DISCRETE COSINE TRANSFORM FOR IMAGE PROCESSING: JPEG COMPRESSION

Discrete Dynamical Models

Alberto ROGANO

2023-2024

Professor: Franco Tomarelli

Contents

1 Introduction

Compression is a widely used technique in image processing to reduce the storage space of an image. In particular, the JPEG compression algorithm is based on the Discrete Cosine Transform (DCT), especially for its strong energy compaction property in lossy compression: a signal's DCT representation tends to concentrate its energy in a small number of coefficients with respect to the whole image. Starting from the Discrete Fourier Transform (DFT) representation, it is possible to introduce the concept of DCT: a finite sequence of data points expressed by a sum of cosine functions oscillating at different frequencies.
In the following we will introduce this mathematical tool and its application in the image processing field. First of all, the DFT is presented as a basic representation for vectors to map them from "time domain" to the "frequency domain". Then, the DCT is introduced as a real transform that involves cosines in all the possible combinations with respect to the boundary conditions at meshpoints and midpoints. Since the main field of application of DCT is image processing, a global overview on the color models is presented and the compression technique is presented. The Joint Photographic Experts Group (JPEG) pointed out the namesake method for lossy compression for digital images, in particular for the digital photography ones. Its algorithm is presented in Section 5 and a graphic user interface has been coded on Matlab to present its application both in RGB and GrayScale cases.

2 Discrete Fourier Transform

The Discrete Fourier Tranform (DFT) is a tool widely used to treat periodic sequence.

Definition 1 *A sequence $\boldsymbol{x}$ is said to be periodic with period N, where $N \in \mathcal{N}$, if $x_k = x_{k+N}$ holds for every k in $\mathcal{N}$.*

Consider a vector $\mathbf{x}$ with N components that represents a periodic sequence, its DFT is the combination of N special basis vectors

$$\mathbf{v}_k = (1, w^k, w^{2k}, \cdots, w^{(N-1)k}),\ k = 0, 1, \cdots, N-1,$$

where w is a complex number defined as $w = e^{\frac{2\pi i}{N}}$. The vectors v_k are the orthogonal columns of the Fourier matrix F. The discrete Fourier series can be written as

$$\mathbf{x} = \sum c_k \mathbf{v}_k \tag{1}$$

or in matricial form as

$$\mathbf{x} = \mathcal{F}\mathbf{c},$$

where c_k are the complex Fourier coefficients that can be obtained for each vector by

$$c_k = \frac{\mathbf{x} \cdot \mathbf{v}_k}{N}.$$

Thanks to the N-periodicity of the sequence w^k, we have $w^{hk} = w^{(h-N)k}$, and the system (9) presented in Appendix A is equivalent to (1).
The matrix $\mathcal{F}$ is a symmetric Vandermonde matrix: its rows are the powers of the distinct numbers $1, w, w^2, \cdots, w^{N-1}$ with exponents $0, 1, \cdots, N-1$:

$$\mathcal{F} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & \omega^{N-1} \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \cdots & \omega^{(N-1)} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

$\mathcal{F}$ is invertible since its determinant is not null and the inverse is

$$\mathcal{F}^{-1} = \frac{1}{N}\bar{\mathcal{F}}.$$

In this way, also an Inverse Discrete Fourier Transform can be defined:

$$\mathbf{Y} = \mathrm{DFT}\mathbf{y},$$
$$\mathbf{y} = \mathrm{DFT}^{-1}\mathbf{Y}.$$

Two key-points of the Fourier analysis are the orthogonality of the vectors $\mathbf{v}_k$ and the speed of calculation [5]. From the first property we can deduce that the complex exponential vectors $\mathbf{v}_k$ are eigenvectors and so they cover an important role in applied mathematics. Since the Fourier matrix is full, the computational cost for transform and inverse transform is $O(N^2)$. The special form $F_{jk} = w^{jk}$ allows us to factorize the computations into very sparse and simple matrices dropping the cost to $\frac{1}{2}NL$, where L is a value such that we assume N a power 2^L. This technique is called Fast Fourier Transform (FFT) and it is the best way to optimize the DFT computation.
The Fourier transform works perfectly with periodic boundary conditions. For example, if we have a second difference matrix we can produce the matrix A_0 with the scheme

$$A_0 = \begin{bmatrix} 2 & -1 & & & -1 \\ -1 & 2 & -1 & & \\ & -1 & \ddots & \ddots & \\ & & \ddots & 2 & -1 \\ -1 & & & -1 & 2 \end{bmatrix},$$

where the diagonals with -1 loop around the upper right and lower left corners, by periodicity $u_N = u_0$ and $u_{N-1} = u_{-1}$, to produce a circulant matrix.
We can verify that $\mathbf{v}_k = (1, w^k, w^{2k}, \cdots, w^{N-1}k)$ is an eigenvector of A_0. It is periodic since $w^N = 1$. The j-th component of $A_0\mathbf{v}_k = \lambda_k\mathbf{v}_k$ is the second difference:

$$\begin{aligned} -w^{(j-1)k} + 2w^{jk} - w^{(j+1)k} &= \left(-w^{-k} + 2 - w^k\right) w^{jk} \\ &= \left(-e^{-2\pi ik/N} + 2 - e^{2\pi ik/N}\right) w^{jk} \\ &= \left(2 - 2\cos\frac{2k\pi}{N}\right) w^{jk}. \end{aligned}$$

