Project 2 of "Metodi Del Calcolo Scientifico"

Academic Year 2019/2020

Simone Paolo Mottadelli, 820786

Contents

1 Introduction

The main purpose of this project was to use the Discrete Cosine Transform 2 (**DCT2**) implementation in an open source environment in order to assess the effects of a compression algorithm on gray scale images.
In particular, this report consists of three sections: Section 2 describes a possible implementation of the DCT2 as presented during the lectures and shows the results obtained from the comparison between such implementation and the one provided by a an open source library in terms of computational complexity; Section 3 focuses on the implementation of a compression algorithm for gray scale images, which does not use a quantization matrix but is very similar to the JPEG algorithm, and, finally, Section 4 concludes this report by briefly summarizing what has been presented in the previous sections.

2 First part

The first part of the project has been implemented using the **Python** [1] programming language because it is open source, very powerful and it provides all the functionalities and libraries needed to achieve the goals of the project. More in detail, three libraries have been used:

- **SciPy** [2], which is a Python-based ecosystem of open-source software for mathematics, science and engineering. It contains a fast implementation of the DCT2;
- **NumPy** [3], which is the core library for scientific computing in Python [1] and which mainly adds support for multidimensional arrays;
- **Matplotlib** [4], which is a comprehensive library for creating static, animated and interactive visualizations in Python [1].

The code has been organized into three modules:

- **my_dct2.py**, containing my implementation of the DCT2;
- **comparator.py**, containing the logic for comparing my DCT2 implementation with the one of the SciPy [2] library;
- **results.py**, containing the logic for saving the results of the comparison on a file and plot the results.

For further details on the implementation of this part of the project, the code is available at `https://bit.ly/2WiofxC`.

2.1 Implementation of the DCT2

The **my_dct2.py** module contains my implementation of the DCT2, which has been realized by applying the DCT1 first on the rows and then on the columns of the matrix in input. This design choice is related to the fact that the "brute-force" implementation of the DCT2 using the following formula has a time complexity equals to $O(n^4)$:

$$c_{kl} = \frac{1}{\sqrt{\alpha_k}\sqrt{\alpha_l}} \sum_{i=1}^{n} \sum_{j=1}^{n} cos\left(\pi k \frac{2i-1}{2n}\right) cos\left(\pi l \frac{2j-1}{2n}\right) m_{ij}$$

where

$$c, m \in \mathbb{R}^{n \times n} \quad \text{and} \quad \alpha_k = \begin{cases} n & k = 0 \\ \frac{n}{2} & k \neq 0 \end{cases}$$

Instead, applying the DCT1 with the following formulas on the matrix in input, first on its rows and then on its columns, has a time complexity equals to $O(n^3)$:

$$c_k = \frac{1}{\sqrt{\alpha_k}} \sum_{i=1}^{n} cos\left(\pi k \frac{2i-1}{2n}\right) m_i$$

where

$$c, m \in \mathbb{R}^{n} \quad \text{and} \quad \alpha_k = \begin{cases} n & k = 0 \\ \frac{n}{2} & k \neq 0 \end{cases}$$

Concerning the code, Figure 1 shows the content of the **my_dct2.py** module. In particular, it contains two functions:

- **my_dct(m)**, which receives an **m** matrix in input and simply computes the DCT1 for each row of **m**;
- **my_dct2(m)**, which receives an **m** matrix in input and calls **my_dct(m)** twice, first on **m** and then on the resulting transposed matrix.

```
# input: m (a numpy matrix)
def my_dct(m):
    nrows = m.shape[0] # get the number of columns of the m matrix
    ncols = m.shape[1] # get the number of rows of the m matrix
    c = zeros((nrows, ncols))
    for row in range(0, nrows):
        for k in range(0, ncols):
            total_sum = 0
            for i in range(1, ncols + 1):
                total_sum += cos(pi * k * (2 * i - 1) / (2 * ncols)) *
                                m[row][i - 1]
            alpha_k = ncols if k == 0 else ncols * 0.5

            c[row][k] = total_sum / sqrt(alpha_k)
    return c

# input: m (a numpy matrix)
def my_dct2(m):
    return my_dct(my_dct(m).transpose()).transpose()
```

