

Data Visualization - Homework 3 - Report

邓淇升 大数据学院 16307110232

1 编程实现基于空间滤波器的平滑算法与锐化算法，显示原图、结果图和对比图。

● **Python script code:**

1_spatial_filtering.py

● **Input & Output:**

Figure A: test_pattern.tif

- test_pattern.tif_smoothing_by_box-11-11_filter_using_*_padding.png
- test_pattern.tif_smoothing_by_gaussian-8_filter_using_*_padding.png

Figure B: circuit_board.tif

- circuit_board.tif_smoothing_by_median_filter_using_*_padding.png

Figure C: hubble.tif

- hubble.tif_smoothing_by_max_filter_using_*_padding.png
- hubble.tif_smoothing_by_min_filter_using_*_padding.png

Figure D: moon.tif

- moon.tif_sharpening_by_laplacian_filter_using_*_padding.png
- moon.tif_sharpening_by_diagonal_laplacian_filter_using_*_padding.png

Figure E: text.tif

- text.tif_sharpening_by_highboost-1_filter_using_*_padding.png
- text.tif_sharpening_by_highboost-4.5_filter_using_*_padding.png

Figure F: lens.tif

- lens.tif_sharpening_by_prewitt_filter_using_*_padding.png
- lens.tif_sharpening_by_sobel_filter_using_*_padding.png

● Description:

Python 源代码给出了基于空间滤波器的平滑算法和锐化算法。根据不同的滤波算法，可将滤波器分为以下类型：

1. 线性平滑 (linear smoothing):

- 盒状滤波器 (box): 窗口元素均相等且归一化
- 高斯滤波器 (gaussian, 标准差为 8): 窗口元素服从二维高斯分布且归一化

2. 非线性平滑 (non-linear smoothing):

- 中值滤波器 (median): 取窗口内的中位数为新像素值
- 最大值滤波器 (max): 取窗口内的最大值为新像素值
- 最小值滤波器 (min): 取窗口内的最小值为新像素值

3. 线性锐化 (linear sharpening):

- 拉普拉斯滤波器 (laplacian): 窗口中心值为 -4, 4-邻域为 1, 4-对角为 0
- 对角拉普拉斯滤波器 (diagonal laplacian): 窗口中心值为 -8, 8-邻域为 1
- 钝化滤波器 (highboost, 系数为 1): $g(x, y) = f(x, y) + 1 \times g_{mask}(x, y)$
- 高提升滤波器 (highboost, 系数为 4.5): $g(x, y) = f(x, y) + 4.5 \times g_{mask}(x, y)$

4. 非线性锐化 (non-linear sharpening):

- 普鲁伊特滤波器 (prewitt): 使用 Prewitt 算子计算梯度矩阵
- 索伯滤波器 (sobel): 使用 Sobel 算子计算梯度矩阵

其中，盒状滤波器的大小为 $11 * 11$ ，高斯滤波器的大小由标准差使用 3σ 法则确定为 $[6\sigma] \times [6\sigma]$ ，其他滤波器的大小均为 $3 * 3$ 。另外，根据不同的扩展算法，可将图像扩展分为以下类型：

1. 零扩展 (zero padding):

- 新区域全部用零填充

2. 镜面扩展 (mirror padding):

- 新区域关于边界与原图像镜面对称

3. 复制扩展 (replicate padding):

