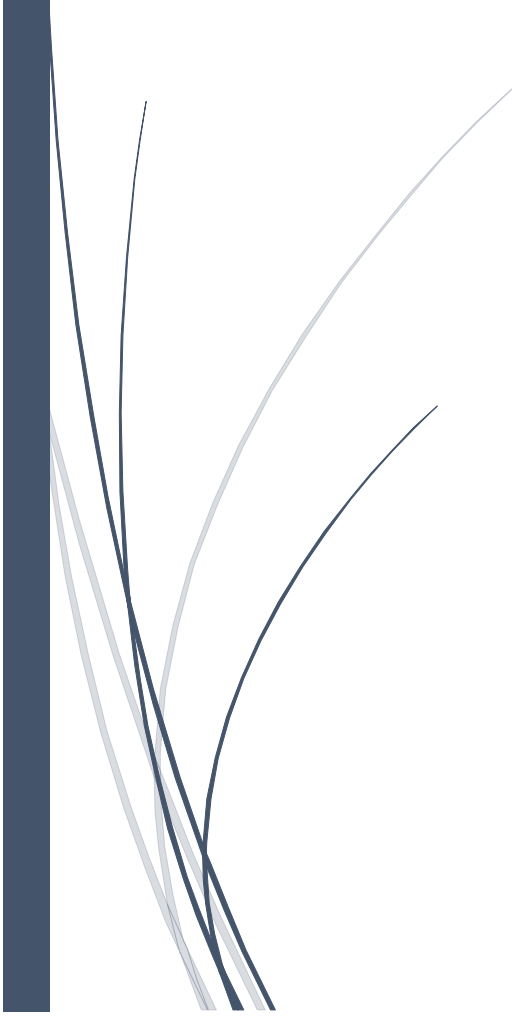
Project 1

Image Processing

2021-2022 Fall Semester



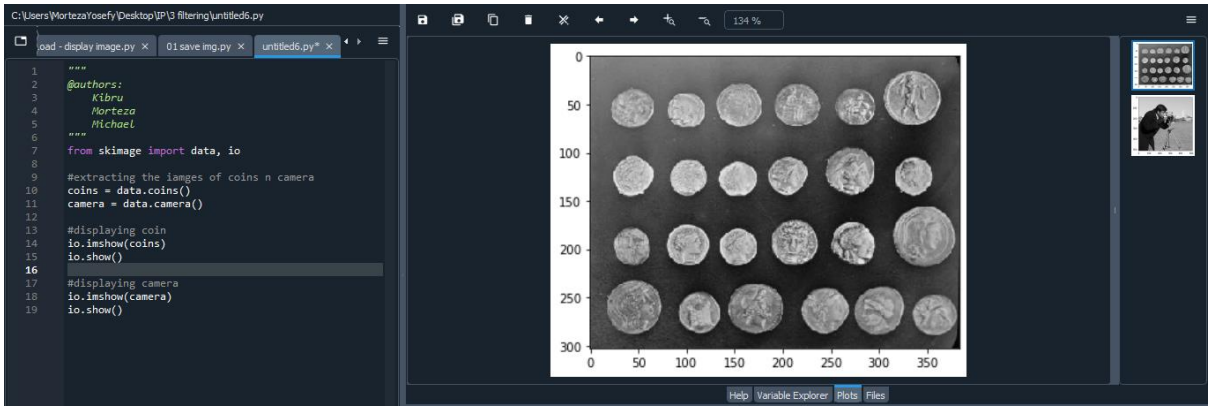
Group Members

- Morteza Yosefy
- Kibru Joba Kuture
- Michael Derece Kebede

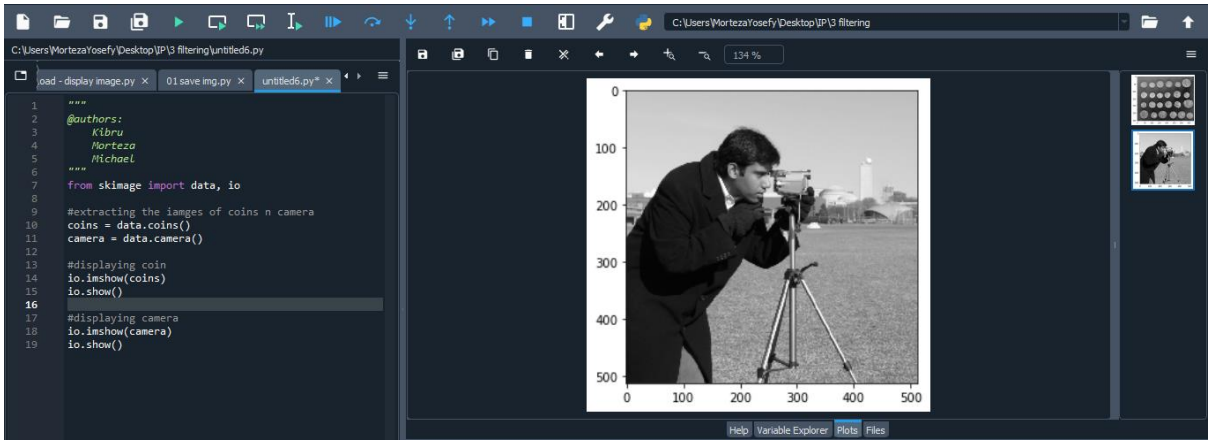
Table of Contents

Load/Read or Saving Images

```
"""
@authors:
    Kibru
    Morteza
    Michael
"""
from skimage import data, io

#extracting the iamges of coins n camera
coins = data.coins()
camera = data.camera()

#displaying coin
io.imshow(coins)
io.show()
```



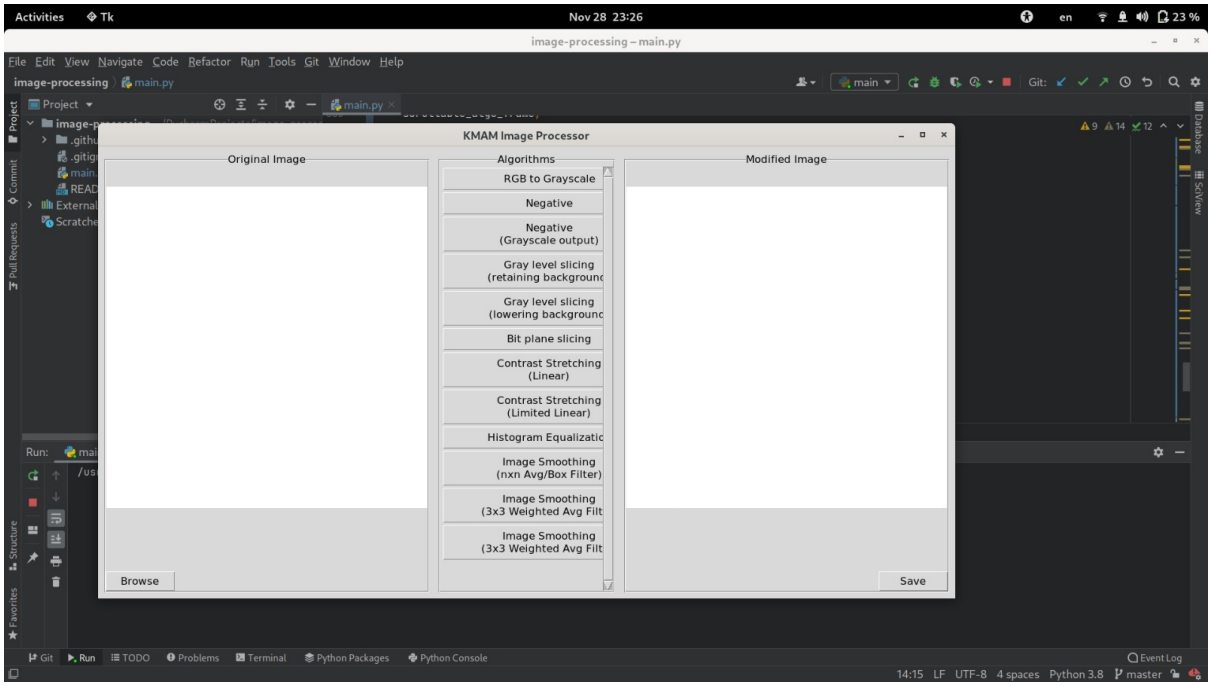

```
#displaying camera
io.imshow(camera)
io.show()
```



And to save an image we can use the following codes:

```
"""
@authors:
    Kibru
    Morteza
    Michael
"""
from skimage import data
import cv2 as cv
pic = data.camera()
cv.imshow('camera', pic)
k = cv.waitKey(0) & 0xFF
if k == 27:
    cv.destroyAllWindows()
#press 's' to save
elif k == ord('s'):
    cv.imwrite(r'C:\image.png', pic)
    cv.destroyAllWindows()
```

