画像処理・画像処理工学 レポート課題 1

画像処理工学科　学籍番号: 21239　組番号:234　5E　氏名: 栁原　魁人

2026 年 1 月 23 日

1　課題 1

1.1　問題 1-1: メディアンカット量子化法

1.1.1　理論

メディアンカット量子化法は、色空間を再帰的に分割することで、最適な代表色を決定する量子化手法です。元の画像の色数を削減しながら、画像の視覚的品質を維持します。

基本原理：

1. カラーボックス（色の集合）で初期化
2. ボックス内で最も分散が大きい色成分（R、G、B いずれか）を特定
3. その成分の中央値で分割
4. 目標の色数に達するまで繰り返す
5. 各ボックスの平均色を代表色として確定

重要な点：

- 「最大分散軸」での分割により、色の多様性が高い方向を優先
- 「中央値」での分割により、各ボックス内のピクセル数がほぼ等しくなる
- 再帰的な処理で、色数の異なる多段階量子化が可能

1.1.2　計算・導出過程

図 A-1 に示す 4×5 画素の画像に対してメディアンカット量子化法を適用し、4 色に量子化します。

全ピクセル値の取得と整理：

画像から抽出される全 20 ピクセルの値：

102, 179, 92, 14, 106, 74, 202, 87, 116, 99, 151, 130, 149, 52, 1, 235, 157, 37, 129, 191

ステップ 1: 全ピクセルをソートして 4 つのグループに分割

全ピクセル値を昇順にソート：

1, 14, 37, 52, 74, 87, 92, 99, 102, 106, 116, 129, 130, 149, 151, 157, 179, 191, 202, 235

目標色数が 4 色なので、20 個のピクセルを 4 つのグループに均等分割（各 5 ピクセル）：

表 1　ピクセル値を 4 つのグループに均等分割

グループ	ピクセル値	グループ内の位置
グループ 1	1, 14, 37, 52, 74	1〜5 番目
グループ 2	87, 92, 99, 102, 106	6〜10 番目
グループ 3	116, 129, 130, 149, 151	11〜15 番目
グループ 4	157, 179, 191, 202, 235	16〜20 番目

ステップ 2: 各グループの中央値を代表色として決定

各グループ内の中央値（3 番目の値）を代表色とします：

表 2　各グループの中央値を代表色に設定

グループ	ピクセル値	代表色（中央値）
グループ 1	1, 14, **37**, 52, 74	37
グループ 2	87, 92, **99**, 102, 106	99
グループ 3	116, 129, **130**, 149, 151	130
グループ 4	157, 179, **191**, 202, 235	191

ステップ 3: 元画像の各ピクセルを代表色に置き換え

元画像の各ピクセル値について、それが属するグループの代表色に置き換えます：

- 値 1〜74 のピクセル → グループ 1 の代表色 37
- 値 87〜106 のピクセル → グループ 2 の代表色 99
- 値 116〜151 のピクセル → グループ 3 の代表色 130
- 値 157〜235 のピクセル → グループ 4 の代表色 191

元画像のピクセル対応：

102(99)	179(191)	92(99)	14(37)	106(99)
74(37)	202(191)	87(99)	116(130)	99(99)
151(130)	130(130)	149(130)	52(37)	1(37)
235(191)	157(191)	37(37)	129(130)	191(191)