- 新区域取原图像的最接近边界值

代码运行时每一个滤波器均使用三种扩展模式进行测试。

● Source Code:

```
1 import os # 文件操作
2 import time # 时间操作
3
4 import numpy as np # 数组操作
5 from PIL import Image # 图像操作
6
7
8 class ImageData:
9
10     # 初始化
11     def __init__(self, image_name):
12         self.image_name = image_name # 图像的名称
13
14     # 矩阵标准化
15     def normalize(self, matrix):
16         positive_matrix = matrix - np.min(matrix) # 平移至非负值
17         return (255 * positive_matrix / np.max(positive_matrix)).astype(int) # 映射至[0, 255]
18
19     # 图像输入
20     def input_image(self):
21         image = Image.open(os.getcwd() + '\\\\' + self.image_name) # 输入图像
22         self.image_matrix = np.array(image.convert('L')) # 将图像转换成灰度矩阵
23         self.image_size = self.image_matrix.size # 图像的面积
24         self.image_shape = self.image_matrix.shape # 图像的形状
25         self.image_height, self.image_width = self.image_shape # 图像的高与宽
26
27     # 图像输出
28     def output_image(self):
29         image_splice = Image.new(mode='L', size=(self.image_width * 3 + 64, self.image_height + 32), color=255) # 生成空白拼接图像
30         image_splice.paste(im=Image.fromarray(self.image_matrix), box=(16, 16)) # 生成原图块化图像
31         image_splice.paste(im=Image.fromarray(self.filter_matrix), box=(self.image_width + 32, 16)) # 生成平滑处理图像
32         image_splice.paste(im=Image.fromarray(self.normalize(self.image_matrix - self.filter_matrix)), box=(self.image_width * 2 + 48, 16)) # 生成对比图像
33         image_splice.save(self.image_name.replace('.', '_') + '_' + self.processing_mode + '_by_' + self.filter_mode + '_filter_using_' + self.padding_mode + '_padding.png') # 保存图像
34         # image_splice.show() # 运行程序时显示图像
35
36     # 零扩展
37     def zero_padding(self):
38         pass # 零扩展矩阵为零矩阵，不需要额外处理
39
40     # 倍数扩展
41     def mirror_padding(self):
42
43         # 四角
44         self.padding_matrix[:self.pad_h, :self.pad_w] = np.rot90(self.image_matrix[1:1 + self.pad_h, 1:1 + self.pad_w], 2) # 左上角
45         self.padding_matrix[:self.pad_h + self.image_height:, :self.pad_w] = np.rot90(self.image_matrix[-1 - self.pad_h:-1, 1:1 + self.pad_w], 2) # 左下角
46         self.padding_matrix[:self.pad_h, self.pad_w + self.image_width:] = np.rot90(self.image_matrix[1:1 + self.pad_h, -1 - self.pad_w:-1], 2) # 右上角
47         self.padding_matrix[:self.pad_h + self.image_height:, self.pad_w + self.image_width:] = np.rot90(self.image_matrix[-1 - self.pad_h:-1, -1 - self.pad_w:-1], 2) # 右下角
48
49         # 四边
50         self.padding_matrix[:self.pad_h, self.pad_w:self.pad_w + self.image_width] = np.fliplr(self.image_matrix[1:1 + self.pad_h, :]) # 上边框
51         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_w + self.image_width] = np.fliplr(self.image_matrix[-1 - self.pad_h:-1, :]) # 下边框
52         self.padding_matrix[:self.pad_h, :self.pad_h + self.image_height, :self.pad_w] = np.fliplr(self.image_matrix[:, 1:1 + self.pad_w]) # 左边框
53         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_h + self.image_height, :self.pad_w + self.image_width] = np.fliplr(self.image_matrix[:, -1 - self.pad_w:-1]) # 右边框
54
55     # 复制扩展
56     def replicate_padding(self):
57
58         # 四角
59         self.padding_matrix[:self.pad_h, :self.pad_w] = self.image_matrix[0, 0] # 左上角
60         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_w] = self.image_matrix[-1, 0] # 左下角
61         self.padding_matrix[:self.pad_h, :self.pad_w + self.image_width] = self.image_matrix[0, -1] # 右上角
62         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_w + self.image_width] = self.image_matrix[-1, -1] # 右下角
63
64         # 四边
65         self.padding_matrix[:self.pad_h, :self.pad_w + self.image_width] = self.image_matrix[0:1, :] # 上边框
66         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_w + self.image_width] = self.image_matrix[-2:-1, :] # 下边框
67         self.padding_matrix[:self.pad_h, :self.pad_h + self.image_height, :self.pad_w] = self.image_matrix[:, 0:1] # 左边框
68         self.padding_matrix[:self.pad_h + self.image_height, :self.pad_h + self.image_height, :self.pad_w + self.image_width] = self.image_matrix[:, -2:-1] # 右边框
69
70     # 扩展算法
71     def padding(self):
72         self.padding_matrix = np.zeros(np.array(self.image_shape) + np.array(self.filter_shape) - 1) # 生成扩延矩阵
73         self.padding_matrix[self.pad_h:self.pad_h + self.image_height, self.pad_w:self.pad_w + self.image_width] = self.image_matrix # 将原图像复制至扩延矩阵中心
74         getattr(self, '%s_padding' % self.padding_mode)() # 对原图像四周进行扩延
75
76     # 均值滤波器
77     def box_filter(self):
78         return np.ones(self.filter_shape) / (self.filter_height * self.filter_width) # 生成标准均值矩阵
79
80     # 高斯滤波器
81     def gaussian_filter(self):
82         gaussian = lambda s, t: np.exp(-(s**2 + t**2) / (2 * self.sigma**2)) # 设定高斯函数
83         gaussian_matrix = np.array([
84             gaussian(h - self.pad_h, w - self.pad_w) # 计算高斯函数
85             for h in range(self.filter_height) for w in range(self.filter_width) # 遍历图像矩阵
86         ]).reshape(self.filter_shape) # 生成高斯矩阵
87         return gaussian_matrix / np.sum(gaussian_matrix) # 生成归一化高斯矩阵
88
89     # 中值滤波器
90     def median_filter(self):
91         return np.median # 求中值
92
93     # 最大值滤波器
94     def max_filter(self):
95         return np.max # 求最大值
96
97     # 最小值滤波器
98     def min_filter(self):
99         return np.min # 求最小值
100
101     # 拉普拉斯滤波器
102     def laplacian_filter(self):
103         return np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]]) # 拉普拉斯模板
104
105     # 对角拉普拉斯滤波器
106     def diagonal_laplacian_filter(self):
107         return np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]]) # 带对角项的拉普拉斯模板
108
109     # 高提升滤波器
110     def highboost_filter(self):
111         return self.gaussian_filter() # 使用高斯滤波器进行平滑
```

```

113 # 普鲁伊特滤波器
114 def prewitt_filter(self):
115     G_x, G_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]]), np.array([[1, 0, 1], [-1, 0, 1], [-1, 0, 1]]) # 普鲁伊特算子
116     return lambda matrix: np.sqrt(np.sum(G_x * matrix)**2 + np.sum(G_y * matrix)**2) # 计算梯度矩阵
117
118 # 索伯滤波器
119 def sobel_filter(self):
120     G_x, G_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]]), np.array([[1, 0, 1], [-2, 0, 2], [-1, 0, 1]]) # 索伯算子
121     return lambda matrix: np.sqrt(np.sum(G_x * matrix)**2 + np.sum(G_y * matrix)**2) # 计算梯度矩阵
122
123 # 空间滤波算法
124 def spatial_filtering(self, linear):
125
126     kernel = getattr(self, '%s_filter' % self.filter_mode.split('-')[0])() # 生成空间滤波窗口（线性滤波）或空间滤波函数（非线性滤波）
127
128     if linear: # 线性滤波
129         self.filter_matrix = np.array(
130             [
131                 int(np.sum(kernel * self.padding_matrix[h:h + self.filter_height, w:w + self.filter_width])) # 计算空间滤波窗口对应卷积
132                 for h in range(self.image_height) for w in range(self.image_width) # 遍历图像矩阵
133             ],
134             dtype=int).reshape(self.image_shape) # 生成空间滤波矩阵
135
136     else: # 非线性滤波
137         self.filter_matrix = np.array(
138             [
139                 int(kernel(self.padding_matrix[h:h + self.filter_height, w:w + self.filter_width])) # 计算空间滤波函数
140                 for h in range(self.image_height) for w in range(self.image_width) # 遍历图像矩阵
141             ],
142             dtype=int).reshape(self.image_shape) # 生成空间滤波矩阵
143
144     if linear and self.processing_mode == 'sharpening': # 线性锐化算法
145         if self.filter_mode.split('-')[0] == 'highboost': # 钝化滤波提升滤波
146             k = float(self.filter_mode.split('-')[1]) # 设定权重系数
147             self.filter_matrix = ((k + 1) * self.image_matrix - k * self.filter_matrix).astype(int) # 返回锐化矩阵
148         else: # 拉普拉斯滤波
149             self.filter_matrix = self.image_matrix - self.filter_matrix # 返回锐化矩阵
150
151     # 设定滤波器尺寸
152     def set_filter_shape(self, filter_mode):
153
154         if filter_mode.split('-')[0] == 'gaussian': # 高斯滤波器
155             self.sigma = float(filter_mode.split('-')[1]) # 标准差
156             G = int(np.ceil(6 * self.sigma)) # 利用3sigma原则设定滤波器边长为6sigma
157             self.filter_shape = (G, G) if G % 2 else (G + 1, G + 1) # 保证滤波器边长为奇数
158         elif filter_mode.split('-')[0] == 'box': # 均值滤波器
159             self.filter_shape = tuple(map(int, filter_mode.split('-')[1:])))
160         elif filter_mode.split('-')[0] == 'highboost': # 高提升滤波器
161             self.sigma = 3 # 标准差
162             self.filter_shape = (19, 19) # 设定滤波器边长为19
163         else: # 其他滤波器
164             self.filter_shape = (3, 3) # 设定滤波器边长为3
165
166         self.filter_height, self.filter_width = self.filter_shape # 滤波器的高与宽
167         self.pad_h, self.pad_w = (self.filter_height - 1) // 2, (self.filter_width - 1) // 2 # 滤波器的中心位置
168
169     # 空间滤波器
170     def spatial_filter(self, linear, processing_mode, padding_mode, filter_mode):
171
172         self.input_image() # 输入图像
173
174         self.processing_mode = processing_mode # 设定处理模式
175         self.padding_mode = padding_mode # 设定扩展模式
176         self.filter_mode = filter_mode # 设定滤波模式
177         self.set_filter_shape(filter_mode) # 设定滤波器尺寸
178
179         self.padding() # 生成扩展矩阵
180         self.spatial_filtering(linear) # 生成空间滤波矩阵
181
182         self.output_image() # 输出图像
183
184     # 测试接口
185     def test(self, linear, processing_mode, filters):
186         print('\nImage: %s\n' % self.image_name) # 图像名称
187         for filter_mode in filters: # 滤波类型
188             for padding_mode in ['zero', 'mirror', 'replicate']: # 扩展类型
189                 start = time.time() # 开始测试时刻
190                 self.spatial_filter(linear, processing_mode, padding_mode, filter_mode) # 使用空间滤波器处理图像
191                 end = time.time() # 结束测试时刻
192                 print('Processing mode: %s & %s padding & %s filter' % (processing_mode, padding_mode, filter_mode)) # 测试模式
193                 print('Program execute time: %.2f s\n' % (end - start)) # 测试时长
194
195     if __name__ == '__main__':
196
197         # 图像: test_pattern.tif, 模式, 线性平滑, 滤波器, 均值 (滤波器尺寸: 11*11)、高斯 (标准差: 8)
198         ImageData('test_pattern.tif').test(linear=True, processing_mode='smoothing', filters=['box-11-11', 'gaussian-8'])
199
200         # 图像: circuit_board.tif, 模式, 非线性平滑, 滤波器, 中值
201         ImageData('circuit_board.tif').test(linear=False, processing_mode='smoothing', filters=['median'])
202
203         # 图像: bubble.tif, 模式, 非线性平滑, 滤波器, 最大值, 最小值
204         ImageData('bubble.tif').test(linear=False, processing_mode='smoothing', filters=['max', 'min'])
205
206         # 图像: moon.tif, 模式, 线性锐化, 滤波器, 拉普拉斯、对角拉普拉斯
207         ImageData('moon.tif').test(linear=True, processing_modes='sharpening', filters=['laplacian', 'diagonal_laplacian'])
208
209         # 图像: text.tif, 模式, 线性锐化, 滤波器, 钝化 (权重系数: 1)、高提升 (权重系数: 4.5)
210         ImageData('text.tif').test(linear=True, processing_modes='sharpening', filters=['highboost-1', 'highboost-4.5'])
211
212         # 图像: lens.tif, 模式, 非线性锐化, 滤波器, 普鲁伊特、索伯
213         ImageData('lens.tif').test(linear=False, processing_mode='sharpening', filters=['prewitt', 'sobel'])

```

除线性锐化算法外，其他算法的核心均为使用滤波器窗口对原图像的每一个窗口进行卷积计算，得到的值即为窗口中心的新像素值。在线性锐化算法中，拉普拉斯系滤波器得到卷积矩阵后需要与原图像计算差值得到新图像，钝化滤波与高提升滤波的原理是从原图像中减去图像的平滑版本以获得锐化版本。

● Results:

