Image compression analysis with FFT algorithm

André Andrade Gonçalves
Instituto Tecnológico da Aeronáutica
São José dos Campos, Brazil
Email: andre.goncalves.8750.ga.br

I. Introduction

Image compression plays a fundamental role in the viable and efficient transmission, storage, and manipulation of visual data. In this context, a powerful and innovative signal processing technique stands out, responsible for revolutionizing algorithms for compression, analysis, and processing of images, audio, and signals in general: the Fourier transform. This project aims to explore the implementation and application of the algorithm called Fast Fourier Transform (FFT), which efficiently performs the Fourier transform, delving into the fundamental principles that make this algorithm a key component in optimizing image compression techniques.

Developed by James Cooley and John Tukey in the 1960s, during the Cold War and the early development of nuclear technologies, this algorithm aimed to analyze seismograph signals to detect anomalies caused by nuclear tests and monitor the development of Soviet warheads. The existing techniques using the Fourier transform to analyze obtained frequencies were powerful but slow and computationally expensive, as it was an $O(n^2)$ algorithm, making them inefficient [3]. Cooley and Tukey's technique sped up this process by developing an $O(n \log n)$ algorithm, called the Fast Fourier Transform (FFT).

Thus, using the FFT, in an image[1], it is possible to transfer pixel information to a new complex abstract space, commonly referred to as the frequency domain. An empirically discovered property of this domain is that most information related to high frequencies has low absolute values. Omitting these frequencies results in little perceivable alteration in the original image, mainly because most naturally occurring or extracted images have little pixel-to-pixel variation, and high-frequency representations, usually associated with unwanted noise, are unnecessary.

By filtering out values with lower magnitudes in the frequency domain, it is possible to remove a significant portion of the image while retaining most of the relevant information from the original image. The compression method uses this approach to store and transfer data from the reduced image in the frequency domain. For the visualization of the original image, it performs the inverse transform to present the compressed image in the real domain.

This project[2] aims to study the functioning of these compression methods under different compression factors and different image sizes using a custom implementation.

[1]Naturally, the FFT algorithm operates only on one-dimensional vectors, but it is possible to extend this result to images by performing two FFTs, one for each dimension, successively.

[2]The link to the project repository can be found at the following link: gitlab.com/omisso/image_compression_cm203_2023.

II. The Discrete Fast Fourier Transform

The Fourier Transform of a given function $f(x) : \mathbb{R} \to \mathbb{R}$ is defined by Equation 1 [2], however, for most applications of this technique in engineering and signal analysis, the data comes discretely, so a new definition is necessary to address such situations. Thus, we define the discrete Fourier transform of a set of N values, as it shows in Equation 2 [3], which will also be an array of N values.

$$F(\xi) = \int_{-\infty}^{\infty} f(x) e^{-2\pi i \xi x} dx \tag{1}$$

$$X_\xi = \sum_{t=0}^{N-1} x_t e^{-2\pi i \xi t / N} \tag{2}$$

Therefore, it is possible to transform a discrete list of values into its respective Fourier transform and then perform the necessary operations in the represented abstract domain. Finally, it is generally necessary to return to the original vector domain, so it is necessary to employ the action of the inverse Fourier transform, which operates on X_ξ to return x_t, which can be calculated through Equation 3. Note that the inverse transform is similar to the normal transform, with the only change being the sign in the exponent and the division by the vector's size, so it is possible to use the same procedure, with slight modifications, for both the calculation of the Fourier transform and its inverse.

$$x_t = \frac{1}{N} \sum_{\xi=1}^{N-1} X_\xi e^{2\pi i \xi t} \tag{3}$$

Note that it is possible to express the relationship between the vectors of the transform $\vec{X} = [X_0, X_1, \cdots, X_{N-1}]^T$ in relation to the input vector $\vec{x} = [x_0, x_1, \cdots, x_{N-1}]^T$, according to a matrix linear transformation, as indicated in Equation 4, where $\omega = e^{-2\pi i/N}$. However, note that to perform both the transform and its inverse, matrix multiplication and, in the latter case, matrix inversion operations are required. These operations are computationally complex with a complexity of $O(n^2)$ [5], meaning they generally demand significant computational power and time.

$$\begin{bmatrix} X_0 \\ X_1 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \cdots & \omega^{(N-1)^2} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix} \quad (4)$$

However, [1] showed that it is possible to use a procedure capable of reducing the complexity of these calculations to $O(n \log n)$, based on a recursive process that extensively utilizes the periodicity of complex exponentials.