User Interface



```
# The program implements various image processing techniques

# import dependencies
import ctypes
import os
from tkinter import *
from tkinter import filedialog, ttk
import cv2
from matplotlib import image as mpimg
from matplotlib import pyplot as plt
import numpy as np
from PIL import Image, ImageTk
import scipy.signal

# use with windows envs only
#ctypes.windll.shcore.SetProcessDpiAwareness(True)

# create root window
root = Tk()
ttk.Style().configure("TButton", justify=CENTER)

# Global variables
width_gui = 1366
height_gui = 720
input_file = ""
output_file = ""
loaded_image = None
edited_image = None
arg_from_usr = None
popup_dialog = None
popup_input_dialog = None


root.title("KMAM Image Processor")
# set minimum size of gui
root.minsize(width_gui, height_gui)


# get user input for some functions
def set_user_arg():
    global arg_from_usr
    arg_from_usr = popup_input_dialog.get()
    popup_dialog.destroy()
    popup_dialog.quit()

#
def open_popup_input(text):
    global popup_dialog, popup_input_dialog
    popup_dialog = Toplevel(root)
```

```
    popup_dialog.resizable(False, False)
    popup_dialog.title("User Input")
    text_label = ttk.Label(popup_dialog, text=text, justify=LEFT)
    text_label.pack(side=TOP, anchor=W, padx=15, pady=10)
    popup_input_dialog = ttk.Entry(popup_dialog)
    popup_input_dialog.pack(side=TOP, anchor=NW, fill=X, padx=15)
    popup_btn = ttk.Button(popup_dialog, text="OK",
command=set_user_arg).pack(pady=10)
    popup_dialog.geometry(f"400x{104 + text_label.winfo_reqheight()}")
    popup_input_dialog.focus()
    popup_dialog.mainloop()


def draw_before_canvas():
    global loaded_image, input_file
    loaded_image = Image.open(input_file)
    loaded_image = loaded_image.convert("RGB")
    img = ImageTk.PhotoImage(loaded_image)
    before_canvas.create_image(
        256,
        256,
        image=img,
        anchor="center",
    )
    before_canvas.img = img


def draw_after_canvas(mimg):
    global edited_image

    edited_image = Image.fromarray(np.uint8(mimg))
    img = ImageTk.PhotoImage(edited_image)
    after_canvas.create_image(
        256,
        256,
        image=img,
        anchor="center",
    )
    after_canvas.img = img


def load_file():
    global input_file
    input_file = filedialog.askopenfilename(
        title="Open an image file",
        initialdir=".",
        filetypes=[("All Image Files", "*.*")],
    )
    draw_before_canvas()
```

```
    # print(f"Image loaded from: {input_file}")


def save_file():
    global input_file, loaded_image, edited_image
    file_ext = os.path.splitext(input_file)[1][1:]
    op_file = filedialog.asksaveasfilename(
        filetypes=[
            (
                f"{file_ext.upper()}",
                f"*.{file_ext}",
            )
        ],
        defaultextension=[
            (
                f"{file_ext.upper()}",
                f"*.{file_ext}",
            )
        ],
    )
    edited_image = edited_image.convert("RGB")
    edited_image.save(op_file)
    # print(f"Image saved at: {output_file}")


# frames
left_frame = ttk.LabelFrame(root, text="Original Image", labelanchor=N)
left_frame.pack(fill=BOTH, side=LEFT, padx=10, pady=10, expand=1)

middle_frame = ttk.LabelFrame(root, text="Algorithms", labelanchor=N)
middle_frame.pack(fill=BOTH, side=LEFT, padx=5, pady=10)

right_frame = ttk.LabelFrame(root, text="Modified Image", labelanchor=N)
right_frame.pack(fill=BOTH, side=LEFT, padx=10, pady=10, expand=1)

# left frame contents
before_canvas = Canvas(left_frame, bg="white", width=512, height=512)
before_canvas.pack(expand=1)

browse_btn = ttk.Button(left_frame, text="Browse", command=load_file)
browse_btn.pack(expand=1, anchor=SW, pady=(5, 0))

# middle frame contents
algo_canvas = Canvas(middle_frame, width=260, highlightthickness=0)
scrollable_algo_frame = Frame(algo_canvas)
scrollbar = Scrollbar(
    middle_frame, orient="vertical", command=algo_canvas.yview, width=15
)
scrollbar.pack(side="right", fill="y")
```

```
algo_canvas.pack(fill=BOTH, expand=1)
algo_canvas.configure(yscrollcommand=scrollbar.set)
algo_canvas.create_window((0, 0), window=scrollable_algo_frame, anchor="nw")
scrollable_algo_frame.bind(
    "<Configure>", lambda _:
algo_canvas.configure(scrollregion=algo_canvas.bbox("all"))
)


# right frame contents
after_canvas = Canvas(right_frame, bg="white", width=512, height=512)
after_canvas.pack(expand=1)

save_btn = ttk.Button(right_frame, text="Save", command=save_file)
save_btn.pack(expand=1, anchor=SE, pady=(5, 0))


# algorithm fns
def RGB2Gray():
    img = mpimg.imread(input_file)
    R, G, B = img[:, :, 0], img[:, :, 1], img[:, :, 2]
    return 0.299 * R + 0.58 * G + 0.114 * B


def callRGB2Gray():
    grayscale = RGB2Gray()
    draw_after_canvas(grayscale)


def negative(set_gray):
    img = RGB2Gray() if (set_gray) else Image.open(input_file)
    img = np.array(img)
    img = 255 - img
    draw_after_canvas(img)


def gray_slice(img, lower_limit, upper_limit, fn):
    # general function
    if lower_limit <= img <= upper_limit:
        return 255
    else:
        return fn


def call_gray_slice(retain):
    img = RGB2Gray()
    # input 100,180
    open_popup_input("Enter lower limit, upper limit\n(Separate inputs with a
comma)")
```

```
    arg_list = arg_from_usr.replace(" ", "").split(",")
    print(arg_list)
    lower_limit = int(arg_list[0])
    upper_limit = int(arg_list[1])
    img_thresh = np.vectorize(gray_slice)
    fn = img if retain else 0
    draw_after_canvas(img_thresh(img, lower_limit, upper_limit, fn))


def bit_slice(img, k):
    # create an image for the k bit plane
    plane = np.full((img.shape[0], img.shape[1]), 2 ** k, np.uint8)
    # execute bitwise and operation
    res = cv2.bitwise_and(plane, img)
    # multiply ones (bit plane sliced) with 255 just for better visualization
    return res * 255


def call_bit_slice():
    global arg_from_usr
    bitplanes = []
    img = cv2.imread(input_file, 0)
    open_popup_input(
        "Enter bit plane no k (0-7)\n(or leave it blank to display all 8
planes together)"
    )
    if not arg_from_usr:
        for k in range(9):
            slice = bit_slice(img, k)
            # append to the output list
            slice = cv2.resize(slice, (171, 171))
            bitplanes.append(slice)

        # concat all 8 bit planes into one image
        row1 = cv2.hconcat([bitplanes[0], bitplanes[1], bitplanes[2]])
        row2 = cv2.hconcat([bitplanes[3], bitplanes[4], bitplanes[5]])
        row3 = cv2.hconcat([bitplanes[6], bitplanes[7], bitplanes[8]])
        final_img = cv2.vconcat([row1, row2, row3])
    else:
        final_img = bit_slice(img, int(arg_from_usr))

    draw_after_canvas(final_img)


def c_stretch(img, r1, r2, s1, s2):
    # general function
    if img < r1:
        return s1
    elif img > r2:
```

```
        return s2
    else:
        return s1 + ((s2 - s1) * (img - r1) / (r2 - r1))


def call_c_stretch(limited):
    # input
    img = RGB2Gray()
    r1 = np.min(img)
    r2 = np.max(img)
    if limited:
        # input 25,220
        open_popup_input("Enter s1,s2\n(Separate inputs with a comma)")
        arg_list = arg_from_usr.replace(" ", "").split(",")
        s1, s2 = int(arg_list[0]), int(arg_list[1])
    else:
        s1, s2 = (0, 255)
    image_cs = np.vectorize(c_stretch)
    draw_after_canvas(image_cs(img, r1, r2, s1, s2))
```