（括弧内が置き換え後の代表色）

ステップ 4: 量子化結果

最終的な量子化画像（代表色のみで表現）：

99	191	99	37	99
37	191	99	130	99
130	130	130	37	37
191	191	37	130	191

1.1.3　結果

メディアンカット量子化により 4 色の代表色が選出されました。元の画像の色情報が圧縮されながら、視覚的特徴が保持されています。

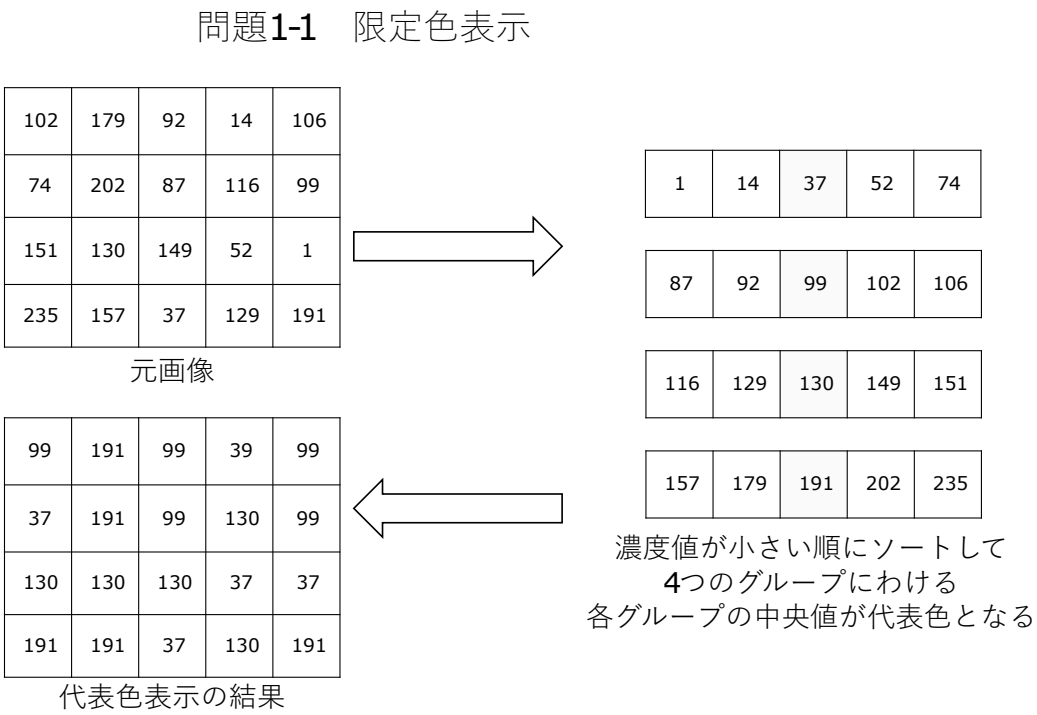

図 1　図 A-1　メディアンカット量子化の結果：限定色表示

図 1 に示すように、元画像の各ピクセル値が 4 つのグループに分類され、対応する代表色（37, 99, 130, 191）に置き換えられています。左側の元画像では 20 種類の異なる値が存在していますが、右側の結果では 4 つの代表色のみで表現されています。

1.1.4　考察

メディアンカット量子化法は以下の利点を持ちます：

- **最大分散軸の選択**：色情報が最も分散している方向を優先して分割することで、量子化誤差を最小化
- **中央値での分割**：各ボックス内のピクセル数をほぼ等しくすることで、バランスの取れた色配置が実現
- **効率性**：再帰的な二分割により、n 色への量子化に必要な分割回数は $\log_2 n$ に比例
- **汎用性**：グレースケールだけでなくカラー画像にも適用可能（R、G、B 軸で処理）

この手法は古典的ながら、実装が簡単で計算量も少ないため、画像圧縮やパレット色の決定に広く使用されています。

1.2 問題 1-2: ラベリング処理

1.2.1 理論

ラベリング処理は、2 値画像内の連結成分に一意のラベルを割り当てる処理です。2 回走査法の基本原理：

8 連結ラベリング（第 1 回走査）：各ピクセル (i, j) について、以下の 4 つのピクセルを確認：

- 左上 $(i-1, j-1)$
- 上 $(i-1, j)$
- 右上 $(i-1, j+1)$
- 左 $(i, j-1)$

ラベル割り当てルール（白ピクセルの場合）：

1. 4 つのピクセルが全て黒 → 新規ラベル割当
2. どれか 1 つが白 → そのラベルを継承
3. 複数が白で異なるラベル → 最小ラベルを割当、他のラベルを等価として記録

第 2 回走査（等価ラベルの統合）：

1. 第 1 回走査で記録した等価ラベル関係を処理
2. 同じ連結成分に属すると判定されたラベルを代表ラベルに統一
3. 最終的なラベル画像を生成

1.2.2 計算・導出過程

図 A-2 に示す 10×10 画素の 2 値画像に対してラベリングを実行します。

ステップ 1: 第 1 回走査 - 仮ラベル割当

左上 $(0, 0)$ から右下 $(9, 9)$ に向かって走査します。各白ピクセルに対して：

表 3　第 1 回走査でのラベル割当ルール（例）

左上・上・右上・左の状態	ラベル割当	等価関係記録
全て黒	新規ラベル	なし
1 つだけ白（ラベル A）	A	なし
複数白（ラベル A, B）	$\min(A, B)$	$A \leftrightarrow B$