The eigenvalues λ_k are real and A_0 is symmetric. Real and imaginary parts of the $\mathbf{v}_k$ must be eigenvectors:

$$c_k = \Re(\mathbf{v}_k) = (1, \cos\frac{2k\pi}{N}, \cdots, \cos\frac{2(N-1)k\pi}{N}),$$
$$s_k = \Im(\mathbf{v}_k) = (0, \sin\frac{2k\pi}{N}, \cdots, \sin\frac{2(N-1)k\pi}{N}).$$

The equal pair of eigenvalues $\lambda_k = \lambda_{N-k}$ gives the two eigenvectors c_k and s_k, except for $\lambda_0 = 0$ that has one eigenvector $c_0 = (1, 1, \cdots, 1)$ and for even N also $\lambda_{N/2} = 4$ with $c_{N/2} = (1, -1, \cdots, 1, -1)$. Those two eigenvectors have length $\sqrt{N}$ while the others $\sqrt{N/2}$ and this is the reason why the real DFT is less attractive than the complex form.

3 Discrete Cosine Transform

In the DFT presentation we always considered complex matrix entries as powers of w.
In the following we will consider just real transforms that involve cosines, introducing the Discrete Cosine Transform (DCT). The DCT has been introduced by N. Ahmed, T. Natarajan, and K. R. Rao in 70's [1]. THey defined a new level of variety and complexity in discrete approximations due to the boundary conditions choice.
Starting from the periodicity of DFT, if we consider the cosines alone, we expect the series to be complete over a half-period $[0, \pi]$. In this way periodicity is no more valid since $\cos 0 \neq \cos \pi$. To produce just cosines we should insert the boundary condition $u^{'}(0) = 0$. On the other boundary we can have both Neumann and Dirichlet boundary conditions:

$$\begin{aligned} &\text{Zero slope:} \quad u^{'}(\pi) = 0 \quad \rightarrow \quad u_k(x) = \cos kx; \\ &\text{Zero value:} \quad u(\pi) = 0 \quad \rightarrow \quad u_k(x) = \cos\left(k + \frac{1}{2}\right) x. \end{aligned} \tag{2}$$

The eigenvalues from $-u_k^{''} = \lambda u_k$ are $\lambda = k^2$ and $\lambda = (k + \frac{1}{2})^2$. When we go to the discrete case for this second order differential equation, we can apply different boundary conditions as shown in (2). The application point can be a meshpoint or a midpoint and this leads to four different approximations. The complete set is extended up to eight for further refinements in this choice leading to eight cosine transforms. Each of the eight matrices has such tridiagonal form:

$$A = \begin{bmatrix} \otimes & \otimes & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & \boxtimes & \boxtimes \end{bmatrix}.$$

For symmetry we should need off-diagonal -1's in the first and last rows so we normalize the value -2 produced by Neumann condition at meshpoint by $\sqrt{2}$ to make it orthogonal. The four stardar types of DCT are described by the basis vectors:

$$\begin{aligned} &\text{DCT-1:} \quad \cos\left(jk\frac{\pi}{N-1}\right) \quad \text{(divide by } \sqrt{2} \text{ when } j \text{ or } k \text{ is 0 or } N-1), \\ &\text{DCT-2:} \quad \cos\left(\left(j + \frac{1}{2}\right) k\frac{\pi}{N}\right) \quad \text{(divide by } \sqrt{2} \text{ when } k = 0), \\ &\text{DCT-3:} \quad \cos\left(j\left(k + \frac{1}{2}\right)\frac{\pi}{N}\right) \quad \text{(divide by } \sqrt{2} \text{ when } j = 0), \\ &\text{DCT-4:} \quad \cos\left(\left(j + \frac{1}{2}\right)\left(k + \frac{1}{2}\right)\frac{\pi}{N}\right). \end{aligned}$$

Each of the vectors described above is a column of the corresponding DCT matrix C_1,C_2,C_3,C_4. Each basis is orthogonal since they are eigenvectors of symmetric second difference matrices.

3.1 Boundary Conditions at Meshpoints and Midpoints

In the problem described in Section 3 we introduced the boundary condition $u^{'} = 0$. This can be discretized in two ways:

- **Whole-sample:** symmetry around the meshpoint $j = 0$, $u_{-1} = u_1$. It extends $(u_0, u_1, \cdots)$ evenly across the left boundary to $(\cdots, u_1, u_0, u_1, \cdots)$. The second order difference operator is discretized as $2u_0 - 2u_1$. The top row of the matrix A can be filled as $\otimes \otimes = 2 \ -2$.

- **Half-sample:** symmetry around the midpoint $j = -\frac{1}{2}$, $u_{-1} = u_0$. It extends the signal to $(\cdots, u_1, u_0, u_0, u_1, \cdots)$ repeating the initial value. The second order difference operator is discretized as $u_0 - u_1$. The top row of the matrix A can be filled as $\otimes \otimes = 1 \ -1$.