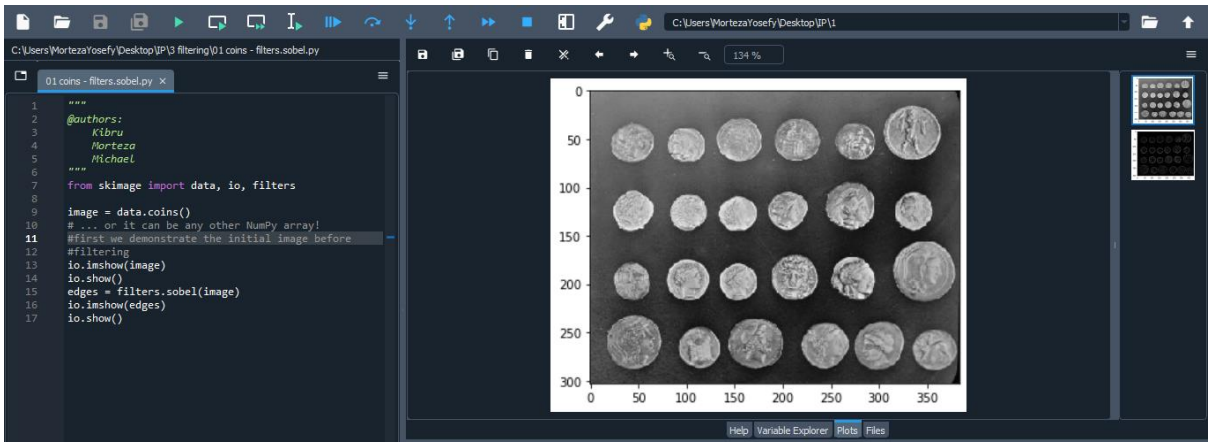
Filtering

Sobel Filter

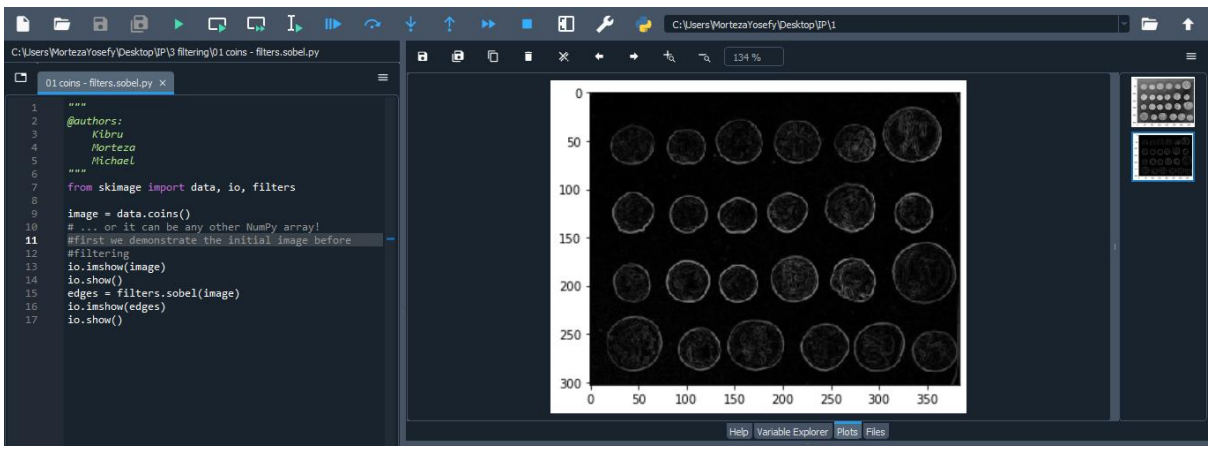
Find edges in an image using the Sobel filter.

```
"""
@authors:
   Kibru
   Morteza
   Michael
"""

from skimage import data, io, filters
image = data.coins()
io.imshow(image)
io.show()
```

Here, first we demonstrate the initial image before the filtering.



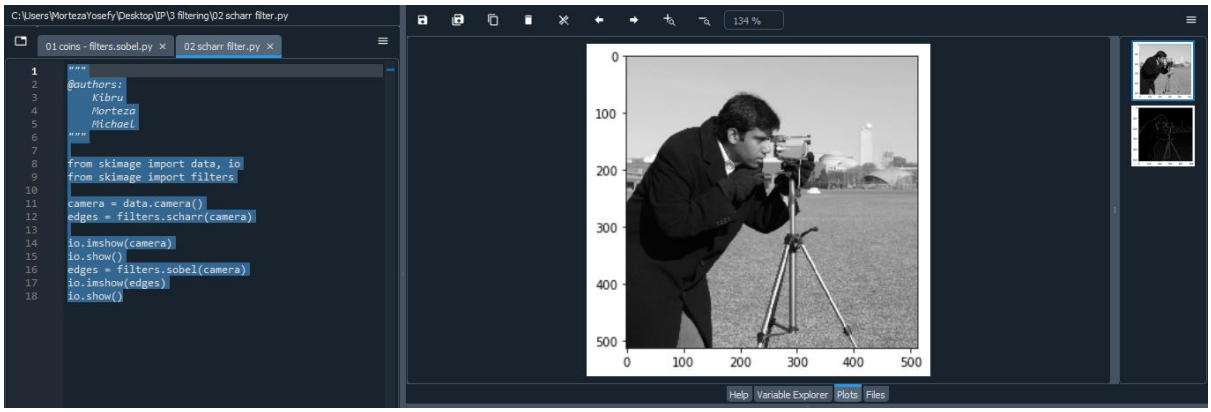

```
edges = filters.sobel(image)
io.imshow(edges)
io.show()
```



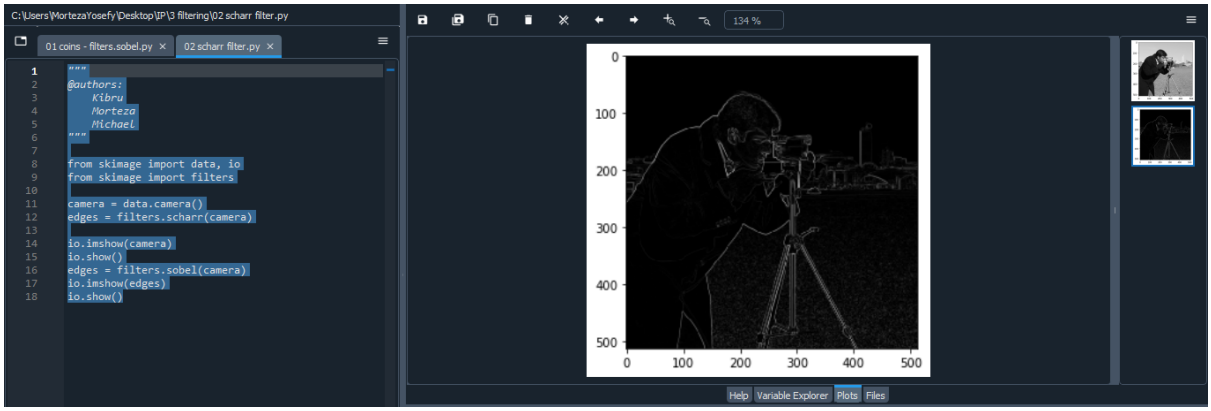
Scharr Filter

Find the edge magnitude using the Scharr transform.

```
"""
@authors:
    Kibru
    Morteza
    Michael
"""
from skimage import data, io
from skimage import filters

camera = data.camera()
edges = filters.scharr(camera)

io.imshow(camera)
io.show()
```




```
edges = filters.sobel(camera)
io.imshow(edges)
io.show()
```

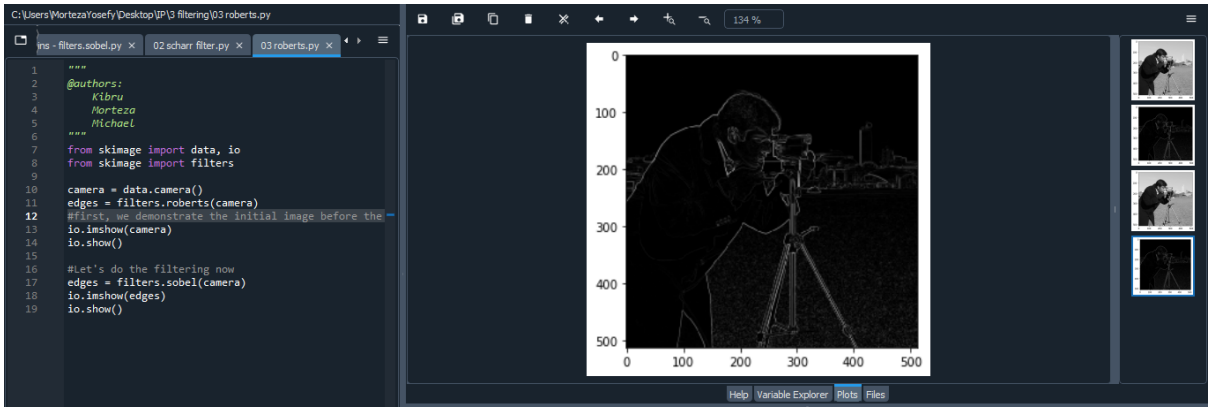


The Scharr operator has a better rotation invariance than other edge filters such as the Sobel or the Prewitt operators.