使用空间滤波器进行处理的结果如下所示。

图像: test_pattern.tif
模式: 线性平滑
滤波器: 盒状滤波器
参数: 滤波器大小为 11 * 11

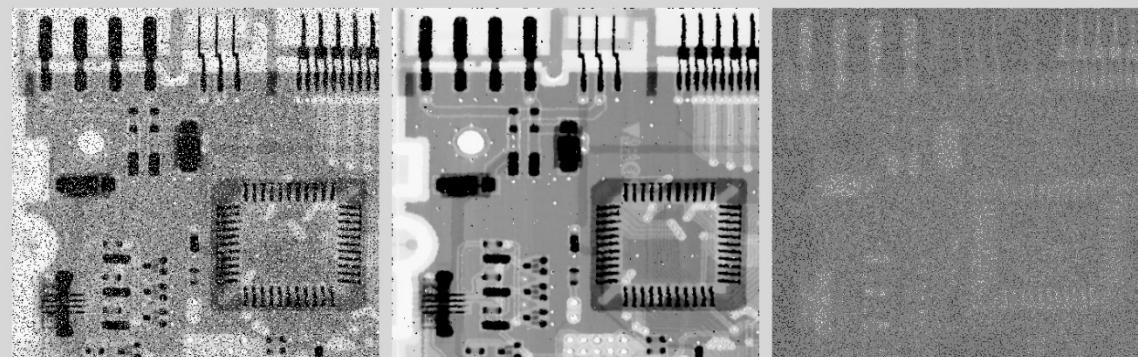
说明: 与另外两种扩展模式相比, 零扩展会产生黑边。结果显示, 不同扩展模式对盒状滤波器的影响不大。理论上盒状滤波器的大小越大, 处理结果会越模糊。图像平均测试时间为 1.40 秒。

原图 结果图 对比图

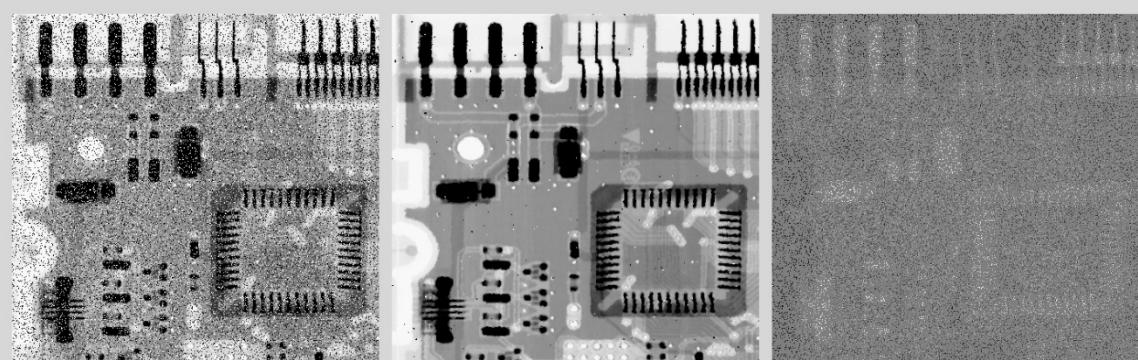
图像: test_pattern.tif
模式: 线性平滑
滤波器: 高斯滤波器
参数: 标准差为 8

