Algorithms for Lossy Data Compression

Author:
Juanita Gómez

Advisor:
Germán Combariza

Department of Mathematics
Pontificia Universidad Javeriana

May 30, 2018

Abstract

The main objective of this thesis is providing an overview of different data compression methods, and their respective implementation using Python. Interest in the compression of information surged when the need of providing a more efficient way of transmitting information in the distance was necessary. The Morse code is one of the first attempts to compress data. Growth in the communication industry then provided the platform for Information Theory to be developed.

Currently, information is shared around the globe, even in outerspace, at great rates, and the demmand for more efficient and qualified methods of compressing data is crucial. Making files smaller can represent significant savings in storage and time, being fundamental for the proper behaviour of every computational application. Although lossless methods provide ensured fidelity their compression rates are not comparable to what can be achieved through lossy methods that take into account probability, redundancy and human perception.

Being able to understand how the processes of data compression are structured can give us a solid substrate to push boundaries on the informatical limitations of the systems nowadays, and propose new approaches to data codification, transmition and management using mathemathical resources wisely. Altogether this can result in pushing computational power to its maximum.

The first chapter explores some generalities on the topic, including basic definitions and a short historical scan. The following chapter is about simple ways of compressing data. Chapter 3 describes quantization types. After that, Chapter 4 and Chapter 5 describe how spectral and transform methods are applied to compress information. The last chapter exemplifies the methods described above in actual applications of image compression using a Python environment to perform the algorithmical operations.

Contents

Chapter 1

Introduction

1.1 Information Theory

One of the things that we need to understand before talking about data compression, is how data is stored. First, we have to talk about **bits**, which are the basic unit of information in computing. The word bit is shortening of the words "**B**inary dig**IT**". Computers use the base-2 system, which means they use binary numbers that have only the values 0 and 1. Bits then, can only have one of these two values and in computers, they are bundled together in groups of 8, which we call **bytes**. Bytes, are the units in which RAM and hard disk capacities are measured and they are also the way in which file sizes are measured. [1]

1.2 Data Compression

"Data compression is the art of reducing the number of bits needed to store or transmit data" [12]. The basic idea behind data compression, is that we have a certain amount of information and we want to change its representation in such a way that it occupies less space in a computer. This is possible because data is represented inside the computer in a format that is longer than what it is strictly necessary. According to information theory, the latter fact is called redundancy. Data is compressed then by reducing its **redundancy**. Depending on the type of information that we are using, different compression algorithms can be used because different data types contain redundancy in a different way.

The most common measure used to express the performance of a compression method is the **compression ratio** which is defined as follows [17]

$$\text{Compression ratio} = \frac{\text{Size of output stream}}{\text{Size of input stream}}$$

A compression ratio of 0.6 means that data occupies 60% of the original size after being compressed. The units of this value are usually bit per bit because this value equals the number of bits in the compressed stream needed, on average, to compress one bit in the input stream.

Another quantity that is used when talking about compression, is the **compression factor** which is the inverse of the compression ratio, defined in the following way.

$$\text{Compression factor} = \frac{\text{Size of input stream}}{\text{Size of output stream}}$$

In this case, if we have values greater than 1, it means the data was compressed and if they are less than 1, the data was expanded.

1.3 Justification

The main reason for doing data compression is the limitation in time and space that we find when dealing with information. Processing and transfering information takes a lot of time when the size of the files is big, and reducing them can definitely make processes faster. Additionally, there is a finite number of resources available to store information; computers, disks, hard drives have a limited capacity and buying extra space can get really expensive. Bigger disks, memories and even space in the cloud cost money and usually we want to store a huge amount of information in files, images, videos and audios which can represent a large sum of money. For this reason, reducing the space that this information occupies is also a way of saving space and money.

1.4 Types of Data Compression

There are two types of data compression, lossy and lossless. We refer to lossy compression when we talk about a method that changes data representation in such a way that information is lost in the process and it cannot be recovered. This kind of data compression discards data that is not important. For example, audio and images can be compressed lossy because humans are not sensitive to every detail of the information that they contain. Lossy methods usually work on separating the important from the unimportant data so that they can be treated independently. Common lossy data compression examples are MP3 for audio, JPEG for images and MPEG for videos.[13]

On the other hand, lossless methods compress information without loosing any. Data can be decompressed exactly to its original value. These methods work by exploiting redundancies in data to find a shorter way to represent the same information. They do this by generating statistical models that use probability of the data to encode it producing shorter sequences. Common lossless algorithms are Huffman Coding and arithmetic coding.

1.5 Historical Overview

One of the first examples of data compression can be traced to Morse code developed in the early nineteenth century. Instead of requiring twenty six separate characters,

the letters in the alphabet could be represented by a sequence of impulses in two differentiable states. This method even considers how frequent the use of each letter is in the English language to assign the least amount of impulses to represent them.

About one century after the Morse code implementation, the communication industry gave birth to information theory. Prior to this, probability was not considered throughly in data transmission and storage associated to the telegraph. After these developments, information theory applied to data compression found further use in cryptography and thermodynamics.

This statistical approach resulted in the first hardware codewords being modified in order to reduce the amount of storage they required and minimize the error in their transmission. With the advent of software in the 1970s data compression became more critical and necessary, even though the information was almost exclusively text.

One decade after image files data compression became essential, and further research lead to methods such as the pointer-based encoding. All the previous enumerated methods implied no data was lost after their decoding, but with the 90s came the need of lossy compression directly related to the limitations of human perception.[21]

Chapter 2

Encoding

2.1 Variable Length Encoding

One of the reasons why information like text is redundant is because it is encoded in ASCII or Unicode which employ fixed-lenght codes. Thus, variable length encoding, which is a statistical based method, can eliminate redundancy by replacing each symbol with a variable length codeword. The idea is to assign short codes to the common symbols and long codes to the rare symbols. One trivial example of this is Morse code in which the most common letters in English like `E` are represented with short codes, in this case "." and the rare letters like `Q` and `Z` are represented by long codes, "__._" and "__.." respectively.

When we use variable length encoding for compressing, first we have to find out the frequency of occurrence in which each symbol occurs. For this we have to do a list of probabilities called the statistical distribution of the symbols. [18] Given a string of characters, the probability of a character can be determined by counting the frequency of the character and dividing it by the length of the string. This probabilities are related to a value that is called **entropy** which measures the smallest number of bits needed on average to represent a symbol a_i and it is defined as:

$$E_i = -P_i \log_2 P_i$$

where P_i is the probability with which the symbol a_i occurs in the data. Thus the entropy of a set of data with n symbols is defined as:

$$E = -\sum_{i=1}^{n} P_i \log_2 P_i$$

The entropy of data is largest when the n probabilities are equal. This is used to measure the data redundancy which is defined as the difference between a symbol set's largest possible entropy and its actual entropy as follows: [17]

$$R = \left[-\sum_{i=1}^{n} P \log_2 P \right] - \left[-\sum_{i=1}^{n} P_i \log_2 P_i \right] \tag{2.1}$$

For doing variable length encoding, we also have to take into account the **prefix property**. This property consists in that once a bit pattern has been assigned as the code of a symbol, no other codes should start with this pattern, i.e. the pattern cannot be the prefix of any other code. This property is the one that ensures that the encoding is not ambiguous and that there is only one way of decoding the symbols. Now we show a brief example of variable length encoding presented in [17]

2.1.1 Implementation

Consider the four symbols a_1, a_2, a_3 and a_4 with probabilities $0.49, 0.25, 0.25$ and 0.01. With these values, the data has entropy $-(0.49 \log_2 0.49 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.01 \log_2 0.01 \approx 1.57$ One correct way to encode the symbols a_1, a_2, a_3 and a_4 taking into account the prefix property is displayed in the following table.

Table 2.1: Variable Length Coding Example

Symbol	Probability	Code
a_1	0.49	1
a_2	0.25	01
a_3	0.25	000
a_4	0.01	001

If we code this step by step we could do it in the following way. The first value "1" is assigned to the first symbol a_1. As this can't be the prefix to any other code, we would continue with symbol a_2 coded as 01 and no other codes could start with 01 so the symbols a_3 and a_4 are coded as 000 and 001 respectively. Note that the shortest code is assigned to the most commonly occurring symbol and the longer codes are assigned to the symbols with the least probability.

One common example of variable length encoding is **Huffman Coding** which builds a list of the symbols in descending order of their probabilities. It then constructs a tree, with a symbol at every leaf, from the bottom up. At each step of the algorithm the two symbols with smallest probabilities are selected and added to the top of the partial tree. Then, they are deleted from the list, and replaced with a symbol that represents the two original symbols. This is repeated until the list is reduced to just one auxiliary symbol. [17]

2.2 Run Length Encoding

Run length encoding takes advantage of the repetition of symbols in a data set to reduce redundancy. If a symbol d occurs n consecutive times in the input stream, we can replace the n occurrences called **run length of** n with only one pair nd. This method is not useful for example at the time of compressing texts because there are not many repetitions of letters, usually they are not longer than two. However it works better for image compression. If we have a bi-level image (one in which each

pixel is represented by only one bit), there are only 2 types of symbols that correspond to white and black pixels so there are runs of pixels that alternate between these two colors. [17]

The fact that runs have different lengths, suggests the use of variable length codes. This means that run length encoding is usually one step in a compression algorithm and it is found combined with variable length encoding, transforms or even quantization which will be explained ahead.

2.2.1 Implementation

A simple example of run length encoding is shown in the following figure.

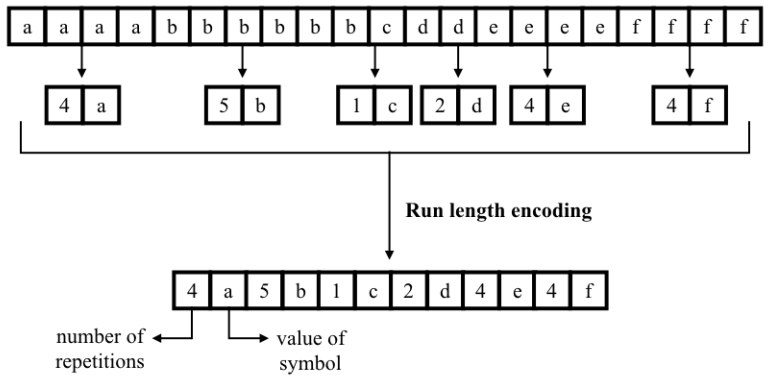

Figure 2.1: Run Length Encoding

In this example, a chain of continuous repeated symbols, is replaced by two values, the number of times that the symbol is repeated and the value of the symbol. Note that the initial stream has 20 symbols, while the output stream has only 12. This gives a compression ratio of $\frac{12}{20} = 0.6$

Chapter 3

Quantization

3.1 Introduction

The term "quantization" means "to restrict a variable to discrete values rather to a continuous set of values" [5]. In data compression, quantization is one of the simplest lossy methods. Quantization can be understood in two ways. If we have digital data, like real numbers, we can quantize them by either rounding them to the nearest integer or converting them into small numbers which take less space than larger ones. On the other hand, if we have analog data, such as a signal, quantizing means digitizing the signal by discretization. This means that for example if we want to generate 8-bit integers out of the signal, we divide it into $2^8 = 256$ intervals and the values within an interval are turned into a single number. Note that doing this means not being able to restore the original set of data for which this method is lossy. Now, we are going to discuss two types of quantization that can be used for data compression. [18]

3.2 Scalar Quantization

We use scalar quantization when the data we are using are numbers. There are several approaches to scalar quantization. Here we are going to mention a few of them but in all we assume that we have a set of 8-bit numbers input one by one from an input stream. The following are used in cases were data is uniformly distributed and the probability of the values in the data is not used.