Roberts

Find the edge magnitude using Roberts' cross operator.

```
"""
@authors:
    Kibru
    Morteza
    Michael
"""
from skimage import data, io
from skimage import filters

camera = data.camera()
edges = filters.roberts(camera)
#first, we demonstrate the initial image before the filtering
io.imshow(camera)
io.show()

#Let's do the filtering now
edges = filters.sobel(camera)
io.imshow(edges)
io.show()
```



Window

Return an n-dimensional window of a given size and dimensionality.

```
from skimage.filters import window
#Return a Hann window with shape (512, 512):
w = window('hann', (512, 512))
#Return a Kaiser window with beta parameter of 16 and shape (256, 256, 35):
w = window(16, (256, 256, 35))
#Return a Tukey window with an alpha parameter of 0.8 and shape (100, 300):
w = window(('tukey', 0.8), (100, 300))
```

LPIFilter2D Filter

Linear Position-Invariant Filter (2-dimensional)

```
def filter_func(r, c, sigma = 1):
        return np.exp(-np.hypot(r, c)/sigma)
filter = LPIFilter2D(filter_func)
```

Unsharp Mask

The unsharp mask filter; the sharp details are identified as the difference between the original image and its blurred version. These details are then scaled and added back to the original image.

```
array = np.ones(shape=(5,5), dtype=np.uint8)*100
array[2,2] = 120
array

np.around(unsharp_mask(array, radius=0.5, amount=2),2)

array = np.ones(shape=(5,5), dtype=np.int8)*100
array[2,2] = 127
np.around(unsharp_mask(array, radius=0.5, amount=2),2)

np.around(unsharp_mask(array, radius=0.5, amount=2, preserve_range=True), 2)
```

Try all threshold

Returns a figure comparing the outputs of different thresholding methods.

```
from skimage.data import text

fig, ax = try_all_threshold(text(), figsize=(10, 6), verbose=False)
```

Threshold_yen

Return threshold value based on Yen's method. Either image or hist must be provided. In case hist is given, the actual histogram of the image is ignored.

```
from skimage.data import camera
image = camera()
thresh = threshold_yen(image)
binary = image <= thresh
```

Gaussian

Multi-dimensional Gaussian filter.

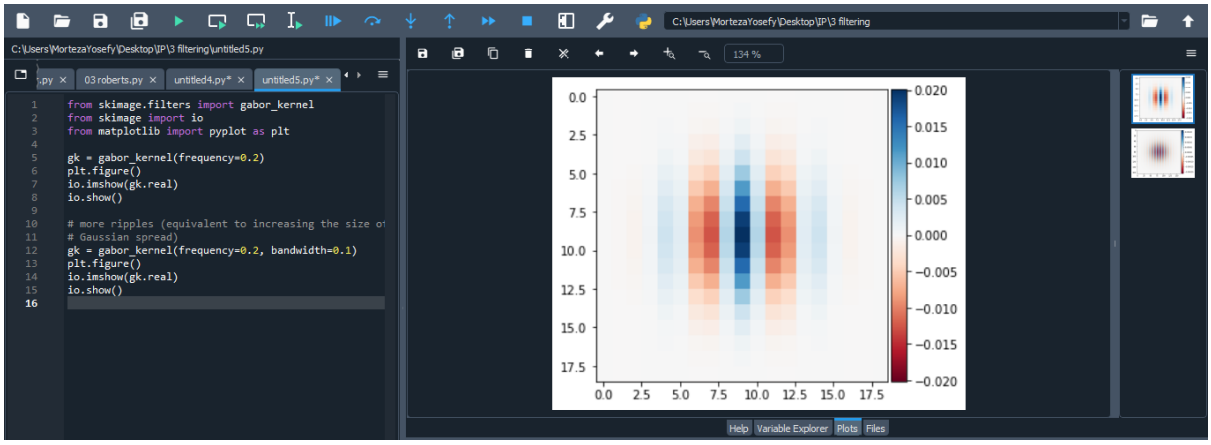
```
a = np.zeros((3, 3))
a[1, 1] = 1
a
gaussian(a, sigma=0.4)  # mild smoothing

gaussian(a, sigma=1)  # more smoothing

# Several modes are possible for handling boundaries
gaussian(a, sigma=1, mode='reflect')

# For RGB images, each is filtered separately
from skimage.data import astronaut
image = astronaut()
filtered_img = gaussian(image, sigma=1, multichannel=True)
```

Gabor kernel

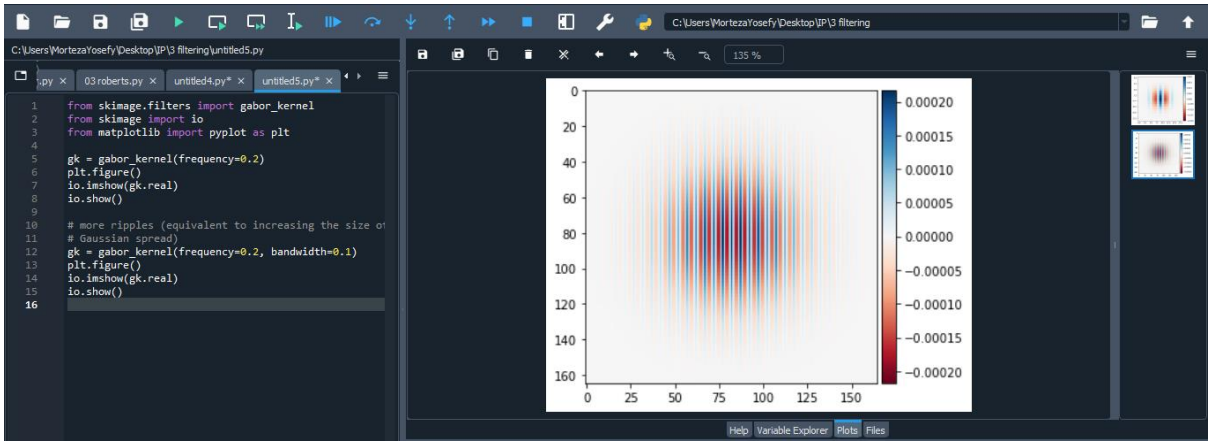
Return complex 2D Gabor filter kernel.

Gabor kernel is a Gaussian kernel modulated by a complex harmonic function. Harmonic function consists of an imaginary sine function and a real cosine function. Spatial frequency is inversely proportional to the wavelength of the harmonic and to the standard deviation of a Gaussian kernel. The bandwidth is also inversely proportional to the standard deviation.

```
from skimage.filters import gabor_kernel
from skimage import io
from matplotlib import pyplot as plt

gk = gabor_kernel(frequency=0.2)
plt.figure()
io.imshow(gk.real)
io.show()
```



```
# more ripples (equivalent to increasing the size of the
# Gaussian spread)
gk = gabor_kernel(frequency=0.2, bandwidth=0.1)
plt.figure()
io.imshow(gk.real)
io.show()
```