At the other boundary we can do the same for $u' = 0$. Substituting $u_N = u_{N-2}$ or $u_N = u_{N-1}$ in the second difference $-u_{N-2} + 2u_{N-1} - u_N$ we can obtain the two forms for Neumann:

- Meshpoint: $\boxtimes \boxtimes = -2 \ 2$
- Midpoint: $\boxtimes \boxtimes = -1 \ 1$

Another alternative can be the Dirichlet one $u(\pi) = 0$. If it is applied to the meshpoint $u_N = 0$ we need to remove the last term of the discretized operator. Instead, the midpoint condition is $u_N + u_{N-1} = 0$.

- Meshpoint: $\boxtimes \boxtimes = -1 \ 2$
- Midpoint: $\boxtimes \boxtimes = -1 \ 3$

The boundary conditions above give us eight combinations and all those give second-order accuracy around their center points.

3.2 The Standard Cosine Transforms

The standard cosine transforms are those where the centering is the same at the two boundaries: both meshpoint centered or both midpoint centered. In the following bulleted list we are gonna explore all of them, expliciting also the diagonal matrix D such that $D^{-1}AD$ is symmetric and orthogonal.

- **DCT-1**
  - Centers: $j = 0$, $j = N - 1$
  - Components: $\cos(jk\frac{\pi}{N-1}$
  - $D_1 = diag(\sqrt{2}, 1, \cdots, 1, \sqrt{2})$
  - Structure:

$$A_1 = \begin{bmatrix} 2 & -2 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -2 & 2 \end{bmatrix}.$$

    We can notice from the structure of A_1 that the boundary conditions applied are whole-sample on the top row and Neumann for the bottom one for $u'(\pi) = 0$. For the symmetry of the matrix we need the diagonalization via D_1. For orthogonality we need the first and last components to be divided by $\sqrt{2}$ when multiplied the eigenvectors by D_1^{-1}. The first and last ones have length $\sqrt{N-1}$, the others $\sqrt{(N-1)/2}$.

- **DCT-2**
  - Centers: $j = -\frac{1}{2}$, $j = N - \frac{1}{2}$
  - Components: $\cos((j + \frac{1}{2})k\frac{\pi}{N})$

- $D_2 = I$
- Structure:

$$A_2 = \begin{bmatrix} 1 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 1 \end{bmatrix}.$$

This is the most used DCT basis vectors, expecially in image processing as we will see in Section 5. The reason of its popularity is related to the case $k = 0$ that gives the flat vector $(1, 1, \cdots, 1)$. The boundary conditions are set at a midpoint $u_{-1} = u_0$ and $u_N = u_{N-1}$.

- **DCT-3**
  - Centers: $j = 0$, $j = N$
  - Components: $\cos\left(j(k + \frac{1}{2})\frac{\pi}{N}\right)$
  - $D_3 = diag(\sqrt{2}, 1, \cdots, 1)$
  - Structure:

$$A_3 = \begin{bmatrix} 2 & -2 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}.$$

From the structure of A_3 we can notice that on the left it is applied a Neumann condition while on the right it is Dirichlet, both at meshpoints. For orthogonality reasons it is important to apply the factor D_3^{-1}.

- **DCT-4**
  - Centers: $j = -\frac{1}{2}$, $j = N - \frac{1}{2}$
  - Components: $\cos\left((j + \frac{1}{2})(k + \frac{1}{2})\frac{\pi}{N}\right)$
  - $D_4 = I$
  - Structure:

$$A_4 = \begin{bmatrix} 1 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 3 \end{bmatrix}.$$

An half-sample boundary condition is applied and so at the midpoint we have Dirichlet for the bottom row. The eigenvectors are even at the left end and odd at the right end and this makes C_4 a symmetric eigenvectors matrix.

3.3 Odd Discrete Cosine Transforms

If we consider that the distance between the centers is no longer an integer as $N+\frac{1}{2}$ and $N-\frac{1}{2}$, we can define other four DCTs. In this case one center is a midpoint and the other one is a meshpoint. Usually the DCTs from five to eight are not used in signal processing since the half-integer periods are a disadvantage, but reflection offers a possible way out.

- **DCT-5**
  - Centers: $j=0$, $j=N-\frac{1}{2}$
  - Components: $\cos\left(jk\frac{\pi}{N-\frac{1}{2}}\right.$
  - $D_5 = diag(\sqrt{2},1,\cdots,1,\sqrt{2})$
  - Structure:

$$A_5 = \begin{bmatrix} 2 & -2 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 1 \end{bmatrix}.$$

  The boundary conditions applied are meshpoint for the top row and midpoint for the Neumann bottom condition. The flat vector $(1,1,\cdots,1)$ is one of the eigenvectors of A_5, corresponding to $k=0$ and $\lambda=0$, that can describe a solid color or a fixed intensity in image processing. Its coding gain is completely comparable to the DCT-2.

- **DCT-6**
  - Centers: $j=-\frac{1}{2}$, $j=N-1$
  - Components: $\cos\left((j+\frac{1}{2})k\frac{\pi}{N-\frac{1}{2}}\right)$
  - $D_6 = diag(1,\cdots,1,\sqrt{2})$
  - Structure:

$$A_6 = \begin{bmatrix} 1 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -2 & 2 \end{bmatrix}.$$

  For the left bound we take in account a midpoint boundary condition while on the meshpoint there is a Neumann condition. We can notice that A_6 has boundary conditions and eigenvector components in reverse order from A_6. As A_5, it contains the flat vector.