说明: 与盒状滤波器相比, 零扩展同样会使处理结果产生黑边。在模糊度方面, 高斯滤波器能在同等程度上获得更大的模糊度。图像平均测试时间为 2.56 秒。

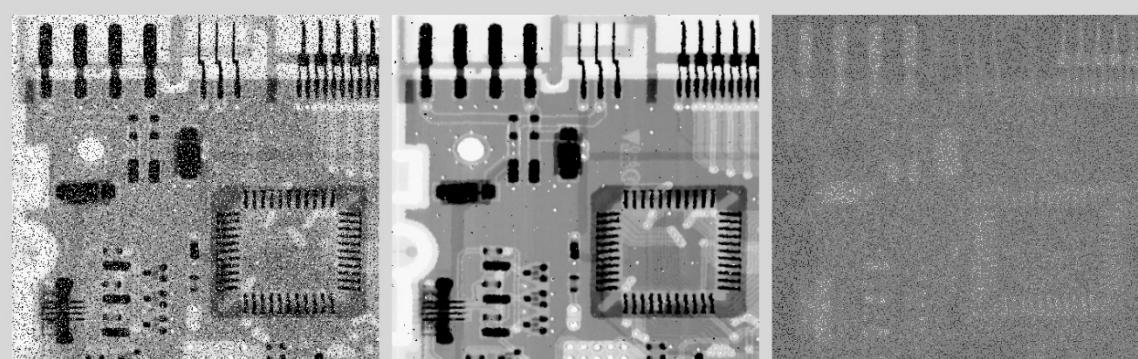
原图 结果图 对比图



零扩展



镜面扩展



复制扩展

原图

结果图

对比图

图像: circuit_board.tif

说明: 相比于均值滤波器, 中值滤波器更有助于去除图像中的椒盐噪声。实验显示, 结果图像的模糊程度不高, 但噪声成功被去除。图像平均测试时间为 5.75 秒。

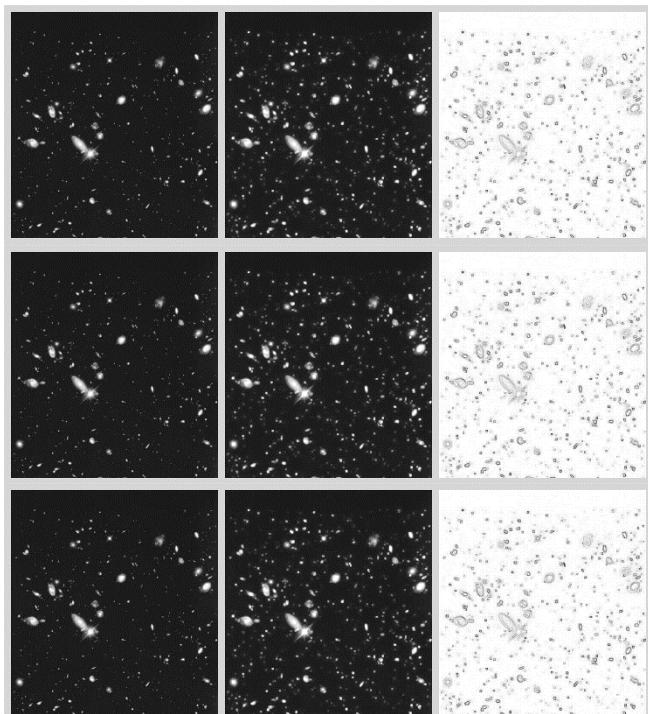
模式: 非线性平滑

滤波器: 中值滤波器

图像: hubble.tif

模式: 非线性平滑

滤波器: 最大值滤波器

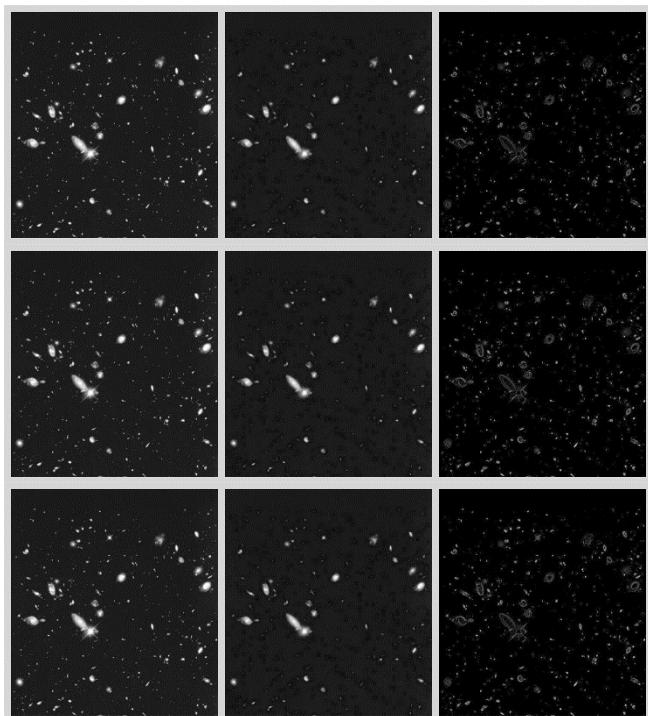


说明: 最大值滤波器主要用于寻找图像中的最亮点。如右所示, 经过处理后, 暗淡的星光会变得更明亮, 有效增强了图像的细节。图像平均测试时间为 1.23 秒。

图像: hubble.tif

模式: 非线性平滑

滤波器: 最小值滤波器



说明: 最小值滤波器主要用于寻找图像中的最暗点, 同时可以起到去除暗部细节的作用。右图的效果是去除较暗的星光, 从而仅保留较亮的星光,。

图像平均测试时间为 1.24 秒。

原图

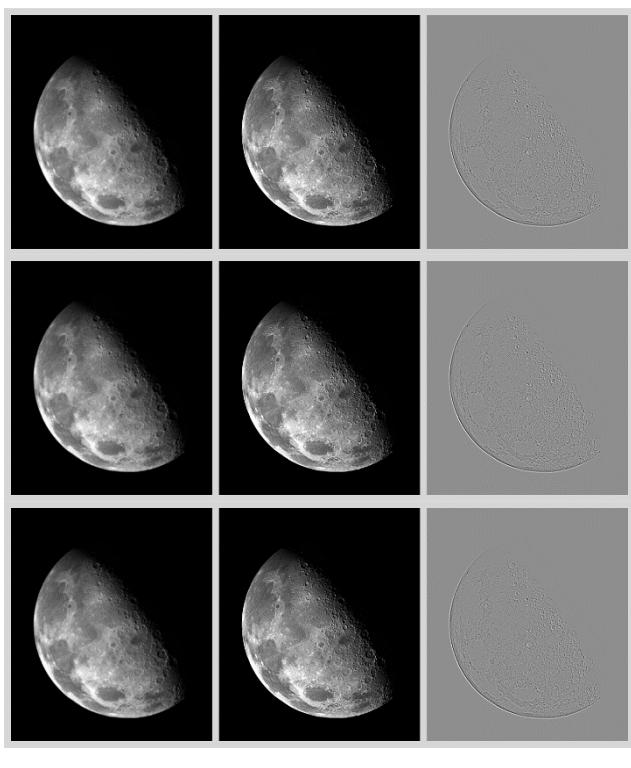
结果图

对比图

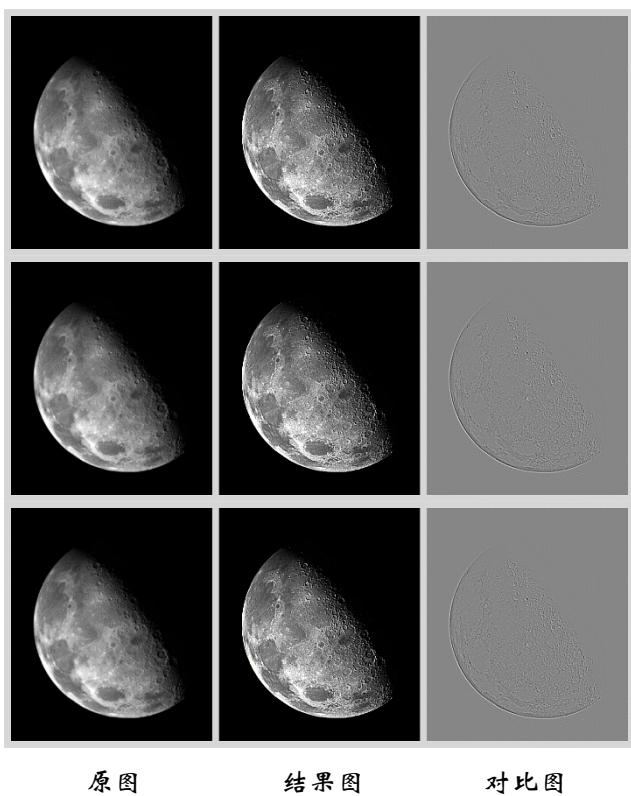
图像: moon.tif

模式: 线性锐化

滤波器: 拉普拉斯滤波器



说明: 锐化模式下, 对比图即为原图的锐化模板结果, 此处对比图已标定至正整数区间。结果显示, 灰度突变的对比度受到增强, 结果图的细节较原图更为锐利。图像平均测试时间为 1.46 秒。



说明: 带对角项的拉普拉斯模板在对角线上会增加额外的细节锐化, 导致结果图的细节相较于普通模板的细节更为锐利。图像平均测试时间为 1.47 秒。

图像: text.tif

模式: 线性锐化

滤波器: 钝化滤波器

参数: 系数为 1



零扩展



镜面扩展



复制扩展

原图

结果图

对比图

图像: text.tif

模式: 线性锐化

滤波器: 高提升滤波器

参数: 系数为 4.5



零扩展



镜面扩展



复制扩展

原图

结果图

对比图

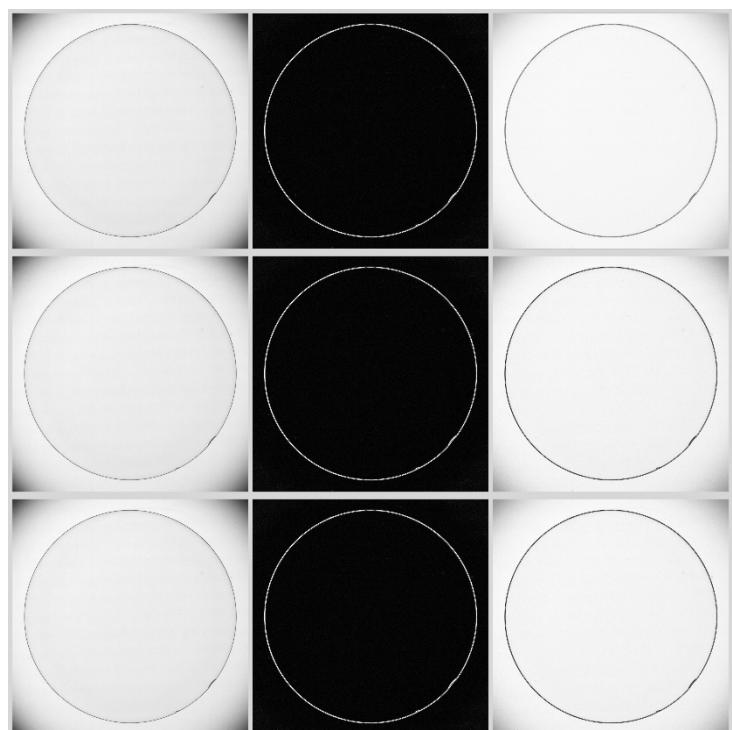
说明: 钝化滤波器与高提升滤波器的算法原理是借助原图像的平滑版本进行加权计算, 从而得到图像的锐化版本。当权重系数为 1 时, 得到的是钝化滤波器; 当权重系数大于 1 时, 则得到高提升滤波器。根据算法原理, 得到的结果图中可能会产生负灰度, 从而导致暗色晕轮出现, 如上面第二幅图所示。实际应用该算法时需要注意权重系数的选取, 尽量保证图像中的所有像素值均为非负值。钝化滤波器和高提升滤波器的平均测试时间为 0.30 秒。

图像: lens.tif

模式: 非线性锐化

滤波器: 普鲁伊特滤波器

说明: 非线性锐化得到的结果图为梯度图像, 使用每一个像素x方向和y方向梯度的平方平均作为新像素值。主要功能是在灰度平坦区域增强较小的突变细节。图像平均测试时间为 12.56 秒。



原图

结果图

对比图

零扩展

镜面扩展

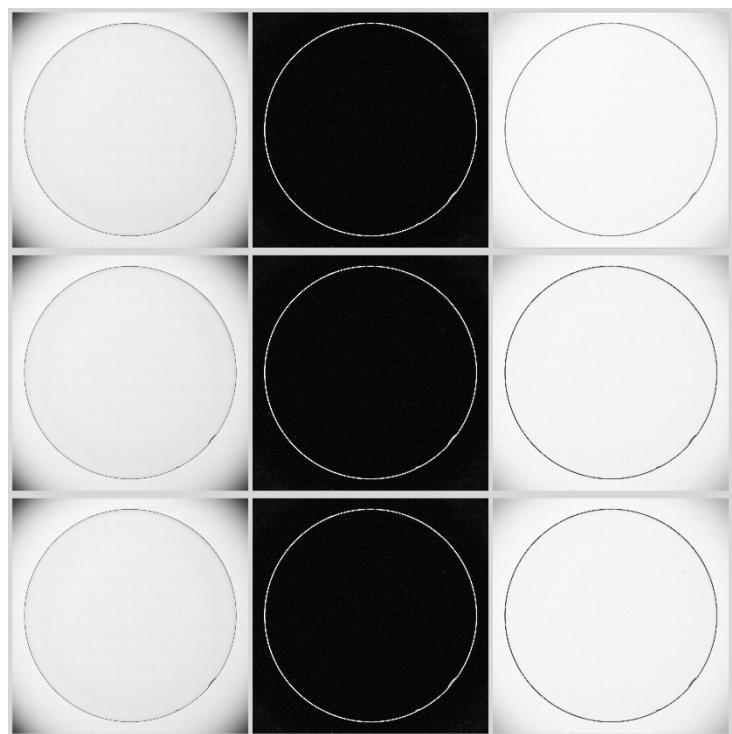
复制扩展

图像: lens.tif

模式: 非线性锐化

滤波器: 索伯滤波器

说明: 索伯滤波器的效果与普鲁伊特滤波器的效果相差不大, 结果表明经过索伯算子滤波的梯度边界会更为明显。图像平均测试时间为 12.57 秒。



原图

结果图

对比图

零扩展

镜面扩展

复制扩展

2 证明二维离散傅里叶变换的卷积定理。

二维离散傅里叶变换 (DFT) 的定义：

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

二维离散傅里叶逆变换 (IDFT) 的定义：

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

二维循环卷积的定义：

$$f(x, y) * h(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x - m, y - n)$$