Histogram

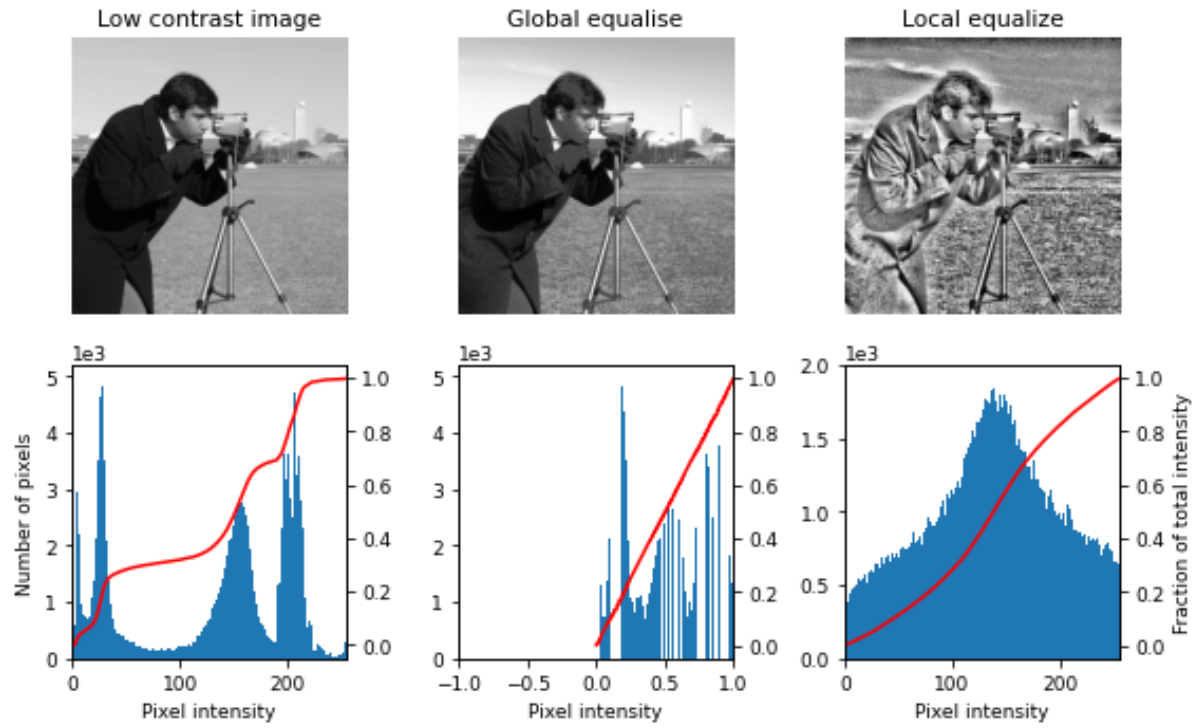
Local Histogram Equalization

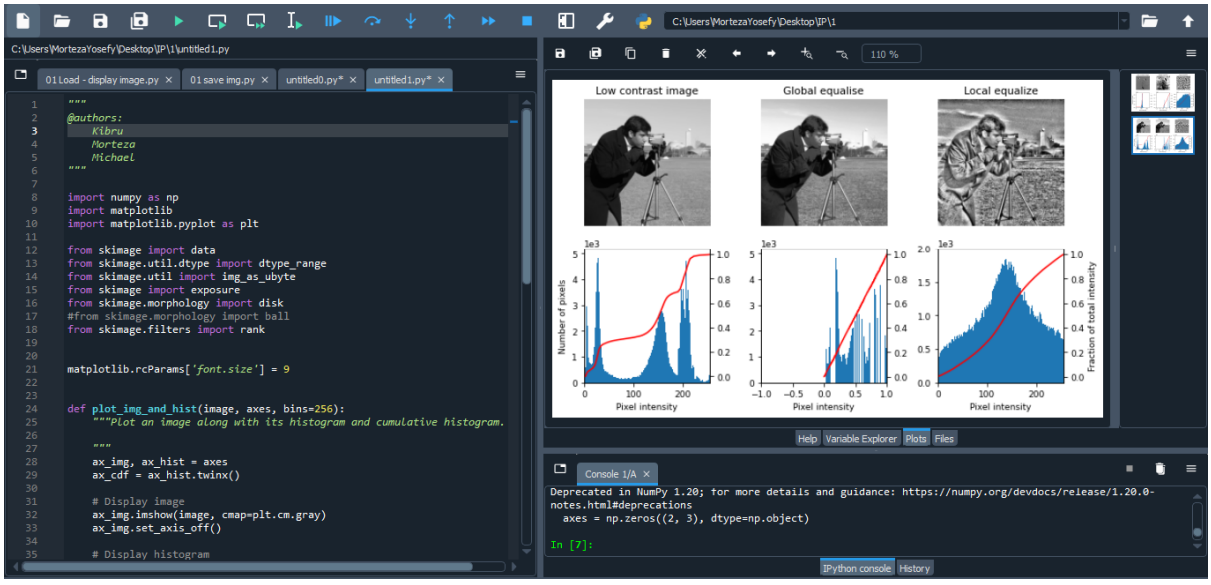
This example enhances an image with low contrast, using a method called *local histogram equalization*, which spreads out the most frequent intensity values in an image.

The equalized image 1 has a roughly linear cumulative distribution function for each pixel neighbourhood.

The local version 2 of the histogram equalization emphasized every local Gray level variations.

These algorithms can be used on both 2D and 3D images.



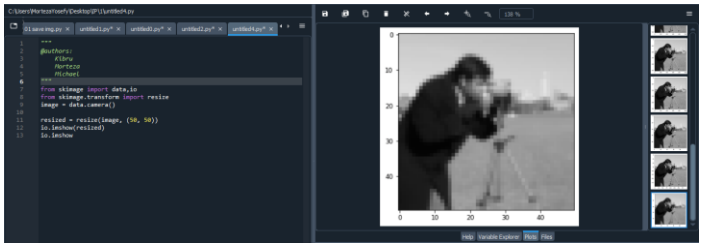


```
"""
@authors:
   Kibru
   Morteza
   Michael
"""

#Importing dependencies
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

from skimage import data
from skimage.util.dtype import dtype_range
from skimage.util import img_as_ubyte
from skimage import exposure
from skimage.morphology import disk
#from skimage.morphology import ball
from skimage.filters import rank


matplotlib.rcParams['font.size'] = 9


def plot_img_and_hist(image, axes, bins=256):
   """Plot an image along with its histogram and cumulative histogram.

   """
   ax_img, ax_hist = axes
   ax_cdf = ax_hist.twinx()

   # Display image
   ax_img.imshow(image, cmap=plt.cm.gray)
   ax_img.set_axis_off()

   # Display histogram
   ax_hist.hist(image.ravel(), bins=bins)
   ax_hist.ticklabel_format(axis='y', style='scientific', scilimits=(0, 0))
   ax_hist.set_xlabel('Pixel intensity')

   xmin, xmax = dtype_range[image.dtype.type]
   ax_hist.set_xlim(xmin, xmax)

   # Display cumulative distribution
   img_cdf, bins = exposure.cumulative_distribution(image, bins)
   ax_cdf.plot(bins, img_cdf, 'r')

   return ax_img, ax_hist, ax_cdf

# Load an example image
img = img_as_ubyte(data.camera())
```

```
# Global equalize
img_rescale = exposure.equalize_hist(img)

# Equalization
selem = disk(30)
img_eq = rank.equalize(img, selem=selem)


# Display results
fig = plt.figure(figsize=(8, 5))
axes = np.zeros((2, 3), dtype=np.object)
axes[0, 0] = plt.subplot(2, 3, 1)
axes[0, 1] = plt.subplot(2, 3, 2, sharex=axes[0, 0], sharey=axes[0, 0])
axes[0, 2] = plt.subplot(2, 3, 3, sharex=axes[0, 0], sharey=axes[0, 0])
axes[1, 0] = plt.subplot(2, 3, 4)
axes[1, 1] = plt.subplot(2, 3, 5)
axes[1, 2] = plt.subplot(2, 3, 6)

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img, axes[:, 0])
ax_img.set_title('Low contrast image')
ax_hist.set_ylabel('Number of pixels')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_rescale, axes[:, 1])
ax_img.set_title('Global equalise')

ax_img, ax_hist, ax_cdf = plot_img_and_hist(img_eq, axes[:, 2])
ax_img.set_title('Local equalize')
ax_cdf.set_ylabel('Fraction of total intensity')


# prevent overlap of y-axis labels
fig.tight_layout()
```

Estimate_transform

```
"""
@authors:
   Kibru
   Morteza
   Michael
"""
from skimage import data,io
from skimage.transform import resize
image = data.camera()

resized = resize(image, (50, 50))
io.imshow(resized)
io.imshow
```

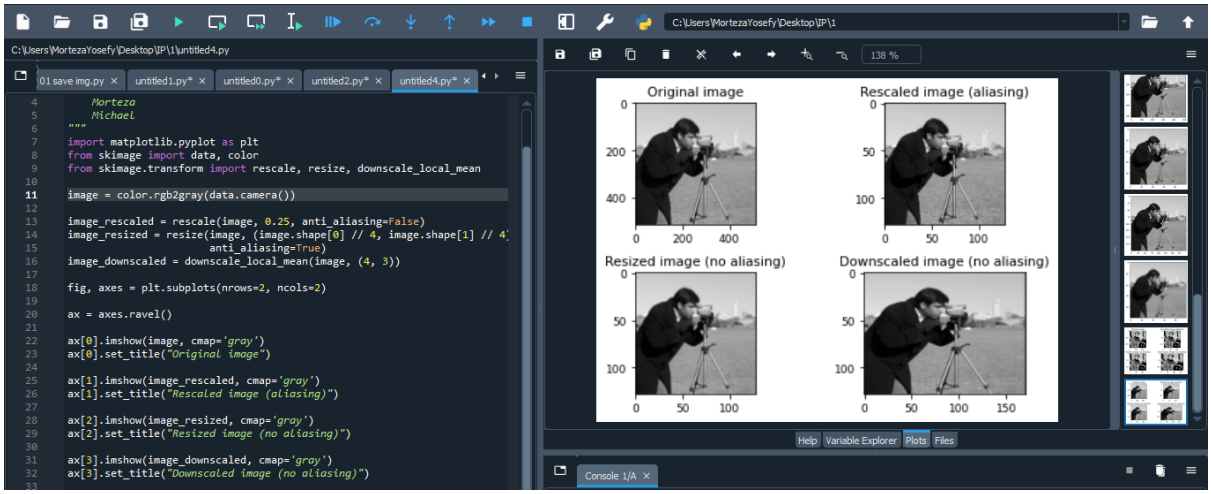


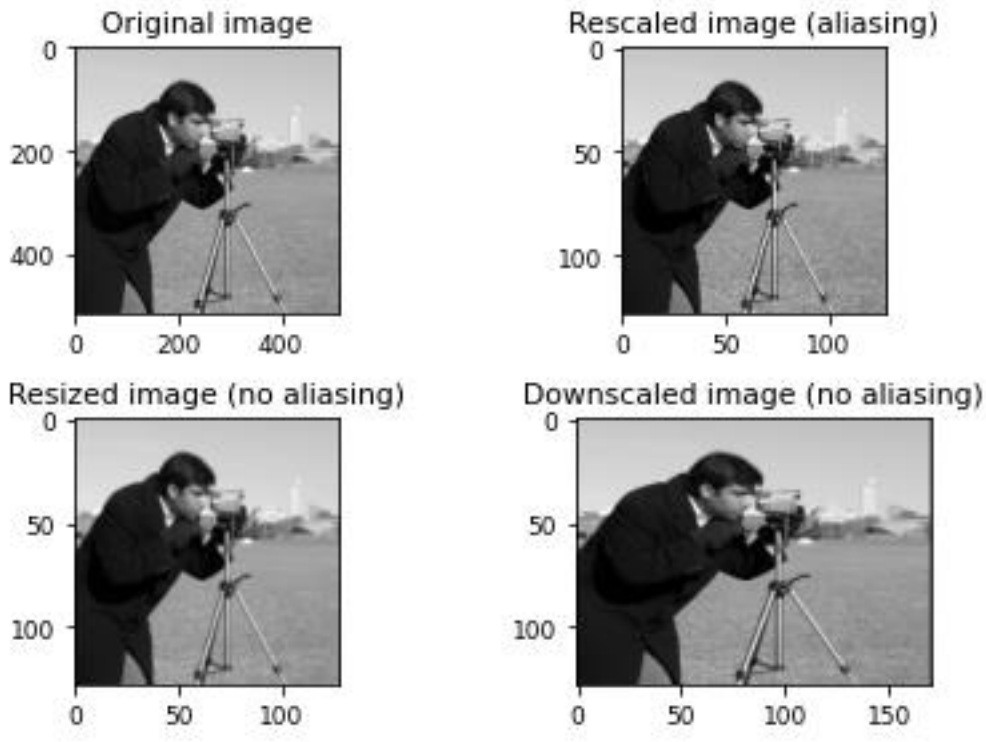
Rescale, resize, and downscale

Rescale operation resizes an image by a given scaling factor. The scaling factor can either be a single floating point value, or multiple values - one along each axis.

