FYS2010: Project

Candidate numbers: 001 and 121

September 29, 2022

Task 1: Classifying Digits

This task will revolve around using image restoring images of digits. Each digit has been corrupted in a different way. The 6th of each digit is shown in figure 1.

- 4: The digits contain wave interference
- 5: The digits have noise
- 6: The digits are in the green channel of the image
- 7: The digits are blurred
- 8: The digits are black-white inverted
- 9: The dynamic range is small
- 0: The digits have salt and pepper noise

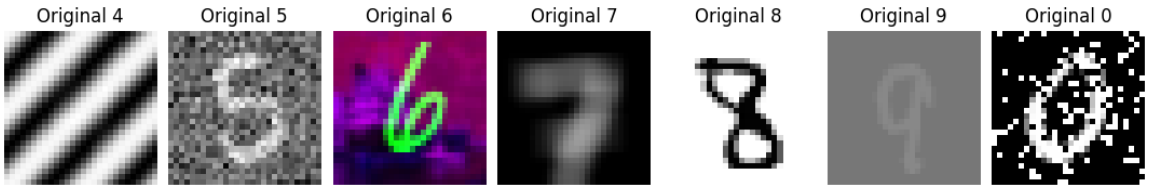

Figure 1: Digits with corruption.

The steps taken to restore each digit are presented here:

- 4: An effective way to remove wave interference is to filter out the unwanted frequencies in the frequency domain. Therefore we convert to frequency domain, filter out corrupting frequency, convert back to spatial domain. Finally, we apply Gaussian smoothing filter to remove some jaggedness.
- 5: The noise looks like random Gaussian noise. Therefore we use median filtering with a 3x3 kernel. After this, we still have some noise left, but it has lower spatial intensity than the digit itself, therefore we remove intensities lower than 120, which seems to have the desired effect of setting to 0 every pixel not part of the digit. Thereafter we apply median filter again to smooth out the digit.
- 6: Discard red and blue color channels, and we are left with a digit that needs no further alterations.
- 7: The image seems to have been blurred, so we sharpen edges with unsharp mask. Apply a customised sigmoid function to spatial intensities over 255/3, in order to increase brightness of the digit.
- 8: Invert image.

- 9: The dynamic range is small, so we expand spatial intensity range to 0-255 by subtracting minimum, and then dividing by maximum, before multiplying with 255.
- 0: We filter out the salt and pepper noise by applying a median filter with a 4x4 kernel.

Figure 2: Digits before and after restoration.

For implementation details, we refer to listing 1. We tested our restoration implementations on a reinforcement learning agent provided to us. We report the confusion matrix in figure 3, and note an overall accuracy of 93%. We note that the digits that were classified inaccurately were 0 and 7. After reviewing these digit images, we feel justified in partly placing the blame of these misclassifications with the originator of the handwriting of these digits.

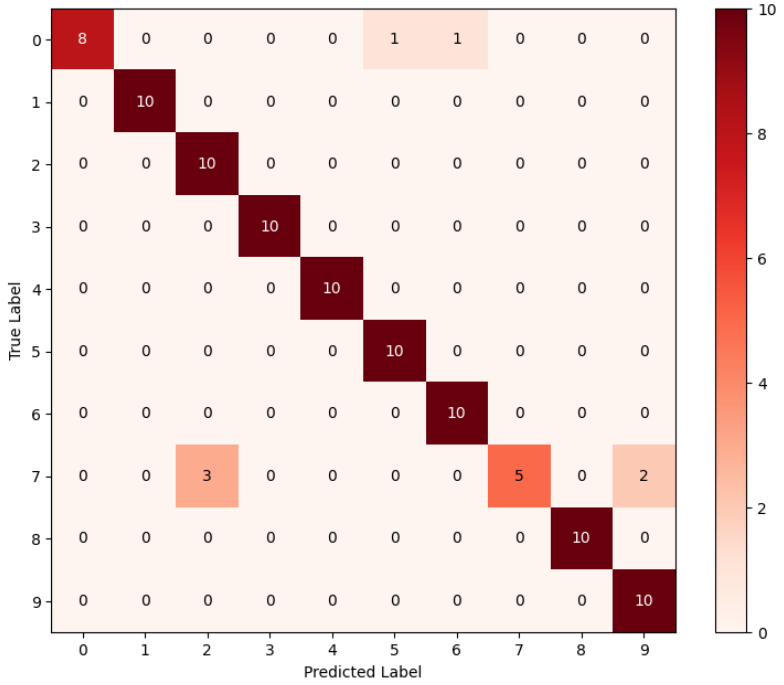

Figure 3: Confusion matrix for digit classification.

