Applications of Fast Fourier Transform in image compression and encryption

Sorana Catrina, Mirela Catrina

January 30, 2024

Abstract

This project aims to explore the applications of the Fourier Transform in image processing, focusing on image compression and encryption. Image compression is firstly analysed only through the FFT, keeping only the most important information from the frequency domain, and then explored through the JPEG compression algorithm. Therefore, implementation of the discrete cosine transform is also discussed through the lens of the FFT and the relationship between the two. The second application discussed in this project follows safe image encryption and decryption with the FFT and three private keys. The implementation follows the proposal by Jaehun Song and Yeon Ho Lee [2] and we aim to clarify some implementation aspects that we feel were not properly discussed in the initial article.

Contents

I Theoretical Overview **1**
- i The Fourier Transform 1
- ii The Discrete Fourier Transform 2
  - ii.1 2D Discrete Fourier Transform . . . 2
- iii The Fast Fourier Transform 2
- iv The radix-2 DIT Implementation 2
  - iv.1 Implementation of FFT 3

II Compression **3**
- i Fourier Based Compression 3
- ii JPEG Compression using the DCT 4
  - ii.1 Compression methodology and implementation 5
  - ii.2 The Discrete Cosine Transform . . . 7
  - ii.3 Relation to the Fast Fourier Transform 7

III Image Encryption using the FFT **10**
- i Encryption 10
- ii Decryption 10

IV Conclusions **11**

I. Theoretical Overview

The Fourier Transform is a tool, or to be more specific an integral transform, that converts a function into a form that describes the frequencies present in the original function. To be more precise, it decomposes the initial wavefunction into the sines and cosines that compose the initial function.

The Fourier Transform is sometimes thought to be one of the most useful and ingenious tools used in mathematics and physics, and we find it in a multitude of domains: starting from sound analysis, to image compression and recently even in Quantum Computing.

i. The Fourier Transform

As we mentioned earlier, the Fourier Transform is an analysis process: it is used to decompose a complex-value function $f(x)$ into its constituent frequencies and amplitudes. Of course, the inverse transformation is meant to transform these frequencies and amplitudes into their sum. In continuous space, the Fourier Transform is the generalization of the complex Fourier series in the limit as $L \to \infty$. Formally, the transformation can be written as follows:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\xi x} dx \tag{1}$$

One can image this transformation as decomposing the initial wave signal into *circular paths*. Formally, this circular paths are represented by $e^{-i2\pi\xi x}$, since due to the Euler Formula we know that these functions trails out circles with corresponding frequencies. A good resource highlitghing this property can be found at betterexplained [1]

The keen eye will wonder about whether we can really decompose *any function* in said frequencies and amplitudes. And this is a good question: can we, for example, decompose a step function into these circular components? Even though strange at first, the answer is yes, we can. We will also point out shortly that while the functions we usually encounter are continous function, and therefore integrable functions, the definition of the Fourier transform also holds for *measurable functions* (also called Lebesgue Integrable function).

The inverse Fourier transform has a similar form:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(\xi) e^{i2\pi\xi x} d\xi \tag{2}$$

[1] Better explained: an interactive guide to Fourier transforms: https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/

Whether we choose the positive exponent for the direct transform or for the inverse transform is up to the user of the transform. Here we chose to go with the convention that the direct transform has negative exponents.

ii. The Discrete Fourier Transform

In discrete space, the Fourier transform converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier Transform, which is a complex valued function (we end up with the exact number of points as in the original data due to the nyquest frequency, or nyquest-shannon sampling theorem). Therefore, for a sequence $\{x_n\} := x_0, x_1, ..., x_{N-1}$ we define the Fourier Transformed sequence the sequence of N complex values $\{X_n\} := X_0, X_1, ..., X_{N-1}$, defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n} \tag{3}$$

The inverse transform is therefore:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi \frac{k}{N} n} \tag{4}$$

We will also point out that there are different ways of normalizing this function. Here, we chose "backward" normalization, meaning we added the term $1/N$ to the inverse transform. This term can be added to the direct transformation in the case of forward normalization, or the term $1/\sqrt{N}$ can be added to both transforms in the case of orthonormal transformation (which is less common).

ii.1 2D Discrete Fourier Transform

In case we want the 2D Fourier Transfrom, the formula becomes:

$$f_{(u,v)} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m,n) e^{-i2\pi(\frac{k}{M}m + \frac{l}{N}n)} \tag{5}$$

We point out that in our application, this is the formula that we need: we want to analyze the applications of the 2D DFT in Image Processing. We can observe that the two functions are separable, meaning if we want the 2D Fourier Transform of an image for example, we can break down the steps as follows:

1. Compute the DFT in one direction (apply the original formula on rows)

2. Compute the DFT on the columns of the resulted matrix

For out implementation, we chose to follow this rule.

iii. The Fast Fourier Transform

Since for the 2D DFT we want to be able to apply the DFT on one direction sequencially (meaning we want to apply it on rows and then on columns) the question becomes: but how are we going to implement the Discrete Fourier Transform on a given vector?

The first naive implementation would be direct implementation of the formula: for each point we go through all the points in our original data and sum up the products of our values and our complex exponent. However, a downfall can be easily detected: the time complexity. Since for each point we go trough all of our data, we end up with complexity scaling quadratically $O(n^2)$.

