INT3404E 20 - Image Processing: Homework 1

Vu Thanh Long

1 Padding Image

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

The function **mean_filter** is used to smooth an image using a mean filter. Here is how it works:
1. The source image is then padded with the previous **padding_img** to produce a **padded_img**.
2. A new image with the same size as the source image is created with all pixels initialized at 0.
3. Each pixel from the padded image is iterated through and grouped into filter-sized sub-matrices, and the mean value of each sub-matrix is recorded in the corresponding pixel of the smoothed image.
4. Finally, the function converts the new image to 8-bit unsigned integer format and returns it.

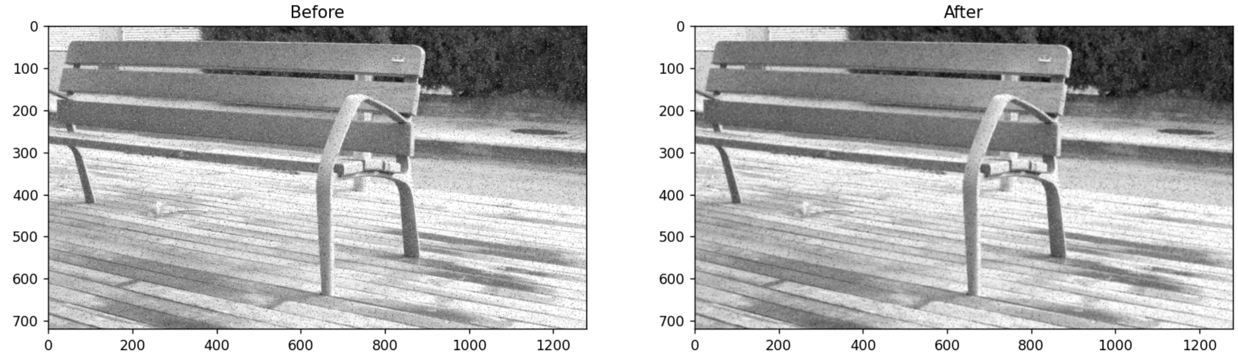


Figure 1: Comparison between original image and mean filtered image

3 Median Filter

```
def median_filter(img, filter_size=3):
    filter_size = filter_size//2
    kernel = np.ones((filter_size, filter_size), dtype = int)
    padded_img = padding_img(img)
    smoothed_img = np.zeros(img.shape)
    h, w = img.shape
    for i in range(filter_size, h + filter_size):
        for j in range(filter_size, w + filter_size):
            temp = padded_img[i - filter_size : i + filter_size + 1, j - filter_size : j + filter_size + 1]
            median = np.median(temp)
            smoothed_img[i - filter_size, j - filter_size] = median
    smoothed_img = smoothed_img.astype(np.uint8)
    return smoothed_img
```

The function **median_filter** is used to smooth an image using a mean filter. Here is how it works:
1. The source image is then padded with the previous **padding_img** to produce a **padded_img**.
2. A new image with the same size as the source image is created with all pixels initialized at 0.
3. Each pixel from the padded image is iterated through and grouped into filter-sized sub-matrices, and the median value of each sub-matrix is recorded in the corresponding pixel of the smoothed image.
4. Finally, the function converts the new image to 8-bit unsigned integer format and returns it.

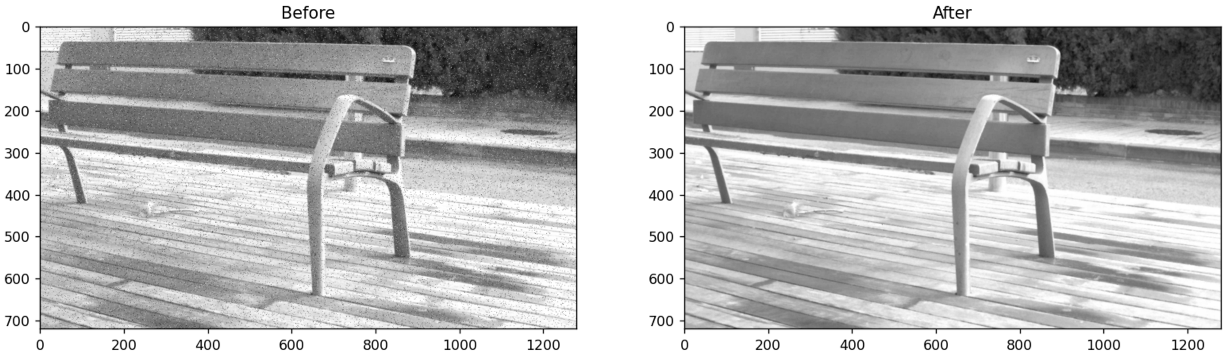


Figure 2: Comparison between original image and median filtered image

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):
    #Calculate MSE
    gt_img = np.array(gt_img)
    smooth_img = np.array(smooth_img)
    mse = np.square(np.subtract(gt_img,smooth_img)).mean()
    if (mse == 0):
        mse = 100
    #Calculate PSNR
    psnr = 10 * math.log10((255**2)/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:

$$PSNR = 10\log_{10}\left(\frac{max_I^2}{MSE}\right) \tag{1}$$

where 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 = data.shape[0]
  DFT = np.zeros(data.shape, dtype=complex)

  for n in range(N):
    for m in range(N):
      DFT[n] += data[m]/(np.exp(2j * np.pi * n * m / N))
    DFT[n] /= N
  return DFT
```

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

$$F(s) = \frac{1}{n}\sum_{s=0}^{N-1} f[n]e^{-i2\pi n/N} \tag{2}$$

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)
    row_fft = []

    h, w = gray_img.shape

    for i in range(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 = np.zeros(gray_img.shape, dtype=complex)
    np.copyto(src2, row_fft.transpose())

    for i in range (w):
      src2[i] = np.fft.fft(src2[i])
    row_col_fft = np.transpose(src2)
    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.

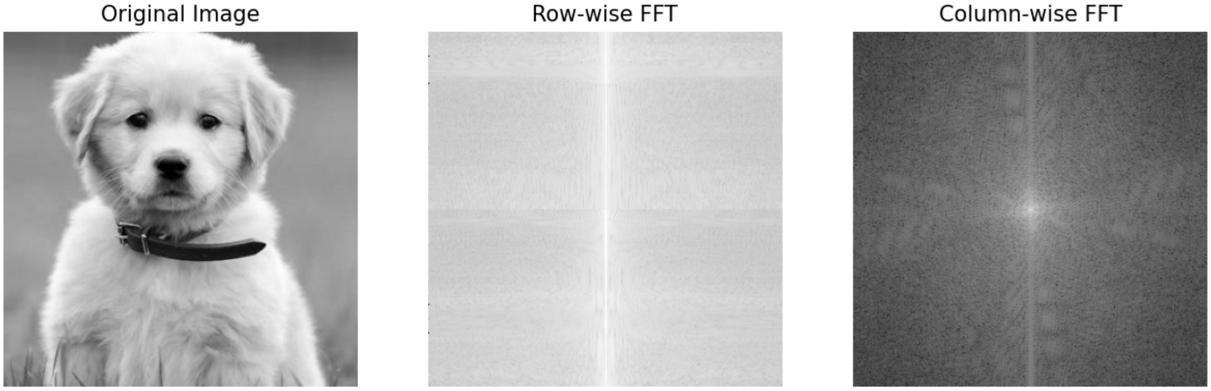


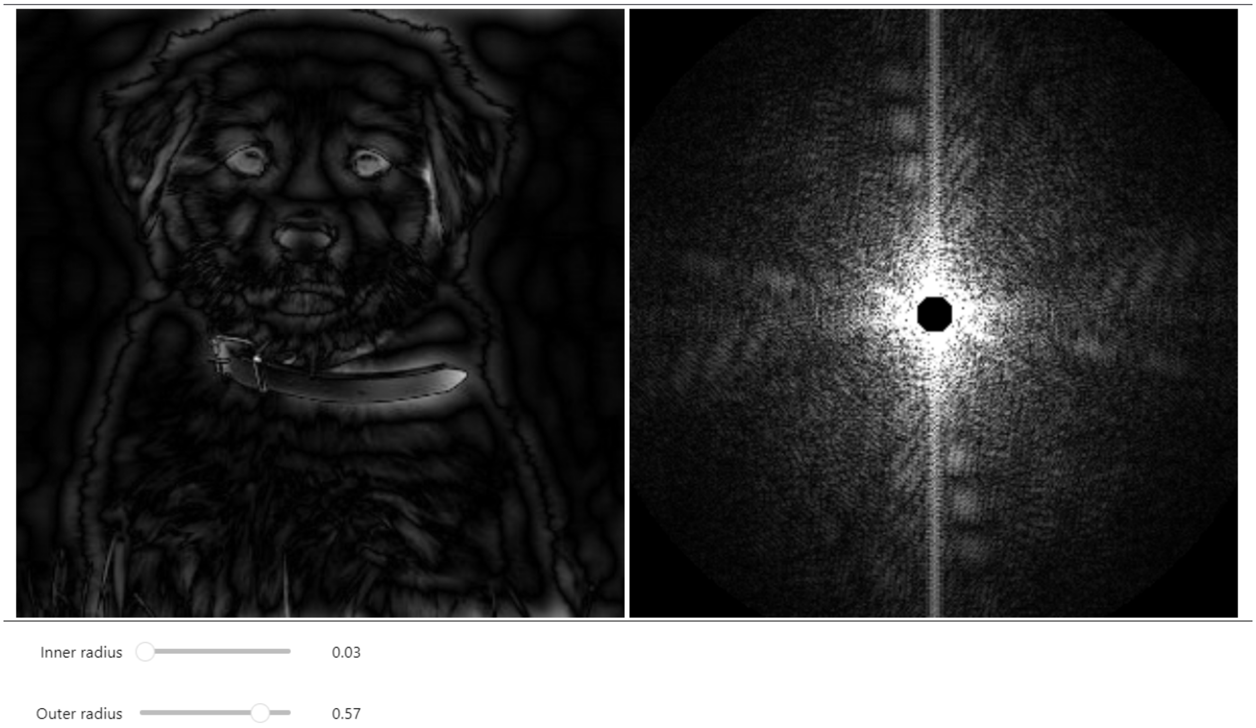
Figure 3: 2D DFT result

7 Filter Frequency

