INT3404E 20 - Image Processing: Homework 2

Vu Nguyet Hang 22028079

April 2024

1 Image Filtering Functions

1.1 Implementation

Replicate padding

Replicate padding adds the closest values outside the boundary, ensuring that values outside the boundary are set equal to the nearest image border value. In this case, we apply replicate padding with a size equals to $filter\_size$ to the input array img by using numpy function $pad()$ and specify the mode as 'edge':

```
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
    """
    pad_size = filter_size // 2
    return np.pad(img, pad_size, mode='edge')
```

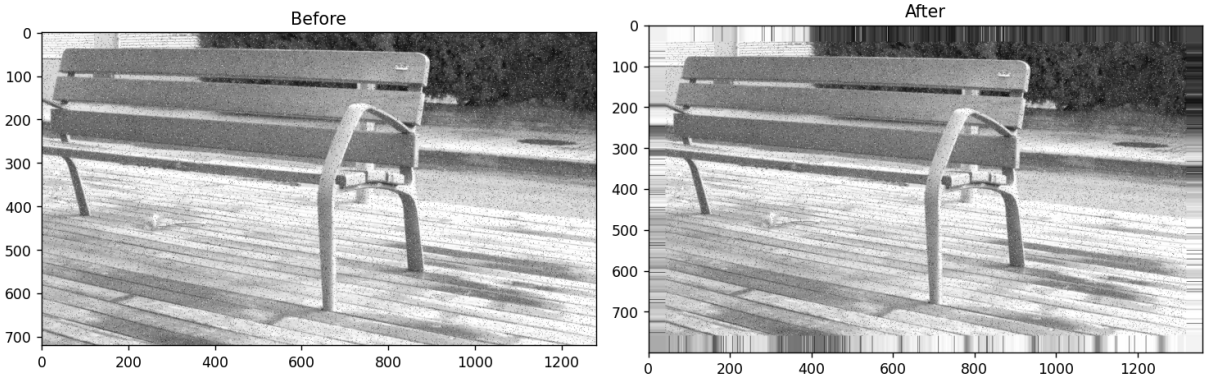

Figure 1: Padding result with filter_size set to 80 for visibility

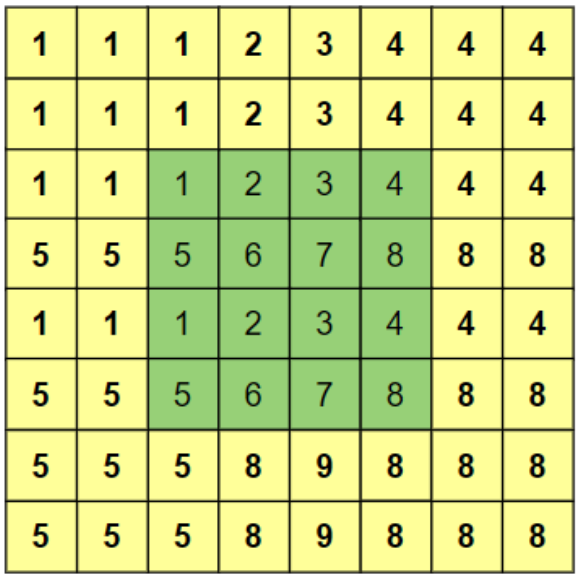

Figure 2: Illustration of replicate padding

Box/mean filter

A low-pass filter that smooths the image by making each output pixel the average of values in a square kernel. Result is shown in Figure 3:

```
def mean_filter(img, filter_size=3):
    """
    Smoothing image with mean 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 mean filter.
    """
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img)
    rows, cols = img.shape

    for i in range(rows):
        for j in range(cols):
            patch = padded_img[i:i+filter_size,j:j+filter_size]
            smoothed_img[i,j] = np.mean(patch)
    return smoothed_img
```

