Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT404

2024–11–13

Assignment 2: Image Processing & Analysis

1 A Morphological Image Processing Pipeline for Medical Images

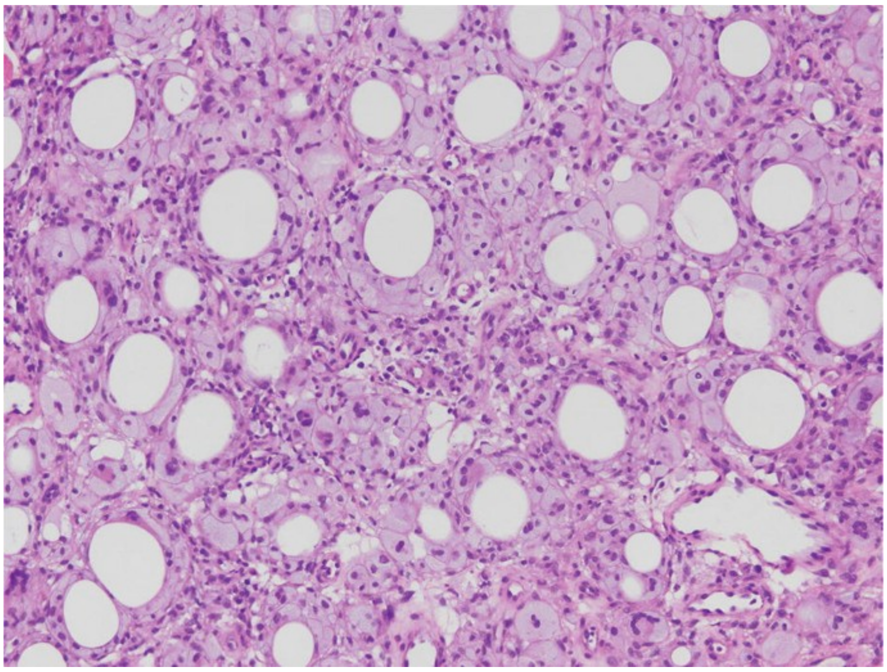


Figure 1: Original Skin Biopsy Image

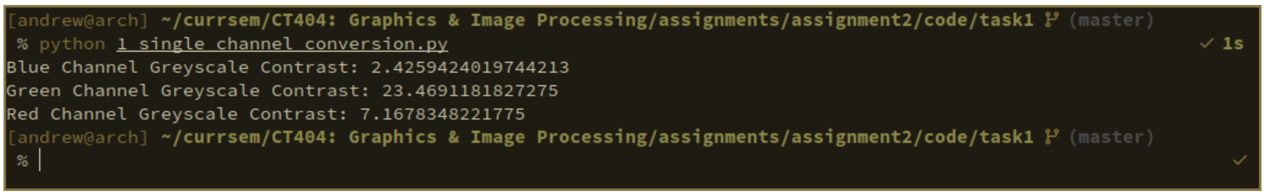
1.1 Conversion to A Single-Channel Image

```python
# Task 1: A Morphological image processing pipeline for medical images
# Task 1.1: Conversion to a single channel image
import cv2

# read in original image (in BGR format)
image = cv2.imread("../../Task1.jpg")

# convert to greyscale
greyscale = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imwrite("./output/greyscale.jpg", greyscale)

# convert to blue channel only
b_channel = image.copy()
b_channel[:, :, 1] = 0
b_channel[:, :, 2] = 0
cv2.imwrite("./output/b_channel.jpg", b_channel)

# convert blue channel to greyscale
b_channel_greyscale = cv2.cvtColor(b_channel, cv2.COLOR_BGR2GRAY)
b_channel_greyscale_contrast = b_channel_greyscale.std()
cv2.imwrite("./output/b_channel_greyscale.jpg", b_channel_greyscale)

# convert to green channel only
g_channel = image.copy()
g_channel[:, :, 0] = 0
```

```python
g_channel[:, :, 2] = 0
cv2.imwrite("./output/g_channel.jpg", g_channel)

# convert green channel to greyscale
g_channel_greyscale = cv2.cvtColor(g_channel, cv2.COLOR_BGR2GRAY)
g_channel_greyscale_contrast = g_channel_greyscale.std()
cv2.imwrite("./output/g_channel_greyscale.jpg", g_channel_greyscale)

# convert to red channel only
r_channel = image.copy()
r_channel[:, :, 0] = 0
r_channel[:, :, 1] = 0
cv2.imwrite("./output/r_channel.jpg", r_channel)

# convert red channel to greyscale
r_channel_greyscale = cv2.cvtColor(r_channel, cv2.COLOR_BGR2GRAY)
r_channel_greyscale_contrast = r_channel_greyscale.std()
cv2.imwrite("./output/r_channel_greyscale.jpg", g_channel_greyscale)

# assess objectively which allows most contrast
print("Blue Channel Greyscale Contrast: "  + str(b_channel_greyscale_contrast))
print("Green Channel Greyscale Contrast: " + str(g_channel_greyscale_contrast))
print("Red Channel Greyscale Contrast: "   + str(r_channel_greyscale_contrast))
```

Listing 1: `1_single_channel_conversion.py`

Since the image has predominant hues of pink-purple, we would expect the green-channel-only image to be the one that yields the highest contrast, as pink & purple colours are made up primarily by the blue & red channels: the dominance of these channels results in little variance in intensity within these channels, and therefore green will have the highest intensity variance. This is proven true by the text output of the above code, where the standard deviation of the greyscale image based off the green channel alone is by far the highest:



```
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 (master)
 % python 1_single_channel_conversion.py                                                          ✓ 1s
Blue Channel Greyscale Contrast: 2.4259424019744213
Green Channel Greyscale Contrast: 23.4691181827275
Red Channel Greyscale Contrast: 7.1678348221775
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 (master)
 % |                                                                                                 ✓
```