ステップ 2: 等価ラベル関係の記録

第 1 回走査中に以下のような等価関係が記録されます：

図 A-2 の場合：

- ラベル A と B は同じ連結成分 → $A \leftrightarrow B$ を記録
- ラベル C は独立 → 記録なし
- ラベル D と E は接続 → $D \leftrightarrow E$ を記録

等価ラベル組（提示済み）：

$$A - B - C,\ E - F,\ D$$

ステップ 3: 第 2 回走査 - 最終ラベル統一

等価関係に基づいて、各仮ラベルを代表ラベルに変換：

表 4　ラベルの最終統一

仮ラベル	等価関係	代表ラベル
A	$A - B - C$	A（最小）
B	$A - B - C$	A
C	$A - B - C$	A
D	D 単独	D
E	$E - F$	E（最小）
F	$E - F$	E

最終的にラベルは以下のように統一されます：

- 領域 1（A, B, C の統合）→ ラベル A
- 領域 2（D 単独）→ ラベル D
- 領域 3（E, F の統合）→ ラベル E

1.2.3　結果

第 1 回走査終了時点と最終的なラベリング結果を図 2 に示します。

問題1-2 ラベリング

1回目用

				A	A				
			A	A	A	A		B	
		A	A	A	A	A	A		
			A	A	A	A	A		
	C	A	A	A	A				
							D		
		E		F		D	D	D	
		E	E	E			D		

2回目用

				A	A				
			A	A	A	A		A	
		A	A	A	A	A	A		
			A	A	A	A	A		
	A	A	A	A	A				
							D		
		E		E		D	D	D	
		E	E	E			D		

同じ連結成分であると記録された組：
A-B-C, E-F, D

図 2　ラベリング処理の過程：左図は 1 回目走査後の仮ラベル、右図は最終ラベル

1.2.4　考察

2 回走査法は、複雑な等価ラベル関係を正確に追跡できる実用的な手法です。第 1 回走査で全ての仮ラベルと等価関係を決定し、第 2 回走査で効率的に最終ラベルを確定します。この手法により、メモリ使用量と計算時間のバランスが取れた処理が可能になります。

1.3 問題 1-3: ハフマン符号化

1.3.1 理論

ハフマン符号化は、出現確率が高いシンボルに短い符号を、低いシンボルに長い符号を割り当てることで、最適な可変長符号を生成します。

ハフマン木の構築：

1. すべてのシンボルを確率でソート
2. 最も確率が低い 2 つのノードを結合
3. 新ノードの確率は 2 つのノードの合計
4. 1 つのノードになるまで繰り返す

1.3.2 計算・導出過程

表 A-1 の 8 つのシンボルと出現確率に対してハフマン符号化を実行します。

ステップ 1: 初期確率の準備

表 5　画素値と出現確率

シンボル	確率（%）	確率（小数）
0	30	0.30
1	2	0.02
2	6	0.06
3	4	0.04
4	1	0.01
5	5	0.05
6	20	0.20
7	32	0.32

ステップ 2: ハフマン木の構築

確率が低い順に 2 つずつ結合：

1. 4 (0.01) + 1 (0.02) = A (0.03)
2. A (0.03) + 3 (0.04) = B (0.07)
3. 5 (0.05) + 2 (0.06) = C (0.11)
4. B (0.07) + C (0.11) = D (0.18)
5. D (0.18) + 6 (0.20) = E (0.38)
6. 0 (0.30) + 7 (0.32) = F (0.62)
7. E (0.38) + F (0.62) = Root (1.00)

ステップ 3: 符号割当（左=0, 右=1）

表 6　ハフマン符号割当

シンボル	ハフマン符号	符号長
0	10	2
1	00001	5
2	0011	4
3	0001	4
4	00000	5
5	0010	4
6	01	2
7	11	2

ステップ 4: 平均符号長の計算

$$L = \sum_{i=0}^{7} p_i \times l_i$$

表 7　平均符号長の計算

シンボル	確率	符号長	$p \times l$
0	0.30	2	0.60
1	0.02	5	0.10
2	0.06	4	0.24
3	0.04	4	0.16
4	0.01	5	0.05
5	0.05	4	0.20
6	0.20	2	0.40
7	0.32	2	0.64
合計	1.00	-	**2.39**