考察二维空间卷积的傅里叶变换：

$$\begin{aligned} \mathcal{F}(f(x, y) * h(x, y)) &= \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} \left[\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) h(x - m, y - n) \right] e^{-2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)} \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) \left[\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x - m, y - n) e^{-2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)} \right] \\ &= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) \left[H(u, v) e^{-2\pi i \left(\frac{um}{M} + \frac{vn}{N}\right)} \right] \\ &= H(u, v) \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) e^{-2\pi i \left(\frac{um}{M} + \frac{vn}{N}\right)} \\ &= F(u, v) H(u, v) \end{aligned}$$

其中第二行到第三行使用了平移特性，即：

$$f(x - x_0, y - y_0) \Leftrightarrow F(u, v) e^{-2\pi i \left(\frac{ux_0}{M} + \frac{vy_0}{N}\right)}$$

证得：

$$f(x, y) * h(x, y) \Leftrightarrow F(u, v) H(u, v)$$

再考虑二维空间乘积的傅里叶变换：

$$\begin{aligned} \mathcal{F}(f(x, y) h(x, y)) &= \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) h(x, y) e^{-2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)} \\ &= \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) \left[\frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} H(m, n) e^{2\pi i \left(\frac{mx}{M} + \frac{ny}{N}\right)} \right] e^{-2\pi i \left(\frac{ux}{M} + \frac{vy}{N}\right)} \end{aligned}$$

$$\begin{aligned}
&= \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} H(m, n) \left[\sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{2\pi i (\frac{mx}{M} + \frac{ny}{N})} e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \right] \\
&= \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} H(m, n) F(u - m, v - n) \\
&= \frac{1}{MN} F(u, v) * H(u, v)
\end{aligned}$$

其中第三行到第四行使用了平移特性，即：

$$f(x, y) e^{2\pi i (\frac{u_0 x}{M} + \frac{v_0 y}{N})} \Leftrightarrow F(u - u_0, v - v_0)$$

证得：

$$f(x, y) h(x, y) \Leftrightarrow \frac{1}{MN} F(u, v) * H(u, v)$$

综上所述有：

$$\begin{aligned}
f(x, y) * h(x, y) &\Leftrightarrow F(u, v) H(u, v) \\
f(x, y) h(x, y) &\Leftrightarrow \frac{1}{MN} F(u, v) * H(u, v)
\end{aligned}$$

二维离散傅里叶变换的卷积定理得证。

3 编程实现基于频域滤波器的低通平滑算法与选择滤波算法，显示原图、原图频谱、结果频谱和结果图。

● **Python script code:**

3_frequency_domain_filtering.py

● **Input & Output:**

Figure G: integrated_circuit.tif

- integrated_circuit.tif_smoothing_by_ideal_lowpass-60_filter_using_*_padding.png
- integrated_circuit.tif_smoothing_by_butterworth_lowpass-60-2_filter_using_*_padding.png
- integrated_circuit.tif_smoothing_by_gaussian_lowpass-60_filter_using_*_padding.png

Figure H: shepp_logan.tif

- shepp_logan.tif_selective_by_notch-ideal-30_filter_using_*_padding.png
- shepp_logan.tif_selective_by_notch-butterworth-30-4_filter_using_*_padding.png
- shepp_logan.tif_selective_by_notch-gaussian-30_filter_using_*_padding.png

● Description:

Python 源代码给出了基于频域滤波器的低通平滑算法和锐化选择滤波算法。根据不同的滤波算法，可将滤波器分为以下类型：

1. 理想低通滤波器 (ideal lowpass filter):

- 阻断以标准差为半径的所有频率

2. 布特沃斯低通滤波器 (butterworth lowpass filter):

- 消除了截止频率的急剧不连续性，但高阶布特沃斯滤波器会有严重的振铃现象

3. 高斯低通滤波器 (gaussian lowpass filter):

- 相同标准差下平滑效果稍差，但无振铃现象

4. 陷波带阻滤波器 (notch reject filter):

- 若干个高通滤波器的乘积，消除若干噪声的陷波

其中，低通滤波器的标准差为 60，布特沃斯低通滤波器的阶数为二阶，陷波带阻滤波器的标准差为 30，调用布特沃斯高通滤波器时阶数为四阶。需要注意的是，高通滤波器可由 1 减去低通滤波器获得。同样，根据不同的扩展算法，可将图像扩展分为以下类型：

1. 零扩展 (zero padding):

- 新区域全部用零填充

2. 镜面扩展 (mirror padding):

- 新区域关于边界与原图像镜面对称

3. 复制扩展 (replicate padding):

