A1_Martinez_Kenneth

October 20, 2025

1 Machine Vision

2 Assignment 1 - Image Processing Fundamentals

2.0.1 Part A: Image types and file formats

Goal: understand types of images such as binary vs grayscale vs RGB and compare their extentions like PNG/JPEG/TIFF. Additionally the tradeoffs of lossy vs lossless.

[54]:
```
# Step #1: Loading OpenCV library to proccess images.
import cv2
import json
import numpy as np
import matplotlib.pyplot as plt
import os
```

[55]:
```
#Step #2: Load config.json for loading the images later
with open("../config/config.json", "r") as config:
    config_data = json.load(config)
    images = config_data["images"]
    print("config uploaded")
    print(images.items())
```

```
config uploaded
dict_items([('cameraman', {'path': '../data/cameraman.tif'}), ('coins', {'path':
'../data/coins.png'}), ('peppers', {'path': '../data/peppers.png'}), ('rice',
{'path': '../data/rice.png'}), ('partE', {'path': '../data/partE.jpg'})])
```

[56]:
```
#Step #3: Open peppers.png and cameraman.tiff
peppers = cv2.imread(images['peppers']['path'])
cameraman = cv2.imread(images['cameraman']['path'], cv2.IMREAD_GRAYSCALE)
#line is for part E, loading here to avoid an issue with the command down at␣
 ↪part E
line = cv2.imread(images['partE']['path'], cv2.IMREAD_GRAYSCALE)
#Step #4: Display size from peppers and cameraman
print(f"peppers.png size: {peppers.shape}")
print(f"cameraman.tif size: {cameraman.shape}")
```

```
peppers.png size: (384, 512, 3)
cameraman.tif size: (256, 256)
```

[57]:
```
#Display class from peppers and ccameraman
print(f"peppers.png class: {peppers.dtype}")
print(f"cameraman.tif class: {cameraman.dtype}")
```

```
peppers.png class: uint8
cameraman.tif class: uint8
```

[58]:
```
#Display range from peppers and ccameraman
print(f"peppers.png range (min, max): {peppers.min()}, {peppers.max()}")
print(f"cameraman.tif range (min, max): {cameraman.min()}, {cameraman.max()}")
```

```
peppers.png range (min, max): 0, 255
cameraman.tif range (min, max): 7, 253
```

[59]:
```
#Step #5: Convert peppers.png to grayscale
original_peppers_gray = cv2.cvtColor(peppers, cv2.COLOR_RGB2GRAY)
#showing the matrix values, in OpenCV when a image is loaded it gets␣
 ↪represented as a Numpy Array which is a matrix.
print(f"original_peppers_gray matrix values:\n {original_peppers_gray[:,:]}")
print(f"class: {original_peppers_gray.dtype}")
```

```
original_peppers_gray matrix values:
 [[43 45 46 … 41 40 40]
 [44 44 45 … 39 39 39]
 [42 43 44 … 39 40 39]
 …
 [96 95 99 … 31 33 31]
 [92 93 96 … 29 32 32]
 [93 94 94 … 29 31 33]]
class: uint8
```

[60]:
```
#Showing original_peppers_gray vs rgb version
fig, axes = plt.subplots(1, 3, figsize=(12,5))

#grayscale
axes[0].imshow(original_peppers_gray, cmap='gray')
axes[0].set_title('Grayscale')
axes[0].axis('off')

#rgb, OpenCV opens all images as bgr (blue, green, red) so we need to convert␣
 ↪bgr to rgb to get the right colors in the image
axes[1].imshow(cv2.cvtColor(peppers, cv2.COLOR_BGR2RGB))
axes[1].set_title('RGB')
axes[1].axis('off')
```

```
axes[2].imshow(cv2.cvtColor(cameraman, cv2.COLOR_BGR2RGB))
axes[2].set_title('Grayscale')
axes[2].axis('off')

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

[61]:
```
#Step #6: Save grayscale image as PNG, TIFF, JPEG at qualities 95, 75, 50 and␣
 ↪check size on disk.


cv2.imwrite('../outputs/original_peppers_gray.png', original_peppers_gray,␣
 ↪params = None)
cv2.imwrite('../outputs/original_peppers_gray.tiff', original_peppers_gray,␣
 ↪params = None)
cv2.imwrite('../outputs/original_peppers_gray_95.jpeg', original_peppers_gray,␣
 ↪[cv2.IMWRITE_JPEG_QUALITY, 95])
cv2.imwrite('../outputs/original_peppers_gray_75.jpeg', original_peppers_gray,␣
 ↪[cv2.IMWRITE_JPEG_QUALITY, 75])
cv2.imwrite('../outputs/original_peppers_gray_50.jpeg', original_peppers_gray,␣
 ↪[cv2.IMWRITE_JPEG_QUALITY, 50])

#check size on disk
#PNG image file
print(f"size on disk, original_peppers_gray.png: {os.path.getsize('../outputs/
 ↪original_peppers_gray.png') / 1024:.2f} KB")
#TIFF quality image file
print(f"size on disk, original_peppers_gray.tiff: {os.path.getsize('../outputs/
 ↪original_peppers_gray.tiff') / 1024:.2f} KB")
#95, 75, 50 quality image file
print(f"size on disk, original_peppers_gray_95: {os.path.getsize('../outputs/
 ↪original_peppers_gray_95.jpeg') / 1024:.2f} KB")
```

```
print(f"size on disk, original_peppers_gray_75: {os.path.getsize('../outputs/
 ↪original_peppers_gray_75.jpeg') / 1024:.2f} KB")
print(f"size on disk, original_peppers_gray_50: {os.path.getsize('../outputs/
 ↪original_peppers_gray_50.jpeg') / 1024:.2f} KB")
```

```
size on disk, original_peppers_gray.png: 90.17 KB
size on disk, original_peppers_gray.tiff: 106.58 KB
size on disk, original_peppers_gray_95: 47.90 KB
size on disk, original_peppers_gray_75: 18.14 KB
size on disk, original_peppers_gray_50: 11.91 KB
```

Since png is a lossless image file the image get just compressed down so the size of the image in disk went to 90.17 KB because Opencv has a default compression level of 3 for png images if the params = None. For tiff they are also lossless image so they get compressed down too but quality is conserved but the size in disk is reduced because opencv has a default compression with Lempel-Ziv-Welch (LZW). For JPEG images the quality could be manipulated to reduce the size of the image in disk so it makes JPEG images lossly type, at 95 we could expect good quality, at 75 we might start seeing some noise in the image, at 50 it would be more obvious we reduced the quality of the image to reduce its sizes during compression.

[62]:
```
#Step #7: Calculate MSE, Get PSNR of each image, Display montage of small crops␣
 ↪showing compression of any artifacts.
peppers_png = cv2.imread('../outputs/original_peppers_gray.png', cv2.
 ↪IMREAD_GRAYSCALE)
peppers_tiff = cv2.imread('../outputs/original_peppers_gray.tiff', cv2.
 ↪IMREAD_GRAYSCALE)
peppers_jpeg_95 = cv2.imread('../outputs/original_peppers_gray_95.jpeg', cv2.
 ↪IMREAD_GRAYSCALE)
peppers_jpeg_75 = cv2.imread('../outputs/original_peppers_gray_75.jpeg', cv2.
 ↪IMREAD_GRAYSCALE)
peppers_jpeg_50 = cv2.imread('../outputs/original_peppers_gray_50.jpeg', cv2.
 ↪IMREAD_GRAYSCALE)

#MSE function
def mse(img1, img2):
    h, w = img1.shape
    diff = cv2.subtract(img1, img2)
    err = np.sum(diff**2)
    return err / (float(h*w)) #mse

# original_peppers_gray vs peppers_png
print(f"MSE orginal_pepper_gray vs png format: {mse(original_peppers_gray,␣
 ↪peppers_png)}")

# original_peppers_gray vs peppers_tiff
print(f"MSE orginal_pepper_gray vs tiff format: {mse(original_peppers_gray,␣
 ↪peppers_tiff)}")
```

```
# original_peppers_gray vs peppers_jpeg_95
print(f"MSE orginal_pepper_gray vs jpeg at 95 format:␣
 ↪{mse(original_peppers_gray, peppers_jpeg_95)}")

# original_peppers_gray vs peppers_jpeg_75
print(f"MSE orginal_pepper_gray vs jpeg at 75 format:␣
 ↪{mse(original_peppers_gray, peppers_jpeg_75)}")

# original_peppers_gray vs peppers_jpeg_50
print(f"MSE orginal_pepper_gray vs jpeg at 50 format:␣
 ↪{mse(original_peppers_gray, peppers_jpeg_50)}")
```

```
MSE orginal_pepper_gray vs png format: 0.0
MSE orginal_pepper_gray vs tiff format: 0.0
MSE orginal_pepper_gray vs jpeg at 95 format: 0.6280161539713541
MSE orginal_pepper_gray vs jpeg at 75 format: 2.5709330240885415
MSE orginal_pepper_gray vs jpeg at 50 format: 4.0824025472005205
```

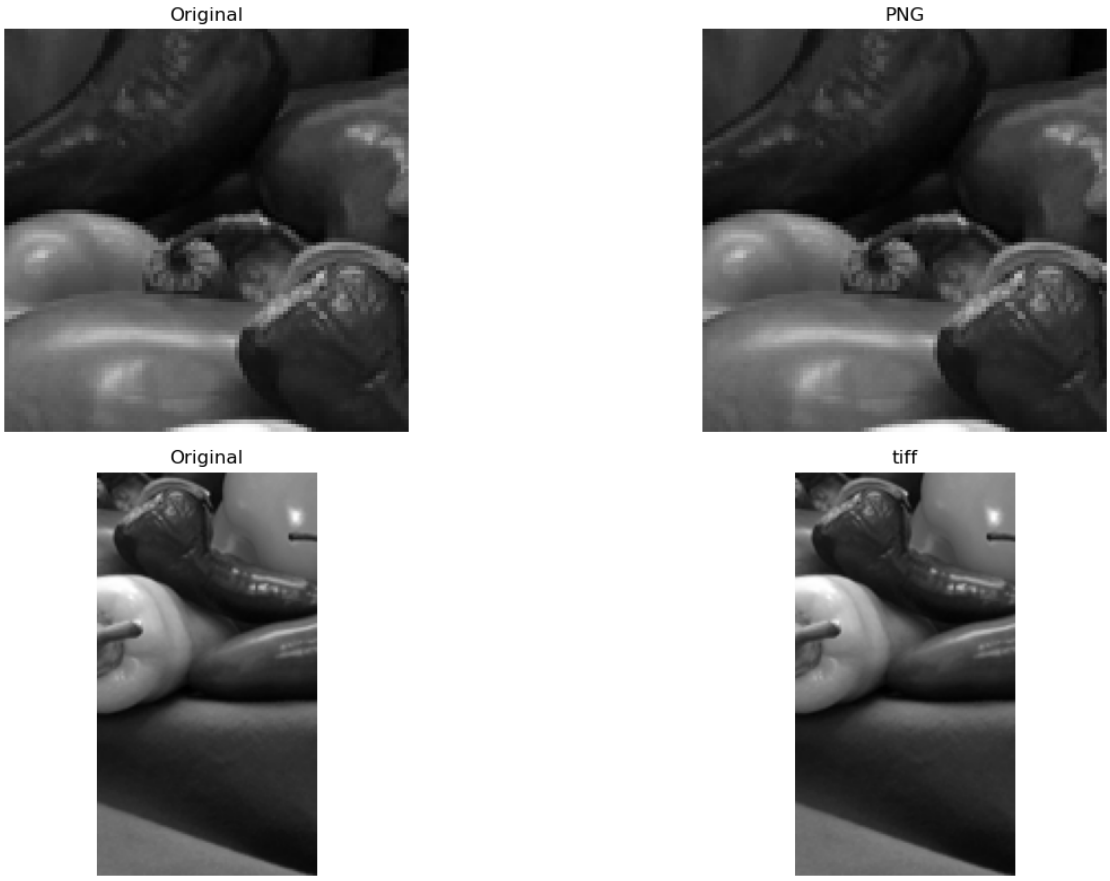
[63]:
```
#PSNR
print(f"PSRN png format: {cv2.PSNR(original_peppers_gray, peppers_png)}")
print(f"PSRN tiff format: {cv2.PSNR(original_peppers_gray, peppers_tiff)}")
print(f"PSRN png format: {cv2.PSNR(original_peppers_gray, peppers_jpeg_95)}")
print(f"PSRN png format: {cv2.PSNR(original_peppers_gray, peppers_jpeg_75)}")
print(f"PSRN png format: {cv2.PSNR(original_peppers_gray, peppers_jpeg_50)}")
```