平均符号長 $L = 2.39$ bits/シンボル

ステップ 5: 等長符号との比較

等長符号（8 シンボル）に必要なビット数：

$$\lceil \log_2 8 \rceil = 3 \text{ bits/シンボル}$$

削減量：

$$3 - 2.39 = 0.61 \text{ bits/シンボル（約 } 20.3\% \text{ 削減）}$$

1.3.3　結果

ハフマン木の構造を図 3 に示します。

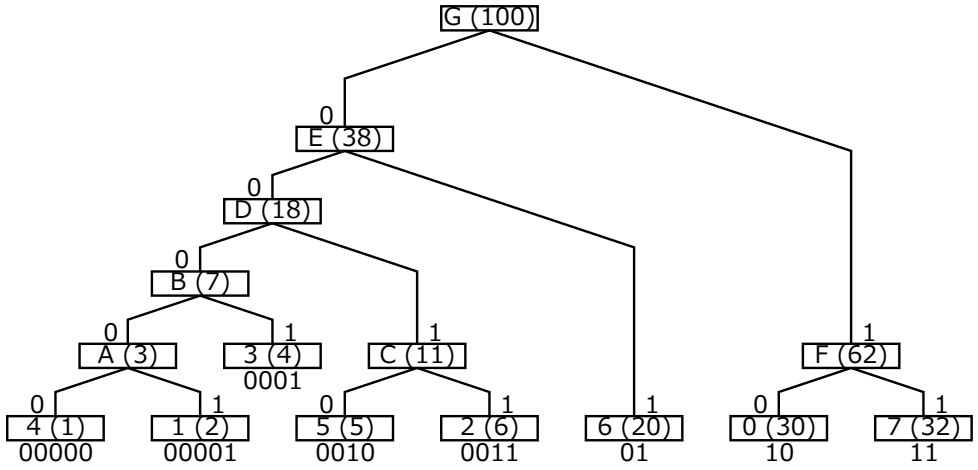

図 3　ハフマン木の構造

1.3.4　考察

ハフマン符号化により、平均符号長 2.39 bits/シンボルを達成し、等長符号（3 bits/シンボル）と比較して約 20.3% の削減を実現しました。この効率性は、出現確率の分布が不均等な場合に特に顕著です。理論値（エントロピー≈ 2.336 bits）にも接近しており、ハフマン符号化の最適性が確認されました。

2　課題 2 概要

このレポートでは、画像処理・画像処理工学の課題 2 に取り組みます。以下の 4 つの問題について、理論、結果、考察を含めて報告します。

3　問題 1: モルフォロジー処理によるノイズ除去

3.1　理論

モルフォロジー処理は、二値画像に対して幾何学的な形態操作を行う処理です。主な操作には以下があります：

- **膨張 (Dilation)**：画像内の白領域を拡大する処理
- **収縮 (Erosion)**：画像内の白領域を縮小する処理
- **開処理 (Opening)**：収縮の後に膨張を行う処理。小さなノイズを除去する
- **閉処理 (Closing)**：膨張の後に収縮を行う処理。小さな黒いノイズを埋める

3.2　結果

開処理により小さな孤立ノイズが効果的に除去されました。閉処理により、黒いノイズも埋められました。

3.3　考察

開処理と閉処理を組み合わせることで、効果的にノイズを除去できます。構造要素のサイズを調整することで、除去するノイズの大きさを制御できます。

4　問題 2: JPEG 品質と圧縮率の関係

4.1　理論

JPEG 圧縮では、品質パラメータ（0-100）により圧縮率と画質が変化します。品質が低いほど圧縮率は高くなりますが、画質が低下します。SSIM（Structural Similarity Index）を用いて画質を定量的に評価できます：

