 1]])
39     return cv2.filter2D(image, -1, kernel)
40
41 def kernel_move_detection_horizontal(image):
42     kernel = np.array([[0, 0, 0],
43                         [-1, 1, 0],
44                         [0, 0, 0]])
45     return cv2.filter2D(image, -1, kernel)
46
47 def kernel_move_detection_vertical(image):
48     kernel = np.array([[0, -1, 0],
49                         [0, 1, 0],
50                         [0, 0, 0]])
51     return cv2.filter2D(image, -1, kernel)
52
53 def kernel_move_detection_diagonal(image):
54     kernel = np.array([[-1, 0, 0],
55                         [0, 1, 0],
56                         [0, 0, 0]])
57     return cv2.filter2D(image, -1, kernel)
58
59 def kernel_horizontal_match_detection(image):
60     kernel = np.array([[-1, -1, -1, -1, -1],
61                         [0, 0, 0, 0, 0],
62                         [1, 1, 1, 1, 1]])
63     return cv2.filter2D(image, -1, kernel)
64
65 def kernel_vertical_match_detection(image):
66     kernel = np.array([[-1, 0, 1],
67                         [-1, 0, 1],
68                         [-1, 0, 1]])
69     return cv2.filter2D(image, -1, kernel)
70
71 def kernel_edge_1(image):
72     kernel = np.array([[-1, 0, -1],
73                         [0, 4, 0],
```

```
74                         [-1,  0, -1]]))
75     return cv2.filter2D(image, -1, kernel)
76
77 def kernel_edge_2(image):
78     kernel = np.array([[[-1, -1, -1],
79                         [-1,  8, -1],
80                         [-1, -1, -1]]])
81     return cv2.filter2D(image, -1, kernel)
82
83 def kernel_edge_3(image):
84     kernel = np.array([[[-1, -1, -1],
85                         [-1,  9, -1],
86                         [-1, -1, -1]]])
87     return cv2.filter2D(image, -1, kernel)
88
89 def kernel_edge_4(image):
90     kernel = np.array([[[-1, -2,  1],
91                         [-2,  4, -2],
92                         [1, -2,  1]]])
93     return cv2.filter2D(image, -1, kernel)
94
95 def kernel_gradient_1(image):
96     kernel = np.array([[[-1,  1,  1],
97                         [1, -2,  1],
98                         [-1, -1, -1]]])
99     return cv2.filter2D(image, -1, kernel)
100
101 def kernel_gradient_2(image):
102     kernel = np.array([[[-1,  1,  1],
103                         [-1, -2,  1],
104                         [-1, -1,  1]]])
105     return cv2.filter2D(image, -1, kernel)
106
107 def kernel_gradient_3(image):
108     kernel = np.array([[[-1,  1,  1],
109                         [-1, -2,  1],
110                         [-1,  1,  1]]])
111     return cv2.filter2D(image, -1, kernel)
```

```
112
113 def kernel_gradient_4(image):
114     kernel = np.array([[-1, -1, 1],
115                         [-1, -2, 1],
116                         [1, 1, 1]])
117     return cv2.filter2D(image, -1, kernel)
118
119 def kernel_gradient_5(image):
120     kernel = np.array([[-1, -1, -1],
121                         [1, -2, 1],
122                         [1, 1, 1]])
123     return cv2.filter2D(image, -1, kernel)
124
125 def kernel_gradient_6(image):
126     kernel = np.array([[1, -1, -1],
127                         [1, -2, -1],
128                         [1, 1, 1]])
129     return cv2.filter2D(image, -1, kernel)
130
131 def kernel_gradient_7(image):
132     kernel = np.array([[1, 1, -1],
133                         [1, -2, -1],
134                         [1, 1, -1]])
135     return cv2.filter2D(image, -1, kernel)
136
137 def kernel_gradient_8(image):
138     kernel = np.array([[1, 1, 1],
139                         [1, -2, -1],
140                         [1, -1, -1]])
141     return cv2.filter2D(image, -1, kernel)
```

得到结果如下：低通滤波器保留了图像中的低频信息，滤除了高频细节。图像看起来更加平滑，边缘细节被模糊化，适用于去除噪声和平滑图像。高通滤波器保留了图像中的高频信息，滤除了低频成分。图像的边缘和细节变得更加明显，但可能会引入噪声，适用于边缘检测和锐化图像。水平边缘检测算子主要检测水平方向上的边缘。图像中水平变化的部分被强调出来，而其他方向的边缘信息较弱。竖直边缘检测算子主要检测竖直方向上的边缘。图像中竖直变化的部分被强调出来，而其他方向的边缘信息较弱。

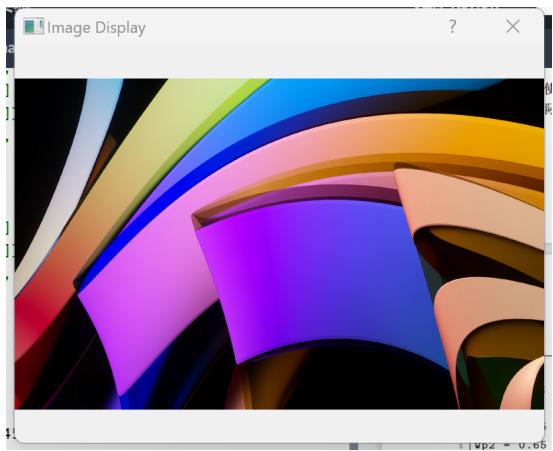


图 4: 低通滤波器算子得到结果



图 5: 高通滤波器算子得到结果

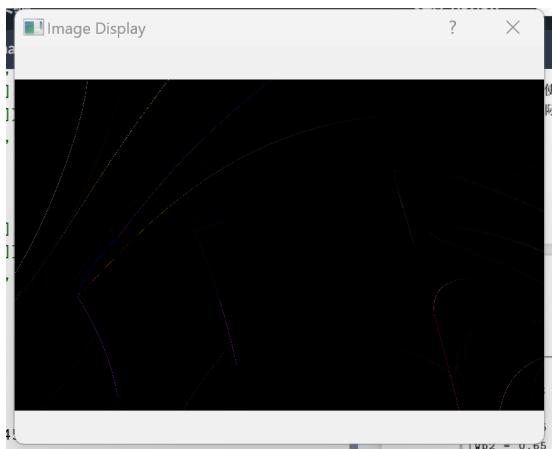


图 6: 水平边缘检测算子得到结果

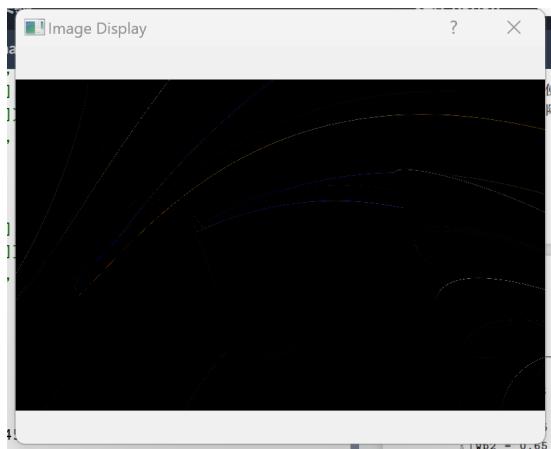


图 7: 竖直边缘检测算子得到结果

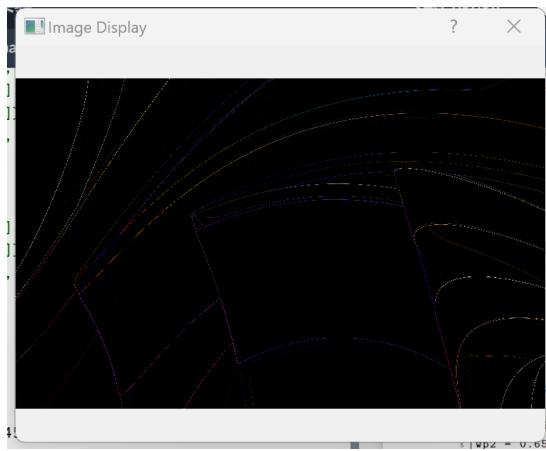


图 8: 边缘检测算子 1 得到结果

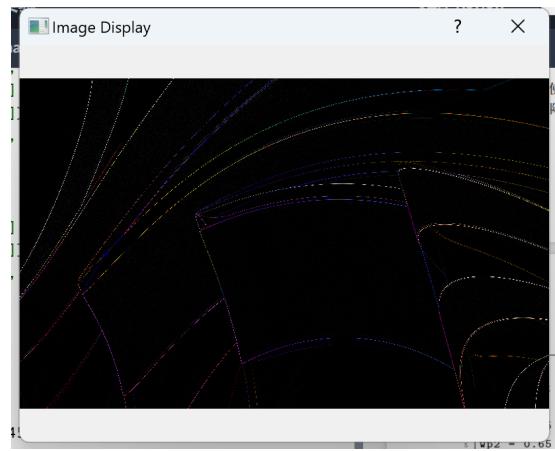


图 9: 边缘检测算子 2 得到结果



图 10: 边缘检测算子 3 得到结果

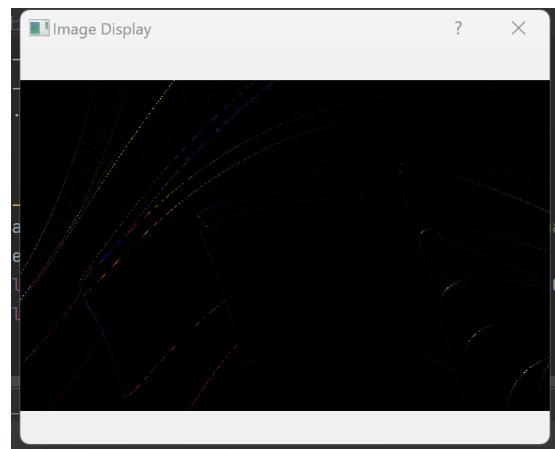


图 11: 边缘检测算子 4 得到结果

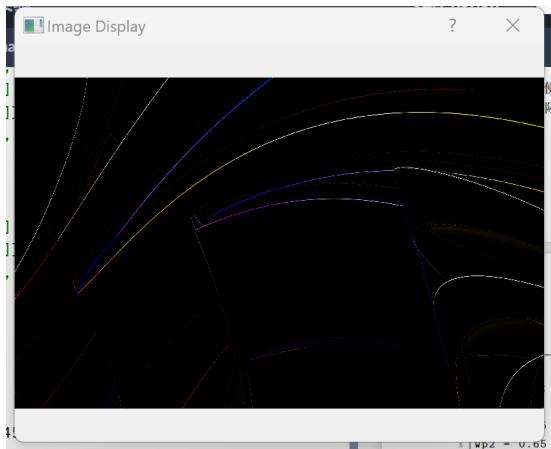


图 12: 梯度方向边缘检测算子 1 得到结果

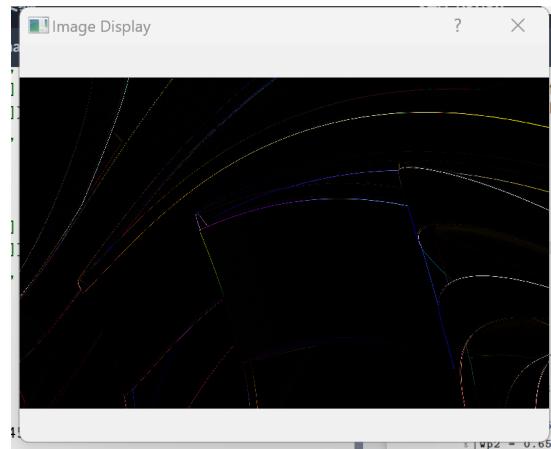


图 13: 梯度方向边缘检测算子 2 得到结果

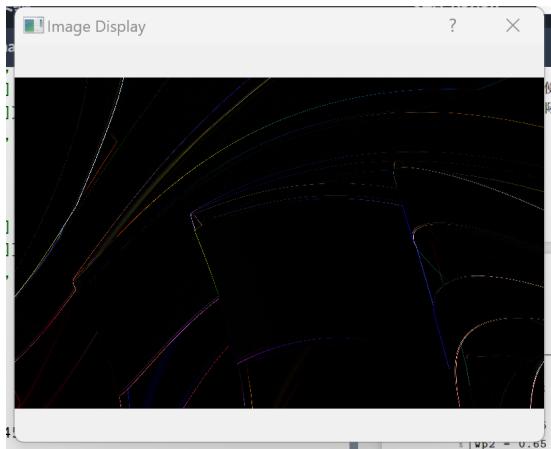


图 14: 梯度方向边缘检测算子 3 得到结果

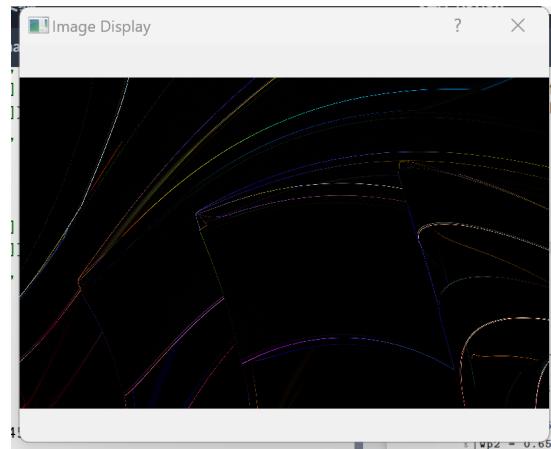


图 15: 梯度方向边缘检测算子 4 得到结果

## 2.4.2 图像基本处理

图像的基本处理包括以下几个内容：

- 增加高斯噪音、椒盐噪声、泊松噪声
- 运用低通滤波、高通滤波
- 检测图像边缘（Canny 算法）
- 加入高斯模糊、中值模糊
- 锐化图片
- 增强对比度

根据上述描述，可以得到以下代码：