```
PSRN png format: 361.20199909921956
PSRN tiff format: 361.20199909921956
PSRN png format: 47.17519007122286
PSRN png format: 40.90857561893803
PSRN png format: 38.582428282720606
```

For MSE we can see that png, tiff are 0.0 so the compression method previously done on them was confirmed to be lossless vs JPEG qualities being lossly and they have values as seen above. For PSRN the idea is to identify degraded image quality by obtaining higher values the compression method was better and the image is close to the quality of the original one. So we can see again above that png, tiff are better with the maximum signal value of the image and jpeg 95, 75, 50 got the values above.

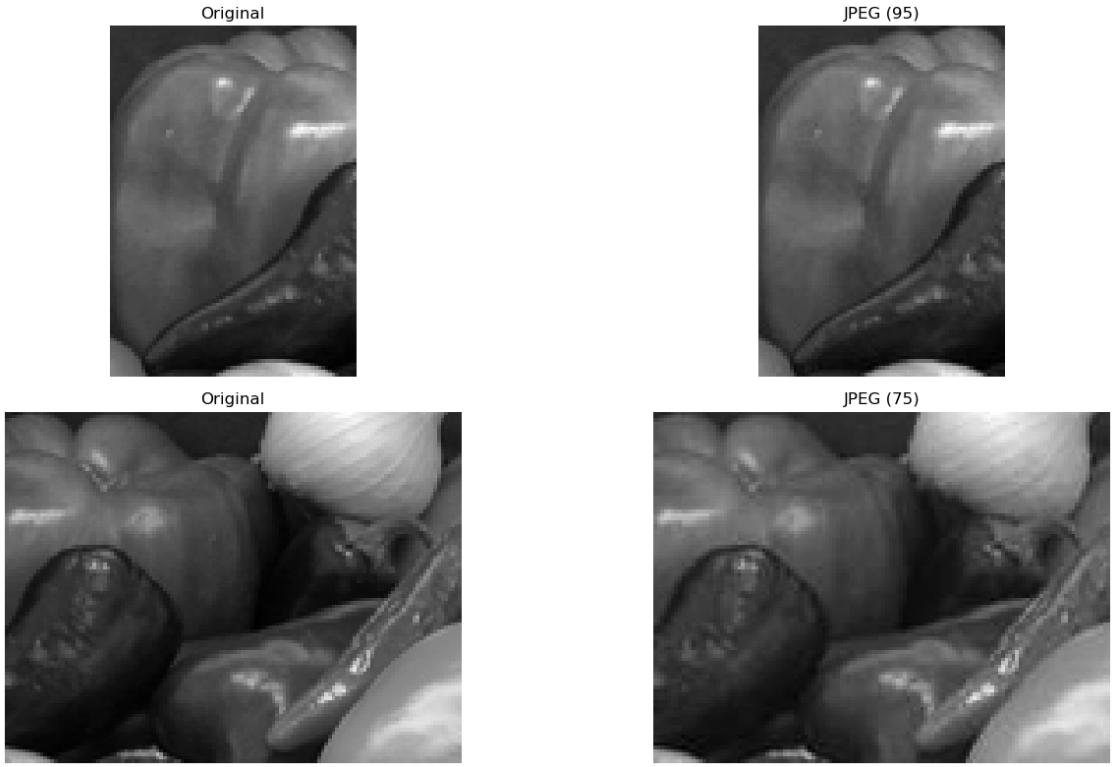
[64]:
```
#Display a montage of small crops showing compression artifacts
#Showing original_peppers_gray vs rgb version
fig, axes = plt.subplots(2, 2, figsize=(15,8))

#crop size
def crop_artifact(img, crop_size):
    x_start, y_start, x_end, y_end = crop_size
    crop = img[y_start:y_end, x_start:x_end]
    return crop
```

```
#original
axes[0, 0].imshow(crop_artifact(original_peppers_gray, [100, 150, 200, 250]),␣
 ↪cmap='gray')
axes[0, 0].set_title('Original')
axes[0, 0].axis('off')

#png
axes[0, 1].imshow(crop_artifact(peppers_png, [100, 150, 200, 250]), cmap='gray')
axes[0, 1].set_title('PNG')
axes[0, 1].axis('off')

#original
axes[1, 0].imshow(crop_artifact(original_peppers_gray, [150, 200, 250, 400]),␣
 ↪cmap='gray')
axes[1, 0].set_title('Original')
axes[1, 0].axis('off')

#tiff
axes[1, 1].imshow(crop_artifact(peppers_tiff, [150, 200, 250, 400]),␣
 ↪cmap='gray')
axes[1, 1].set_title('tiff')
axes[1, 1].axis('off')

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



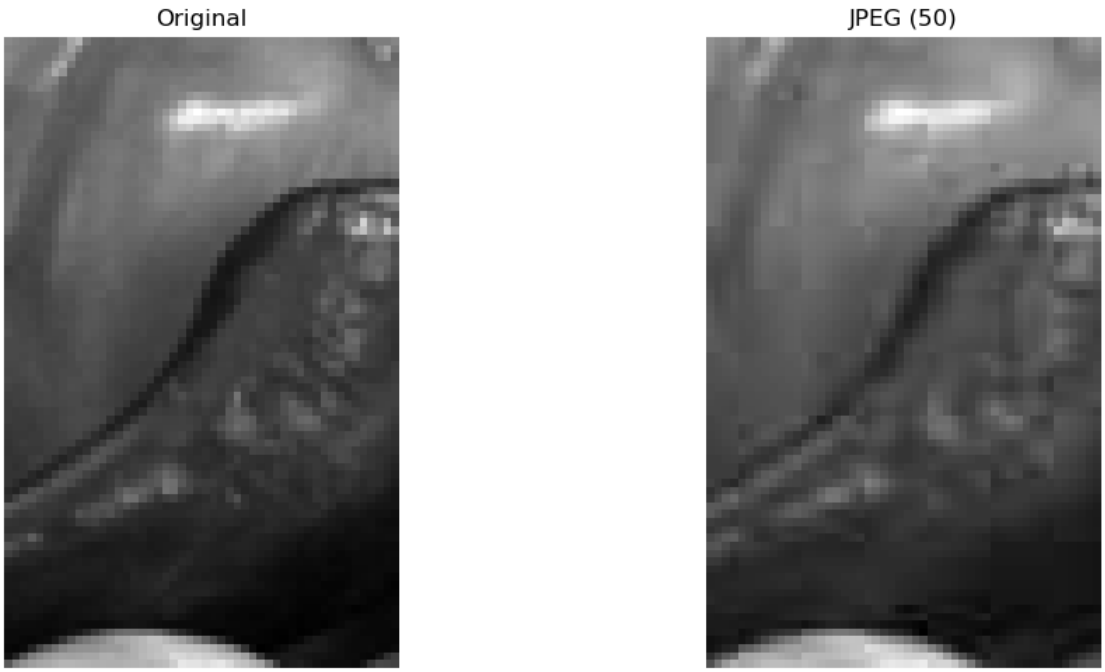
[65]:
```
fig, axes = plt.subplots(2, 2, figsize=(15,8))

#original
axes[0, 0].imshow(crop_artifact(original_peppers_gray, [70, 100, 140, 200]),␣
 ↪cmap='gray')
axes[0, 0].set_title('Original')
axes[0, 0].axis('off')

#JPEG 95
axes[0, 1].imshow(crop_artifact(peppers_jpeg_95, [70, 100, 140, 200]),␣
 ↪cmap='gray')
axes[0, 1].set_title('JPEG (95)')
axes[0, 1].axis('off')

#original
axes[1, 0].imshow(crop_artifact(original_peppers_gray, [120, 100, 250, 200]),␣
 ↪cmap='gray')
axes[1, 0].set_title('Original')
axes[1, 0].axis('off')
```

```
#JPEG 75
axes[1, 1].imshow(crop_artifact(peppers_jpeg_75, [120, 100, 250, 200]),␣
 ↪cmap='gray')
axes[1, 1].set_title('JPEG (75)')
axes[1, 1].axis('off')

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



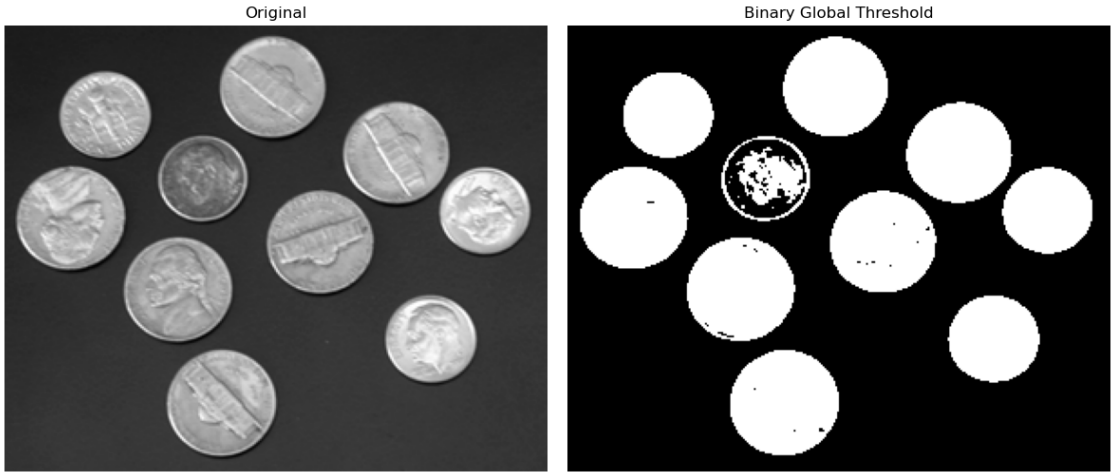
[66]:
```
fig, axes = plt.subplots(1, 2, figsize=(12,5))