Listing 1: Task 1

```python
import numpy as np
from scipy.ndimage import median_filter
from skimage.filters import unsharp_mask
from scipy.ndimage import gaussian_filter

def clean_dig0(img):
    '''Function to recover digits of 0'''
    # Looks like salt and pepper noise
    # Use median filtering
    return median_filter(img, size=(4,4)).astype(np.uint8)

def clean_dig123(img):
    '''Function to recover digits of 1, 2, and 3'''
    # Use 3 least significant bits
    imgs = [np.bitwise_and(img[:,:,i],7) for i in range(3)]
    # Normalize images
    # Then run a median filter with 3x3 kernel
    return [median_filter((im/np.max(im))*255, size=(3,3)).astype(np.uint8) for im in imgs]

def clean_dig4(img):
    '''Function to recover digits of 4'''
    # Take fourier transform
    ft = np.fft.fftshift(np.fft.fft2(img))
    # Filter out frequencies at given positions
    ft[12,12] = ft[11,11]
    ft[16,16] = ft[17,17]
    # Take inverse ft
    new_img = np.abs(np.fft.ifft2(ft))
    # Remove pixels below 125 intensity
    new_img = np.where(new_img > 125, new_img, 0)
    # Normalize to 0-255 range
    new_img /= np.max(new_img)
    new_img *= 255
    # Run a gaussian filter
    new_img = gaussian_filter(new_img, 0.5)

    return new_img.astype(np.uint8)

def clean_dig5(img):
    '''Function to recover digits of 5'''
    # Run median filter
    i = median_filter(img, size=(3,3))
    # Filter out intensities under 120
    i = np.where(i > 120, i, 0)
    # Normalize to 0-255
    i = 255 / np.max(i) * i
    # Run median filter
    i = median_filter(i, size=(3,3))
    return i.astype(np.uint8)

def clean_dig6(img):
    '''Function to recover digits of 6'''
    # Digits are in green channel
    return img[:,:,1].astype(np.uint8)
```

```
55
56  def clean_dig7(img):
57      # Sharpen, and increase bright pixels
58      '''Function to recover digits of 7'''
59      # Sharpen images
60      sharp_img = unsharp_mask(img, radius=4, amount=4)
61      # Use customised sigmoid function at intensities over 255*0.33
62      sigmoid_img = 255 * np.where(sharp_img < 0.33, sharp_img, (1+np.exp(4-10*sharp_img))**-1)
63      return sigmoid_img.astype(np.uint8)
64
65  def clean_dig8(img):
66      '''Function to recover digits of 8'''
67      # Invert colors
68      i = -1*img + 255
69      return i.astype(np.uint8)
70
71  def clean_dig9(img):
72      '''Function to recover digits of 9'''
73      # Values are compressed. Decompress values. To 0-255
74      new_img = img - np.min(img)
75      new_img = new_img * (255 / np.max(new_img))
76      return new_img.astype(np.uint8)
```

Task 2: Aliasing

a)

Spatial aliasing effects, such as line jaggedness or stripes can be found in down-sampled images. This occurs when the image signal is down-sampled to a frequency less than twice as high as the highest frequency found in the original image's frequency domain (called the Nyquist rate) [1, p. 233-234]. It occurs because two frequencies become indistinguishable at certain sampling frequencies.

b)

In the provided original image at its original resolution, there are no indications of any distortion artifacts that would derive from aliasing. However, after the downsampling of the image to 50%, we can see some spatial aliasing in the pattern of the trousers of the woman pictured, as the stripes seem to switch from going vertically to going horizontally, as well as contain some jaggedness (see figure 4). This aliasing artifact derives from the signal interaction between the frequency domain (Fourier domain) becoming ambiguous.

Figure 4: The original image and the 50% down sampled image.

c)

We now want to downsample the image while either trying to filter out or completely avoid the aliasing artifacts. However, since within an downsampling imaging system there are always going to be some form of aliasing artifacts present [1, p. 233]. The approach we have chosen is to filter out the aliasing artifacts by applying the Gaussian smoothing filter on the downsampled image. This entails that the image "blur" that derives when using the Gaussian filter will then reduce or remove some of the details of the aliasing artifacts (spatial aliasing artifacts) ultimately filtering out the aliasing artifacts (see figure 5).

