# Fast Fourier Transform (FFT) Optimization for Image Processing

Course work by Kiril Shiyan

## Abstract

This work shows that FFT based convolution is significantly more faster than ordinary one, explains in theory why it is faster and proves it on examples using programming language Python.

## Introduction

Many problems in different domains, ranging from optics and radar to signal processing and acoustics involve using Fourier transform for a certain function. Since Discrete Fourier transform (DFT) is very slow and is O(n^2) algorithm, it`s not applicable and scalable for real world tasks, such as signal compression and signal processing. Therefore, an enhanced algorithm, Fast Fourier transform (FFT), which is a recursive version of DFT, was discovered and since it`s complexity is $O(n*\log_2 n)$, it became wildly used in all sorts of signal processing tasks. There`s another wildly used algorithm called convolution, which allows for a wide variety of operations on images, such as sharpening, smoothing, edge detection etc. But convolution is O(n^2) as well, so to make it efficient we need to combine FFT and convolution to create FFT based convolution which is $O(n*\log_2 n)$ algorithm as well. Aim of this course work is to compare performance of ordinary convolution and convolution which uses FFT, to show just how more efficient it is in real world applications.

# Methods

## Discrete Fourier Transform

Discrete Fourier transform is equivalent of the Continuous Fourier transform for signals knows only at specific instances N separated by time intervals T, meaning we have a finite set of data, unlike infinite for a Continuos Fourier transform.
Let f(t) be a continuous signal representing some data. Let N samples of this signal be denoted as f[0], f[1] and so on up to f[N-1]
Fourier Transform of original signal f(t) would be:

$$F(j\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

We could regard each value f[k] as a single impulse which has a certain area, therefore, since function is limited to certain points, we can show represent $F(j\omega)$ as next:

$$\begin{aligned} F(j\omega) &= \int_{o}^{(N-1)T} f(t)e^{-j\omega t}dt \\ &= f[0]e^{-j0} + f[1]e^{-j\omega T} + \ldots + f[k]e^{-j\omega kT} + \ldots f(N-1)e^{-j\omega(N-1)T} \end{aligned}$$

$$\text{ie.} \quad F(j\omega) = \sum_{k=0}^{N-1} f[k]e^{-j\omega kT}$$

Courtesy of Oxford University [3]

Similar to Continuous Discrete transform, DFT can be calculated over a finite time interval in case function is periodic, so, since for DFT we have a finite range of data, DFT treats values as though they are periodic. Example of this show on next figure.

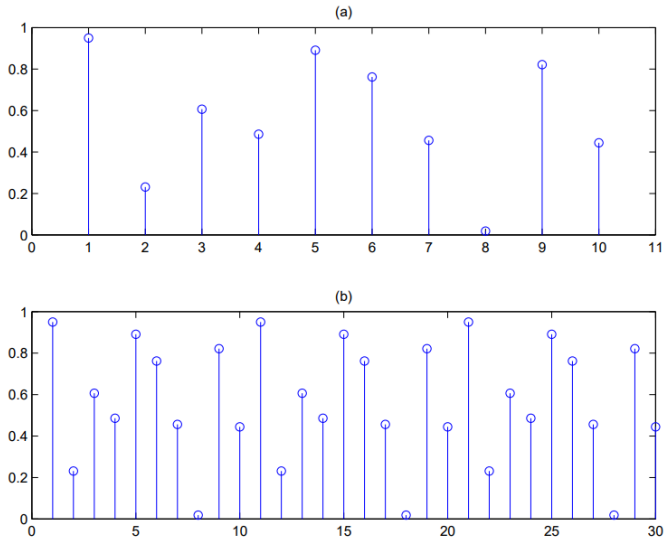



Figure 7.1: *(a) Sequence of $N = 10$ samples. (b) implicit periodicity in DFT.*

Courtesy of Oxford University [3]

Since DFT treats values as though they are periodic, we need to calculate DFT for fundamental frequency and it`s harmonics (one cycle per sequence, 1/(N * T) Hz) and it`s harmonics. General formula will be:

$$F[n] = \sum_{k=0}^{N-1} f[k] e^{-j\frac{2\pi}{N}nk} \qquad (n = 0 : N-1)$$

Courtesy of Oxford University [3]

For a single f[k] value. For all values from f[0] to f[n-1] DFT can be calculated as following:

$$\begin{pmatrix} F[0] \\ F[1] \\ F[2] \\ \vdots \\ F[N-1] \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & W & W^2 & W^3 & \dots & W^{N-1} \\ 1 & W^2 & W^4 & W^6 & \dots & W^{N-2} \\ 1 & W^3 & W^6 & W^9 & \dots & W^{N-3} \\ \vdots & & & & & \\ 1 & W^{N-1} & W^{N-2} & W^{N-3} & \dots & W \end{pmatrix} \begin{pmatrix} f[0] \\ f[1] \\ f[2] \\ \vdots \\ f[N-1] \end{pmatrix}$$

where $W = \exp(-j2\pi/N)$ and $W = W^{2N}$ etc. $= 1$.