```
1 import cv2
2 import numpy as np
3
4 def add_gaussian_noise(image, mean=0, var=0.01):
5     row, col, ch = image.shape
6     sigma = var ** 0.5
7     gauss = np.random.normal(mean, sigma, (row, col, ch))
8     noisy = image + gauss.reshape(row, col, ch)
9     return np.clip(noisy, 0, 255).astype(np.uint8)
10
11 def add_salt_and_pepper_noise(image, s_vs_p=0.5, amount
12     ↪ =0.004):
13     row, col, ch = image.shape
14     out = np.copy(image)
15     # Salt mode
16     num_salt = np.ceil(amount * image.size * s_vs_p)
17     coords = [np.random.randint(0, i - 1, int(num_salt))
18               for i in image.shape]
19     out[tuple(coords)] = 1
20
21     # Pepper mode
22     num_pepper = np.ceil(amount * image.size * (1. - s_vs_p))
23     coords = [np.random.randint(0, i - 1, int(num_pepper))
24               for i in image.shape]
```

```
24     out[tuple(coords)] = 0
25
26
27 def add_poisson_noise(image):
28     vals = len(np.unique(image))
29     vals = 2 ** np.ceil(np.log2(vals))
30     noisy = np.random.poisson(image * vals) / float(vals)
31     return np.clip(noisy, 0, 255).astype(np.uint8)
32
33
34 def apply_low_pass_filter(image):
35     kernel = np.ones((5, 5), np.float32) / 25
36     return cv2.filter2D(image, -1, kernel)
37
38 def apply_high_pass_filter(image):
39     kernel = np.array([[0, -1, 0],
40                       [-1, 4, -1],
41                       [0, -1, 0]])
42     return cv2.filter2D(image, -1, kernel)
43
44 def detect_edges(image):
45     # 使用 Canny 算法检测边缘
46     edges = cv2.Canny(image, 100, 200)
47     # 创建一个彩色图像用于显示边缘
48     edge_colored = cv2.cvtColor(edges, cv2.COLOR_GRAY2BGR)
49     return edge_colored
50
51
52 def apply_gaussian_blur(image, ksize=5):
53     # 高斯模糊
54     return cv2.GaussianBlur(image, (ksize, ksize), 0)
55
56 def sharpen_image(image):
57     kernel = np.array([[-1, -1, -1],
58                       [-1, 9, -1],
59                       [-1, -1, -1]])
60     return cv2.filter2D(image, -1, kernel)
61
```

```
62 def apply_median_blur(image, ksize=5):  
63     # 中值模糊  
64     return cv2.medianBlur(image, ksize)  
65  
66 def adjust_contrast(image, contrast=1.5):  
67     # 增强对比度  
68     f = 131 * (contrast - 1)  
69     alpha_c = f / 127 + 1  
70     gamma_c = 127 * (1 - alpha_c)  
71     image = cv2.addWeighted(image, alpha_c, image, 0, gamma_c  
    ↪ )  
72     return image
```

得到的结果如下图所示：

图 16 中的高斯噪音处理在图像中引入了随机分布的亮度变化，使图像显得较为“脏”，模拟了图像传感器的噪音情况。图 17 的高斯模糊对图像进行了平滑处理，使边缘和细节变得模糊，效果自然和平滑。图 18 的椒盐噪声处理在图像中引入了随机分布的黑白点，用于模拟图像传输过程中产生的噪音。图 19 中的锐化处理使图像的边缘和细节更加清晰，整体对比度有所提高，但过度锐化可能会引入噪音。图 20 的边缘检测算子提取了图像中的边缘信息，结果图像仅显示边缘轮廓，其他部分变为黑色，这些边缘检测算法包括 Sobel、Prewitt 和 Canny 等，通过计算图像梯度找到边缘位置。图 21 的增强对比度处理使图像的亮部更亮，暗部更暗，从而提高图像的对比度，使图像更鲜明和有层次感，但过度处理可能会丢失部分细节。这些处理方法展示了不同算法在图像分析和计算机视觉中的广泛应用，噪声和模糊处理展示了图像在传输和采集过程中可能遇到的干扰，而锐化和对比度增强则是常用的图像增强技术，边缘检测则用于提取图像的轮廓信息。

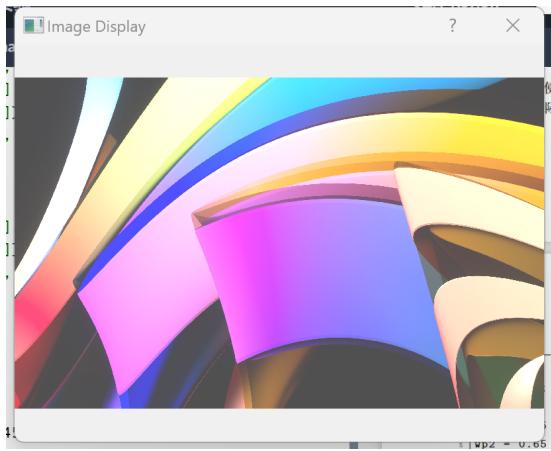


图 16: 高斯噪音得到结果



图 17: 高斯模糊得到结果

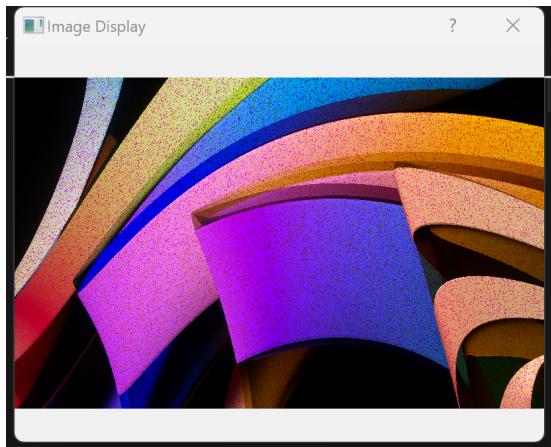


图 18: 椒盐噪声得到结果

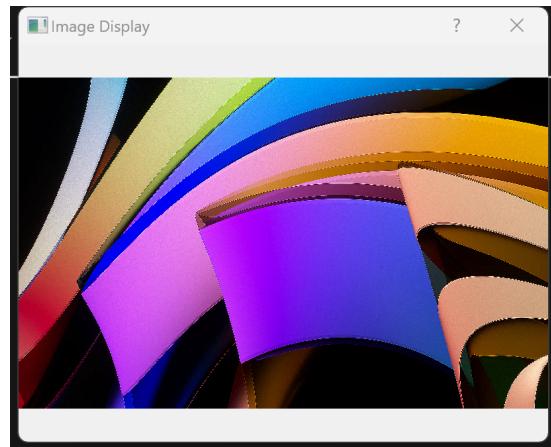


图 19: 锐化处理得到结果



图 20: 边缘检测得到结果

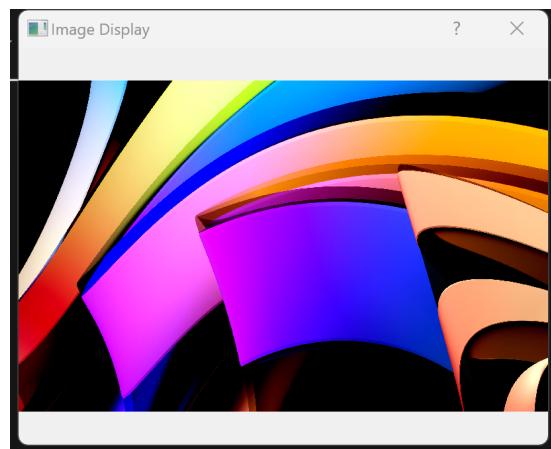


图 21: 增强对比度得到结果

### 2.4.3 图像特效处理

图像的基本处理包括以下几个内容：

- 马赛克处理
- 运用低通滤波、高通滤波
- 检测图像边缘（Canny 算法）
- 加入高斯模糊、中值模糊
- 锐化图片
- 增强对比度