- One easy way for quantizing, is to delete the last significant four bits of each item achieving a compression factor of 2.
- As the input stream has 8-bit numbers, this means they are in the range from 0 to 255, so a better approach is to select a parameter d, which is the spacing between the values, and generate a sequence $d, 2d, \ldots kd$ where $(k+1)d > 255$ and $ks \leq 255$. With this sequence, we quantize the values of our input array to the nearest value of the sequence.
- Another approach similar to the one mentioned above is to divide the range of values into segments of size $2d+1$ and centering it on the range $[0, 255]$ which

ends up with a sequence in which any value in the range is no more than 16 units away from these values. As in the case before, we quantize the values of our input array to the nearest value of the sequence.

3.3 Vector Quantization

Unlike scalar quantization, vector quantization is done in groups and it is used when adjacent symbols in data are correlated. This is why it is useful for audio and image compression. In order to achieve vector compression we first set a value N which is the parameter that determines the size of the groups of symbols to work with, which are called **vectors**. After determining the size, the most important thing in vector quantization is to create a **codebook**, which is a group of vectors to which the data is going to be quantized. The idea is to read the data vector by vector and compress it by finding the vector in the codebook that is closest to it.

There are several ways to construct the codebook and it depends on the distribution of the symbols in the data. One way, not very efficient though, is to read the whole data and construct the best codebook for it. Another way is having a static codebook based on training documents. A third one, consists in modifying the codebook each time that new data is added and compressed.

Chapter 4

Spectral Methods

4.1 Introduction

One of the lossy methods used for data compression is called Low-Rank Matrix Approximation and it relies on spectral decomposition of a matrix. In this case, we are talking specifically about a set of data represented by a matrix in which compression is going to be done by reducing its rank. For example if we have a matrix A of dimension n that represents data points, we can re-represent it as a matrix of dimension d with $d < n$. [15] This method is very useful for image compression, for example, because as we will see later, digitally, images are represented as matrices.

4.2 Spectral Decomposition

In order to study the spectral decomposition of a matrix, first we need to talk about the **spectrum**. The spectrum of a linear operator, T, on an inner-product space ,$\mathbb{V}$, over $\mathbb{F} = \mathbb{C}$ or $\mathbb{R}$, is the set of $\lambda \in \mathbb{F}$ for which the operator $T_\lambda := T - \lambda I$ is singular (not invertible) with I the identity operator. Let $\mathbb{V}$ be a vector space over the scalar field $\mathbb{F} = \mathbb{C}$ or $\mathbb{R}$, a scalar $\lambda \in \mathbb{F}$ is an **eigenvalue** of a linear operator T on $\mathbb{V}$ if the null space:

$$\mathbb{N}_\lambda = \{\mathbf{x} \in \mathbb{V} : (T - \lambda I)\mathbf{x} = 0\} \tag{4.1}$$

of the linear operator $T_\lambda = T - \lambda I$ is non trivial. That is, the spectrum of T is the set of eigenvalues of this operator. If λ is an eigenvalue of T, then any non-zero $\mathbf{x} \in \mathbb{N}$ is called an **eigenvector** corresponding to the eigenvalue λ. [3]

Theorem 1 (Spectral Theorem)**.** *Let A be a $n \times n$ normal matrix. Then there is a unitary matrix U and a diagonal matrix $\Lambda = diag\{\lambda_1, \lambda_2, ...\lambda_n\}$ such that $A = U\Lambda U^*$ where U^* is the transpose-conjugate of U.* [3]

To proof this theorem, we have to prove a weaker result first:

Theorem 2 (Unitary-triangular matrix decomposition)**.** *For any $n \times n$ matrix A given, there exists a matrix U and an upper triangular matrix T, both unitary such that*

$$A = UTU^* \tag{4.2}$$

Proof. To prove this we are going to use induction. It is easy to see that the theorem holds trivially for $n = 1$. Now, for $n > 1$, let $A \in \mathbb{C}^{n\times n}$ and let $\mathbf{v_1}$ be an unit eigenvector associated with λ_1 eigenvalue of A. Now we can build an orthonormal basis $\{\mathbf{v_1}, \mathbf{v_2}, ...\mathbf{v_n}\}$ of $\mathbb{C}^n$ by extending $\mathbf{v_1}$ and construct the unitary matrix

$$V = [\mathbf{v_1}\mathbf{v_2}...\mathbf{v_n}]$$

Now note that

$$V^*AV = \begin{bmatrix} \mathbf{v_1^*} \\ \vdots \\ \mathbf{v_n^*} \end{bmatrix} \begin{bmatrix} \lambda_1\mathbf{v_1} * \dots * \end{bmatrix} = \begin{bmatrix} \lambda_1 & \vdots & * & \dots & * \\ \dots & \dots & \dots & \dots & \dots \\ O & \vdots & & B & \end{bmatrix}$$

where O is a zero column vector (because $\mathbf{v_2}, \dots \mathbf{v_n}$ are orthogonal to $\mathbf{v_1}$) and B is an $(n-1)\times(n-1)$ matrix. Applying the induction hypothesis, we can say that

$$B = WT_1W^*$$

where W is a unitary matrix and T_1 is an upper triangular, both $(n-1)\times(n-1)$ Now, let U be

$$U = V \begin{bmatrix} 1 & \vdots & O \\ \dots & \dots & \dots \\ O & \vdots & W \end{bmatrix}$$

we have the following

$$\begin{aligned} U^*AU &= \begin{bmatrix} 1 & \vdots & O \\ \dots & \dots & \dots \\ O & \vdots & W^* \end{bmatrix} \begin{bmatrix} \lambda_1 & \vdots & * & \dots & * \\ \dots & \dots & \dots & \dots & \dots \\ O & \vdots & & B & \end{bmatrix} \begin{bmatrix} 1 & \vdots & O \\ \dots & \dots & \dots \\ O & \vdots & W \end{bmatrix} \\ &= \begin{bmatrix} \lambda_1 & \vdots & * & \dots & * \\ \dots & \dots & \dots & \dots & \dots \\ O & \vdots & & W^*BW & \end{bmatrix} = \begin{bmatrix} \lambda_1 & \vdots & * & \dots & * \\ \dots & \dots & \dots & \dots & \dots \\ O & \vdots & & T_1 & \end{bmatrix} \end{aligned}$$

where the matrix on the right is upper-triangular. As U is unitary, we can write A as $A = UTU^*$. [3] □

Using this last theorem, we can proceed to prove Theorem 1.

Proof. Relying on the last theorem, we know we can write matrix A as $A = UTU^*$. We will show that when A is normal, T is a diagonal matrix. Recall that for this result we assume A is normal, so we have:

$$\begin{aligned} U(T*T)U* =& (UT^*U^*)(UTU^*) = A^*A \\ =& AA^* = (UTU^*)(UT^*U^*) = U(TT^*)U^* \end{aligned}$$

This means that $T^*T = TT^*$. Note that the first entry of the left side is $|t_{11}|^2$ and the first entry of the right side is $|t_{11}|^2 + |t_{12}|^2 + \ldots + |t_{1n}|^2$, from what we have

$$|t_{11}|^2 = |t_{11}|^2 + |t_{12}|^2 + \ldots + |t_{1n}|^2$$

so

$$t_{12} = 0, t_{13} = 0, \ldots t_{1n} = 0$$

and we can write T as

$$\begin{bmatrix} 1 & \vdots & O \\ \ldots & \ldots & \ldots \\ O & \vdots & T_1 \end{bmatrix} \tag{4.3}$$

with T_1 upper triangular. If we apply Theorem 1 again, we obtain $T_1^*T_1 = T_1T_1^*$ and so

$$t_{23} = 0, t_{24} = 0, \ldots t_{2n} = 0$$

using the same procedure as before. If we repeat the process again, we end up having that $T = \Lambda$ is a diagonal matrix concluding that $A = U\Lambda U^*$ as we wanted. [3] □

4.3 Singular Value Decomposition

The Singular Value Decomposition pretends to extend the concept of spectral decomposition for square matrices, to a decomposition for rectangular matrices. For this, let B an $m \times n$ and A its Gram matrix defined by

$$A = BB^*$$

First we need to verify that this matrix A is normal so that we can use the spectral decomposition theorem.

Theorem 3. *Let B be an $m \times n$ matrix and A its Gram matrix $A = BB^*$ is positive semi-definite and self-adjoint hence it is normal.*

For the proof of this theorem see [3]

Now that we have A normal, we can say it has spectral decomposition

$$A = U\Lambda U^* \tag{4.4}$$

where $\Lambda = \text{diag}\{\lambda_1, \ldots, \lambda_m\}$ is a diagonal matrix and $U = \left[\mathbf{u_1}, \ldots, \mathbf{u_m}\right]$ is a unitary matrix, where the $(\lambda_j, \mathbf{u_j})'s$ are pairs of eigenvalue-eigenvector of the matrix A. As A is positive semi-definite, all its eigenvalues are positive and hence we can write $\lambda_j = \sigma_j^2$ where

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r \geq \sigma_{r+1} = \ldots = \sigma_m = 0$$

for some $r \leq m$. (Note that if $r = m$ then $\{\sigma_{r+1}, \ldots \sigma_m\}$ is an empty set) Using this, we can write Λ as

$$\Lambda = \text{diag}\{\sigma_1^2, \ldots, \sigma_r^2, 0 \ldots, 0\}$$

Now let

$$\Sigma_r = \text{diag}\{\sigma_1, \ldots, \sigma_r\}$$

This drives us to the following definition.

Definition 1. *The diagonal entries $\sigma_1 \ldots \sigma_r$ of Σ_r defined before are called the (non zero)* ***singular values*** *of B.* [3]

Theorem 4 (Reduced SVD theorem)**.** *Let B be an $m \times n$ matrix with* $\text{rank}(B) = r$*. Then there exist unitary matrices U_1 and V_1 with dimensions $m \times r$ and $n \times r$ respectively such that B has the reduced singular value decomposition*

$$B = U_1 \Sigma_r V_1^* \tag{4.5}$$

where $\Sigma_r = \text{diag}\{\sigma_1, \ldots, \sigma_r\}$ and $\sigma_1 \geq \sigma_2 \ldots \geq \sigma_r \geq 0$

Proof. Let $A = BB^*$, then as we saw before, A has the spectral decomposition

$$A = U \Lambda U^*$$

for some U unitary and $\Lambda = \text{diag}\{\sigma_1^2, \ldots, \sigma_m^2\}$ with

$$\sigma_1 \geq \sigma_2 \ldots \geq \sigma_r \geq \sigma_{r+1} = \ldots = \sigma_m = 0$$

and $0 \leq r \leq min\{m, n\}$. We can see that $\text{rank}(A) = \text{rank}(BB^*) = \text{rank}(B) = \text{rank}(\Lambda) = r$ so now we are going to write U as $U = [U_1 \vdots U_2]$ where U_1 is the $m \times r$ matrix that consists of the first r columns of U. Now note that from (4.4) we have that

$$U_1^* B B^* U_1 = U_1^* U \Lambda U^* U_1 = [I_r O] \Lambda [I_r O]^* = (\Sigma_r)^2 \tag{4.6}$$

and $U_2^* B B^* U_2 = U_2^* U \Lambda U^* U_2 = O$, so

$$U_2^* B = O \tag{4.7}$$

Now, let V_1 be

$$V_1 = B^* U_1 \Sigma_r^{-1} \tag{4.8}$$

And note that

$$U_1 \Sigma_r V_1^* = U_1 \Sigma_r (B^* U_1 \Sigma_r^{-1})^* = U_1 \Sigma_r (\Sigma_r^{-1 *} U_1^* B) = B \tag{4.9}$$

so V_1 satisfies (4.5). From (4.5) and (4.8) we have that

$$\begin{aligned} U^*(B - U_1 \Sigma_r V_1^*) &= \begin{bmatrix} U_1^* B \\ U_2^* B \end{bmatrix} - \begin{bmatrix} I_r \\ O \end{bmatrix} \Sigma_r V_1^* \\ &= \begin{bmatrix} U_1^* B - \Sigma_r V_1^* \\ O \end{bmatrix} = O \end{aligned}$$

As U^* is nonsingular, then $U_1^* B - \Sigma_r V_1^*$ and so $B - U_1 \Sigma_r V_1^*$; hence $B = U_1 \Sigma_r V_1^*$ [3] □

Now, with this result, we can proof easily the stronger version of this theorem.