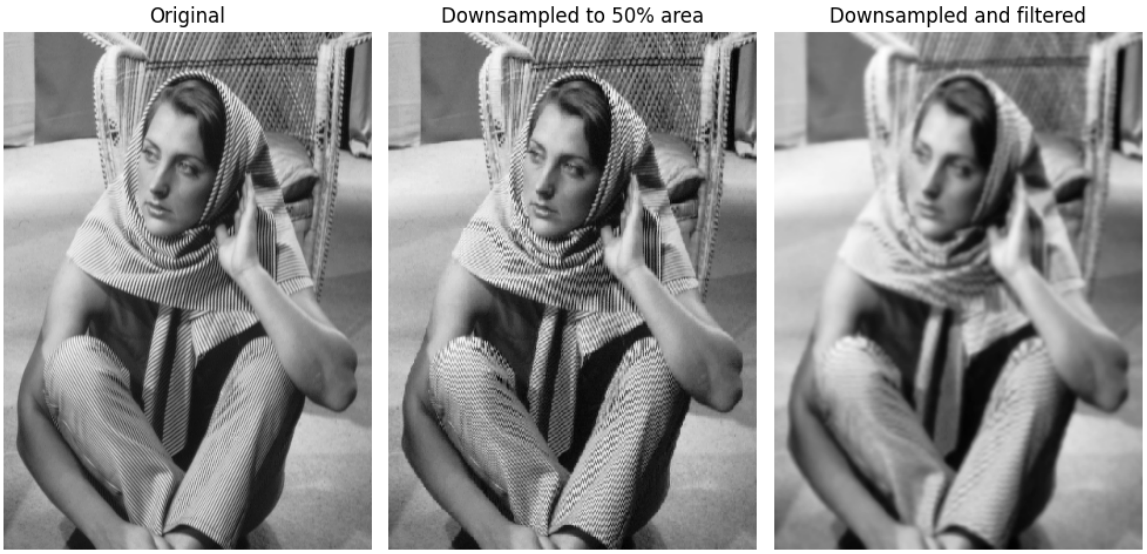

Figure 5: The original image, the 50% down sampled image and down sampled filtered image.

d)

In this task, we concluded that a sufficient algorithm to achieve sharpening the edges of the down-sampled image would be that of the provided functionality of the scikit-image package as of function known as *unsharp_mask*. This is on the basis that it directly operates and identified the sharp components based upon the discrepancy between the original image and the blurred image [2]. This choice was based on the premise of the blurred artifact that we introduced when applying the Gaussian filter is one of the criteria for applying this particular sharpening function. The result after using the different components/functions can be seen in the figure 6. We can see that it has yielded a significantly sharper image in contrast to the previous unprocessed image. This increase and recovery of the image's sharpness can be seen within the line edges within the image, and in the girl's facial features being more profound and distinct.

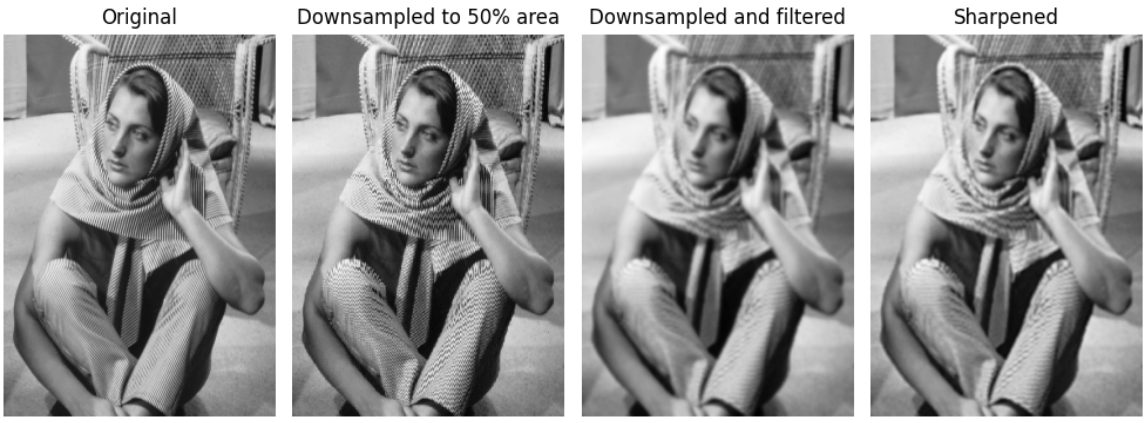

Figure 6: The original image, the 50% down sampled image, the down sampled filtered image and the sharpened image.

For the implementation of these tasks, we refer to listings 2, 3 and 4.

Listing 2: Task 2. b

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

scale = 0.5

# Read image
org_img = np.array(Image.open("VeiledGirl.tif"))

def downsample(img, scale):
    # Calculate shape of downscaled image
    org_shp = np.array(img.shape)
    new_shp = np.uint(np.sqrt(scale) * org_shp)

    # Calculate indexes
    x, y = np.ogrid[0:new_shp[0], 0:new_shp[1]]
    x, y = np.uint(x*(org_shp[1]/new_shp[1])), np.uint(y*(org_shp[0]/new_shp[0]))

    # Read original image at indexes
    return img[x,y]

ds_img = downsample(org_img, scale)

# Plot
fig, ax_arr = plt.subplots(1, 2, figsize=(10, 10))
ax1, ax2 = ax_arr.ravel()

ax1.imshow(org_img, cmap="Greys_r")
ax1.set_title("Original")

ax2.imshow(ds_img, cmap="Greys_r")
ax2.set_title(f"Downsampled to {scale:.0%} area")

for ax in ax_arr.ravel():
    ax.set_axis_off()
plt.tight_layout()
plt.show()
```

Listing 3: Task 2. c