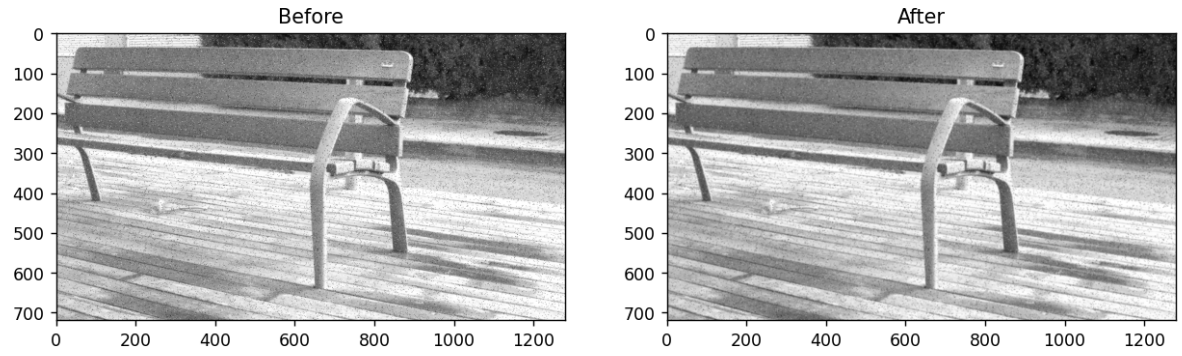

Figure 3: $mean\_filter$ with $filter\_size = 3$

Median filter

Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the mean of neighboring pixel values, it replaces it with the median of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value.

```
def median_filter(img, filter_size=3):
    padded_img = padding_img(img, filter_size)
    smoothed_img = np.zeros_like(img)
    rows, cols = img.shape

    for i in range(rows):
        for j in range(cols):
            patch = padded_img[i:i+filter_size,j:j+filter_size]
            smoothed_img[i,j] = np.median(patch)
    return smoothed_img
```

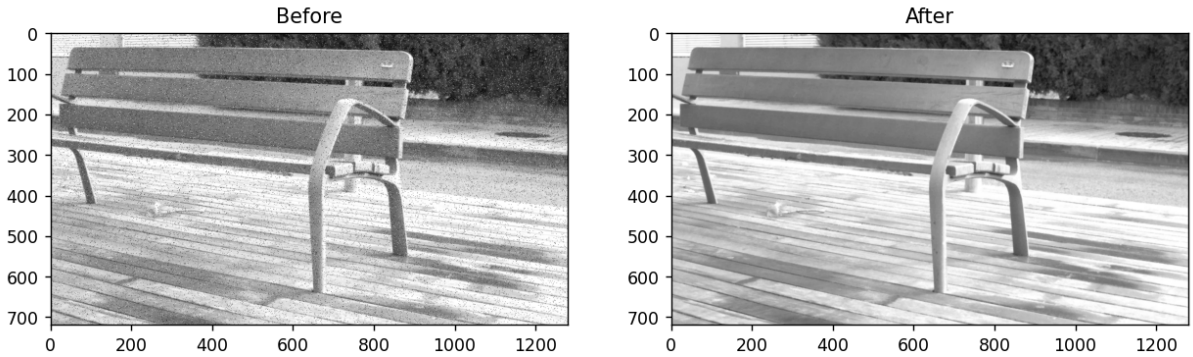

Figure 4: $median\_filter$ with $filter\_size = 3$

1.2 Evaluation

To compare the quality of those two filters for the provided image, we implement the Peak Signal-to-Noise Ratio (PSNR) metric

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX^2}{MSE} \right)$$

where MAX is the maximum possible pixel value (typically 255 for 8-bit images), and MSE is the Mean Square Error between the two images.

```python
def psnr(gt_img, smooth_img):
    # Calculate the Mean Square Error (MSE)
    mse = np.mean((gt_img - smooth_img) ** 2)

    # If MSE is zero, two images are exactly the same, so return infinity
    if mse == 0:
        return float('inf')

    # Check the image depth and decide the maximum pixel value
    max_pixel = 0
    if gt_img.dtype == np.uint8:
        max_pixel = 255.0
    elif gt_img.dtype == np.uint16:
        max_pixel = 65535.0
    else:
        max_pixel = 1.0

    # Calculate the PSNR score
    psnr_score = 20 * math.log10(max_pixel / math.sqrt(mse))

    return psnr_score
```

PNSR scores for the filters are:

1. Mean filter: 31.60889963499979
2. Median filter: 37.11957830085524

→ Considering the median filter's higher score, it is therefore a better choice for the provided image.