- **DCT-7**
  - Centers: $j=0$, $j=N-\frac{1}{2}$
  - Components: $\cos\left(j(k+\frac{1}{2})\frac{\pi}{N-\frac{1}{2}}\right)$
  - $D_7 = diag(\sqrt{2},1,\cdots,1)$

– Structure:

$$A_7 = \begin{bmatrix} 2 & -2 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 3 \end{bmatrix}.$$

The boundary condition applied for the right bound are Dirichlet at a midpoint.

- **DCT-8**
  - Centers: $j = -\frac{1}{2}$, $j = N$
  - Components: $\cos\left((j+\frac{1}{2})(k+\frac{1}{2})\frac{\pi}{N+\frac{1}{2}}\right)$
  - $D_8 = I$
  - Structure:

$$A_8 = \begin{bmatrix} 1 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & -1 & 2 & -1 \\ & & & & -1 & 2 \end{bmatrix}.$$

The boundary conditions applied for the right bound are Dirichlet at a meshpoint.

4 Image Processing

In Image Processing we usually split images into two main categories: bitmaps and vector. Vector images are expressed by mathematical formulas to draw lines and curves. Instead, bitmaps of dimension $M \times N$ are based on pixel patterns expressed via element with a specific amplitude and position. Depending on the the information contained in each pixel, we can divide the image in three different numerical color models:

- **Binary Image:** each pixel is stored as a single bit of value 0 or 1, attributed to black and withe respectively.
- **Grayscale Image:** each pixel has intensity information that add a color depth between black and white of dimension L. The scale goes from 0 as the darkest black to $L-1$ as the brightest white. The gray scales pixels are described by a rational number that represent the luminance using 8-bit values that range in $[0, 255]$. The Grayscale representation is usually used for simplicity, artistic effect or just to reduce the file size.

Figure 1: Grayscale image of a dog. *Passetto, Ancona.*

- **RGB Image:** each pixel needs a tuple of numbers to represent the information. This leads to represent the image as a three-dimensional matrix. Since each color can be generated from the combination of the primary three as red (R), green (G) and blue (B), each value in the tuple associates the intensity for each of them. We usually refer to 24-bit images that use 8 bits for each channel. The number of colors that can be supported by the RGB combinations is $16 \cdot 777 \cdot 216$.

Figure 2: RGB image of a dog. *Passetto, Ancona.*

It could be useful to convert RGB images to grayscale and vice-versa. One of the most used methods is the luminosity one that combines the three channels according to their wavelengths to obtain the gray level. It is described by Formula 3 [2].

$$\text{Grayscale} = 0.299\text{R} + 0.587\text{G} + 0.114\text{B} \tag{3}$$

4.1 Lossy and Lossless Compression

Image compression is a process that reduces image files size in order to take up less storage space. Its role is crucial in web and applications development since compressed images load faster and this improves the overall performances. The compressions can be of two types:

- **Lossy:** it allows to reduce the image size consistently deleting a part of information contained in it. The drawback of this method is a grainy resolution that has qualitative bad results and does not allow to reproduce correctly the original image. After compression, the information are deleted and so it is not possible to recover them and restore the starting point. An example of Lossy compression is the JPEG algorithm that we will develop in Section 5.

- **Lossless:** it allows to reduce the image size deleting useless bit, duplicates or those that do not influence the final result. Even if the compressed image could have higher quality, this procedure is far more expensive and the final result could have less impact in terms of storage space saved.

There is no better compression method than the other but it depends on the context where we are applying it. If we need to save storage space keeping a good quality, Lossy compression is the best option but, for example, if we are working on a photography website, it could be necessary to keep each detail and perform a Lossless one.

5 JPEG Compression Algorithm

The Joint Photographic Experts Group developed a compression algorithm that aims to minimize the file size of the photographic image files. The DCT is the crucial tool for this algorithm and, in particular, the most used ones are DCT-2 and DCT-4 since they have a FFT implementation that makes them really useful in the computations.
Images are structured as in Section 4 and so we can deduce that they are not infinite and not periodic. Since the boundaries are not coherents, it is not possible to make them periodic just extending the series since we would create a discontinuity. This could lead to a slow decay of Fourie coefficients and a "ringing" effect. To overcome this problem we reflect the image across the boundary making the double-length periodic extension continous with cosine transforms.
We can present the JPEG compression algorithm starting from a two-dimensional GrayScale image where we apply a DCT-2 transform. It is applied before to rows and then to columns since 2D is seen as a tensor product of 1D with 1D. The first step is to divide the image into 8×8 blocks of pixels in order to produce 64 coefficients. The following step is a quantization one that puts the coefficients into a discrete set of bins. In this way it is easier to erase a set of them keeping just those that represents the image clearly for a good compression level. Taking a smaller number of bits for pixel, we obtain an image really compressed with a severe blocking.