Theorem 5 (Full SVD theorem)**.** *Let B an $m \times n$ matrix with* $\text{rank}(B) = r$*. Then there exist unitary matrices U and V with dimensions $m \times m$ and $n \times n$ respectively such that B has the full singular value decomposition*

$$B = USV* \tag{4.10}$$

where S is an $m \times n$ matrix defined by

$$S = \begin{bmatrix} \Sigma_r \\ \dots \\ O \end{bmatrix} \text{ or } S = \begin{bmatrix} \Sigma_r \vdots O \end{bmatrix} \tag{4.11}$$

if $r = n < m$ or $r = m < n$ respectively. If B has real entries, then U and V can be chosen orthogonal.

[3], [20]

Proof. To proof this, let $A = BB^*$ again and $V_1 = B^*U_1\Sigma_r^{-1}$ and $U = [U_1 \vdots U_2]$ defined as before. From this and equation 4.6 we have

$$V_1 * V_1 = \Sigma_r^{-1}U_1^*BB^*U_1\Sigma_r^{-1} = \Sigma_r^{-1} = \Sigma_r^{-1}({\Sigma_r}^2\Sigma_r^{-1} = I_r \tag{4.12}$$

This means that V_1 is orthogonal in $\mathbf{C}^n$ and we can extend it to an unitary matrix $V = [V_1 \vdots V_2]$ introducing matrix V_2 that is made up with orthonormal columns. Using this,

$$B = U_1\Sigma_r V_1^* = U_1[\Sigma_r O][V_1 V_2]^* = [U_1 U_2] \begin{bmatrix} \Sigma_r & O \\ O & O \end{bmatrix} V^* = USV^* \tag{4.13}$$

[3] □

Note that to compute the SVD of a rectangular matrix B we have to compute the square roots of the eigenvalues of BB^*. From the full SVD we can get reduced the SVD keeping the first r columns of U and V, and the other way around it can also be done by extending Σ_r to S, U_1 to U and V_1 to V as shown in the proof (check figure 4.1).

The following result is important for our application of SVD decomposition

Theorem 6. *Unitary transformations preserve singular values*

Proof. Let $B \in \mathbb{C}^{m,n}$ and W_1 and W_2 unitary matrixes of dimensions m and n. Then

$$(W_1BW_2)(W_1BW_2)^* = W_1BW_2W_2^*B^*W_1^* = W_1BB^*W_1*$$

so $(W_1BW_2)(W_1BW_2)^*$ and BB^* are similar and have the same eigenvalues so the singular values of W_1BW_2 are the same singular values of B [3] □

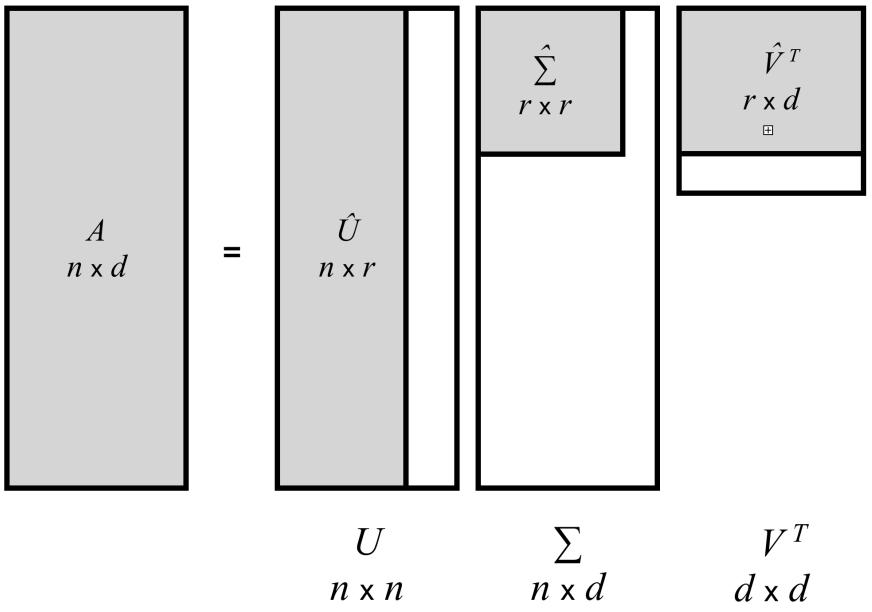

Figure 4.1: Full SVD and Reduced SVD

4.4 Low Rank Approximation

Now that we studied how to decompose a matrix B in its principal components, we can use this to approximate B as a lower rank matrix. If we want to approximate our matrix B of rank n to a matrix B_k of rank $k < n$, we have to compute the SVD of B, $B = USV^*$ and choose the k greatest singular values of this. Note that as we said before, in our matrix S, the singular values are ordered from greatest to least so doing this equals to computing

$$B_k = U_k S_k V_k^*$$

where U_k is the $n \times k$ matrix that results from keeping the first k columns of U, V_k^* is the $k \times d$ matrix that results from keeping the first k rows of V^*, and S_k is the $k \times k$ matrix that results from keeping the first k rows and columns of S corresponding to the largest singular values of B. [15]

4.4.1 Approximation Error

The first thing we want to know is how accurate is this approximation. For this we need to reference the Frobenius norm in matrices.

Definition 2. *The **Frobenius norm** of an $m \times n$ matrix $B = [b_{jk}]$ is defined by*

$$||B||_F = \left(\sum_{j=1}^{m} \sum_{k=1}^{n} |b_{jk}|^2 \right)^{1/2}. \tag{4.14}$$

[3], [20]

However, with this definition it is not easy to calculate the Frobenius norm of a matrix so we use the following theorem.

Theorem 7. *Let $B \in \mathbb{C}^{m,n}$ with $rank(B) = r$ then:*

$$||B||_F = \left(\sum_{j=1}^{r} \sigma_j^2 \right)^{1/2} \tag{4.15}$$

where σ_i are the singular values of B.

Proof. Let $A = BB^*$, and lets calculate $Tr(A)$ the trace of A.

$$\mathrm{Tr}(A) = \sum_{k=1}^{n} a_k, k = \sum_{k=1}^{n} \left(\sum_{l=1}^{m} b_k, \overline{lb_k, l} \right) \tag{4.16}$$

$$= \sum_{k=1}^{n} \sum_{l=1}^{m} |b_l, k|^2 = ||B||_F^2 \tag{4.17}$$

Note that $\mathrm{Tr}(A)$ agrees with the Frobenius norm of B. However, it is known that $\mathrm{Tr}(A)$ is the sum of the eigenvalues of A and recall that this are the squares of the singular values of B so:

$$||B||_F = (\mathrm{Tr}(A))^{1/2} = \left(\sum_{j=1}^{r} \sigma_j^2) \right)^{1/2} \tag{4.18}$$

where $\sigma_1, \ldots, \sigma_r$ are the singular values of B.[3] □

Now with this norm, we can calculate the error involved in doing lower-rank matrix approximation, and we can also prove that if we want to reduce the rank of a matrix this is the best way to do it.

Theorem 8. *Let B be a $m \times n$ matrix with rank r and singular values $\sigma_1 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = 0$. Then for any integer d, with $1 \leq d \leq r$ the following sum provides the best approximation of B under the Frobenius norm by all matrices of rank $\leq k$.*

$$B_d = \sum_{j=1}^{d} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \tag{4.19}$$

With this approximation, the error is given by:

$$||B - B_d||_F^2 = \sum_{j=d+1}^{r} \sigma_j^2 \tag{4.20}$$

where

$$||B - B_k||_F^2 \leq ||B - C||_F^2 \tag{4.21}$$

for all $m \times n$ matrices C with rank $\leq d$. B_d is the unique best approximation of B under the Frobenius norm.

Proof. Let S be as defined in (4.1) and let S_d be the matrix obtained by replacing each $\sigma_1, \ldots, \sigma_d$ with 0. This way, we have:

$$B - B_d = US_dV^* \tag{4.22}$$

so

$$||B - B_d||_F^2 = ||US_dV^*||_F^2 = ||S_d||_F^2 = \sum_{j=d+1}^{r} \sigma_j^2 \tag{4.23}$$

as we wanted. Now to prove (4.21) suppose $C \in \mathbf{C}^{m,n}$ with rank $k \leq d$ provides the best approximation to B under the Frobenius norm so $||B - C||_F^2 \leq ||B - B(d)||_F^2$. It follows that

$$||B - C||_F^2 \leq \sum_{j=d+1}^{r} \sigma_j^2. \tag{4.24}$$

If we consider U and V the unitary matrices in the SVD of B, let $G = U^*CV$ with rank k so $C = UGV^*$ and:

$$\begin{aligned} ||B - C||_F =& ||USV^* - UGV^*||_F & (4.25)\\ =& ||U(S - G)V^*||_F = ||S - G||_F (\text{using theorem 6}) & (4.26) \end{aligned}$$

For $||S - G||_F$ to be minimum among all G, G should be given by:

$$G = \begin{bmatrix} \Sigma_r' & 0 \\ 0 & 0 \end{bmatrix} \tag{4.27}$$

where $\Sigma_r' = \text{diag}\{g_1, g_2, \ldots, g_r\}$ with $g_j \geq 0$ so

$$||S - G||_F^2 = \sum_{j=1}^{r} |s_{j,j} - g_j|^2 \tag{4.28}$$

Recall that the rank of G is $k \leq r$ so only k of the $g_1, g_2, \ldots, g_r$ are non zero. In this case, the minimum of the sum is achieved only when these k non-zero entries match the largest k values of $\sigma_1, \ldots, \sigma_r$. This means $g_1 = \sigma_1, \ldots, g_k = \sigma_k$ and $g_j = 0$ for $j > k$. With this, we get:

$$||B - C||_F^2 = ||S - G||_F^2 = \sum_{j=k+1}^{r} \sigma_j^2 \tag{4.29}$$

Using (4.24) we can conclude that $k = d$, hence:

$$||B - C||_F^2 = \sum_{j=d+1}^{r} \sigma_j^2 \tag{4.30}$$

and as $C = UGV^*$ with

$$G = \begin{bmatrix} \Sigma_r' & 0 \\ 0 & 0 \end{bmatrix} \tag{4.31}$$

implying that $C = B(d)$ as we wanted. [3] □

4.5 Data Compression

Now, we can take a look at how to use rank approximation for data compression. Let D be an $n \times p$ matrix containing data where each row is a data point. If n and p are both large the storage of this matrix is going to occupy a large space in the computer and any operations or analysis is going to take a lot of running time. For this reason it is useful to find a way of storing this data occupying less computer space losing the least amount of information possible.

$$\text{Let } D = \begin{bmatrix} d_1^T \\ d_2^T \\ \vdots \\ d_n^T \end{bmatrix} \tag{4.32}$$

Note that each d_i^T is a row of the Data matrix D.