Figure 1: My DCT2 implementation

2.2 Comparison with the SciPy library

After having implemented the DCT2 as described in Subsection 2.1, an empirical evaluation study has been conducted with the aim of comparing my DCT2 implementation with the one provided by the SciPi [2] library in terms of their computational complexity. All the experiments have been conducted with the same machine equipped with an Intel Xeon CPU ES-2667 v3 processor and 16 GB of RAM.
First of all, the two algorithms have been executed with square matrices of increasing dimension and their execution time has been registered. This has been achieved by implementing the code inside the *comparator.py* module, whose logic is very simple:

1. It generates a random matrix of dimension $N \times N$
2. It registers the execution time of my DCT2 algorithm
3. It registers the execution time of the DCT2 of the SciPy [2] library
4. it returns to step 1 until the maximum predetermined dimension of the matrices has been reached

To conduce the experiments, the maximum predetermined dimension of the matrices has been set to $N = 1000$ and the results have been collected using

the **results.py** module, containing the logic for saving the results of each experiment on a file and for plotting the results on a semi-logarithmic scale.
Note that the actual execution time of an algorithm is strongly influenced by the scheduler of the operating system, which decides how much CPU time to allocate to each process. To mitigate this problem, 4 runs of the experiments have been conducted and the merged results are shown in Figure 2. In particular, the plot shows that my implementation of the DCT2 has approximately a cubic time complexity, while the algorithm provided by the SciPy [2] library is much more efficient, since it provides a faster implementation of the DCT2 (**FFT**), which has a time complexity equals to $O(nlog(n))$, and it optimizes the computation with parallel computing. In addition, it is possible to see that the curve generated by the algorithm of the SciPy [2] library is much more irregular because it depends on its actual implementation.