6 Graphic User Interface for JPEG

In this section it will be shown how a graphic (G) user (U) interface (I) has been implemented on Matlab and how to use it to apply the JPEG compression to an RGB image. The underlying idea is the one proposed by the Matlab documentation [6] that has been developed for more complex cases.
The code is contained in Appendix B.
The interface structure is shown in Figure 3 and it is composed by:

- **Load Button:** this button needs to load the selected image
- **Compression Slider:** the slider allows us to select manually how many non-zero coefficients of the mask we want to keep
- **Compression Button:** this button needs to apply the compression function
- **Original Image window:** once the image is loaded it is displayed in this area
- **Compressed Image window:** once the image is compressed it is displayed in this area
- **Errors panel:** once the compression is applied, three metrics are computed and displayed

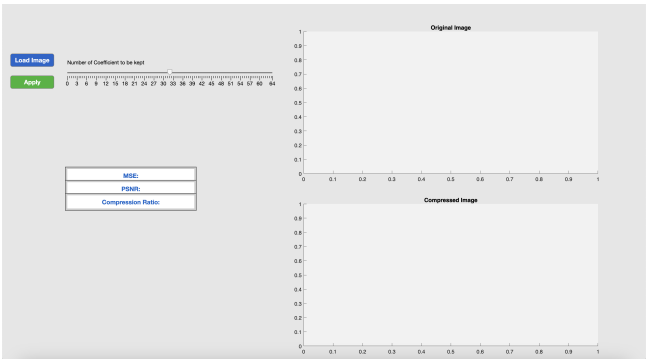

Figure 3: Graphic User Interface for JPEG compression

First of all, the image is loaded from the local directory clicking on the button 'Load Image' and then is shown in the 'Original Image' window. Then, we want to apply the compression function thanks to the 'Apply' button and its structure is based on the Algorithm 1.

Algorithm 1 Image compression algorithm in the RGB case

1: Load the image
2: Check dimensions
3: Convert image to double
4: Extract the three color channels
5: Define the DCT function and apply it
6: Define a cover mask and apply it
7: Define the inverse DCT function and apply it
8: Gather all the channels in the compressed image

The JPEG compression algorithm relies on the division of the image into 8×8 blocks and so we have to check the adaptability of the image dimensions and, eventually, resize them. For real cases this could be a drawback since it is not applicable to any picture.
The RGB image loaded is stored as 8-bit so it is better to convert all the pixel values in the range $[0, 1]$ to have more precision and avoid imperfections in the computations for the DCT transform.

```
% im2double converts an image to double precision
Image = im2double(Image);
```

In Section 4 we presented the structure of an RGB image with the three channels for color red, green and blue. The compression will be applied to each of them separately in order to apply it homogeneously to each pixel. Then the DCT-2 for a block of dimension 8×8 is defined and applied to each channel. If we have a matrix T that defines the DCT, we can apply it a block of the right dimensions as in Formula 4.

$$\text{New}_A = T \times A \times T^{'} \tag{4}$$

```
% dctmtx defines a n-by-n discrete cosine transform matrix
T = dctmtx(8);
dct = @(block_struct) T * block_struct.data * T';
% blockproc processes the image matrix applying the function given in
   input to each block of size [m n]
Rb = blockproc(R,[8 8],dct);
Gb = blockproc(G,[8 8],dct);
Bb = blockproc(B,[8 8],dct);
```

For each block it is defined a mask of dimension 8×8 that has just 0 or 1 and, when it is applied to the transformed block, it makes null a specific number of coefficients. The underlying idea is that in this way we are deleting a specific number of coefficients in each block, for each channel, reducing the information contained in the image. The goal of the compression is usually to find the good trade-off of number of coefficient and resolution of the new image. In our interface, the non-zero values of the mask are chosen manually from a slider in the integer range $[0, 64]$.
If we want to recover the compressed image we should apply the inverse DCT as in Formula 5 to all the channels.

$$\text{Inv} = T^{'} \times \text{New}_A \times T \tag{5}$$

Finally, we collect the three colors into the RGB tensor and we display it in the "Compressed Image" window.
Once the image is compressed, three metrics [4] are computed and displayed in the left panel.

- **MSE**: the mean (M) squared (S) error (E) is the most common used metric for the image quality estimation. It represents the second discrete moment of the error between each pixel in two images. Usually to compute it, we split the computations for each channel and then we take the mean of the values. For each color we can compute it following the formula

$$\mathrm{MSE}_{\mathrm{channel}} = \frac{1}{MN}\sum_{m=0}^{M}\sum_{n=0}^{N}[\hat{g}(n,m) - g(n,m)]^2,$$

where $g(x,y)$ and $\hat{g}(x,y)$ are two images represented by each pixel in position (x,y). The global MSE is computed by

$$\mathrm{MSE} = \frac{1}{3}(\mathrm{MSE}_{\mathrm{R}} + \mathrm{MSE}_{\mathrm{G}} + \mathrm{MSE}_{\mathrm{B}}).$$

- **PSNR**: the peak (P) signal (S) to noise (N) ratio (R) is a metric used to calculate the ratio between the maximum possible signal power and the power of the distorting noise which affects the quality of its representation. It is introduced to measure the quality of reconstruction of lossy image compression codes. It is computed by

$$\mathrm{PSNR} = 10 \cdot \log_{10}\left(\frac{\mathrm{Peak\_Value}^2}{\mathrm{MSE}}\right),$$

where the $\mathrm{Peak}_{\mathrm{Value}}$ is the maximal in the image data. In our case, since the data are converted in doubles in range $[0,1]$, we will alway keep it equal to one. Usually we consider a good compression, one where the PSNR is more or equal to $30dB$ and so in the GUI the panel will be colored by gree in that case and in red otherwise.