1. **Calculate center of data points**
   The first thing we have to do is to calculate the average c_i of each i-th column D building the vector $c = [c_1, c_2, \ldots, c_p]^T$

2. **Center data points**
   Now, we use these averages to center these data points at the origin.

$$\text{Let } E = \begin{bmatrix} d_1^T - c^T \\ d_2^T - c^T \\ \vdots \\ d_n^T - c^T \end{bmatrix} \tag{4.33}$$

3. **Calculate the SVD of** E Calculate the matrices V^T, U and Σ of the singular value decomposition of $E = U\Sigma V^T$. Recall that in Σ the singular values are organized in the diagonal from greatest to least.

4. **Rank-k approximation** Decide the singular values that are negligible. If we want a rank-k approximation, we choose the k greatest, which correspond to the first k entries of the diagonal matrix Σ, and we take the corresponding k right singular vectors $\mathbf{v_1}, \ldots, \mathbf{v_s}$ in the matrix $\hat{V} = [v_1, \ldots, v_s]$.

5. **Store necessary information**
   Store the matrices $\hat{V}, Y = E\hat{V}$ and c

6. **Get back an approximate to original** D After we stored our information in a different way, it is important to know how to get back to a matrix D' which is an approximation to to the original matrix D. Note that if we calculate

$E' = Y\hat{V}^T$, we obtain

$$\begin{aligned} E' =& E\hat{V}\hat{V}^T \\ =& E[\mathbf{v_1}, \ldots, \mathbf{v_k}, 0, \ldots, 0]V^T \\ =& U\Sigma V^T[\mathbf{v_1}, \ldots, \mathbf{v_k}, 0, \ldots, 0]V^T \\ =& U\Sigma \begin{bmatrix} I_{k\times k} & 0 \\ 0 & 0 \end{bmatrix} V^T \\ =& U\Sigma_k V^T \end{aligned}$$

So E' is the $rank - k$ approximation of E and

$$D' = E' + \begin{bmatrix} c^T \\ c^T \\ \vdots \\ c^T \end{bmatrix} \tag{4.34}$$

is a matrix approximate to D with rank k. [2]

4.5.1 Storage difference

Note that if D is size $n \times p$, then it has np entries while $\hat{V}, Y$ and c have nk, pk and p entries respectively. If k is comparatively smaller than p, there is a significant compression in terms of storage. If, for example, we are storing matrices as numpy arrays in Python, each float element of the array occupies 8 bytes. Hence, a matrix of $n \times p$ would occupy $n \times p \times 8$ bytes of memory but if we store $\hat{V}, Y$ and c we would occupy $(nk + pk + p) \times 8$ bytes of memory.

4.6 Implementation

Now we are going to see an example from [2] of this algorithm for a small matrix. Consider the matrix D which represents 5 data points in $\mathbb{R}^3$

$$\begin{bmatrix} 3.4 & 2 & 6 \\ 3.4 & 5 & 0 \\ 0.4 & 2 & 3 \\ 0.4 & 2 & 6 \\ -2.6 & -1 & 0 \end{bmatrix}$$

The vector c corresponding to the center of the points is $c = [1, 2, 3]^T$

Using this we can calculate the corresponding E matrix and find its SVD decomposition.

$$E = \begin{bmatrix} 2.4 & 0 & 3 \\ 2.4 & 3 & -3 \\ -0.6 & 0 & 0 \\ -0.6 & 0 & 3 \\ -3.6 & -3 & -3 \end{bmatrix}$$

$$E = \begin{bmatrix} -0.51 & -0.24 & 0.69 & 0.07 & 0.44 \\ -0.15 & 0.85 & -0.10 & 0.28 & 0.41 \\ 0.06 & -0.04 & -0.26 & -0.77 & 0.57 \\ -0.22 & -0.45 & -0.61 & 0.49 & 0.38 \\ 0.82 & -0.12 & 0.27 & 0.28 & 0.41 \end{bmatrix} \begin{bmatrix} 6.75 & 0 & 0 \\ 0 & 5.61 & 0 \\ 0 & 0 & 1.50 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} -0.66 & 0-39 & 0.65 \\ -0.43 & 0.52 & -0.74 \\ -0.62 & -0.76 & -0.17 \end{bmatrix}^T$$

Now we calculate the 2-rank approximation because the third singular value is significantly smaller than the other two. Now we compute

$$Y = E[v_1 v_2] = \begin{bmatrix} -3.4371 & -1.3669 \\ -0.9983 & 4.7688 \\ 0.3935 & -0.2316 \\ -1.4697 & -2.5249 \\ 5.5117 & -0.6454 \end{bmatrix} \tag{4.35}$$

At the end we only have to store $Y, \hat{V} = [v_1, v_2]$ and c. In this case we do not se a lot the benefits of the procedure because of the small dimensions of our initial matrix, but for larger data sets, the difference between the stored and the initial information is considerable as we saw before.

Now lets compute D' which is an approximation of our initial matrix D, so we can figure out how good is this approximation.

$$E' = Y\hat{V}^T = \begin{bmatrix} 1.7264 & 0.7694 & 3.1796 \\ 2.4955 & 2.8909 & 3.0255 \\ 0.3474 & 0.2885 & 0.0673 \\ 0.0108 & 0.6730 & 2.8429 \\ 3.8637 & 2.6988 & 2.9297 \end{bmatrix}$$

$$D' = \begin{bmatrix} 2.7264 & 2.7694 & 6.1796 \\ 3.4955 & 4.8909 & 0.0255 \\ 0.6526 & 1.7115 & 2.9327 \\ 0.9892 & 1.3270 & 5.8429 \\ 2.8637 & 0.6988 & 0.0703 \end{bmatrix}$$

D' looks similar to D but we can calculate the Frobenius norm of the difference of this two matrices.

$$||D - D'||_F = 1.497$$

Chapter 5

Transform Methods

5.1 Introduction

A transform is a tool that is used to solve problems in different areas. In this case, the idea is to change mathematical quantities to a different form in which we can take advantage of useful properties. A transform itself does not compress data but, as it allows to change representation of quantities, it enables to use compression algorithms like quantization or variable length encoding, described above, in a better way. [18]

The formal definition of a transform in mathematics is *"a rule used to exchange one set of objets for another"*. [10] One of the important properties that a transform should have for it to be useful in a practical way, is that it should be invertible. With this in mind a transform, will be an invertible linear transformation on a vector space. [10]

5.1.1 Types of transforms

There are two types of transforms, the orthogonal transforms and the subband transforms. An **orthogonal** transform converts a list of n correlated numbers into transform coefficients of which, the first is large and contains much of the information of the original data and the rest are small and contain less important features. The computation of this transforms is done using the inner product of finite-length signals. In this case, data is compressed by replacing small coefficients by variable length codes or by quantizing. The classic example of an orthogonal transform is the Fourier transform, which we will introduce ahead.

A **subband** transform, known also as a wavelet, is computed by convolving the signal but the information is encoded by portions, corresponding to a particular spatial scale. For example, in the case of image compression, it separates the vertical, horizontal, and diagonal constituents of the image, so each can be compressed differently. This can be useful to control the relative amounts of error in different parts of the input signal frequency. [6] Later, we will talk about the Haar transform which is the simplest wavelet transform usually employed in image compression.

5.2 Fourier Transform

The Fourier transform is a linear transformation that is used to change functions represented in the time domain, to functions represented in the frequency domain [19]. A function $g(t)$ is said to be represented in the time domain if the parameter used is the time, i.e. it varies over time. $g(t)$ is said to be represented in the frequency domain when we try to represent it as a combination of waves where the function varies with respect to the frequency. Usually, this transform is used for periodic functions, however it can be also used for some non periodic functions under certain conditions.

Definition 3. *A function $g(t)$ is **periodic** if there exists a constant $P \neq 0$ such that $g(t+P) = g(t)$ for all values of t.*

If a function is not periodic, the Fourier expansion will have infinite number of frequencies.

Definition 4. *For a continuous function $g(t)$, the **Fourier transform** and its inverse are given by*

$$G(s) = \int_{-\infty}^{\infty} g(t)[\cos(2\pi st) - i\sin(2\pi st)]dt \tag{5.1}$$

$$g(t) = \int_{-\infty}^{\infty} G(t)[\cos(2\pi st) + i\sin(2\pi st)]dt \tag{5.2}$$

[17]

As this method is usually implemented computationally, we have to use the **Discrete Fourier transform** [16].

Definition 5. *For a continuous function $g(t)$, the **Discrete Fourier transform** is given by*

$$G(s) = \sum_{t=0}^{n-1} g(t)\left[\cos\left(\frac{2\pi st}{n}\right) - i\sin\left(\frac{2\pi st}{n}\right)\right] \tag{5.3}$$

$$= \sum_{t=0}^{n-1} g(t)e^{-ifs}, 0 \leq f \leq n-1, \tag{5.4}$$

and its inverse will be

$$g(t) = \frac{1}{n}\sum_{s=0}^{n-1} G(s)\left[\cos\left(\frac{2\pi st}{n}\right) + i\sin\left(\frac{2\pi st}{n}\right)\right] \tag{5.5}$$

$$= \sum_{t=0}^{n-1} se^{ist}, 0 \leq s \leq n-1, \tag{5.6}$$

[17]

If $g(t)$ is a periodic function, we say that $G(s)$ is its corresponding Fourier series.

In practice, we usually find discrete signals, so we are going to calculate the discrete Fourier transform of a signal $\mathbf{h} = (h_0, \ldots, h_{N-1})$. The idea is to find a vector $\mathbf{b} = (b_0, \ldots, b_{N-1})$ so that for each $k = 0, \ldots, N-1$:

$$h_k = \sum_{n=0}^{N-1} b_n e^{2\pi i n k/N} \tag{5.7}$$

It is easy to see, that for finding these coefficients we can write:

$$b_k = \frac{1}{N} \sum_{n=0}^{N-1} h_n e^{-2\pi i n k/N}, k = 0, \ldots, N-1 \tag{5.8}$$

[7]
Note that 5.7 and 5.8 are linear transformations so they can be expressed using matrices. Consider the matrix W of dimension $N \times N$, with entries $(j,k) = (1/\sqrt{N})e^{2\pi i j k/n}$, then:

$$\mathbf{b} = W\mathbf{h}$$
$$\mathbf{h} = \overline{W}\mathbf{b}$$

where $\bar{W}$ is the complex conjugate matrix of W. Hence, if we denote W_k the k-th column of the matrix W, we can also see the Fourier transform as

$$W_k = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 \\ e^{2\pi i k/N} \\ \vdots \\ e^{2\pi j k/N} \\ \vdots \\ e^{2\pi (N-1) k/N} \end{bmatrix} \tag{5.9}$$

[10]

Note that each column W_k oscillates with period N, and the frequency of oscillation increases as $2\pi k/N$ gets closer to π, i.e., when k is almost $N/2$. Taking this into account, high frequency oscillations correspond to columns at the middle of the matrix W, and the columns on the left and right of W oscillate at lower frequencies.