#original
axes[0].imshow(crop_artifact(original_peppers_gray, [100, 120, 150, 200]),␣
 ↪cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

#JPEG 95
axes[1].imshow(crop_artifact(peppers_jpeg_50, [100, 120, 150, 200]),␣
 ↪cmap='gray')
axes[1].set_title('JPEG (50)')
```

```
axes[1].axis('off')

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



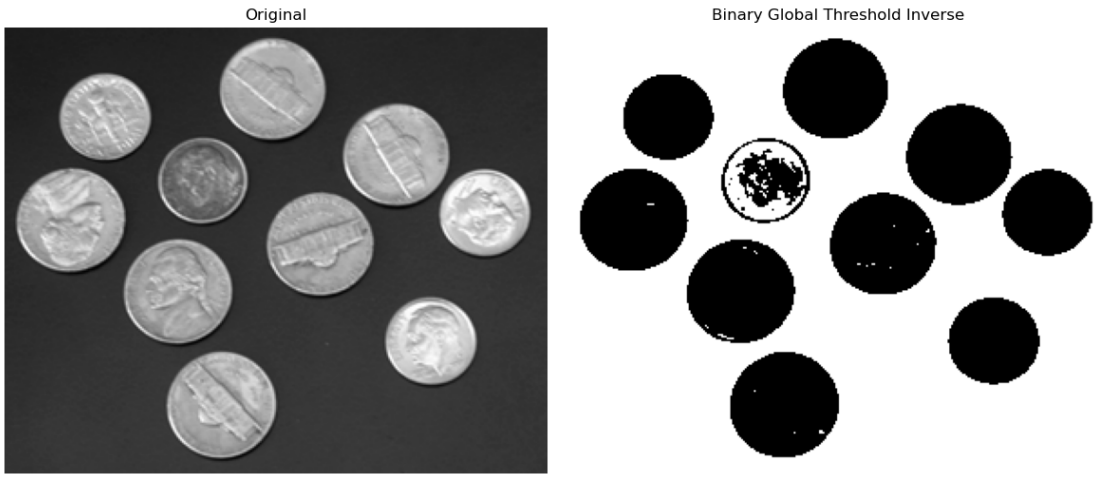
[67]:
```
#Step #8: Binary image
coins = cv2.imread(images['coins']['path'], cv2.IMREAD_GRAYSCALE)
ret, binary_thresh = cv2.threshold(coins, 127, 255, cv2.THRESH_BINARY)

fig, axes = plt.subplots(1, 2, figsize=(12,5))

#original
axes[0].imshow(coins, cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

#Binary threshold
axes[1].imshow(binary_thresh, cmap = 'gray')
axes[1].set_title('Binary Global Threshold')
axes[1].axis('off')

plt.tight_layout()
plt.show()

foreground = np.count_nonzero(binary_thresh)  # Count non-zero (white) pixels
```

```
background = binary_thresh.size - foreground   # Total pixels minus foreground

print(f"Foreground: {foreground}, Background: {background}")
```



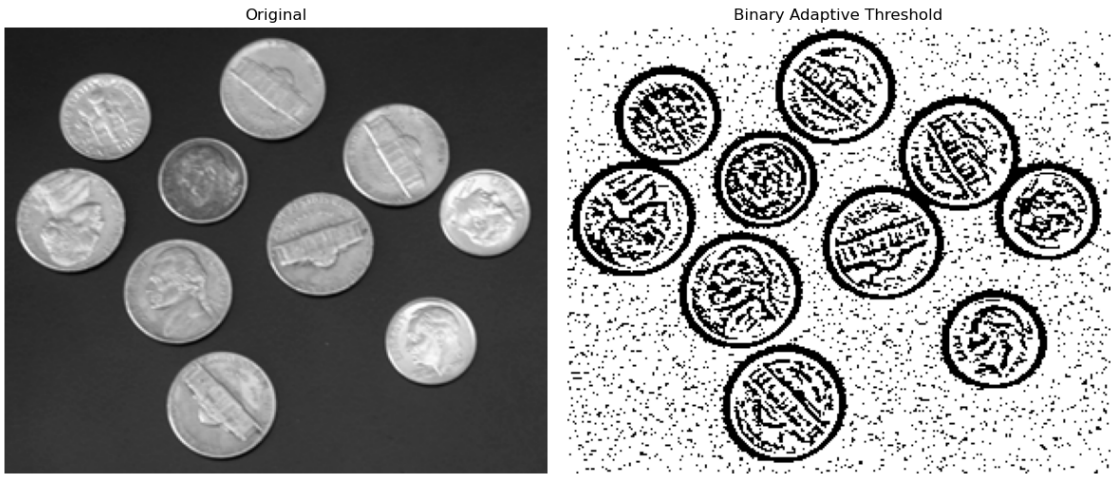

```
Foreground: 22238, Background: 51562
```

[68]:
```
ret, binary_thresh_inv = cv2.threshold(coins, 127, 255, cv2.THRESH_BINARY_INV)

fig, axes = plt.subplots(1, 2, figsize=(12,5))

#original
axes[0].imshow(coins, cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

#Binary threshold inverse
axes[1].imshow(binary_thresh_inv, cmap = 'gray')
axes[1].set_title('Binary Global Threshold Inverse')
axes[1].axis('off')

plt.tight_layout()
plt.show()

foreground = np.count_nonzero(binary_thresh_inv)  # Count non-zero (white)␣
↪pixels
background = binary_thresh_inv.size - foreground   # Total pixels minus␣
↪foreground

print(f"Foreground: {foreground}, Background: {background}")
```



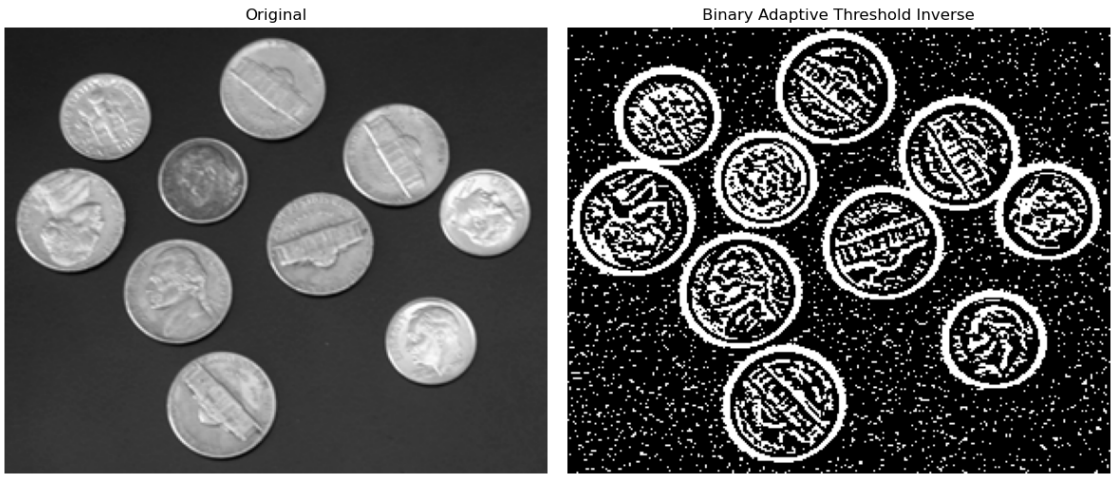

```
Foreground: 51562, Background: 22238
```

[69]:
```
adaptive_thresh = cv2.adaptiveThreshold(coins, 255, cv2.
  ↪ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)

fig, axes = plt.subplots(1, 2, figsize=(12,5))

#original
axes[0].imshow(coins, cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

#Binary adaptive threshold
axes[1].imshow(adaptive_thresh, cmap = 'gray')
axes[1].set_title('Binary Adaptive Threshold')
axes[1].axis('off')

plt.tight_layout()
plt.show()

foreground = np.count_nonzero(adaptive_thresh)  # Count non-zero (white) pixels
background = adaptive_thresh.size - foreground   # Total pixels minus foreground

print(f"Foreground: {foreground}, Background: {background}")
```




```
Foreground: 56055, Background: 17745
```

[70]:
```
adaptive_thresh_inv = cv2.adaptiveThreshold(coins, 255, cv2.
 ↪ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 11, 2)

fig, axes = plt.subplots(1, 2, figsize=(12,5))

#original
axes[0].imshow(coins, cmap='gray')
axes[0].set_title('Original')
axes[0].axis('off')

#Binary adaptive threshold
axes[1].imshow(adaptive_thresh_inv, cmap = 'gray')
axes[1].set_title('Binary Adaptive Threshold Inverse')
axes[1].axis('off')

plt.tight_layout()
plt.show()

foreground = np.count_nonzero(adaptive_thresh_inv)  # Count non-zero (white)␣
 ↪pixels
background = adaptive_thresh_inv.size - foreground   # Total pixels minus␣
 ↪foreground

print(f"Foreground: {foreground}, Background: {background}")
```




```
Foreground: 17745, Background: 56055
```

2.0.2 Reflection of part A

1- PNG and TIFF were lossless and JPEG was lossly. File size could track visual quality in the case of JPEG images if their quality is being modified like the case of 50, we got a size of 11.91 KB which means the quality of the image probably was loose a bit to compress it down.

2- PSNR is adequate to compare JPEG qualities because MSE do not take into account any biological factor such as human vision, MSE is pure numerical evaluation of the error between original image and compressed one. So PSRN look at he maximum signal of the image to determine difference between images.

3- Global vs Adaptive binarizing methods, global takes a single threshold value for the entire image meanwhile adaptive calculates different threshold for smaller regions in the image to consider the lighting contions an image could have.

4- When inverting the image, the background change from black to white so if whe have a line in black that a robot could follow inverting the image could give us better track of the width of the image and the robot could precisely follow it.