- **Compression Ratio**: this percentage is related to the initial slider choice of the number of coefficients that we want to keep during the compression. It represents the proportion of global values kept after the DCT application.

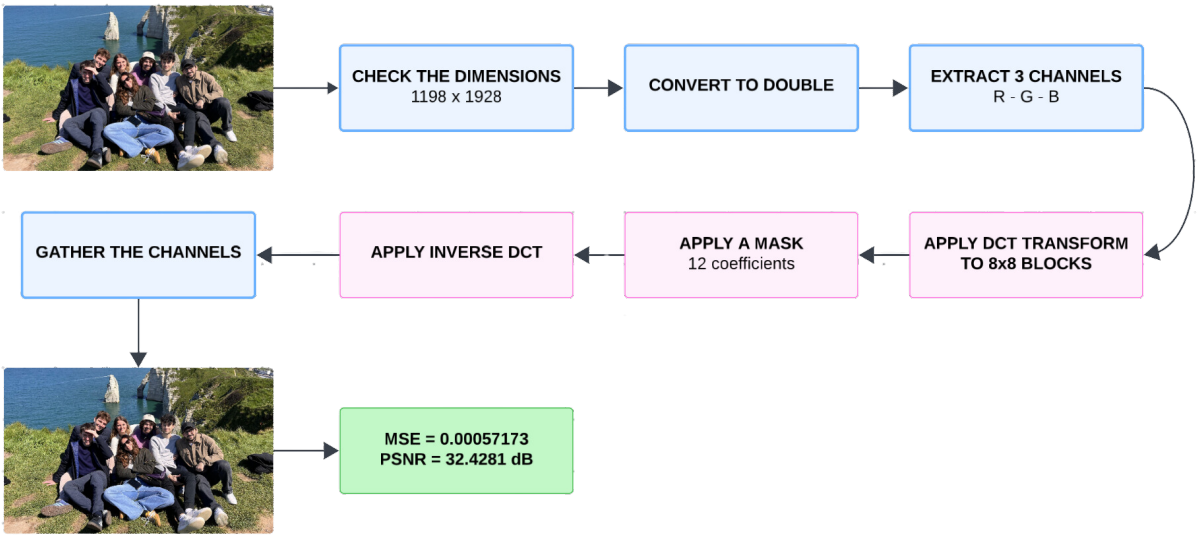

Figure 4: Compression algorithm flow chart. *Road trip in Normandie, France*

6.1 GrayScale Variation

In image processing it could be necessary to use an image in GrayScale instead of RGB. In order to do so we can apply the Matlab function **rgb2gray** that converts the truecolor image to a Grayscale by eliminating the hue and saturation information while retaining the luminance. In the compression function the procedure will be the same but we will not work with the three channels separately but on the direct pixel matrix as shown in Algorithm 2.

Algorithm 2 Image compression algorithm in the GrayScale case

1: Load the image
2: Check dimensions
3: Convert image to double
4: Define the DCT function and apply it
5: Define a cover mask and apply it
6: Define the inverse DCT function and apply it

6.2 Examples

In this subsection we will explore two examples, one in RGB and the other one in GrayScale. First of all we can test the algorithm efficiency with a picture taken by me in Ancona, *Arco di Traiano*.

(a) Original Image (b) 9 coefficients compression (c) 4 coefficients compression

Figure 5: Three versions of the same image

From Figure 5 we can notice that 9 is already a good number of coefficients to obtain a good reconstruction of the original image while just 4 do not perform correctly and the image in Figure 5c has a worse resolution. We can see the the error metrics and the new compressed size in the tables below. The original image size is 1.1 mB so the gain with the first compression in terms of storage saving is 34% while the second one is 65%, despite the worse quality.

Coefficients kept	9
MSE	0.00000856
PSNR	50.6757 dB
Compression Ratio	56.25%
Size	730 kB

Coefficients kept	4
MSE	0.00188610
PSNR	27.24444 dB
Compression Ratio	9.3750%
Size	391 kB

Table 1: Comparison of the errors and the size of the image after having compressed

The next example will be on a GrayScale image, in particular a photo shot taken by me in Paris, *'Place Igor-Stravinsky - Fontaine Stravinsky'*.

(a) Original Image

(b) 9 coefficients compression

(c) 4 coefficients compression

Figure 6: Three versions of the same image

From Figure 6 we can notice that 9 is still a good number of coefficients to obtain a good reconstruction of the original image. In the tables below there are all the metrics that describe the compression.

Coefficients kept	9
MSE	0.00019744
PSNR	37.0457 dB
Compression Ratio	56.25%
Size	1.5 mB

Coefficients kept	4
MSE	0.00377709
PSNR	24.2284 dB
Compression Ratio	9.3750%
Size	606 kB

Table 2: Comparison of the errors and the size of the image after having compressed

7 Conclusion

The paper presented the application of the Discrete Cosine Transform and its particular application to the JPEG compression algorithm. From the examples presented in Section 6.2 we noticed that the role of this mathematical tool is crucial for image storing. The choice of the number of coefficients in our implementation is the decision on which the algorithm is based and finding a good trade off in image size and compression ratio is the goal to obtain a good resolution for the result.
The implementation presented above is the classic one but other alternatives have been studied like the Overlapped Block Transform that reduces the "blockiness" of the compressed picture or the Adaptive Block Transform that adapts it the to different sized images. The future developments of this algorithm could be really impactful on the image processing field leading to faster applications in the web field.