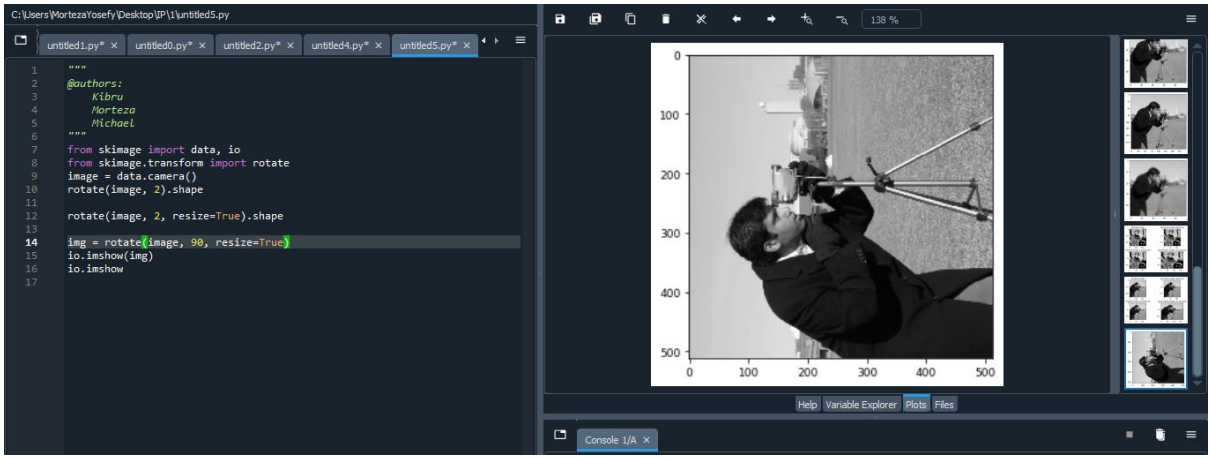
Note that when down-sampling an image, *resize* and *rescale* should perform Gaussian smoothing to avoid aliasing artifacts. See the *anti_aliasing* and *anti_aliasing_sigma* arguments to these functions.

Downscale serves the purpose of down-sampling an n-dimensional image by integer factors using the local mean on the elements of each block of the size factors given as a parameter to the function.



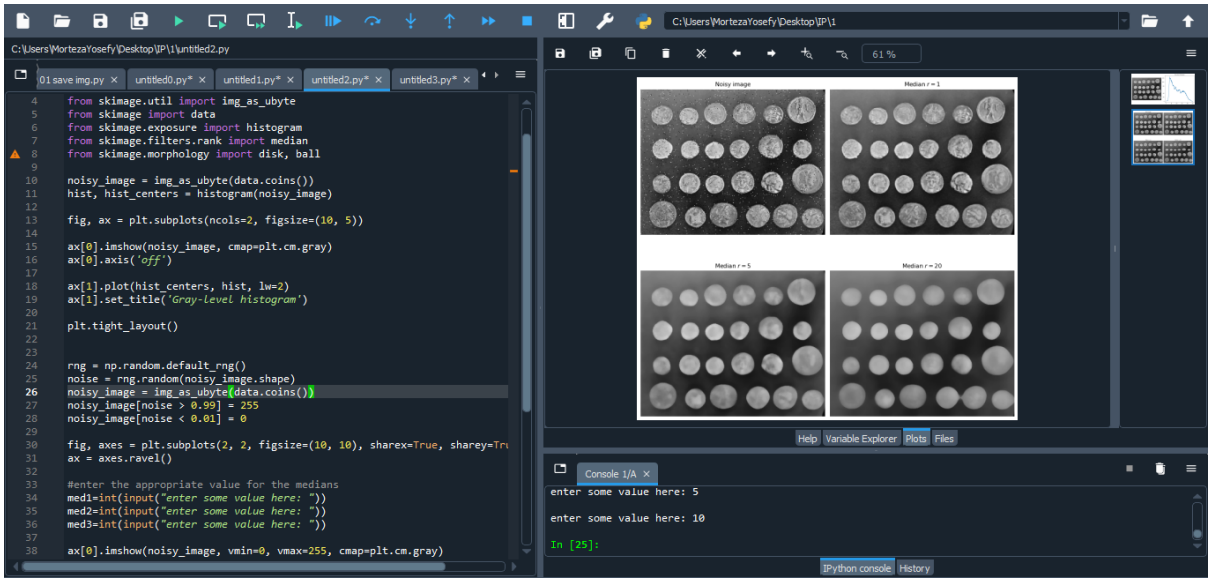

Original image
Rescaled image (aliasing)
Resized image (no aliasing)
Downscaled image (no aliasing)

Rotate


"""
@authors:
Kibru
Morteza
Michael
"""
from skimage import data, io
from skimage.transform import rotate
image = data.camera()
rotate(image, 2).shape
rotate(image, 2, resize=True).shape
img = rotate(image, 90, resize=True)
io.imshow(img)
io.imshow

skimage.exposure.rescale_intensity

Some noise is added to the image: 1% of pixels are randomly set to 255, 1% are randomly set to 0. The **median** filter is applied to remove the noise.



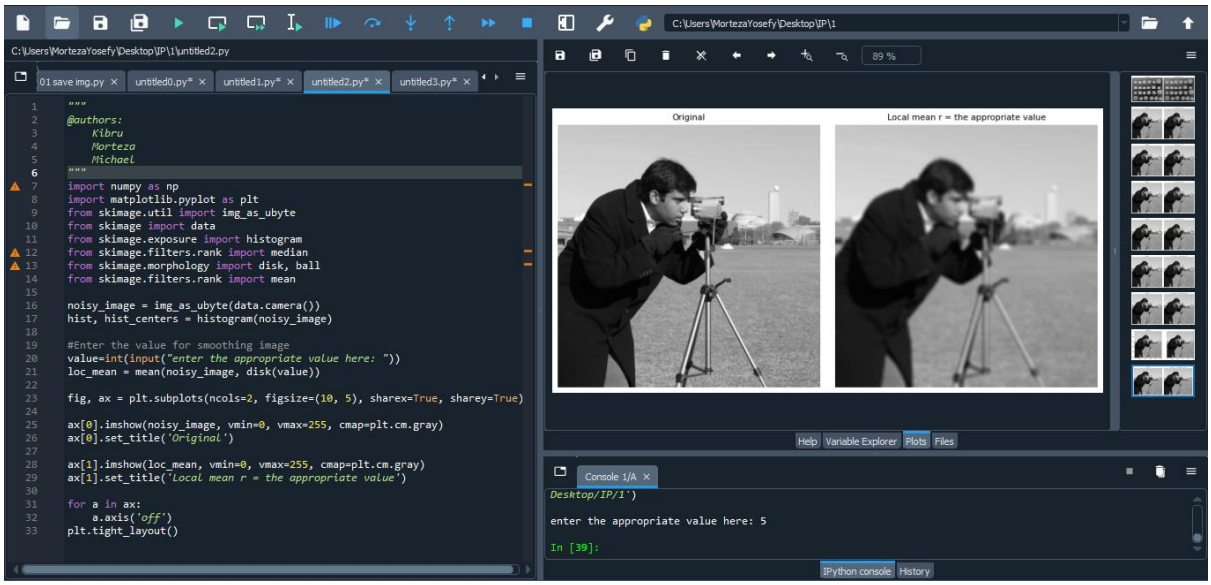

```
from skimage.util import img_as_ubyte
from skimage import data
from skimage.exposure import histogram
from skimage.filters.rank import median
from skimage.morphology import disk, ball

noisy_image = img_as_ubyte(data.coins())
hist, hist_centers = histogram(noisy_image)

fig, ax = plt.subplots(ncols=2, figsize=(10, 5))

ax[0].imshow(noisy_image, cmap=plt.cm.gray)
ax[0].axis('off')

ax[1].plot(hist_centers, hist, lw=2)
ax[1].set_title('Gray-level histogram')

plt.tight_layout()

rng = np.random.default_rng()
noise = rng.random(noisy_image.shape)
noisy_image = img_as_ubyte(data.coins())
noisy_image[noise > 0.99] = 255
noisy_image[noise < 0.01] = 0

fig, axes = plt.subplots(2, 2, figsize=(10, 10), sharex=True, sharey=True)
ax = axes.ravel()
```