Courtesy of Oxford University [3]

As seen. we need to calculate a sum of N elements N times, so DFT is O(n^2) algorithm

## Fast Fourier Transform

Since DFT is too slow, a better algorithm, FFT was created.
Suppose we separated DFT into odd and even sequences such as
$n = 2r \; if \; r \; is \; even$
$n = 2r + 1 \; uf \; r \; is \; odd$
where r = 1, 2, … $N/2 - 1$

By doing certain transformations, we can come up with formula that gives us DFT as a sum of two factors, odd and even one, which is somewhat faster because those can be calculated in parallel.

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r)}{N}} \quad + \quad x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(2r+1)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r)}{N}} \quad + \quad x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(2r)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(r)}{N/2}} \quad + \quad x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1] e^{\frac{-j2\pi k(r)}{N/2}}$$

$$x[k] = x_{even}[k] \; + \; e^{\frac{-j2\pi k}{N}} x_{odd}[k]$$

Courtesy of Cory Maklin [5]

By splitting DFT more than one time, we can get better performance and get an algorithm with time complexity $O(n * logn)$

What if we keep splitting?

$$\frac{N}{2} \longrightarrow 2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$$

$$\frac{N}{4} \longrightarrow 2\left(2\left(\frac{N}{4}\right)^2 + \frac{N}{2}\right) + N = \frac{N^2}{4} + 2N$$

$$\frac{N}{8} \longrightarrow 2\left(2\left(2\left(\frac{N}{8}\right)^2 + \frac{N}{4}\right) + \frac{N}{2}\right) + N = \frac{N^2}{8} + 3N$$

$$\vdots$$

$$\frac{N}{2^P} \longrightarrow \frac{N^2}{2^P} + PN = \frac{N^2}{N} + (\log_2 N)N = N + (\log_2 N)N$$

$$\sim O(N + Nlog_2 N) \sim O(Nlog_2 N)$$

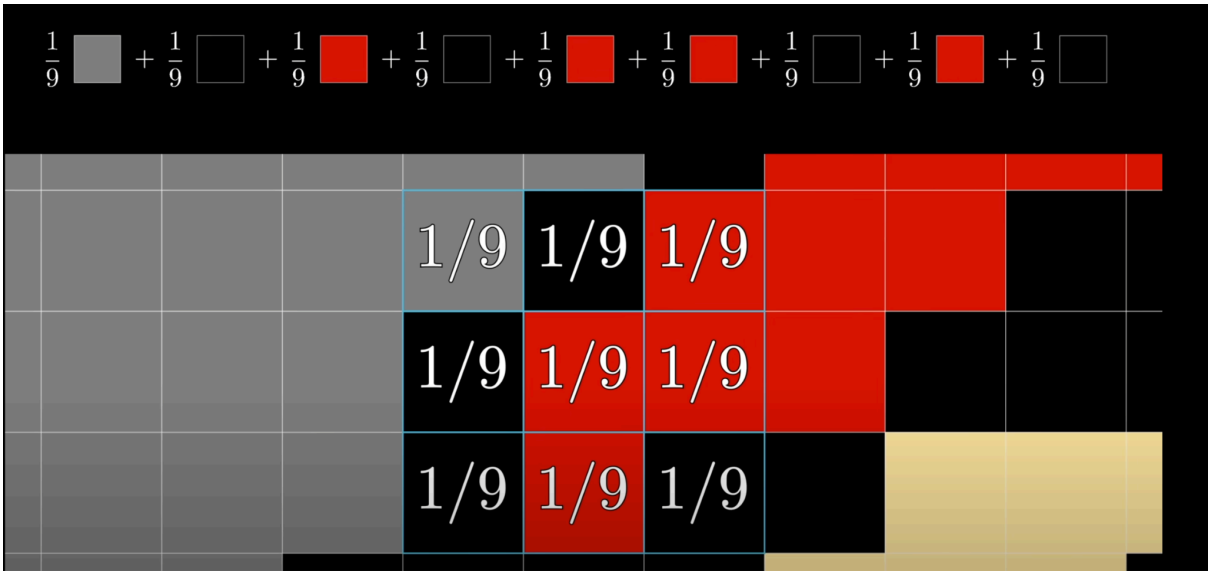
Courtesy of Cory Maklin [5]

