画像処理レポート 4

235738B 越後 玲輝

2025 年 6 月 29 日

1　平均化フィルタの調査

教科書 P.107 図 5.5 の加重平均化フィルタ（３ｘ３）および（５ｘ５）と、

P.106 図 5.3 の平均化フィルタ（３ｘ３）および（５ｘ５）を、任意の画像に適用し、それぞれのフィルタの効果を”比較検討”しなさい。

違いがわかるような適切な比較検討方法ないし実験を自由に考案して実施。

1.1　ソースコード

Listing 1: 平均化フィルタ

```
import cv2
from matplotlib import pyplot as plt
import numpy as np
from IPython.display import Image, display

img = cv2.imread("./syake_image.webp")

filter_image = cv2.blur(img, (5, 5)) #5の平均化フィルタをかけるx5

image1_path_output = "./syake_filter5x5.png" 出力画像のパス        #
cv2.imwrite(image1_path_output, filter_image) 保存       #

fig = plt.figure() グラフのインスタンス化 #
ax1 = fig.add_subplot(111)

ax1.imshow(filter_image, cmap = 'gray')
fig.tight_layout()
plt.show()
plt.close()
```

Listing 2: ガウシアンフィルタ

```
1  import cv2
2  from matplotlib import pyplot as plt
3  import numpy as np
4  from IPython.display import Image, display
5
6  img = cv2.imread("./syake_image.webp")
7
8  filter_image = cv2.GaussianBlur(img, (5, 5), sigmaX=1) ガウシアン
       フィルタ#
9
10 image1_path_output = "./syake_filter_Gauss_5x5.png" 出力画像のパス
             #
11 cv2.imwrite(image1_path_output, filter_image) 保存              #
12
13 fig = plt.figure() グラフのインスタンス化 #
14 ax1 = fig.add_subplot(111)
15
16 ax1.imshow(filter_image, cmap = 'gray')
17 fig.tight_layout()
18 plt.show()
19 plt.close()
```

1.2　出力画像

(a) 平均化フィルタ 3 × 3

(b) 平均化フィルタ 5 × 5

(c) ガウシアンフィルタ 3 × 3

(d) ガウシアンフィルタ 5 × 5

図 1: 4 種類のフィルタ比較

1.3　比較、目視での画像確認

図 1 に示したように、同一の入力画像に対して 4 種類の平滑化フィルタを適用し、その効果を比較した。

(a) 平均化フィルタ 3 × 3 は、ノイズ除去効果は控えめだが、輪郭が比較的保たれており、元画像の情報を大きく損なわない処理となった。

(b) 平均化フィルタ 5 × 5 では、より広範囲にわたって平均化されるため、全体的にぼやけた印象となった。特にキャラクターの顔や背景の輪郭が著しく失われており、ノイズ除去には有効だが、情報損失も大きい。

(c) ガウシアンフィルタ 3 × 3 は、平均化と比べて中央に重みを持つため、エッジ部分の情報がより保持され、滑らかさとディテールの両立が見られた。実用上のバランスに優れる。

(d) ガウシアンフィルタ 5 × 5 では、より強く滑らかに処理され、背景やノイズは効果的に除去されているものの、キャラクターの輪郭もややぼやけている。視認性は高いが、情報保持には注意が必要である。