```
1 import cv2
2 import numpy as np
3
4 def apply_mosaic(image, block_size=10):
5     """
6         Apply a mosaic (pixelation) effect to the image.
7         :param image: Input image
8         :param block_size: Size of each block
9         :return: Image with mosaic effect
10    """
11    for i in range(0, image.shape[0], block_size):
12        for j in range(0, image.shape[1], block_size):
13            image[i:i+block_size, j:j+block_size] = np.median(
14                image[i:i+block_size, j:j+block_size], axis
15                =(0, 1)).astype(np.uint8)
16
17    return image
18
19
20 def apply_vignette(image, strength=0.5):
21     """
22         Apply a vignette effect to the image.
23         :param image: Input image
24         :param strength: Strength of the vignette effect
25         :return: Image with vignette effect
26     """
27
28    rows, cols = image.shape[:2]
```

```

24     kernel_x = cv2.getGaussianKernel(cols, cols * strength)
25     kernel_y = cv2.getGaussianKernel(rows, rows * strength)
26     kernel = kernel_y * kernel_x.T
27     mask = 255 * kernel / np.linalg.norm(kernel)
28     mask = mask.astype(np.uint8)
29     vignette = np.dstack([mask] * 3)
30     return cv2.addWeighted(image, 1, vignette, -1, 0)
31
32 def apply_thermal_effect(image):
33     """
34     Simulate a thermal camera effect.
35     :param image: Input image
36     :return: Image with thermal effect
37     """
38     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
39     heatmap = cv2.applyColorMap(gray, cv2.COLORMAP_JET)
40     return heatmap
41
42 def apply_sketch_effect(image, sigma_s=60, sigma_r=0.07):
43     """
44     Convert an image to a sketch-like effect using edge-
45         ↪ preserving filtering.
46     :param image: Input image
47     :param sigma_s: Range between 0 and 200.
48     :param sigma_r: Range between 0 and 1.
49     :return: Sketch-like effect image
50     """
51
52     return cv2.stylization(image, sigma_s=sigma_s, sigma_r=
53         ↪ sigma_r)
54
55 def apply_cartoon_effect(image):
56     """
57     Apply a cartoon effect to an image.
58     :param image: Input image
59     :return: Cartoon-styled image
60     """
61
62     # Use bilateral filter for edge-aware smoothing.
63     smooth = cv2.bilateralFilter(image, d=9, sigmaColor=75,

```

```

    ↪ sigmaSpace=75)

60 # Convert to grayscale and apply median blur
61 gray = cv2.cvtColor(smooth, cv2.COLOR_BGR2GRAY)
62 gray = cv2.medianBlur(gray, 7)
63 # Detect edges and enhance them
64 edges = cv2.adaptiveThreshold(gray, 255, cv2.
    ↪ ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, blockSize
    ↪ =9, C=2)
65 # Combine the edges and the smoothed image
66 cartoon = cv2.bitwise_and(smooth, smooth, mask=edges)
67 return cartoon

```

得到的结果如下图：

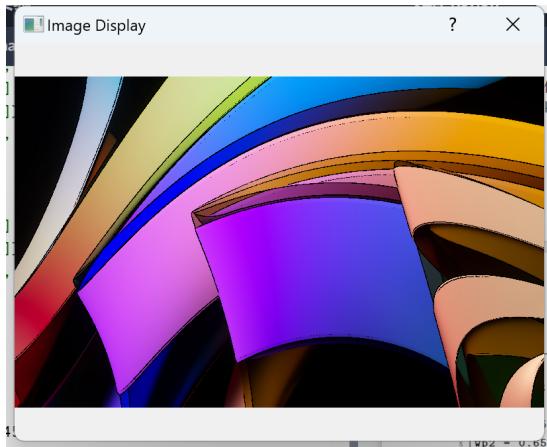


图 22：图像卡通化处理得到结果

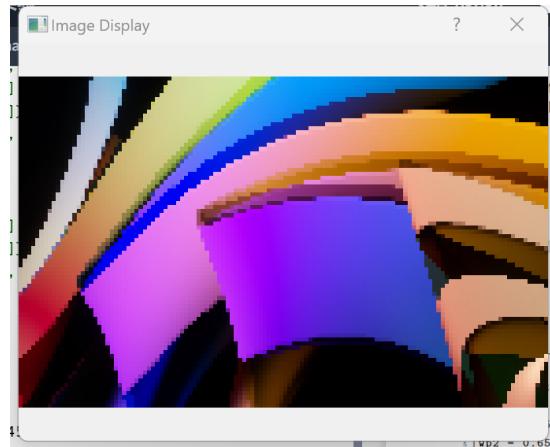


图 23：图像马赛克处理得到结果

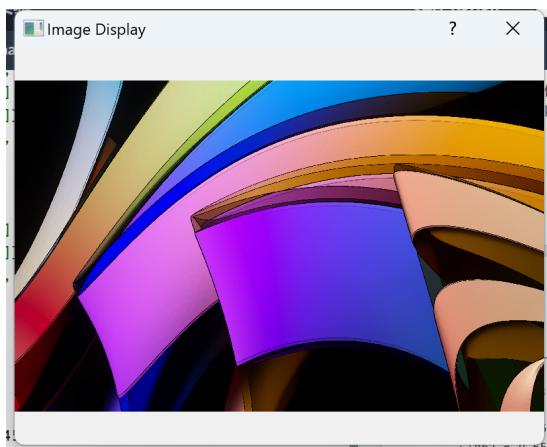


图 24：图像素描化处理得到结果

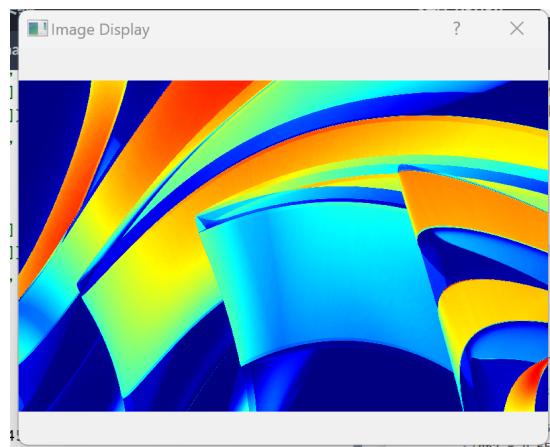


图 25：图像热度图处理得到结果

图 22 展示了图像卡通化处理的结果，该处理通过减少颜色数量和强调边缘，使图像看起来像卡通图。图 23 中的图像马赛克处理使图像被分割成小块，每块内

的像素值相同，类似于隐私保护的模糊效果。图 24 的图像素描化处理将图像转化为黑白素描效果，保留了主要的边缘和轮廓，使其看起来像手绘素描。图 25 的图像热度图处理将图像的颜色映射到热度图上，用不同颜色表示不同的强度值，从而突出图像中的强度变化。这些处理方法展示了图像处理的多种技术，卡通化和素描化处理常用于艺术效果和滤镜，马赛克处理用于隐私保护，而热度图处理则用于数据可视化和分析。

#### 2.4.4 图像变换处理

图像变换处理包括以下两个内容：

- 图像任意倍数放大缩小
- 图像任意角度旋转

根据上述描述，可以得到以下代码：

```
1 import cv2
2 import numpy as np
3
4
5 def resize_image(image, scale_factor):
6     """
7         Resize an image by a given scale factor.
8         :param image: Input image.
9         :param scale_factor: Scale factor for resizing, e.g., 2
10            ↪ for doubling the size, 0.5 for halving.
11         :return: Resized image.
12     """
13
14     new_dimensions = (int(image.shape[1] * scale_factor), int
15                        ↪ (image.shape[0] * scale_factor))
16     resized_image = cv2.resize(image, new_dimensions,
17                                ↪ interpolation=cv2.INTER_LINEAR)
18
19     return resized_image
20
21
22
23 def rotate_image(image, angle, scale=1.0):
24     """
25         Rotate an image by a given angle.
26         :param image: Input image.
27         :param angle: Angle in degrees to rotate the image.
28            ↪ Positive values mean counter-clockwise rotation.
29         :param scale: Scale factor during rotation. Default is
30            ↪ 1.0.
31         :return: Rotated image.
32     """
33
34     center = (image.shape[1] // 2, image.shape[0] // 2)
```