In this sense, the transform of a signal $\mathbf{x}$ is a vector $\hat{\mathbf{x}}$ that contains the amplitudes of the fundamental frequencies of $\mathbf{x}$. The transform gives us control over the frequencies that are not relevant to the interpretation of the signal's quality. This is the reason why we can use this transform to compress data by removing information that is less important. When we replace an entry in $\hat{\mathbf{x}}$ with zero, we remove the

corresponding frequency from $\mathbf{x}$. This is done through quantizing, reducing and it reduces precision of coefficients providing a method for lossy compression. The k-th Fourier coefficient of a signal is what is needed to determine the importance of its frequency component in the direction W_k

5.12 can be used to write the 2D Fourier transform. Recall that the W_k was the $k-$th column of the Fourier transform matrix W. We can define an array f_{uv}, that works as a basis for the 2D Fourier transform in which the (j,k) entry is defined as:

$$f_{uv}(j,k) = W_u(j)\overline{W_v(k)} = \frac{1}{N}e^{2\pi iju/N}e^{-2\pi ikv/N} \tag{5.10}$$

With this, we can define the two dimensional Fourier transform $\hat{f}$ as:

$$\hat{f} = \frac{1}{N}\sum_{n=0}^{N-1} f(j,k)e^{-2\pi i(ju-kv)/N} \tag{5.11}$$

5.2.1 Implementation

Lets consider the following example from [10] in which we consider the approximation of a vector $\mathbf{y}$ using its Fourier Transform. Let $\mathbf{y}$ be described as follows

$$\mathbf{y} = \begin{bmatrix} 20.0 \\ 14.2 \\ 0.0 \\ -10.0 \\ -15.0 \\ -10.0 \\ 0.0 \\ 14.2 \end{bmatrix} \tag{5.12}$$

The Fourier transform calculated as it was explained before.

$$\hat{\mathbf{y}} = \begin{bmatrix} 4.74 \\ 24.47 \\ 1.77 \\ 0.27 \\ -1.20 \\ 0.27 \\ 1.76 \\ 24.47 \end{bmatrix} \tag{5.13}$$

The first thing that we have to do, to evaluate our approximation, is to get the magnitudes of this vector. In this case, as all the numbers are real, we only take the

absolute value of each number.

$$\mathbf{Abs}(\hat{\mathbf{y}}) = \begin{bmatrix} 4.74 \\ 24.47 \\ 1.77 \\ 0.27 \\ 1.20 \\ 0.27 \\ 1.76 \\ 24.47 \end{bmatrix} \tag{5.14}$$

Note that there are some values that are really small compared to others. Lets see what happens if we make $\hat{\mathbf{y}}(3) = \hat{\mathbf{y}}(4) = \hat{\mathbf{y}}(5) = \hat{\mathbf{y}}(6) = \hat{\mathbf{y}}(7) = 0$. Then we obtain a new vector $\hat{\mathbf{z}}$:

$$\hat{\mathbf{z}} = \begin{bmatrix} 4.74 \\ 24.47 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 24.47 \end{bmatrix} \tag{5.15}$$

Note that in this case, the 3 nonzero coordinates of $\hat{\mathbf{z}}$ are equal to the corresponding 3 coordinates of $\hat{\mathbf{y}}$ because all of these values are real and positive in $\hat{\mathbf{y}}$. However, if this is not the case, in $\hat{\mathbf{z}}$ we would have only the norm of the values. For this reason, after computing our vector $\hat{\mathbf{z}}$, we have to take the corresponding nonzero coordinates in vector $\hat{\mathbf{y}}$ to recover all the information of the values. We are going to call this $\hat{\mathbf{w}}$. In this case,

$$\hat{\mathbf{w}} = \begin{bmatrix} 4.74 \\ 24.47 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 24.47 \end{bmatrix} \tag{5.16}$$

To verify what is the error of this approximation, we first find $\mathbf{w}$, a vector that has $\hat{\mathbf{w}}$ as its Fourier transform. We obtain the following:

$$\mathbf{w} = \begin{bmatrix} 18.98 \\ 13.91 \\ 1.68 \\ -10.56 \\ -15.63 \\ -10.56 \\ 1.68 \\ 13.91 \end{bmatrix} \tag{5.17}$$

Now we can calculate $\mathbf{z} - \mathbf{y}$

$$\mathbf{w} - \mathbf{y} = \begin{bmatrix} 1.02 \\ 0.28 \\ -1.68 \\ 0.56 \\ 0.63 \\ -0.56 \\ -1.68 \\ 0.29 \end{bmatrix} \tag{5.18}$$

This looks like a relatively good approximation of vector $\mathbf{y}$. However evaluating its error depends on the objectives of our approximation. What we did, can be seen as a projection of a vector into a subspace of smaller dimension. With this in mind, if we want to use this for compression, it works better when the subset of the data type space from which we will select vectors to compress is thin in several directions.

5.3 Wavelet Transform

We saw before that the Fourier transform can be used to reveal properties of a signal in the frequency domain, however it does not allow us to analyze a signal in time and frequency simultaneously because it does not tell us when a frequency is active in a specific time. Wavelet transforms solve this problem because given a signal we can select a time interval and use the wavelet transform to isolate the frequencies in that specific interval. In other words, the idea of wavelets is to analyze a function or a signal according to scale. It shows the relationship between time-frequency analysis and waves concentrated in a small area. [17] *"The wavelet transform is a tool that cuts up data or functions or operators into different frequency components, and then studies each component with a resolution matched to its scale"* [4]

5.4 Haar Transform

The Haar transform is the simplest wavelet transform and it works by calculating averages and difference of values in a matrix of correlated values, to produce another matrix which is sparse or nearly sparse, i.e., a matrix in which many of its entries are 0. [8] The Haar transform is based on the Haar functions whose basis functions are defined in the following way:

$$h_0(x) := h_{00}(x) = \frac{1}{\sqrt{N}}, \text{ for } 0 \leq x \leq 1, \tag{5.19}$$

$$h_k(x) := h_{pq}(x) = \frac{1}{\sqrt{N}} \begin{cases} 2^{p/2}, & \frac{q-1}{2^p} \leq x < \frac{q-1/2}{2^p}, \\ -2^{p/2}, & \frac{q-1/2}{2^p} \leq x < \frac{q}{2^p}, \\ 0, & \text{otherwise for } x \in [0,1] \end{cases} \tag{5.20}$$

[17] With this definition, a matrix A_N of order $N \times N$ has elements i, j defined by $h_i(j)$ where $i = 0, 1, \ldots, N-1$ and $j = 0/N, 1/N, \ldots (N-1)/N$ The Haar transform of a matrix B of order $N \times N$ with $N = 2^n$ is the product $A_N P A_N$.

5.4.1 Implementation

For a one dimensional array of n entries, with n being a power of 2 (if it is not a power of 2, we extend the array by appending copies of the last entries), we calculate the Haar transform by dividing the array into $n/2$ pairs of consecutive entries, computing the average and the difference of each pair which results in $n/2$ averages and $n/2$ differences and repeating this procedure again until we obtain one average and $n-1$ differences. The averages are a representation of the original values with a coarse resolution, while differences are called **detail coefficients** and they represent the extra data needed to reconstruct the original data. Because the values are correlated these differences are usually small values and along with the averages, they are sufficient to reconstruct the original values.[18]

Now, we are going to ilustrate how this works with 8 correlated values using an example from [18]. However there were some modifications made to some values in order to obtain accurate results in the calculations.

Let $v = (31, 32, 33.5, 33.5, 31.5, 34.5, 32, 28)$. Now we compute the four averages $(31+32)/2 = 31.5, (33.5+33.5)/2 = 33.5, (31.5+34.5)/2 = 33$, and $(32+28)/2 = 30$ and the four differences $31-32 = -1$, $33.5-33.5 = 0$, $31.5-34.5 = -3$, and $32-28 = 4$. With this we get the vectors $a_1 = (31.5, 33.5, 33, 30)$ and $d_1 = (-1, 0, -3, 4)$. Repeating this procedure for a_1 we get $a_2 = (32.5, 31)$ and $d_2 = (-2, 3)$. The last iteration of this gives us $a_3 = 32$ and $d_3 =$. Now the Haar transform of our original array v is constituted by this last average a_3 and the 7 differences of d_3, d_2 and d_1 together, i.e $h = (32, 1, -2, 3, -1, 0, -3, 4)$. Note that the Haar transform of a one-dimensional array of dimension N ends up being another array with one large entry which is the average of the original values and $N-1$ small entries which can be quantized or encoded with variable length codes to achieve compression.

These calculations can be done using matrix multiplication. If we have a vector v it is possible to construct a matrix M_1 in such a way that when it is multiplied by the vector, we get the averages and differences that we need. Consider for example the matrix

$$M_1 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{5.21}$$

Lets see how if we multiply this matrix by our vector v we get the same values that we had.

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 31 \\ 32 \\ 33.5 \\ 33.5 \\ 31.5 \\ 34.5 \\ 32 \\ 28 \end{bmatrix} = \begin{bmatrix} 31.5 \\ 33.5 \\ 33 \\ 30 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix} \tag{5.22}$$

Now taking A_2 and A_3 as follows, we can compute the next steps of the transformation:

$$M_2 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, M_3 = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.23}$$

$$M_2 \cdot \begin{bmatrix} 31.5 \\ 33.5 \\ 33 \\ 30 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix} = \begin{bmatrix} 32.5 \\ 31 \\ -2 \\ 3 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix}, M_3 \cdot \begin{bmatrix} 32.5 \\ 31 \\ -2 \\ 3 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix} = \begin{bmatrix} 32 \\ 1 \\ -2 \\ 3 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix} \tag{5.24}$$

Note that if we construct the matrix $M = M_3 M_2 M_1$, we can apply M to a vector v to obtain its transform. The matrix M is as follows:

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{5.25}$$

Note that

$$\begin{bmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 31 \\ 32 \\ 33.5 \\ 33.5 \\ 31.5 \\ 34.5 \\ 32 \\ 28 \end{bmatrix} = \begin{bmatrix} 32 \\ 1 \\ -2 \\ 3 \\ -1 \\ 0 \\ -3 \\ 4 \end{bmatrix} \tag{5.26}$$

This matrix M can also be used to compute its transform of a matrix D in the following way. First, we apply M to all the columns of the matrix D computing the following product MD. Then, we apply M to the rows of this result so we compute the product $(MD)^T$ and we calculate its transpose. This way, for a matrix D , we can campute its Haar transform by:

$$D_{\text{tr}} = (M(MD)^T)^T) = MDM^T \tag{5.27}$$

And for computing the inverse transform we can use the following:

$$M^{-1}(M^{-1} \cdot D_{\text{tr}}^T)^T \tag{5.28}$$

In applications, we use the normalized Haar transform which is orthonormal and thus it has the property that its inverse its simply its transpose, for which it is easier to use computationally. For $n = 8$, it can be defined as follows:

$$\begin{bmatrix} \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{5.29}$$

If we compute the Haar transform of our initial array v using this matrix, we obtain:

$$\begin{bmatrix} \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & \frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} & -\frac{1}{\sqrt{8}} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \cdot \begin{bmatrix} 31 \\ 32 \\ 33.5 \\ 33.5 \\ 31.5 \\ 34.5 \\ 32 \\ 28 \end{bmatrix} = \begin{bmatrix} 64\sqrt{2} \\ \sqrt{2} \\ -2 \\ 3 \\ -\frac{\sqrt{2}}{2} \\ 0 \\ -\frac{3\sqrt{2}}{2} \\ 2\sqrt{2} \end{bmatrix} \tag{5.30}$$

Chapter 6

Image Compression

6.1 Introduction

To use compression methods on images, first we have to understand how images are stored in a computer. The size of an image in terms of storage is determined by its number of pixels. Pixels are the small dots that create the images. When we say an image has a resolution of 1024×1024, it means the image is a square made of 1024×1024 pixels. In this case, we will be using two dimensional images stored as three dimensional arrays as we will explain ahead. However these algorithms can be used for n-dimensional images stored as m-dimensional arrays. In the case of 2D images stored as 3 dimensional arrays, each pixel is represented internally as a 24-bit number where the percentages of red, green and blue occupy 8-bits (1 byte) each. These 8-bits correspond to a number from 0 to 255 represented in binary, using RGB color model where colors are reproduced by adding different quantities of red, blue and green. For example, the color black is reproduced when the value of each of these three colors is 0 and the color white is reproduced when the value of each of these three colors is 255. As for each pixel we have 1 byte for each color, an image that has a resolution of 1024×1024 occupies $1024 \times 1024 \times 3 = 3145728$ bytes.