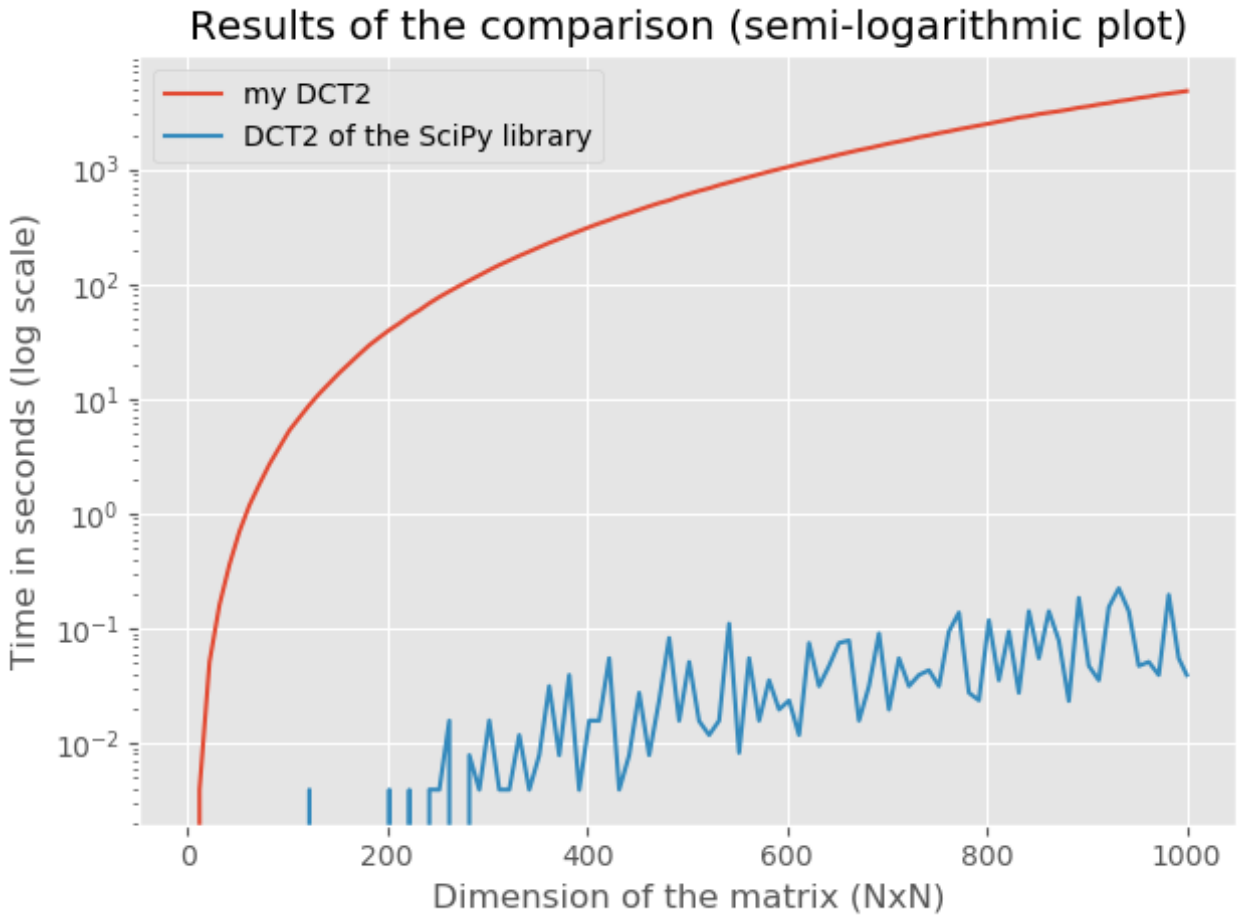


Figure 2: Results of the comparison between my DCT2 implementation and the one of the SciPy [2] library

3 Second part

This second part of the project has also been implemented using the Python [1] programming language for the same reasons described in Section 2.
Moreover, since this part of the project required to work on **.bmp** images, an additional library, **Pillow** [5], has been used for opening and manipulating the images in input.
From an architectural point of view, the software has been organized into the following modules:

- **main.py**, which is the entry point of the software and which controls the execution flow of the program;
- **input_parser.py**, which implements the logic for parsing the user input, that is, the **F** and **d** parameter as well as the **path** in the file system of where the image file is located;
- **image_processing.py**, which implements the compression algorithm for the gray scale images;
- **results.py**, which realizes the logic for showing the results of the compression algorithm to the user, i.e., it displays the image in input and the compressed image side by side at the end of the execution in an interactive window.

For further details on the implementation of this part of the project, the code is available at `https://bit.ly/2WiofxC`.

3.1 Software functioning

To execute the program, the user has to open a CMD window and enter the following command:

```
> python main.py -f <int value> -d <int value> -path <filepath>
```

where

- **-f** is the option to specify the value of the **F** parameter, that is, the dimension of the Minimum Coded Unit (MCU) on which the DCT2 is executed;
- **-d** is the option to specify the value of the **d** parameter, that is, the cutoff threshold of the frequencies;
- **-path** is the option to specify the location of the image in the file system.

For example, the following command executes the program on an image with $F = 8$ and $d = 6$:

```
> python main.py -f 8 -d 6 -path image.bmp
```

More in detail, when the program is executed, the user input is validated by the **input_parser.py** module, which first checks that all the required parameters have been entered and then checks the admissibility of the values of the parameters. For instance, it checks whether the value of the d parameter falls in the interval [0, 2F - 2] or whether the image exists in the file system.
After the input is validated, the image is loaded from the file system, stored in a matrix and processed by the **image_processing.py** module. The latter encapsulates the heart of the compression algorithm and is shown in Figure 3 and in Figure 4.
In particular, the **process_image(img_to_process , f, d)** function, shown in Figure 3, receives in input the image to process and the values of the F and d parameters and it applies the following steps on each pixel square of dimension $F \times F$, aka Minimum Coded Unit or MCU, starting from the upper-left corner of the image:

1. It applies the DCT2 of the SciPy [2] library on the MCU;
2. For each coefficient (pixel) of the MCU, it sets it to zero if its position, which is identified by the pair of indices (k, l), is such that $k + l \geq d$;
3. It applies the IDCT2 of the SciPy [2] library on the MCU;
4. it rounds the coefficients of the MCU to the nearest integer value and it adjusts their values to make them fall in the [0, 255] interval.