A. Recursive Algorithm

Assuming initially that N is a power of two with $N \geq 1$, such that, by separating the sum into terms of even and odd indices, it is possible to simplify the relationship and find the result presented in Equation 5. Thus, we express the ξ-th term transformed as a function of the transform of two other sequences: $x_{even} = [x_0, x_2, ..., x_{N-2}]$ and $x_{odd} = [x_1, x_3, ..., x_{N-1}]$. This establishes a recursive relationship for the calculation of the discrete Fourier transform, with the base case being the transform of a sequence with a single element $X_0([x_0]) = x_0$.

$$X_\xi = \sum_{k=0}^{N/2-1} x_{2k} e^{-2\pi i \xi k/(N/2)} + e^{-2\pi i \xi/N} \sum_{k=0}^{N/2-1} x_{2k+1} e^{-2\pi i \xi k/(N/2)} \quad (5)$$

Thus, it is possible to determine the recursive relationship from degree N to degree $N/2$, following Equation 6. On the other hand, an even greater simplification allows for the simultaneous calculation of the transform of two terms. Due to the periodicity of the complex exponential term, it is possible to demonstrate the result presented in Equation 7, further expediting the calculation of the transform associated with a sequence.

$$X_\xi = X_{even,\xi} + e^{-2\pi i \xi/N} X_{odd,\xi} \quad (6)$$

$$X_{\xi+N/2} = X_{par,\xi} - e^{-2\pi i \xi/N} X_{impar,\xi} \quad (7)$$

Thus, as demonstrated by [1], this is an algorithm with a complexity of $O(n \log_2 n)$, representing an algorithm that grows almost linearly, as shown in Figure 1. This signifies a significant difference in computational time compared to an $O(n^2)$ algorithm, enabling efficient calculations of Fourier transforms for engineering and signal processing applications as known today.

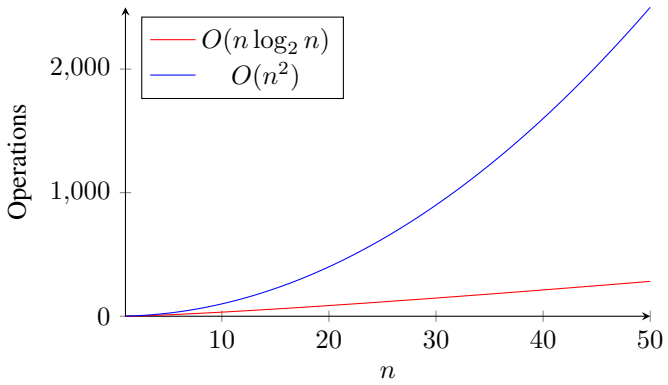


Figure 1: Complexity comparison between $O(n \log_2 n)$ and $O(n^2)$.

Note that this algorithm only works in the case where the vector size is a power of two. However, it is possible to apply this method to any vector by artificially adding a sequence of zeros to the end of the vector until its size reaches the nearest power of two. Even with this procedure of adding to the vector size, which is not necessary in the matrix calculation method of Equation 4, the FFT method is still considerably faster because adding these zeros at most increases the number of recursion steps by one.

B. Inplace Algorithm

Another implementation of the recursive algorithm is possible by performing the recursive operations directly on the same input vector, which saves both variables and memory dedicated to managing recursion, especially when dealing with vectors of considerable size. The inplace FFT algorithm requires the prior organization of elements in the vector strategically and subsequently performing the calculations presented in Equation 6.