```
def filter_frequency(orig_img, mask):
  #Transform using fft2
  fft_transform = np.fft.fft2(orig_img)
  #Shift frequency coefs to center using fftshift
  fft_shift = np.fft.fftshift(fft_transform)
  #Filter in frequency domain using the given mask
  f_img = fft_shift * mask
  #Shift frequency coefs back using ifftshift
  fft_inverse_transform = np.fft.ifftshift(f_img)
  #invert transform using ifft2
  img = np.fft.ifft2(fft_inverse_transform)
  #Return the modulo of the real values in the image arrays
  return np.abs(f_img), np.abs(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.



Inner radius 0.03

Outer radius 0.57

Figure 4: Frequency filter result

8 Create Hybrid Image

```
def create_hybrid_img(img1, img2, r):
  y, x = np.indices(img_1.shape)
  center = np.array(img_1.shape)/2
  #Transform using fft2
  fft_transform1 = np.fft.fft2(img1)
  fft_transform2= np.fft.fft2(img2)
  #Shift frequency coefs to center using fftshift
  fft_shift1 = np.fft.fftshift(fft_transform1)
  fft_shift2 = np.fft.fftshift(fft_transform2)
  #Create mask with given radius
  mask = np.sqrt((x - center[0])**2 + (y - center[1])**2) <= r
  #Filter the freq domain of the first image with the mask
  #Then filter that of the second image with the inverse of the mask
  f_img1 = fft_shift1 * mask
  f_img2 = fft_shift2 * ~mask
  hybrid_f_img = f_img1 + f_img2
  # #Shift frequency coefs back using ifftshift
  fft_inverse_transform = np.fft.ifftshift(hybrid_f_img)
  # #invert transform using ifft2
  hybrid_img = np.fft.ifft2(fft_inverse_transform)
  #Return the modulo of the real values in the image arrays
  return np.abs(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.

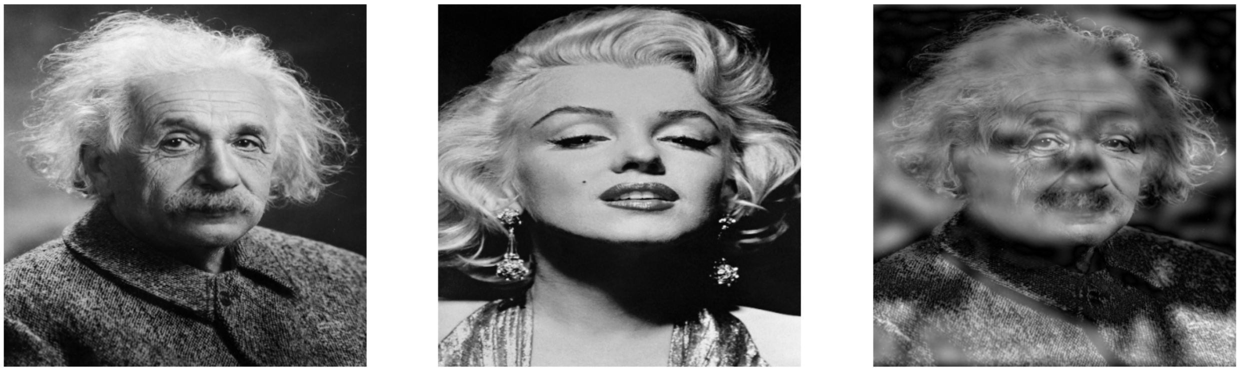


Figure 5: Hybrid Image