以上の比較から、ガウシアンフィルタは平均化フィルタよりも滑らかさと情報保持のバランスに優れ、特に 3 × 3 サイズは最も自然な視覚効果を与えることを体感した。

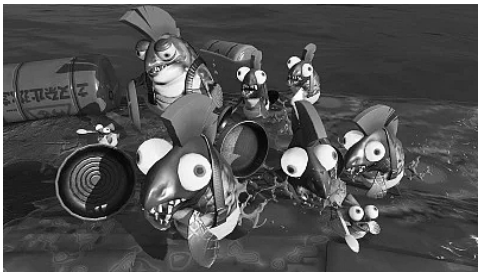
2　鮮鋭化フィルタの調査

教科書 P.123 の可変鮮鋭化フィルタを任意の画像に適用し、その効果を調べなさい。
効果の違いがわかるような適切な比較検討方法ないし実験を自由に考案して実施。

2.1　ソースコード

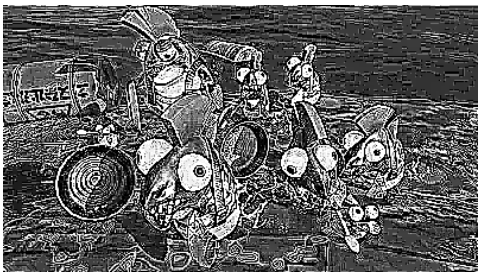
```
1  import cv2
2  from matplotlib import pyplot as plt
3
4  image1_path = "syake_image.webp" 画像パス指定　#
5  image1 = cv2.imread(image1_path,0) 画像読み込みでグレースケールでカラー
       #(0,1)
6
7  def unsharp_masking(img, kx, ky, k):
8      img_copy = img.astype('int16').copy() 符号付き#16整数に変換してコ
           ピーbit
9      img_gaussian = cv2.GaussianBlur(img_copy, (kx, ky) ,sigmaX
           =1) ガウシアンフィルタ#(3x3)で画像をぼ
           かす
10     diff_img = img_copy -img_gaussian 元画像とぼかし画像の差分を計算　#
11     img_k = diff_img * k 差分に　# k を掛けて、どれだけ強調するかを調整
12     result = img_copy + img_k
13     return result
14
15 image1_unsharp_masked = unsharp_masking(image1, 3, 3, 100)
16
17 image1_path_output = "./syake_unsharp_masked_3x3_k=100.png" 出力画
       像のパス#
18 cv2.imwrite(image1_path_output, image1_unsharp_masked) 保存 #
19
20 fig = plt.figure() グラフのインスタンス化 #
21 ax1 = fig.add_subplot(111)
22
23 ax1.imshow(image1_unsharp_masked, cmap = 'gray')
24 fig.tight_layout()
25 plt.show()
26 plt.close()
```

2.2 出力画像

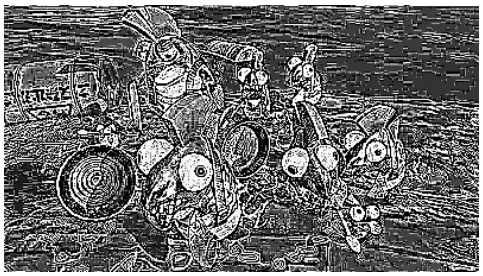


(a) 鮮鋭化フィルタ k=1

(b) 鮮鋭化フィルタ k=10



(c) 鮮鋭化フィルタ k=50



(d) 鮮鋭化フィルタ k=100

図 2: k の値を変化、比較

2.3 比較、目視での画像確認

図 2 に示したように、係数 k の値を変化させることで、鮮鋭化の強さが大きく異なることがわかる。

(a) の $k = 1$ では、エッジがほどよく強調され、画像全体の視認性が向上しているが、不自然さはない。バランスの取れた出力である。

(b) の $k = 10$ では、鮮鋭化が強くなり、細部がさらに明瞭になるが、背景の一部にノイズが目立ち始めた。鮮鋭化の効果が強く出る一方で、画像としての自然さはやや損なわれる。

(c) の $k = 50$ および (d) の $k = 100$ では、エッジが過剰に強調され、画像全体が輪郭線だらけのような印象となった。特に $k = 100$ では、画像としての再利用が困難なほど情報が劣化している。

この結果から、k の値は画像の目的に応じて適切に設定する必要がある。一般的な使用では、$k = 1 \sim 5$ 程度が妥当であり、それ以上では鮮鋭化による副作用（ノイズ増強、視認性低下）が顕著となることが確認された。

3 輪郭抽出

任意の画像から輪郭を抽出しなさい。できるだけ綺麗な輪郭を抽出すること。
そのためにどのような工夫や処理を行ったか結果・コードとともに記述すること。

3.1 ソースコード

Listing 3: 輪郭抽出

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

image_path = "syake_image.webp" 画像パス指定                #
image = cv2.imread(image_path,0) 画像読み込みでグレースケールでカラー
        #(0,1)
image_color = cv2.imread(image_path,1)ガウシアンフィルタ

#3でノイズ除去x3
blur = cv2.GaussianBlur(image, (3, 3), sigmaX=1)

#エッジ検出で輪郭抽出Canny
edges = cv2.Canny(blur, threshold1=100, threshold2=200)輪郭の追跡

#
contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)元画像に輪郭を
    描画

#
cv2.drawContours(image_color, contours, -1, (0, 255, 0), 2)

image_path_output = "./syake_contours.png" 出力画像のパス#
cv2.imwrite(image_path_output, image_color) 保存 #

plt.subplot(1, 3, 3)
plt.title("Contours")
plt.imshow(cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB))
```

