Python-based Image Processing with FFT

Hasan Amin

January 2, 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.

```
class FFT2D:
    def __init__(self, array):
        self.array = array
        self.rows = array.shape[0]
        self.cols = array.shape[1]
        self.original_shape = None

    def fft1d(self, x):
        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):
    # 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

    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
```

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.

```
        return FFT2D(fft_cols)

    def ifft1d(self, x):
        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
```

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.

```
        return np.concatenate([first_half, second_half]) / 2

    def ifft2d(self, original_shape):
        # 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])
        cropped_ifft = ifft_rows[: original_shape[0], : original_shape[1]]
```