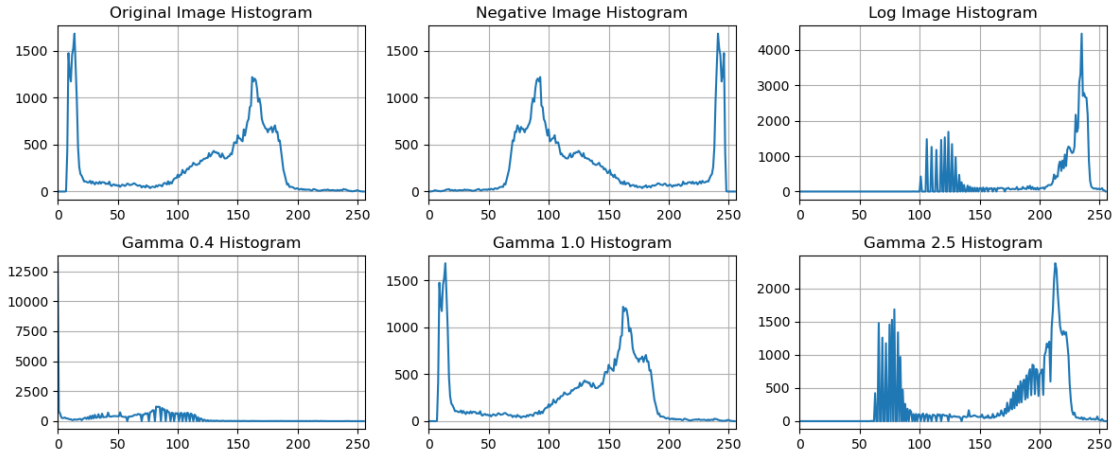
2.0.3 Part B: Intensity transforms and geometry

[71]:
```
#Step 1: Creating a negative, log and gamma image
#Negative image
cameraman_negative = abs(255-cameraman)

#Log image
c = 255 / np.log(1 + np.max(cameraman))
cameraman_log = c * (np.log(cameraman + 1))
cameraman_log = np.array(cameraman_log, dtype = np.uint8)

#Gamma
```

```
def adjust_gamma(img, gamma_value):
    """It builds a lookup table to map pixel values to their adjusted gamma␣
 ↪values"""
    inv_gamma = 1.0/gamma_value
    table = np.array([((i / 255.0) ** inv_gamma) * 255 for i in np.arange(0,␣
 ↪256)]).astype("uint8")
    return cv2.LUT(img, table)

cameraman_gamma_04 = adjust_gamma(cameraman, 0.4)
cameraman_gamma_1 = adjust_gamma(cameraman, 1.0)
cameraman_gamma_25 = adjust_gamma(cameraman, 2.5)

fig, axes = plt.subplots(2, 3, figsize=(12,5))

#original
axes[0,0].imshow(cameraman, cmap='gray')
axes[0,0].set_title('Original')
axes[0,0].axis('off')

#Negative image
axes[0,1].imshow(cameraman_negative, cmap='gray')
axes[0,1].set_title('Negative Image')
axes[0,1].axis('off')

#Log Image
axes[0,2].imshow(cameraman_log, cmap='gray')
axes[0,2].set_title('Log Image')
axes[0,2].axis('off')

#Gamma 4.0
axes[1,0].imshow(cameraman_gamma_04, cmap='gray')
axes[1,0].set_title('Gamma 0.4')
axes[1,0].axis('off')

#Gamma 1.0
axes[1,1].imshow(cameraman_gamma_1, cmap='gray')
axes[1,1].set_title('Gamma 1.0')
axes[1,1].axis('off')

#Gamma 2.5
axes[1,2].imshow(cameraman_gamma_25, cmap='gray')
axes[1,2].set_title('Gamma 2.5')
axes[1,2].axis('off')

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

[72]:
```
#Calculating Histogram and showing them
original_hist = cv2.calcHist([cameraman], [0], None, [256], [0, 256])
negative_hist = cv2.calcHist([cameraman_negative], [0], None, [256], [0, 256])
log_hist = cv2.calcHist([cameraman_log], [0], None, [256], [0, 256])
gamma04_hist = cv2.calcHist([cameraman_gamma_04], [0], None, [256], [0, 256])
gamma1_hist = cv2.calcHist([cameraman_gamma_1], [0], None, [256], [0, 256])
gamma25_hist = cv2.calcHist([cameraman_gamma_25], [0], None, [256], [0, 256])

fig, axes = plt.subplots(2, 3, figsize=(12,5))

#original
axes[0,0].plot(original_hist)
axes[0,0].set_title('Original Image Histogram')
axes[0,0].set_xlim([0, 256])
axes[0,0].grid(True)

#Negative image
axes[0,1].plot(negative_hist)
axes[0,1].set_title('Negative Image Histogram')
axes[0,1].set_xlim([0, 256])
axes[0,1].grid(True)

#Log Image
axes[0,2].plot(log_hist)
axes[0,2].set_title('Log Image Histogram')
axes[0,2].set_xlim([0, 256])
axes[0,2].grid(True)
```

```
#Gamma 4.0
axes[1,0].plot(gamma04_hist)
axes[1,0].set_title('Gamma 0.4 Histogram')
axes[1,0].set_xlim([0, 256])
axes[1,0].grid(True)

#Gamma 1.0
axes[1,1].plot(gamma1_hist)
axes[1,1].set_title('Gamma 1.0 Histogram')
axes[1,1].set_xlim([0, 256])
axes[1,1].grid(True)

#Gamma 2.5
axes[1,2].plot(gamma25_hist)
axes[1,2].set_title('Gamma 2.5 Histogram')
axes[1,2].set_xlim([0, 256])
axes[1,2].grid(True)

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



[73]:
```
#Step 2: Contrast streching & equalization

def contrastStretch(I, r1, r2, s1, s2):
    s = (I.astype(float) - r1) * (s2 - s1) / (r2 - r1) + s1
    s = np.clip(s, 0, 255)
    return s.astype(np.uint8)

cameraman_contrastStretch = contrastStretch(cameraman, np.min(cameraman), np.
 ↪max(cameraman), 0, 255 )
```

```
print(f"Original - Min: {np.min(cameraman)}, Max: {np.max(cameraman)}")
print(f"Stretched - Min: {np.min(cameraman_contrastStretch)}, Max: {np.
 ↪max(cameraman_contrastStretch)}")


fig, axes = plt.subplots(1, 2, figsize=(12,5))

axes[0].imshow(cameraman, cmap='gray')
axes[0].set_title('Original Cameraman')
axes[0].axis('off')

axes[1].imshow(cameraman_contrastStretch, cmap='gray')
axes[1].set_title('Contrast Stretched')
axes[1].axis('off')
```

```
Original - Min: 7, Max: 253
Stretched - Min: 0, Max: 255
```

[73]:
```
(np.float64(-0.5), np.float64(255.5), np.float64(255.5), np.float64(-0.5))
```

[74]:
```
#Comparing with equalizeHist in opencv which is equal to histeq, there is not␣
 ↪imadjust in opencv
cameraman_eqhist = cv2.equalizeHist(cameraman)
results = np.hstack((cameraman, cameraman_eqhist))

plt.figure(figsize=(12, 5))
plt.imshow(results, cmap='gray')
plt.title("Cameraman Original vs equalizeHist OpenCV")
plt.axis('off')
```

```
plt.show()
```

Since OpenCV do not have a built-in function for imadjust like Matlab, the comparison would be between both the custom function vs equalizeHist results.

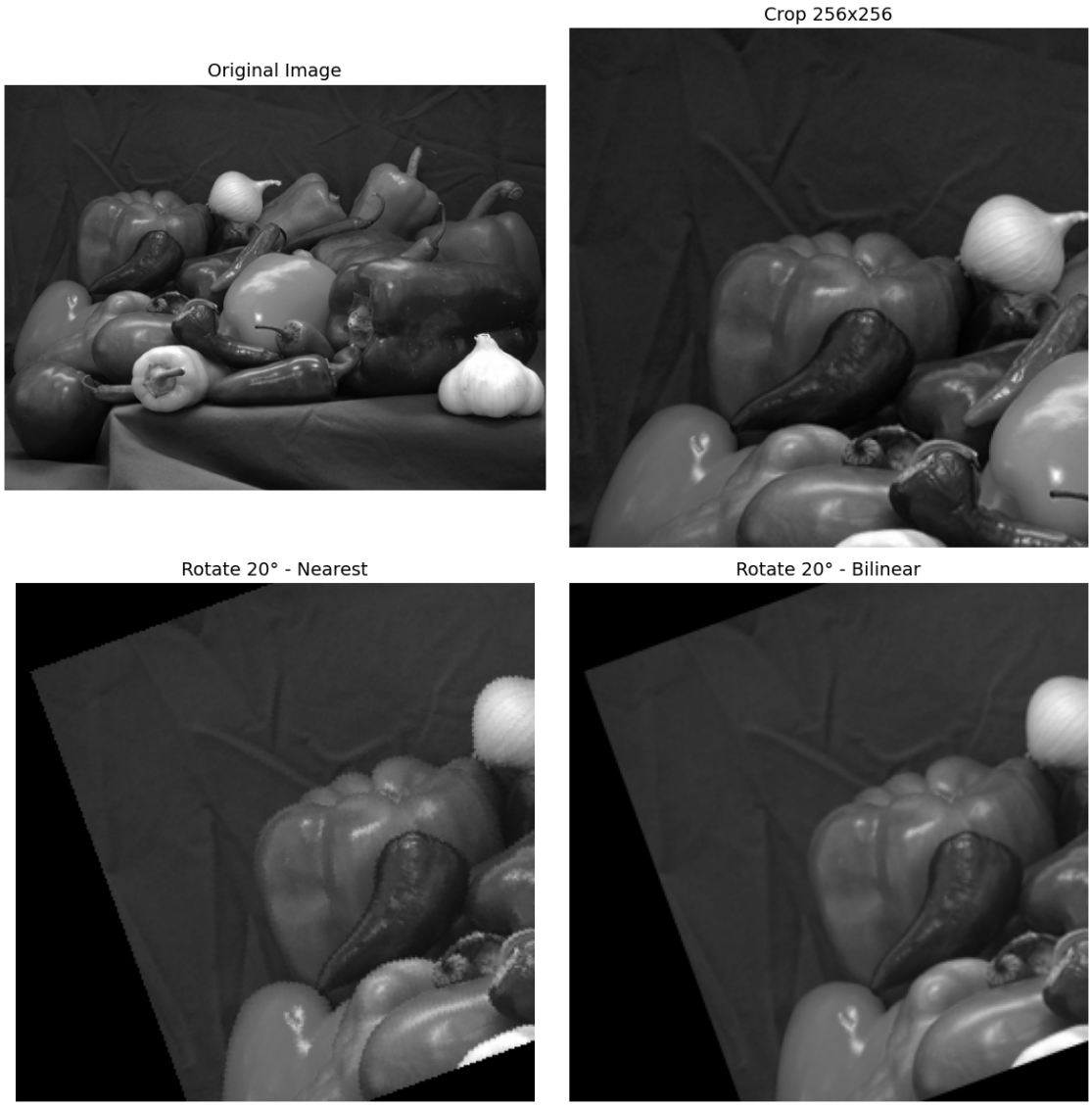
1- contrastStretch: has a linear transformation that proportinally spreads existing values from [7, 253] to [0, 255] and it preserves the shape of the histogram which have minimal visual change in the resultant image.

