Python-based Image Processing with FFT

Hasan Amin and Ehtasham Khattak

January 9, 2024

1 Introduction

The Fast Fourier Transform (FFT) is an algorithmic breakthrough that expedites the calculation of the Discrete Fourier Transform (DFT) from $O(N^2)$ to $O(N \log N)$ complexity, where N signifies the dataset size. By swiftly converting time-domain signals into their frequency domain counterparts, FFT revolutionized signal processing by significantly reducing computational demands. This efficiency enables rapid analysis of signal frequencies, finding extensive applications in various fields like image processing, telecommunications, and scientific research, where rapid and efficient frequency analysis is paramount.

2 Abstract

This project focuses on implementing the Fast fourier Transformation in python and then use it for various Image processing tasks

3 Mathematical Insights

3.1 Discrete Fourier Transformation (DFT)

The DFT takes in a sequence of complex values and breaks them into their constituent frequencies in terms of sines and cosines.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-i2\pi kn}{N}}$$

where X_k represents the $k\text{-}th$ frequency component of the signal or image. This transformation breaks down a sequence of complex values x_n into its constituent frequencies, revealing the amplitude and phase information of each frequency within the signal. This form of signal can be processed for instance the frequecies can be averaged and reduce the overall size of the array.Moreover filters can be applied to sharpen or blur the image. To implement a 2D transformation we apply the process on rows and columns

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) \cdot e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

where $F(u,v)$ Represents the 2D frequency-domain signal after performing the DFT. This signal is a function of two discrete variables; u and v which denote the frequencies in the rows of length M and columns of length N, respectively.

3.2 Inverse Discrete Fourier Transformation (IDFT)

The IDFT reverses the effect of DFT and returns us the original signal sequence

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{\frac{i2\pi kn}{N}}$$

Similarly for a 2D DFT the corresponding IDFT is

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v) \cdot e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}$$

4 Algorithm Analysis

4.1 The Radix-2 Cooley-Tukey Approach

Here we implement the radix-2 Cooley-Tukey approach to implement the Fast Fourier Transform.The Cooley-Tukey method ingeniously employs a divide-and-conquer strategy, breaking down the DFT computation into successively smaller DFTs. By recursively decomposing the transform, utilizing the periodicity properties of complex exponentials, and exploiting symmetries within the computation, the radix-2 Cooley-Tukey algorithm dramatically reduces the number of arithmetic operations required for calculating the Fourier Transform. Its elegance lies in its ability to efficiently handle input sizes that are powers of 2, leveraging this characteristic to optimize memory access patterns and computational complexity, thereby significantly accelerating the computation of the Fourier Transform in processing data.

4.2 The Fast Fourier Transformation (FFT)

The factor which makes the Fourier Transform 'Fast' is the implementation of the technique called Divide and conquer here we split our 2D Array into even and odd entries and apply the fourier transformation recursively.

```
def _fft1d(self, x):
    '''
    Applies 1-dimensional Fast Fourier Transform (FFT) using recursive
    ↪  Cooley-Tukey algorithm.

    Parameters
    ----------
    - x (numpy.ndarray): Input 1D numpy array.

    Returns:
    - numpy.ndarray: FFT of the input array.

    This method recursively computes the 1D FFT of the input array using
    ↪  the Cooley-Tukey algorithm. It splits the input array into even
    ↪  and odd indices, computes their respective FFTs, and combines them
    ↪  to produce the final FFT result.
    '''
    N = len(x)
    if N <= 1:
```

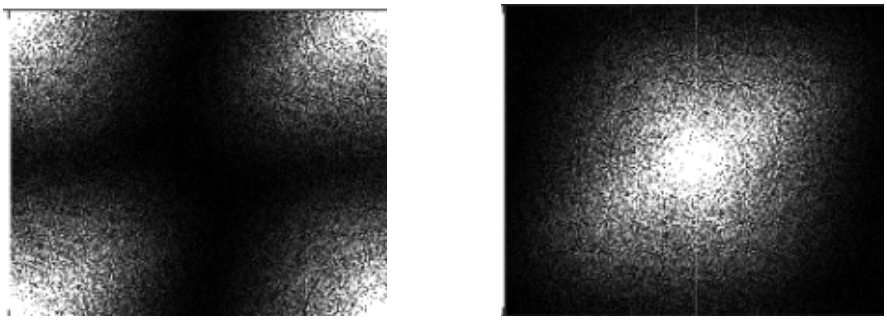
```
        return x
    even = self._fft1d(x[::2])
    odd = self._fft1d(x[1::2])
    factor = np.exp(-2j * np.pi * np.arange(N // 2) / N)
    first_half = even + factor * odd
    second_half = even - factor * odd
    return np.concatenate([first_half, second_half])
```