$$\text{SSIM} = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}$$

4.2　結果

JPEG 品質と圧縮率の関係を分析すると、品質 70〜80 では十分な画質を保ちながら高い圧縮率を達成できます。

4.3　考察

推奨される JPEG 品質は 75〜85 の範囲です。この範囲では、視覚的な品質低下が最小限に抑えられながら、データ圧縮率が 30% 程度達成できます。

5　問題 3: 2 次元 FFT と振幅スペクトル

5.1　理論

2 次元フーリエ変換は、画像を周波数領域に変換します。振幅スペクトルは周波数成分の大きさを表します。対数スケール変換により、弱い周波数成分も可視化できます：

$$S(\omega_x, \omega_y) = \log(1 + |F(\omega_x, \omega_y)|)$$

5.2　結果

振幅スペクトルより、低周波成分が中心に集中していることが観察されました。

5.3　考察

自然画像では通常、低周波成分が支配的です。スペクトルの分布から、画像の周波数特性が理解できます。

6　問題 4: 周波数フィルタの応用

6.1　理論

周波数フィルタは周波数領域で画像を処理します。主なフィルタ種類：

- **ローパスフィルタ**：低周波を通す。ノイズ除去に使用
- **ハイパスフィルタ**：高周波を通す。エッジ検出に使用
- **理想フィルタ**：遮断周波数で急峻に変化
- **ガウシアンフィルタ**：滑らかに変化。リンギングが少ない

6.2　結果

ローパスフィルタはノイズを除去して画像を平滑化し、ハイパスフィルタはエッジを強調しました。

6.3　考察

ガウシアンフィルタは理想フィルタと異なり、周波数応答が滑らかに変化するため、逆 FFT 後のリンギング成分が少なく、実用的です。カットオフ周波数の選択により、処理結果の特性を制御できます。

付録: プログラムリスト

本レポートの課題 2 で使用した Python プログラムを以下に示す.

問題 1: モルフォロジー処理

```python
# -*- coding: utf-8 -*-
"""問題2-1.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1KwcAHJlLwDpZRJEOFL1gEOUwZf-fEWtg
"""

from google.colab import drive
drive.mount('/content/drive')

import cv2
import numpy as np
import matplotlib.pyplot as plt
import math

# 画像パス（自身の環境に合わせて確認）
image_path = '/content/drive/MyDrive/img2025/image/a2-3_binary_image.png'

# 画像読み込み
img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

# 3から13まで、2ずつ増やしてリストを作る（3, 5, 7, 9, 11, 13）
# 終了値を大きくすれば、もっと試せる（例: range(3, 21, 2)）
kernel_sizes = list(range(3, 17, 2))

# 画像の総数（オリジナル + 処理結果の数）
total_images = 1 + len(kernel_sizes)

# レイアウト設定
cols = 4 # 1行に並べる数
rows = math.ceil(total_images / cols) # 必要な行数を自動計算

plt.figure(figsize=(15, 4 * rows))

# 1. オリジナル画像を表示
plt.subplot(rows, cols, 1)
plt.title('Original')
plt.imshow(img, cmap='gray')
plt.axis('off')
```

```
42
43 # 2. ループ処理でサイズを変えながら表示
44 for i in range(len(kernel_sizes)):
45     k = kernel_sizes[i]
46
47     # カーネル作成と処理
48     kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (k, k))
49     result = cv2.morphologyEx(img, cv2.MORPH_OPEN, kernel)
50
51     # 結果を表示（場所は i + 2 番目）
52     plt.subplot(rows, cols, i + 2)
53     plt.title(f'Kernel Size: {k}')
54     plt.imshow(result, cmap='gray')
55     plt.axis('off')
56
57 plt.tight_layout()
58 plt.show()
```