- 新区域取原图像的最接近边界值

代码运行时每一个滤波器均使用三种扩展模式进行测试。

● Source Code:

```
1 import os # 文件操作
2 import time # 时间操作
3
4 import numpy as np # 数组操作
5 from PIL import Image # 图像操作
6
7
8 class ImageData:
9
10     # 初始化
11     def __init__(self, image_name):
12         self.image_name = image_name # 图像的名称
13
14     # 矩阵标准化
15     def normalize(self, matrix):
16         positive_matrix = matrix - np.min(matrix) # 平移至非负值
17         return (255 * positive_matrix / np.max(positive_matrix)).astype(int) # 映射至[0, 255]
18
19     # 矩阵中心化
20     def centerize(self, matrix):
21         height, width = matrix.shape # 矩阵的高与宽
22         return matrix * np.array([-1 if (x + y) % 2 else 1 for x in range(width)] for y in range(height)) # 乘以平移因子
23
24     # 矩阵频谱
25     def spectrum(self, matrix):
26         return self.normalize(np.log(1 + np.abs(matrix))) # 使用对数变换标定频谱
27
```

```

28     # 矩阵中心距离
29     def distance(self, u, v):
30         return np.sqrt((u - self.image_height)**2 + (v - self.image_width)**2) # 频域点至滤波器中心的距离
31
32     # 图像输入
33     def input_image(self):
34         image = Image.open(os.getcwd() + '\\\\' + self.image_name) # 输入图像
35         self.image_matrix = np.array(image.convert('L')) # 将图像转换成灰度矩阵
36         self.image_size = self.image_matrix.size # 图像的面积
37         self.image_shape = self.image_matrix.shape # 图像的形状
38         self.image_height, self.image_width = self.image_shape # 图像的高与宽
39
40     # 图像输出
41     def output_image(self):
42         image_splice = Image.new(mode='L', size=(self.image_width * 4 + 80, self.image_height + 32), color=215) # 生成空白拼接图像
43         image_splice.paste(im=Image.fromarray(self.image_matrix), box=(16, 16)) # 生成原图图像
44         image_splice.paste(im=Image.fromarray(self.spectrum(self.pre_frequency)).resize(self.image_shape, box=(self.image_width + 32, 16)), # 生成原图频谱
45             self.image_width + 32, 16)) # 生成原图频谱
46         image_splice.paste(im=Image.fromarray(self.spectrum(self.post_frequency)).resize(self.image_shape, box=(self.image_width * 2 + 48, 16)), # 生成频域处理结果频谱
47             self.image_width * 2 + 48, 16)) # 生成频域处理结果
48         image_splice.paste(im=Image.fromarray(self.image_matrix), box=(self.image_width * 3 + 64, 16)) # 生成频域处理结果
49         image_splice.save(self.image_name.replace('.', '_') + '_' + self.processing_mode + '_by_' + self.filter_mode + '_filter_using_' + self.padding_mode + '_padding.png') # 保存图像
50         # image_splice.show() # 运行程序时显示图像
51
52     # 零扩展
53     def zero_padding(self):
54         pass # 原扩展矩阵为零矩阵，不需要额外处理
55
56     # 镜面扩展
57     def mirror_padding(self):
58         self.padding_matrix[self.image_height:, self.image_width:] = np.rot90(self.image_matrix, 2) # 右下方
59         self.padding_matrix[:, self.image_width:] = np.fliplr(self.image_matrix) # 下方
60         self.padding_matrix[:, self.image_height, self.image_width:] = np.fliplr(self.image_matrix) # 右方
61
62     # 复制扩展
63     def replicate_padding(self):
64         self.padding_matrix[:, self.image_height:, self.image_width:] = self.image_matrix[-1, -1] # 右下方
65         self.padding_matrix[:, self.image_height:, self.image_width] = self.image_matrix[-2:-1, :] # 下方
66         self.padding_matrix[:, :self.image_height, self.image_width:] = self.image_matrix[:, -2:-1] # 右方
67
68     # 扩展算法
69     def padding(self):
70         self.padding_matrix = np.zeros(2 * np.array(self.image_shape)) # 生成扩展矩阵
71         self.padding_matrix[:, self.image_height, self.image_width] = self.image_matrix # 将原图像复制至扩展矩阵左上角
72         getattr(self, '%s_padding' % self.padding_mode)() # 对原图像进行扩展
73
74     # 理想低通滤波器
75     def ideal_lowpass_filter(self, args):
76         sigma = float(args[0]) # 标准差
77         return np.array([[1 if self.distance(u, v) <= sigma else 0 for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 返回理想低通滤波器
78
79     # 布特沃斯低通滤器
80     def butterworth_lowpass_filter(self, args):
81         sigma, n = float(args[0]), int(args[1]) # 标准差, 阶数
82         return np.array([[1 / (1 + (self.distance(u, v) / sigma)**(2 * n)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 返回布特沃斯低通滤波器
83
84     # 高斯低通滤波器
85     def gaussian_lowpass_filter(self, args):
86         sigma = float(args[0]) # 标准差
87         return np.array([[np.exp(-self.distance(u, v)**2 / (2 * sigma**2)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 返回高斯低通滤波器
88
89     # 陷波滤波器
90     def notch_filter(self, args):
91
92         ######
93         # 调试时使用。
94         # from matplotlib import pyplot as plt # 绘图操作
95         # plt.imsave(self.spectrum(self.pre_frequency)) # 计算原图频谱
96         # plt.savefig(self.image_name.replace('.', '_') + '_spectrum_using_' + self.padding_mode + '_padding.png') # 保存原图频谱
97         # plt.show() # 显示原图频谱
98         #####
99
100    notch = np.array([
101        (35, 35), (195, 35), (270, 35), (360, 35), (435, 35), (595, 35), (835, 35), (995, 35), (1070, 35), (1160, 35), (1235, 35), (1395, 35), # 第一列
102        (35, 195), (195, 195), (270, 195), (360, 195), (435, 195), (595, 195), (835, 195), (995, 195), (1070, 195), (1160, 195), (1235, 195), (1395, 195), # 第二列
103        (35, 270), (195, 270), (270, 270), (360, 270), (435, 270), (595, 270), (835, 270), (995, 270), (1070, 270), (1160, 270), (1235, 270), (1395, 270), # 第三列
104        (35, 360), (195, 360), (270, 360), (360, 360), (435, 360), (595, 360), (835, 360), (995, 360), (1070, 360), (1160, 360), (1235, 360), (1395, 360), # 第四列
105        (35, 435), (195, 435), (270, 435), (360, 435), (435, 435), (595, 435), (835, 435), (995, 435), (1070, 435), (1160, 435), (1235, 435), (1395, 435), # 第五列
106        (35, 595), (195, 595), (270, 595), (360, 595), (435, 595), (595, 595), (835, 595), (995, 595), (1070, 595), (1160, 595), (1235, 595), (1395, 595) # 第六列
107    ]) - self.image_shape # 陷波中心
108
109    NRF = np.ones(2 * np.array(self.image_shape)) # 陷波带阻滤波器
110    if args[0] == 'ideal': # 理想低通滤波器
111        sigma = float(args[1]) # 标准差
112        for (u_k, v_k) in notch: # 逼近陷波中心
113            NRF *= np.array([[1 if self.distance(u - u_k, v - v_k) <= sigma else 1 for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (u_k, v_k)
114            NRF *= np.array([[1 if self.distance(u + u_k, v + v_k) <= sigma else 1 for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (-u_k, -v_k)
115            print('Notch: u_k = %d, v_k = %d... Done!...' % (u_k, v_k)) # 调试信息
116    elif args[0] == 'butterworth': # 布特沃斯高通滤波器
117        sigma, n = float(args[1]), int(args[2]) # 标准差, 阶数
118        for (u_k, v_k) in notch: # 逼近陷波中心
119            NRF *= np.array([[1 / (1 + (self.distance(u - u_k, v - v_k) / sigma)**(2 * n)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (u_k, v_k)
120            NRF *= np.array([[1 / (1 + (self.distance(u + u_k, v + v_k) / sigma)**(2 * n)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (-u_k, -v_k)
121            print('Notch: u_k = %d, v_k = %d... Done!...' % (u_k, v_k)) # 调试信息
122    elif args[0] == 'gaussian': # 高斯高通滤波器
123        sigma = float(args[1]) # 标准差
124        for (u_k, v_k) in notch: # 逼近陷波中心
125            NRF *= np.array([[1 - np.exp(-self.distance(u - u_k, v - v_k)**2 / (2 * sigma**2)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (u_k, v_k)
126            NRF *= np.array([[1 - np.exp(-self.distance(u + u_k, v + v_k)**2 / (2 * sigma**2)) for v in range(2 * self.image_width)] for u in range(2 * self.image_height)]) # 陷波: (-u_k, -v_k)
127            print('Notch: u_k = %d, v_k = %d... Done!...' % (u_k, v_k)) # 调试信息
128
129    # 频域滤波算法
130    def frequency_domain_filtering(self):
131
132        # 频域滤波五步法
133        self.padding() # 第一步: 扩展图像
134        self.pre_frequency = np.fft.fft2(self.centerize(self.padding_matrix)) # 第二步: 计算傅里叶变换
135        self.post_frequency = getattr(self, '%s_filter' % self.filter_mode.split('-')[0])(self.filter_mode.split('-')[1:]) * self.pre_frequency # 第三步: 计算频域滤波矩阵
136        result = self.centerize(np.real(np.fft.ifft2(self.post_frequency))).astype(int) # 第四步: 计算傅里叶逆变换
137        self.result_matrix = result[:self.image_height, :self.image_width] # 第五步: 提取图像
138
139        # 频域滤波器
140        def frequency_domain_filter(self, processing_mode, padding_mode, filter_mode):
141
142            self.input_image() # 输入图像
143
144            self.processing_mode = processing_mode # 设定处理模式
145            self.padding_mode = padding_mode # 设定扩展模式
146            self.filter_mode = filter_mode # 设定滤波模式
147
148            self.frequency_domain_filtering() # 生成频域滤波矩阵
149
150            self.output_image() # 输出图像

```

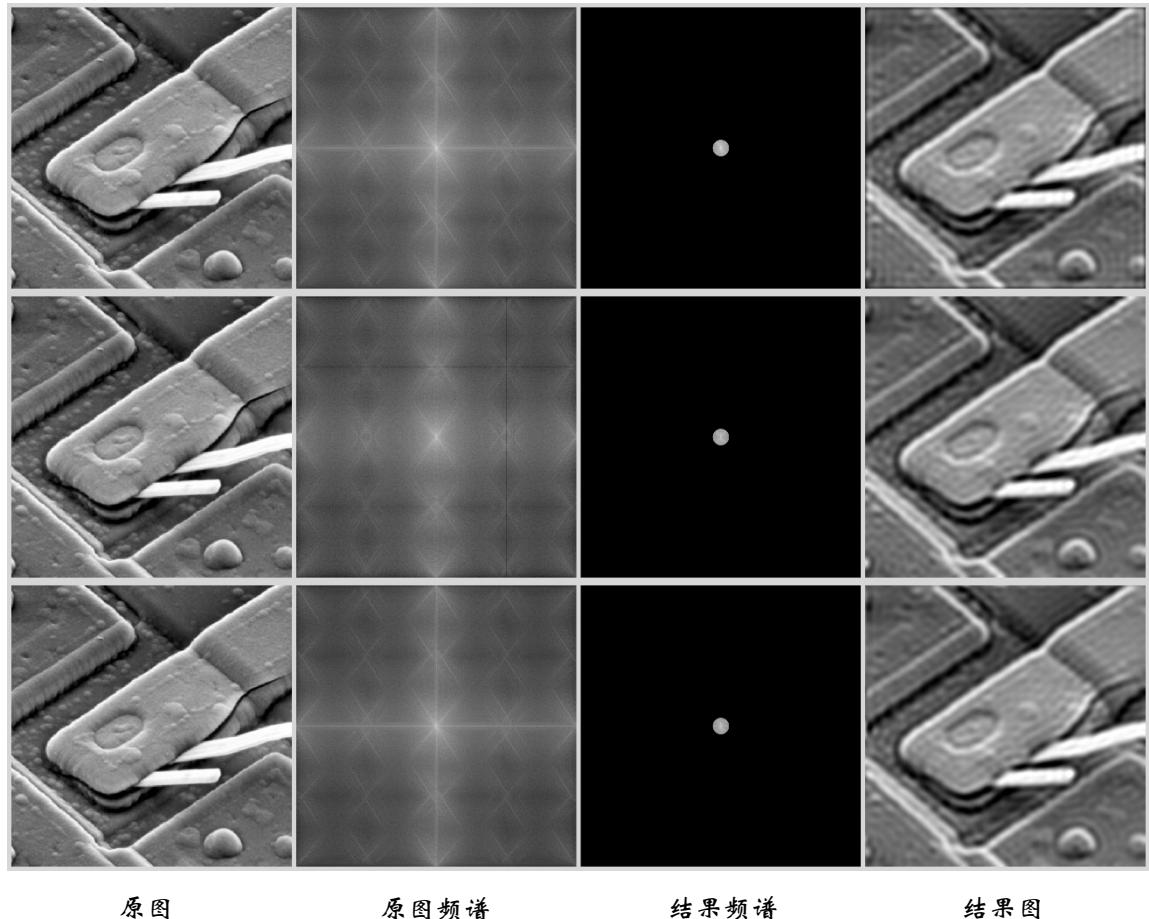
```

152     # 测试接口
153     def test(self, processing_mode, filters):
154         print('\nImage: %s' % self.image_name) # 图像名称
155         for filter_mode in filters: # 滤波类型
156             for padding_mode in ['zero', 'mirror', 'replicate']: # 扩展类型
157                 start = time.time() # 开始测试时刻
158                 self.frequency_domain.filter(processing_mode, padding_mode, filter_mode) # 使用频域滤波器处理图像
159                 end = time.time() # 结束测试时刻
160                 print('Processing mode: %s & %s padding & %s filter' % (processing_mode, padding_mode, filter_mode)) # 测试模式
161                 print('Program execute time: %.2f s\n' % (end - start)) # 测试时长
162
163
164     if __name__ == '__main__':
165
166         # 图像: integrated_circuit.tif, 模式: 低通平滑, 滤波器: 理想 (标准差: 60), 布特沃斯 (标准差: 60, 阶数: 2)、高斯 (标准差: 60)
167         ImageData('integrated_circuit.tif').test(processing_mode='smoothing', filters=['ideal_lowpass-60', 'butterworth_lowpass-60-2', 'gaussian_lowpass-60'])
168
169         # 图像: shepp_logan.tif, 模式: 选择性, 滤波器: 理想 (标准差: 30), 布特沃斯 (标准差: 30, 阶数: 4)、高斯 (标准差: 30)
170         ImageData('shepp_logan.tif').test(processing_mode='selective', filters=['notch-ideal-30', 'notch-butterworth-30-4', 'notch-gaussian-30'])

