INT3404E 20 - Image Processing: Homeworks 2

Cao Trung Hieu

1 Exercise 1

1.1 Padding image

```python
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 kernel with the size of filter_size, the padded image will be the same size
    as the original image.
    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
    return np.pad(np.array(img), path_width=filter_size//2, mode="edge")
```

The above function uses the np.pad function from the numpy library to add padding tho the image:

- **pad_width** parameter determines the thickness of the padding.
- **mode = 'edge'** parameter means that the padding will replicate the edge values of the image.

1.2 Mean filter

```python
def mean_filter(img, filter_size=3):
    """
    Smoothing image with mean square filter with the size of filter_size.
    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
    smoothed_img = np.zeros_like(img)
    img = padding_img(img, filter_size=filter_size)
    w, h = smoothed_img.shape

    for i in range(w):
        for j in range(h):
            smoothed_img[i,j] = np.mean(img[i: i + filter_size, j: j + filter_size])

    return smoothed_img.astype(np.douple)
```

The function **mean_filter** is used to smooth an image using a mean filter. There are four main steps to smooth an image:

1. Creating a new image of the same size as the origin image.
2. It pads the image using the above **padding_img** function.

3. The function then iterates over the pixels of the new image and extracts a sub-matrix of the same size as the filter for each corresponding pixel on padded image. It calculates the mean of the pixel values in the sub-matrix using **np.mean()** and sets this value to the new image.

4. Finally, the function converts the new image tho double format and returns it.

Figure 1 show the result of using **mean_filter** function to smooth a noise image.

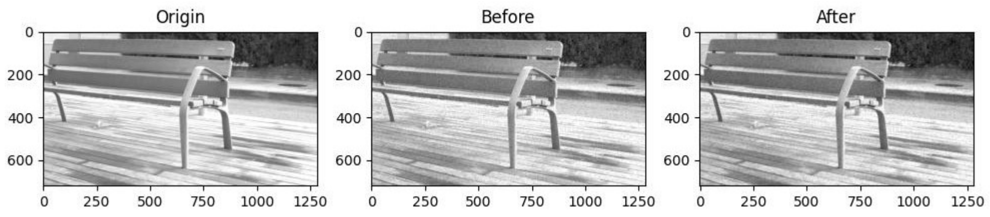

Figure 1: Mean filter

1.3 Median filter

```python
def median_filter(img, filter_size=3):
    """
        Smoothing image with median square filter with the size of filter_size.
        Use replicate padding for the image.
        Inputs:
            img: cv2 image: original image
            filter_size: int: size of square filter
        Return:
            smoothed_img: cv2 image: the smoothed image with median filter.
    """
    # Need to implement here
    smoothed_img = np.zeros_like(img)
    img = padding_img(img, filter_size=filter_size)
    w, h = smoothed_img.shape

    for i in range(w):
        for j in range(h):
            sub_img = img[i: i + filter_size, j: j + filter_size]
            smoothed_img[i,j] = np.median(sub_img)

    return smoothed_img.astype(np.douple)
```

The function **median_filter** is used to smooth an image using a median filter. There are four main steps to smooth an image:

1. Creating a new image of the same size as the origin image.

2. It pads the image using the above **padding_img** function.

3. The function then iterates over the pixels of the new image and extracts a sub-matrix of the same size as the filter for each corresponding pixel on padded image. It calculates the median of the pixel values in the sub-matrix using **np.median()** and sets this value to the new image.

4. Finally, the function converts the new image tho double format and returns it.

Figure 2 show the result of using **median_filter** function to smooth a noise image.

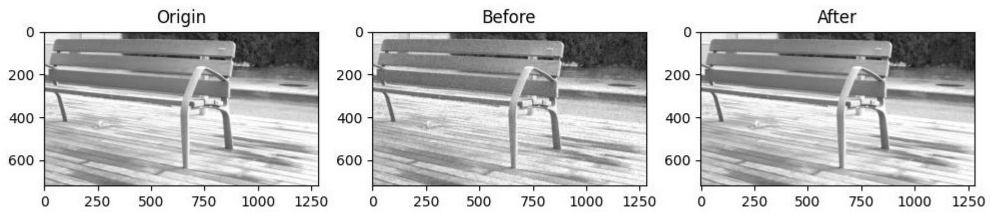

Figure 2: Median filter

1.4 Calculating PSNR

The PSNR block computes the peak signal-to-noise ratio, in decibels, between two images. This ratio is used as a quality measurement between the original and a compressed image. The higher the PSNR, the better the quality of the compressed, or reconstructed image.
The mean-square error (MSE) and the peak signal-to-noise ratio (PSNR) are used to compare image compression quality. The MSE represents the cumulative squared error between the compressed and the original image, whereas PSNR represents a measure of the peak error. The lower the value of MSE, the lower the error.
To compute the PSNR, the block first calculates the mean-squared error using the following equation:

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(O(i,j) - D(i,j))^2 \tag{1}$$