```python
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter

def downsample(img, scale):
    org_shp = np.array(img.shape)
    new_shp = np.uint(np.sqrt(scale) * org_shp)
    x, y = np.ogrid[0:new_shp[0], 0:new_shp[1]]
    x, y = np.uint(x*(org_shp[1]/new_shp[1])), np.uint(y*(org_shp[0]/new_shp[0]))
    return img[x,y]

scale = 0.5
```

```

# Read image
org_img = np.array(Image.open("VeiledGirl.tif"))

# Downsample
ds_img = downsample(org_img, scale)

# Add gaussian filter
fil_img = gaussian_filter(ds_img, sigma=1.4)

# Plot
fig, ax_arr = plt.subplots(1, 3, figsize=(10, 10))
ax1, ax2, ax3 = ax_arr.ravel()

ax1.imshow(org_img, cmap="Greys_r")
ax1.set_title("Original")

ax2.imshow(ds_img, cmap="Greys_r")
ax2.set_title(f"Downsampled to {scale:.0%} area")

ax3.imshow(fil_img, cmap="Greys_r")
ax3.set_title(f"Downsampled and filtered")

for ax in ax_arr.ravel():
    ax.set_axis_off()
plt.tight_layout()
plt.show()
```

Listing 4: Task 2. d

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter
from skimage.filters import unsharp_mask


def downsample(img, scale):
    org_shp = np.array(img.shape)
    new_shp = np.uint(np.sqrt(scale) * org_shp)
    x, y = np.ogrid[0:new_shp[0], 0:new_shp[1]]
    x, y = np.uint(x*(org_shp[1]/new_shp[1])), np.uint(y*(org_shp[0]/new_shp[0]))
    return img[x,y]

scale = 0.5

# Read image
org_img = np.array(Image.open("VeiledGirl.tif"))

# Downsample
ds_img = downsample(org_img, scale)

# Add gaussian blur
fil_img = gaussian_filter(ds_img, sigma=1.4)

# Add sharpening filter
shp_img = unsharp_mask(fil_img, radius=1.4, amount=1.4)
```

```python

# Plot
fig, ax_arr = plt.subplots(1, 4, figsize=(10, 10))
ax1, ax2, ax3, ax4 = ax_arr.ravel()

ax1.imshow(org_img, cmap="Greys_r")
ax1.set_title("Original")

ax2.imshow(ds_img, cmap="Greys_r")
ax2.set_title(f"Downsampled to {scale:.0%} area")

ax3.imshow(fil_img, cmap="Greys_r")
ax3.set_title(f"Downsampled and filtered")

ax4.imshow(shp_img, cmap="Greys_r")
ax4.set_title(f"Sharpened")

for ax in ax_arr.ravel():
    ax.set_axis_off()
plt.tight_layout()
plt.show()
```

Task 3: Retina Feature Extraction

a)

SLIC (Simple Linear Iterative Clustering) can be understood as an extension of the k-means algorithm. These are algorithms used to cluster datapoints together. The difference is that calculating the distance between points in the SLIC algorithm also involves the difference in spatial intensity (color). The algorithm iteratively moves cluster centers, called centroids, based on the average of the points belonging to them. What follows is a rough explanation of the algorithm, based on [1, p. 774-5]. (For a more detailed explanation, we refer to them.)

1. **Initialize centroids.** Place the centroid evenly spaced throughout the grid (alternatively, place them randomly). Thereafter, move them to the lowest gradient position in their 3x3 neighborhood, and assign them the color of the pixel on which they are positioned.
2. **Assign each pixel to a centroid.** This is done by finding the centroid to which the pixel has the shortest distance, where spatial intensity is included as one or more dimensions.
3. **Update centroids.** Each centroid is updated and is given the average value of the pixels belonging to it.
4. **Check for convergence.** Calculate how much the centroids have moved – if the movement has reached a given lower threshold, stop the iteration. Else, go to the second step again.
5. **Post-processing:** Before stopping the algorithm, check that each superpixel is continuous. While this step is not trivial to implement, it will not be covered in this report. Each superpixel region can then be colored with the average superpixel color.

The parameters would typically be k (number of centroids) and ϵ, ie. threshold for convergence. Some implementations may support weighting spatial and colorspatial distances.

b)

We want to apply the SLIC algorithm to an RGB image of a retina to ultimately achieve a mask that isolates the blood vessels and lesions in the retina. When the SLIC algorithm is applied to the retina image, we can see how the algorithm is able to cluster large pixel regions with similar colors together (see figure 7), but does not necessarily cluster smaller regions together: For instance, the bright spot constituting the optical nerve is quite well clustered, but notably, hemorrhages present in the image do not constitute their own regions. After trying multiple combinations of parameters, we conclude that pre-processing of the image is needed to achieve a result in which different components are clustered together.

Figure 7: SLIC algorithm from SciKit-Image library applied with k=200, compactness=0.1.

c)

Before continuing, we choose a color channel on which to focus our efforts. After isolating the color channels (see figure 8), we conclude that the green channel contains the most information, in the form of higher contrast between the lesions and blood vessels and their surroundings and we therefore choose this as the basis for further experiments.