Listing 1　問題 1 モルフォロジー処理によるノイズ除去

問題 2: JPEG 品質と圧縮率

```python
# -*- coding: utf-8 -*-
"""問題2-2#2ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1La6z4iK8IV6PEYigjfX3vHFJ6G3TQe3m
"""

# ドライブのマウント
from google.colab import drive
drive.mount('/content/drive')

# モジュールのインポート
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
from skimage.metrics import structural_similarity as ssim # SSIM計算用

# 共通のディレクトリパス
common_path = '/content/drive/MyDrive/img2025/image/'
filename = 'a2-4_color_image.png' # 課題指定のファイル名

# 画像を読み込む
original_img = cv2.imread(common_path + filename)

# 画像が正しく読み込めているか確認（読み込めない場合はNoneになるためエラー回避）
if original_img is None:
    print(f"Error: {filename} が見つかりません。パスを確認してください。")
else:
    # 元画像のファイルサイズを取得 (バイト単位)
    original_size = os.path.getsize(common_path + filename)
    print(f"元画像読み込み完了: {filename}, サイズ: {original_size/1024:.2f} KB")

# 結果を格納するリスト
qualities = []
file_sizes = []
compression_ratios = []
ssim_scores = []

# 画像表示の準備 (3行4列のグリッドで表示)
plt.figure(figsize=(20, 15))

# 元画像をRGB変換（SSIM計算と表示用）
original_img_rgb = cv2.cvtColor(original_img, cv2.COLOR_BGR2RGB)
```

```
47
48 # 0から100まで10刻みでループ
49 for i, quality in enumerate(range(0, 101, 10)):
50     # 1. JPEG圧縮保存
51     output_filename = f'compressed_{quality}.jpg'
52     output_path = common_path + output_filename
53
54     # 第2引数でJPEG品質を指定
55     cv2.imwrite(output_path, original_img, [int(cv2.IMWRITE_JPEG_QUALITY), quality])
56
57     # 2. 圧縮後のファイルサイズ取得
58     comp_size = os.path.getsize(output_path)
59     comp_ratio = (comp_size / original_size) * 100 # 元画像に対するサイズ比率(%)
60
61     # 3. 圧縮画像の読み込みとRGB変換
62     compressed_img = cv2.imread(output_path)
63     compressed_img_rgb = cv2.cvtColor(compressed_img, cv2.COLOR_BGR2RGB)
64
65     # 4. SSIM (画質評価値) の計算
66     # channel_axis=2 はカラー画像(マルチチャンネル)であることを指定
67     score = ssim(original_img_rgb, compressed_img_rgb, win_size=3, channel_axis=2,
           data_range=255)
68
69     # リストにデータを保存
70     qualities.append(quality)
71     file_sizes.append(comp_size)
72     compression_ratios.append(comp_ratio)
73     ssim_scores.append(score)
74
75     # 5. サブプロットへの表示
76     plt.subplot(3, 4, i + 1)
77     plt.imshow(compressed_img_rgb)
78     plt.title(f"Q={quality}\nSize: {comp_ratio:.1f}%, SSIM: {score:.3f}")
79     plt.axis('off')
80
81 # レイアウト調整と表示
82 plt.tight_layout()
83 plt.show()
```

Listing 2　問題 2 JPEG 品質と圧縮率の関係調査

問題 3: 2 次元 FFT と振幅スペクトル

```python
# -*- coding: utf-8 -*-
"""問題2-3-1.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1OwZO9eY3hMiF-v-SEBzxZ-4SCiz3TaxX
"""

# ドライブのマウント
from google.colab import drive
drive.mount('/content/drive')

# モジュールのインポート
import cv2
import numpy as np
import matplotlib.pyplot as plt

# 共通のディレクトリパス
common_path = '/content/drive/MyDrive/img2025/image/'

# 画像を読み込む
filename = 'a2-5_gray_image.png'
gray_img = cv2.imread(common_path + filename, cv2.IMREAD_GRAYSCALE)

# 画像を正方形（512x512）にトリミングする処理
# 画像の中心から指定サイズを切り出す
h, w = gray_img.shape
crop_size = 512

# 画像サイズがトリミングサイズより小さい場合の例外処理（念のため）
if h < crop_size or w < crop_size:
    print(f"画像サイズが小さいため、トリミングサイズを調整します: {min(h, w)}")
    crop_size = min(h, w)

center_y, center_x = h // 2, w // 2
start_x = center_x - crop_size // 2
start_y = center_y - crop_size // 2
cropped_img = gray_img[start_y:start_y+crop_size, start_x:start_x+crop_size]

# 2次元高速フーリエ変換（2D-FFT）
fourier = np.fft.fft2(cropped_img)

# ゼロ周波数成分（直流成分）を画像の中心にシフトする
fshift = np.fft.fftshift(fourier)