To organize the vector, it is necessary to determine where each element of the sequence $[x_t]$ will be directed in the base recursion steps. To understand this organization, consider a vector of 8 elements. The element at index 6 is odd, so it is directed to $X_{even,\xi}$. In this new vector, it will be at index 3 ($[x_0, x_2, x_4, x_6]$), so it will be directed to $X_{odd,\xi}$. In this new vector, it will be at index 1 ($[x_2, x_6]$), so it will finally go to the $X_{odd,\xi}$ component. Note that the sequence of steps for index 6 is $(even, odd, odd)$ and that the binary representation of 6 is $(110)_2$. Considering $even = 0$ and $odd = 1$, the sequence becomes $(even, odd, odd) = (011)_2$, which is the binary representation of 6 in reverse. It can be proven that this fact holds for any other index [4]. The entire process can be seen in Equation 8, where it is clear that each index went to the position corresponding to its binary reversed representation.

$$\begin{aligned} &[x_{000}, x_{001}, x_{010}, x_{011}, x_{100}, x_{101}, x_{110}, x_{111}] \\ &X \rightarrow [X_{even}, X_{odd}] \\ &[x_{000}, x_{010}, x_{100}, x_{110}, x_{001}, x_{011}, x_{101}, x_{111}] \\ &X \rightarrow [X_{even}, X_{odd}] \\ &[x_{000}, x_{100}, x_{010}, x_{110}, x_{001}, x_{101}, x_{011}, x_{111}] \end{aligned} \quad (8)$$

With this organization, the calculation of the transform associated with each pair $(2i, 2i+1)$, where $i < N/2$, is performed, and the result is allocated in the same index interval $(2i, 2i+1)$. Subsequently, the calculation of the transform associated with positions $(4i, 4i+1, 4i+2, 4i+3)$, where $i < N/4$, is carried out, and the result is allocated in the same index interval, and so on until, in the last layer, the entire vector is transformed.

This method, like the recursive one shown earlier, also works only when the size of the vector is a power of two. However, just like the previous method, you can add zeros to the vector until its size reaches a power of two.

III. Methodology

A. Implementation

In this section, the main modules and functions implemented will be discussed and presented.

1) FFT Module: This module contains the basic implementations of the functions to be used, as well as auxiliary functions to facilitate implementation and code readability. Below is a description of the main functions implemented in the FFT module:

- **fft(x, inv=1):** Function that performs the fast Fourier transform of a one-dimensional vector with a size corresponding to a power of two using the recursive algorithm.
- **fft_inplace(x, inv=1):** Function that performs the fast Fourier transform of a one-dimensional vector with a size corresponding to a power of two using the algorithm that transforms the vector itself, without using recursion, optimizing memory allocation.
- **fft2(img, tech):** Function that performs the fast Fourier transform of a two-dimensional vector with a size corresponding to a power of two, using the algorithm chosen through its `tech` parameter. It first transforms the rows and then the columns of the vector.
- **ifft(X):** Function that performs the fast inverse Fourier transform of a one-dimensional vector with a size corresponding to a power of two using the recursive algorithm. It uses the implementation of **fft()** with the parameter $inv = -1$ and divides by the vector size at the end.
- **ifft_inplace(X):** Function that performs the fast inverse Fourier transform of a one-dimensional vector with a size corresponding to a power of two using the algorithm that transforms the vector itself, without using recursion, optimizing memory allocation. It uses the implementation of **fft_inplace()** with the parameter $inv = -1$ and divides by the vector size at the end.
- **ifft2(img, tech):** Function that performs the fast inverse Fourier transform of a two-dimensional vector with a size corresponding to a power of two, using the algorithm chosen through its `tech` parameter. To reverse the effect of **fft2()**, this function performs inverse transforms first in relation to the columns and then in relation to the rows, dividing by the vector size at the end.

2) Image Processing Module: This module is responsible for processing images to prepare them for input into compression techniques, adjusting the output state (removing complex numbers and adjusting the image type), as well as performing compression and color channel adjustment operations. The functions used are listed below:

- **type_adjustment(img):** Adjusts the output image to limit values to the supported pixel color range: $[0, 255]$, and adjusts the output type to $np.uint8$, which is a type accepted by the PIL image library.
- **expand(img):** Expands the dimensions of the image to the nearest power of two, adding zeros by default, to utilize the developed techniques.
- **contract(img, original_shape):** Contracts the image, undoing the operation of the previous function.
- **compress(img, compression_factor, tech):** Performs image compression with possibly more than one color channel, according to the technique specified in the Introduction of this work, using the algorithm specified by the $tech$ parameter, which can be $recursive$ or $inplace$.
- **compress_monotone(img, compression_factor, tech):** Performs compression of a single-channel image (the image must be one-dimensional), according to the technique specified in the Introduction of this work, using the algorithm specified by the $tech$ parameter, which can be $recursive$ or $inplace$.