Figure 2: Output of `1_single_channel_conversion.py`

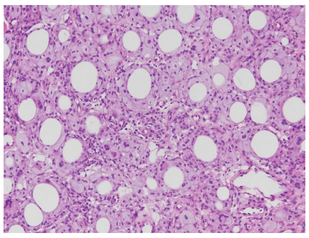


Figure 3: Original image

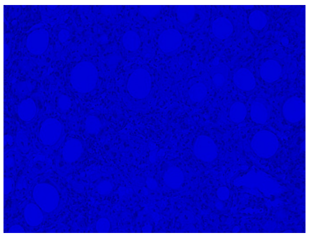


Figure 5: B-Channel

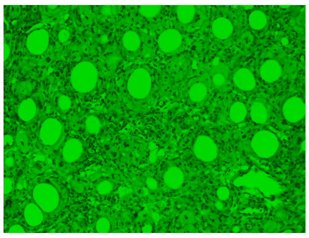


Figure 7: G-Channel

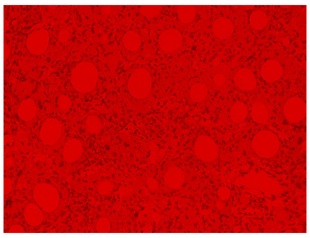


Figure 9: R-Channel

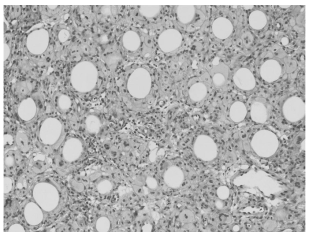


Figure 4: Greyscale original

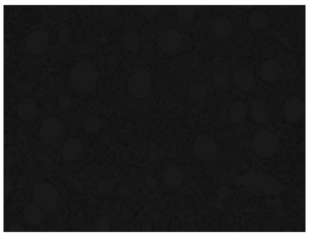


Figure 6: B-Greyscale

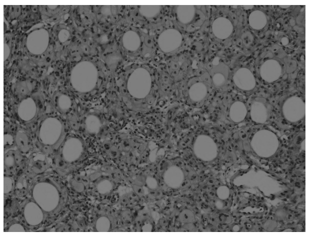


Figure 8: G-Greyscale

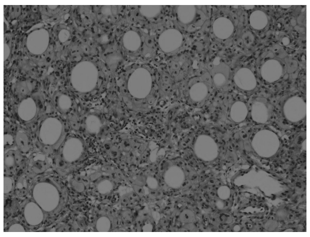


Figure 10: R-Greyscale

My selected single-channel image is the greyscale version of the green-channel-only image, as it yields the greatest contrast:

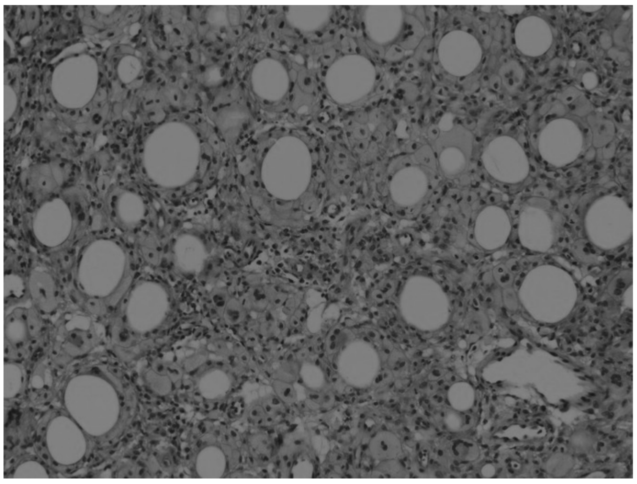
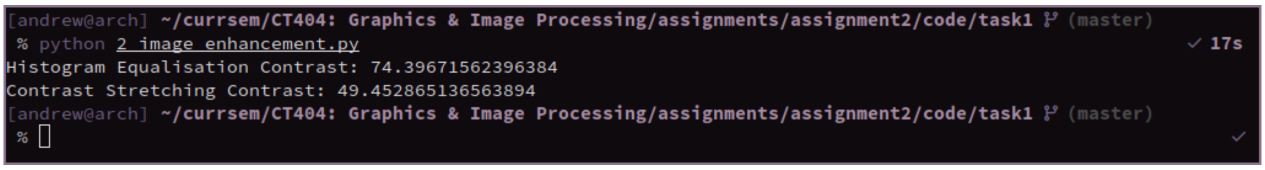


Figure 11: Selected single-channel image: greyscale green-channel-only

1.2 Image Enhancement

```python
# Task 1.2: Image Enhancement
import cv2

# read in chosen single-channel greyscale image
image = cv2.imread("./output/g_channel_greyscale.jpg", cv2.IMREAD_GRAYSCALE)

# apply histogram equalisation
equalised_image = cv2.equalizeHist(image)
equalised_image_contrast = equalised_image.std()
cv2.imwrite("./output/histogram_equalised.jpg", equalised_image)

# apply contrast stretching
stretched_image = cv2.normalize(image, None, 0, 255, cv2.NORM_MINMAX)
stretched_image_contrast = stretched_image.std()
cv2.imwrite("./output/contrast_stretched.jpg", stretched_image)

print("Histogram Equalisation Contrast: " + str(equalised_image_contrast))
print("Contrast Stretching Contrast: "    + str(stretched_image_contrast))
```

Listing 2: `2_image_enhancement.py`



```
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 (master)
 % python 2_image_enhancement.py                                                          ✓ 17s
Histogram Equalisation Contrast: 74.39671562396384
Contrast Stretching Contrast: 49.452865136563894
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 (master)
 %                                                                                          ✓
```

Figure 12: Output of `2_image_enhancement.py`

I chose to use the histogram equalisation technique as it gave the best contrast, as seen from the calculated standard deviation in contrast above and in the output images below.