```
26     rotation_matrix = cv2.getRotationMatrix2D(center, angle,
27         ↪ scale)
28     new_dimensions = (image.shape[1], image.shape[0])
29     rotated_image = cv2.warpAffine(image, rotation_matrix,
30         ↪ new_dimensions)
31     return rotated_image
```

得到的结果如下图：

图 26 展示了图像缩小处理的结果，图像整体被缩小，显示出原图像的全貌，但细节可能会丢失。图 27 展示了图像放大处理的结果，图像整体被放大，细节变得更加清晰，但可能会引入模糊或像素化问题。图 28 和图 29 分别展示了图像旋转处理的结果 1 和结果 2，图像经过不同角度的旋转后，呈现出旋转后的视角效果。这些处理方法展示了图像几何变换的多种技术，缩放和旋转是常见的图像变换操作，应用于图像调整、对齐和视角变换等场景。

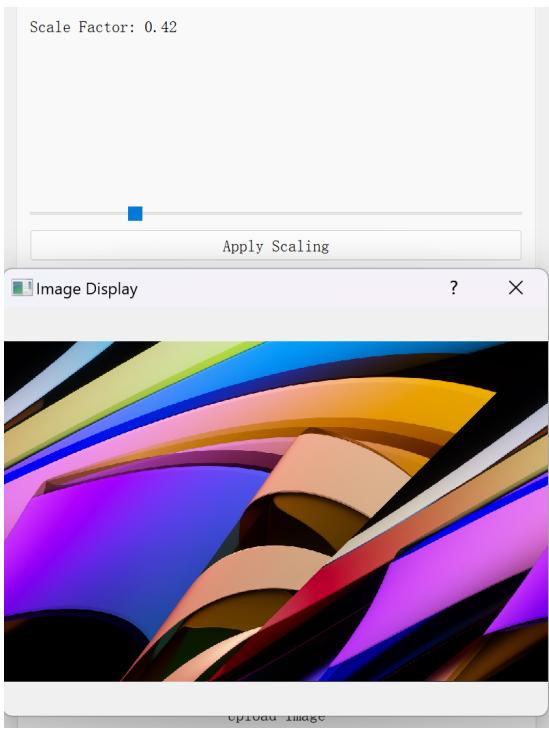


图 26: 图像缩小处理得到结果

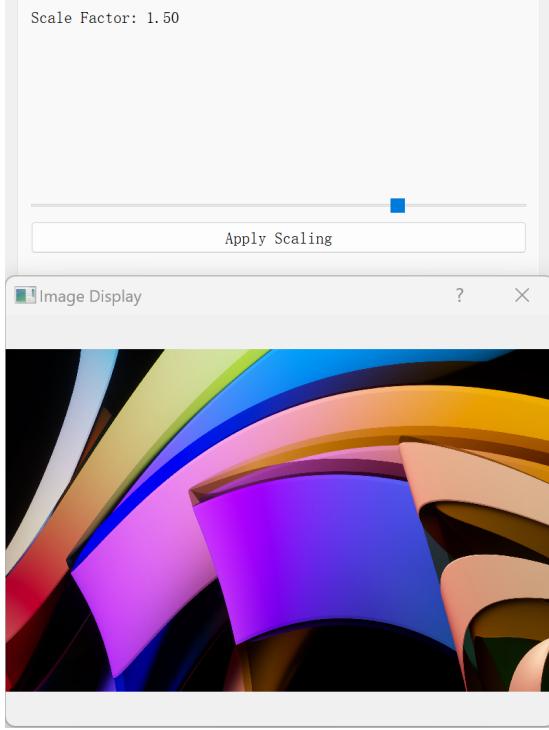


图 27: 图像放大处理得到结果

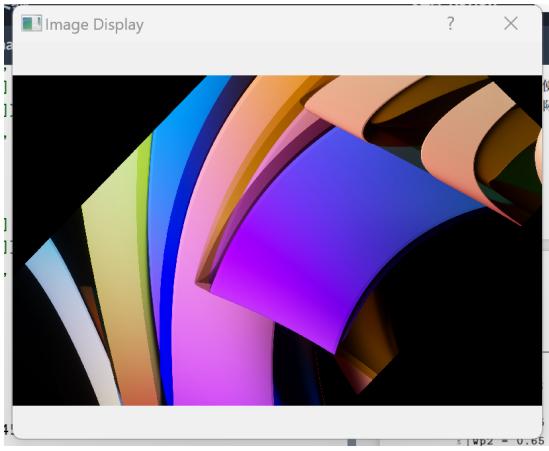


图 28: 图像旋转处理得到结果 1



图 29: 图像旋转处理得到结果 2

## 2.4.5 GUI 用户交互页面

这段代码是一个使用 PyQt5 创建的图像处理应用程序的完整框架，它整合了图像卷积操作、添加噪声、应用图像效果以及图像变换等功能。下面是各部分功能的详细说明：

- **类定义和初始化**

- `ImageDisplayWindow`: 一个专门用于显示处理后的图像的对话框类。它包含一个 `QLabel` 用于显示图像。
- `ImageProcessorApp`: 主窗口类，负责创建和管理应用程序的用户界面和功能。

- **主窗口布局与控件**

- 在 `ImageProcessorApp` 的构造函数中，创建了四个标签页，分别对应不同的处理功能：卷积操作、图像处理、图像效果和图像变换。
- 使用 `QTabWidget` 来管理这些标签页，允许用户在不同功能间切换。
- 添加了一个"Upload Image" 按钮，允许用户上传本地图像文件进行处理。

- **图像上传与显示**

- `upload_image` 方法使用 `QFileDialog` 来打开文件选择对话框，用户选择图像后，图像将被加载并转换成 RGB 格式，然后显示在 `ImageDisplayWindow` 中。

- **图像处理控件的设置**

- 每个处理页 (`conv_page`, `process_page`, `effects_page`, `transform_page`) 都通过特定的方法 (`create_convolution_controls`, `create_process_controls`, `create_effects_controls`, `create_transformation_controls`) 来设置，这些方法添加具体的处理按钮和控件。

- **应用滤镜的通用方法**

- `apply_filter` 方法接受一个处理函数，应用该函数到当前加载的图像，并显示处理结果。

- **运行应用**

- 在文件的最底部，创建了 QApplication 的实例，并显示主窗口，启动事件循环。

这个应用程序通过图形用户界面提供了丰富的图像处理功能，用户可以通过直观的操作来查看各种图像处理效果。这不仅有助于图像处理的学习和研究，也适用于实际的图像处理任务。

```
1 from PyQt5 import QtWidgets, QtGui, QtCore
2 from PyQt5.QtWidgets import QMainWindow, QLabel, QTabWidget,
3     ↪ QVBoxLayout, QPushButton, QFileDialog, QWidget, QDialog,
4     ↪ QLineEdit
5
6 from PyQt5.QtWidgets import QSlider
7 from PyQt5.QtCore import Qt
8 import cv2
9 import sys
10
11 # 引入处理模块
12 import convolution_kernels
13 import image_process
14 import image_effects
15 import image_transformations
16
17 class ImageDisplayWindow(QDialog):
18     def __init__(self, parent=None):
19         super().__init__(parent)
20         self.setWindowTitle("Image Display")
21         self.setGeometry(150, 150, 800, 600)
22         self.image_label = QLabel(self)
23         self.image_label.resize(800, 600)
24         self.image_label.setAlignment(QtCore.Qt.AlignCenter)
25
26     def display_image(self, img):
27         img = QtGui.QImage(img.data, img.shape[1], img.shape
28             ↪ [0], QtGui.QImage.Format_RGB888).rgbSwapped()
29         pixmap = QtGui.QPixmap.fromImage(img)
30         self.image_label.setPixmap(pixmap.scaled(self.
31             ↪ image_label.size(), QtCore.Qt.KeepAspectRatio))
32         self.show()
```

```
29 class ImageProcessorApp(QMainWindow):
30     def __init__(self):
31         super().__init__()
32         self.setWindowTitle("Image Processing App")
33         self.setGeometry(100, 100, 800, 600)
34
35         self.image_display_window = ImageDisplayWindow(self)
36
37         self.image = None
38
39         # 创建标签页
40         tab_widget = QTabWidget(self)
41         self.conv_page = QWidget()
42         self.process_page = QWidget() # 更改变量名
43         self.effects_page = QWidget()
44         self.transform_page = QWidget()
45
46         tab_widget.addTab(self.conv_page, "Convolution
47             ↪ Operations")
48         tab_widget.addTab(self.process_page, "Image Process")
49             ↪ # 更改标签页标题
50         tab_widget.addTab(self.effects_page, "Image Effects")
51         tab_widget.addTab(self.transform_page, "Image
52             ↪ Transformations")
53
54         self.create_convolution_controls()
55         self.create_process_controls() # 更改方法名称
56         self.create_effects_controls()
57         self.create_transformation_controls()
58
59         layout = QVBoxLayout()
60         layout.addWidget(tab_widget)
61
62         container = QWidget()
63         container.setLayout(layout)
64         self.setCentralWidget(container)
65
66         # 添加上传图片按钮
```

```
64     upload_button = QPushButton("Upload Image", self)
65     upload_button.clicked.connect(self.upload_image)
66     layout.addWidget(upload_button)
67
68     def upload_image(self):
69         file_name, _ = QFileDialog.getOpenFileName(self, "
70             ↪ Open Image", "", "Image Files (*.png *.jpg *.
71             ↪ jpeg *.bmp)")
72         if file_name:
73             self.image = cv2.cvtColor(cv2.imread(file_name),
74                 ↪ cv2.COLOR_BGR2RGB)
75             self.image_display_window.display_image(self.
76                 ↪ image)
77
78     def apply_filter(self, filter_func):
79         if self.image is not None:
80             processed_image = filter_func(self.image)
81             self.image_display_window.display_image(
82                 ↪ processed_image)
83
84     def create_convolution_controls(self):
85         layout = QVBoxLayout()
86         # 创建低通滤波器按钮
87         lp1_button = QPushButton("Apply Lowpass Filter 1")
88         lp1_button.clicked.connect(lambda: self.apply_filter(
89             ↪ convolution_kernels.kernel_lowpass_1))
90         layout.addWidget(lp1_button)
91
92         lp2_button = QPushButton("Apply Lowpass Filter 2")
93         lp2_button.clicked.connect(lambda: self.apply_filter(
94             ↪ convolution_kernels.kernel_lowpass_2))
95         layout.addWidget(lp2_button)
96
97         lp3_button = QPushButton("Apply Lowpass Filter 3")
98         lp3_button.clicked.connect(lambda: self.apply_filter(
99             ↪ convolution_kernels.kernel_lowpass_3))
100        layout.addWidget(lp3_button)
```