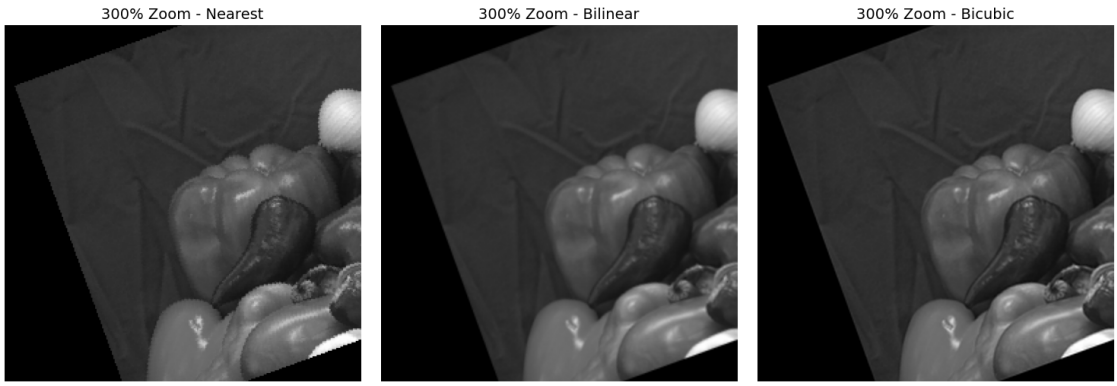
2- equalizeHist: has a non-linear transformation over the histogram shape of the image, it would try to create a uniform histogram by spreading equal number of pixels at each intensity level that results in having darker colors for example in the coat and shadows, background and buildings look brigther, there is more contrast, and there is more visible noise in the image.

[75]:
```
#Step 3: Crop & rotate (interpolation)

peppers_256x256 = peppers_png[0:256, 0:256]

#Rotation function to pass a single time and not individually by interpolation␣
 ↪method
def rotate_img(im, angle, interp_metod, scale=1.0):
    h, w = im.shape[:256]
    center = (w // 2, h // 256)
    rotation_matrix = cv2.getRotationMatrix2D(center, angle, scale)
    rotated_img = cv2.warpAffine(im, rotation_matrix, (w, h),␣
 ↪flags=interp_metod)
    return rotated_img
```

```
R20_nearest = rotate_img(peppers_256x256, 20, cv2.INTER_NEAREST)
R20_bilinear = rotate_img(peppers_256x256, 20, cv2.INTER_LINEAR) #INTERN_LINEAR␣
 ↪is bilinear interpolation in OpenCV
R20_bicubic = rotate_img(peppers_256x256, 20, cv2.INTER_CUBIC) #INTERN_CUBIC is␣
 ↪Bicubic interpolation in OpenCV

#300% Zoom and comment on aliasing and smoothness
Z300_nearest = cv2.resize(R20_nearest, None,fx=3.0, fy=3.0, interpolation=cv2.
 ↪INTER_NEAREST)
Z300_bilinear = cv2.resize(R20_bilinear, None,fx=3.0, fy=3.0, interpolation=cv2.
 ↪INTER_LINEAR)
Z300_bicubic = cv2.resize(R20_bicubic, None,fx=3.0, fy=3.0, interpolation=cv2.
 ↪INTER_CUBIC)

fig1, axes1 = plt.subplots(2, 2, figsize=(12, 12))

#original
axes1[0,0].imshow(peppers_png, cmap='gray')
axes1[0,0].set_title('Original Image', fontsize=14)
axes1[0,0].axis('off')

#Crop 256x256
axes1[0,1].imshow(peppers_256x256, cmap='gray')
axes1[0,1].set_title('Crop 256x256', fontsize=14)
axes1[0,1].axis('off')

#rotate 20 degrees nearest
axes1[1,0].imshow(R20_nearest, cmap='gray')
axes1[1,0].set_title('Rotate 20° - Nearest', fontsize=14)
axes1[1,0].axis('off')

#rotate 20 Deg, bilinear
axes1[1,1].imshow(R20_bilinear, cmap='gray')
axes1[1,1].set_title('Rotate 20° - Bilinear', fontsize=14)
axes1[1,1].axis('off')

plt.tight_layout()
plt.show()

# Figure 2: 300% Zoom Comparison
fig2, axes2 = plt.subplots(1, 3, figsize=(15, 5))

#zoom 300% nearest interp.
axes2[0].imshow(Z300_nearest, cmap='gray')
axes2[0].set_title('300% Zoom - Nearest', fontsize=14)
axes2[0].axis('off')
```

```
#zoom 300% bilinar interp.
axes2[1].imshow(Z300_bilinear, cmap='gray')
axes2[1].set_title('300% Zoom - Bilinear', fontsize=14)
axes2[1].axis('off')

#zoom 300% bicubic interp.
axes2[2].imshow(Z300_bicubic, cmap='gray')
axes2[2].set_title('300% Zoom - Bicubic', fontsize=14)
axes2[2].axis('off')

plt.tight_layout()
plt.show()

print(f"Rotated shape: {R20_nearest.shape}")
print(f"Zoomed shape: {Z300_nearest.shape}")
```






```
Rotated shape: (256, 256)
Zoomed shape: (768, 768)
```

In above code execution we can see that nearest interpolation is the worst between all thre cases producing low quality images when rotating and in the 300% zoom it can be seen some blocky areas around the edges. For Bilinear, it smoothed out some of the blocky patterns but there is some blur effects around the edges. For bicubic, we can see a smooth transition better than the other 2 methods and has least aliasing and higher smoothness.

2.0.4 Reflection:

1. When is gamma < 1 helpful? What about gamma > 1? Gamma < 1 is helpful in situations where the image is too dark and we might want to apply some brightness to it. While gamma > 1 is better for images that are too bright so we can make them more darker if we apply values above 1.
2. What visual evidence tells you which interpolation was used? For determining which interpolation method was used we can simply look at the image, if it has some blocky edges then nearest interpolation was used. If the image has some blur edges bilinear interpolation was the method used and if the image have smooth edges and least aliasing then it could be said that bicubic interpolation was used.
3. In Robotics, how might histogram equalization help a mobile robot to navigate? Histogram equalization could improves the robot's vision when lighting conditions are different. It could be used to enhance contrast, find hidden details for example in dark corners or shadowed areas in the image. Importantly, it can be used to normalize the lighting conditions.

2.0.5 Part C: Spatial filtering

Goal: Smooth and sharpen images via neighborhood operations.

[76]:
```
#Step 1: Moving averages
```

```python
#function movingAverage1D
def movingAverage1D(x, w):
    output_array = np.zeros_like(x, dtype=float)
    pad = w // 2
    x_padded = np.pad(x, pad, mode='edge')
    for i in range(len(x)):
        window = x_padded[i : i + w]
        output_array[i] = np.mean(window) #normalized here
    return output_array

#Fake 1D signal to validate
signal_1D = [12, 2, 5, 9, 10, 20]
signal_movingAverage = movingAverage1D(signal_1D, 3)
print(f"Moving Average 1D of a signal: {signal_movingAverage}")
```

```
Moving Average 1D of a signal: [ 8.66666667  6.33333333  5.33333333  8.
13.         16.66666667]
```

Validate results: Step 1: Padding pad = 3 // 2 = 1

x_padded with edge replication: [12, 12, 2, 5, 9, 10, 20, 20] first value (12) is replicated at the start, last value (20) is replicated at the end

Step 2: averages for each position

Position 0: Window: x_padded[0:3] = [12, 12, 2] Average: (12 + 12 + 2) / 3 = 26 / 3 = 8.67

Position 1: Window: x_padded[1:4] = [12, 2, 5] Average: (12 + 2 + 5) / 3 = 19 / 3 = 6.33

Position 2: Window: x_padded[2:5] = [2, 5, 9] Average: (2 + 5 + 9) / 3 = 16 / 3 = 5.33

Position 3: Window: x_padded[3:6] = [5, 9, 10] Average: (5 + 9 + 10) / 3 = 24 / 3 = 8.0

Position 4: Window: x_padded[4:7] = [9, 10, 20] Average: (9 + 10 + 20) / 3 = 39 / 3 = 13.0

Position 5: Window: x_padded[5:8] = [10, 20, 20] Average: (10 + 20 + 20) / 3 = 50 / 3 = 16.67

[77]:
```python
#function boxFilter2D
def boxFilter2D(I, k):
    output = np.zeros_like(I, dtype=float)
    pad = k // 2
    I_padded = np.pad(I, pad, mode='edge')
    for i in range(I.shape[0]):
        for j in range(I.shape[1]):

            window = I_padded[i:i+k, j:j+k]
            output[i, j] = np.mean(window)
    return output

#apply custom function
k = 3
```

```
cameraman_boxF = boxFilter2D(cameraman, k)

#apply equivalent imfilter matlab function but in OpenCV
kernel = np.ones((k, k))/ (k*k)
filter2d = cv2.filter2D(cameraman, -1, kernel, borderType=cv2.BORDER_REPLICATE)

res = np.hstack((cameraman_boxF, filter2d))

plt.figure(figsize=(12, 5))
plt.imshow(res, cmap='gray')
plt.title("Custom boxFilter2D vs filter2D OpenCV")
plt.axis('off')
plt.show()

print(f"Max difference: {np.max(cameraman_boxF - filter2d)}")
print(f"Mean difference: {np.mean(cameraman_boxF - filter2d)}")
```


```
Max difference: 0.4444444444444571
Mean difference: 0.0012969970703125017
```

Comparing the custom boxFilter2d vs filter2D(built-in method OpenCV), we can see that the maximum difference is about 0.44 and the mean difference is around 0.0013 so we can conclude that the custom method is working as intended and their is negligible difference between custom and built-in method.