Therefore, we feel the need of a more efficient implementations. We will highlight a popular algorithm for performing the Fast Fourier Transform: *the Cooley-Tukey FFT*. This is the most common class of FFTs, and the main idea behind it is re-expressing the DFT of an arbitrary size N in terms of two smaller DFTs, recursively, in order to reduce the computation time to logarithmic scale $O(NlogN)$. There are multiple implementation of this class of algorithms, therefore we chose to focus on the most common and widespread one: the radix 2 DIT implementation.

iv. The radix-2 DIT Implementation

The main idea behind the radix-2 DIT (decimation in time) algorithm is to devide the vector of size N into two interleaved (hence the name radix-2) vectors of half the size, at each recursive step. We emphasize the primary steps for finding out the formula:

- Start from the usual representation of the DFT formula and split the function x_n into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m + 1$.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n e^{-i2\pi \frac{k}{N} n} \\ X_k &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N} 2mk} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \end{aligned} \tag{6}$$

- Factor out one *twiddle factor* and rewrite in terms of Even sums and Odd sums

$$\begin{aligned} X_k &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk} + e^{-\frac{2\pi i}{N} k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk} \\ X_k &= E_k + e^{\frac{-2\pi i}{N} k} O_k \end{aligned} \tag{7}$$

- Use the periodicity of the complex exponential to find out the formula for the second half of the array:

$$X_{k+\frac{N}{2}} = E_k - e^{\frac{-2\pi i}{N} k} O_k \tag{8}$$

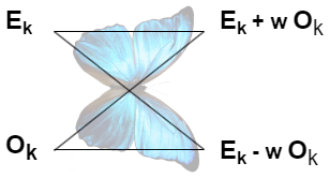

Figure 1: *Butterfly algorithm*

Therefore, summarising the formulas presented earlier, we can express the DFT of length N recursively in terms of two DFTs of size N/2 as follows. We point out that this is the core of the radix-2 DIT fast Fourier Transform, and that the algorithm gains its speed by reusing the results of intermediate computations to compute multiple DFT outputs.

$$\begin{aligned} X_k &= E_k + e^{\frac{-2\pi i}{N}k} O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{\frac{-2\pi i}{N}k} O_k \end{aligned} \tag{9}$$

We can therefore notice that the outputs are obtained by +/- combinations of E_k and O_k, which is simply a size-2 DFT. This is sometimes called a butterfly-type schema, since the graphic representation resembles one 1

iv.1 Implementation of FFT

Although we know have reduced the theoretical time complexity, we can take the efficient implementation one step further. In order for the point to become clearer, we will point out the currently discussed implementation in pseudocode 1.

Algorithm 1 Recursive FFT

```
Input: the input vector v
Output: the fourier transform of the input vector
  function FFT_RECURSIVE(v)
    N ← length(v)
    if N <= 1 then return x
    end if
    twiddle_factor ← exp(−2jπ/N)
    omega ← 1
    even_part ← fft_recursive(v[0 :: 2])        ▷ 1
    odd_part ← fft_recursive(v[1 :: 2])         ▷ 2
    for k in range(N//2) do
      x[k] ← even[k] + omega ∗ odd[k]
      x[k + N//2] ← even[k] − omega ∗ odd[k]
      omega ← omega ∗ twiddle_factor
    end for
  end function
```

From this, we can see that while the time complexity is $O(NlogN)$, our implementation is slowed down by recursive calls and copies of each vector (7, 8). Therefore, we want to strive for an implementation with the same time complexity, but which does the calculations in-place. For this, we firstly observe the pattern that emerges from the vector, in Figure 2. We can see that if we ordered the vector in the order given by the last level of recursion we would be able to implement the calculations in place. But *what is this order?*. It turns out it is *the bit reversal permutations*, but in order to fully understand this we will look closely at what exactly happens at each recursive level in the 8-point radix-2 DIT implementation.

1. The first step is splitting the data based on even indices and odd indices. But, *how do our vectors actually look like?* Well, looking at the bit representation, the first half of the newly form vector has all the indices that have the following form in binary representation: $xx0$. The odd indices are of course represented by $xx1$.

2. One step furhter, we see that the now even indices are also split in two:

   - The "even" indices of the even part: $x00$
   - The "odd" indices of the even part: $x10$

By this, we can see a pattern emerging. That being, the first half of the final permutation has the last bit 0, and the second to last bit change value every N/4 times. The bit before this one, changes every N/8 steps. By looking closely at this, we see that this is exactly the *reverse* of what happens in our indices. This is therefore the reason we need the bit reversal. The algorithm is presented:

Another step for improvement in the case of multiple FFTs being perfomed on vectors of the same size (as in the case of our image) could be to calculate the bit reversal permutation only once and storing it into a separate array. This would result in the bit reversal to only be calculated once.

II. Compression

Compression involves the reduction of the size of an image file while preserving essential visual information, aiming to strike a balance between data compression and image quality. By minimizing the storage space required and facilitating faster transmission across networks, image compression not only optimizes resource utilization but also plays a fundamental role in various applications, including multimedia, web content, medical imaging, and remote sensing. Different compression schemes can be used in order to preserve as much information and also reduce the storage needed.

i. Fourier Based Compression

Fourier based compression refers to the compression technique that is based on the Fourier Transform. This idea written in a concise manner is as follows:

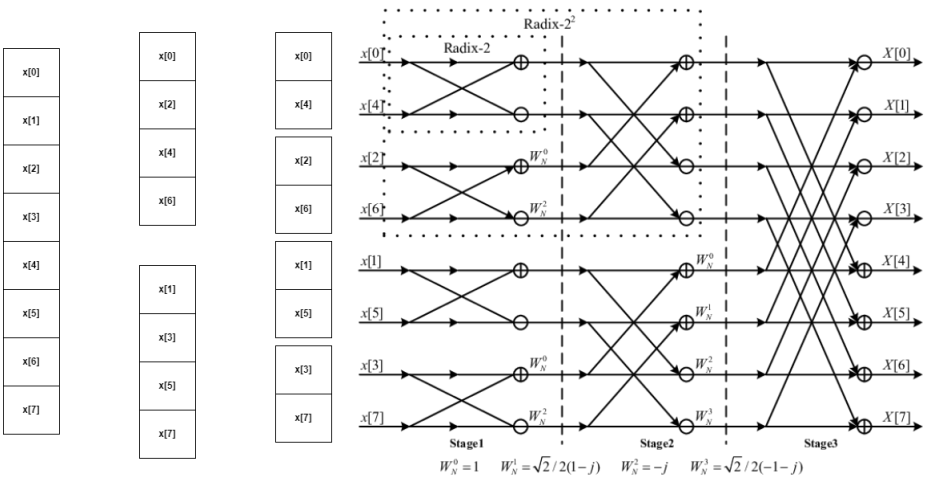

Figure 2: *8 point FFT*

Algorithm 2 Efficient FFT

Input: the input vector v
Output: the fourier transform of the input vector

```
function FFT(v)
    N ← length(v)
    logN ← log(N)
    ▷ Bit reverse permutation
    for i in range(N) do
        if i <= reverse(i) then
            swap(v[i], v[reverse(i)])
        end if
    end for
    ▷ Start implementing the FFT
    len ← 2
    while len <= n do
        twiddle_factor ← exp(−2jπ/N)
        for i in range(N) do
            w ← 1
            for j in range(N//2) do
                e ← v[i + j]
                o ← v[i + j + len/2] ∗ w
                ▷ Butterfly algorithm
                a[i + j] ← e + v
                a[i + j + len/2] ← e − v
                w ← w ∗ twiddle_factor
            end for
        end for
        len ← len ∗ 2
    end while
end function
```

1. Apply the 2D Fourier Transform to an image
2. Keep only the top x percent highest values (compared by magnitude, since these are complex values)
3. When decompressing the image, apply the Inverse Fourier Transform to the stored values.

This compression technique is part of a more primitive type of compressions, and while it saves a lot of space, we can observe different artifacts in the retrieved images due to this technique.

Figure 3: *Left: the original image (16MB), Right: the compressed image (2.39MB)*

ii. JPEG Compression using the DCT

JPEG (the short form of Joint Photographic Experts Group) image compression is one of the most widely used methods for lossy compression. It typically achieves a 10:1 compression rate with little perceptible feature loss compared to the original image. It is particularly well-suited for images in the web developing community, given the small image size that leads to faster loading times, contributing to a smoother user experience and improved SEO (Search Engine Optimization) ranking.

Compression methods are either lossy or lossless. Lossless compression usually constitutes encoding algorithms (i.e. Huffman encoding) that store all the information in the image in a condensed form, that can be decoded

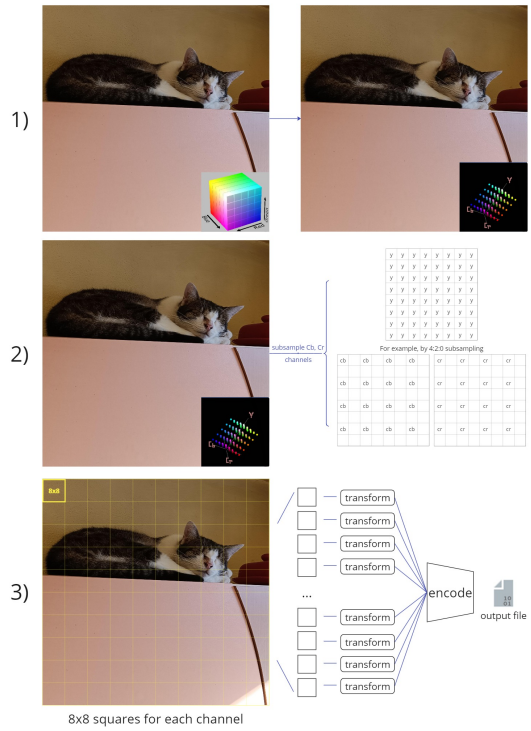

Figure 4: *JPEG compression steps as follows: 1) color space conversion, 2) subsampling (optional), 3) splitting and processing MCUs (minimum coded units) blocks individually and then encoding them together*

at open and reconstruct the image without loss of data. Well-known lossless compression formats include *.png*, *.tiff*, *lossless .webp* for images, but it is mostly used in text and data files. On the other hand, lossy compression lies on the underlying statistic data present in images, discarding visually unidentifiable details and leading to higher compression rates.

JPEG compression steps are defined in the ISO/IEC 10918-7:2023 standard (initially defined in 1992 by the ISO/IEC 10918-1:1994 standard) and T.873 (06/21) ITU-T records (initially, T.81 (09/92)).

ii.1 Compression methodology and implementation

The compression methodology can be visually explained using the figure 4.

Now let's explain each step.

Firstly, we transform the image from the RGB format to the YCbCr format. The reason for this transformation is tightly related to the human eye: we are more sensitive to changes in intensity, or luminosity, rather than color itself. The YCbCr takes advantage of this by separating the luminescence component (Luma, Y) and the chrome components (Cb, Cr).

Now, in the second step from the 4 we expand on the advantages of the YCbCr format stated above. Thus, we will keep all the information from the Luma channel at this step, since subsampling from it will lead to visible loss of data. On the other hand, subsampling from the Cb and Cr channels will not be visible to the human eye, and therefore we can apply subsampling. Subsampling can be of various forms, encoded in the $J : a : b$ notation.

- J is representative of the total number of pixels in the horizontal sampling region and most cases will be 4.
- a is the number of pixels sampled amongst the first row of pixels as defined by J.
- b is the number of pixels sampled amongst the second row of pixels as defined by J.