## Convolution

Another method, wildly used in image processing, and not connected with Fourier transform, is a method knows as convolution.
Formula for finding a convolution of two functions is as follows:

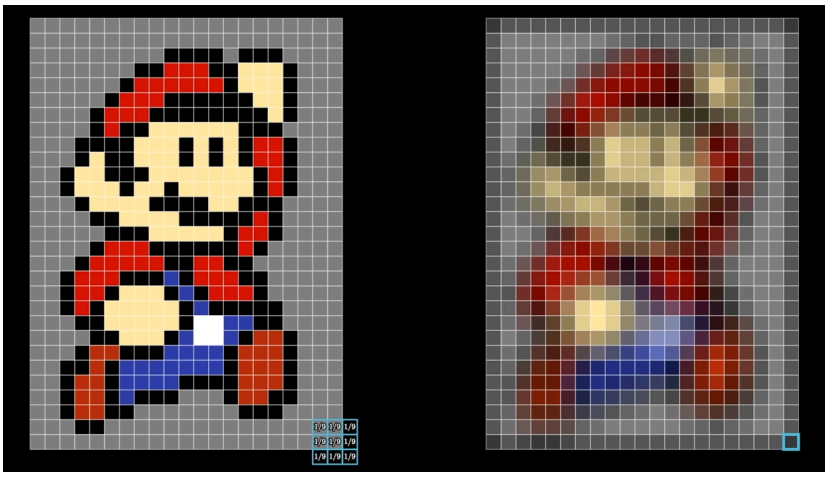
$$(f * g)(t) = \int_{-\infty}^{\infty} f(x).g(t-x)dx$$

By taking convolution, using a signal and a mask, we can find average of given pixel and pixel surrounding it, weighted by mask.




Courtesy of 3Blue1Brown [6]

Here`s example using a 3x3 mask with values of 1/9. As seen from image, we take a value in given point weighted by a value of mask and add all surrounding pixels, also weighted by mask values. Since it is essentially the same as taking the average of surrounding values, using such a mask will smooth image. Result of convolution:

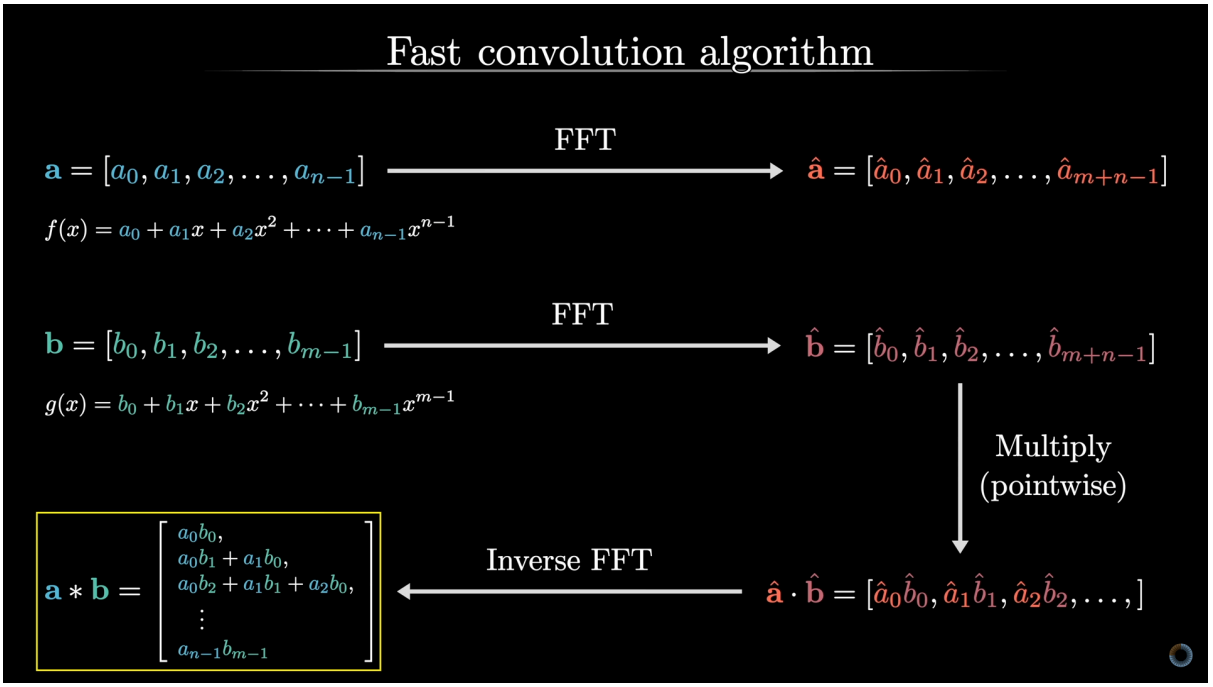


Courtesy of 3Blue1Brown [6]

Convolution, as well as DFT, is a O(n^2) operation, because it needs to compute all possible combinations of image and mask, therefore, just as

DFT, it`s not applicable and scalable for real world application. That`s why FFT and convolution were combined, to create a very fast FFT based convolution algorithm.

FFT based convolution




Courtesy of 3Blue1Brown [6]

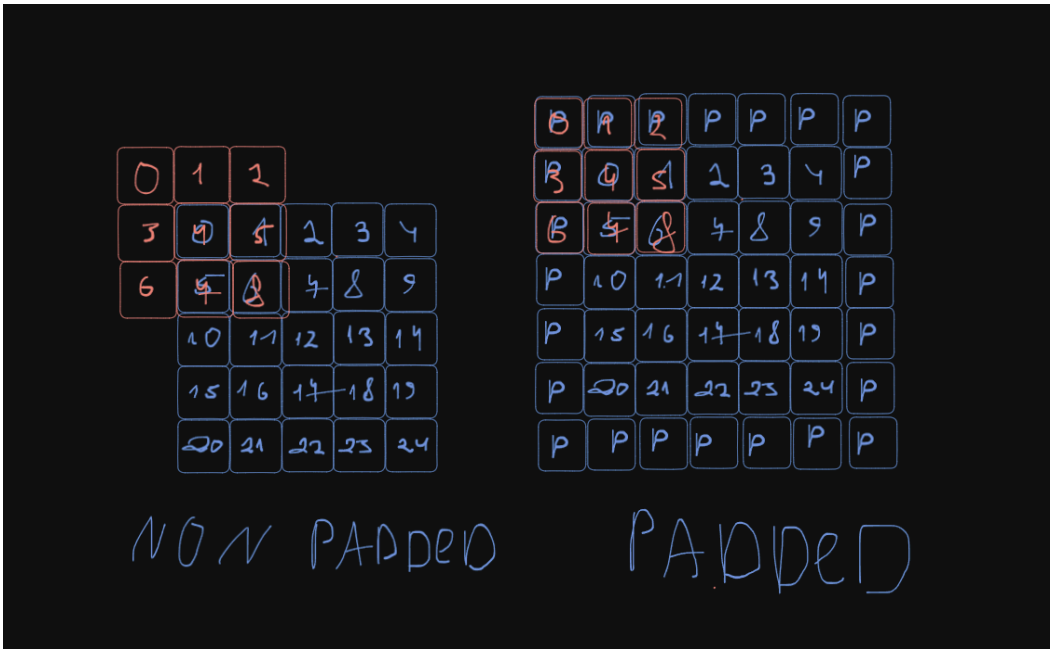
To simplify convolution, what we can do is represent our function as a polynomial and find a coefficients for that polynomial. After this, we can apply a FFT, which is O(n*log(n)) operation, over given function and it`s coefficients and find coefficients in frequency domain. After this, we apply pointwise multiplication to coefficients of two functions and get a multiplication of those values in frequency domain. After this, we can apply inverse FFT again and receive a convolution of given function in spatial domain. From that we can say that convolution in spatial domain is equal to multiplication in frequency domain.

# Methodology

To compare performance of a ordinary convolution and FFT based convolution, I`ll use Python script, which implements both algorithms. First, just convolution in spatial domain. For this I`ll use a Python numpy library and following script:

```python
def rotate_180(array, M, N):
    out = np.zeros((M, N))
    for i in range(M):
        for j in range(N):
            out[i, N-1-j] = array[M-1-i, j]
    return out

def convolution2d(image, kernel):
    kernel_rotated = rotate_180(kernel, kernel.shape[0], kernel.shape[1])

    image_height, image_width = image.shape
    kernel_height, kernel_width = kernel.shape

    pad_height = kernel_height // 2
    pad_width = kernel_width // 2

    padded_image = np.pad(image, ((pad_height, pad_height), (pad_width, pad_width)), mode='constant')

    output = np.zeros_like(image)

    for i in range(image_height):
        for j in range(image_width):
            output[i, j] = np.sum(padded_image[i:i+kernel_height, j:j+kernel_width] * kernel_rotated)

    return output
```

Since convolution requires to rotate kernel, we do it on line 12.
Then, on lines 14 and 15 we take dimensions of image and kernel.
On lines 17 and 18 we calculate padding size, which is required because calculating values for pixels will fail under certain conditions.
Demonstration:




On line 22 we add padding to image.