We apply the $fft1d$ function recursively to the $fft2d$. Also, observe the padding constructed of powers of two.This is one of the key characteristics of Cooley-Tukey implementation.

```
def fft2d(self, shape=None):
    '''
    Applies 2-dimensional Fast Fourier Transform (FFT) using the
    ↪  Cooley-Tukey algorithm.

    Parameters
    ----------
    - shape (tuple, optional): Target shape of the FFT result. If
    ↪  provided, the result will be adjusted to fit this shape.

    Returns:
    - FFT2D: An object with the 2D FFT array.

    If shape is specified and:
    - Both dimensions are smaller or equal to the original array, the
    ↪  resulting FFT will be cropped.
    - One dimension exceeds the original array, the result will be
    ↪  zero-padded along that dimension.
    - Both dimensions exceed the original array, the result will be
    ↪  zero-padded to fit the specified shape.
    '''
    # Pad rows and columns to the nearest power of 2
    rows_power_of_2 = 2 ** int(np.ceil(np.log2(self.array.shape[0])))
    cols_power_of_2 = 2 ** int(np.ceil(np.log2(self.array.shape[1])))

    padded_array = np.zeros((rows_power_of_2, cols_power_of_2))
    padded_array[: self.array.shape[0], : self.array.shape[1]] =
    ↪  self.array
```

```
# Apply 1D FFT along rows
fft_rows = np.array([self._fft1d(row) for row in padded_array])
# fft_rows = np.array([self.fft1d(row) for row in self.array])

# Apply 1D FFT along columns
fft_cols = np.array(
    [self._fft1d(fft_rows[:, i]) for i in range(cols_power_of_2)]
).T
fft_cols=FFT2D(fft_cols).fftshift()
if shape is not None:
    if shape[0] <= self.rows and shape[1] <= self.cols:
        fft_cols = fft_cols[: shape[0], : shape[1]]
    elif shape[0] > self.rows and shape[1] <= self.cols:
        # Shape exceeds rows, but fits within columns
        zeros = np.zeros(shape, dtype=np.complex128)
        zeros[: self.rows, : self.cols] = fft_cols[: self.rows, :
        ↪  shape[1]]
        fft_cols = zeros
    if shape[0] <= self.rows and shape[1] > self.cols:
        # Shape fits within rows, but exceeds columns
        zeros = np.zeros(shape, dtype=np.complex128)
        zeros[: self.rows, : self.cols] = fft_cols[: shape[0], :
        ↪  self.cols]
        fft_cols = zeros
    else:
        zeros = np.zeros(shape, dtype=np.complex128)
        zeros[: self.rows, : self.cols] = fft_cols[: self.rows, :
        ↪  self.cols]
        fft_cols = zeros
# self.array = fft_cols
return FFT2D(fft_cols)
```

You should notice in applying the Fast-Fourier Transform to the 2D Array, after the transformation, we applied the shift function to the obtained *fft_cols*. Actually, in applying the FFT, the DC component which has a frequency of 0 Hz (or relatively the lowest frequency) moves towards the edges. This function basically recenters it.



(a) Before applying the shift (b) After applying the shift

Figure 1: Effect of applying the shift function

```
def fftshift(self):
    '''
    Performs a shift of the FFT array to center the zero frequency
    ↪  component.

    Returns:
    - numpy.ndarray: Shifted FFT array.
    '''
    mid_row,mid_col=self.rows//2,self.cols//2
    shifted_rows = np.concatenate((self.array[mid_row:, :],
    ↪  self.array[:mid_row, :]), axis=0)
    shifted_arr = np.concatenate((shifted_rows[:, mid_col:],
    ↪  shifted_rows[:, :mid_col]), axis=1)

    # return FFT2D(shifted_arr)
    return shifted_arr
```