Various subsampling triplets can be used, but the most widely-used one is $4 : 2 : 0$. Some examples are illustrated below:

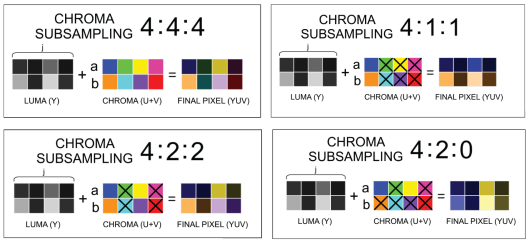

Figure 5: *Subsampling triplets examples*

The third step illustrated in figure 4 is composed of multiple steps. Firstly, we split each channel, individually, into 8x8 blocks, referred to as MCUs (minimum coded units). Each block will be transformed independently (and therefore the transform process may be parallelized). Given that any smaller matrix is considered to not have enough information for compression, thus leading to a low quality compressed image, and that any larger matrix may be harder to operate fast on (hardware not optimized for it, transforms taking longer time), the 8x8 size is considered optimal.

First, let's discuss the transformation applied to a MCU. We will use the figure 6 as support.

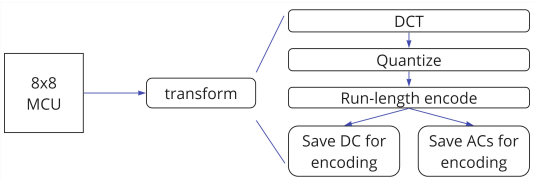

Figure 6: *Subsampling triplets examples*

The first step of the transform consists of passing the 8x8 MCU through the DCT transform and retrieving the resulting matrix. This step is explained in-depth in Section ii.2.

The quantization takes advantage of the fact that the DCT transform will have the low frequencies stored in the top-left corner and the high frequencies, that we can get rid of, towards the bottom-right corner. Quantization is, simply explained, just multiplying the resulted matrix with hard-coded quantization matrices obtained by JPEG

through intensive testing. Therefore, we display the most-used matrices used in jpeg quantization:

$$Q_{Luma} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 36 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$Q_{Chrome} = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

Thus, the matrix before the run-length encoding is a sparse matrix that only has some non-null elements in the top-left corners, obtained by:

$$QuantizedMatrix_{chnl} = DCT(MCU)/Q_{chnl},$$
$$chnl \in \{Y, Cb, Cr\}, Q_{Cb} = Q_{Cr} = Q_{Chrome}, Q_Y = Q_{Luma}$$

Now we are ready for the Run-length encoding. The run-length encoding (RLE) step plays a crucial role in efficiently representing and storing image data. It focuses on minimizing redundancy within the values obtained from the previous step. RLE operates by identifying consecutive sequences of identical values, or "runs", and replacing them with a compact representation that specifies the value and the length of each run. This process effectively reduces the amount of data needed to represent the image, optimizing storage and transmission efficiency. By condensing repeated information, run-length encoding contributes to the overall compression of JPEG images, alongside the previously presented steps.

RLE takes advantage of the form of the *QuantizedMatrix*: given, the top-left corner with few values, and in rest mostly zeros. Therefore is utilizes a zig-zag scan that encodes values in a 64-value array. The zig-zag scan is illustrated in Figure 7:

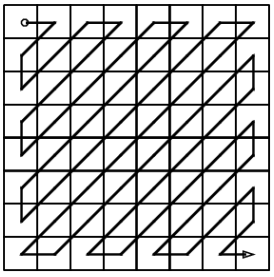

Figure 7: *Zig-Zag scan on a 8x8 matrix*

Then, this sequence is compressed by removing consecutive subsequences of zeros and incorporating the necessary data in the remaining values. By that, we map all non-null values:

$$EE : \mathbb{N}^* \to \mathbb{N}^3, E(x) = (l, b, v)$$

Representing:

- l = run length, the number of consecutive zeros that precede the current non-zero element in the zig-zag scan output array
- b = the bit count, the number of bits used to encode the amplitude value that follows, as determined by the Huffman encoding scheme, we will detail this later.
- v = the amplitude, the amplitude of the quantized DCT coefficient.

For now, we could simplify the triplet into the pair (l, v) (our implementation of the encoding only utilizes this pair). Thus, given the 64-sized input array, we will now have a much shorter array with the length of the number of non-null elements, without loss of data (their exact positions are encoded using the l element, that can transfer each value into its position in the array and, consequently, the matrix). The ending of the encoding of an initial 64-sized input array encoding will always be marked by the triplet $(0, 0, 0)$.

This array will be encoded using the Huffman Tree and encoding algorithm. The Huffman Tree will associate a bit with each pair. The most important element to note is that more frequent pairs will have shorter corresponding bit arrays, thus condensing the saved bit array. Also DC (Direct Current) values (the values at $(0, 0)$ position in the matrix) and AC (Alternate Current) values (the other 63 values) will be stored separately. There will be 4 trees:

- **Luminence DC** - The DC values from all the Luma MCUs
- **Luminence AC** - The AC values from all the Luma MCUs
- **Chroma DC** - The DC values from the Cb and Cr MCUs, combined
- **Chroma AC** - The AC values from the Cb and Cr MCUs, combined

After pushing each pair to the Huffman encoding, we will save the resulting bit array in the final file, alongside the 4 trees (that we will need for decoding). The encoded jpeg file will contain the elements represented in figure 8.

Huffman coding consists in the following steps:

- Sort elements (i.e. triplets) according to frequency and store them in a priority queue.
- Build the tree by getting the top two nodes, called children, by frequency. Unite them in a new node. The node's frequency will be the sum of the selected 2 children and will have the left and right children as selected.