A Fourier Tranform

Consider f as a real, continuous, differentiable with continuous derivative and periodic function in one variable. Taking a suitable series of numbers c_n, f can be written as

$$f(x) = \sum_{n \in \mathcal{Z}} c_n e^{inx},$$

where the RHS is called Fourier Series [3] expansion of the function. The Fourier coefficients c_n are computed by

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(t) e^{-int} dt. \tag{6}$$

Assuming N as an even number, we can define a numerical approximation of the integral 6 as a discretization of the function f as evaluated in only N equally spaced points. Using the trapezoidal rule and assuming the periodicity $f(0) = f(2\pi)$:

$$c_n = \frac{1}{N} \sum_{k=0}^{N-1} f(\frac{2k\pi}{N})(e^{2\pi i/N})^{-kn}, \tag{7}$$

where we can define the terms as $y_k = f(2k\pi/N)$ and $w = e^{2\pi i/N} = \cos\frac{2\pi}{N} + i \sin\frac{2\pi}{N}$. We can rewrite 7 as

$$c_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k w^{-kn}. \tag{8}$$

We look for a trigonometric polynomial p which interpolates f as

$$p(x) = \sum_{h=-N/2}^{N/2-1} \gamma_h e^{ihx}, \quad p(2k\pi/N) = f(2k\pi/N) = y_k \quad k = 0, 1, \cdots, N-1,$$

that is equivalent to the linear algebraic system

$$y_k = \sum_{h=-N/2}^{N/2-1} \gamma_h w^{hk} \quad k = 0, 1, \cdots, N-1. \tag{9}$$