[78]:
```
#Step 2: Gaussian smoothing
```

```
# custom function, it is to apply gaussian filter passing different sigma␣
 ↪values. It use the equivalent function to imgaussfit in opencv
#called GaussianBlur
def apply_gaussian(img, sigma):
    ksize = int(2 * np.ceil(3 * sigma) + 1)
    return cv2.GaussianBlur(img, (ksize, ksize), sigmaX=sigma)

sigmas = [1, 2, 4]
filtered_imgs = []

for sigma in sigmas:
    filtered_img = apply_gaussian(cameraman, sigma)
    filtered_imgs.append(filtered_img)

# Display comparison
fig, axes = plt.subplots(2, 2, figsize=(14, 14))

axes[0,0].imshow(cameraman, cmap='gray')
axes[0,0].set_title('Original Cameraman', fontsize=14)
axes[0,0].axis('off')

axes[0,1].imshow(filtered_imgs[0], cmap='gray')
axes[0,1].set_title('Gaussian sigma=1 (slight blur)', fontsize=14)
axes[0,1].axis('off')

axes[1,0].imshow(filtered_imgs[1], cmap='gray')
axes[1,0].set_title('Gaussian sigma=2 (moderate blur)', fontsize=14)
axes[1,0].axis('off')

axes[1,1].imshow(filtered_imgs[2], cmap='gray')
axes[1,1].set_title('Gaussian sigma=4 (heavy blur)', fontsize=14)
axes[1,1].axis('off')

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

Comparison

1. sigma = 1, shows slight blur with almost all details still preserved for example the tripod legs remain sharp and the cameraman's facial features are visible. There was minimal noise suppression and it is good to use for cases where slight smoothing is needed.
2. sigma = 2, it shows a more moderate blur in the image, there is more softening applied to the image and some features are starting to get more blur. It remove the medium-frequency noise while kept the objects recognizable.
3. sigma = 4, it shows a heavier blur where significant loss of the objects details are extremely blur. It did remove all noise but it introduced too much detail loss which gave the aspect of the image being out of focus and hard to recognize some objects.
4. Overall, sigma 1 - 2 are okay to get some smothness results without loosing too much detail of the image. sigma = 4 could be useful when there is more noise in the image, which

cameraman.tiff probably did not have so applying that value introduced too much detail loss.

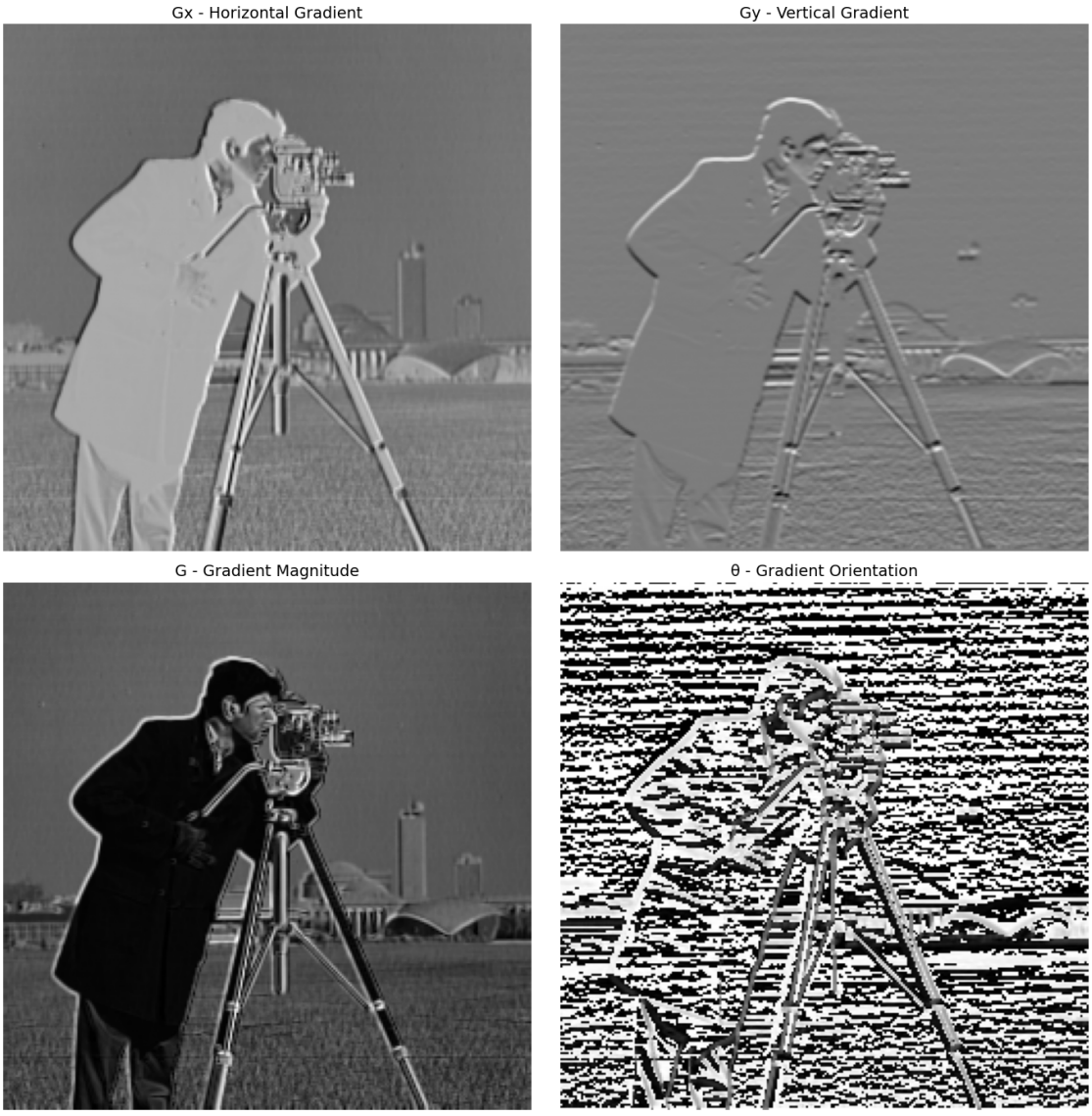
[79]:
```
#Step 3: unsharp masking and high-boost filtering

#custom function
def unsharp_mask(img, blur_img, alpha):
    img_f = img.astype(float)
    img_blur_f = blur_img.astype(float)

    img_sharp = img_f + alpha * (img_f - img_blur_f)

    img_sharp = np.clip(img_sharp, 0, 255)
    return img_sharp.astype(np.uint8)

alphas = [0.5, 1.0, 1.5]
unsharp_imgs = []

for alpha in alphas:
    unsharp = unsharp_mask(cameraman, filtered_imgs[1], alpha)
    unsharp_imgs.append(unsharp)

# Display comparison
fig, axes = plt.subplots(2, 2, figsize=(14, 14))

axes[0,0].imshow(cameraman, cmap='gray')
axes[0,0].set_title('Original Cameraman', fontsize=14)
axes[0,0].axis('off')

axes[0,1].imshow(unsharp_imgs[0], cmap='gray')
axes[0,1].set_title('Unsharp Mask alpha=0.5', fontsize=14)
axes[0,1].axis('off')

axes[1,0].imshow(unsharp_imgs[1], cmap='gray')
axes[1,0].set_title('Unsharp Mask alpha=1.0', fontsize=14)
axes[1,0].axis('off')

axes[1,1].imshow(unsharp_imgs[2], cmap='gray')
axes[1,1].set_title('Unsharp Mask alpha=1.5', fontsize=14)
axes[1,1].axis('off')

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

[80]:
```
#Step 4: Laplacian

laplacian = cv2.Laplacian(cameraman, cv2.CV_64F, ksize=3) # CV_64F according to␣
↪documentation is used to handle negative values.
laplacian_abs = np.uint8(np.absolute(laplacian))
resL = np.hstack((cameraman, laplacian_abs))

plt.figure(figsize=(12, 5))
plt.imshow(resL, cmap='gray')
plt.title("Original Image vs Laplacian Image")
plt.axis('off')
plt.show()
```

Halo Effect Explanation The halo effect is a consequence of applying Laplacian filtering to an image. This effect appears as a halo-like outline around the edges of objects due to the Laplacian operator producing both positive and negative responses when intensity transitions occur sharply. Specifically, the brighter side of an edge receives a bright halo, while the darker side receives a dark halo. The prominence and width of these halos depend on the sharpness of the intensity transition sharper edges produce more pronounced halos.

[81]:
```
#Step 5: SObel gradients

#custom sobelXY
def sobelXY(img):

    #Sobel Kernels
    Sx = np.array([[-1, 0, 1],
                   [-2, 0, 0],
                   [-1, 0, 1]], dtype=np.float32)

    Sy = np.array([[1, 2, 1],
                   [0, 0, 0],
                   [-1, -2, -1]], dtype=np.float32)

    #Convolve
    Gx = cv2.filter2D(img, cv2.CV_64F, Sx, borderType=cv2.BORDER_REPLICATE)
    Gy = cv2.filter2D(img, cv2.CV_64F, Sy, borderType=cv2.BORDER_REPLICATE)

    #Gradient
    G = np.hypot(Gx, Gy)
```

```
    #Compute gradient orientation
    thetha = np.arctan2(Gy, Gx)

    return Gx, Gy, G, thetha

Gx, Gy, G, thetha = sobelXY(cameraman)

fig, axes = plt.subplots(2, 2, figsize=(14, 14))

axes[0,0].imshow(Gx, cmap='gray')
axes[0,0].set_title('Gx - Horizontal Gradient', fontsize=14)
axes[0,0].axis('off')

axes[0,1].imshow(Gy, cmap='gray')
axes[0,1].set_title('Gy - Vertical Gradient', fontsize=14)
axes[0,1].axis('off')

axes[1,0].imshow(G, cmap='gray')
axes[1,0].set_title('G - Gradient Magnitude', fontsize=14)
axes[1,0].axis('off')

axes[1,1].imshow(thetha, cmap='gray')
axes[1,1].set_title('  - Gradient Orientation', fontsize=14)
axes[1,1].axis('off')

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



2.0.6 Reflection

1. Why do larger kernels blur more? What tradeoffs did you see with sigma? When larger kernels are used the more pixels are averaged together resulting in more local variations being smoothed out and they would look very blur. Depending the value of sigma if it is for example there would be some blur but not enough to loose fine details or sharp edges, if we increment the value of sigma the more noise would be remove from the image but the tradeoff is loosing image fine details and they would become too blury and the fine details might not be able to be recognized.

2. How is a high-pass filter related to "original minus low-pass"? We know that a high-pass filter removes low-frequency information from an image, preserving only high-frequency details like edges and textures. When a low-pass filter is used, it removes high-frequency information,

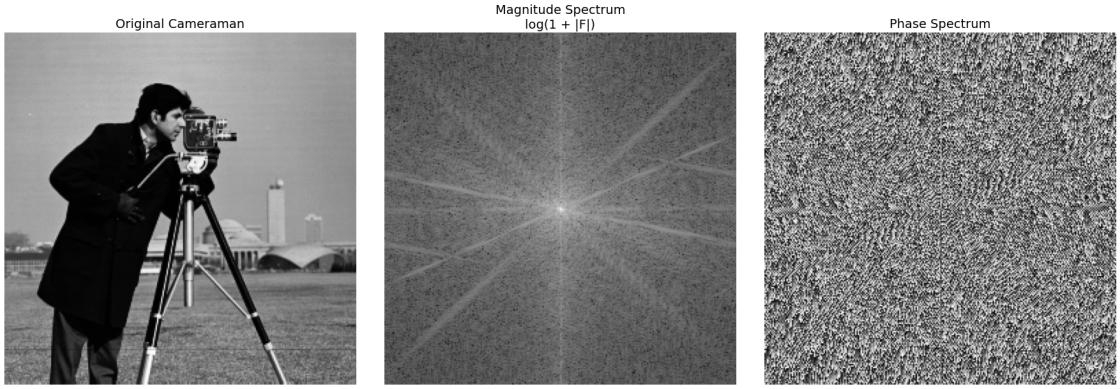
keeping only smooth, gradual transitions. Based on this understanding, "original minus low-pass" is related to a high-pass filter because the original image contains both high and low frequencies. When we subtract the low-pass filtered result, we are effectively removing the low-frequency components from the original and therefore, only the high-frequency information remains, which is exactly what a high-pass filter produces.

3. What kinds of structures appear more strongly in Gx vs Gy? In Gx, vertical structures such as the cameraman's body, tripod legs, and vertical building edges would appear more strongly in Gx. Meanwhile, in Gy horizontal structures like the ground horizon, cameraman's shoulders would appear more strongly in Gy.

2.0.7 Part D: Frequency-domain filtering

Goal: See how filters act in the Fourier domain and design a notch (band-reject) filter to remove periodic noise.

[82]:
```
#Step 1: magnitude and phase