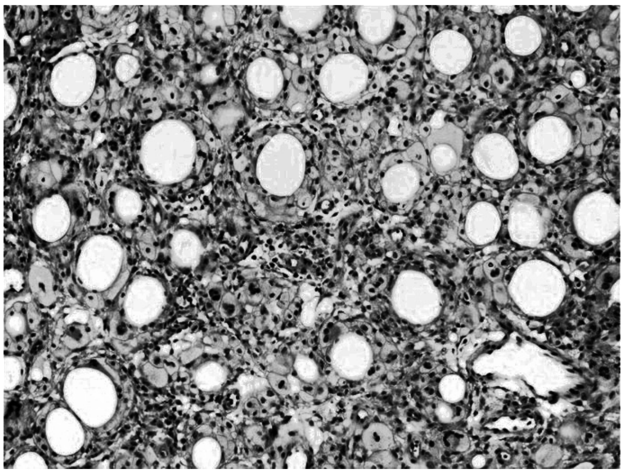


Figure 13: Histogram-equalised image

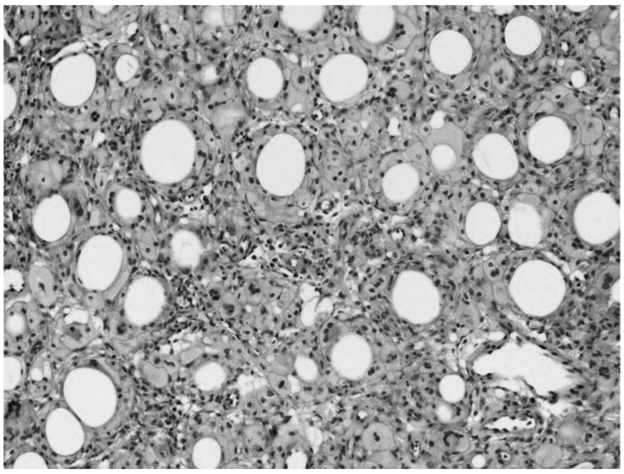
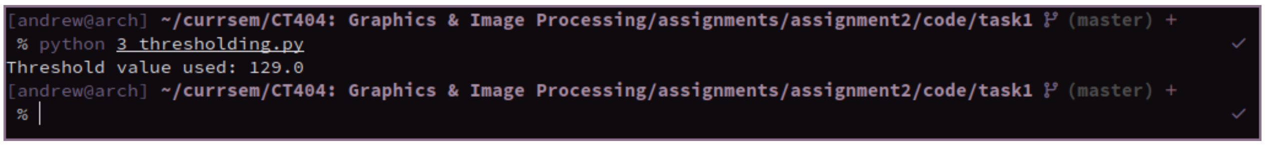


Figure 14: Contrast-stretched image

1.3 Thresholding

```python
# Task 1.3: Thresholding
import cv2

# read in chosen enhanced image
image = cv2.imread("./output/histogram_equalised.jpg", cv2.IMREAD_GRAYSCALE)

# perform otsu thresholding to find the optimal threshold
threshold_value, otsu_thresholded = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
cv2.imwrite("./output/otsu.jpg", otsu_thresholded)

print("Threshold value used: " + str(threshold_value))
```

Listing 3: `3_thresholding.py`



```
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 ᚠ (master) +
 % python 3_thresholding.py                                                                              ✓
Threshold value used: 129.0
[andrew@arch] ~/currsem/CT404: Graphics & Image Processing/assignments/assignment2/code/task1 ᚠ (master) +
 % |                                                                                                     ✓
```

Figure 15: Output of `3_thresholding.py`

I used Otsu's algorithm to find the optimal threshold value that best separated the foreground (objects of interest) from the background. As can be seen from the above output, the optimal value chosen was 129.

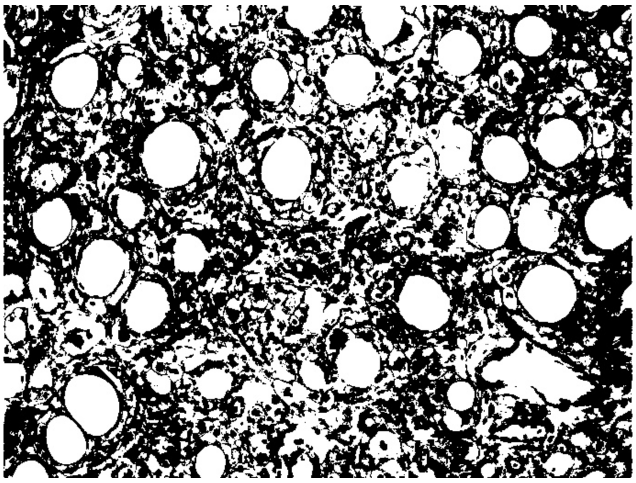


Figure 16: Image with Otsu thresholding

1.4 Noise Removal

```python
# Task 1.4: Noise Removal
import cv2

# read in thresholded image
image = cv2.imread("./output/otsu.jpg", cv2.IMREAD_GRAYSCALE)

# try several different sizes of structuring element (must be odd)
for kernel_size in range(1, 32, 2):
    # define a disk-shaped structuring element
    structuring_element = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (kernel_size, kernel_size))

    # apply morphological opening to remove noise
    opened_image = cv2.morphologyEx(image, cv2.MORPH_OPEN, structuring_element)
    cv2.imwrite(f"./output/kernel_size_{kernel_size}.jpg", opened_image)
```