This function makes use of the two helper functions shown in Figure 4, which implement the aforementioned steps 2 and 4, respectively.
Note that when the division of the image into square blocks of pixels of size $F \times F$ occurs, i.e., when the MCUs are generated, there might be some "leftovers" if the size of the image is not divisible by F. Therefore, the algorithm simply ignores those "leftovers", but does not delete them from the image. However, this should not be noticed by the human eye if the value for F is small enough compared to the image size. It's also worthwhile to notice that if F is larger than the image size, the image is not compressed, because the whole image is considered as a "leftover". This implementation solution has been chosen because there are no explicit constraints on the F parameter.

```
# input: the image to process
# input: the value of the F parameter
# input: the value of the d parameter
def process_image(img_to_process, f, d):

    # work on a copy of the image in input
    img = img_to_process.copy()

    # get the dimensions of the image
    nrows = img.shape[0]
    ncols = img.shape[1]

    # compute the DCT2 on each MCU starting from the upper-left corner
    # of the image
    for start_mcu_row in range(0, nrows, f):
        for start_mcu_col in range(0, ncols, f):

            # find the pixel positions to identify a MCU
            end_mcu_row = start_mcu_row + f
            end_mcu_col = start_mcu_col + f

            # extract the mcu from the image
            mcu = img[start_mcu_row:end_mcu_row, start_mcu_col:end_mcu_col]

            # if the mcu is a FxF square matrix, then compress it
            if mcu.shape[0] == f and mcu.shape[1] == f:
                mcu = dctn(mcu, type=2, norm="ortho") # apply dct2
                mcu = cutoff_frequencies(mcu, d)
                mcu = idctn(mcu, type=2, norm="ortho") # apply idct2
                mcu = adjust_coefficients(mcu)

            # save the processed mcu in its original position in the image
            img[start_mcu_row:end_mcu_row, start_mcu_col:end_mcu_col] = mcu

    return img
```

Figure 3: Implementation of the compression algorithm

```python
# This function is used to cutoff the frequencies of the MCU in input
def cutoff_frequencies(mcu, d):
    for k in range(0, mcu.shape[0]):
        for l in range(0, mcu.shape[1]):
            if k + l >= d:
                mcu[k, l] = 0
    return mcu

# This function is used to round the coefficients of the MCU and
#adjusts them in such a way that their values fall in the [0, 255] interval
def adjust_coefficients(mcu):
    for i in range(0, mcu.shape[0]):
        for j in range(0, mcu.shape[1]):
            mcu[i, j] = max(0, min(round(mcu[i, j]), 255))
    return mcu
```

Figure 4: Helper functions used by the compression algorithm

After having processed the image, the **results.py** module is responsible for showing the results of the computation to the user, that is, it displays the image before and after the compression in an interactive screen, where the user can zoom in or zoom out the images to validate the quality of the compression, as also shown in Figure 5.