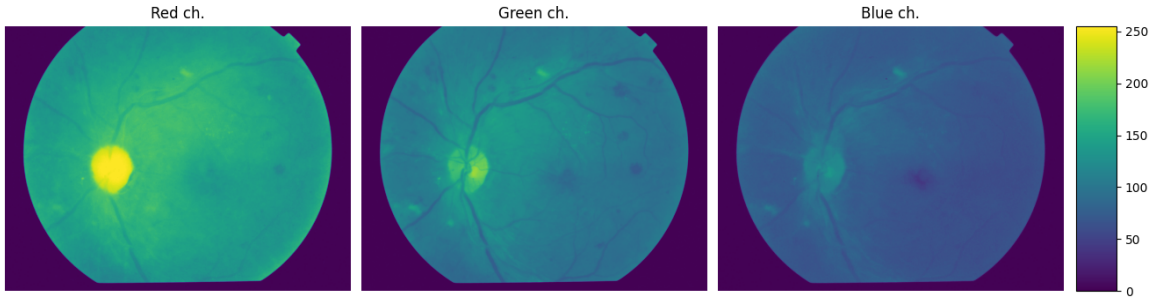

Figure 8: Color channels of the retina image isolated. We believe that the green channel seems to have the best potential for further experiments.

d)

We enhanced the green channel image by a combination of image processing techniques (fig. 9a). First we run a fine Gaussian filter, in order to smooth out high frequency noise. Then we apply an unsharp mask to sharpen edges, and then run a Gaussian filter a second time. After this, we run a global histogram equalization, and then a sigmoid function which increases contrast in the range of roughly 0.35-0.70, and compresses it elsewhere. Then we apply the SLIC algorithm, with K=400, and compactness=0.07 (fig. 9b). When calculating the values for the superpixels, we tried using both the colors from the original green channel(fig. 9c), and the enhanced green channel(fig. 9d). We note that the enhancements make the contrast between the blood vessels and lesions and their surrounding quite high, the superpixels did not align very well with the regions of interest.

Furthermore, when finding a suitable range for the cutoff for the mask, no range was found that provided a satisfying trade-off between including regions of interests and regions not of interest. Any range chosen would necessarily include non-interesting regions. See figure 10 for these masks.

e)

Histogram equalization has the effect of increasing contrast in an image. It works by "shifting" each bin in the histogram such that the cumulutive distribution function approximates a straight line, which implies that the PDF approximates a flat line, and that the pixels are therefore approximately equally distributed across the colorspace. The Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm works by processing an image in segments, and after computing the new histogram (PDF), the histogram is cut off [3], at a certain threshold, and the missing "area" is added evenly throughout the histogram.

