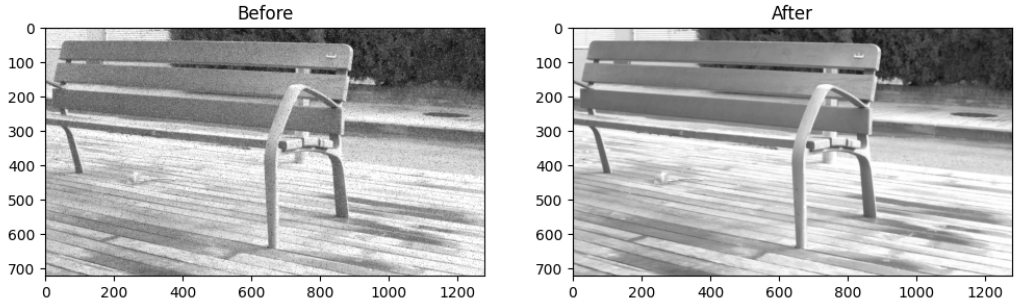
INT3404E 20 - Image Processing: Homeworks 2

Nguyen Ngoc Tuan - 21020237

1 Ex1.py



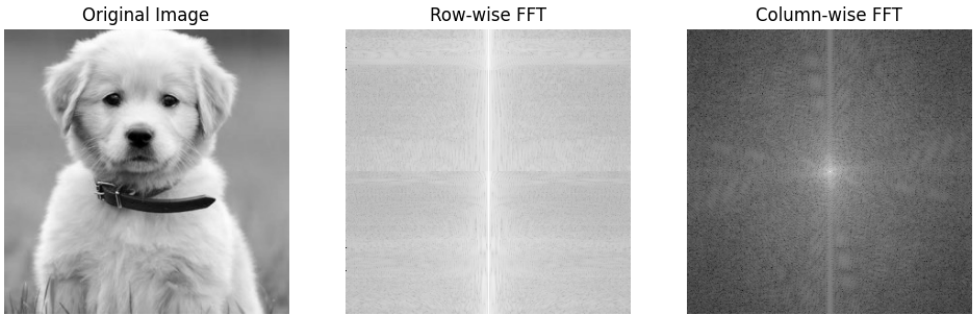

```python
import cv2
import matplotlib.pyplot as plt
import numpy as np
import math
import os


def read_img(img_path):
    """
        Read grayscale image
        Inputs:
        img_path: str: image path
        Returns:
```

```
        img: cv2 image
    """
    return cv2.imread(img_path, 0)


def padding_img(img, filter_size=3):
    """
    The surrogate function for the filter functions.
    The goal of the function: replicate padding the image such that when applying the kerne
    WARNING: Do not use the exterior functions from available libraries such as OpenCV, sci
    Inputs:
        img: cv2 image: original image
        filter_size: int: size of square filter
    Return:
        padded_img: cv2 image: the padding image
    """
  # Need to implement here
    height, width = img.shape[:2]
    pad_width = filter_size // 2
    padded_img = np.pad(img, pad_width, mode='edge')
    return padded_img

def mean_filter(img, filter_size=3):
    """
    Smoothing image with mean square filter with the size of filter_size. Use replicate pad
    WARNING: Do not use the exterior functions from available libraries such as OpenCV, sci
    Inputs:
        img: cv2 image: original image
        filter_size: int: size of square filter,
    Return:
        smoothed_img: cv2 image: the smoothed image with mean filter.
    """
  # Need to implement here
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            neighborhood = padded_img[i:i+filter_size, j:j+filter_size]
            mean_value = np.mean(neighborhood)
            smoothed_img[i, j] = mean_value

    return smoothed_img.astype(np.uint8)


def median_filter(img, filter_size=3):
    """
    Smoothing image with median square filter with the size of filter_size. Use replicate p
    WARNING: Do not use the exterior functions from available libraries such as OpenCV, sci
    Inputs:
```

```
        img: numpy array: original image
        filter_size: int: size of square filter
    Return:
        smoothed_img: numpy array: the smoothed image with median filter.