Lossy image compression is based on the fact that we can lose features of an image to which the eye is not sensitive. This is due to spatial redundancy which means that neighboring pixels are highly correlated. If we choose a specific pixel within an image it is very likely that pixels close to it are exactly the same color or very close to it. Image compression is achieved by reducing redundancy that is caused mainly because of the correlation of the pixels.

6.2 Image Compression SVD

One of the best ways to portrait how SVD works is to use it on image compression. The basic idea behind this, is to use the low rank approximation of the matrices that represent the RGB quantities of an image, to reduce the size of it. We are going to show how this is done using python with a code adapted from [11]

6.2.1 Implementation

In order to demonstrate this, we take the following image from [9] which has 3536×2824 pixels.

Figure 6.1: Original Image for Compression

The first thing we are going to do is to import the image that we want to compress and import the libraries that we are going to use for this code.

```
import numpy as np
import scipy.misc
from numpy import linalg
from PIL import Image
image = np.array(Image.open('Graphics/Javeriana.jpg')). #Import Image
```

When we import the image like this, using a numpy array, we get a 3 dimensional array $2824 \times 3536 \times 3$ corresponding to the 3 matrices of the RGB code which contain the intensities of red, blue and green.

Now we have to normalize the intensity values for each pixel, dividing them by 255 and we want to calculate the size in pixels of the image and the space needed to store the image in bytes.

```
image = image / 255   #Normalize the intensity values
row, col, _ = image.shape
print "The number of pixels of this image is: ", row, "x", col #Number of pixels of original image
image_bytes = image.nbytes  #Calculate original space occupied
print "The space (in bytes) needed to store this image is", image_bytes
```

With this, our array "image" ends up being a $2824 \times 3536 \times 3$ array of values from one to cero, "row" and "col" end up being the numbers 2824 and 3536 respectively

and "image_bytes" ends up being 239655936 which is $2824 \times 3536 \times 3 \times 8$.

Now we have to separate our array "image" in three 2-dimensional matrices in order to use each matrix color independently, and we do full singular value decomposition for each of this matrices.

```
red = image[:, :, 0]
green = image[:, :, 1]
blue = image[:, :, 2]
# Full Singular Value Decomposition
red_U, red_d, red_V = np.linalg.svd(red, full_matrices=True)
green_U, green_d, green_V = np.linalg.svd(green, full_matrices=True)
blue_U, blue_d, blue_V = np.linalg.svd(blue, full_matrices=True)
```

After this we get 6 matrices and 3 vectors corresponding to the matrices V, U and vector d of the singular value decomposition of each of the matrices "red", "green" and "blue".

Now, using the SVD decomposition we are going to do the k-rank approximation to compress each of the matrices. We will do this for different values of k to compare the images but we will only show the code once.

```
k=50 #Store only k rows
red_U_k = red_U[:, 0:k]
red_V_k = red_V[0:k, :]
green_U_k = green_U[:, 0:k]
green_V_k = green_V[0:k, :]
blue_U_k = blue_U[:, 0:k]
blue_V_k = blue_U[0:k, :]
red_d_k = red_d[0:k]
green_d_k = green_d[0:k]
blue_d_k = blue_d[0:k]
```

With this code, we get for each color, a matrix U_k that is $2824 \times k$, a matrix V_k that is $k \times 3536$ and a vector of length k. In this case we have $k = 50$ so we have $2824 \times 50, 50 \times 3536$ and 50 respectively. If we wanted to store this matrices, we would use 7633200 bytes. However, in section 2.6 we observed that the best way to store the data is storing matrices $\hat{V}, Y$ and vector c in Step 5, so we calculate this for each one of the three matrices as explained before and we calculate how many bytes are necessary to store them.

```
#Calculate data necessary to recover original approximation
red_c=np.mean(red, axis=0)
np.outer(np.ones([1,col]),red_c)
red_E=red-red_c
red_V_t= red_V.transpose()
red_HatV= red_V_t[:,0:k]
red_Y=np.dot(red_E,red_HatV)

green_c=np.mean(green, axis=0)
np.outer(np.ones([1,col]),green_c)
green_E=green-green_c
green_V_t= green_V.transpose()
green_HatV= green_V_t[:,0:k]
green_Y=np.dot(green_E,green_HatV)

blue_c=np.mean(blue, axis=0)
np.outer(np.ones([1,col]),blue_c)
blue_E=blue-blue_c
blue_V_t= blue_V.transpose()
blue_HatV= blue_V_t[:,0:k]
blue_Y=np.dot(blue_E,blue_HatV)

compressed_bytes = sum([matrix.nbytes for matrix in
                [red_Y, red_HatV, red_c, green_Y, green_HatV, green_c, blue_Y, blue_HatV, blue_c]])
```

With this, we get that the total compressed bytes for $k = 50$ is 7716864 and now we can calculate the storage ratio for this SVD.

```
storage_ratio = compressed_bytes/image_bytes
```

The storage ratio in this case is approximately 0.00322 which means that we are only storing 0.322% of the original information.

Now we can reconstruct an approximation of the original matrices using our stored data and observe the resulting image.

```
#Recover approximation of original image

red_E1=np.dot(red_Y,red_HatV.transpose())
red_cmat=np.outer(np.ones([1,row]),red_c)
red_aprox=red_E1+red_cmat

green_E1=np.dot(green_Y,green_HatV.transpose())
green_cmat=np.outer(np.ones([1,row]),green_c)
green_aprox=green_E1+green_cmat

blue_E1=np.dot(blue_Y,blue_HatV.transpose())
blue_cmat=np.outer(np.ones([1,row]),blue_c)
blue_aprox=blue_E1+blue_cmat

#Reconstruct the three dimensional array
image_reconstructed = np.zeros((row, col, 3))

image_reconstructed[:, :, 0] = red_aprox
image_reconstructed[:, :, 1] = green_aprox
image_reconstructed[:, :, 2] = blue_aprox

#Correct the pixels where intensity value is outside [0,1]
image_reconstructed[image_reconstructed < 0] = 0
image_reconstructed[image_reconstructed > 1] = 1

scipy.misc.imsave('Figure5_2.jpg', image_reconstructed)
```

The image that we obtain using $k = 50$ is the following.

Figure 6.2: Compressed Image with k=50

Now the last step is to calculate the Frobenius norm of the difference of our original matrices with our approximations and to calculate the relative error using the initial frobenius norm. As we have 3 different matrices, one for each color intensity, we calculate the average of the frobenius norm calculated per component.

```
error1=linalg.norm(red-red_aprox)
error2=linalg.norm(green-green_aprox)
error3=linalg.norm(blue-blue_aprox)
error=(error1+error2+error3)/3

frob1=linalg.norm(red)
frob2=linalg.norm(green)
frob3=linalg.norm(blue)
frob=(frob1+frob2+frob3)/3

relerror=error/frob
```

For this case we get a relative error of 0.2576. We observe that this image that we got using $k = 50$ is very different from the original image and the differences are actually perceived by the eye. For this reason, we are going to try with larger values for k, and present the resulting information in the table below.

Table 6.1: SVD Image compression: Storage, Ratio relative error with different k

Value of k	Compressed Storage	Compressed Ratio	R.Error	Image
2824 (orignal)	239655936	-	0	Figure 6.1
50	7716864	0.00322	0.2576	Figure 6.2
100	15348864	0.0640	0.1697	Figure 6.3
200	30612864	0.1277	0.1298	Figure 6.4
500	76404864	0.3188	0.0795	Figure 6.5
800	122196664	0.5099	0.0573	Figure 6.6

Figure 6.3: Compressed Image with k=100

Figure 6.4: Compressed Image with k=200

Figure 6.5: Compressed Image with k=500

Figure 6.6: Compressed Image with k=800

6.3 Image Compression Haar

Images can be compressed using transforms by transforming its pixels to a representation where they are not correlated. They can be encoded independently making it simpler to construct statistical models and apply compression algorithms like quantization. Image transforms are used to reduce the redundancy of the images by reducing sizes of most pixels, and to isolate the important parts of the image from the unimportant by separating the frequencies of the image. Low frequencies correspond to the important features of an image, while high frequencies are related to the details of it. For this reason, high frequency pixels can be heavily quantized allowing effective lossy compression in which only unimportant details of the image are lost.

Now we are going to see how the Haar Transform explained before can be used to compress images. The first thing that we have to do is to generalize the algorithm explained for a one dimensional array to a two dimensional matrix, which in this case represents an image. There are two ways to do this: the standard decomposition and the pyramid decomposition.

The standard decomposition of an array $N \times N$ consists in calculating the Haar transform for each row of the image first, which results in a column of averages and $N - 1$ columns of differences and then doing the same for each resulting column. With this procedure, we end up getting an average at the top-left corner of our

array, a row at the top containing averages of differences and the rest of the rows containing differences. This procedure can be seen in the figure 6.7 lelow

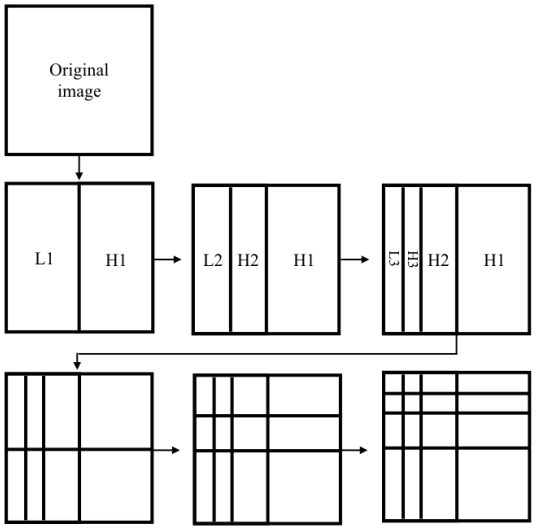

Figure 6.7: Standard Image Wavelet Transform

The pyramid decomposition alternates between rows and columns for the computation of the wavelet transform. This is done in the following way: first, we calculate the averages and differences for all the rows creating averages in the left half of the image and differences in the right half. Then we calculate the averages and differences for all the columns resulting in averages in the top-left quadrant of the image and differences in the rest. After that, we do the same in the top-left quadrant in each iteration, which at the end results in an average on the top-left corner and differences in the rest. This procedure can be seen in the figure 6.8.

In either method, we end with a transformed image that has an average at the top-left corner and small numbers in the rest of the image that can be compressed (lossy compression) using quantization and variable length codes.

A quantity that is used in this compression method is the **sparseness ratio** which measures the amount of coefficients discarded from the original image array by quantizing them to zero. It is defined as:

$$\textbf{Sparseness ratio} = \frac{\textbf{Number of nonzero wavelet coefficients}}{\textbf{Number of coefficients left}} \tag{6.1}$$

For a good image compression, we have to find a sparseness ratio that allows a good equilibrium between the compression ratio and the possibility of reconstructing images.