Listing 4: `4_noise_removal.py`

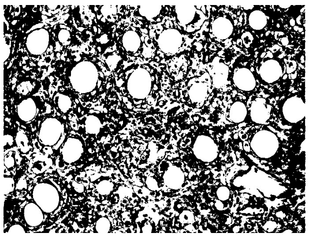


Figure 17: `kernel_size = 1`

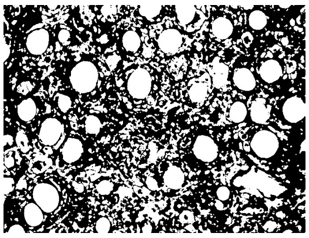


Figure 18: `kernel_size = 3`

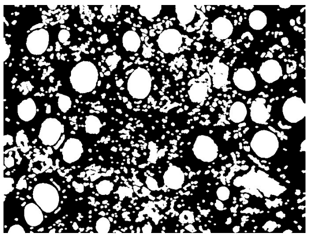


Figure 19: `kernel_size = 5`

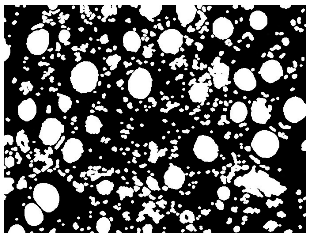


Figure 20: `kernel_size = 7`

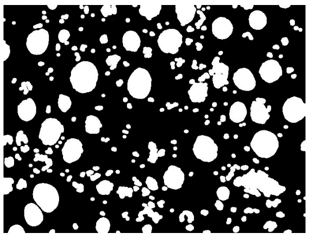


Figure 21: `kernel_size = 9`

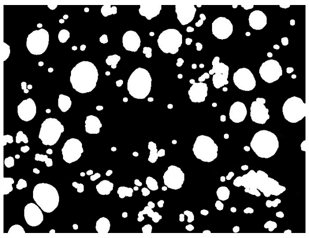


Figure 22: `kernel_size = 11`

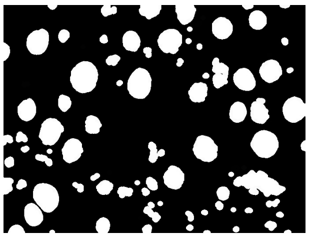


Figure 23: `kernel_size = 13`

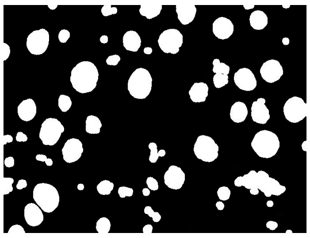


Figure 24: `kernel_size = 15`

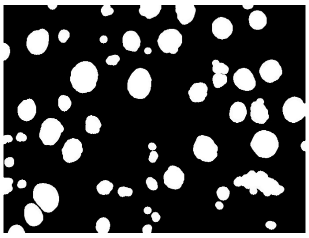


Figure 25: `kernel_size = 17`

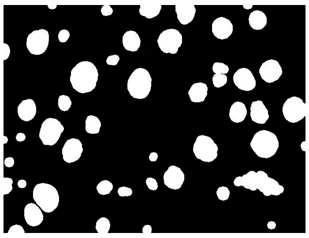


Figure 26: `kernel_size = 19`

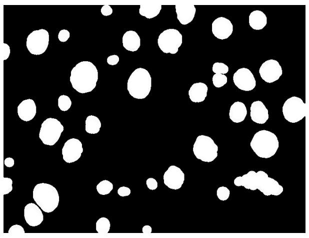


Figure 27: `kernel_size = 21`

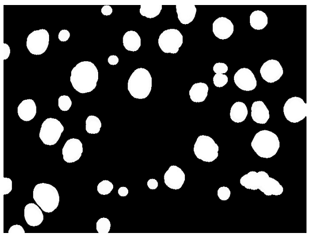


Figure 28: `kernel_size = 23`

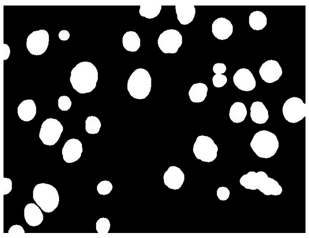


Figure 29: `kernel_size = 25`

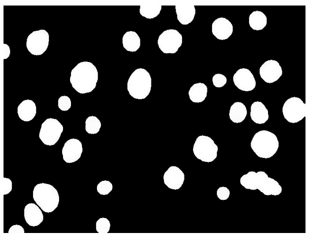


Figure 30: `kernel_size = 27`

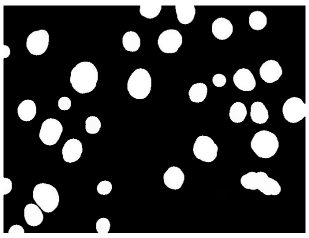


Figure 31: `kernel_size = 29`

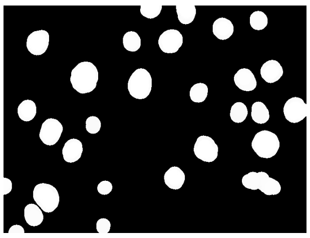


Figure 32: `kernel_size = 31`

I chose to open the image with a structuring element that had `kernel_size = 17` as it seemed to give the optimal balance between removing noise without significantly reducing the size of the remaining fat globules.