```
27 plt.axis('off')
28
29 plt.tight_layout()
30 plt.show()
```

3.2 出力画像

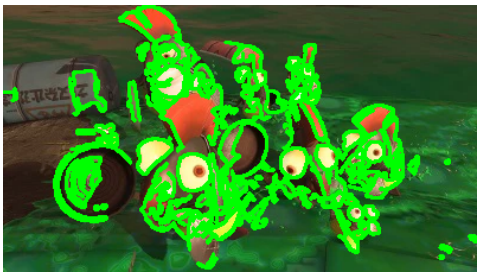


図 3: 輪郭抽出結果

3.3 輪郭抽出における工夫

輪郭をできるだけ綺麗に抽出するために、以下のような前処理およびパラメータ調整を行った。

- **ノイズ除去のための前処理**として、入力画像に対してガウシアンフィルタを適用した。これにより、Canny 法における誤検出（偽エッジ）を抑制し、安定した輪郭抽出が可能。
- **Canny エッジ検出**を用いて、画素値の急激な変化を抽出した。閾値は `threshold1=100`, `threshold2=200` に設定し、細部のエッジは除外しつつも主要な構造が抽出されるように調整。
- **外輪郭のみを抽出するために**、`cv2.findContours()` のモードとして `cv2.RETR_EXTERNAL` を使用した。これにより、内部構造のノイズや細かい輪郭を排除し、視認性の高い主要な輪郭だけを残すことができた。
- 輪郭描画には `cv2.drawContours()` を使用し、元画像（カラー）に対して緑色の線で描画した。これにより、元画像の情報を保ちつつ、輪郭の強調部分が一目でわかるように工夫した。

これらの工夫により、過剰なノイズを含まず、対象物の輪郭が明確に抽出された画像を得ることができた。特に、前処理としてのガウシアンフィルタと、輪郭の絞り込みとしての `RETR_EXTERNAL` の組み合わせが個人的に気に入っている。

4　周波数フィルタ（ローパスフィルタ）を任意の画像に適用

4.1　ソースコード

```
1  import numpy as np
2  import cv2
3
4  def lowpass_filter(src, sigma=0.5):
5
6      src = np.fft.fft2(src)
7      height, width = src.shape
8      cy, cx = int(height / 2), int(width / 2)
9
10     fsrc = np.fft.fftshift(src)
11
12     y, x = np.ogrid[:height, :width]
13     mask = np.exp(-((x - cx)**2 + (y - cy)**2) / (2 * (sigma *
           min(height, width) / 2)**2))
14
15     fdst = fsrc * mask
16     fdst = np.fft.ifftshift(fdst)
17     dst = np.fft.ifft2(fdst)
18
19     dst_real = np.real(dst)
20     dst_normalized = cv2.normalize(dst_real, None, 0, 255, cv2.
           NORM_MINMAX)
21
22     return dst_normalized.astype(np.uint8)
23
24 def main():
25     # ガウス分布のパラメータ（小さいほどフィルタの影響が強くなる）
26     sigma = 0.3
27
28     # 入力画像を読み込み
29     img = cv2.imread("syake_image.webp")
30
```

```
31      # 画像をRGBRed, Green, のチャンネル画像に分割Blue1
32      img_blue, img_green, img_red = cv2.split(img)
33
34      # ローパスフィルタ処理
35      himg_blue = lowpass_filter(img_blue, sigma)
36      himg_green = lowpass_filter(img_green, sigma)
37      himg_red = lowpass_filter(img_red, sigma)
38
39      # 画像に戻すRGB
40      himg = cv2.merge((himg_blue, himg_green, himg_red))
41
42      # 処理結果を出力
43      cv2.imwrite("syake_lowpass_filter.png", himg)
44
45  if __name__ == "__main__":
46      main()
```

4.2 出力画像

(a) ローパスフィルタ前

(b) ローパスフィルタ後

図 4: ローパスフィルタ実行結果

4.3 ローパスフィルタの効果

図 4 に示すように、ローパスフィルタ適用後の画像では全体的にエッジがなだらかになり、シャープな輪郭がぼけた状態となっている。これは高周波成分（ディテール）が除去されたことを示しており、ノイズ除去や滑らかさ向上の効果が出ていることがわかる。

この処理は、プリプロセッシング（前処理）としてノイズ軽減に用いることができる。