```
94 # 创建高通滤波器按钮
95 hp1_button = QPushButton("Apply Highpass Filter 1")
96 hp1_button.clicked.connect(lambda: self.apply_filter(
97     convolution_kernels.kernel_highpass_1))
98 layout.addWidget(hp1_button)
99
100 hp2_button = QPushButton("Apply Highpass Filter 2")
101 hp2_button.clicked.connect(lambda: self.apply_filter(
102     convolution_kernels.kernel_highpass_2))
103 layout.addWidget(hp2_button)
104
105 hp3_button = QPushButton("Apply Highpass Filter 3")
106 hp3_button.clicked.connect(lambda: self.apply_filter(
107     convolution_kernels.kernel_highpass_3))
108 layout.addWidget(hp3_button)
109
110 # 创建移动检测滤波器按钮
111 move_horizontal_button = QPushButton("Horizontal
112     ↪ Movement Detection")
113 move_horizontal_button.clicked.connect(
114     lambda: self.apply_filter(convolution_kernels.
115         ↪ kernel_move_detection_horizontal))
116 layout.addWidget(move_horizontal_button)
117
118 move_vertical_button = QPushButton("Vertical Movement
119     ↪ Detection")
120 move_vertical_button.clicked.connect(
121     lambda: self.apply_filter(convolution_kernels.
122         ↪ kernel_move_detection_vertical))
123 layout.addWidget(move_vertical_button)
124
125 move_diagonal_button = QPushButton("Diagonal Movement
126     ↪ Detection")
127 move_diagonal_button.clicked.connect(
128     lambda: self.apply_filter(convolution_kernels.
129         ↪ kernel_move_detection_diagonal))
130 layout.addWidget(move_diagonal_button)
```

```
123 # 创建边缘检测按钮
124 for i in range(1, 5):
125     button = QPushButton(f"Apply Edge Detection {i}")
126     button.clicked.connect(lambda _, k=i: self.
127         ↪ apply_filter(getattr(convolution_kernels, f'
128             ↪ kernel_edge_{k})))
129
130     layout.addWidget(button)
131
132 # 创建梯度检测按钮
133 for i in range(1, 9):
134     button = QPushButton(f"Apply Gradient {i}")
135     button.clicked.connect(
136         lambda _, k=i: self.apply_filter(getattr(
137             ↪ convolution_kernels, f'kernel_gradient_{
138                 ↪ k}')))
139
140     layout.addWidget(button)
141
142     self.conv_page.setLayout(layout)
143
144 def create_process_controls(self):
145     layout = QVBoxLayout()
146
147     # 添加 Gaussian 噪声按钮和均值滑动条
148     gauss_noise_button = QPushButton("Add Gaussian Noise"
149         ↪ )
150
151     # 均值滑动条
152     gauss_mean_slider = QSlider(Qt.Horizontal)
153     gauss_mean_slider.setMinimum(-255) # 最小值, 示例为
154         ↪ 允许负均值
155     gauss_mean_slider.setMaximum(255) # 最大值
156     gauss_mean_slider.setValue(0) # 默认值为0
157     gauss_mean_label = QLabel("Mean: 0")
158     gauss_mean_slider.valueChanged.connect(
159         lambda value: gauss_mean_label.setText(f"Mean: {
160             ↪ value}"))
161
162     layout.addWidget(gauss_mean_label)
```

```
154     layout.addWidget(gauss_mean_slider)
155
156     # 方差设置为默认值，不提供滑动条
157     default_variance = 0.1  # 默认方差值
158
159     gauss_noise_button.clicked.connect(lambda: self.
160         ↪ apply_filter(
161             lambda img: image_process.add_gaussian_noise(img,
162                 ↪ gauss_mean_slider.value(), default_variance
163                 ↪ )
164         ))
165     layout.addWidget(gauss_noise_button)
166
167     # 添加 Salt and Pepper 噪声按钮和滑动条
168     sp_noise_button = QPushButton("Add Salt and Pepper
169         ↪ Noise")
170
171     sp_amount_slider = QSlider(Qt.Horizontal)
172     sp_amount_slider.setMinimum(1)
173     sp_amount_slider.setMaximum(100)  # 代表比例从0.01到1
174         ↪ .00
175     sp_amount_slider.setValue(4)  # 默认值为0.04
176     sp_amount_slider.valueChanged.connect(lambda value:
177         ↪ sp_amount_label.setText(f"Amount: {value /
178             ↪ 1000:.3f}"))
179     sp_amount_label = QLabel("Amount: 0.004")
180     layout.addWidget(sp_amount_label)
181     layout.addWidget(sp_amount_slider)
182
183     sp_noise_button.clicked.connect(lambda: self.
184         ↪ apply_filter(
185             lambda img: image_process.
186                 ↪ add_salt_and_pepper_noise(img, 0.5,
187                     ↪ sp_amount_slider.value() / 1000)
188         ))
189     layout.addWidget(sp_noise_button)
190
191     # 添加其他图像处理操作的按钮
192     self.add_other_image_processing_buttons(layout)
```

```
182  
183     self.process_page.setLayout(layout)  
184  
185     def add_other_image_processing_buttons(self, layout):  
186         # 创建其他图像处理按钮并添加到布局  
187         low_pass_button = QPushButton("Apply Low Pass Filter"  
188             ↪ )  
189         high_pass_button = QPushButton("Apply High Pass  
190             ↪ Filter")  
191         edge_detect_button = QPushButton("Detect Edges")  
192         gaussian_blur_button = QPushButton("Apply Gaussian  
193             ↪ Blur")  
194         sharpen_button = QPushButton("Sharpen Image")  
195         median_blur_button = QPushButton("Apply Median Blur")  
196         adjust_contrast_button = QPushButton("Adjust Contrast  
197             ↪ ")  
198  
199         low_pass_button.clicked.connect(lambda: self.  
200             ↪ apply_filter(image_process.apply_low_pass_filter  
201             ↪ ))  
202         high_pass_button.clicked.connect(lambda: self.  
203             ↪ apply_filter(image_process.  
204                 ↪ apply_high_pass_filter))  
205         edge_detect_button.clicked.connect(lambda: self.  
206             ↪ apply_filter(image_process.detect_edges))  
207         gaussian_blur_button.clicked.connect(  
208             lambda: self.apply_filter(lambda img:  
209                 ↪ image_process.apply_gaussian_blur(img, 5)))  
210         sharpen_button.clicked.connect(lambda: self.  
211             ↪ apply_filter(image_process.sharpen_image))  
212         median_blur_button.clicked.connect(  
213             lambda: self.apply_filter(lambda img:  
214                 ↪ image_process.apply_median_blur(img, 5)))  
215         adjust_contrast_button.clicked.connect(  
216             lambda: self.apply_filter(lambda img:  
217                 ↪ image_process.adjust_contrast(img, 1.5)))  
218  
219         layout.addWidget(low_pass_button)
```

```
207     layout.addWidget(high_pass_button)
208     layout.addWidget(edge_detect_button)
209     layout.addWidget(gaussian_blur_button)
210     layout.addWidget(sharpen_button)
211     layout.addWidget(median_blur_button)
212     layout.addWidget(adjust_contrast_button)
213
214     def create_effects_controls(self):
215         layout = QVBoxLayout()
216
217         # 添加马赛克效果按钮和滑动条
218         mosaic_button = QPushButton("Apply Mosaic Effect")
219         mosaic_slider = QSlider(Qt.Horizontal)
220         mosaic_slider.setMinimum(5)    # 最小块大小
221         mosaic_slider.setMaximum(50)   # 最大块大小
222         mosaic_slider.setValue(10)    # 默认块大小为 10
223         mosaic_label = QLabel("Block Size: 10")
224         mosaic_slider.valueChanged.connect(lambda value:
225             ↪ mosaic_label.setText(f"Block Size: {value}"))
226         mosaic_button.clicked.connect(lambda: self.
227             ↪ apply_filter(
228                 lambda img: image_effects.apply_mosaic(img,
229                     ↪ mosaic_slider.value())))
230
231         layout.addWidget(mosaic_label)
232         layout.addWidget(mosaic_slider)
233         layout.addWidget(mosaic_button)
234
235         # 添加晕影效果按钮和滑动条
236         vignette_button = QPushButton("Apply Vignette Effect"
237             ↪ )
238         vignette_slider = QSlider(Qt.Horizontal)
239         vignette_slider.setMinimum(10)   # 最小强度
240         vignette_slider.setMaximum(100)  # 最大强度，对应于
241             ↪ 0.1 到 1.0 的范围
242         vignette_slider.setValue(50)    # 默认强度为 0.5
243         vignette_label = QLabel("Vignette Strength: 0.5")
244         vignette_slider.valueChanged.connect(
```

```
240         lambda value: vignette_label.setText(f"Vignette  
241             ↪ Strength: {value / 100:.2f}"))
242     )
243     vignette_button.clicked.connect(lambda: self.
244         ↪ apply_filter(
245             lambda img: image_effects.apply_vignette(img,
246                 ↪ vignette_slider.value() / 100)
247         ))
248     layout.addWidget(vignette_label)
249     layout.addWidget(vignette_slider)
250     layout.addWidget(vignette_button)

251     # 添加其他效果的按钮
252     thermal_button = QPushButton("Apply Thermal Effect")
253     thermal_button.clicked.connect(lambda: self.
254         ↪ apply_filter(image_effects.apply_thermal_effect)
255         ↪ )
256     layout.addWidget(thermal_button)

257     sketch_button = QPushButton("Apply Sketch Effect")
258     sketch_button.clicked.connect(lambda: self.
259         ↪ apply_filter(
260             lambda img: image_effects.apply_sketch_effect(img
261                 ↪ , 60, 0.07) # 默认参数
262         ))
263     layout.addWidget(sketch_button)

264     cartoon_button = QPushButton("Apply Cartoon Effect")
265     cartoon_button.clicked.connect(lambda: self.
266         ↪ apply_filter(image_effects.apply_cartoon_effect)
267         ↪ )
268     layout.addWidget(cartoon_button)

269     self.effects_page.setLayout(layout)

270

271     def create_transformation_controls(self):
272         layout = QVBoxLayout()
```