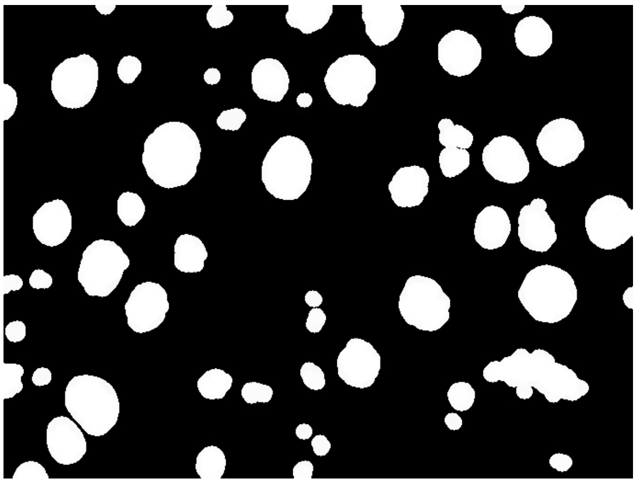


Figure 33: Chosen noise threshold: `kernel_size = 17`

1.5 Extraction of Binary Regions of Interest / Connected Components

```python
import cv2
import numpy as np

# Load and pre-process binary image
binary_image = cv2.imread("./output/kernel_size_17.jpg", cv2.IMREAD_GRAYSCALE)
binary_image = cv2.medianBlur(binary_image, 3)

# Step 1.5: Extraction of Binary Regions of Interest / connected components
num_labels, labels, stats, centroids = cv2.connectedComponentsWithStats(binary_image, connectivity=8)
```

Listing 5: Task 1.5 section of `5-7.py`

I'm not sure why, but no matter what level of noise removal I tried the connected components extraction with, the connected components always came out quite jagged. To correct for this, I did some additional noise reduction by using a blur on the image to remove some of the white noise that was appearing in. I also used a higher value of connectivity with `connectivity=8`, keeping components that were touching at all rather than components that just shared an edge.

1.6 Filtering of Fat Globules

```python
# Initialize an empty mask for filtered regions
filtered_mask = np.zeros(binary_image.shape, dtype=np.uint8)

total_globules = 0

# Task 1.6: Filtering of Fat Globules
for i in range(1, num_labels):
    area = stats[i, cv2.CC_STAT_AREA]

    # Calculate compactness
    perimeter = cv2.arcLength(cv2.findContours((labels == i).astype(np.uint8), cv2.RETR_EXTERNAL,
    ↪  cv2.CHAIN_APPROX_SIMPLE)[0][0], True)
    compactness = (perimeter ** 2) / area if area > 0 else 0

    if (300 < area) and (compactness < 27):
        total_globules += 1
        filtered_mask[labels == i] = 255

cv2.imwrite("./output/filtered_fat_globules.jpg", filtered_mask)
print("Total globules: " + str(total_globules))
```

Listing 6: Task 1.6 section of `5-7.py`

I filtered the fat globules based off size & compactness, using the compactness measure to remove globules that were not globule-shaped and the area measure to remove globules that were too small to be a globule. I used a maximum compactness of 27 and a minimum area of 300 to filter the globules, resulting in a total of 35 fat globules.

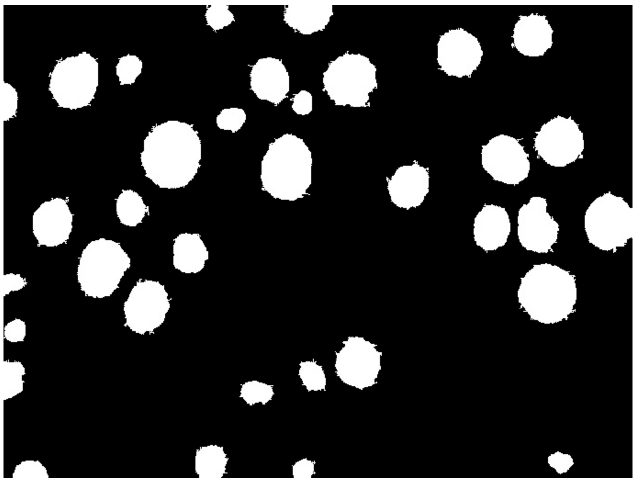


Figure 34: Result of fat globule filtering

1.7 Calculation of the Fat Area

```python
# Task 1.7: Calculation of the Fat Area
# Total area of the image in pixels (excluding the background)
total_image_area = binary_image.shape[0] * binary_image.shape[1]

# Total fat area (in pixels)
fat_area = np.sum(filtered_mask == 255)

# Calculate fat percentage
fat_percentage = (fat_area / total_image_area) * 100
print(f"Fat Area Percentage: {fat_percentage:.2f}%")
```

Listing 7: Task 1.7 section of `5-7.py`

The percentage of the image covered by fat globules was 15.33%.