3) Utilities Module: This additional module contains the implementation of some functions not necessarily related, used in other modules to facilitate code application and readability. The implemented functions are:

- **get_inv_binary_associate(i, size):** Computes the number whose binary representation reversed is equal to the binary representation of the parameter i. This function is used to perform the bit-wise inversion operation used in the in-place algorithm. The $size$ parameter is necessary because some inverse representations may differ depending on how many bits are available for the allocation of a number, for example: $(0001)_2^{-1} = (1000)_2 \neq (1)_2 = (1)_2^{-1}$ even though $(0001)_2 = (1)_2$.
- **swap(arr, index1, index2):** Swaps the contents of indices $index1$ and $index2$ in the array arr.
- **bitwise_invert(arr, n, log_n):** Performs the bit-wise inversion operation on the array arr. This occurs when the content of position i is moved to the position whose binary representation is the reverse of the binary representation of i.

IV. Performance Comparisons

The comparisons performed to analyze the operation mode of the implemented function were threefold:

1) Comparison of the execution time of the Fourier transform of vectors of varying sizes between the implemented functions and a baseline function from the *numpy* library.
2) Comparison of original images with their compressed versions for different types of images, especially to

analyze how the same compression factor can vary the final result for images of different sizes.

3) Comparison of different compression factors for the same image to determine how the visual result varies with the gradual increase in the compression factor.

V. Results

The execution time of each algorithm was compared by performing the Fourier transform of vectors of progressively larger sizes. The times were calculated from the average of 100 executions for each algorithm, run on an Acer Aspire 5 computer with 8GB of RAM. The comparison graph between the implemented functions and the *numpy* library function is shown in Figure 2.

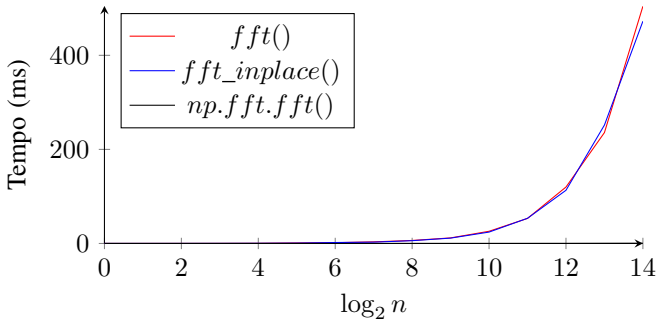


Figure 2: Execution time for each function based on a random array of size n, with the x axis being $\log_2 n$.

It is noticed that the function from the *numpy* library behaved considerably more efficient - about a thousand times - more efficient, even though it still has a complexity of $O(n \log_2 n)$. Thus, the presentation of the execution time of this function was shown separately in Figure 3.

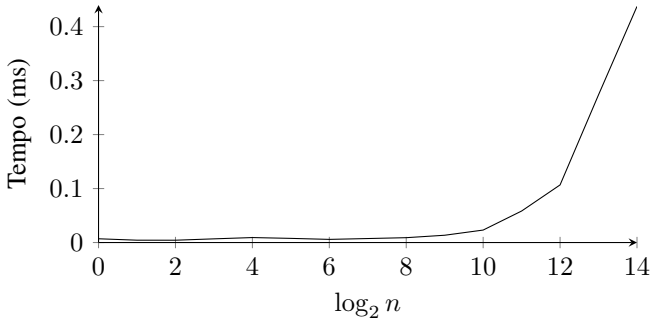


Figure 3: Execution time for the function $np.fft.fft()$ based on a random array of size n, with the x axis being $\log_2 n$.