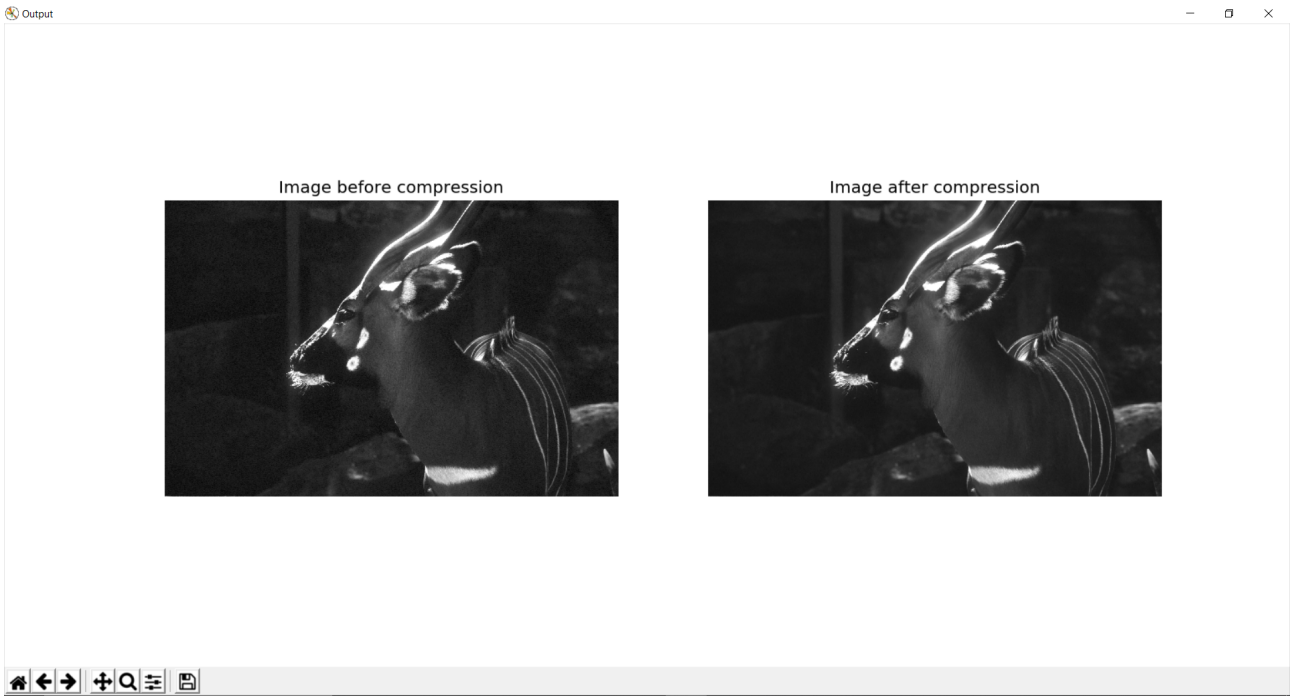


Figure 5: Example of output

3.2 Experimentation

This subsection presents some experiments that have been done to evaluate the effectiveness of the compression algorithm presented in Subsection 3.1.
In particular, the compression algorithm seems to work quite well on images that are well nuanced, even if a low value of the d parameter is chosen, as shown in Figure 6, where the values chosen for the parameters are $F = 10$ and $d = 3$. The two images are almost identical to the human eye at this zoom level.
Instead, the same parameter setting on high contrast images leads to a greater

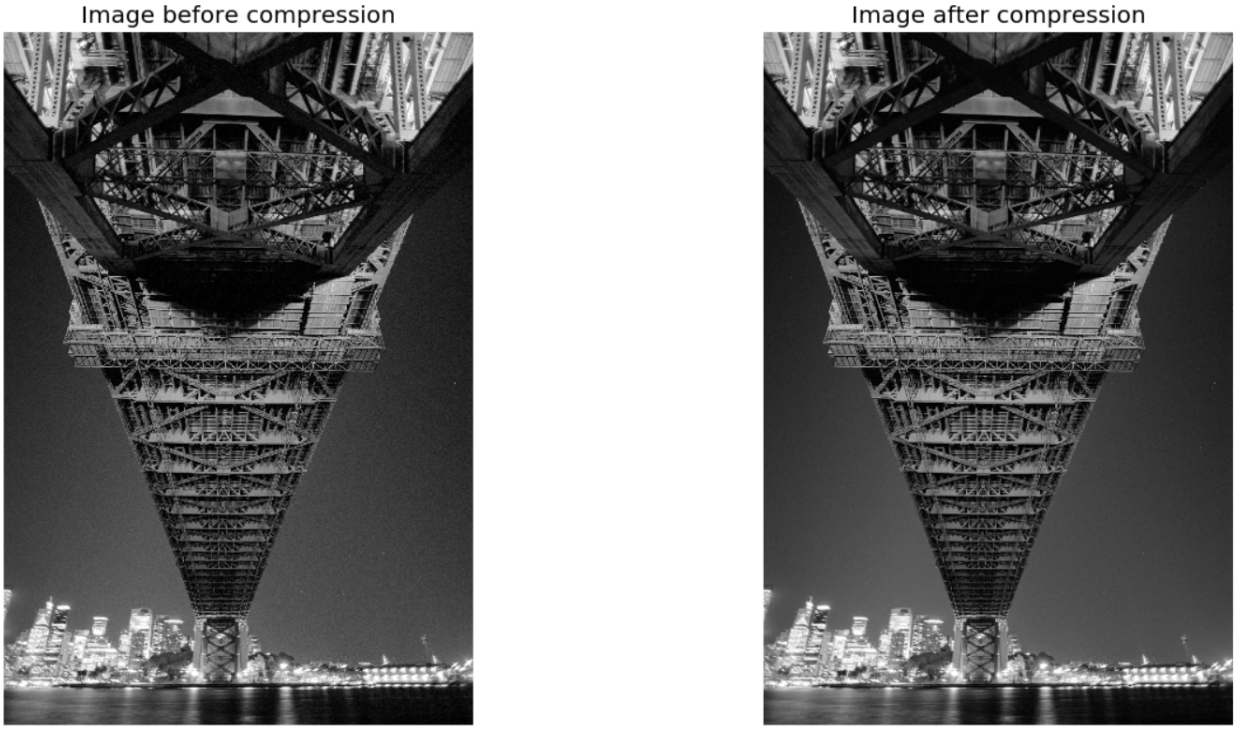
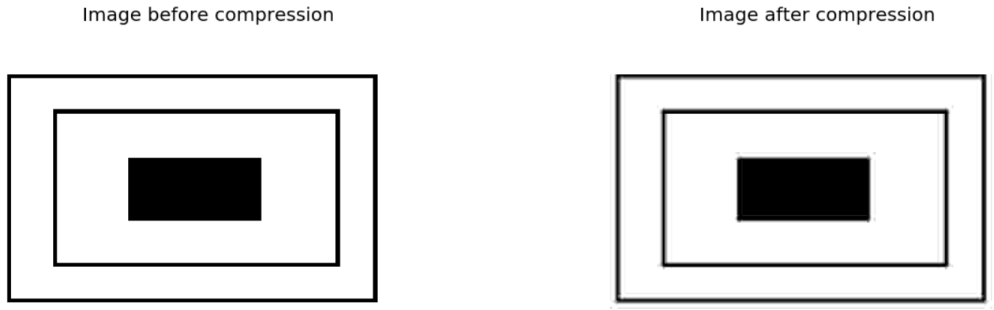