2 Filtering of Images in Spatial & Frequency Domains

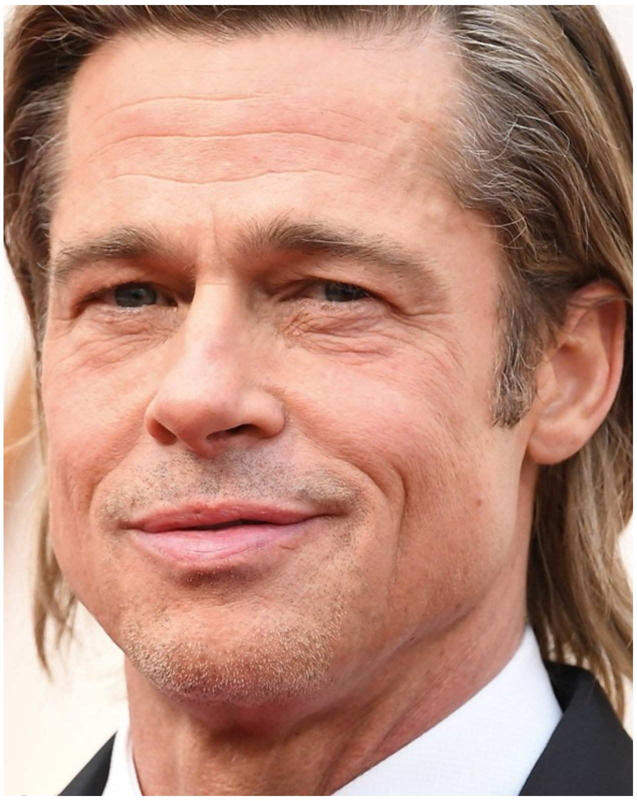


Figure 35: Original Facial Image

2.1 Spatial Domain

```
# Task 2.1: Spatial Domain
image = cv2.imread("../../Task2.jpg")

kernel_size = (15, 15)
variance = 2

smoothed_image = cv2.GaussianBlur(image, kernel_size, variance)

cv2.imwrite("./output/1_spatial_domain.jpg", smoothed_image)
```

Listing 8: Task 2.1 section of `task2.py`

After some experimentation, I chose parameter values of `kernel_size = (15,15)` and `variance = 2` as, in my opinion, these yielded the best balance between blurring imperfections like wrinkles without causing the entire image to become too blurry.

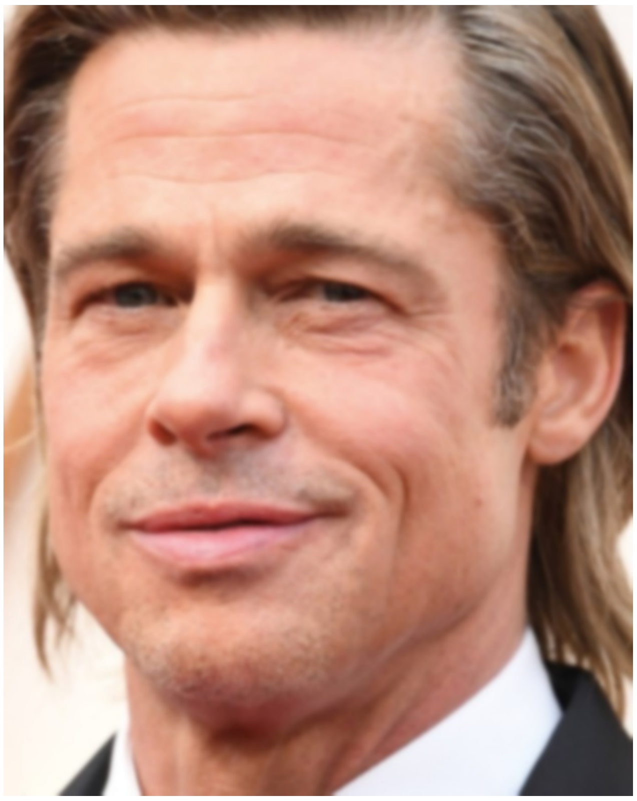


Figure 36: Output of `1_spatial_domain.jpg`

2.2 Frequency Domain Low-Pass Filter

```python
# Task 2.2: Frequency Domain Low-Pass Filter
gaussian_kernel = cv2.getGaussianKernel(kernel_size[0], variance)
gaussian_kernel_2d = gaussian_kernel @ gaussian_kernel.T
fft_gaussian = np.fft.fft2(gaussian_kernel_2d)

# shift zero frequency component to center
fft_gaussian_shifted = np.fft.fftshift(fft_gaussian)

# calculate the magnitude spectrum for visualization
magnitude_spectrum = np.log(np.abs(fft_gaussian_shifted) + 1)

# Plot the magnitude spectrum (Frequency Domain Representation)
plt.imshow(magnitude_spectrum, cmap='gray')
plt.axis('off')
plt.savefig("./output/2_frequency_domain_low-pass_filter.jpg", bbox_inches='tight', pad_inches=0)
```