- Insert the new node into the priority queue.
- Do this until there is only one element left in the queue, the root.

The codings will then be obtained by traversing the tree, noting a new 0 when going left on a child and 1 when going right. We note that all the actual triplets will be the leaves of the tree and therefore when decoding a bit array we will read it bit by bit until we reach a leaf. The leaf is the triplet in line, and we will continue the reading and traverse the tree again, from the root.

The coding can be done also in a more space and time-efficient way. We can code so that the sequence of code words (identified by numbers via their binary digit expressions) is increasing: forming consecutive numbers when the code length is unaltered and adjoining zeros when the code length increases. In JPEG a code word must not consist of only 1's. We can avoid this by adding provisionally an extra value whose frequency is half (for instance) of the frequency of the last and least value (and finally remove a code from the codes of the largest length).

Decoding is therefore straightforward.

We read the bit arrays, decode them using the read trees. We then process each obtained MCU in the order we saved them in the file, by first de-quantizing it (using the specified matrix) and then applying IDCT to it. We then put the MCU values in the image we will output. Once the MCUs are all decoded we can display the image.

Results of our implementation of the jpeg can be observed in figure 9.

Compared to our FFT implementation, we can see a higher compression rate (also, our compression rate is different from the JPEG due to the simplified flow and simplified binary data we used and saved; for example, we do not save the quantization tables, nor the application header and other details). Also, we can observe that there is no visible loss to the compressed image, as opposed to the FFT compression where artifacts were visible.

ii.2 The Discrete Cosine Transform

The Discrete Cosine Transform (DCT) is a Fourier-related transform that is widely used in signal processing and data compression. Similat to how the DFT expresses a finite dequence of data points in terms of a sum of cosines and sines (that is, complex exponentials) the DCT expresses the data in terms of a sum of cosine functions oscillating at different frequencies. An important distinction regarding the transformed vector is that while the DFT returns a complex-valued series of numbers representing amplitudes and phases, the DCT returns a sequence of *only real numbers*. Mainly, there are 8 standard DCT variants, out of which 4 are most used. Here, we will use the type-II DCT, which sometimes is simply called the DCT. With that being said, suppose we have a sequence of N real numbers$\{x_n\} := x_0, x_1, ..., x_{N-1}$. The discrete cosine transform of this sequence is another sequence of N *real valued* elements $\{X_n\} := X_0, X_1, ..., X_{N-1}$. calculated based on the following formula (Equation 13

$$\begin{aligned} X_k &= \alpha_k \sum_{n=0}^{N-1} x_n cos(\frac{\pi}{N}(n+\frac{1}{2})k) \\ \alpha_0 &= \frac{1}{\sqrt{N}} \\ \alpha_k &= \frac{2}{\sqrt{N}} \end{aligned} \tag{10}$$

The inverse transform is given by the type-III DCT:

$$x_n = \frac{1}{\sqrt{2}}X_0 + \sqrt{\frac{2}{N}} \sum_{k=0}^{N-1} X_k cos(\frac{\pi}{N}(n+\frac{1}{2})k) \tag{11}$$

Expanding on this formula, we easily arrive at the DCT2D.

$$\begin{aligned} X_{pq} &= \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x_{mn} cos\frac{\pi(2m+1)p}{2M} cos\frac{\pi(2n+1)q}{2N} \\ 0 &\le p \le M-1 \\ 0 &\le q \le N-1 \end{aligned} \tag{12}$$

$$\alpha_p = \begin{cases} \frac{1}{\sqrt{M}} & \text{p=0} \\ \sqrt{\frac{2}{M}} & 1 \le \text{p} \le M-1 \end{cases}$$

$$\alpha_q = \begin{cases} \frac{1}{\sqrt{M}} & \text{q=0} \\ \sqrt{\frac{2}{M}} & 1 \le \text{q} \le N-1 \end{cases}$$

In order to better grasp the concept of the DCT and of the basis functions that are used for the decomposition we can take a look at the basis functions represented as a 8x8 matrix (here we use 8x8 since the DCT in our case is applied to blocks of size 8).

For generating these basis function we simply implemented the basis functions as given in Equation 12. The code concisely given with the use of the pseudocode in algoritm 3

One example of how the algorithm decomposes an image into its representative functions is given in Figure 11. We can see how the DCT of the final image reveals the frequencies that represent the basis functions presented.

ii.3 Relation to the Fast Fourier Transform

With the scope of implementation in mind, we will remind of the relation between the FFT and the DCT. While multiple efficient implementations of the DCT are available (that is, for example, for a 8x8 block, a matrix is often used to represent the result of the transform) we chose to implement the DCT based on the FFT. Therefore, it is essential to represent the direct and reversed DCT based on the FFT. The equations describing these are as follows [1]:

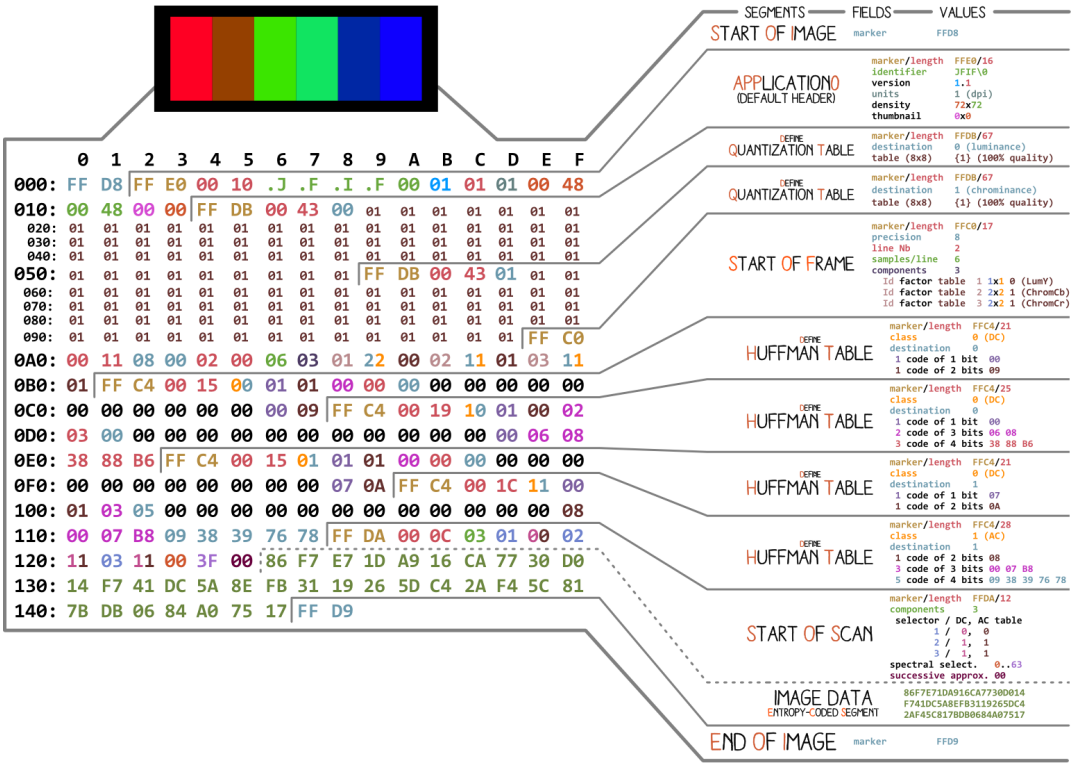

Figure 8: *Dissected JPEG format*

Algorithm 3 Basis functions formation DCT

$matrices = zeros((8,8,8,8))$
for `u, v - pozitiile din grid` **do**
  $basis\_function \leftarrow zeros((8,8))$
  for $x \leftarrow$ `0, 8, 1` **do**
    for $y \leftarrow$ `0, 8, 1` **do**
      $alpha_u \leftarrow \sqrt{\frac{2}{8}}$ if u==0 else $\sqrt{\frac{1}{8}}$
      $alpha_v \leftarrow \sqrt{\frac{2}{8}}$ if v==0 else $\sqrt{\frac{1}{8}}$
      $basis\_function[x,y] = alpha_u \cdot alpha_v \cdot \cos\frac{(2x+1)u\pi}{16} \cdot \cos\frac{(2y+1)v\pi}{16}$
    end for
    $matrices[u,v] = basis\_function$
  end for
end for

Figure 9: *Left: the original image (16MB), Right: the compressed image (370KB binary)*

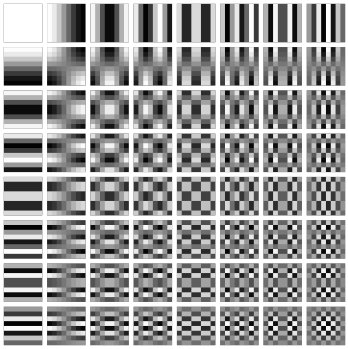

Figure 10: *Basis functions for blocks of size 8x8*

Theorem II.1 (Direct DCT) *Let $y = D_N x$ represent the N point DCT for the vector x. Then, the dct is represented as*

$$y_n = c_{n,N} \left(\cos\left(\pi \frac{n}{2N}\right) R((F_N x^{(1)})_n) + \sin\left(\pi \frac{n}{2N}\right) I((F_N x^{(1)})_n) \right) \tag{13}$$

Where $c_{0,N} = 1$ and $c_{n,N} = \sqrt{2}$ for $n \geq 1$ and $x^{(1)} \in \mathbb{R}^N$ is defined by:

$$(x^{(1)}))_k = x_{2k} \text{ for } 0 \leq k \leq N/2 - 1$$

$$(x^{(1)})_{N-k-1} = x_{2k+1} \text{ for } 0 \leq k \leq N/2 - 1$$

Theorem II.2 (Inverse DCT) *Let $x = (D_N)^T y$ be the IDCT for y. Let z be a vector with the 0th component $\frac{1}{c_{0,N}} y_0$ and the rest with $\frac{1}{c_{n,N}} e^{\pi i n/(2N)}(y_n - i y_{N-n})$. Then:*

$$x^{(1)} = (F_N)^H z, \tag{14}$$

Where $x^{(1)}$ is defined the same as previously.
We also highlight that in this implementation, the FFT is present in the orthonormal form, as opposed to the backward normalized form we've been working with so far. The algorithms are also present in Algorithm 4 and Algorithm 5 respectively. These algorithms follow directly from the formulas. We used different functions considered to already be available, such as a function that returns the real part and a function that returns an imaginary part of a given vector.