2 Fourier Transform

2.1 1D Fourier Transform

To calculate the Discrete Fourier Transform on a one-dimensional signal with formula:

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

where s represents the frequency, and n denotes the sampling order.

We follow these steps:

1. **Initialization:** an array DFT of complex numbers with the same length as the input data, filled with zeros. This array will store the results of the DFT computation.
2. **Nested Loops:** The function then iterates over all frequencies s from 0 to N-1, where N is the length of the input data. Within this loop, it iterates over all time samples n from 0 to N-1.
3. **Computing the DFT:** For each frequency s, it calculates the corresponding DFT coefficient by multiplying each sample of the input signal data with the corresponding complex exponential term $e^{-i2\pi sn/N}$ and summing up these terms for all time samples.

```
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)
  DFT = np.zeros(N, dtype=complex)
  for s in range(N):
    for n in range(N):
      WN = np.exp(—2j * np.pi * s * n / N)
      DFT[s] += data[n] * WN
  return DFT
```

2.2 2D Fourier Transform

The procedure to simulate a 2D Fourier Transform is as follows:

1. Conducting a Fourier Transform on each row of the input 2D signal. This step transforms the signal along the horizontal axis.
2. Perform a Fourier Transform on each column of the previously obtained result.

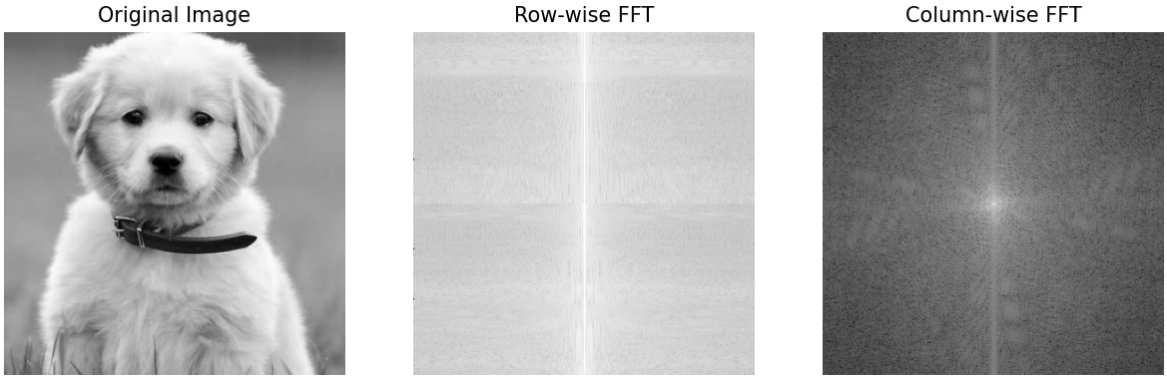

Figure 5: Output after performing 2D Fourier Transform

```python
def DFT_2D(gray_img):
    """
    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
    """
    H, W = gray_img.shape
    # Conducting a Fourier Transform on each row of the input 2D signal
    row_fft = np.zeros_like(gray_img, dtype=np.complex_)
    for i in range(H):
        row_fft[i, :] = np.fft.fft(gray_img[i, :])

    # Perform a Fourier Transform on each column of the previously
    obtained result
    row_col_fft = np.zeros_like(row_fft, dtype=np.complex_)
    for j in range(W):
        row_col_fft[:, j] = np.fft.fft(row_fft[:, j])

    return row_fft, row_col_fft
```