    """
    # Need to implement here
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img, dtype=np.uint8)

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            neighborhood = padded_img[i:i+filter_size, j:j+filter_size]
            median_value = np.median(neighborhood)
            smoothed_img[i, j] = median_value

    return smoothed_img


def psnr(gt_img, smooth_img):
    """
        Calculate the PSNR metric
        Inputs:
            gt_img: cv2 image: groundtruth image
            smooth_img: cv2 image: smoothed image
        Outputs:
            psnr_score: PSNR score
    """
    # Need to implement here
    assert gt_img.shape == smooth_img.shape, "Input images must have the same dimensions"
    assert gt_img.dtype == smooth_img.dtype, "Input images must have the same data type"
    mse = np.mean((gt_img - smooth_img) ** 2)
    max_pixel_value = np.iinfo(gt_img.dtype).max
    psnr_score = 20 * np.log10(max_pixel_value) - 10 * np.log10(mse)

    return psnr_score


def show_res(before_img, after_img):
    """
        Show the original image and the corresponding smooth image
        Inputs:
            before_img: cv2: image before smoothing
            after_img: cv2: corresponding smoothed image
        Return:
            None
    """
    plt.figure(figsize=(12, 9))
    plt.subplot(1, 2, 1)
    plt.imshow(before_img, cmap='gray')
```

```
    plt.title('Before')

    plt.subplot(1, 2, 2)
    plt.imshow(after_img, cmap='gray')
    plt.title('After')
    plt.show()


if __name__ == '__main__':
    img_noise = "./ex1_images/noise.png" # <- need to specify the path to the noise image
    img_gt = "./ex1_images/ori_img.png"  # <- need to specify the path to the gt image
    img = read_img(img_noise)
    filter_size = 3

    # Mean filter
    mean_smoothed_img = mean_filter(img, filter_size)
    show_res(img, mean_smoothed_img)
    print('PSNR score of mean filter: ', psnr(img, mean_smoothed_img))

    # Median filter
    median_smoothed_img = median_filter(img, filter_size)
    show_res(img, median_smoothed_img)
    print('PSNR score of median filter: ', psnr(img, median_smoothed_img))
```

2 Ex2.py



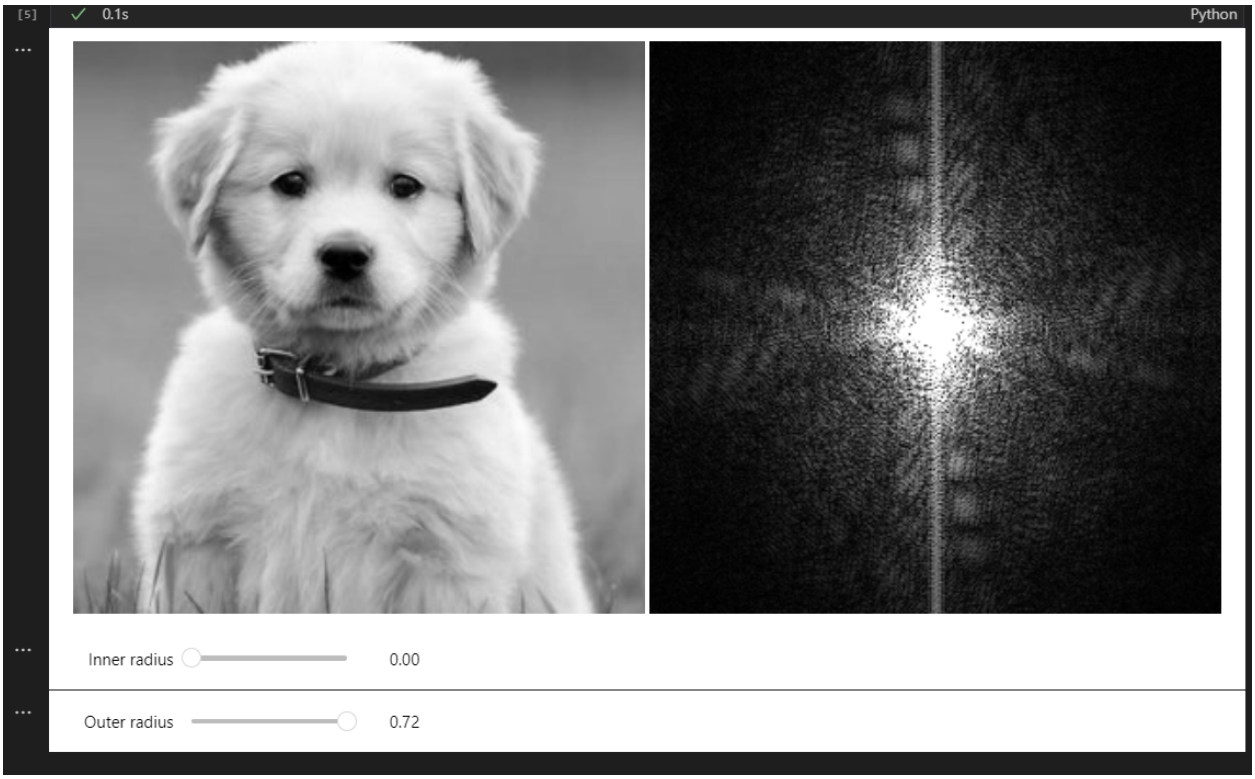

```python
import numpy as np
from skimage import io as io_url
import matplotlib.pyplot as plt


def DFT_slow(data):
    """
    Implement the discrete Fourier Transform for a 1D signal
    params:
        data: Nx1: (N, ): 1D numpy array
    returns:
        DFT: Nx1: 1D numpy array
    """
    N = len(data)
    n = np.arange(N)
    k = n.reshape((N, 1))
    exp_term = np.exp(-2j * np.pi * k * n / N)
    return np.dot(exp_term, data)


def show_img(origin, row_fft, row_col_fft):