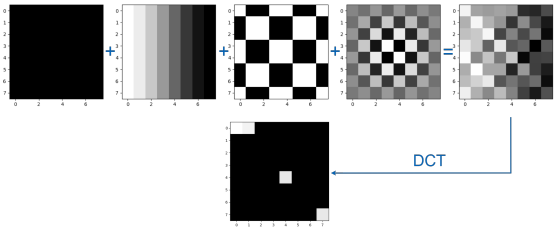

Figure 11: *DCT Decomposition of example image*

Algorithm 4 DCT based on FFT

Input: the input vector v
Output: the dct transform of the vector
function DCT_IMPL(x)
$N \leftarrow length(x)$
if N <= 1 **then return** x
end if
$x\_reorg \leftarrow horizontal\_stack((x[0::2], x[::-2]))$
$y \leftarrow fft(x\_reorg)$
$rp \leftarrow real(y)$ ▷ gets the real part
$ip \leftarrow imag(y)$ ▷ gets the imaginary part
for i *in* $range(N)$ **do**
$angle \leftarrow i\pi/(2N)$
$y[i] \leftarrow cos(angle) * rp[i] + sin(angle) * ip[i]$
end for
$y[1:] \leftarrow y[1:] * \sqrt{2}$
end function

Algorithm 5 IDCT based on FFT

Input: the input dct vector y
Output: the idct transform of the vector
function IDCT_IMPL(y)
$N \leftarrow length(y)$
if N <= 1 **then return** y
end if
$z = zeros(N)$ ▷ init with 0+0i
$z[0] \leftarrow y[0]$
for i *in* $range(1, N)$ **do**
▷ The constant
$z[i] \leftarrow exp(i\pi 1j/(2N))/\sqrt{2}$
▷ The part depending on y
$z[i] \leftarrow z[i] * (y[i] - 1j * y[N-i])$
end for
$x \leftarrow ifft(z, 'orthonormal')$
for i *in* $range(0, N, 2)$ **do**
$x[i] = real(z[i/2])$
end for
for i *in* $range(1, N, 2)$ **do**
$x[i] = imag(z[N - i/2 - 1])$
end forreturn x
end function

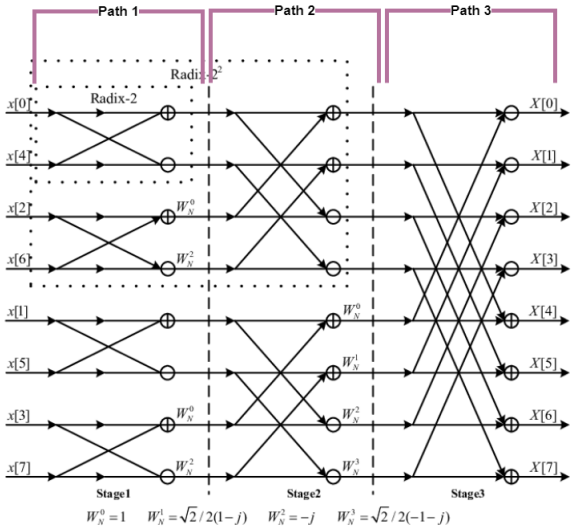

Figure 12: *Paths present in the 8 point FFT*

III. Image Encryption using the FFT

The second application we explored revolved around image encryption. More precisely, we focused on the proposal by Jaehun Song and Yeoun Ho Lee [2]. They proposed an encryption schema using 3 private keys: the path from the FFT butterfly algorithm, number of right bit shifts, a number M' with which different twiddle factors are composed and used. They were able to prove the robustness of this encryption keys and therefore the efficiency of the FFT in yet another application.

We aim to implement the proposal and further explain some details that we found a bit harder to grasp. In order to do this, we must define the first element of the key: *the path*.

The path basically represents the level of recursion our encryption is targetting.

i. Encryption

With this in mind we are ready to explain the main idea of the encryption: at the given path, we apply 2 different operations:

- Right bit shift: we apply right bit shifts in an iterative manner (the number of iterations is given by the second key) to the indices of the result.
- Apply an extra twiddle factor multiplications. This twiddle factor is defined in our diagrams as $\tilde{w}$ and is calculated: $\tilde{w} = e^{-\frac{i2\pi}{M'}}$, where M' is the third key.

In order to better emphasize the steps used in the encryption phase of the algorithm we illustrated the example of a 8-point radix-2 DIT implementation in Figure 14. Therefore, the two main steps can be identified: the indices reordering by bit shifting and the multiplication with the extra twiddle factor. We remind that M' used by our algorithm can be any integer, and the choice of this integer determines the *extra twiddle factor*. An example of how such encryption looks is presented in Figure along with the three keys.

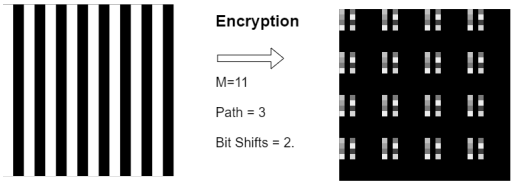

Figure 13: *Encryption example*

ii. Decryption

Decryption is done by applying the inverse procedure on the encrypted image. By this we mean that if we encrypted at path *path*, the decryption will be done at path $log(N) - path$. This is the part the we feel was harder in this application.

Simply put, the decryption has the following steps:

- At step $log(N) - path$, remove the extra twiddle factor (we will add this later, to the correctly ordered array). This is done by grouping the results into groups of length $path^2$ and dividing by $(w^{group\_perm})^{index\_in\_group}$. This is the generalization of the example explained in the initial article.
- After this step, we apply a bit reversal permutation. This is done in order to be able to remove the *extra twiddle factor*, since after the bit reversal, the extra twiddle factors will be in order of increasing power.
- Remove the extra twiddle factor by dividing with the factor raised to the corresponding power.
- Revert the right bit shifts by left bit shifts in order to correctly rearrange the data
- Reorder data by reverse bit shift (this is done in order to undo the reordering done earlier in order to help us remove the extra twiddle factor)
- Add the twiddle factors that were removed to the now correctly order array
- This will become the new data for the next recursion step

A decryption example for a 16-point radix-2 DIT is presented in the original article, so here we chose to represent the 8-point radix-2 DIT in Figure 15. The last three steps are from our decryption are not presented in the image. The decryption works only if the right keys are used. This is also demonstrated in the original paper, in which the robustness of the algorithm is also discussed. Here, we applied the decryption algorithm on the previously encrypted image with correct keys and with one wrong key each. The results are presented in Figure 16.