Where, O represents the matrix data of original image. D represents the matrix data of degraded image. m represents the numbers of rows of pixels and i represents the index of that row of the image. n represents the number of columns of pixels and j represents the index of that column of the image.
In the previous equation, M and N are the number of rows and columns in the input images. Then the block computes the PSNR using the following equation:

$$PSRN = 10\log_{10}(\frac{R^2}{MSE}) \tag{2}$$

In the previous equation, R is the maximum fluctuation in the input image data type. For example, if the input image has a double-precision floating-point data type, then R is 1. If it has an 8-bit unsigned integer data type, R is 255, etc.
The following function provides a method to calculate PSRN between two images.

```python
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
    MSE =  np.mean((gt_img - smooth_img)**2)
    if MSE == 0:
        return 100

    MAX = 255.0

    return 20 * math.log10(MAX / math.sqrt(MSE))
```

Using mean filter in figure 1 resulted in a PSNR score of 26.202.
Using mean filter in figure 2 resulted in a PSNR score of 36.977.
The above results can be replicated by running the ex1.python file in HW2 directory, which are all provided in my github repository.

2 Excercise 2

2.1 DFT_Slow

```python
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
  """
  # You need to implement the DFT here

  return np.fft.fft(np.array(data))
```

This function calculate the DFT of a given 1-D signal based on this formula:

$$F(s) = \frac{1}{N}\sum_{s=0}^{N-1} f[n]e^{-i2\pi sn/N} \tag{3}$$

Therefore, it calculates the DFT in a straightforward but inefficient manner, with a time complexity of $O(N^2)$. For large data sets, a Fast Fourier Transform (FFT) algorithm would typically be used instead, as it can compute the same result in O(N log N) time.
The above **DFT_Slow()** function uses **np.fft.fft()** to calculate calculate the DFT of a given 1-D signal.

2.2 DFT_2D

```python
"""
    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 image
    """
    # You need to implement the DFT here
    H, W = gray_img.shape

    row_fft = np.ones((H, W))
    row_col_fft = np.ones((H, W))

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

    return row_fft, row_col_fft
```

This function calculates the 2D DFT, to achieve this, as stated in hw2' objective, we will solely utilize the above **DFT_Slow()** function, designed for one-dimensional Fourier Transforms. The procedure to simulate a 2D Fourier Transform is as follows:

1. By first performing the DFT on each row of the image, we have the row-wise DFT of the original signal.

2. The function will then transpose the row-wise DFT matrix and perform the DFT on each row of the image (i.e performing the DFT on each column of the resulting matrix). By performing another transpose on that matrix, we will get a 2-D DFT of a 2-D signal.

This process is known as the Row-Column Algorithm for the 2D DFT. The final result can be seen in figure 3

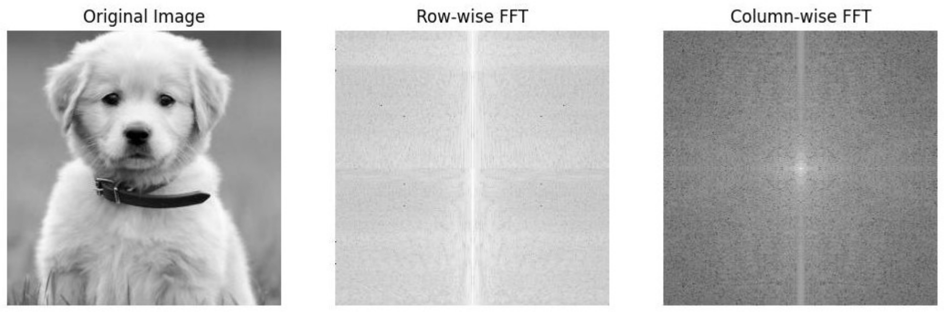

Figure 3: DFT 2D

2.3 Filter frequency

```python
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
    fft_img = np.fft.fft2(orig_img)
    shifted_fft_img = np.fft.fftshift(fft_img)
    filter_fft_img = shifted_fft_img * mask
    filter_shifted_fft_img = np.fft.ifftshift(filter_fft_img)
    img = np.abs(np.fft.ifft2(filter_shifted_fft_img))
    f_img = np.abs(filter_fft_img)
    return f_img, img
```

This function can be used to perform frequency domain filtering on an image, which can be useful for tasks such as noise reduction or feature extraction. The process is as follow:

1. The function takes two inputs: orig_img, which is the original image, and **mask**, which is the frequency filter.
2. Firstly, it computes the 2D Fourier Transform of the original image using the function **np.fft.fft2()**. All the zero-frequency component will then be shifted to the center of the spectrum using **np.fft.fftshift()**.
3. It then multiplies the shifted Fourier Transform by the mask to apply the frequency filter. This operation is performed element-wise. It then shifts the zero-frequency component back to the original place using **np.fft.ifftshift()**.
4. It computes the inverse 2D Fourier Transform of the filtered spectrum using **np.fft.ifft2()**. The result is a complex-valued image, so it takes the absolute value to get a real-valued image.
5. Finally, it returns the absolute value of the filtered Fourier Transform in order to fit with other scope of the notebook (which is a complex-valued spectrum) and the filtered image.