```

```
    """
    Show the original image, row-wise FFT, and column-wise FFT

    params:
        origin: (H, W): 2D numpy array
        row_fft: (H, W): 2D numpy array
        row_col_fft: (H, W): 2D numpy array
    """
    fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(12, 8))
    axs[0].imshow(origin, cmap='gray')
    axs[0].set_title('Original Image')
    axs[0].axis('off')
    axs[1].imshow(np.log(np.abs(np.fft.fftshift(row_fft))), cmap='gray')
    axs[1].set_title('Row-wise FFT')
    axs[1].axis('off')
    axs[2].imshow((np.log(np.abs(np.fft.fftshift(row_col_fft)))), cmap='gray')
    axs[2].set_title('Column-wise FFT')
    axs[2].axis('off')
    plt.show()


def DFT_2D(gray_img):
    """
    Implement the 2D Discrete Fourier Transform
    Note that: dtype of the output should be complex_
    params:
        gray_img: (H, W): 2D numpy array

    returns:
        row_fft: (H, W): 2D numpy array that contains the row-wise FFT of the input image
        row_col_fft: (H, W): 2D numpy array that contains the column-wise FFT of the input
    """
    H, W = gray_img.shape
    row_fft = np.zeros_like(gray_img, dtype=np.complex_)
    row_col_fft = np.zeros_like(gray_img, dtype=np.complex_)

    for i in range(H):
        row_fft[i, :] = DFT_slow(gray_img[i, :])

    for j in range(W):
        row_col_fft[:, j] = DFT_slow(row_fft[:, j])

    return row_fft, row_col_fft


if __name__ == '__main__':
    # Load the image
    img = io_url.imread('https://img2.zergnet.com/2309662_300.jpg')
    gray_img = np.mean(img, -1)  # Convert to grayscale
```

```
71
72      # Compute the 2D DFT
73      row_fft, row_col_fft = DFT_2D(gray_img)
74
75      # Display the results
76      show_img(gray_img, row_fft, row_col_fft)
77  import numpy as np
78  from skimage import io as io_url
79  import matplotlib.pyplot as plt
80
81
82  def DFT_slow(data):
83      """
84      Implement the discrete Fourier Transform for a 1D signal
85      params:
86          data: Nx1: (N, ): 1D numpy array
87      returns:
88          DFT: Nx1: 1D numpy array
89      """
90      N = len(data)
91      n = np.arange(N)
92      k = n.reshape((N, 1))
93      exp_term = np.exp(-2j * np.pi * k * n / N)
94      return np.dot(exp_term, data)
95
96
97  def show_img(origin, row_fft, row_col_fft):
98      """
99      Show the original image, row-wise FFT, and column-wise FFT
100
101     params:
102         origin: (H, W): 2D numpy array
103         row_fft: (H, W): 2D numpy array
104         row_col_fft: (H, W): 2D numpy array
105     """
106     fig, axs = plt.subplots(nrows=1, ncols=3, figsize=(12, 8))
107     axs[0].imshow(origin, cmap='gray')