Listing 9: Task 2.2 section of `task2.py`

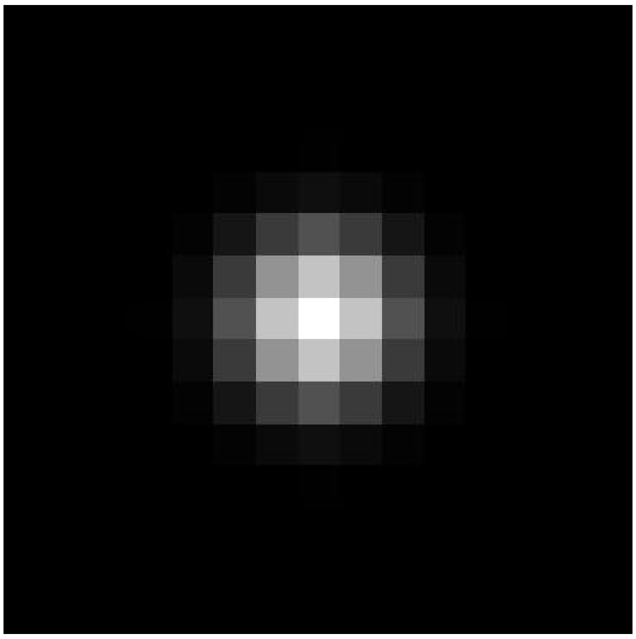


Figure 37: Zero-centered low-pass filter of Gaussian Kernel

2.3 Frequency Domain Filtering

```python
# Task 2.3: Frequency Domain Filtering for
channels = cv2.split(image)
filtered_channels = []

for channel in channels:
    fft_channel = np.fft.fft2(channel)

    # shift the zero frequency component to the center
    fft_channel_shifted = np.fft.fftshift(fft_channel)

    # create a Gaussian filter the same size as the channel
    gaussian_kernel = cv2.getGaussianKernel(kernel_size[0], variance)
    gaussian_kernel_2d = gaussian_kernel @ gaussian_kernel.T

    # pad the Gaussian filter to match the size of the image channel
    gaussian_kernel_padded = np.pad(gaussian_kernel_2d,
                                ((0, fft_channel_shifted.shape[0] - gaussian_kernel_2d.shape[0]),
                                 (0, fft_channel_shifted.shape[1] - gaussian_kernel_2d.shape[1])),
                                mode='constant', constant_values=0)

    # shift the padded filter in the frequency domain
    fft_gaussian_padded_shifted = np.fft.fftshift(np.fft.fft2(gaussian_kernel_padded))

    # apply the low-pass filter to the channel
    low_pass_filtered = fft_channel_shifted * fft_gaussian_padded_shifted

    # perform the inverse FFT to get the filtered channel in the spatial domain
    ifft_filtered = np.fft.ifft2(np.fft.ifftshift(low_pass_filtered))

    # take the real part and normalize it
    filtered_channel = np.real(ifft_filtered)
    filtered_channel = np.clip(filtered_channel, 0, 255).astype(np.uint8)

    # append the filtered channel to the list
    filtered_channels.append(filtered_channel)

filtered_image_color = cv2.merge(filtered_channels)
```

```

cv2.imwrite("./output/3_filtered_color_image.jpg", filtered_image_color)
```

Listing 10: Task 2.3 section of `task2.py`

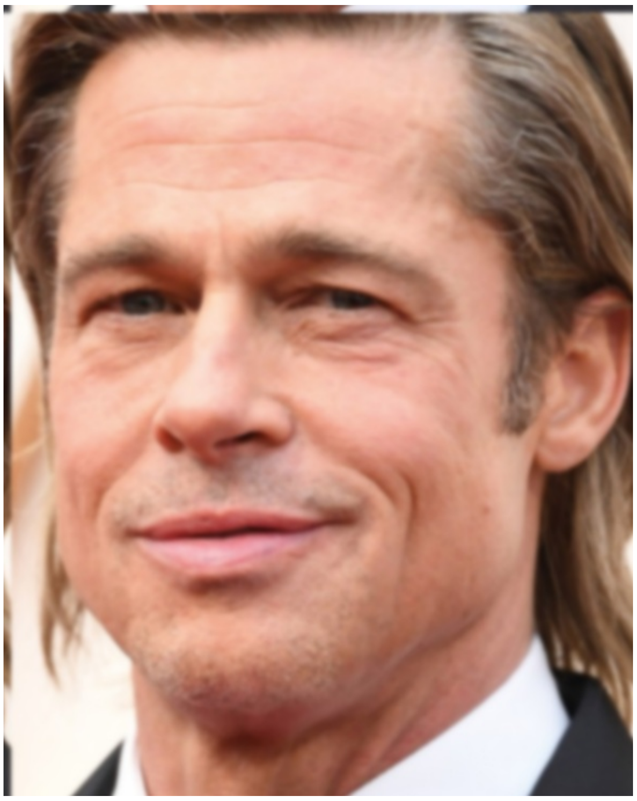


Figure 38: Frequency domain filtered image

The low pass filter type used was the same as in Section 2.2: a Gaussian filter with `(15,15)` and `2`.