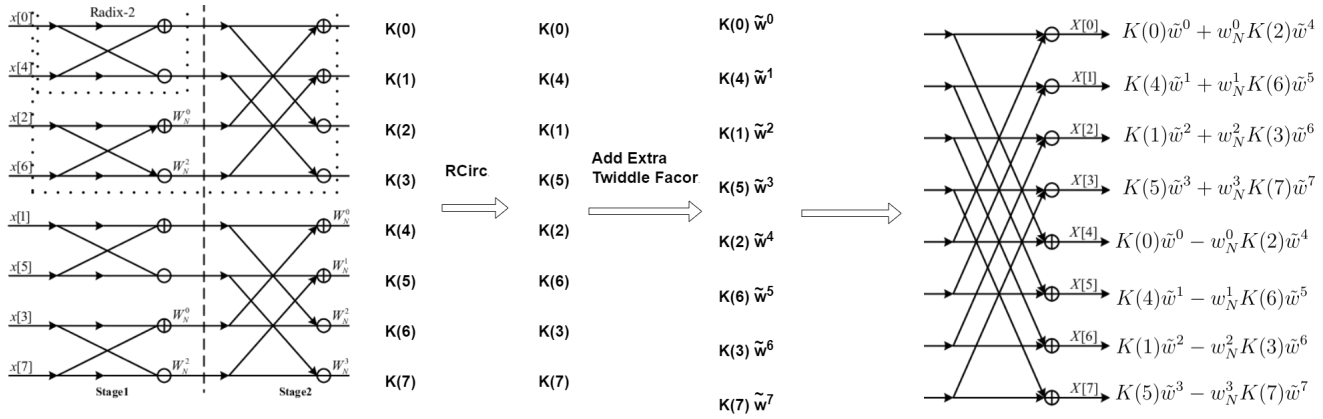

Figure 14: *Encryption scheme for path=1 and 2 right bit shifts*

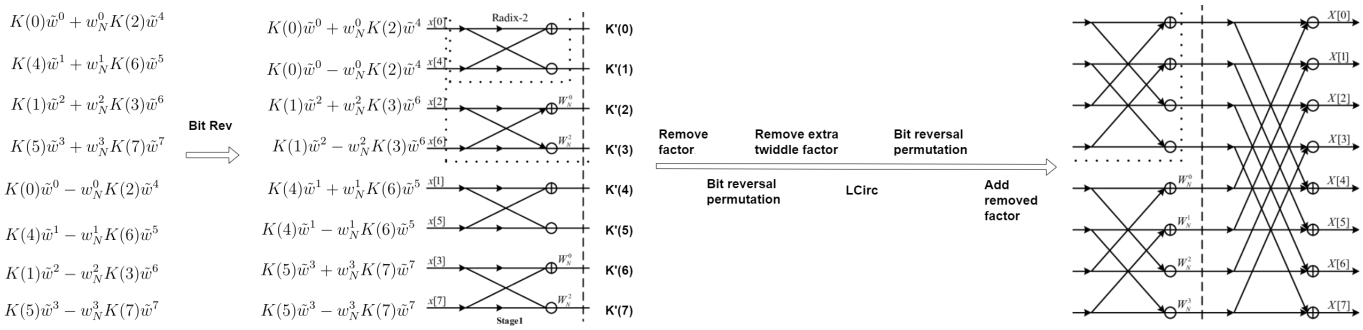

Figure 15: *Decryption diagram*

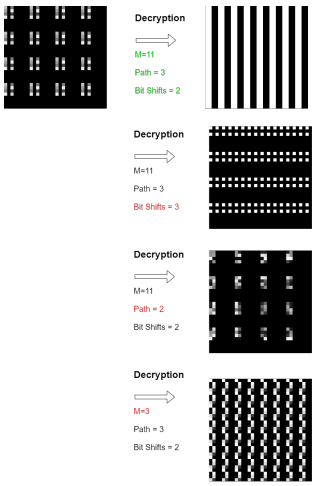

Figure 16: *Decryption with right and wrong keys*

IV. Conclusions

In this project we explored different applications of the Fourier Transform in image processing. We started with an examination of image compression through this lens of the FFT. We discussed different implementations of the FFT and also spotlighted one of the most popular compression algorithms for images: the JPEG compression. Within this context, we pointed out the main steps involved and also emphasized how FFT can play a role in implementing the discrete cosine transform - a fundamental element of the JPEG compression. Transitioning our focus, we shifted towards image encryption, revealing yet another facet of FFT's applicability. Our exploration led us to an ecryption scheme that uses 3 keys, one designating the path (the step in the recursion), another representing the extra twiddle factor and a supplementary integer indicating the bit shifts.

> Mathematics compares the most diverse phenomena and discovers the secret analogies that unite them.
>
> *Joseph Fourier*

References

[1] MAT-INF2360 – Applications of Linear Algebra. *Chapter 4 - Implementation of the DFT and the DCT*.

[2] Jaehun Song and Yeon Ho Lee. "Optical image encryption using different twiddle factors in the butterfly algorithm of fast Fourier transform". In: *Optics Communications* 485 (2021), p. 126707. ISSN: 0030-4018. DOI: https://doi.org/10.1016/j.optcom.2020.126707. URL: https://www.sciencedirect.com/science/article/pii/S0030401820311263.