108     axs[0].set_title('Original Image')
109     axs[0].axis('off')
110     axs[1].imshow(np.log(np.abs(np.fft.fftshift(row_fft))), cmap='gray')
111     axs[1].set_title('Row-wise FFT')
112     axs[1].axis('off')
113     axs[2].imshow((np.log(np.abs(np.fft.fftshift(row_col_fft)))), cmap='gray')
114     axs[2].set_title('Column-wise FFT')
115     axs[2].axis('off')
116     plt.show()
117
118
119 def DFT_2D(gray_img):
```

```
    """
    Implement the 2D Discrete Fourier Transform
    Note that: dtype of the output should be complex_
    params:
        gray_img: (H, W): 2D numpy array

    returns:
        row_fft: (H, W): 2D numpy array that contains the row-wise FFT of the input image
        row_col_fft: (H, W): 2D numpy array that contains the column-wise FFT of the input
    """
    H, W = gray_img.shape
    row_fft = np.zeros_like(gray_img, dtype=np.complex_)
    row_col_fft = np.zeros_like(gray_img, dtype=np.complex_)

    for i in range(H):
        row_fft[i, :] = DFT_slow(gray_img[i, :])

    for j in range(W):
        row_col_fft[:, j] = DFT_slow(row_fft[:, j])

    return row_fft, row_col_fft


if __name__ == '__main__':
    # Load the image
    img = io_url.imread('https://img2.zergnet.com/2309662_300.jpg')
    gray_img = np.mean(img, -1)  # Convert to grayscale

    # Compute the 2D DFT
    row_fft, row_col_fft = DFT_2D(gray_img)

    # Display the results
    show_img(gray_img, row_fft, row_col_fft)
```

3 Ex3.ipynb




```
import ipywidgets as widgets
import matplotlib.pyplot as plt
import PIL.Image
import numpy as np
import urllib
from skimage.transform import resize
from matplotlib.image import imread
import os
from IPython.display import display
from skimage import io as io_url
import cv2
import numpy as np
from PIL import Image

# Create image widgets
image3_spatial = widgets.Image(format='png', width=500, height=500, description='Spatial')
image3_freq    = widgets.Image(format='png', width=500, height=500, description='Frequency
sidebyside     = widgets.HBox([image3_spatial, image3_freq])

# Create slider/select widgets
slider_inner   = widgets.FloatSlider(value=0,      min=0, max=1,      step=0.01, descriptio
slider_outer   = widgets.FloatSlider(value=1.44/2, min=0, max=1.44/2, step=0.01, descriptio

```

```
buf            = io.BytesIO()

orig_img = io_url.imread('https://img2.zergnet.com/2309662_300.jpg')
orig_img = np.mean(orig_img, -1)

x = np.fft.fftfreq(orig_img.shape[0]);
y = np.fft.fftfreq(orig_img.shape[1]);

xv, yv = np.meshgrid(x, y)
xv = np.fft.fftshift(xv)
yv = np.fft.fftshift(yv)

def filter_frequency(orig_img, mask):
    """
    You need to remove frequency based on the given mask.