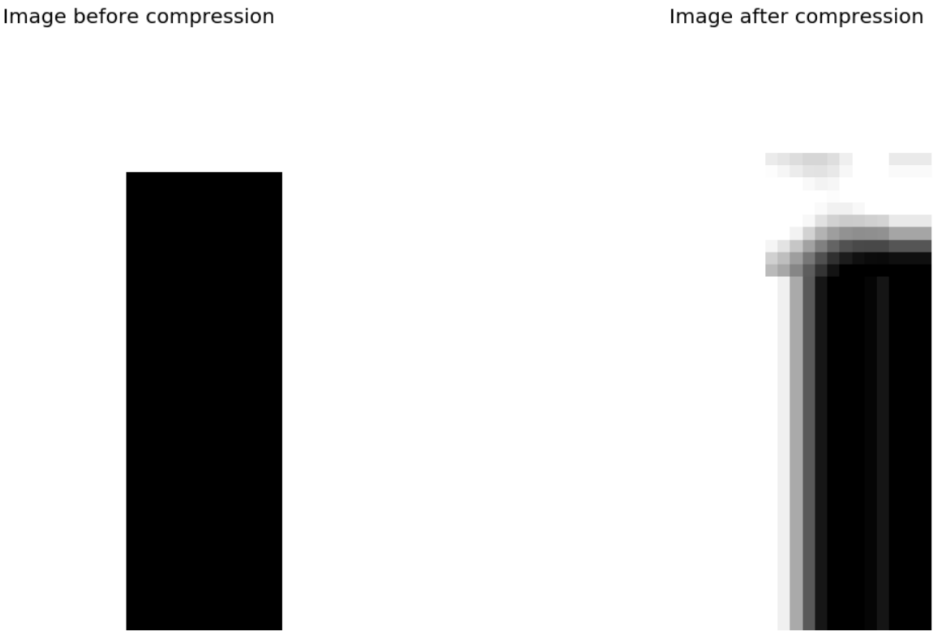
Figure 6: Example with a nuanced image ($F = 10$, $d = 3$)