```
269  
270     # 添加缩放控制滑动条和按钮  
271     scale_label = QLabel("Scale Factor: 1.0", self)  
272     scale_slider = QSlider(Qt.Horizontal, self)  
273     scale_slider.setMinimum(1)  
274     scale_slider.setMaximum(200)    # 设定范围, 例如 0.1 到  
275         ↪ 2.0 倍, 内部表示为 10 到 200  
276     scale_slider.setValue(100)    # 默认值为 1.0, 内部表示  
277         ↪ 为 100  
278     scale_slider.valueChanged.connect(lambda value:  
279         ↪ scale_label.setText(f"Scale Factor: {value /  
280             ↪ 100:.2f}"))  
281  
282     scale_button = QPushButton("Apply Scaling")  
283     scale_button.clicked.connect(lambda: self.  
284         ↪ apply_filter(  
285             lambda img: image_transformations.resize_image(  
286                 ↪ img, scale_slider.value() / 100)  
287         ))  
288     layout.addWidget(scale_label)  
289     layout.addWidget(scale_slider)  
290     layout.addWidget(scale_button)  
291  
292     # 添加旋转控制滑动条和按钮  
293     rotate_label = QLabel("Rotate Angle: 0°", self)  
294     rotate_slider = QSlider(Qt.Horizontal, self)  
295     rotate_slider.setMinimum(0)  
296     rotate_slider.setMaximum(360)    # 允许旋转的角度从 0  
         ↪ 到 360 度  
297     rotate_slider.setValue(0)    # 默认值为 0 度  
298     rotate_slider.valueChanged.connect(lambda value:  
299         ↪ rotate_label.setText(f"Rotate Angle: {value}°"))  
300  
301     rotate_button = QPushButton("Apply Rotation")  
302     rotate_button.clicked.connect(lambda: self.  
303         ↪ apply_filter(  
304             lambda img: image_transformations.rotate_image(  
305                 ↪ img, rotate_slider.value())))
```

```
297     ))
298     layout.addWidget(rotate_label)
299     layout.addWidget(rotate_slider)
300     layout.addWidget(rotate_button)
301
302     self.transform_page.setLayout(layout)
303
304
305 if __name__ == "__main__":
306     app = QtWidgets.QApplication(sys.argv)
307     main_window = ImageProcessorApp()
308     main_window.show()
309     sys.exit(app.exec_())
```

## 2.5 课程设计结果

### 2.5.1 GUI 交互界面效果

GUI 展示效果如下：

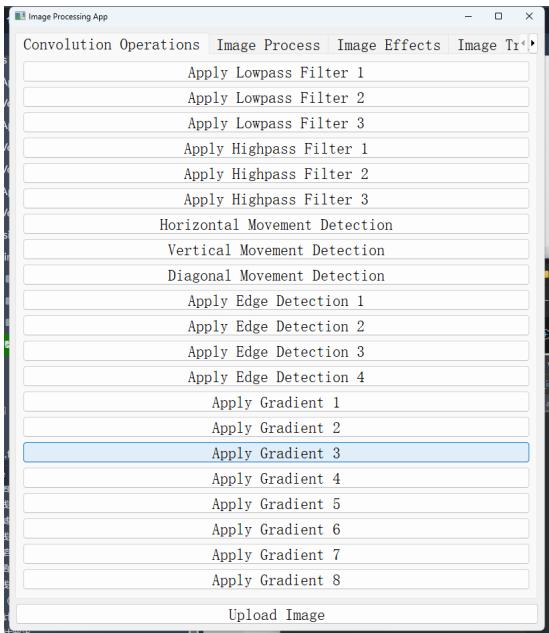


图 30：图像卷积处理选择页面

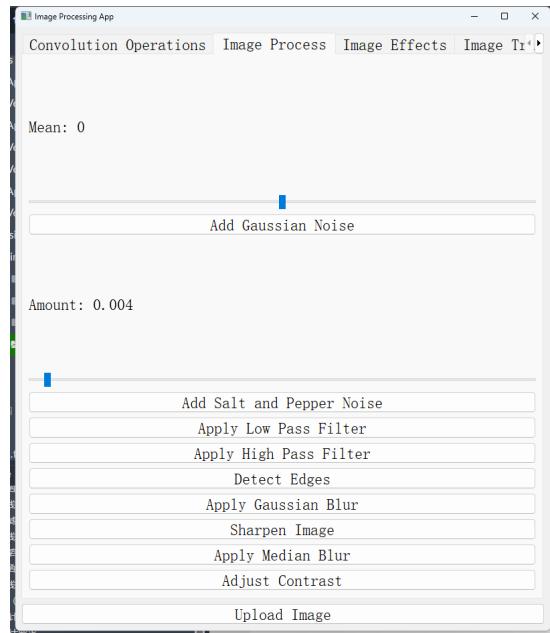


图 31：图像基本处理选择页面

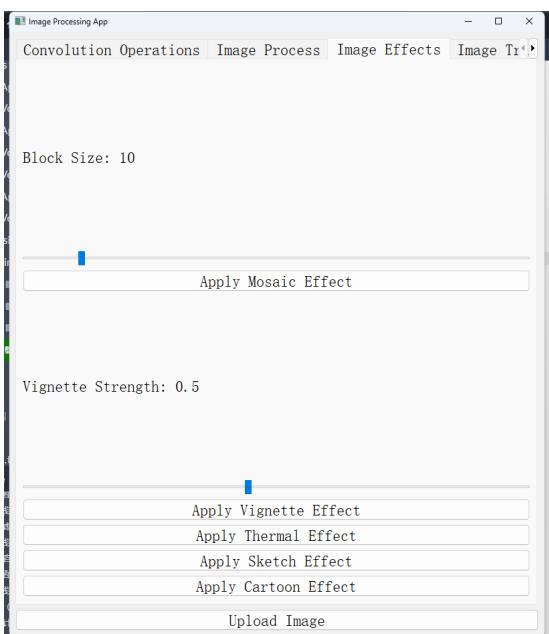


图 32：图像特效处理选择页面

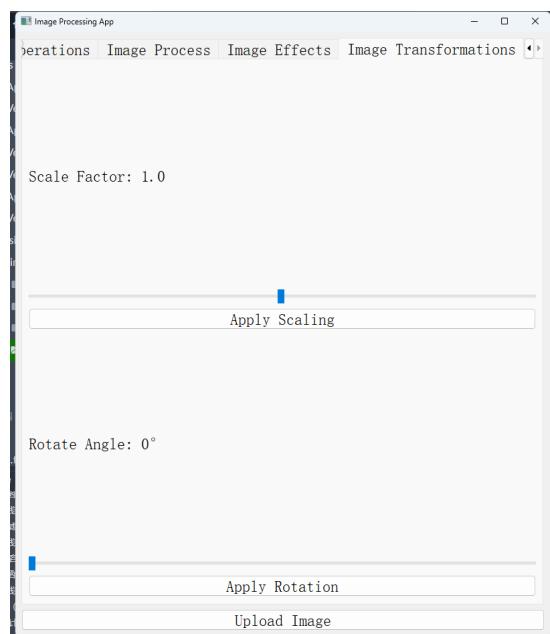


图 33：图像变换处理选择页面

在点击 Upload Image 上传图片后，点击处理按钮（可以选择拖动滑动条来选择参数）后图片会显示在 Display 窗口中。

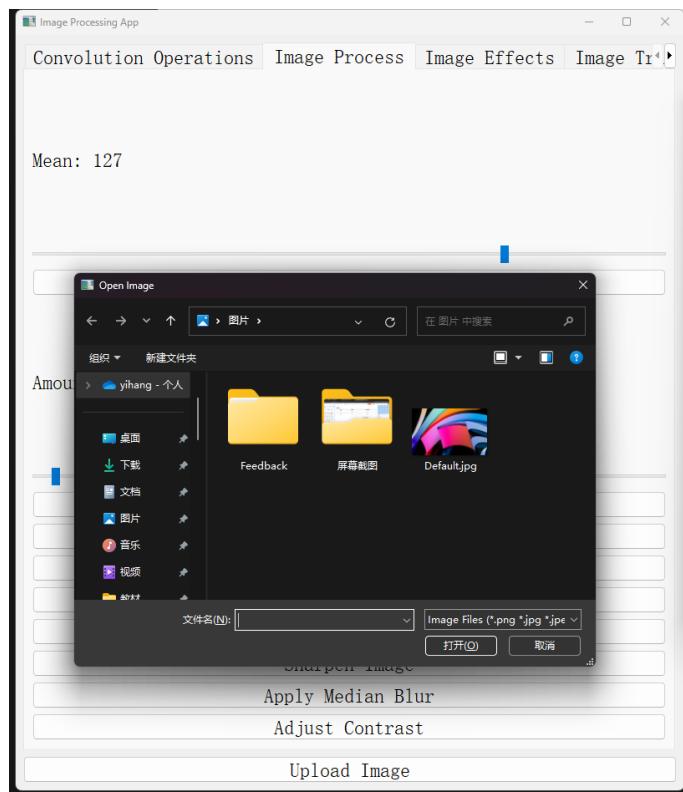


图 34: 上传图片选择页面

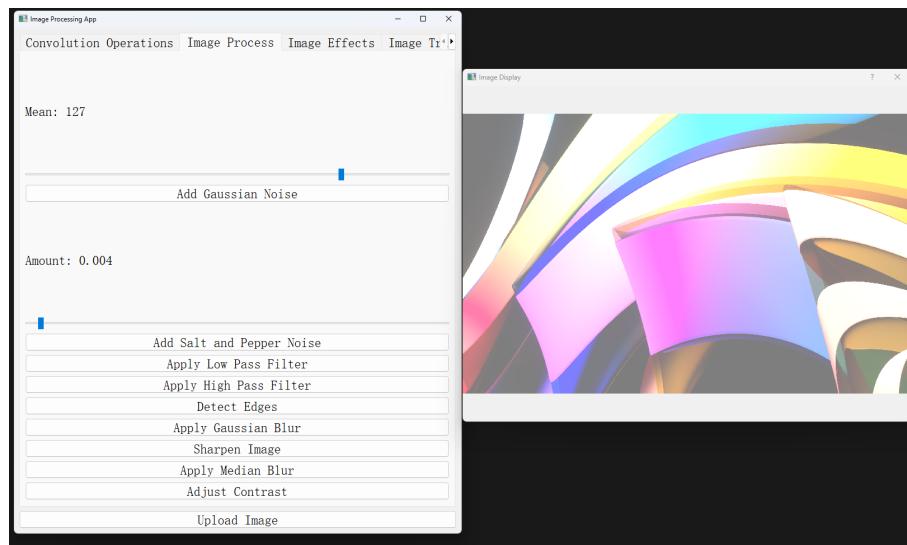


图 35: 拖动滑动条调整函数参数传入页面

### 3 课程设计 2（自拟）:fNIRS 信号的处理

#### 3.1 选题背景

##### 3.1.1 fNIRS 信号

功能近红外光谱 (functional Near-Infrared Spectroscopy, fNIRS) 是一种用于监测大脑活动的非侵入式成像技术。fNIRS 通过测量大脑皮层血氧浓度的变化，反映神经活动的动态变化。当神经元活动增加时，局部的血流量和血氧水平也会相应增加，这一现象被称为血氧水平依赖 (Blood Oxygen Level Dependent, BOLD) 效应。fNIRS 主要测量两种血红蛋白浓度的变化：氧合血红蛋白 (HbO) 和去氧血红蛋白 (HbR)。

##### 3.1.2 fNIRS 测量原理

fNIRS 设备通过发射近红外光 (700-900 纳米波长) 穿透头皮和颅骨到达大脑皮层，再通过探测器接收反射光。由于 HbO 和 HbR 对近红外光的吸收率不同，可以通过分析反射光的强度变化来计算 HbO 和 HbR 的浓度变化。典型的 fNIRS 系统包括多个光源和探测器对，这些对以特定的排列方式放置在头皮上，覆盖感兴趣的大脑区域。

##### 3.1.3 应用场景

fNIRS 广泛应用于认知神经科学、心理学和医学研究中。例如，研究者可以使用 fNIRS 监测大脑在执行特定任务（如语言处理、注意力任务）时的活动变化，或评估不同刺激对大脑活动的影响。此外，fNIRS 也用于临床诊断和康复训练，如评估脑损伤患者的功能恢复情况。

#### 3.2 课程设计目的

- 光密度转换：将原始光强度信号转换为光密度信号。
- 贝尔-兰伯特定律：应用贝尔-兰伯特定律将光密度信号转换为 HbO 和 HbR 浓度变化。
- 滤波处理：应用带通滤波器去除低频和高频噪声。
- 基线校准：对信号进行基线校准以去除基线漂移。

### 3.3 课程设计原理

#### 3.3.1 光密度转换

光密度转换是指将光强度信号转换为光密度信号的过程。在近红外光谱(NIRS)测量中，光强度信号表示通过组织的光的强度。然而，这个信号受到多种因素的影响，包括组织厚度、吸收和散射等。为了更好地分析这些信号，我们通常需要将光强度信号转换为光密度信号。

光密度(Optical Density, OD)定义为光强度的对数比，公式如下：

$$OD = \log_{10} \left( \frac{I_0}{I_t} \right)$$

其中：

- $I_0$  是入射光强度。
- $I_t$  是透射光强度。

光密度的变化与血氧浓度的变化有直接关系，因为血氧饱和度会影响组织对不同波长光的吸收。因此，通过计算光密度，我们可以更准确地估计血氧浓度。

#### 3.3.2 血氧浓度转换

血氧浓度转换是将光密度信号转换为氧合血红蛋白(HbO)和脱氧血红蛋白(HbR)浓度的过程。这一步骤通常使用贝尔-兰伯特定律(Beer-Lambert Law) [2]来完成。

贝尔-兰伯特定律描述了光在均匀介质中传播时的衰减行为。对于多重波长近红外光谱，定律可以表示为：

$$OD(\lambda) = \epsilon_{HbO}(\lambda) \cdot C_{HbO} \cdot L + \epsilon_{HbR}(\lambda) \cdot C_{HbR} \cdot L$$

其中：

- $OD(\lambda)$  是波长  $\lambda$  处的光密度。
- $\epsilon_{HbO}(\lambda)$  和  $\epsilon_{HbR}(\lambda)$  分别是氧合血红蛋白和脱氧血红蛋白在波长  $\lambda$  处的摩尔吸收系数。
- $C_{HbO}$  和  $C_{HbR}$  分别是氧合血红蛋白和脱氧血红蛋白的浓度。
- $L$  是光在组织中的传播路径长度。

通过测量不同波长下的光密度，我们可以构建一个方程组来求解  $C_{HbO}$  和  $C_{HbR}$ 。这需要对每个波长的摩尔吸收系数进行校准和测量，并假设光传播路径长度  $L$  是已知的或可近似的。

### 3.3.3 带通滤波处理

带通滤波器是一种允许特定频率范围内的信号通过，同时抑制该范围之外频率成分的滤波器。对于 fNIRS 信号处理，带通滤波器的应用主要是为了去除不需要的噪声，同时保留有用的生理信号。

**fNIRS 信号中的频率成分** fNIRS 信号中包含多种频率成分：

- **低频成分** ( $<0.01$  Hz)：通常与缓慢的生理变化（如体动、呼吸等）相关。
- **中频成分** (0.01 Hz - 0.1 Hz)：主要反映血氧浓度变化，代表大脑活动相关的信号。
- **高频成分** ( $>0.1$  Hz)：通常为高频噪声，如电器噪声和高频生理噪声。

在 fNIRS 信号处理中，感兴趣的频率范围主要是 0.01 Hz 到 0.1 Hz 左右，因为这个范围内的信号主要反映大脑的血氧变化，是我们希望保留的有用信号。

带通滤波器通过允许特定频率范围内的信号通过，同时衰减该范围之外的信号来工作。其原理可以用以下公式表示：

$$y(t) = \int_{-\infty}^{\infty} x(\tau)h(t - \tau)d\tau \quad (6)$$

其中：

- $y(t)$  是滤波后的输出信号。
- $x(t)$  是输入信号（即原始 fNIRS 信号）。
- $h(t)$  是滤波器的冲激响应。

带通滤波器的冲激响应  $h(t)$  设计成只允许中频成分通过，而对低频和高频成分进行衰减。

在本设计中，带通滤波器的实现如下：