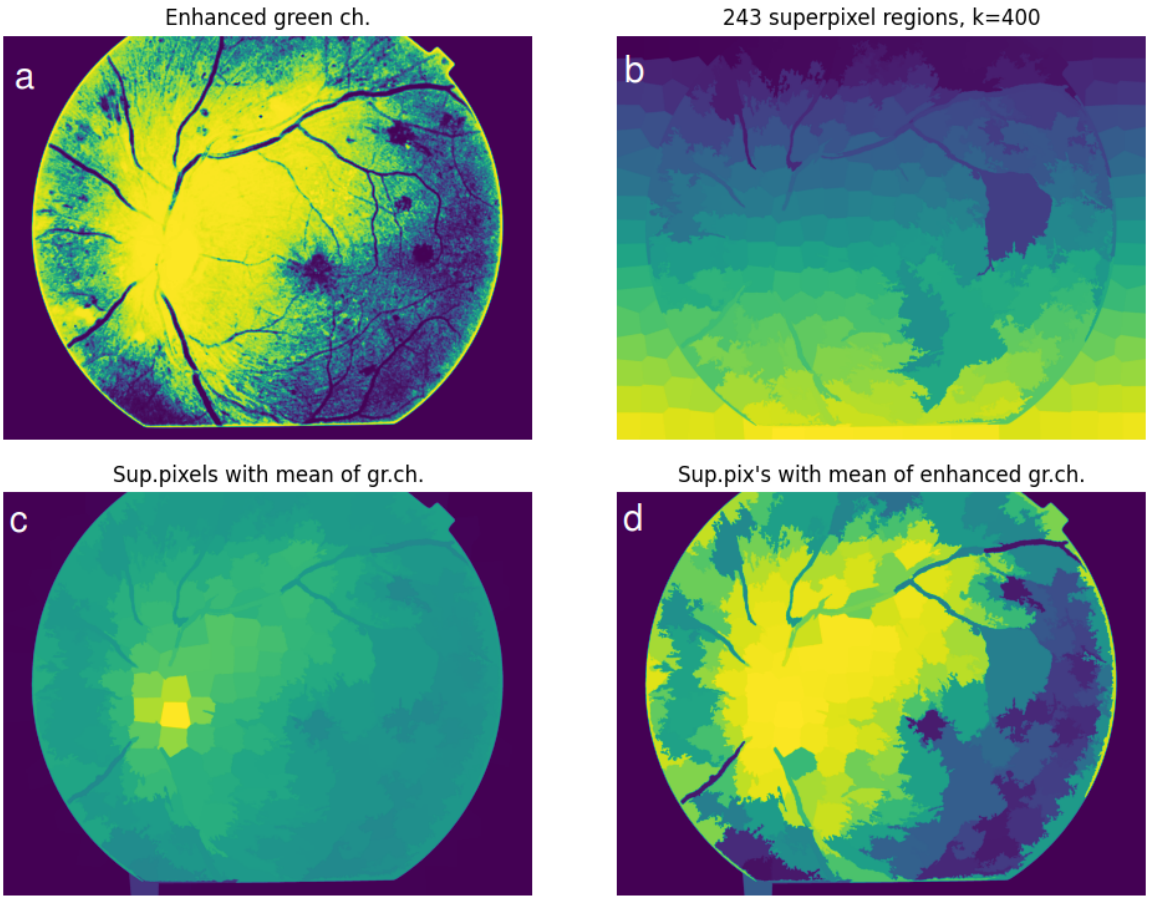

Figure 9: a: Enhanced green channel. b: Result of running SLIC on fig a. c: Using values from original green channel image to fill superpixels. d: Using values from enhanced green channel to fill superpixels.

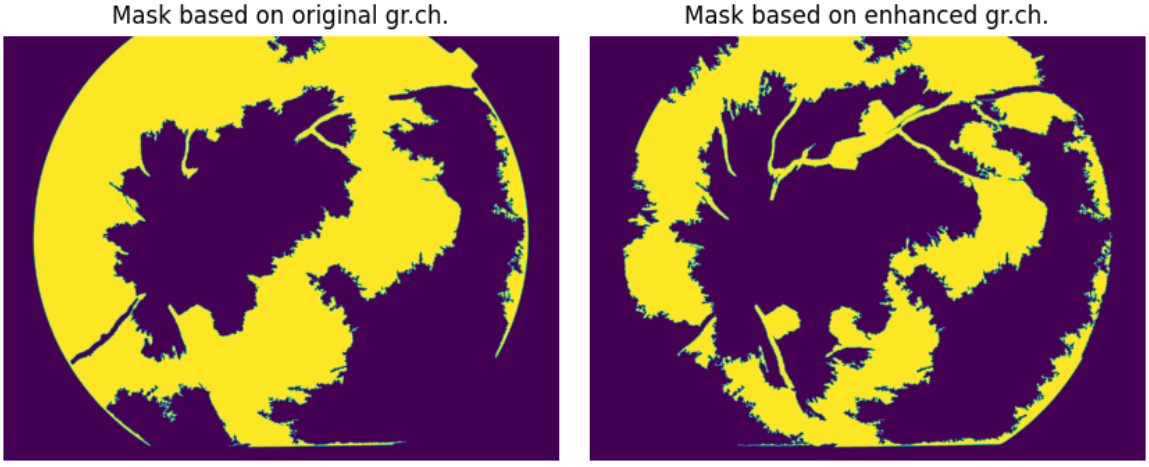

Figure 10: Masks corresponding to figures 9c and 9d

f)

For the final attempt at making a mask, we first apply CLAHE to the green channel, after which we apply unsharp mask, then gaussian filter, and then a sigmoid function(fig. 11a). We then apply SLIC to this enhanced image (fig. 11b). Then we use the enhanced image to fill the superpixels (fig. 11c), and create our final mask by removing all superpixels except those in a suitable range (fig. 11d). This range was acquired by inspecting the superpixels which we wanted to keep, and which ones we wanted to mask off. We note that the result, while not achieving full cover of the regions of interest, achieve better and more precise coverage than our attempts in figure 10.

For the implementation of these tasks, we refer to listings 5, 6, 7 and 8 below.