4.3 The Inverse Fast Fourier Transformation (IFFT)

We first simply implement the IFFT for 1D array and then use recursion to apply it to our 2D array.

```
def ifft1d(self, x):
    '''
    Applies 1-dimensional Inverse Fast Fourier Transform (IFFT) using a
    ↪  recursive algorithm.
```

```
    Parameters
    ----------
    - x (numpy.ndarray): Input 1D numpy array.

    Returns:
    - numpy.ndarray: IFFT of the input array.

    This method recursively computes the 1D IFFT of the input array using
    ↪  a recursive algorithm. It splits the input array into even and odd
    ↪  indices, computes their respective IFFTs, and combines them to
    ↪  produce the final IFFT result. The signs of the factor are
    ↪  reversed to perform the inverse operation.
    '''
    N = len(x)
    if N <= 1:
        return x
    even = self.ifft1d(x[::2])
    odd = self.ifft1d(x[1::2])
    # Reverse the sign for inverse
    factor = np.exp(2j * np.pi * np.arange(N // 2) / N)
    first_half = even + factor * odd
    second_half = even - factor * odd
    return np.concatenate([first_half, second_half]) / 2
```

To reverse the effect of the FFT, we sequentially reverse fft on columns and then on rows, the mathematics is the same as of the IDFT, however, similar to the FFT the IFFT uses the divide snd conquer technique to improve code efficiency and reduce the time complexity from $O(N^2)$ to $O(N \log N)$. Hence, we implement the IFFT for our 2D array.

```
def ifft2d(self, original_shape):
    '''
    Applies 2-dimensional Inverse Fast Fourier Transform (IFFT) to a 2D
    ↪  array.

    Parameters
    ----------
    - original_shape (tuple): The original shape of the input array before
    ↪  padding.
```

```
    Returns:
    - numpy.ndarray: 2D array with the original shape after applying
    ↪  IFFT.
    '''
    # Apply 1D inverse FFT along columns
    ifft_cols = np.array([self.ifft1d(col) for col in self.array.T])

    # Apply 1D inverse FFT along rows
    ifft_rows = np.array([self.ifft1d(row) for row in ifft_cols.T])
    # ifft_rows=FFT2D(ifft_rows).ifftshift()
    cropped_ifft = ifft_rows[: original_shape[0], : original_shape[1]]

    return np.real(cropped_ifft)
```

5 Filters

In image processing, filters play a pivotal role in manipulating and enhancing images. These filters are mathematical transformations applied to modify pixel values within an image. High-pass filters accentuate edges and details by amplifying rapid changes in pixel values, effectively highlighting high-frequency components, such as edges and fine textures, while suppressing lower frequencies. Conversely, low-pass filters work inversely, emphasizing smooth regions and gradual transitions by attenuating high-frequency variations. For instance, a typical example of a high-pass filter is the Laplacian filter, which sharpens image details by highlighting abrupt changes in intensity. On the other hand, low-pass filters like the Gaussian filter smooth images by reducing noise and blurring details, achieved by averaging pixel values within a local neighborhood defined by a Gaussian kernel.

6 Gaussian Filter

6.1 Mathematics

We implemented the gaussian filter as follows:

$$L(u,v) = e^{-\frac{D(u,v)^2}{2\sigma^2}}$$

$$D(u,v) = \sqrt{(u-\frac{M}{2})^2 + (v-\frac{N}{2})^2}$$

where $L(u,v)$ is the low pass Gaussian filter(blurring) and $D(u,v)$ is the distance from the center of the Fourier transformed image to the point (u,v). Here (M,N) represent the number of rows and columns of the array. We can subsequenty implement the High-pass filter (which sharpens the boundaries with higher frequencies for edge detection) $H(u,v)$ by $1-L(u,v)$. Here we can increase/decrease σ to decrease/increase the blurring respectively.

6.2 Implementation

```
def gaussian_filter(image,sigma=10,ifft=True,low_pass=True):
    '''
    Applies a Gaussian filter to the input image in the frequency domain.

    Parameters
```