On line 22 we create output image which had identical size to input image(some data in padding will be lost, but it is irrelevant for us.)
Lines 24 to 26 are the calculation of pixel values themselves. We do it by taking kernel_width by kernel_height segment of padded_image and multiply them pointwise by kernel_rotated, and then summing together values of the resulting array to get required pixel values.

Testing algorithm against a python library scipy showed that algorithm is working:

```
if __name__ == "__main__":
    image = np.array([[1, 2, 3, 3, 3],
                      [4, 5, 6, 5, 6],
                      [7, 8, 9, 5, 6],
                      [7, 8, 9, 5, 6],
                      [7, 8, 9, 5, 6]])

    kernel = np.array([[1, 0, -1],
                       [1, 0, -1],
                       [1, 0, -1]])

    result = convolution2d(image, kernel)
    print("Result of custom 2D convolution:")
    print(result)
    print("Result of scipy 2D convolution:")
    print(scipy.signal.convolve2d(image, kernel))
```

```
Result of custom 2D convolution:
[[  7   4   1   0  -8]
 [ 15   6  -2  -3 -13]
 [ 21   6  -6  -6 -15]
 [ 24   6  -9  -9 -15]
 [ 16   4  -6  -6 -10]]
Result of scipy 2D convolution:
[[  1   2   2   1   0  -3  -3]
 [  5   7   4   1   0  -8  -9]
 [ 12  15   6  -2  -3 -13 -15]
 [ 18  21   6  -6  -6 -15 -18]
 [ 21  24   6  -9  -9 -15 -18]
 [ 14  16   4  -6  -6 -10 -12]
 [  7   8   2  -3  -3  -5  -6]]
```

As seen from result, we lose two rows and two columns of values created by convolution, but we are interested only in values which exist on original image.

Second script will be implementation of FFT based convolution.

```
30  def fft_convolve2d(image, kernel):
31      image_diff = (kernel.shape[0] - 1, kernel.shape[1] - 1)
32      kernel_diff = (image.shape[0] - 1, image.shape[1] - 1)
33
34      padded_image = np.pad(image, ((0, image_diff[0]), (0, image_diff[1])), mode='constant')
35      padded_kernel = np.pad(kernel, ((0, kernel_diff[0]), (0, kernel_diff[1])), mode='constant')
36
37      fft_image = np.fft.fft2(padded_image)
38      fft_kernel = np.fft.fft2(padded_kernel)
39
40      fft_result = fft_image * fft_kernel
41
42      result = np.fft.ifft2(fft_result)
43
44      result = result[image_diff[0] // 2:-(image_diff[0] // 2)]
45      result = [row[image_diff[1] // 2:-(image_diff[1] // 2)] for row in result]
46
47      return np.real(result)
```

On lines 31 and 32 we calculate differences, that we need to add to image and kernel. That is required because image and kernel should be of the same size to perform pointwise multiplication.
On lines 34 and 35 we add those differences to image and kernel using numpy pad method.
On lines 37 and 38 we get FFT of image and kernel and on line 40 we multiply them to get a result.
On line 42 we get an inverse FFT of result and on lines 44 and 45 we remove padding that we added in line 34.