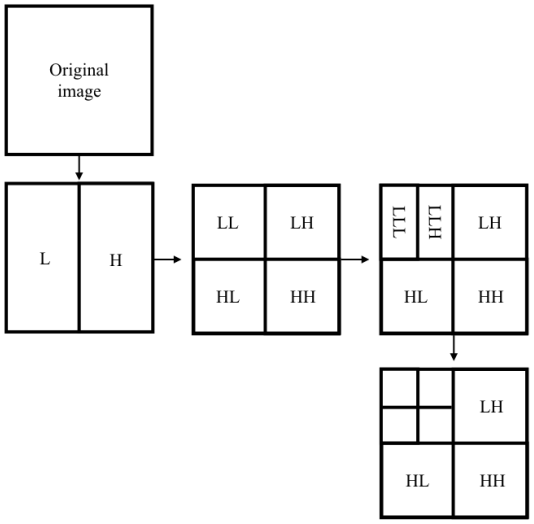

Figure 6.8: Pyramid Image Wavelet Transform

6.3.1 Implementation

As we said before, color images are stored as 3 dimensional array, which can be separated in the three components that correspond to the RGB representation. Now we are going to see a way of implementing this haar transform for image compression with the image that we had used before.
The first thing we are going to do is to import the image that we want to compress, normalize it, separate it in its RGB components and import the libraries that we are going to use for this algorithm

```
import numpy as np
import math
import scipy.misc
from PIL import Image
from numpy import linalg

        #Import and separate RGB components
        image = np.array(Image.open('Graphics/Javeriana.jpg'))    #Import Image
        image = image / 255.0
        red = image[:, :, 0]
        green = image[:, :, 1]
        blue = image[:, :, 2]
```

As we saw before, the Haar matrix is a $2^k \times 2^k$ for which it is necessary that our matrices to which we calculate the transform, are squared with dimension 2^k. For finding k, we find which is the biggest dimension of our matrices and we calculate its logarithm base 2 and approximate it to the next integer. After findin k, we have to complete our matrices corresponding to the intensities of RGB, with zeros, so that they have dimension $2^k \times 2^k$

```
        #Convert matrices into one dimension that is 2^k
        r=max(red.shape)
        s=int(pow(2,math.ceil(math.log(r,2))))
        r2 = np.zeros((s, s))
        r2[:red.shape[0], :red.shape[1]] = red

        g2 = np.zeros((s, s))
        g2[:green.shape[0], :green.shape[1]] = green

        b2 = np.zeros((s, s))
        b2[:blue.shape[0], :blue.shape[1]] = blue
```

After determining the dimensions of the matrices that we are working with, we have to calculate the haar matrix of dimension 2^k with the following code adapted from [14]

```
#Haar Matrix calculation (Dim n)
N = s
p = np.array([0, 0])
q = np.array([0, 1])
n = int(np.log2(N))

for i in np.arange(1, n):ÂăÂăÂăÂă
p = np.concatenate((p, i * np.ones(2**i)))ÂăÂăÂăÂă
t = np.arange(1, 2**i + 1)ÂăÂăÂăÂă
q = np.concatenate((q, t))

Hr = np.zeros([N, N])
Hr[0,:] = 1;

for i in np.arange(1, N):ÂăÂăÂăÂă
P = p[i]; # probably is actually (0, i)ÂăÂăÂăÂă
Q = q[i]; # probably is actually (0, iÂăÂăÂăÂă)ÂăÂăÂăÂă
for j in np.arange(N * (Q - 1) / (2**P), N * (( Q - 0.5) / (2**P)), dtype = int):ÂăÂăÂăÂăÂăÂăÂăÂăÂă
Hr[i, j] = 2**(PÂăÂăÂăÂă/2)ÂăÂăÂăÂă
for j in np.arange(N * (( Q - 0.5) / 2**P), (N * (Q / 2**P)), dtype = int):ÂăÂăÂăÂăÂăÂăÂăÂă
Hr[i, j] = -(2**(P/2))
Hr = Hr * (1/ np.sqrt(N))
```

With this Haar matrix, we can calculate the Haar transform of our 3 matrices corresponding to the RGB intensities of size 2^k.

```
#Calculating Haar transform
r2T=np.matmul(np.matmul(Hr,r2),np.transpose(Hr))
g2T=np.matmul(np.matmul(Hr,g2),np.transpose(Hr))
b2T=np.matmul(np.matmul(Hr,b2),np.transpose(Hr))
```

After having the Haar transforms of the 3 matrices, we can perform the quantization. For this, we choose a value of k, which is a threshold that establishes that the values with absolute value less than k, are quantized to zero.

```
#Quantization
k = 2
rquantT= np.asarray(r2T)
low= np.absolute(rquantT) < k # Where values are low
rquantT[low] = 0
nonzeror=np.sum(1 - low.astype(int))

vecg2T=np.absolute(np.reshape(g2T,s*s))
mig=np.min(vecg2T[np.nonzero(vecg2T)])
mag=np.max(vecg2T[np.nonzero(vecg2T)])

gquantT= np.asarray(g2T)
low= np.absolute(gquantT) < k # Where values are low
gquantT[low] = 0
nonzerog=np.sum(1 - low.astype(int))

vecb2T=np.absolute(np.reshape(b2T,s*s))
mib=np.min(vecb2T[np.nonzero(vecb2T)])
mab=np.max(vecb2T[np.nonzero(vecb2T)])

bquantT= np.asarray(b2T)
low= np.absolute(bquantT) < k # Where values are low
bquantT[low] = 0
nonzerob=np.sum(1 - low.astype(int))
```

After performing the quantization, calculating the inverse Haar transform is the first step towards reconstructing the image. After that, we the compute matrices of the original size by deleting the rows and columns that we added at the beginning. Finally, we combine the three matrices to get our 3 dimensional array, we multiply it by 255 to undo the normalization and save it as an image.

```
#Calculating Haar Inverse transform and Reconstructing the Image
rquant= np.matmul(np.matmul(np.transpose(Hr),rquantT),Hr)
gquant=np.matmul(np.matmul(np.transpose(Hr),gquantT),Hr)
bquant=np.matmul(np.matmul(np.transpose(Hr),bquantT),Hr)

image_reconstructed = np.zeros((red.shape[0], red.shape[1], 3))

image_reconstructed[:, :, 0] = rquant[:red.shape[0], :red.shape[1]]
image_reconstructed[:, :, 1] = gquant[:green.shape[0], :green.shape[1]]
image_reconstructed[:, :, 2] = bquant[:blue.shape[0], :blue.shape[1]]
```

```
scipy.misc.imsave('HaarCompressed4.jpg', image_reconstructed * 255.0)
```

For the error, we use the same approach that we used for low rank matrix approximation, with the Frobenius norm.

```
#Error Calculation
error1=linalg.norm(red-rquant[:red.shape[0], :red.shape[1]])
error2=linalg.norm(green-gquant[:green.shape[0], :green.shape[1]])
error3=linalg.norm(blue-bquant[:blue.shape[0], :blue.shape[1]])
error=(error1+error2+error3)/3

frob1=linalg.norm(red)
frob2=linalg.norm(green)
frob3=linalg.norm(blue)
frob=(frob1+frob2+frob3)/3

relerror=error/frob
```

Now, we try compressing our image for different values of "`val`" and we present the results in the following table.

Table 6.2: Haar Image Compression: Sparseness Ratio and Relative error with different `val`

Value of val	Sparseness ratio	R.Error	Image
original	1	0	Figure 6.1
2	0.00040	0.2471	Figure 6.9
0.5	0.00412	0.1643	Figure 6.10
0.25	0.01222	0.1272	Figure 6.11
0.125	0.03417	0.0938	Figure 6.12
0.0625	0.09317	0.0622	Figure 6.13

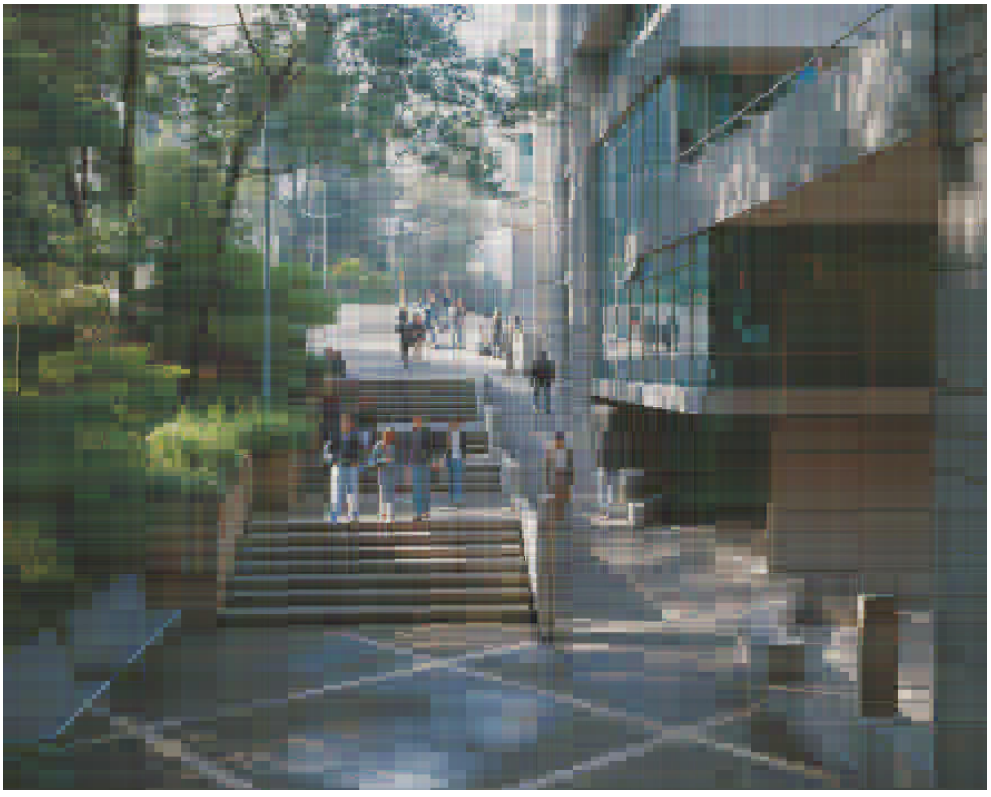

Figure 6.9: Compressed Image with k=2

Figure 6.10: Compressed Image with k=0.5

Figure 6.11: Compressed Image with k=0.25

Figure 6.12: Compressed Image with k=0.125

Figure 6.13: Compressed Image with k=0.0625

Appendix

A SVD Implementation for Data Files

Here, we show the implementation of the SVD algorithm using Python. We start with a file named "Data.csv" and we want to see how much a new file with an approximate matrix weigths, doing this $rank - k$ approximation. In the algorithm, the compressed data is stored in the file "compressedData.csv"

```
#Data Compression Using SVD

#Data.csv stores the data that we want to compress
import numpy as np
Data = np.loadtxt('Data.csv', delimiter=',')
#Center the data
c=np.mean(Data, axis=0)
np.outer(np.ones([1,1000]), c)
E=Data-c
print(E)

#Compute SVD of our matrix E
(U,S,Vt)=np.linalg.svd(E)
print(S)

#We can take significant singular values as the ones greater than 0.5
sign=np.where(S>0.5)
r=np.shape(sign)
print('The number of significant values r is',r[1])

#Generate matrices  Y,ÌĆV and the center  c storing only the first r columns of Vt
Vt_t=Vt.transpose()
HatV=Vt_t[:, 0:r]
Y=np.dot(E,HatV)

#Store  Y,ÌĆV ,c to "compressedData.csv"
np.savetxt('compressedData.csv', Y, delimiter=',')
f_handle = open('compressedData.csv', 'ab')
np.savetxt(f_handle, HatV, delimiter=',')
np.savetxt(f_handle, c, delimiter=',')
f_handle.close()

#Use the stored data to get the approximate data matrix
E1=np.dot(Y,HatV.transpose())
cmat=np.outer(np.ones([1,1000]), c)
Daproxx=E1+cmat
print(Daproxx[0,:])
print('')
print(Data[0,:])
```