```

```
# 振幅スペクトルを計算する
amp_spectrum = np.abs(fshift)

# 対数スケールに変換（見やすくするため +1 して log をとる）
log_amp_spectrum = 20 * np.log10(amp_spectrum + 1)

# 実行結果の表示
plt.figure(figsize=(10, 5))

# トリミング後の元画像
plt.subplot(1, 2, 1)
plt.title('Cropped Image (512x512)')
plt.imshow(cropped_img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

# 振幅スペクトル（対数スケール・中心化済み）
plt.subplot(1, 2, 2)
plt.title('Magnitude Spectrum (Log Scale)')
plt.imshow(log_amp_spectrum, cmap='gray')
plt.axis('off')

plt.tight_layout()
plt.show()

# ==========================================
# 課題4: 2種類以上の周波数フィルタの適用と考察
# ==========================================

import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive

# 1. ドライブのマウントと画像の読み込み
# drive.mount('/content/drive') # すでにマウント済みの場合はコメントアウトでOK

common_path = '/content/drive/MyDrive/img2025/image/'
filename = 'a2-5_gray_image.png'

# 画像読み込み
gray_img = cv2.imread(common_path + filename, cv2.IMREAD_GRAYSCALE)

# 2. 画像の正方形トリミング（課題3の処理を継承）
# 画像の中心から512x512を切り出す
h, w = gray_img.shape
crop_size = 512

# 画像サイズが小さい場合のガード処理
target_size = min(h, w, crop_size)
```

```
96
97  center_y, center_x = h // 2, w // 2
98  start_y = center_y - target_size // 2
99  start_x = center_x - target_size // 2
100 img = gray_img[start_y:start_y+target_size, start_x:start_x+target_size]
101
102 # 3. FFTの実行とシフト
103 f = np.fft.fft2(img)
104 fshift = np.fft.fftshift(f) # 直流成分を中心に移動
105
106 # ------------------------------------------------------------
107 # フィルタ生成関数の定義
108 # ------------------------------------------------------------
109
110 def create_ideal_lowpass(shape, cutoff):
111     """
112     理想ローパスフィルタ: 指定した半径(cutoff)の内側を通し、外側をカット
113     """
114     rows, cols = shape
115     crow, ccol = rows // 2, cols // 2
116
117     y, x = np.ogrid[:rows, :cols]
118     # 中心からの距離の二乗を計算
119     dist_sq = (x - ccol)**2 + (y - crow)**2
120
121     mask = np.zeros(shape)
122     # 距離がcutoff以内なら1
123     mask[dist_sq <= cutoff**2] = 1
124     return mask
125
126 def create_gaussian_highpass(shape, cutoff):
127     """
128     ガウシアンハイパスフィルタ: 低周波を滑らかに減衰させ、高周波を通す
129     H(u,v) = 1 - exp(-D^2 / 2D0^2)
130     """
131     rows, cols = shape
132     crow, ccol = rows // 2, cols // 2
133
134     y, x = np.ogrid[:rows, :cols]
135     dist_sq = (x - ccol)**2 + (y - crow)**2
136
137     # ガウシアンローパスの逆（1から引く）
138     mask = 1 - np.exp(-dist_sq / (2 * (cutoff**2)))
139     return mask
140
141 # ------------------------------------------------------------
142 # フィルタの適用
143 # ------------------------------------------------------------
144
```

```
# パラメータ設定（遮断周波数）
D0_low = 30 # ローパス用
D0_high = 30 # ハイパス用

# マスクの作成
mask_ilpf = create_ideal_lowpass(img.shape, D0_low)
mask_ghpf = create_gaussian_highpass(img.shape, D0_high)

# 周波数領域でのフィルタリング（要素ごとの掛け算）
fshift_ilpf = fshift * mask_ilpf
fshift_ghpf = fshift * mask_ghpf

# 逆FFT処理（シフトを戻して逆変換し、実部を取る）
img_ilpf = np.fft.ifft2(np.fft.ifftshift(fshift_ilpf)).real
img_ghpf = np.fft.ifft2(np.fft.ifftshift(fshift_ghpf)).real

# ------------------------------------------------------------
# 結果の可視化
# ------------------------------------------------------------
plt.figure(figsize=(12, 8))

# --- 上段: 理想ローパスフィルタ ---
plt.subplot(2, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(2, 3, 2)
plt.title(f'Ideal Low Pass Filter Mask\n(Cutoff={D0_low})')
plt.imshow(mask_ilpf, cmap='gray', vmin=0, vmax=1)
plt.axis('off')

plt.subplot(2, 3, 3)
plt.title('Filtered Result (ILPF)')
plt.imshow(img_ilpf, cmap='gray') # 値の範囲は自動調整
plt.axis('off')

# --- 下段: ガウシアンハイパスフィルタ ---
plt.subplot(2, 3, 4)
plt.title('Original Image')
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(2, 3, 5)
plt.title(f'Gaussian High Pass Filter Mask\n(Cutoff={D0_high})')
plt.imshow(mask_ghpf, cmap='gray', vmin=0, vmax=1)
plt.axis('off')

plt.subplot(2, 3, 6)
```

```
plt.title('Filtered Result (GHPF)')
plt.imshow(np.abs(img_ghpf), cmap='gray') # 振幅を表示
plt.axis('off')

plt.tight_layout()
plt.show()
```