2.3 Fourier Transform Applications

2.3.1 Frequency Removal Function

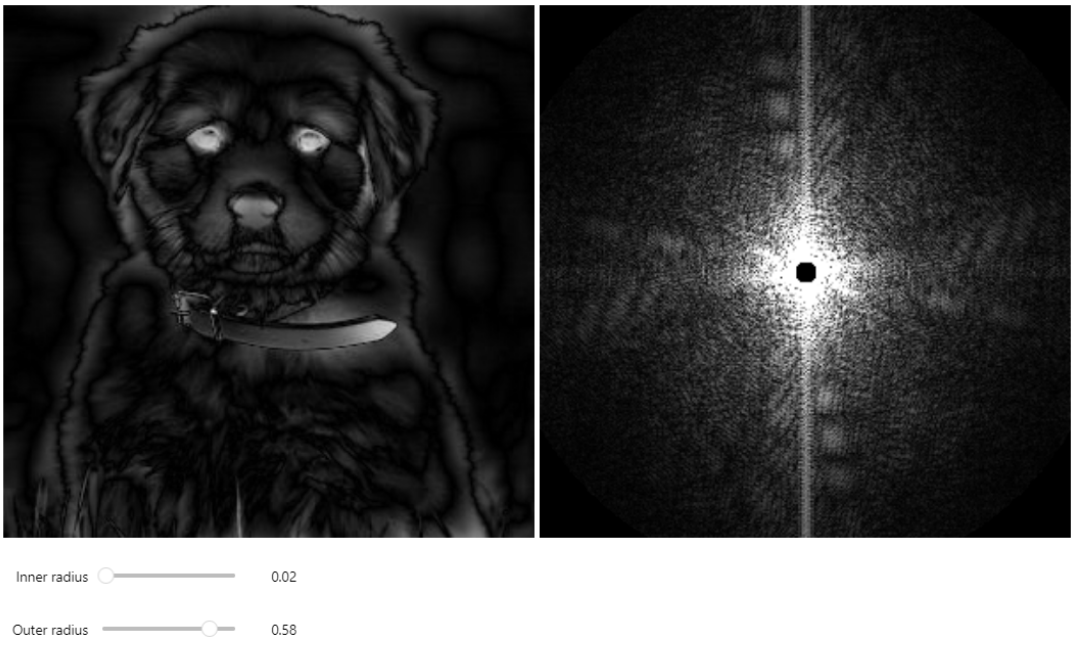

Figure 6: Output for 2D Frequency Removal Exercise

To filter frequency in an image based on a given mask, we implement the steps as listed in *hw2_title*. Beside, noted that the resulting images may contain complex numbers. They represent both amplitude and phase information of the signal. However, when visualizing the filtered spatial image, we typically only want to display the magnitude of the signal, disregarding the phase information. The last step is required as a consequence.

```
def filter_frequency(orig_img, 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
  """
  # 1. Transform using fft2
  img_fft = np.fft.fft2(orig_img)

  # Shift frequency coefs to center using fftshift
  img_ffshift = np.fft.fftshift(img_fft)

  # Filter in frequency domain using the given mask
  f_img = img_ffshift * mask

  # Shift frequency coefs back using ifftshift
  f_img_shift = np.fft.ifftshift(f_img)

  # Invert transform using ifft2
  img = np.fft.ifft2(f_img_shift)

  # Extract absolute value for visualization
  img = np.abs(img)
  f_img = np.abs(f_img)
  return f_img, img
```

2.3.2 Creating a Hybrid Image

The basic idea is to combine the low frequencies from one image (img1) with the high frequencies from another image (img2).

1. Convert both images into the frequency domain using np.fft.fft2.
2. The frequency representations are then centered using np.fft.fftshift. This step ensures that the low frequencies are located around the center of the resulting arrays.
3. Creating the mask: A mask with the same dimensions as the frequency-domain representation of the first image is initialized with zeros. Inside the loop, the distance of each pixel (i, j) from the center of the mask is calculated. Check if the distance is less than or equal to the provided radius (r). If the condition is true, it means this pixel falls within the circle defined by the radius. So, the corresponding element in the mask (mask[i, j]) is set to 1.
   Understanding the Mask: This mask essentially creates a circular region in the frequency domain. Pixels within the circle (closer to the center) correspond to lower frequencies, while those outside represent higher frequencies.