User can set the median value here:

```
#enter the appropriate value for the medians
med1=int(input("enter some value here: "))
med2=int(input("enter some value here: "))
med3=int(input("enter some value here: "))

ax[0].imshow(noisy_image, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[0].set_title('Noisy image')

ax[1].imshow(median(noisy_image, disk(med1)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[1].set_title('Median $r=1$')

ax[2].imshow(median(noisy_image, disk(med2)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[2].set_title('Median $r=5$')

ax[3].imshow(median(noisy_image, disk(med3)), vmin=0, vmax=255, cmap=plt.cm.gray)
ax[3].set_title('Median $r=20$')

for a in ax:
    a.axis('off')

plt.tight_layout()
```

Image Smoothing

The example hereunder shows how a local **mean** filter smooths the camera man image.




```
"""
@authors:
   Kibru
   Morteza
   Michael
"""
```

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.util import img_as_ubyte
from skimage import data
from skimage.exposure import histogram
from skimage.filters.rank import median
from skimage.morphology import disk, ball
from skimage.filters.rank import mean

noisy_image = img_as_ubyte(data.camera())
hist, hist_centers = histogram(noisy_image)

#Enter the value for smoothing image
value=int(input("enter the appropriate value here: "))
loc_mean = mean(noisy_image, disk(value))

fig, ax = plt.subplots(ncols=2, figsize=(10, 5), sharex=True, sharey=True)

ax[0].imshow(noisy_image, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[0].set_title('Original')

ax[1].imshow(loc_mean, vmin=0, vmax=255, cmap=plt.cm.gray)
ax[1].set_title('Local mean r = the appropriate value')

for a in ax:
    a.axis('off')
plt.tight_layout()
```

Morphological Operations

Erosion

Return greyscale morphological erosion of an image.

Morphological erosion sets a pixel at (i,j) to the minimum over all pixels in the neighborhood centered at (i,j). Erosion shrinks bright regions and enlarges dark regions.

```
# Erosion shrinks bright regions
import numpy as np
from skimage.morphology import square
bright_square = np.array([[0, 0, 0, 0, 0],
                          [0, 1, 1, 1, 0],
                          [0, 1, 1, 1, 0],
                          [0, 1, 1, 1, 0],
                          [0, 0, 0, 0, 0]], dtype=np.uint8)
erosion(bright_square, square(3))
```

Dilation

Return greyscale morphological dilation of an image.

Morphological dilation sets a pixel at (i,j) to the maximum over all pixels in the neighborhood centered at (i,j). Dilation enlarges bright regions and shrinks dark regions.

```
# Dilation enlarges bright regions
import numpy as np
from skimage.morphology import square
bright_pixel = np.array([[0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0],
                         [0, 0, 1, 0, 0],
                         [0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0]], dtype=np.uint8)
dilation(bright_pixel, square(3))
```

Label

```
import numpy as np
x = np.eye(3).astype(int)
print(x)
print(label(x, connectivity=1))
print(label(x, connectivity=2))
print(label(x, background=-1))

x = np.array([[1, 0, 0],
              [1, 1, 5],
              [0, 0, 0]])
print(label(x))
```

Medial Axis

Compute the medial axis transform of a binary image

```
square = np.zeros((7, 7), dtype=np.uint8)
square[1:-1, 2:-2] = 1
square

medial_axis(square).astype(np.uint8)
```

Opening

Return greyscale morphological opening of an image.

The morphological opening on an image is defined as an erosion followed by a dilation. Opening can remove small bright spots (i.e. “salt”) and connect small dark cracks. This tends to “open” up (dark) gaps between (bright) features.

```
# Open up gap between two bright regions (but also shrink regions)
import numpy as np
from skimage.morphology import square
bad_connection = np.array([[1, 0, 0, 0, 1],
                           [1, 1, 0, 1, 1],
                           [1, 1, 1, 1, 1],
                           [1, 1, 0, 1, 1],
                           [1, 0, 0, 0, 1]], dtype=np.uint8)
opening(bad_connection, square(3))
```

Remove small holes

Remove contiguous holes smaller than the specified size.

```
from skimage import morphology
a = np.array([[1, 1, 1, 1, 1, 0],
              [1, 1, 1, 0, 1, 0],
              [1, 0, 0, 1, 1, 0],
              [1, 1, 1, 1, 1, 0]], bool)
b = morphology.remove_small_holes(a, 2)
b

c = morphology.remove_small_holes(a, 2, connectivity=2)
c
d = morphology.remove_small_holes(a, 2, in_place=True)
d is a
```

Remove Small Objects

Remove objects smaller than the specified size.

Expects ar to be an array with labeled objects, and removes objects smaller than min_size. If `ar` is bool, the image is first labeled. This leads to potentially different behavior for bool and 0-and-1 arrays.

```
from skimage import morphology
a = np.array([[0, 0, 0, 1, 0],
              [1, 1, 1, 0, 0],
              [1, 1, 1, 0, 1]], bool)
b = morphology.remove_small_objects(a, 6)
b

c = morphology.remove_small_objects(a, 7, connectivity=2)
c

d = morphology.remove_small_objects(a, 6, in_place=True)
d is a
```

Skeletonize

Compute the skeleton of a binary image.

Thinning is used to reduce each connected component in a binary image to a single-pixel wide skeleton.

```
X, Y = np.ogrid[0:9, 0:9]
ellipse = (1./3 * (X - 4)**2 + (Y - 4)**2 < 3**2).astype(np.uint8)
ellipse


skel = skeletonize(ellipse)
skel.astype(np.uint8)
```

Self-Assessment

	Requirements	Done	Pt	Explanation	Score
1	Load/Read, Saving Images	☒	10		
2	User Interface/ Form Prep.	☒	10		
3	Image Filtering Process	☒	10		
4	Histogram Display and Threshold	☒	10		
5	(Resíng, Rotation, Cropping, Swirl…) including at least 5 diff processes	☒	10		
6	Exposure rescale intensity	☒	10		
7	Morphological Processes	☒	10		
8	Video Processing	☒	10		
9	Report: requirements, details, specification…	☒	20		
Total points out of 100:					100

Source codes

We also have written some pure python codes regarding to some image processing techniques:

```
# The program implements various image processing techniques

# import dependencies
import ctypes
import os
from tkinter import *
from tkinter import filedialog, ttk
import cv2
from matplotlib import image as mpimg
from matplotlib import pyplot as plt
import numpy as np
from PIL import Image, ImageTk
import scipy.signal

# Use with windows envs only
#ctypes.windll.shcore.SetProcessDpiAwareness(True)

# create root window
root = Tk()
ttk.Style().configure("TButton", justify=CENTER)

# Global variables
width_gui = 1366
height_gui = 720
input_file = ""
output_file = ""
loaded_image = None
edited_image = None
arg_from_usr = None
popup_dialog = None
popup_input_dialog = None


root.title("KMAM Image Processor")
# set minimum size of gui
root.minsize(width_gui, height_gui)


# get user input for some functions
def set_user_arg():
    global arg_from_usr
    arg_from_usr = popup_input_dialog.get()
    popup_dialog.destroy()
    popup_dialog.quit()

#
def open_popup_input(text):
    global popup_dialog, popup_input_dialog
    popup_dialog = Toplevel(root)
    popup_dialog.resizable(False, False)
    popup_dialog.title("User Input")
    text_label = ttk.Label(popup_dialog, text=text, justify=LEFT)
    text_label.pack(side=TOP, anchor=W, padx=15, pady=10)
```

```
    popup_input_dialog = ttk.Entry(popup_dialog)
    popup_input_dialog.pack(side=TOP, anchor=NW, fill=X, padx=15)
    popup_btn = ttk.Button(popup_dialog, text="OK",
command=set_user_arg).pack(pady=10)
    popup_dialog.geometry(f"400x{104 + text_label.winfo_reqheight()}")
    popup_input_dialog.focus()
    popup_dialog.mainloop()


def draw_before_canvas():
    global loaded_image, input_file
    loaded_image = Image.open(input_file)
    loaded_image = loaded_image.convert("RGB")
    img = ImageTk.PhotoImage(loaded_image)
    before_canvas.create_image(
        256,
        256,
        image=img,
        anchor="center",
    )
    before_canvas.img = img


def draw_after_canvas(mimg):
    global edited_image

    edited_image = Image.fromarray(np.uint8(mimg))
    img = ImageTk.PhotoImage(edited_image)
    after_canvas.create_image(
        256,
        256,
        image=img,
        anchor="center",
    )
    after_canvas.img = img


def load_file():
    global input_file
    input_file = filedialog.askopenfilename(
        title="Open an image file",
        initialdir=".",
        filetypes=[("All Image Files", "*.*")],
    )
    draw_before_canvas()
    # print(f"Image loaded from: {input_file}")


def save_file():
    global input_file, loaded_image, edited_image
    file_ext = os.path.splitext(input_file)[1][1:]
    op_file = filedialog.asksaveasfilename(
        filetypes=[
            (
                f"{file_ext.upper()}",
                f"*.{file_ext}",
```