```
1 if isinstance(self.filter_para, list) \
2 and len(self.filter_para) >= 4 \
3 and isinstance(self.filter_para[0], (int, float)) \
4 and isinstance(self.filter_para[1], (int, float)):
5     raw_haemo = raw_haemo.filter(self.filter_para[0], self.
6         ↪ filter_para[1],
7             h_trans_bandwidth=self.
8                 ↪ filter_para[2],
```

```
l_trans_bandwidth=self.  
    ↪ filter_para[3])
```

这里的滤波处理包含以下几个步骤：

1. **检查滤波参数的有效性**: 确保 `filter_para` 参数是有效的，且包含了带通滤波的必要参数（下限频率、上限频率、高通带宽和低通带宽）。
2. **应用带通滤波器**: 使用 `mne` 库的 `filter` 方法对血氧浓度信号 (`raw_haemo`) 进行带通滤波处理。

通过设置这些参数，带通滤波器可以有效地去除 fNIRS 信号中的低频和高频噪声，只保留 0.01 Hz 到 0.1 Hz 范围内的有用信号。

带通滤波的效果是去除 fNIRS 信号中的低频漂移和高频噪声，使得信号中的有用成分（主要是血氧浓度的变化）更加突出和清晰。这对于后续的信号分析和特征提取至关重要，因为它提高了信号的信噪比 (SNR)，使得我们可以更准确地检测和分析大脑活动。

### 3.3.4 基线校准

基线校准 (Baseline Correction) 是 fNIRS 信号处理中一个重要的步骤，旨在去除测量过程中由于各种原因（如仪器漂移、头皮血流变化等）导致的基线漂移。通过基线校准，可以提高信号的准确性和可靠性，使得后续的分析更加精确。

基线漂移是指在 fNIRS 信号中，随时间变化的慢速波动，这种波动与实际的血氧变化无关，而是由于测量过程中的各种非生理因素引起的。这些因素包括但不限于：

- **仪器漂移**: 由于设备本身的电子元件特性，测量信号可能会随着时间发生漂移。
- **头皮血流变化**: 头皮和颅骨中的血流变化会影响光的吸收和散射，导致基线漂移。
- **环境因素**: 如温度变化、光源稳定性等都会对测量结果产生影响。

基线校准的目的是通过调整信号的基线部分，使其保持在一个相对稳定的水平。通常，基线校准通过从信号中减去基线段的平均值来实现。基线段通常选择在实验开始前或特定条件下进行测量的稳定时间段。

假设我们有一个 fNIRS 信号  $x(t)$ ，其基线段为  $x_{\text{baseline}}(t)$ ，基线校准的过程可以表示为：

$$x_{\text{corrected}}(t) = x(t) - \bar{x}_{\text{baseline}} \quad (7)$$

其中， $\bar{x}_{\text{baseline}}$  是基线段  $x_{\text{baseline}}(t)$  的平均值：

$$\bar{x}_{\text{baseline}} = \frac{1}{T} \int_0^T x_{\text{baseline}}(t) dt \quad (8)$$

其中， $T$  是基线段的时间长度。

基线校准的步骤通常包括以下几个部分：

1. **选择基线段**：在实验开始前或特定条件下进行测量的稳定时间段，选择作为基线段。
2. **计算基线段的平均值**：对基线段进行平均，得到基线的平均值  $\bar{x}_{\text{baseline}}$ 。
3. **校准信号**：从整个信号中减去基线段的平均值，使得校准后的信号基线部分保持在一个相对稳定的水平。

通过基线校准，可以有效去除 fNIRS 信号中的基线漂移，使得信号的基线部分保持在一个相对稳定的水平。这对于后续的信号分析和特征提取非常重要，因为它提高了信号的准确性，减少了非生理因素的干扰。

基线校准前，信号中可能存在显著的慢速波动，使得信号的基线不稳定，难以准确识别实际的血氧变化。基线校准后，这些慢速波动被去除，信号的基线部分变得平稳，更容易识别和分析血氧浓度的变化。

## 3.4 课程设计内容

### 3.4.1 初始化变量并读取数据

```
1 win_ch_data = []
2 win_ch_sick = []
3 if "load_data" in self.debug_mode: print(filename)
4 tag_zero, raw_intensity_arr = self.read_data_from_file(
    ↪ filename)
5 tag_table_all = self.make_tag_table_all(tag_zero)
6 tag_table = self.make_tag_table_conditional(tag_zero)
```

初始化变量：`win_ch_data` 和 `win_ch_sick` 用于存储窗口化后的数据和眩晕值。

读取数据：调用 `read_data_from_file` 函数读取 fNIRS 数据和标签。

### 3.4.2 处理为 MNE 对象