    Params:
      orig_img: numpy image
      mask: same shape with orig_img indicating which frequency hold or remove
    Output:
      f_img: frequency image after applying mask
      img: image after applying mask
    """
    # You need to implement this function
    f_img = np.fft.fft2(orig_img)

    f_img_shifted = np.fft.fftshift(f_img)

    f_img_filtered_shifted = f_img_shifted * mask

    f_img_filtered = np.fft.ifftshift(f_img_filtered_shifted)

    img = np.abs(np.fft.ifft2(f_img_filtered))

    return np.abs(f_img_filtered_shifted), img

def on_value_change3(change):
    mask = (np.sqrt(xv**2 + yv**2) < slider_outer.value) & \
           (np.sqrt(xv**2 + yv**2) >= slider_inner.value)
    mask = np.float32(mask)

    fimg, img = filter_frequency(orig_img, mask)
    buf.seek(0)
    tmp = PIL.Image.fromarray(255*img/(img.max()+0.0001))
    tmp = tmp.convert('L')
    tmp.save(buf, 'png')
    image3_spatial.value = buf.getvalue()

    buf.seek(0)
    tmp = PIL.Image.fromarray(255*np.log(0.0001*fimg + 1))
```

```
    tmp = tmp.convert('L')
    tmp.save(buf, 'png')
    image3_freq.value = buf.getvalue()


slider_inner.observe(on_value_change3, names='value')
slider_outer.observe(on_value_change3, names='value')

on_value_change3(0)

display(sidebyside)
display(slider_inner)
display(slider_outer)
```



```
def read_img(img_path, img_size=(512, 512)):
  """
    +   c     nh
    + C h u y n  t h  n h  grayscale
    + Thay   i   k ch  t h   c   nh   th nh img_size
  """
  img = cv2.imread(img_path, 0)
  img = cv2.resize(img, img_size)
  return img


def create_hybrid_img(img1, img2, r):
  """
  Create hydrid image
  Params:
    img1: numpy image 1
    img2: numpy image 2
    r: radius that defines the filled circle of frequency of image 1. Refer to the homeworl
  """
  f_img1 = np.fft.fftshift(np.fft.fft2(img1))
  f_img2 = np.fft.fftshift(np.fft.fft2(img2))

```

```
    rows, cols = img1.shape[:2]
    crow, ccol = rows // 2, cols // 2
    mask = np.zeros((rows, cols), dtype=np.uint8)
    mask[crow-r:crow+r, ccol-r:ccol+r] = 1

    f_img1_filtered = f_img1 * mask
    f_img2_filtered = f_img2 * (1 - mask)

    f_hybrid = f_img1_filtered + f_img2_filtered

    hybrid_img = np.abs(np.fft.ifft2(np.fft.ifftshift(f_hybrid)))

    hybrid_img = np.clip(hybrid_img, 0, 255).astype(np.uint8)

    return hybrid_img

    image_1_path = "./ex1_images/girl.png" # <-- need to change
    image_2_path = "./ex1_images/dog.png" # <-- need to change
    img_1 = read_img(image_1_path)
    img_2 = read_img(image_2_path)
    hybrid_img = create_hybrid_img(img_2, img_1, 14)
    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 15))
    axes[0].imshow(img_1, cmap="gray")
    axes[0].axis("off")
    axes[1].imshow(img_2, cmap="gray")
    axes[1].axis("off")
    axes[2].imshow(hybrid_img, cmap="gray")
    axes[2].axis("off")
```