The final result can be seen in figure 4 below.

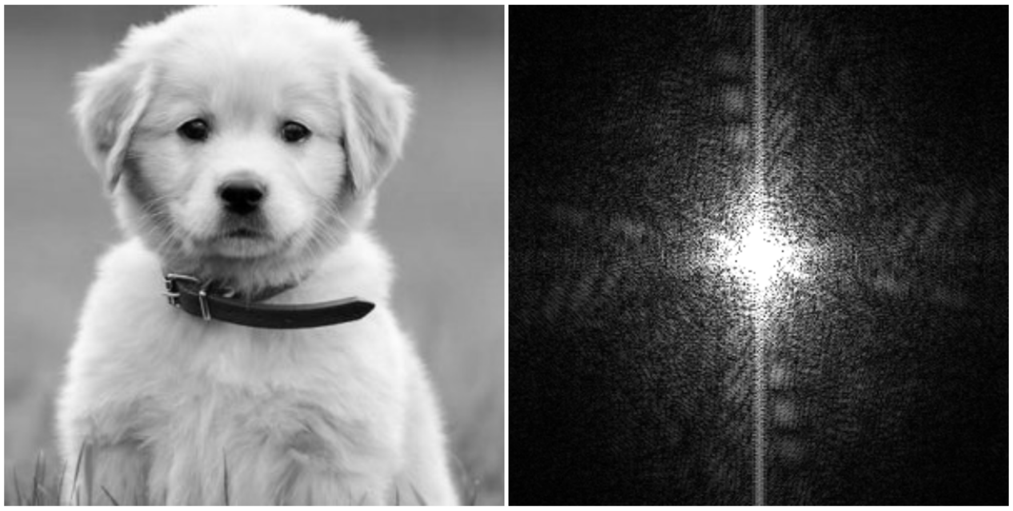

Figure 4: Filter frequency

2.4 Hybrid image

```python
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 homework title to know more.
    """
    # You need to implement the function

    fft_img1 = np.fft.fft2(img1)
    fft_img2 = np.fft.fft2(img2)

    shift_img1 = np.fft.ifftshift(fft_img1)
    shift_img2 = np.fft.ifftshift(fft_img2)

    y, x = np.indices(img1.shape)
    center = np.array(img1.shape) // 2
    distance = ((x - center[0])**2 + (y - center[1])**2)**0.5
    mask = distance <= r

    masked_img = shift_img1 * mask + shift_img2 * ~mask

    shifted_masked_img = np.fft.ifftshift(masked_img)

    hybrid_img = np.abs(np.fft.ifft2(shifted_masked_img))

    return hybrid_img
```

A hybrid image is an image that is perceived in one of two different ways, depending on viewing distance, based on the Fourier components of the image. This function can be used to create hybrid images for visual perception experiments, among other applications. The process is as follow:

1. The function takes two images **img1** and **img2**, and a radius **r** as inputs.
2. It computes the 2D Fourier Transform of both images. All the zero-frequency component will then be

shifted to the center of the spectrum.

3. A circular mask will be created with radius **r**, centered at the middle of the image. The mask is a binary image that is True inside the circle and False outside.

4. The function applies the mask to the shifted Fourier Transform of **img1** and the inverse of the mask to the shifted Fourier Transform of **img2**. This operation is performed element-wise. The result is a hybrid spectrum that contains the low frequencies from **img1** and the high frequencies from **img2**. It then shifts the zero-frequency component back to the original place.

5. It computes the inverse 2D Fourier Transform of the hybrid spectrum. The result is a complex-valued image, so it takes the absolute value to get a real-valued image and returns the hybrid image.

The final result can be seen in figure 5 below.

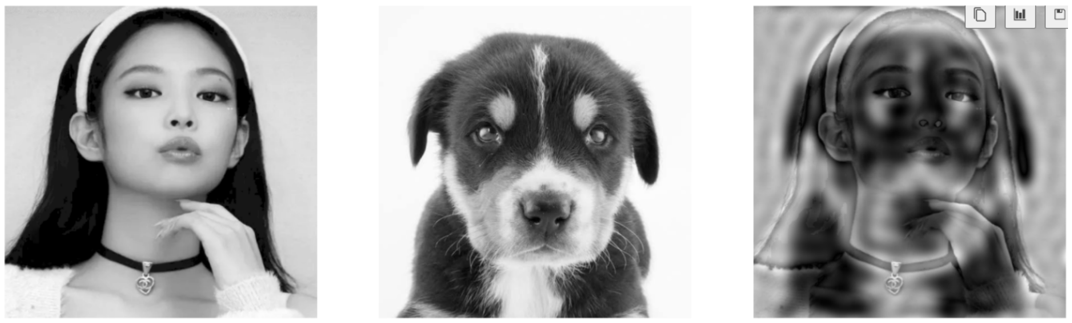

Figure 5: Hybrid image