B SVD Implementation for Image Compression

Here, we show the implementation of the SVD algorithm for image compression using Python. We start with an image called "Image.jpg" which has to be stored in the same folder as the Python file with the code. At the end, the compressed image, is saved as "compressedImage.png".

```
#Image Compression Using SVD

import numpy as np
import scipy.misc
```

```
from numpy import linalg
from PIL import Image
image = np.array(Image.open('Image.jpg'))   #Import Image

image = image / 255   #Normalize the intensity values in each pixel
row, col, _ = image.shape
print "The number of pixels of this image is: ", row, "x", col #Number of pixels of original image
image_bytes = image.nbytes  #Calculate original space occupied
print "The space (in bytes) needed to store this image is", image_bytes

red = image[:, :, 0]
green = image[:, :, 1]
blue = image[:, :, 2]
# Full Singular Value Decomposition
red_U, red_d, red_V = np.linalg.svd(red, full_matrices=True)
green_U, green_d, green_V = np.linalg.svd(green, full_matrices=True)
blue_U, blue_d, blue_V = np.linalg.svd(blue, full_matrices=True)


k=50 #Store only k rows
red_U_k = red_U[:, 0:k]
red_V_k = red_V[0:k, :]
green_U_k = green_U[:, 0:k]
green_V_k = green_V[0:k, :]
blue_U_k = blue_U[:, 0:k]
blue_V_k = blue_U[0:k, :]

red_d_k = red_d[0:k]
green_d_k = green_d[0:k]
blue_d_k = blue_d[0:k]

#Calculate data necessary to recover original approximation

red_c=np.mean(red, axis=0)
np.outer(np.ones([1,col]),red_c)
red_E=red-red_c
red_V_t= red_V.transpose()
red_HatV= red_V_t[:,0:k]
red_Y=np.dot(red_E,red_HatV)

green_c=np.mean(green, axis=0)
np.outer(np.ones([1,col]),green_c)
green_E=green-green_c
green_V_t= green_V.transpose()
green_HatV= green_V_t[:,0:k]
green_Y=np.dot(green_E,green_HatV)

blue_c=np.mean(blue, axis=0)
np.outer(np.ones([1,col]),blue_c)
blue_E=blue-blue_c
blue_V_t= blue_V.transpose()
blue_HatV= blue_V_t[:,0:k]
blue_Y=np.dot(blue_E,blue_HatV)


compressed_bytes = sum([matrix.nbytes for matrix in
                        [red_Y, red_HatV, red_c, green_Y, green_HatV, green_c, blue_Y, blue_HatV, blue_c]])


storage_ratio = compressed_bytes/image_bytes

#Recover approximation of original image

red_E1=np.dot(red_Y,red_HatV.transpose())
red_cmat=np.outer(np.ones([1,row]),red_c)
red_aprox=red_E1+red_cmat

green_E1=np.dot(green_Y,green_HatV.transpose())
green_cmat=np.outer(np.ones([1,row]),green_c)
green_aprox=green_E1+green_cmat

blue_E1=np.dot(blue_Y,blue_HatV.transpose())
blue_cmat=np.outer(np.ones([1,row]),blue_c)
blue_aprox=blue_E1+blue_cmat

#Reconstruct the three dimensional array
image_reconstructed = np.zeros((row, col, 3))

image_reconstructed[:, :, 0] = red_aprox
image_reconstructed[:, :, 1] = green_aprox
image_reconstructed[:, :, 2] = blue_aprox


#image_reconstructed[image_reconstructed < 0] = 0  #Correct the pixels where intensity value is outside [0,1]
#image_reconstructed[image_reconstructed > 1] = 1

scipy.misc.imsave('compressedImage.jpg', image_reconstructed)
#fig = plt.figure()
#imgplot = plt.imshow(image_reconstructed)
#fig.savefig('compressedImage.png')
#Forbenius norm of the difference
error1=linalg.norm(red-red_aprox)
error2=linalg.norm(green-green_aprox)
error3=linalg.norm(blue-blue_aprox)
error=(error1+error2+error3)/3
```

```
frob1=linalg.norm(red)
frob2=linalg.norm(green)
frob3=linalg.norm(blue)
frob=(frob1+frob2+frob3)/3

relerror=error/frob
print k
print compressed_bytes
print storage_ratio
print error
print relerror

print k
print compressed_bytes
print storage_ratio
print error
```

C Haar Implementation for Image Compression

Here, we show the implementation of the Haar Transform algorithm for image compression using Python. We start with an image called "`Image.jpg`" which has to be stored in the same folder as the Python file with the code. At the end, the compressed image, is saved as "`compressedImage.png`".

```
#Image Compression Using Haar

import numpy as np
import math
import scipy.misc
from PIL import Image
from numpy import linalg

#Import and separate RGB components
image = np.array(Image.open('Graphics/Javeriana.jpg'))   #Import Image
image = image / 255.0
red = image[:, :, 0]
green = image[:, :, 1]
blue = image[:, :, 2]

#Convert matrices into one dimension that is 2^k
r=max(red.shape)
s=int(pow(2,math.ceil(math.log(r,2))))
r2 = np.zeros((s, s))
r2[:red.shape[0], :red.shape[1]] = red

g2 = np.zeros((s, s))
g2[:green.shape[0], :green.shape[1]] = green

b2 = np.zeros((s, s))
b2[:blue.shape[0], :blue.shape[1]] = blue


#Haar Matrix calculation (Dim n)
N = s
p = np.array([0, 0])
q = np.array([0, 1])
n = int(np.log2(N))

for i in np.arange(1, n):
    p = np.concatenate((p, i * np.ones(2**i)))
    t = np.arange(1, 2**i + 1)
    q = np.concatenate((q, t))

Hr = np.zeros([N, N])
Hr[0,:] = 1;

for i in np.arange(1, N):
    P = p[i]; # probably is actually (0, i)
    Q = q[i]; # probably is actually (0, i)
    for j in np.arange(N * (Q - 1) / (2**P), N * (( Q - 0.5) / (2**P)), dtype = int):
        Hr[i, j] = 2**(P/2)
    for j in np.arange(N * (( Q - 0.5) / 2**P), (N * (Q / 2**P)), dtype = int):
        Hr[i, j] = -(2**(P/2))
Hr = Hr * (1/ np.sqrt(N))

#Calculating Haar transform
r2T=np.matmul(np.matmul(Hr,r2),np.transpose(Hr))
g2T=np.matmul(np.matmul(Hr,g2),np.transpose(Hr))
b2T=np.matmul(np.matmul(Hr,b2),np.transpose(Hr))

#Quantization
k = 2
```

```
rquantT= np.asarray(r2T)
low= np.absolute(rquantT) < k # Where values are low
rquantT[low] = 0
nonzeror=np.sum(1 - low.astype(int))

vecg2T=np.absolute(np.reshape(g2T,s*s))
mig=np.min(vecg2T[np.nonzero(vecg2T)])
mag=np.max(vecg2T[np.nonzero(vecg2T)])

gquantT= np.asarray(g2T)
low= np.absolute(gquantT) < k # Where values are low
gquantT[low] = 0
nonzerog=np.sum(1 - low.astype(int))

vecb2T=np.absolute(np.reshape(b2T,s*s))
mib=np.min(vecb2T[np.nonzero(vecb2T)])
mab=np.max(vecb2T[np.nonzero(vecb2T)])

bquantT= np.asarray(b2T)
low= np.absolute(bquantT) < k # Where values are low
bquantT[low] = 0
nonzerob=np.sum(1 - low.astype(int))


#Calculating Haar Inverse transform and Reconstructing the Image
rquant= np.matmul(np.matmul(np.transpose(Hr),rquantT),Hr)
gquant=np.matmul(np.matmul(np.transpose(Hr),gquantT),Hr)
bquant=np.matmul(np.matmul(np.transpose(Hr),bquantT),Hr)

image_reconstructed = np.zeros((red.shape[0], red.shape[1], 3))

image_reconstructed[:, :, 0] = rquant[:red.shape[0], :red.shape[1]]
image_reconstructed[:, :, 1] = gquant[:green.shape[0], :green.shape[1]]
image_reconstructed[:, :, 2] = bquant[:blue.shape[0], :blue.shape[1]]

scipy.misc.imsave('ImageCompressed.jpg', image_reconstructed * 255.0)

#Error Calculation
error1=linalg.norm(red-rquant[:red.shape[0], :red.shape[1]])
error2=linalg.norm(green-gquant[:green.shape[0], :green.shape[1]])
error3=linalg.norm(blue-bquant[:blue.shape[0], :blue.shape[1]])
error=(error1+error2+error3)/3

frob1=linalg.norm(red)
frob2=linalg.norm(green)
frob3=linalg.norm(blue)
frob=(frob1+frob2+frob3)/3

relerror=error/frob

#Sparseness ratio

spr=float(nonzeror)/(low.shape[0]*low.shape[1])
spg=float(nonzerog)/(low.shape[0]*low.shape[1])
spb=float(nonzerob)/(low.shape[0]*low.shape[1])


spRatio=(spr+spg+spb)/3

print k
print spRatio
print relerror
```

Bibliography

[1] M. Brain. How bits and bytes work, Apr 2000.

[2] X. Chen. Lecture notes in numerical linear algebra, August 2017.

[3] C. K. Chui and Q. Jiang. *Applied Mathematics*. Springer, 2013.

[4] I. Daubechies et al. Ten lectures on wavelets. In *CBMS-NSF regional conference series in applied mathematics*, volume 61, 1991.

[5] Dictionary.com. quantize, 2018.

[6] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. Mit media laboratory vision and modeling technical report - subband coding - chapter 4: Subband transforms. *Kluwer Academic Press*, 137, 2003.

[7] A. Greenbaum and T. P. Chartier. *Numerical methods: design, analysis, and computer implementation of algorithms*. Princeton University Press, 2012.

[8] B. Husen. Haar wavelet image compression, 2010.

[9] P. U. Javeriana. Especializaciones pontificia universidad javeriana, N.D.

[10] P. D. Johnson Jr, G. A. Harris, and D. Hankerson. *Introduction to information theory and data compression*. CRC press, 2003.

[11] T. Kurics. The singular value decomposition in a nutshell, 2016.

[12] M. Mahoney. Data compression explained. *mattmahoney. net, updated May*, 7, 2012.

[13] C. A. Paixao and F. C. Coelho. Matrix compression methods. *PeerJ PrePrints*, 3:e849v1, Feb. 2015.

[14] K. Pawar. Code for generating haar matrix - file exchange - matlab central, Jan 2014.

[15] T. Roughgarden and G. Valieant. Lecture notes in the modern algorithmic toolbox - lecture 9: The singular value decomposition (svd) and low-rank matrix approximations, May 2017.

[16] K. B. S. Allen Broughton. *Discrete Fourier Analysis and Wavelets: Applications to Signal and Image Processing*. Wiley, 2nd edition, 2018.

[17] D. Salomon. *Data compression: the complete reference*. Springer Science and Business Media, 2004.

[18] D. Salomon. *A concise introduction to data compression*. Springer Science & Business Media, 2007.

[19] E. P. Simoncelli and E. Adelson. Fourier transform. *Hypermedia Image Processing Reference*, 1990.

[20] L. N. Trefethen and D. Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[21] S. Wolfram. Some historical notes, 2002.