#computing 2D fourier
F2D = np.fft.fft2(cameraman)
F2D_shift = np.fft.fftshift(F2D)

magnitude = np.abs(F2D_shift)
phase = np.angle(F2D_shift)

#Displaying log(1+abs(F))

magnitude_log = np.log(1+magnitude)

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

axes[0].imshow(cameraman, cmap='gray')
axes[0].set_title('Original Cameraman', fontsize=14)
axes[0].axis('off')

axes[1].imshow(magnitude_log, cmap='gray')
axes[1].set_title('Magnitude Spectrum\nlog(1 + |F|)', fontsize=14)
axes[1].axis('off')

axes[2].imshow(phase, cmap='gray')
axes[2].set_title('Phase Spectrum', fontsize=14)
axes[2].axis('off')

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



Low vs High frequencies Low frequencies appeared at the center of the magnitude spectrum. The bright central region represents smooth, gradual intensity changes and the overall structure of the image. In contrast, high frequencies appeared farther from the center toward the edges of the spectrum. These dimmer outer regions represent fine details, sharp edges, and textures in the image.

[83]:
```python
#Step 2: Design LPF/HPF in frequency domain

#custom filter function
def filters(shape, D0, filter_type, n=2):
    rows, cols = shape
    c_row, c_cols = rows // 2, cols // 2 #center of rows and cols
    y, x = np.ogrid[:rows, :cols]
    D = np.sqrt((x - c_cols)**2 + (y - c_row)**2)

    match filter_type:
        case 'ideal':
            LPF = (D <= D0).astype(float)
        case 'gaussian':
            LPF = np.exp(-(D**2) / (2 * (D0**2)))
        case 'butterworth':
            LPF = 1 / (1 + (D / D0)**(2 * n))

    HPF = 1 - LPF
    return LPF, HPF

#custom function to apply filters
def apply_filter(img, filter_mask):
    Fourier = np.fft.fft2(img)
    F_shift = np.fft.fftshift(Fourier)
    F_filtered = F_shift * filter_mask
    F_inv_shift = np.fft.ifftshift(F_filtered)
    img_filtered = np.real(np.fft.ifft2(F_inv_shift))
```

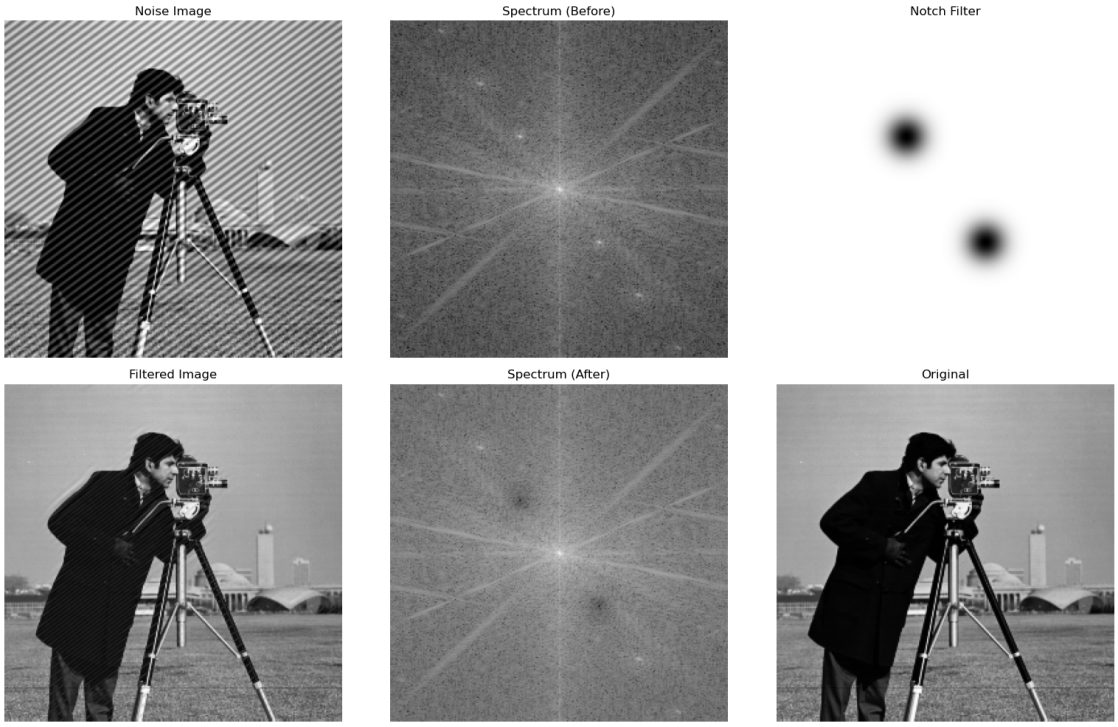
```
    return np.clip(img_filtered, 0, 255).astype(np.uint8)

cutoffs = [20, 60]
filter_types = ['ideal', 'gaussian', 'butterworth']
gaussian_freq_results = {}

fig, axes = plt.subplots(len(cutoffs) * 2, len(filter_types) + 1, figsize=(16,␣
 ↪12))

for i, D0 in enumerate(cutoffs):
    for j, ftype in enumerate(filter_types):
        LPF, HPF = filters(cameraman.shape, D0, ftype)
        img_lpf = apply_filter(cameraman, LPF)
        img_hpf = apply_filter(cameraman, HPF)

        if ftype == 'gaussian':
            gaussian_freq_results[D0] = img_lpf

        row_lpf = i * 2
        row_hpf = i * 2 + 1

        if j == 0:
            axes[row_lpf, 0].imshow(cameraman, cmap='gray')
            axes[row_lpf, 0].set_title('Original')
            axes[row_lpf, 0].axis('off')
            axes[row_hpf, 0].imshow(cameraman, cmap='gray')
            axes[row_hpf, 0].set_title('Original')
            axes[row_hpf, 0].axis('off')

        axes[row_lpf, j+1].imshow(img_lpf, cmap='gray')
        axes[row_lpf, j+1].set_title(f'{ftype.capitalize()} LPF\nD0={D0}')
        axes[row_lpf, j+1].axis('off')

        axes[row_hpf, j+1].imshow(img_hpf, cmap='gray')
        axes[row_hpf, j+1].set_title(f'{ftype.capitalize()} HPF\nD0={D0}')
        axes[row_hpf, j+1].axis('off')

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

[84]:
```
#Gaussian in the spatial domain
sigma_20 = 20 / 3 # conversion from D0 to sigma
sigma_60 = 60 / 3

gaussian_spatial_20 = cv2.GaussianBlur(cameraman, (0, 0), sigmaX=sigma_20)
gaussian_spatial_60 = cv2.GaussianBlur(cameraman, (0, 0), sigmaX=sigma_60)

# reusing previous results
gaussian_freq_20 = gaussian_freq_results[20]
gaussian_freq_60 = gaussian_freq_results[60]

# Display comparison
fig, axes = plt.subplots(2, 2, figsize=(12, 12))

axes[0,0].imshow(gaussian_freq_20, cmap='gray')
axes[0,0].set_title('Gaussian Freq Domain\nD0=20')
axes[0,0].axis('off')
```

```
axes[0,1].imshow(gaussian_spatial_20, cmap='gray')
axes[0,1].set_title('Gaussian Spatial Domain\n sigma = 7')
axes[0,1].axis('off')

axes[1,0].imshow(gaussian_freq_60, cmap='gray')
axes[1,0].set_title('Gaussian Freq Domain\nD0=60')
axes[1,0].axis('off')

axes[1,1].imshow(gaussian_spatial_60, cmap='gray')
axes[1,1].set_title('Gaussian Spatial Domain\n sigma = 20')
axes[1,1].axis('off')

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

Discussion Similiraties and differences From the three types of filters (ideal, Gaussian, Butterworth) it can be seen that they produce similar blurring effects in low-pass filter mode and they also extract edges in high-pass filter mode while large cutoff radii D0 = 60 perserved more details than the small value D0 = 20. The differences lay on their transition characteristics, ideal filters create sharp cutoffs causing ringing near edges. Gaussian filters offers smoothest transitions with not artifacts. Butterworth filter provide a balance output compare to the others two. Gaussian evaluation in the spatial domain vs frequency, it can be seen that the approaches both output different results. The spatial creates a heavier blur effect while the frequency domain maintains better edge definition and details.