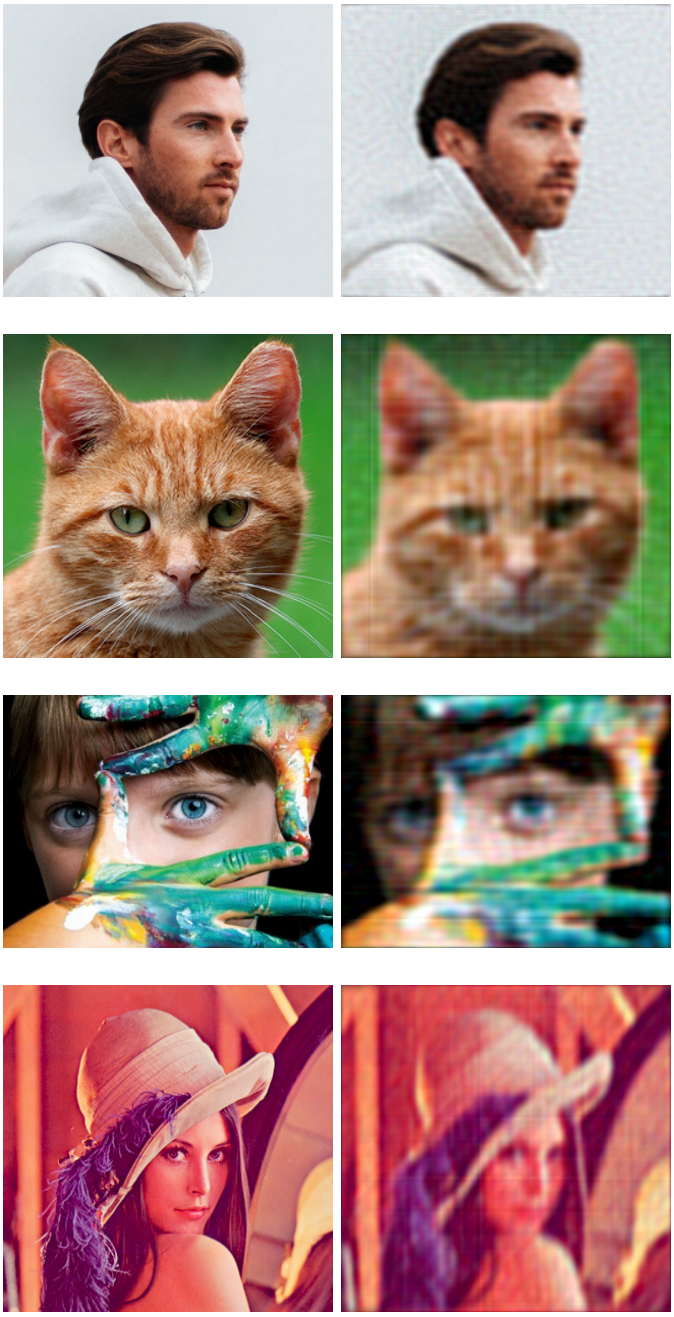
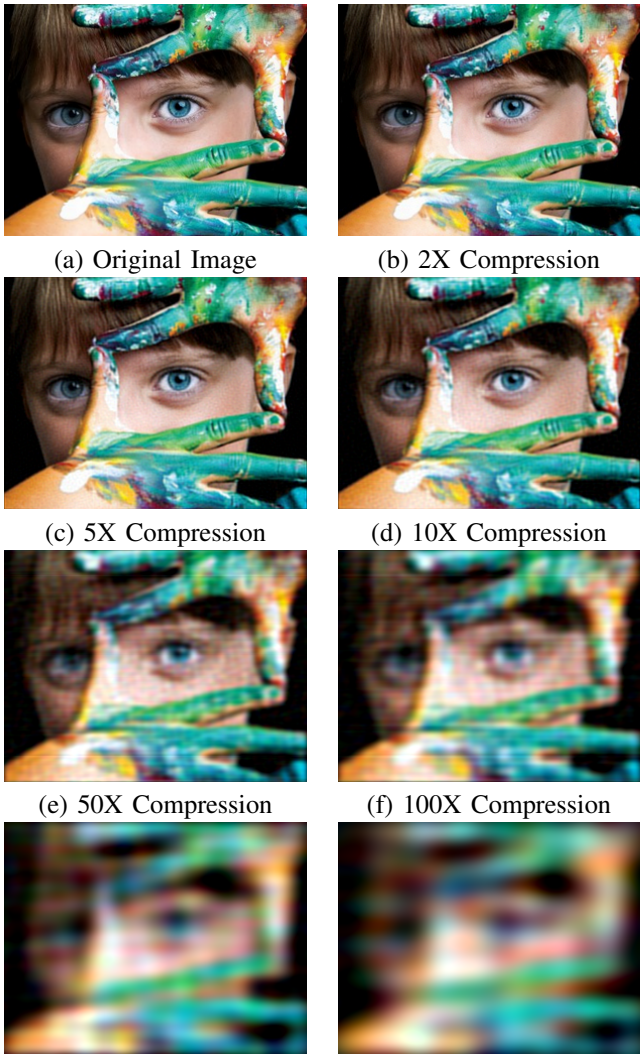


