INT3404E 20 - Image Processing: Homework 2

Nguyen Minh Kien

1 Padding Image:

```python
def padding_img(img, filter_size=3):
    image_array = np.array(img)
    padded_img = np.pad(image_array, pad_width=filter_size // 2, mode = 'edge')
    return padded_img
```

The above function uses the **np.pad** function from the numpy library to add padding to the image:
- The ***pad_ width*** parameter determines the thickness of the padding. As the function requires the padded image to have the same size as the original image, the kernel size (filter size) is divided by 2 to satisfy this requirements.
- The ***mode = 'edge'*** parameter means that the padding will replicate the edge values of the image.

2 Mean Filter:

```python
def mean_filter(img, filter_size=3):
    img_array = padding_img(img, filter_size)
    h, w = img.shape
    img_new = np.zeros([h, w])
    pad_width = filter_size // 2
    for i in range(pad_width, h + pad_width):
        for j in range(pad_width, w + pad_width):
            temp = img_array[i - pad_width : i + pad_width + 1, j - pad_width : j + pad_width + 1]
            mean = np.mean(temp)
            img_new[i - pad_width, j - pad_width] = mean
    img_new = img_new.astype(np.uint8)
    return img_new
```

The function **mean_filter** is used to smooth an image using a mean filter. Here is how it works:
1. Firstly, it pads the image using the previously mention **padding_img** function.
2. A new image of the same size as the original image will then be created with all pixels set to 0.
3. The function then iterates over the pixels of the padded image, extracting a sub-matrix of the same size as the filter for each pixel. It calculates the mean of the pixel values in the sub-matrix using ***np.mean()*** and sets the corresponding pixel in the new image to this value.
This process has the effect of smoothing the images, as each pixel's value is replaced with the average of the values in its neighborhood.
4. Finally, the function converts the new image to 8-bit unsigned integer format and returns it.

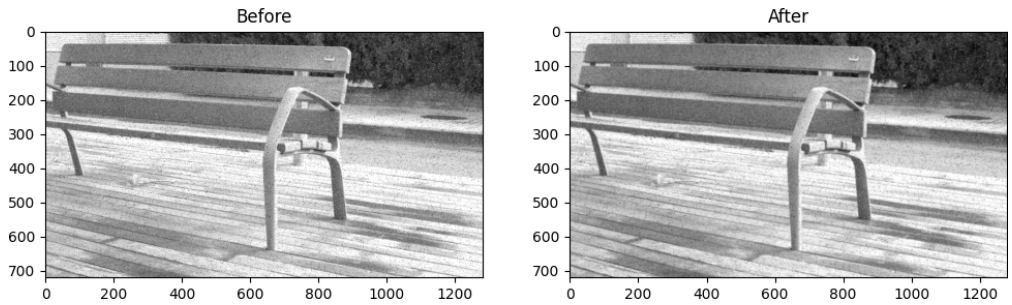

Figure 1: Comparison between before and after apply Mean filter

3 Median Filter:

```
def median_filter(img, filter_size=3):
    img_array = padding_img(img, filter_size)
    h, w = img.shape
    img_new = np.zeros([h, w])
    pad_width = filter_size // 2
    for i in range(pad_width, h + pad_width):
        for j in range(pad_width, w + pad_width):
            temp = img_array[i - pad_width : i + pad_width + 1, j - pad_width : j + pad_width + 1]
            median = np.median(temp)
            img_new[i - pad_width, j - pad_width] = median
    img_new = img_new.astype(np.uint8)
    return img_new
```

The function **median_filter** is used to smooth an image using a median filter. Here is how it works:
1. Firstly, it pads the image using the previously mention **padding_img** function.
2. A new image of the same size as the original image will then be created with all pixels set to 0.
3. The function then iterates over the pixels of the padded image, extracting a sub-matrix of the same size as the filter for each pixel. It calculates the median of the pixel values in the sub-matrix using ***np.median()*** and sets the corresponding pixel in the new image to this value.
This has the effect of smoothing the images, as each pixel's value is replaced with the median of the pixel values in its neighborhood.
4. Finally, the function converts the new image to 8-bit unsigned integer format and returns it.