```

1 info = mne.create_info(ch_names=self.ch_names, ch_types=""
2   ↪ fnirs_cw_amplitude", sfreq=self.fs, verbose=None)
3 for i in range(16):
4     loc = np.zeros(12)
5     loc[0:3] = [0, 0, 0]
6     loc[9:12] = self.wavelengths[i]
7     info['chs'][i]['loc'] = loc
8
9 raw_intensity = mne.io.RawArray(raw_intensity_arr.T, info,
10   ↪ verbose=False)
11 raw_intensity.load_data()
12 montage = mne.channels.make_standard_montage('artinis-octam'on
13   ↪ ')
14 raw_intensity.set_montage(montage)

```

创建信息对象：使用 `mne.create_info` 创建包含通道信息的对象。

设置通道位置：为每个通道设置位置信息和波长信息。

创建 Raw 对象：将原始光强度数据转换为 MNE 的 `RawArray` 对象，并加载数据和设置空间掩膜（montage）。

### 3.4.3 转换光密度与血氧密度

```

1 raw_od = mne.preprocessing.nirs.optical_density(raw_intensity
2   ↪ )
3 raw_haemo = mne.preprocessing.nirs.beer_lambert_law(raw_od,
4   ↪ ppf=0.1)

```

光密度转换：将光强度信号转换为光密度信号。

血氧浓度转换：使用贝尔-兰伯特定律将光密度信号转换为血氧浓度信号（HbO 和 HbR）。

### 3.4.4 制作 MNE 任务表并滤波

```

1 anno_onsets, anno_durations, anno_descriptions = self.
2   ↪ make_annotations(tag_table, data_length=len(raw_haemo))
3 annoatations = mne.Annotations(anno_onsets, anno_durations,
4   ↪ anno_descriptions)
5 raw_haemo.set_annotations(annoatations)

```

```

4
5 if isinstance(self.filter_para, list) and len(self.
6     ↪ filter_para) >= 4 and isinstance(self.filter_para[0], (
7     ↪ int, float)) and isinstance(self.filter_para[1], (int,
8     ↪ float)):
9     raw_haemo = raw_haemo.filter(self.filter_para[0], self.
10         ↪ filter_para[1], h_trans_bandwidth=self.filter_para
11         ↪ [2], l_trans_bandwidth=self.filter_para[3])

```

制作任务表：将标签转换为 MNE 的注释对象，并添加到数据中。

滤波处理：应用带通滤波器去除低频和高频噪声。

### 3.4.5 基线校准 fNIRS 信号

```

1 def correct_baseline(self):
2     if self._baseline_correct:
3         print("data have been corrected!")
4         return 0
5     else:
6         baseline_dict={}
7         for sub,block,condition,win in self.win_dict.keys():
8             if condition[0] == "B":
9                 baseline_data_source = self.win_dict[sub,
10                     ↪ block,condition,win][0]
11                 baseline_data = baseline_data_source.mean(
12                     ↪ axis=0)
13                 baseline_dict.update({(sub,block,condition
14                     ↪ [1:]):baseline_data})
15
16         for sub,block,condition,win in self.win_dict.keys():
17             if condition[0] == "B":
18                 continue
19             source_data = self.win_dict[sub,block,condition,
20                 ↪ win][0]
21             ms = self.win_dict[sub,block,condition,win][1]
22             baseline_data = baseline_dict[sub,block,condition
23                 ↪ ]
24             corrected_data = source_data - baseline_data
25             self.win_dict.update({(sub,block,condition,win):[
26

```

```

    ↪ corrected_data,ms])
21 if "correct_baseline" in self.debug_mode:
22     print('baseline_data:', baseline_data)
23     print('source_data:', source_data)
24     print('corrected_data:', corrected_data)
25 selected_keys = list(self.selected_dict.keys())
26 self.selected_dict={}
27 for key in selected_keys:
28     self.selected_dict.update({key:self.win_dict[key
29         ] [0] })
self._baseline_correct = True

```

### 3.4.6 fNIRS 信号可视化展示

```

1 import pickle
2 import matplotlib.pyplot as plt
3
4 # 定义加载PKL文件的函数
5 def load_fnirs_data(file_path):
6     with open(file_path, 'rb') as file:
7         data = pickle.load(file)
8     return data
9
10 # 指定PKL文件路径
11 file_path = 'fnirs.pkl'
12
13 # 加载数据
14 fnirs_data = load_fnirs_data(file_path)
15
16 # 输出数据 (根据实际数据结构进行处理)
17 print(fnirs_data)
18
19 # 打印数据的键和值的基本信息
20 for key, value in fnirs_data.items():
21     print(f"Key: {key}")
22     print(f"Array shape: {value.shape}")
23     print(f"Array data (first 5 rows):\n{value[:5]}\n")
24

```

```
25 # 选择一个键进行可视化
26 key_to_plot = "33, 2, 'BR', 0"
27 data_to_plot = fnirs_data[key_to_plot]
28
29 # 绘制数据
30 plt.figure(figsize=(10, 6))
31 plt.plot(data_to_plot)
32 plt.title(f'Data for key: {key_to_plot}')
33 plt.xlabel('Sample Index')
34 plt.ylabel('Signal Value')
35 plt.legend([f'Channel {i+1}' for i in range(data_to_plot.
36     ↪ shape[1])], loc='upper right')
37 plt.show()
```

### 3.5 课程设计结果

经过上述处理，我们先后可以得到以下的 fNIRS 信号：

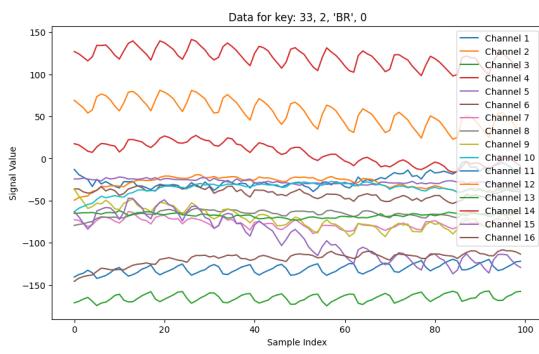


图 36: 无基线校准处理与带通滤波处理结果

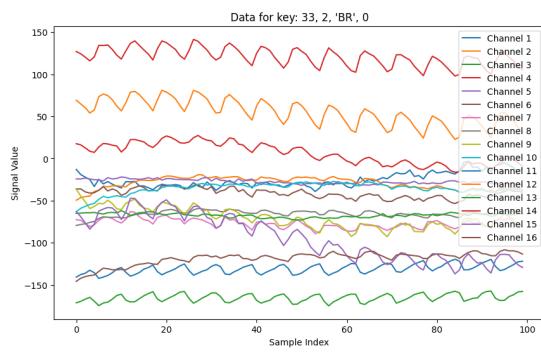


图 37: 基线校准处理结果

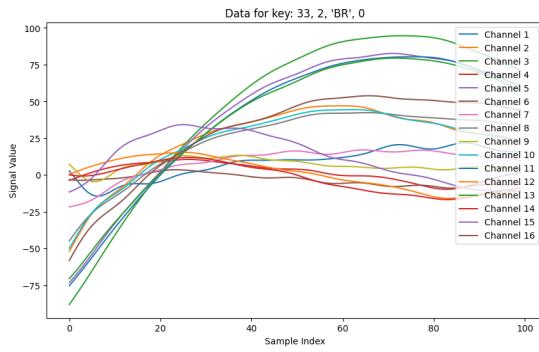


图 38: 0.05-0.7Hz 带通滤波处理结果

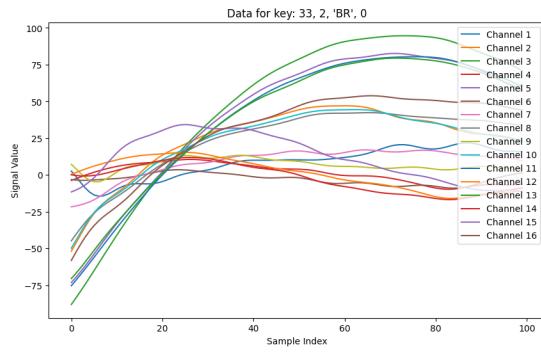


图 39: 0.05-0.7Hz 带通滤波处理 + 基线校准处理结果

在经过了光密度转换和血氧浓度转换后，带通滤波与基线校准的处理是选择性的，需要根据后续处理要求决定。

图 36 展示了未经过基线校准处理的 fNIRS 信号。可以看到，信号中存在显著的基线漂移和噪声，难以直接用于后续分析。图 37 展示了经过基线校准处理后的 fNIRS 信号，相比未处理的信号，基线漂移被有效去除，信号的稳定性显著提高。

图 38 展示了应用 0.05-0.7Hz 带通滤波后的 fNIRS 信号。带通滤波器成功去除了高频和低频噪声，保留了主要的血氧变化信号。然而，仍存在一定程度的基线漂移。图 39 展示了在带通滤波的基础上，进一步应用基线校准处理的结果。结合两种处理方法，信号的质量得到了显著改善，基线漂移和噪声均被有效去除，信号更为平稳和清晰。

通过上述处理步骤，最终得到了质量更高的 fNIRS 信号。这些处理步骤包括：基线校准、带通滤波以及二者的结合应用。实验结果表明，基线校准和带通滤波的联合应用可以显著提高 fNIRS 信号的质量，为后续的信号分析和特征提取提供了可靠的数据基础。

## 4 鸣谢

在完成本次实验和论文的过程中，我深深感受到教师的指导对于学生成长的重要性。在此，我要特别感谢我的数字信号处理老师宁更新老师。他在整个教学和指导过程中表现出的敬业精神和无私奉献令我深受感动。

宁老师不仅在课堂上为我们详细讲解了数字信号处理的基本原理和技术，还在实验过程中给予了我大量的指导和帮助。无论是遇到理论上的困惑还是实验中的难题，他总是耐心地解答，帮助我逐步理解和解决问题。宁老师严谨的科研态度和一丝不苟的工作作风，让我学到了如何在科研中保持专注和细致。

通过宁老师的教导，我不仅掌握了许多专业知识，更学会了如何进行科学研究，如何设计和执行实验，以及如何分析和解释实验结果。这些宝贵的经验和知识将对我未来的学习和科研道路产生深远的影响。

宁老师的鼓励和支持也给予了我极大的动力，使我在面对困难时能够坚持不懈，不断进步。他的学术造诣和高尚的师德，将成为我未来学习和工作的榜样和指引。

再次衷心感谢宁更新老师的辛勤付出和无私帮助！他的教诲将伴随我在今后的学习和科研道路上不断前行。

## 参考文献

- [1] 宁更新. 数字信号处理实验教程. September 2012.
- [2] Daniel C. Harris. *Quantitative Chemical Analysis*. Macmillan Higher Education, 9th edition, 2015.