```
            )
        ],
        defaultextension=[
            (
                f"{file_ext.upper()}",
                f"*.{file_ext}",
            )
        ],
    )
    edited_image = edited_image.convert("RGB")
    edited_image.save(op_file)
    # print(f"Image saved at: {output_file}")


# frames
left_frame = ttk.LabelFrame(root, text="Original Image", labelanchor=N)
left_frame.pack(fill=BOTH, side=LEFT, padx=10, pady=10, expand=1)

middle_frame = ttk.LabelFrame(root, text="Algorithms", labelanchor=N)
middle_frame.pack(fill=BOTH, side=LEFT, padx=5, pady=10)

right_frame = ttk.LabelFrame(root, text="Modified Image", labelanchor=N)
right_frame.pack(fill=BOTH, side=LEFT, padx=10, pady=10, expand=1)

# left frame contents
before_canvas = Canvas(left_frame, bg="white", width=512, height=512)
before_canvas.pack(expand=1)

browse_btn = ttk.Button(left_frame, text="Browse", command=load_file)
browse_btn.pack(expand=1, anchor=SW, pady=(5, 0))

# middle frame contents
algo_canvas = Canvas(middle_frame, width=260, highlightthickness=0)
scrollable_algo_frame = Frame(algo_canvas)
scrollbar = Scrollbar(
    middle_frame, orient="vertical", command=algo_canvas.yview, width=15
)
scrollbar.pack(side="right", fill="y")
algo_canvas.pack(fill=BOTH, expand=1)
algo_canvas.configure(yscrollcommand=scrollbar.set)
algo_canvas.create_window((0, 0), window=scrollable_algo_frame, anchor="nw")
scrollable_algo_frame.bind(
    "<Configure>", lambda _:
algo_canvas.configure(scrollregion=algo_canvas.bbox("all"))
)


# right frame contents
after_canvas = Canvas(right_frame, bg="white", width=512, height=512)
after_canvas.pack(expand=1)

save_btn = ttk.Button(right_frame, text="Save", command=save_file)
save_btn.pack(expand=1, anchor=SE, pady=(5, 0))


# algorithm fns
```

```
def RGB2Gray():
    img = mpimg.imread(input_file)
    R, G, B = img[:, :, 0], img[:, :, 1], img[:, :, 2]
    return 0.299 * R + 0.58 * G + 0.114 * B


def callRGB2Gray():
    grayscale = RGB2Gray()
    draw_after_canvas(grayscale)


def negative(set_gray):
    img = RGB2Gray() if (set_gray) else Image.open(input_file)
    img = np.array(img)
    img = 255 - img
    draw_after_canvas(img)


def gray_slice(img, lower_limit, upper_limit, fn):
    # general function
    if lower_limit <= img <= upper_limit:
        return 255
    else:
        return fn


def call_gray_slice(retain):
    img = RGB2Gray()
    # input 100,180
    open_popup_input("Enter lower limit, upper limit\n(Separate inputs with a
comma)")
    arg_list = arg_from_usr.replace(" ", "").split(",")
    print(arg_list)
    lower_limit = int(arg_list[0])
    upper_limit = int(arg_list[1])
    img_thresh = np.vectorize(gray_slice)
    fn = img if retain else 0
    draw_after_canvas(img_thresh(img, lower_limit, upper_limit, fn))


def bit_slice(img, k):
    # create an image for the k bit plane
    plane = np.full((img.shape[0], img.shape[1]), 2 ** k, np.uint8)
    # execute bitwise and operation
    res = cv2.bitwise_and(plane, img)
    # multiply ones (bit plane sliced) with 255 just for better visualization
    return res * 255


def call_bit_slice():
    global arg_from_usr
    bitplanes = []
    img = cv2.imread(input_file, 0)
    open_popup_input(
        "Enter bit plane no k (0-7)\n(or leave it blank to display all 8
planes together)"
```

```
    )
    if not arg_from_usr:
        for k in range(9):
            slice = bit_slice(img, k)
            # append to the output list
            slice = cv2.resize(slice, (171, 171))
            bitplanes.append(slice)

        # concat all 8 bit planes into one image
        row1 = cv2.hconcat([bitplanes[0], bitplanes[1], bitplanes[2]])
        row2 = cv2.hconcat([bitplanes[3], bitplanes[4], bitplanes[5]])
        row3 = cv2.hconcat([bitplanes[6], bitplanes[7], bitplanes[8]])
        final_img = cv2.vconcat([row1, row2, row3])
    else:
        final_img = bit_slice(img, int(arg_from_usr))

    draw_after_canvas(final_img)


def c_stretch(img, r1, r2, s1, s2):
    # general function
    if img < r1:
        return s1
    elif img > r2:
        return s2
    else:
        return s1 + ((s2 - s1) * (img - r1) / (r2 - r1))


def call_c_stretch(limited):
    # input
    img = RGB2Gray()
    r1 = np.min(img)
    r2 = np.max(img)
    if limited:
        # input 25,220
        open_popup_input("Enter s1,s2\n(Separate inputs with a comma)")
        arg_list = arg_from_usr.replace(" ", "").split(",")
        s1, s2 = int(arg_list[0]), int(arg_list[1])
    else:
        s1, s2 = (0, 255)
    image_cs = np.vectorize(c_stretch)
    draw_after_canvas(image_cs(img, r1, r2, s1, s2))


def plot_histogram(label, img, index):
    hist, bins = np.histogram(img, 256, [0, 256])
    cdf = hist.cumsum()
    cdf_normalized = cdf * float(hist.max()) / cdf.max()
    plt.subplot(1, 2, index)
    plt.title(label)
    plt.plot(cdf_normalized, color="b")
    plt.hist(img.flatten(), 256, [0, 256], color="r")
    plt.xlim([0, 256])
    plt.legend(("cdf", "histogram"), loc="upper left")
    plt.xlabel("Pixel intensity")
```

```
    plt.ylabel("Distirbution")
    plt.tight_layout()


def histogram_eq():
    plt.figure(num=1, figsize=(11, 5), dpi=100)
    img = cv2.imread(input_file, 0)
    plot_histogram("Original Histogram", img, 1)
    equ_img = cv2.equalizeHist(img)
    plot_histogram("Equalized Histogram", equ_img, 2)
    draw_after_canvas(equ_img)
    plt.show()


def correlate(image, filter):
    filtered_image = image
    for i in range(image.shape[-1]):
        filtered_image[:, :, i] = scipy.signal.correlate2d(
            image[:, :, i], filter, mode="same", boundary="symm"  # extended
padding
        )
    filtered_image = filtered_image[:, :, ::-1]  # converts BGR to RGB
    return filtered_image


def box_filter():
    global arg_from_usr
    open_popup_input("Enter n for (nxn) filter")
    arg_from_usr = int(arg_from_usr)
    filter = np.ones([arg_from_usr, arg_from_usr], dtype=int)
    filter = filter / (arg_from_usr ** 2)
    image = cv2.imread(input_file)
    filtered_image = correlate(image, filter)
    draw_after_canvas(filtered_image)


def wt_avg_filter():
    filter = [
        [1 / 16, 2 / 16, 1 / 16],
        [2 / 16, 4 / 16, 2 / 16],
        [1 / 16, 2 / 16, 1 / 16],
    ]
    image = cv2.imread(input_file)
    filtered_image = correlate(image, filter)
    draw_after_canvas(filtered_image)


# algorithm btns
ttk.Button(
    scrollable_algo_frame, text="RGB to Grayscale", width=30,
command=callRGB2Gray
).pack(expand=1, padx=5, pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Negative",
```

```
    width=30,
    command=lambda: negative(set_gray=False),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Negative\n(Grayscale output)",
    width=30,
    command=lambda: negative(set_gray=True),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Gray level slicing\n(retaining background)",
    width=30,
    command=lambda: call_gray_slice(retain=True),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Gray level slicing\n(lowering background)",
    width=30,
    command=lambda: call_gray_slice(retain=False),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Bit plane slicing",
    width=30,
    command=call_bit_slice,
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Contrast Stretching\n(Linear)",
    width=30,
    command=lambda: call_c_stretch(limited=False),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Contrast Stretching\n(Limited Linear)",
    width=30,
    command=lambda: call_c_stretch(limited=True),
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Histogram Equalization",
    width=30,
    command=histogram_eq,
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Image Smoothing\n(nxn Avg/Box Filter)",
```

```
    width=30,
    command=box_filter,
).pack(pady=2, ipady=2)

ttk.Button(
    scrollable_algo_frame,
    text="Image Smoothing\n(3x3 Weighted Avg Filter)",
    width=30,
    command=wt_avg_filter,
).pack(pady=2, ipady=2)

# ttk.Button(
#     scrollable_algo_frame,
#     text="Image Smoothing\n(3x3 Median Filter)",
#     width=30,
#     command=wt_avg_filter,
# ).pack(pady=2, ipady=2)

# ttk.Button(
#     scrollable_algo_frame,
#     text="Image Smoothing\n(3x3 Weighted Median Filter)",
#     width=30,
#     command=wt_avg_filter,
# ).pack(pady=2, ipady=2)

root.mainloop()
```