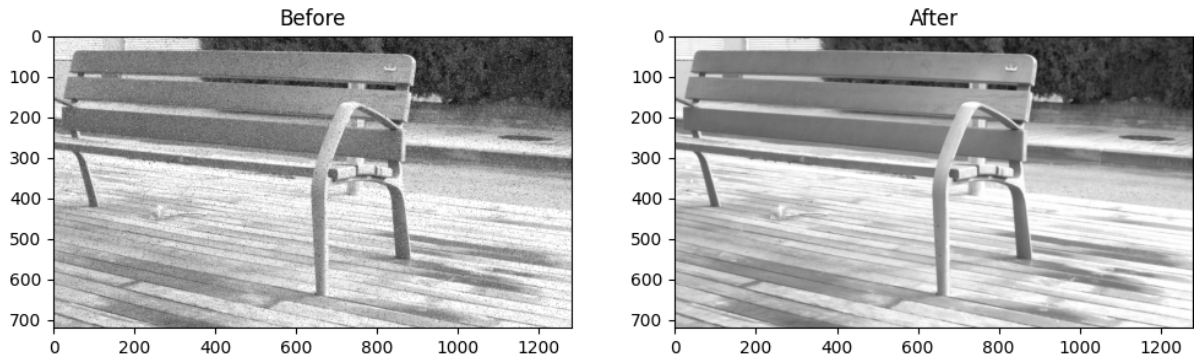

Figure 2: Comparison between before and after apply Median filter

4 PSNR Score:

Peak Signal-to-Noise Ratio (PSNR) is a popular metric for measuring the quality of reconstructed images in the field of image processing. This function can be used to assess the quality of an image after it has been compressed or otherwise processed, compared to the original image. A higher PSNR indicates that the reconstruction is of higher quality. The following function provides a method to calculate PSNR between two images.

```
def psnr(gt_img, smooth_img):
    mse = np.mean((gt_img - smooth_img) ** 2)
    if (mse == 0):
        return 100
    max_pixel = 255.0
    psnr = 20 * math.log10(max_pixel / math.sqrt(mse))
    return psnr
```

Here is how it works:
1. The function first calculates the **Mean Squared Error** (MSE) between the two images. If the MSE is 0 (which means the two images are identical), it returns 100.
2. If MSE in the previous step is not equal to 0, the process will be continued. The function calculate the PSNR using the formula below:

$$PSNR = 20 \times \log_{10}(\frac{max_I}{\sqrt{MSE}})$$

where, $\boldsymbol{max_I}$ is the maximum possible pixel value of the image (255 for an 8-bit grayscale image), and return this value.
Using mean filter in **figure** 1 resulted in a PSNR score of **31.608**.
Using mean filter in **figure** 2 resulted in a PSNR score of **37.119**.
The above results can be replicated by running the ex1.python file in HW2 directory, which are all provided in my github repository.

5 1-D Discrete Fourier Transform:

```
def DFT_slow(data):
    N = len(data)
    res = []
    for i in range (0, N):
      temp = 0
      for k in range (0, N):
          e = np.exp(2j * np.pi * i * k / N)
          temp += data[k] / e
      res.append(temp)
    return np.array(res)
```

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

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

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.