2.4 Comparison

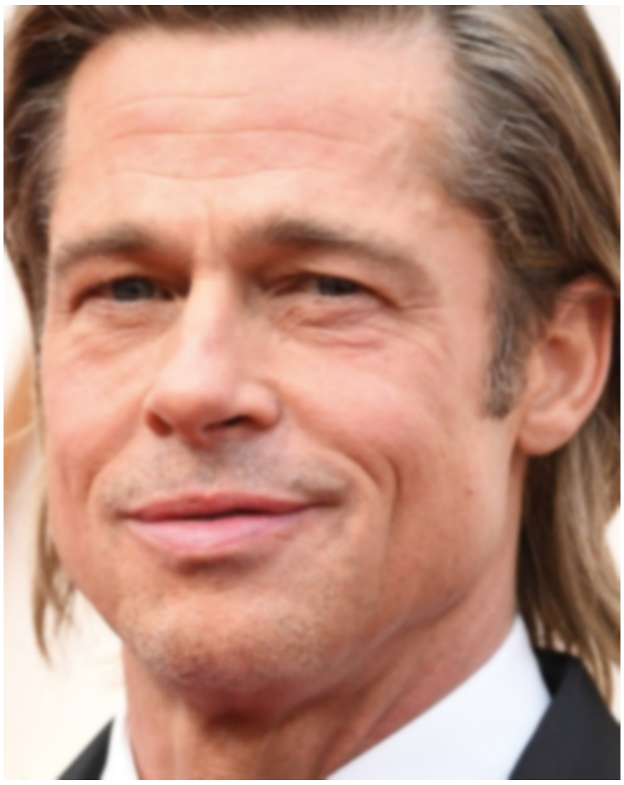


Figure 39: Spatial domain filtered image

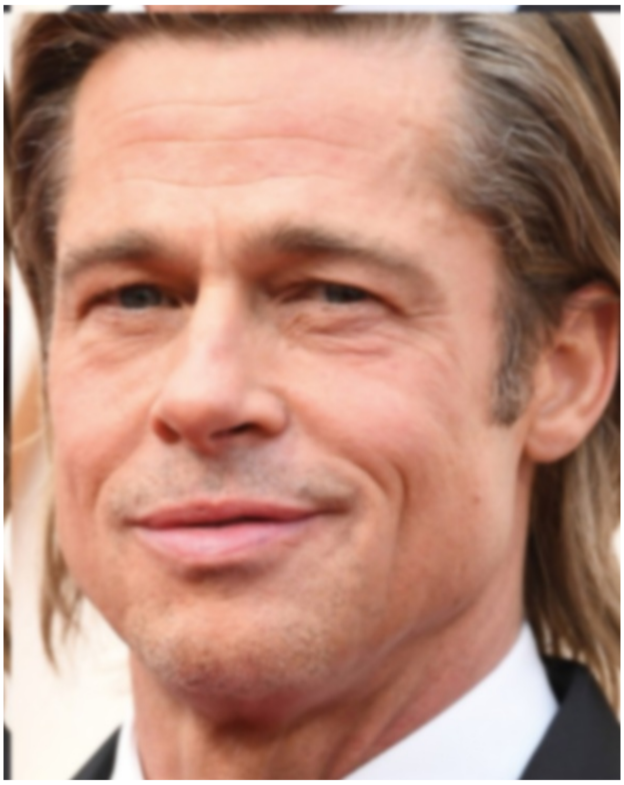


Figure 40: Frequency domain filtered image

The two images are very similar, having shared the same type of low pass filter. However, the frequency domain filtered image has retained more colour range, and has an overall less blurred appearance. The spatial domain filtering has applied a general blur across the entire image, making the filtering more obvious. On the other hand, the frequency domain filtering is more subtle and reduces the visibility of wrinkles while retaining some definition in the eyes, lips, and stubble. Overall, I would prefer the frequency domain filtering for this task; despite its increased complexity, it creates a more subtle and convincing effect that would be easily mistakable for an unfiltered image.

2.5 Unseen Image Testing

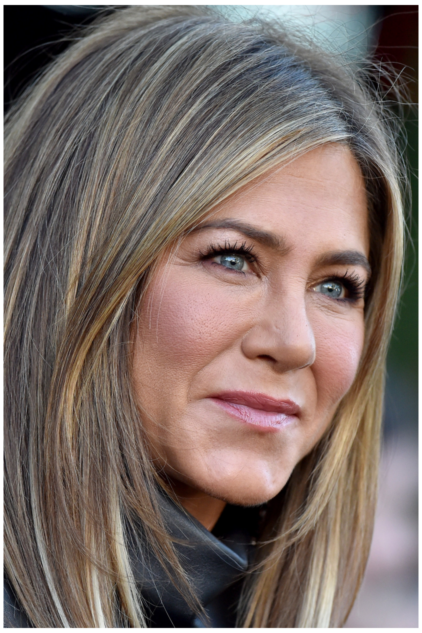


Figure 41: Original Image

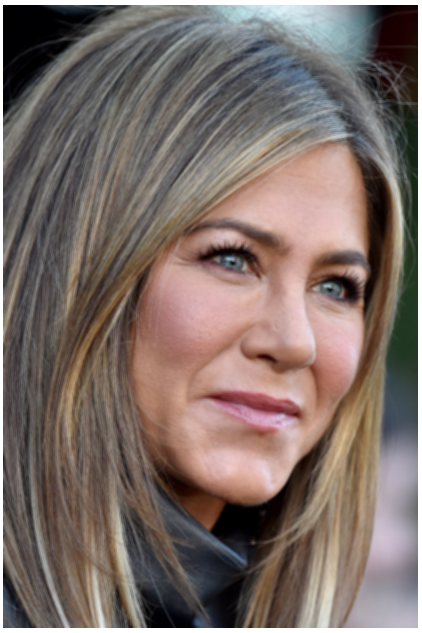


Figure 42: Spatial domain filtered image

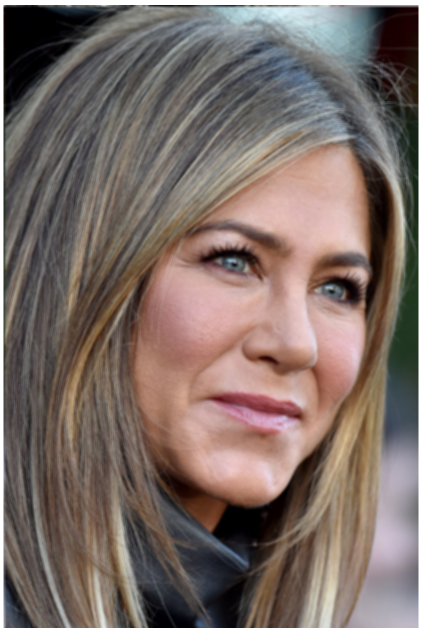


Figure 43: Frequency domain filtered

Again, the two filtered images are very similar. In my opinion, the frequency domain filtered image performed better here again, as it has a more dynamic colour range and more subtle blurring. The skin looks very artificially smoothed in the spatial domain filtered image, but more natural in the frequency domain filtered image. The hair looks more blurred and out-of-focus in the spatial domain filtered image, but looks sharper and more natural in the frequency domain filtered image.