By setting the mask values to 1 inside the circle and 0 outside, the function controls which frequencies to keep from each image.

4. Combining Frequencies:

   hybrid_fft: The mask is then used to combine the frequency representations of the two images.

   img1_fftshift * mask: This part keeps the low frequencies from the first image by multiplying it with the mask (where values inside the circle become 1).
   img2_fftshift * (1 - mask): This keeps the high frequencies from the second image by multiplying it with the inverted mask where values outside the circle become 1.

5. Inverse Transformation: The combined frequency representation (hybrid_fft) is then shifted back using np.fft.ifftshift and transformed back into the spatial domain using np.fft.ifft2.

6. Finally, the absolute value of the resulting image (hybrid_image) is returned. This ensures the pixel values are within the expected range for an image.

```python
def create_hybrid_img(img1, img2, r):
    """
  Params:
    img1: numpy image 1
    img2: numpy image 2
    r: radius that defines the filled circle of frequency of image 1.
  """
  # Transform using fft2
  img1_fft = np.fft.fft2(img1)
  img2_fft = np.fft.fft2(img2)

  # Shift frequency coefs to center using fftshift
  img1_fftshift = np.fft.fftshift(img1_fft)
  img2_fftshift = np.fft.fftshift(img2_fft)

  # Create a mask based on the given radius (r) parameter
  mask = np.zeros(img1_fftshift.shape)
  for i in range(img1_fftshift.shape[0]):
      for j in range(img1_fftshift.shape[1]):
          dist = np.sqrt((i - img1_fftshift.shape[0] // 2) ** 2
          + (j - img1_fftshift.shape[1] // 2) ** 2)
          if dist <= r:
              mask[i, j] = 1

  # Combine frequency of 2 images using the mask
  hybrid_fft = img1_fftshift * mask + img2_fftshift * (1 - mask)

  # Shift frequency coefs back using ifftshift
  hybrid_fftshift = np.fft.ifftshift(hybrid_fft)

  # Invert transform using ifft2
  hybrid_image = np.fft.ifft2(hybrid_fftshift)

  return np.abs(hybrid_image)
```

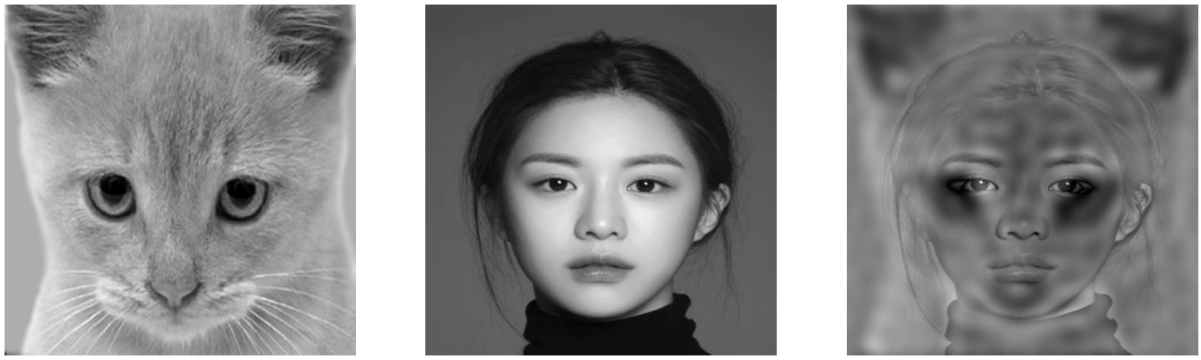

Figure 7: Input images and resulting hybrid image

3 Source code

Source code and images are placed in GitHub: link.