6 2-D Discrete Fourier Transform:

```
def DFT_2D(gray_img):
    src = np.array(gray_img)
    H, W = src.shape
    row_fft, row_col_fft = [], []
    for i in range (0, H):
        temp = np.fft.fft(src[i, 0 : W])
        # temp = DFT_slow(src[i, 0 : W])
        row_fft.append(temp)
    row_fft = np.array(row_fft)
    src2 = row_fft.transpose()
    for i in range (0, W):
        temp = np.fft.fft(src2[i, 0 : H])
        # temp = DFT_slow(src2[i, 0 : H])
        row_col_fft.append(temp)
    row_col_fft = np.array(row_col_fft).transpose()
    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 np.fft.fft 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 below.

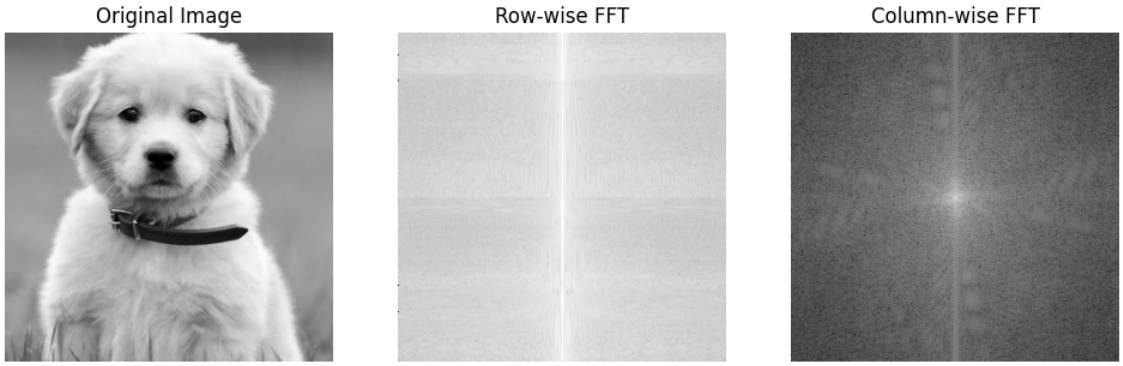

Figure 3: 2D DFT result

7 Filter Frequency:

```
def filter_frequency(orig_img, mask):
    fft_transform = np.fft.fft2(orig_img)
    shifted_fft_transform = np.fft.fftshift(fft_transform)
    filtered_fft_transform = shifted_fft_transform * mask
    filtered_fft_transform_shifted = np.fft.ifftshift(filtered_fft_transform)
    filtered_img = np.abs(np.fft.ifft2(filtered_fft_transform_shifted))
    return np.abs(filtered_fft_transform), filtered_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.

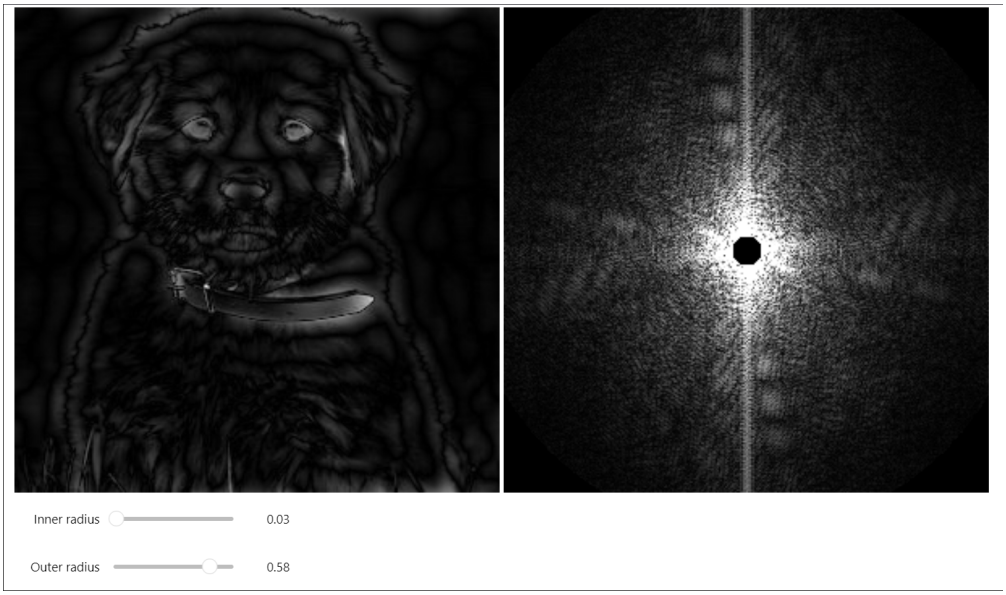

Figure 4: Frequency filter result

8 Create Hybrid Image:

```
def create_hybrid_img(img1, img2, r):
    fft_img1 = np.fft.fft2(img1)
    fft_img2 = np.fft.fft2(img2)
    shifted_img1 = np.fft.fftshift(fft_img1)
    shifted_img2 = np.fft.fftshift(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 = shifted_img1 * mask + shifted_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