B GUI Code

The GUI has been implemented on Matlab following the Algorithm 1. The following listing contains the source code.

```
function GUIcompressor

    % Create the figure
    fig = uifigure('Name', 'Image Compressor', 'Position', [100 100 800 500]);

    originalImage = [];
    compressedImage = [];

    % Button to load the image
    loadButton = uibutton(fig, 'Position', [20 350 100 30], 'Text', 'Load Image', ...
                          'ButtonPushedFcn', @(btn, event) loadImage());
    loadButton.FontSize = 14;
    loadButton.FontWeight = 'bold';
    loadButton.BackgroundColor = [0.1, 0.4, 0.8];  % Azzurro
    loadButton.FontColor = [1, 1, 1];

    % Slider to select the image compression level
    compressionSlider = uislider(fig, 'Position', [150 335 200 3], 'Limits', [0 64], 'Value', 32);
    compressionSliderLabel = uilabel(fig, 'Position', [150 345 200 30], 'Text', 'Number of Coefficient to be kept');

    % Button to apply the compression
    compressButton = uibutton(fig, 'Position', [20 300 100 30], 'Text', 'Apply', ...
                              'ButtonPushedFcn', @(btn, event) compressImage());
    compressButton.FontSize = 14;
    compressButton.FontWeight = 'bold';
    compressButton.BackgroundColor = [0.2, 0.7, 0.2];  % Verde
    compressButton.FontColor = [1, 1, 1];

    % Area to visualize the original image
    originalImageAxes = uiaxes(fig, 'Position', [400 250 300 200]);
    title(originalImageAxes, 'Original Image');
    fig.Color = [0.9, 0.9, 0.9];
    originalImageAxes.Color = [0.95, 0.95, 0.95];

    % Area to visualize the compressed image
    compressedImageAxes = uiaxes(fig, 'Position', [400 20 300 200]);
    title(compressedImageAxes, 'Compressed Image');
    compressedImageAxes.Color = [0.95, 0.95, 0.95];

    % Comment labels to the buttons
    loadButton.Tooltip = 'Select image to be loaded';
```

```
    compressButton.Tooltip = 'Apply the selected compression';
    compressionSlider.Tooltip = 'Select the number of coefficients to
       be kept in the blocks';

    % MSE label
    msePanel = uipanel(fig, 'Position', [145 245 300 40], '
       BackgroundColor', [0.8 0.8 0.8]);
    msePanel.BorderType = 'line';
    msePanel.BorderWidth = 1;
    msePanel.BorderColor = [0.2 0.2 0.2];
    mseLabel = uilabel(msePanel, 'Position', [5 5 290 30], 'Text', '
       MSE:');
    mseLabel.BackgroundColor = [1 1 1];
    mseLabel.FontSize = 14;
    mseLabel.FontWeight = 'bold';
    mseLabel.HorizontalAlignment = 'center';
    mseLabel.FontColor = [0 0.4 0.8];

    % PSNR label
    psnrPanel = uipanel(fig, 'Position', [145 215 300 40], '
       BackgroundColor', [0.8 0.8 0.8]);
    psnrPanel.BorderType = 'line';
    psnrPanel.BorderWidth = 1;
    psnrPanel.BorderColor = [0.2 0.2 0.2];
    psnrLabel = uilabel(psnrPanel, 'Position', [5 5 290 30], 'Text', '
       PSNR:');
    psnrLabel.BackgroundColor = [1 1 1];
    psnrLabel.FontSize = 14;
    psnrLabel.FontWeight = 'bold';
    psnrLabel.HorizontalAlignment = 'center';
    psnrLabel.FontColor = [0 0.4 0.8];

    % Compression Ratio label
    crPanel = uipanel(fig, 'Position', [145 185 300 40], '
       BackgroundColor', [0.8 0.8 0.8]);
    crPanel.BorderType = 'line';
    crPanel.BorderWidth = 1;
    crPanel.BorderColor = [0.2 0.2 0.2];
    crLabel = uilabel(crPanel, 'Position', [5 5 290 30], 'Text', '
       Compression Ratio:');
    crLabel.BackgroundColor = [1 1 1];
    crLabel.FontSize = 14;
    crLabel.FontWeight = 'bold';
    crLabel.HorizontalAlignment = 'center';
    crLabel.FontColor = [0 0.4 0.8];

    % Function to load the image
    function loadImage()
        [file, path] = uigetfile({'*.jpg;*.png;*.bmp', 'Images (*.jpg,
            *.png, *.bmp)'});
        if isequal(file, 0)
            return;
        end
```

```
        imgPath = fullfile(path, file);
        originalImage = im2double((imread(imgPath)));
        imshow(originalImage, 'Parent', originalImageAxes);
    end

    % Function to apply the compression
    function compressImage()
        if isempty(originalImage)
            uialert(fig, 'Load an Image!', 'Error');
            return;
        end

        compressionLevel = compressionSlider.Value;

        % Resize the image
        [rows, cols, ~] = size(originalImage);
        rows = floor(rows / 8) * 8;
        cols = floor(cols / 8) * 8;
        I = originalImage(1:rows, 1:cols, :);

        % Convert to double
        I = im2double(I);

        % Extract the three channels
        R = I(:,:,1);
        G = I(:,:,2);
        B = I(:,:,3);

        % Define and apply the DCT
        T = dctmtx(8);
        dct = @(block_struct) T * block_struct.data * T';
        Rb = blockproc(R,[8 8],dct);
        Gb = blockproc(G,[8 8],dct);
        Bb = blockproc(B,[8 8],dct);

        % Create a dinamic mask
        mask = createMask(compressionLevel);

        % Apply the mask to the blocks
        Rb2 = blockproc(Rb,[8 8],@(block_struct) mask .* block_struct.
           data);
        Gb2 = blockproc(Gb,[8 8],@(block_struct) mask .* block_struct.
           data);
        Bb2 = blockproc(Bb,[8 8],@(block_struct) mask .* block_struct.
           data);

        % Define and apply the inverse DCT
        invdct = @(block_struct) T' * block_struct.data * T;
        IR = blockproc(Rb2,[8 8],invdct);
        IG = blockproc(Gb2,[8 8],invdct);
        IB = blockproc(Bb2,[8 8],invdct);

        % Unify the channels to obtain the compressed image
```

```
138         compressedImage = cat(3,IR,IG,IB);
139 
140         % Compute MSE
141         mse_R = mean((R(:) - IR(:)).^2);
142         mse_G = mean((G(:) - IG(:)).^2);
143         mse_B = mean((B(:) - IB(:)).^2);
144         MSE = (mse_R+mse_B+mse_G)/3;
145 
146         format long
147 
148         % Compute PSNR
149         psnr_total = 10 * log10(1 / MSE);
150 
151         % Compute Compression Ratio
152         total_coeffs = numel(mask) * (rows / 8) * (cols / 8);
153         kept_coeffs = sum(mask(:) ~= 0) * (rows / 8) * (cols / 8);
154         compression_ratio = kept_coeffs / total_coeffs * 100;
155 
156         mseLabel.Text = sprintf('MSE: %.8f ', MSE);
157         psnrLabel.Text = sprintf('PSNR: %.4f dB%',psnr_total);
158         if psnr_total < 30
159             psnrLabel.FontColor = [1, 0, 0];
160         else
161             psnrLabel.FontColor = [0, 1, 0];
162         end
163         crLabel.Text = sprintf('Compression Ratio: %.4f%%',
               compression_ratio);
164 
165         % Show the compressed image
166         imshow(compressedImage, 'Parent', compressedImageAxes);
167         imwrite(compressedImage, 'boh.jpg');
168     end
169 
170     % Function to create the dynamic mask
171     function mask = createMask(level)
172         threshold = round(level);
173         mask = ones(8);
174         for i = 1:8
175             for j = 1:8
176                 if i + j > threshold
177                     mask(i, j) = 0;
178                 end
179             end
180         end
181     end
182 end
```

References

[1] N. Ahmed, T. Natarajan, and K.R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23(1):90–93, 1974.

[2] Saravanan Chandran. Color image to grayscale image conversion. pages 196 – 199, 04 2010.

[3] Ernesto Salinelli and Franco Tomarelli. *Discrete Dynamical Models*. Springer, 1 edition, 2014.

[4] U. Sara, M. Akter, and M. Uddin. Image quality assessment through fsim, ssim, mse and psnrâĂŤa comparative study. *Journal of Computer and Communications*, 7:8–18, 2019.

[5] Gilbert Strang. The discrete cosine transform. *SIAM Review*, 41(1):135–147, 1999.

[6] Inc. Âĺ 1994-2024 The MathWorks. dct - documentazione, 2024. Available on line.