Figure 4: Example of 100 times compression applied to 4 test images. On the left is the original image, and on the right is the compressed image..

On the other hand, the effect of 100 times compression was compared for 4 different benchmark images, as shown in Figure 4. This comparison aims to illustrate how the compression factor may appear different for various image sizes. For instance, the second image seems much more compressed than the first image, primarily because the second image is originally smaller than the first, with a resolution of 230x300 compared to the first image's 498x562.

Finally, the results were compared for the same image but with an increasing degree of compression, as depicted in Figure 5.



(a) Original Image (b) 2X Compression

(c) 5X Compression (d) 10X Compression

(e) 50X Compression (f) 100X Compression

(g) 500X Compression (h) 1000X Compression

Figure 5: Example of increasing compression for an image.

VI. Conclusion

Regarding the efficiency of the implemented functions, it is evident that the *numpy* library's function achieved significantly higher operational efficiency, approximately 1000 times faster than the implemented functions. Concerning the execution time of the two implemented functions ($fft()$ and $fft\_inplace()$), there is no significant difference between these two functions. They practically exhibit the same efficiency. However, it is worth noting that the $inplace$ implementation requires less memory access than the recursive implementation, which can be considered an advantage of the latter over the former.

In terms of the relationship between visual compression and image size, it is observed that it is a proportional relationship. The larger the image, the greater the compression factor that can be applied before the visual change becomes significant. This is mainly because large images, when converted to the frequency domain, contain large amounts of memory dedicated to high frequencies (which can be discarded due to their reasonably small modulus), providing more room for compression compared to smaller images.

Finally, based on Figure 5, it is noticeable that as the compression factor increases, the image typically becomes progressively more homogeneous due to the elimination of high frequencies responsible for abrupt changes. On the other hand, imperfections in the image become more pronounced in the horizontal and vertical directions as the compression factor increases, such as disjointed lines. This phenomenon occurs due to the compression method used, which compresses in these directions separately.

References

[1] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19 (1965). URL: `http://cr.yp.to/bib/entries.html#1965/cooley`, pp. 297–301. ISSN: 0025–5718.

[2] David Kammler. *A First Course in Fourier Analysis*. 2nd ed. Southern Illinois University, 2008. ISBN: 9780511619700. DOI: https : / / doi . org / 10 . 1017 / CBO9780511619700.

[3] Henri J. Nussbaumer. "The Fast Fourier Transform". In: *Fast Fourier Transform and Convolution Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 80–111. ISBN: 978-3-642-81897-4. DOI: 10.1007/978-3-642-81897-4_4. URL: https://doi.org/10.1007/978-3-642-81897-4_4.

[4] JoséM. Pérez-Jordá. "In-place self-sorting fast Fourier transform algorithm with local memory references". In: *Computer Physics Communications* 108.1 (1998), pp. 1–8. ISSN: 0010-4655. DOI: https://doi.org/10.1016/S0010-4655(97)00116-1. URL: https://www.sciencedirect.com/science/article/pii/S0010465597001161.

[5] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.