```

● Results:

使用频域滤波器进行处理的结果如下所示。



图像: integrated_circuit.tif
模式: 低通平滑
滤波器: 理想滤波器
参数: 标准差为 60

说明: 如上所示, 不同扩展模式的频谱会略有差异。理想低通滤波器的截止频率有急剧的不连续性, 由于盒状函数的傅里叶变换为 sinc 状函数, 外侧波瓣会导致结果图的振铃现象。图像平均测试时间为 8.44 秒。

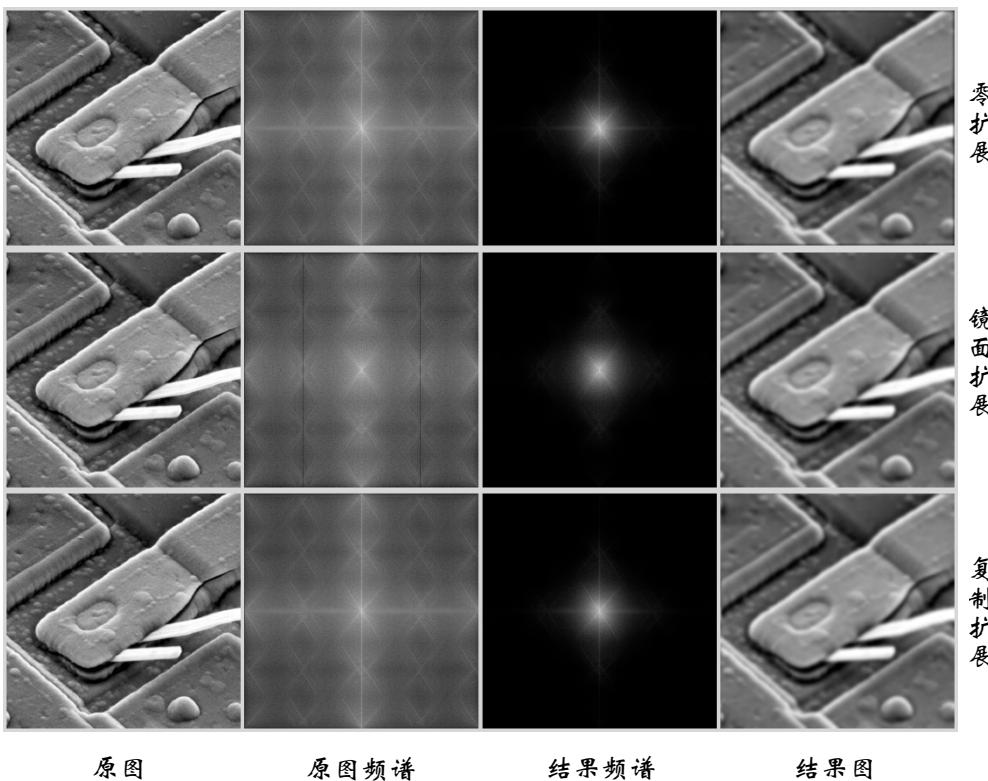
图像: integrated_circuit.tif

模式: 低通平滑

滤波器: 布特沃斯滤波器

参数: 标准差为 60, 阶数为 2

说明: 归因于在低频与高频之间的平滑过渡, 二阶布特沃斯低通滤波器能有效防止振铃现象的产生。如右图所示, 处理结果获得一定程度的模糊, 细节处没有产生振铃现象。图像平均测试时间为 12.63 秒。