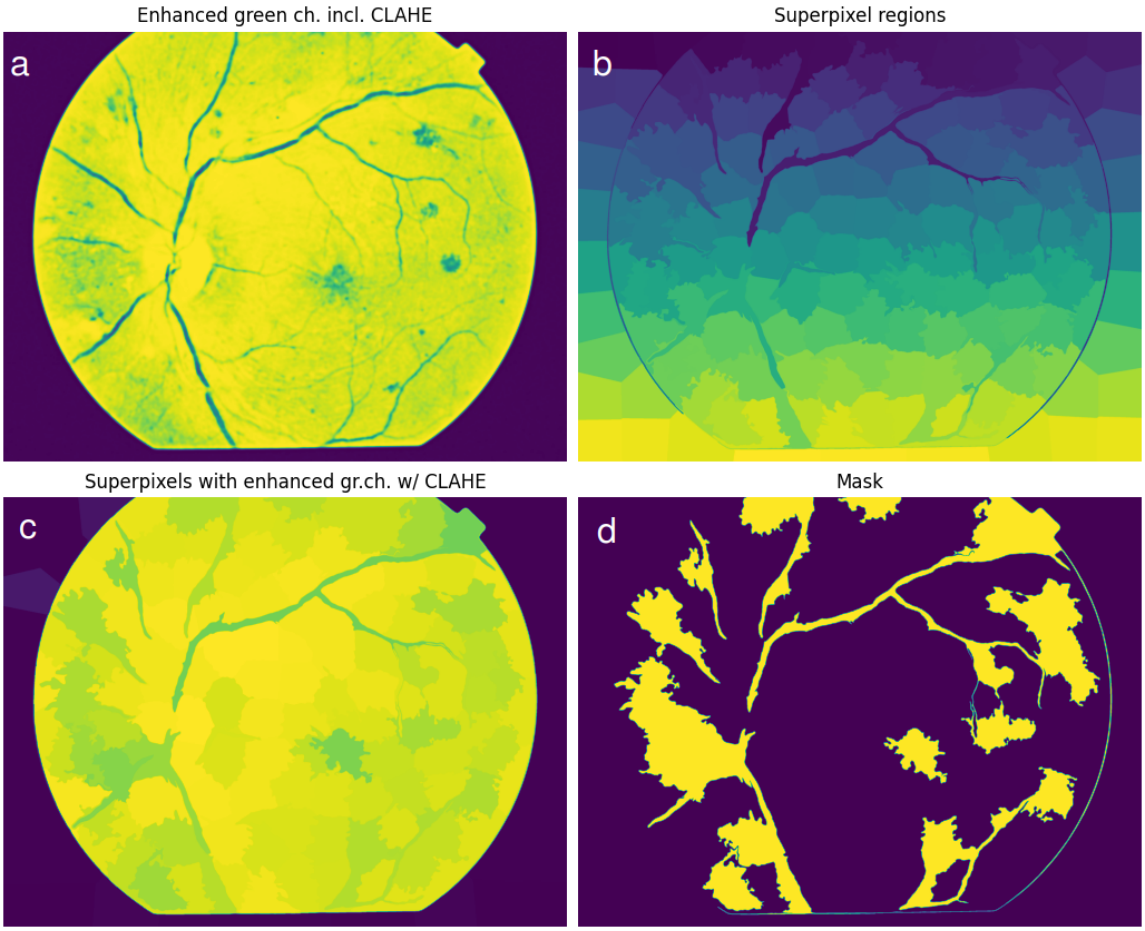

Figure 11: Final attempt at making a mask using CLAHE and SLIC. a: Preprocessed green channel. b: SLIC superpixels. c: Superpixels with average value computed from a. d: Final mask produced by keeping only a select range of spatial intensities from c.

Listing 5: Task 3b

```python
import numpy as np
from PIL import Image
from skimage import segmentation
from skimage.measure import regionprops
import matplotlib.pyplot as plt

K = 200

# Read image
org_img = np.array(Image.open("retina-image.png"))


# Calculate SLIC
slic = segmentation.slic(org_img, n_segments=K, compactness=0.1)

# Make array for resulting image
new_img = np.zeros_like(org_img, dtype=np.uint8)
# Calculate mean region value and write to region in resulting image
for k in range(2*K):
    if k in slic[:,:]:
        new_img[np.where(slic == k)] = np.average(org_img[np.where(slic == k)], axis=0)


fig, ax_arr = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(20, 20))
ax1, ax2, ax3 = ax_arr.ravel()

ax1.imshow(org_img)
ax1.set_title("Original image")

ax2.imshow(new_img)
ax2.set_title(f"{K} superpixels with avg. region value.")

ax3.imshow(slic)
ax3.set_title("Superpixel regions")

for ax in ax_arr.ravel():
    ax.set_axis_off()

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

Listing 6: Task 3c

```python
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

# Read image
org_img = np.array(Image.open("retina-image.png"))

fig, ax_arr = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(15, 15))
ax1, ax2, ax3 = ax_arr.ravel()

```

```
ax1.imshow(org_img[:,:,0], vmin=0, vmax=255)
ax1.set_title("Red ch.")

ax2.imshow(org_img[:,:,1], vmin=0, vmax=255)
ax2.set_title("Green ch.")

im = ax3.imshow(org_img[:,:,2], vmin=0, vmax=255)
ax3.set_title("Blue ch.")

for ax in ax_arr.ravel():
    ax.set_axis_off()

plt.tight_layout()
fig.subplots_adjust(right=0.8)
cbar_ax = fig.add_axes([0.81, 0.4, 0.03, 0.2])
fig.colorbar(im, cax=cbar_ax)

plt.show()
```

Listing 7: Task 3d

```
"""
Create a mask for blood vessels and lesions.
"""

from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
from skimage import segmentation
from skimage import exposure
from skimage.measure import regionprops
from skimage.filters import unsharp_mask
from scipy.ndimage import gaussian_filter

K = 400

# STEP 1
# Read green channel of image
org_img = np.array(Image.open("retina-image.png"))[:,:,1]
org_img = gaussian_filter(org_img, sigma=1.8)


# STEP 2
# Use image enhancements (denoise, sharpen) to make blood vessels
# more visible
fil_img_tmp = unsharp_mask(org_img, radius=10, amount=2.0)
fil_img = np.zeros_like(fil_img_tmp, dtype=np.uint8)
fil_img[:,:] = 255 * fil_img_tmp[:,:]

fil_img_tmp = gaussian_filter(fil_img, sigma=1)

fil_img_tmp = exposure.equalize_hist(fil_img_tmp)
fil_img_tmp = 1/(1+np.exp(-8*(fil_img_tmp-0.5)))
fil_img[:,:] = 255 * fil_img_tmp[:,:]