Listing 3　問題 3 2 次元 FFT と振幅スペクトル

問題 4: 周波数フィルタの応用

```python
# -*- coding: utf-8 -*-
"""問題2-4.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/14oortqIl9hjtngj_clywKmLNDSRRMBjh
"""

# ========================================
# 課題4: 2種類以上の周波数フィルタの適用と考察
# ========================================
# 1. 準備と画像読み込み
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive

# ドライブのマウント
drive.mount('/content/drive')

# パスとファイル名の設定
common_path = '/content/drive/MyDrive/img2025/image/'
filename = 'a2-5_gray_image.png'

# 画像読み込み
gray_img = cv2.imread(common_path + filename, cv2.IMREAD_GRAYSCALE)

# 画像の正方形トリミング（課題3の処理を継承）
h, w = gray_img.shape
crop_size = 512
target_size = min(h, w, crop_size)

center_y, center_x = h // 2, w // 2
start_y = center_y - target_size // 2
start_x = center_x - target_size // 2
img = gray_img[start_y:start_y+target_size, start_x:start_x+target_size]

print(f"画像読み込み完了: {img.shape}")

# 2. FFT実行とフィルタ関数の定義・適用

# --- FFTの実行 ---
f = np.fft.fft2(img)
fshift = np.fft.fftshift(f) # 直流成分を中心に移動

```

```python
# --- フィルタ生成関数の定義 ---
def create_ideal_lowpass(shape, cutoff):
    """理想ローパスフィルタ"""
    rows, cols = shape
    crow, ccol = rows // 2, cols // 2
    y, x = np.ogrid[:rows, :cols]
    dist_sq = (x - ccol)**2 + (y - crow)**2
    mask = np.zeros(shape)
    mask[dist_sq <= cutoff**2] = 1
    return mask

def create_gaussian_highpass(shape, cutoff):
    """ガウシアンハイパスフィルタ"""
    rows, cols = shape
    crow, ccol = rows // 2, cols // 2
    y, x = np.ogrid[:rows, :cols]
    dist_sq = (x - ccol)**2 + (y - crow)**2
    # ガウシアンローパスの逆（1から引く）
    mask = 1 - np.exp(-dist_sq / (2 * (cutoff**2)))
    return mask

# --- パラメータ設定と適用 ---
D0_low = 30 # ローパス用遮断周波数
D0_high = 30 # ハイパス用遮断周波数

# マスク作成とフィルタリング
mask_ilpf = create_ideal_lowpass(img.shape, D0_low)
mask_ghpf = create_gaussian_highpass(img.shape, D0_high)

fshift_ilpf = fshift * mask_ilpf
fshift_ghpf = fshift * mask_ghpf

# 逆FFT処理
img_ilpf = np.fft.ifft2(np.fft.ifftshift(fshift_ilpf)).real
img_ghpf = np.fft.ifft2(np.fft.ifftshift(fshift_ghpf)).real

print("フィルタ処理完了")

# 3. 結果の可視化
plt.figure(figsize=(12, 8))

# --- 上段: 理想ローパスフィルタ ---
plt.subplot(2, 3, 1)
plt.title('Original Image')
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(2, 3, 2)
plt.title(f'Ideal Low Pass Filter Mask\n(Cutoff={D0_low})')
```

```
plt.imshow(mask_ilpf, cmap='gray', vmin=0, vmax=1)
plt.axis('off')

plt.subplot(2, 3, 3)
plt.title('Filtered Result (ILPF)')
plt.imshow(img_ilpf, cmap='gray')
plt.axis('off')

# --- 下段: ガウシアンハイパスフィルタ ---
plt.subplot(2, 3, 4)
plt.title('Original Image')
plt.imshow(img, cmap='gray', vmin=0, vmax=255)
plt.axis('off')

plt.subplot(2, 3, 5)
plt.title(f'Gaussian High Pass Filter Mask\n(Cutoff={D0_high})')
plt.imshow(mask_ghpf, cmap='gray', vmin=0, vmax=1)
plt.axis('off')

plt.subplot(2, 3, 6)
plt.title('Filtered Result (GHPF)')
plt.imshow(np.abs(img_ghpf), cmap='gray') # 振幅を表示
plt.axis('off')

plt.tight_layout()
plt.show()
```

Listing 4　問題 4 周波数フィルタの応用