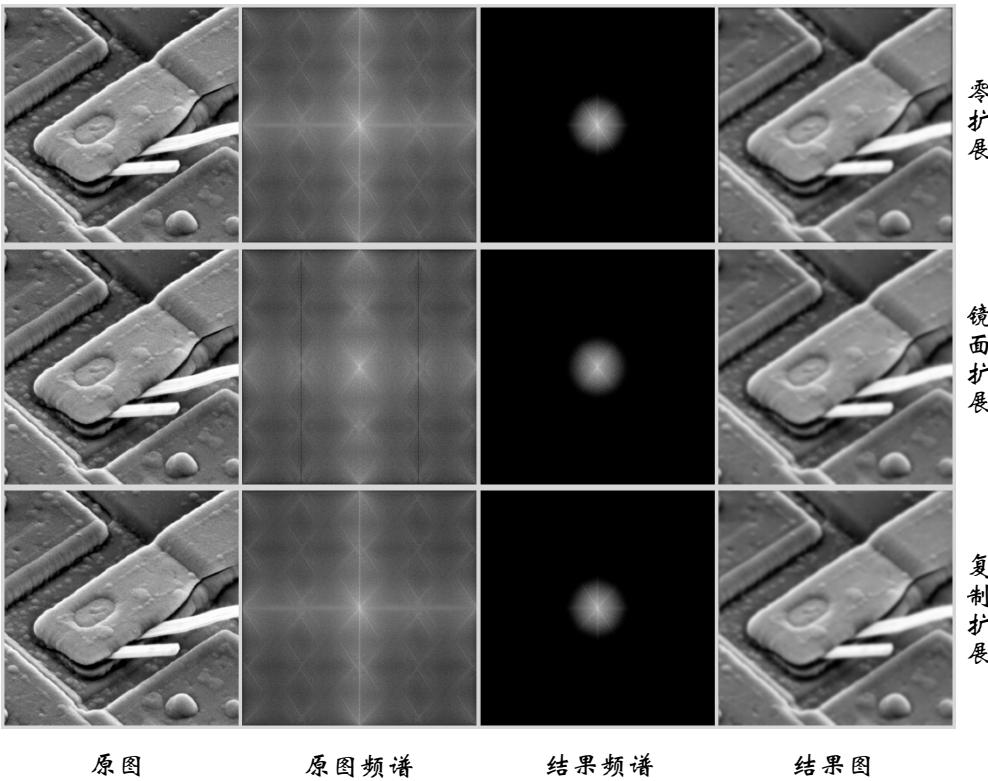
图像: integrated_circuit.tif

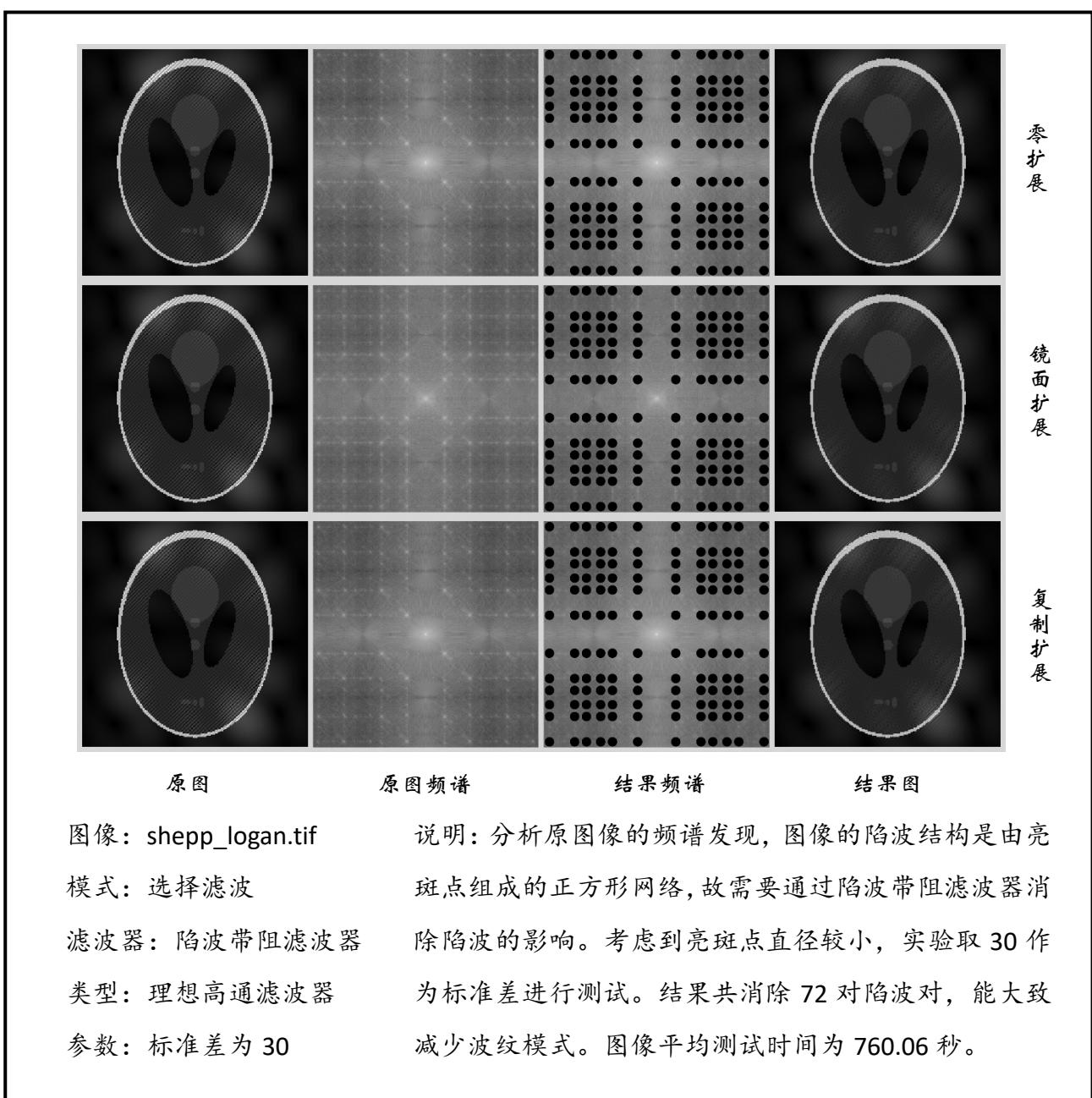
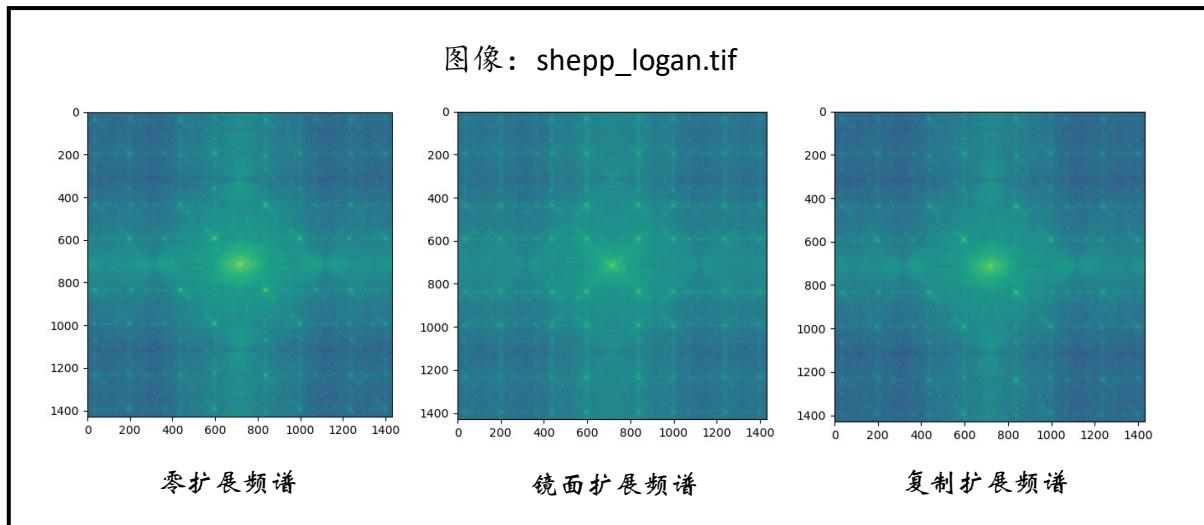
模式: 低通平滑

滤波器: 高斯滤波器

参数: 标准差为 60

说明: 高斯低通滤波器完全避免了振铃现象的产生, 但在相同标准差下模糊效果不及布特沃斯低通滤波器。如右图所示, 与上图比较, 模糊效果稍差。图像平均测试时间为 15.55 秒。





图像: integrated_circuit.tif

模式: 选择滤波

滤波器: 陷波带阻滤波器

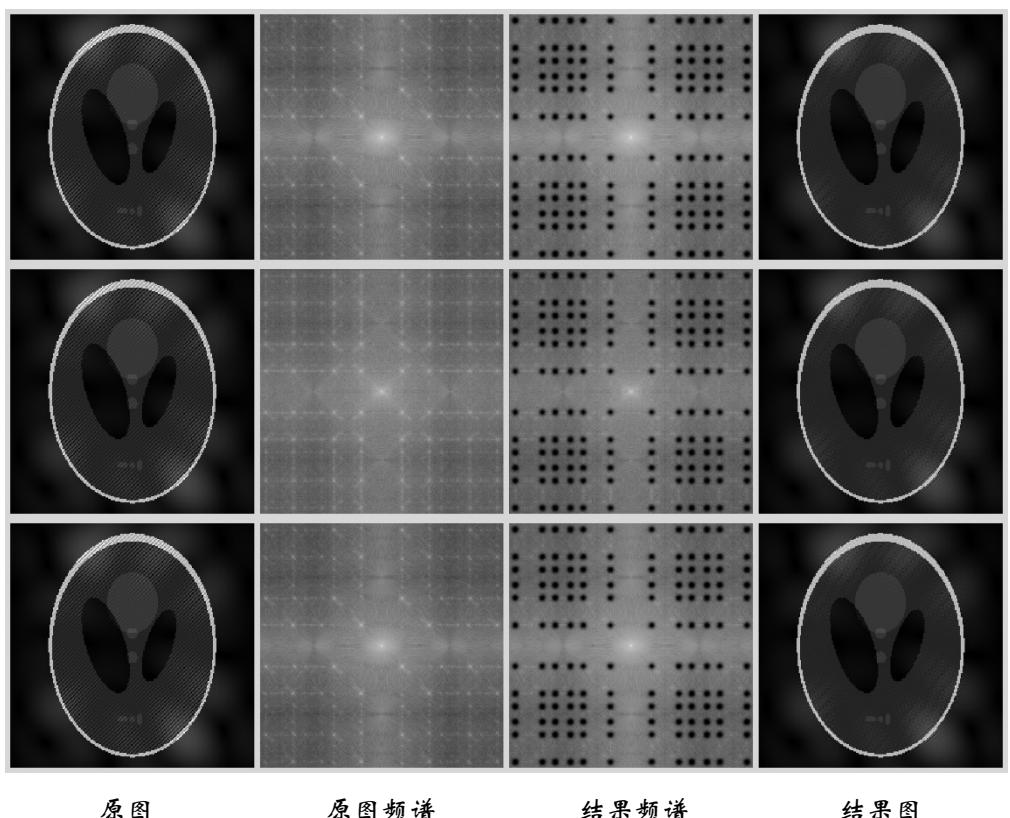
类型: 布特沃斯高通滤波器

参数: 标准差为 30, 阶数为 4

说明: 布特沃斯高通滤波器的
截止频率边缘较为平滑, 对原
图的影响也较小。与理想高通
滤波器相比, 消除的效果更好。

图像平均测试时间为 1099.36

秒。



图像: integrated_circuit.tif

模式: 选择滤波

滤波器: 陷波带阻滤波器

类型: 高斯高通滤波器

参数: 标准差为 30

说明: 结果显示, 高斯高通滤
波器的效果在三种类型中最
好。观察频谱可知, 高斯高通
的处理模式对原频谱影响最
小。虽然得到结果仍然不是最
优, 但在视觉上已经大致消除了
波纹模式。图像平均测试时
间为 1435.71 秒。