Testing algorithm against a python library scipy showed that algorithm is working:

```
if __name__ == "__main__":
    image = np.array([[1, 2, 3, 3, 3],
                      [4, 5, 6, 5, 6],
                      [7, 8, 9, 5, 6],
                      [7, 8, 9, 5, 6],
                      [7, 8, 9, 5, 6]])

    kernel = np.array([[1, 0, -1, 0, 1],
                       [1, 0, -1, 0, 1],
                       [1, 0, -1, 0, 1],
                       [1, 0, -1, 0, 1],
                       [1, 0, -1, 0, 1]])
    print("Result of custom 2d fft convolution:")
    print(fft_convolve2d(image, kernel).astype(int))
    print("Result of scipy 2D fft convolution:")
    print(scipy.signal.fftconvolve(image, kernel).astype(int))
```

```
Result of 2d fft convolution:
[[ 6 -1  9  2  3]
 [ 8 -4 13  5  6]
 [10 -7 17  8  8]
 [ 8 -8 16  9  9]
 [ 6 -8 12  9  9]]
Result of scipy 2D convolution:
[[ 1  2  2  1  1  0  0  3  3]
 [ 5  7  4  1  5  0  0  8  9]
 [12 15  6 -1  9  2  2 12 14]
 [19 23  8 -4 13  5  5 17 20]
 [26 31 10 -7 17  8  8 22 26]
 [25 29  8 -8 16  9  8 19 23]
 [20 24  6 -8 12  9  8 14 17]
 [14 16  4 -5  8  5  5  9 11]
 [ 7  8  2 -2  4  2  3  5  6]]
```

As seen from tests, FFT convolution introduces slight uncertainties in result, but since for images we have values from 0 to 255, it won`t have a noticeable effect on image result.

# Evaluation

For testing performance of simple convolution and FFT based convolution, we`ll apply a range of filters for a certain image.

Transformations will be applied to this image:


Courtesy of Gray Line (link)

This image is 1000 pixel by 1000 pixel so it is large enough to show difference between O(n^2) and O(n*$\log_2 n$) algorithms.

Following script was used to calculate performance of functions:

```python
from convolution import  convolution2d, fft_convolve2d
import cv2
import numpy as np
import time

if __name__ == "__main__":
    im = cv2.imread("./resources/cropped.jpg", cv2.IMREAD_GRAYSCALE)

    egde_detection_kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
    sharpen_kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
    gausian_blur = np.array([[1/16, 2/16, 1/16], [2/16, 4/16, 2/16], [1/16, 2/16, 1/16]])

    startTime = time.time()
    convolution2d(im, egde_detection_kernel)
    endTime = time.time()
    print("Convolution with edge detection kernel: " + str(endTime - startTime))

    startTime = time.time()
    fft_convolve2d(im, egde_detection_kernel)
    endTime = time.time()
    print("FFT convolution with edge detection kernel: " + str(endTime - startTime))

    startTime = time.time()
    convolution2d(im, sharpen_kernel)
    endTime = time.time()
    print("Convolution with sharpen kernel: " + str(endTime - startTime))

    startTime = time.time()
    fft_convolve2d(im, sharpen_kernel)
    endTime = time.time()
    print("FFT convolution with sharpen kernel: " + str(endTime - startTime))

    startTime = time.time()
    convolution2d(im, gausian_blur)
    endTime = time.time()
    print("Convolution with gausian blur kernel: " + str(endTime - startTime))

    startTime = time.time()
    fft_convolve2d(im, gausian_blur)
    endTime = time.time()
    print("FFT convolution with gausian blur kernel: " + str(endTime - startTime))
```

Here, we just call each function for each kernel, to change it`s speed in different conditions.

Result of execution:

```
Convolution with edge detection kernel: 4.413542985916138
FFT convolution with edge detection kernel: 0.2323625087738037
Convolution with sharpen kernel: 4.410077333450317
FFT convolution with sharpen kernel: 0.23506712913513184
Convolution with gausian blur kernel: 4.426570177078247
FFT convolution with gausian blur kernel: 0.2345573902130127
```

# Conclusion

As seen from evaluation part, FFT based implementation of convolution is significantly more faster than ordinary one, which proves theory about convolution and FFT based convolution. As well as applying FFT optimisation on practice, I also deepen my knowledge about DFT, FFT, convolution and FFT based convolution.

# References


1) Portland State University. Discrete Fourier Transform and Fast Fourier Transform. https://web.pdx.edu/~daescu/mth428_528/hpc_fft.pdf
2) MathWorks, Cleve Moler and Steve Eddins, 2001. https://www.mathworks.com/company/technical-articles/faster-finite-fourier-transforms-matlab.html
3) Oxford University, Lecture 7- The Discrete Fourier Transform https://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf
4) Princeton University, Robert J. Vanderbei, Fast Fourier Optimization, 2010 https://vanderbei.princeton.edu/tex/ffOpt/ffOptMPCrev4.pdf
5) Cory Maklin, Fast Fourier Transform Explained, 08.02.2024, https://builtin.com/articles/fast-fourier-transform
6) 3Blue1Brown, But what is convolution?, 18.11.2022, https://www.youtube.com/watch?v=KuXjwB4LzSA