loss of information; in fact, some artifacts and noise are generated in proximity of contrast jumps, as shown in Figure 7. This is related to the Gibbs phenomenon in signal processing, which states that near a jump discontinuity in a signal there will always be an error in the form of an overshoot, no matter how many terms are included in the Fourier series [6].
The Gibbs phenomenon becomes more pronounced if the value of the F parameter is big, while it reduces if its value is small. This idea is best clarified by looking at Figures 8 and 9.
It's worthwhile to mention also the boundary cases, such as using a value equals to zero for the d parameter, which leads to a complete loss of information, thus resulting in a black image, as shown in Figure 10, and the case of using the

maximum value of d, which leads to obtain an image that is not compressed at all, as shown Figure 11.
Finally, as a last extreme case, if a huge value for F is used, bigger than the size of the image, the whole image is a "leftover" and no compression is applied since no MCU blocks exist, as already anticipated in Subsection 3.1. The resulting image is unmodified, as shown by an example in Figure 12.



(a)



(b)

Figure 7: (a) Example with a high contrast image ($F = 10$, $d = 3$). (b) The same as (a) but zoomed in to show the artifacts introduced by the compression algorithm

Figure 8: Gibbs phenomenon when ($F = 30$, $d = 3$)

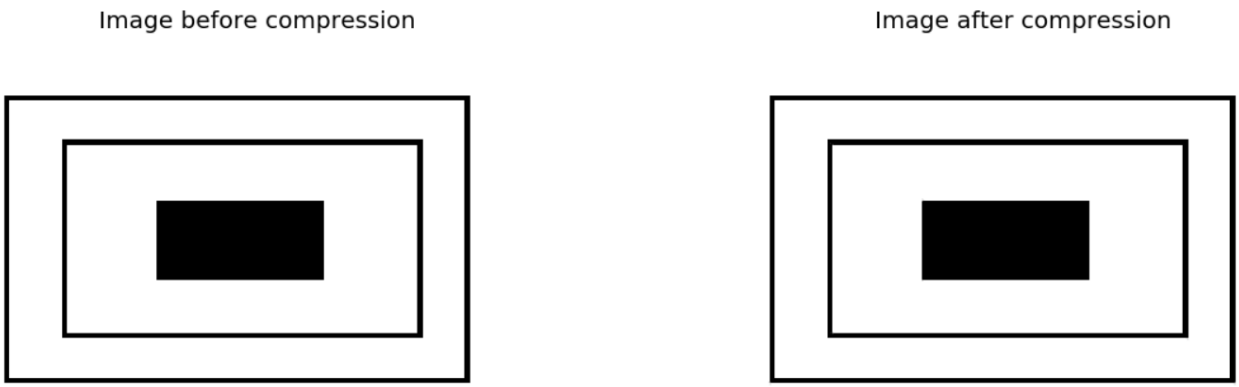


Figure 9: Gibbs phenomenon when ($F = 3$, $d = 3$)

Figure 10: Boundary case: minimum value of d ($F = 10$, $d = 0$)

Figure 11: Boundary case: maximum value of d ($F = 10$, $d = 18$)

Figure 12: Extreme case: huge value of $F = 30000$

4 Conclusion

This report has first presented in Section 2 a possible implementation of the DCT2 in an open source environment and a comparative study between such implementation and the one provided by the SciPy [2] library. The study has shown that the implementation of the SciPy [2] library is much more efficient in terms of computational complexity, because it exploits both the Fast Fourier Transform algorithm and parallel computing to optimize the computation.
In conclusion, Section 3 has presented an implementation of a compression algorithm for gray scale images, very similar to JPEG, and an experimentation study to assess its effectiveness. In particular, the study has shown that the compression algorithm is effective with nuanced images, but it is not optimal with high contrast ones, because artifacts and noise can be generated especially in correspondence of their contrast jumps, as explained by the Gibbs phenomenon.

References

[1] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[2] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[3] Travis Oliphant. *Guide to NumPy*. 01 2006.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[5] Pillow. `https://pypi.org/project/Pillow/7.1.2/`. Accessed: 2020-04-26.

[6] Gibbs' phenomenon. In *Operations on Fourier series — MIT Course in mathematics*. Cambridge MA, 2011. MIT OpenCourseWare.