[85]:
```
#Step 3: Notch Filter
#sinusodial stripe pattern
rows, cols = cameraman.shape
u0, v0 = 30, 40
X, Y = np.meshgrid(np.arange(cols), np.arange(rows))
stripe = 50 * np.sin(2 * np.pi * (u0 * X / cols + v0 * Y / rows))
noise_img = np.clip(cameraman + stripe, 0, 255).astype(np.uint8)

#FFT
F_noise = np.fft.fftshift(np.fft.fft2(noise_img))

#custom function notch filter
def notch(shape, u0, v0, radius):
    c_row, c_col = shape[0] // 2, shape[1] // 2
    y, x = np.ogrid[:shape[0], :shape[1]]
    D1 = np.sqrt((x - (c_col + u0))**2 + (y - (c_row + v0))**2)
    D2 = np.sqrt((x - (c_col - u0))**2 + (y - (c_row - v0))**2)
    return (1 - np.exp(-D1**2 / (2 * radius**2))) * (1 - np.exp(-D2**2 / (2 *␣
 ↪radius**2)))

# Apply notch filter
notch = notch(cameraman.shape, u0, v0, radius=10)
F_filtered = F_noise * notch
filtered_img = np.real(np.fft.ifft2(np.fft.ifftshift(F_filtered))).clip(0, 255).
 ↪astype(np.uint8)

# Display
fig, axes = plt.subplots(2, 3, figsize=(16, 10))
images = [noise_img, np.log(1 + np.abs(F_noise)), notch,
          filtered_img, np.log(1 + np.abs(F_filtered)), cameraman]
titles = ['Noise Image', 'Spectrum (Before)', 'Notch Filter',
          'Filtered Image', 'Spectrum (After)', 'Original']

for ax, img, title in zip(axes.flat, images, titles):
    ax.imshow(img, cmap='gray')
    ax.set_title(title, fontsize=12)
    ax.axis('off')

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



Reflection

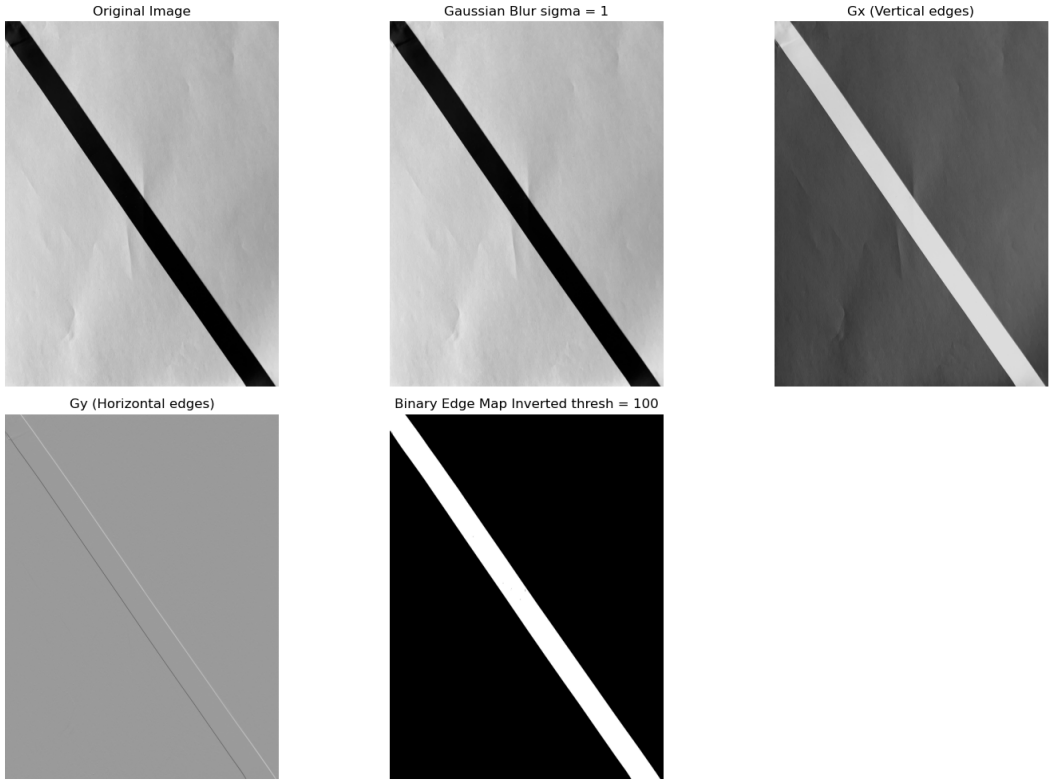
1. What advantages did the frequency-domain approach give you for removing the synthetic interference? The advantages is the possibility of removing specific interference by isolating specific frequencies that peaks due to interference on them so they get removed. It can help to prevent the distortion of the desired signal frequency and provide more stable results.
2. How would you pick cutoff radii? I would do a trial and error method to choose the right cutoff radii until I can visually see my desired result. However it also depends if I am applying filters let say spatial frequency it would basically be influenced by the low or high frequencies if I want to remove them or keep them.

2.0.8 Part E: Edge detection and a robotics mini task

Goal: Use edges to infer simple scene structures for a robot.

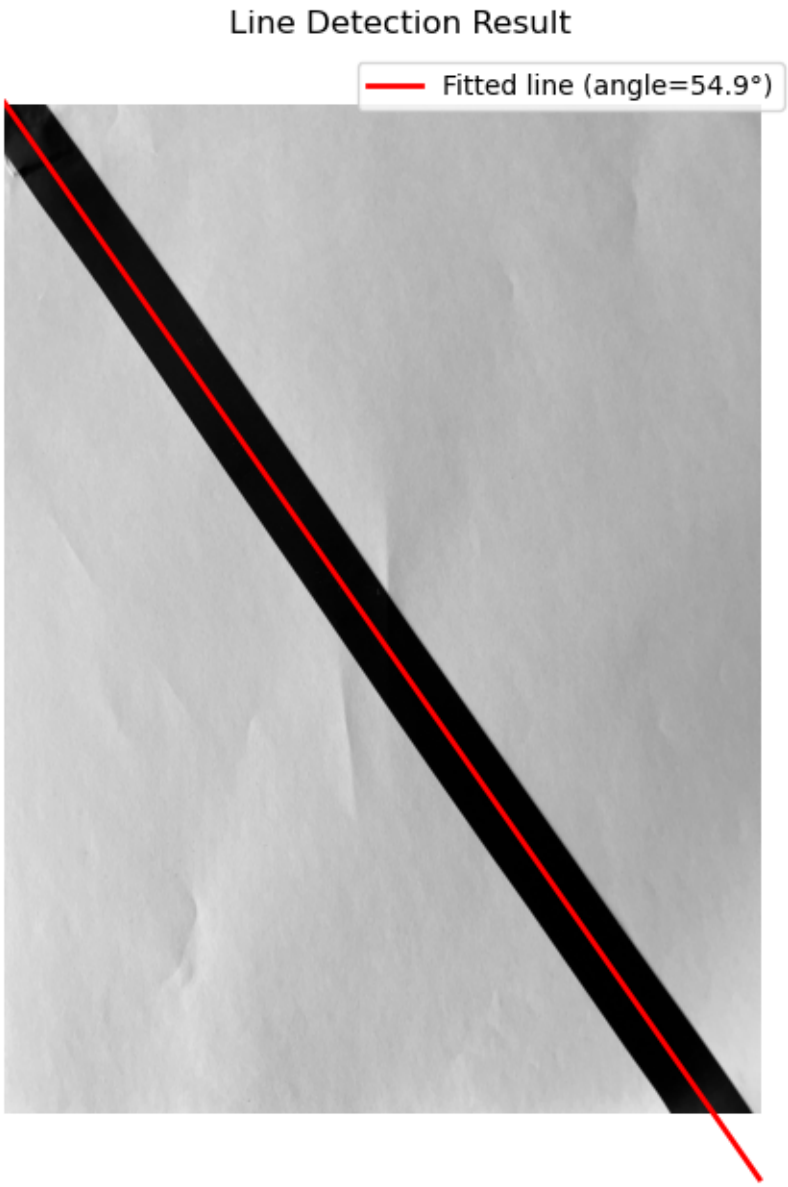
[86]:
```
#Step 1: Apply a mild Gaussian blur sigma = 1, I have to upload the line image␣
↪at the beginning of part A because it was giving me errors
#here.

#Gaussian Blur
line_gaussian = apply_gaussian(line, 1)

#Step 2:
#SobelXY
```

```
Gx, Gy, G, theta = sobelXY(line_gaussian)

#threshold G for binary map
thresh = 100

binary_map = (G > thresh).astype(np.uint8) * 255
binary_map_inv = cv2.bitwise_not(binary_map)

fig, axes = plt.subplots(2, 3, figsize=(15, 10))

axes[0,0].imshow(line, cmap='gray')
axes[0,0].set_title('Original Image')
axes[0,0].axis('off')

axes[0,1].imshow(line_gaussian, cmap='gray')
axes[0,1].set_title(f'Gaussian Blur sigma = 1')
axes[0,1].axis('off')

axes[0,2].imshow(Gx, cmap='gray')
axes[0,2].set_title('Gx (Vertical edges)')
axes[0,2].axis('off')

axes[1,0].imshow(Gy, cmap='gray')
axes[1,0].set_title('Gy (Horizontal edges)')
axes[1,0].axis('off')

axes[1,1].imshow(binary_map_inv, cmap='gray')
axes[1,1].set_title(f'Binary Edge Map Inverted thresh = {thresh}')
axes[1,1].axis('off')

axes[1,2].axis('off')

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



[87]:
```
#Step 3: Extracting edeges pixels of the line

#Edges pixels
edge_pixels = np.where(binary_map_inv == 255)
y_cord = edge_pixels[0]
x_cord = edge_pixels[1]

#Polyfit, y = mx + b is the formula
coeff = np.polyfit(x_cord, y_cord, 1)
m, b = coeff

#getting orientation
agnle_deg = np.degrees(np.arctan(m))

#fitting line
x_line = np.array([0, binary_map.shape[1]])
y_line = m * x_line + b

fig, ax = plt.subplots(figsize=(8, 8))
ax.imshow(line, cmap='gray')
ax.plot(x_line, y_line, 'r-', linewidth=2, label=f'Fitted line␣
 ↪(angle={angle_deg:.1f}°)')
```

```
ax.set_title('Line Detection Result')
ax.legend()
ax.axis('off')
plt.show()

print(f"Robot should steer by {angle_deg:.2f}°")
```




```
Robot should steer by 54.94°
```

Reflectionabs

1. Which preprocessing made the line more stable? I would say that gaussian blur helped to enhance the image but sobel made the line more stable to determine its edges pixels to fit the line properly. Probably if we use Canny edges detection it would be better than sobel but because sobel looks at the intensity of the pixels in the image it is more than enough to be used with a threshold of 100 to get accurate results.

2. In real robotic systems, what challenges could cause this method to fail? A camera not properly calibrated could lead to issues, lighting conditions could make the edges detection hard to perform and thats why filters must be used to enhance the quality of the image to inspect. Latency of performing image processing onboard robots could also reduce the performance and lead to missing edges of black lines as an example.

2.0.9 Part F: Reflection

1. Spatial vs frequency domain: When would you prefer one over the other in practice? Spatial will be prefer to use for simple and localized operations, it is computational efficient and can be used for real-time applications such as robotics. While, frequency domain is useful for precise frequency control and if it is needed to remove specific frequency components. Additionally frequency domain is computationally faster because FFT algorithm. Overall, the choice depend on the specific tasks to accomplish and the level of complexity.

2. What's one thing that surprised you in this assignment? How much it took to finish it (joke!). It surprise me how I could use images in grayscale for example to filter a white background and just focus on the black line to fine the path a robot should take to navigate acurately. I know it takes more than an assignment to understand the full potential of computer vision in robotics but learning the principles is always good.