# STEP 3
# Apply SLIC again to the resultant image
```

```
# Show the segmentation, replace segments with their avg value
slic = segmentation.slic(fil_img, n_segments=K, compactness=0.07)

new_img = np.zeros_like(fil_img, dtype=np.uint8)
new_fil_img = np.zeros_like(fil_img, dtype=np.uint8)


k_arr = np.intersect1d(np.arange(2*K), slic)
for k in k_arr:
    idx = np.where(slic == k)
    new_img[idx] = np.average(org_img[idx], axis=0)
    new_fil_img[idx] = np.average(fil_img[idx], axis=0)

# STEP 4
# Select a threshold value and keep only segments associated with blood
# vessels, to obtain a blood vessel mask.

mask1 = np.where(new_img >= 100, 1, 0)
mask1 = np.where(new_img <= 118, mask1, 0)
mask2 = np.where(new_fil_img >= 124, 1, 0)
mask2 = np.where(new_fil_img <= 205, mask2, 0)

fig, ax_arr = plt.subplots(2, 4, sharex=True, sharey=True, figsize=(15, 15))
ax1, ax2, ax3, ax4, ax5, ax6, ax7, ax8 = ax_arr.ravel()

ax1.imshow(org_img)
ax1.set_title("Green ch. of orig. image")

ax2.imshow(fil_img)
ax2.set_title("Enhanced green ch.")

ax3.imshow(slic)
ax3.set_title(f"{len(k_arr)} superpixel regions, k={K}")

ax5.imshow(new_img)
ax5.set_title(f"Sup.pixels with mean of gr.ch.")

ax7.imshow(mask1)
ax7.set_title(f"Mask based on original gr.ch.")

ax8.imshow(new_fil_img)
ax8.set_title(f"Sup.pix's with mean of enhanced gr.ch.")

ax8.imshow(mask2)
ax8.set_title(f"Mask based on enhanced gr.ch.")

for ax in ax_arr.ravel():
    ax.set_axis_off()

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

Listing 8: Task 3f

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
```

```
4
5  from skimage.exposure import equalize_adapthist
6  from skimage.segmentation import slic
7  from skimage.filters import unsharp_mask
8  from scipy.ndimage import gaussian_filter
9
10 def sigmoid(x):
11     return 1 / (1 + np.exp(-11*(x-0.22)))
12
13 K = 200
14
15 # STEP 1
16 # Read green channel of image
17 org_img = np.array(Image.open("retina-image.png"))[:,:,1]
18
19 # Pre-process image
20 # Apply CLAHE
21 cla_img = equalize_adapthist(org_img)
22
23 # Sharpen edges
24 cla_img = unsharp_mask(cla_img, radius=10, amount=3.0)
25
26 # Apply Gaussian filter
27 cla_img = gaussian_filter(cla_img, sigma=7)
28
29 # Apply sigmoid
30 cla_img = np.where(sigmoid(cla_img) > 0, sigmoid(cla_img), 0)
31
32
33 # Apply SLIC
34 slic_img = slic(cla_img, n_segments=K, compactness=0.05) # K = 200, compactness = 0.05
35
36 cla_img *= (255 / np.max(cla_img))
37
38
39 # Calculate mean region value and write to region in resulting image
40 new_img = np.zeros_like(org_img)
41 for k in range(2*K):
42     if k in slic_img[:,:]:
43         new_img[np.where(slic_img == k)] = np.average(cla_img[np.where(slic_img == k)], axis=0)
44
45 # im2: (204-230)
46 mask = np.where(new_img >= 204, 1, 0)
47 mask = np.where(new_img <= 230, mask, 0)
48
49 fig, ax_arr = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(15, 15))
50 ax1, ax2, ax3, ax4 = ax_arr.ravel()
51
52
53 ax1.imshow(cla_img)
54 ax1.set_title("Enhanced green ch. incl. CLAHE")
55
56 ax2.imshow(slic_img)
57 ax2.set_title("Superpixel regions")
58
59 ax3.imshow(new_img)
60 ax3.set_title("Superpixels with enhanced gr.ch. w/ CLAHE")
```

```
61
62  ax4.imshow(mask)
63  ax4.set_title("Mask")
64
65  for ax in ax_arr.ravel():
66      ax.set_axis_off()
67  plt.tight_layout()
68  plt.show()
```

References

[1] R.C. Gonzalez and R.E. Woods. *Digital Image Processing, eBook, Global Edition*. Pearson Education, 2018. ISBN: 9781292223070. URL: https://books.google.no/books?id=Vma8vQEACAAJ.

[2] *Module: filters-skimage v0.19.2 docs*. https://scikit-image.org/docs/stable/api/skimage.filters.html. Accessed: 2022-05-19.

[3] Koki HONDA et al. "CLAHE Implementation and Evaluation on a Low-End FPGA Board by High-Level Synthesis". In: *IEICE Transactions on Information and Systems* E104.D (Dec. 2021), pp. 2048–2056. DOI: 10.1587/transinf.2021PAP0006.