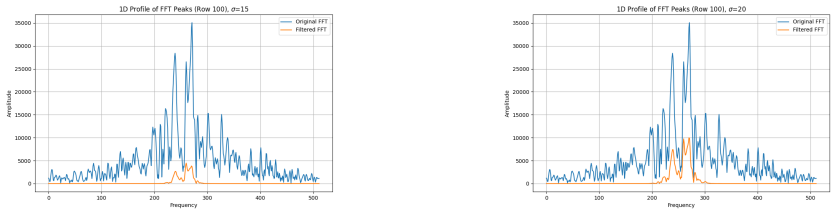
```
    ----------
    - image (numpy.ndarray): Input image array.
    - sigma (float, optional): Cut-off frequency for the Gaussian filter.
    ↪  Defaults to 10.
    - ifft (bool, optional): If True, performs an inverse Fourier
    ↪  transform (IFFT) to obtain the filtered image in spatial domain.
    ↪  Defaults to True.
    - low_pass (bool, optional): If True, applies a low-pass Gaussian
    ↪  filter. If False, applies a high-pass filter. Defaults to True.

    Returns:
    If `ifft` is True:
    - filtered_image (numpy.ndarray): Filtered image in spatial domain.

    If `ifft` is False:
    - filtered_image_fft (numpy.ndarray): Filtered image in the frequency
    ↪  domain.
    - image_fft (numpy.ndarray): Original image in the frequency domain.
    '''
    image_obj = FFT2D(image)
    image_fft = image_obj.fft2d()
    M,N=image_fft.array.shape
    L=np.zeros((M,N),dtype=np.float32)
    for u in range(M):
        for v in range(N):
            D=np.sqrt((u-M/2)**2 + (v-N/2)**2)
            L[u,v]=np.exp(-D**2/(2*sigma**2))
    if low_pass:
        filtered_image_fft=(image_fft*FFT2D(L)).ifftshift()

    else:
        H=1-L
        filtered_image_fft=(image_fft*FFT2D(H)).ifftshift()

    if ifft:
        filtered_image = np.abs(filtered_image_fft.ifft2d(image.shape))
        return filtered_image
    else:
        return filtered_image_fft.fftshift(), image_fft
```

Observe after we applied the filter we Reverse the fft shift we applied earlier to restore the image to its original form. The implementation of the ifft-shift is as follows:

```
def ifftshift(self):
    '''
    Undoes the shift performed by fftshift to restore the original FFT
    ↪  array.

    Returns:
    - numpy.ndarray: Unshifted FFT array.
    '''
    mid_row,mid_col=self.rows//2,self.cols//2
     # Undo the shift along columns
    shifted_cols = np.concatenate((self.array[:, -mid_col:], self.array[:,
    ↪  :-mid_col]), axis=1)

    # Undo the shift along rows
    unshifted_arr = np.concatenate((shifted_cols[-mid_row:, :],
    ↪  shifted_cols[:-mid_row, :]), axis=0)

    return FFT2D(unshifted_arr)
```

6.3 Results

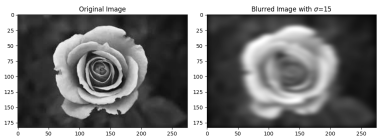
Firstly we present the graphical presentation of how on varying σ, the frequencies are filtered using the $L(u, v)$:



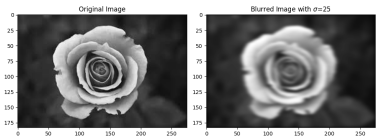
(a) σ=15 (b) σ=20

Figure 2: Low pass Frequency Filtering

Consequently, we can observe the results of our filter on an actual image:



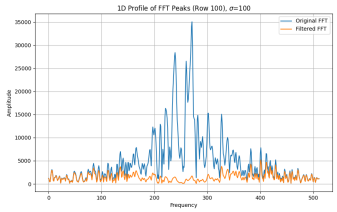
(a) σ=15



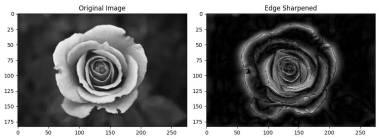
(b) σ=25

Figure 3: Image Blurring

Now let us observe the results of our High-pass filter on the frequencies and the image respectively.After applying the high-pass filter, we examine the impact on both the frequency domain and the resultant image. In the frequency domain, the high-pass filter accentuates high-frequency components, highlighting abrupt changes and edges while suppressing lower frequencies. This emphasis on rapid intensity changes aids in edge detection and enhances image details. Meanwhile, in the spatial domain, the high-pass filter amplifies the differences between neighboring pixels, resulting in sharper edges and finer details within the image.



(a) $H[u, v]$ frequency filtering



(b) Edge Sharpening

Figure 4: High-pass frequency filter and Image result