Title: Differential entropy of digital pictures

Author: Laia Grifell Guàrdia

Advisors: Arjen Lenstra
           Maria Paz Morillo Bosch

Department: Matematica Aplicada IV

Differential entropy of digital pictures

Laia Grifell Guàrdia

Advisor:
Arjen Lenstra
Maria Paz Morillo Bosch

Nom del Departament
To all those who have helped me, in any possible way during the last four years. Specially to my parents, my grandmother and my uncle Jordi, for going out of their way to do so.
Abstract

Keywords: image, JPEG, compression, hash function

In this project we analyse the viability of the JPEG compression as a hash function in order to use that to improve smartphone security. In order to do that we will take an image and create a second one copying the first but applying random changes of a controlled and previously decided order. We then apply the JPEG compression to both images for all possible qualities available and compare the differences.
1. Introduction

Nowadays smartphones are used all the time, for all sorts of things. Amongst which are accessing our bank accounts, paying bills etc. information that we would rather keep to ourselves. However people do not realize that a smartphone is not as cryptographically secure as one would hope.

Professor Lenstra had the idea of obtaining a cryptographic key from photographs obtained with the smartphone and then analyse how to use that in the various aspects of cryptography (confidentiality, data integrity, authentication etc). This would mean an increase to the length of the key compared to the key derivation methods used currently while still keeping the key random (and even improving the randomness in some situations). When saying that the key remains random we are actually saying that the distribution of possible keys for the available set (in this situation the pixels of a photo) is undistinguishable from a uniform distribution.

However, simply taking one picture and using it as the key wouldn’t be secure as most people’s settings include uploading the photos to the cloud which makes the access to this photograph easier (and therefore the key). This situation can be solved by taking a large number of pictures and then randomly choose a single one for use.

All the photographs in smartphones are obtained in a JPEG compressed format. This compression format allows us to decide a parameter $q$ that refers to the quality of the picture and will be described in the project. What this project intends to test is that for any two given pictures (in JPEG compression and for any quality $q$) the difference is large enough, even if the pictures are taken at almost the same time from an exact position. In order to see that, we will obtain the photos in a RAW format, and to those photos we will apply random changes obtaining a differential difference between the original picture and the modified version. Once we have these two pictures, we will obtain the JPEG compression of both and compare them, obtaining a percentage difference, and we will do this for every quality $q$ available. What we need to see is that the differences obtained are large enough so we can consider that the resulting pictures are different.

Therefore the aim of the project is to prove that there exists a large enough quality $q$ so that the JPEG compression is a cryptographic hash function. Taking into account that a hash function is such a function that maps digital data of arbitrary size to digital data of a fixed size (we refer to the result as hash) and the ideal cryptographic hash function complies with the following properties:

- It is easy to compute the hash value for a given message.
- It is infeasible to generate a message from its hash.
- It is infeasible to change a message without modifying the hash.
- It is infeasible to find two messages with the same hash (There is no collision).

Then, before we start we already have the first two properties as obtaining the JPEG compression of a given photograph is easy while obtaining the RAW image from an already compressed one is impossible (The compression is lossy, some information is lost that we cannot retrieve). We will need to analyse if the JPEG compression follows the two last properties but we can see that both can be simplified to one. Having that a message can be modified without changing the hash
is like saying that there are two different messages with the same hash. Therefore the goal of the project will be to study whether the JPEG compression has collisions or not while hoping that there are not, because if there are the photographs will not be usable as keys. While we study that we will also need to study the effect of the quality $q$ and see whether the differences between images increase or decrease for an increase of $q$.

In image [1] we can see the same image for four different quality parameters. However we cannot appreciate the original tiff picture as LaTex does not support it. As we can see, the quality of the images is very different. Now we need to see whether the almost identical photos will be more similar when compressed for a low $q$ or for a higher one.

The hypothesis is that as the $q$ increases, so will the percentage difference and that is because the more information we lose, the more likely the random changes are to be ignored as irrelevant information.

![Compressed photos with different qualities](image1.png)

(a) Compressed photo with a quality of $q = 6$

(b) Compressed photo with a quality of $q = 25$

(c) Compressed photo with a quality of $q = 50$

(d) Compressed photo with a quality of $q = 94$

It is also worth mentioning that the idea is to use the uncompressed pixels as a key rather than using the compressed data because the second would not be indistinguishable from uniform because it contains many markers that can be known beforehand and are always followed by a sequence of bits that, in some situations, have to follow certain rules assuring that the distribution is easily differentiated from a uniform one.

Finally, all the result tables referred to in this project can be found in [https://drive.google.com/folderview?id=0B4C54T5S9HzjflRzUXhDrMzE2LV8zYnBkM3daZE1TmMxTzI1NhWmhMV1FejNpTGJkMjU5ZTA&usp=sharing](https://drive.google.com/folderview?id=0B4C54T5S9HzjflRzUXhDrMzE2LV8zYnBkM3daZE1TmMxTzI1NhWmhMV1FejNpTGJkMjU5ZTA&usp=sharing)
Chapter 1
Required specifications

In this section we define the JPEG and TIFF format and compression specifications as we require them for the project.

1. The JPEG compression

JPEG stands for Joint Photographic Experts Group and refers to a lossy compression method for continuous-tone still images and, informally, to the committee that defined this compression method. In this document when we refer to JPEG we will refer to the file format "jpg" and not to any other possible formats defined also by the JPEG committee (which can also be referred as JPEG).

As JPEG is a compression method and not a file format we have to use other file formats for interchange of JPEG-compressed images such as JFIF (usually used to exchange images through the world wide web) or Exif (commonly used by digital cameras) although they are not usually distinguished. A lossy compression method (such as JPEG) is based on the discrete cosine transform (DCT) which allows substantial compression (reduces the information required to generate the image) but still produces an image with high fidelity to the encoder’s source image. JPEG is the simplest DCT-based coding process, the baseline sequential process although the extended JPEG format can also be defined. However we will only consider the basic JPEG format.

The definition of the JPEG compression method requires to define the encoder, the decoder and the interchange format (or the compressed data format).

1.1. The encoder/decoder. The encoder takes a digital source image data and table specifications as input and generates a compressed image data as output through a specified set of procedures. As we have a DCT based encoding process we apply a FDCT (forward discrete cosine transform), a Quantizer and an Entropy encoder (both requiring the table specifications). The precision for the baseline process is 8-bit (other DCT-based modes might have 12-bit precision).

In Figure 1 we can see a scheme for a single-component image encoding process.
1. REQUIRED SPECIFICATIONS

Fig. 1. Single-component image DCT-based encoder

(grayscale image). For a multi-component image (such as JPEG) all processes specified operate on each image component independently, therefore the simplification is valid for defining purposes.

Along the encoding process we need to note that due to the use of the FDCT and IDCT we will have terms that cannot be represented with perfect accuracy. The FDCT takes an 8x8 block from the source image data and transform it into a set of 64 values (known as DCT coefficients), one of them is referred to as the DC coefficient and all the other 63 are known as AC coefficients. As we are considering the sequential DCT-based mode the 8x8 blocks are typically input block by block from left to right, and block-row by block-row from top to bottom. Then for each block we can apply the scheme from Figure 1 and directly save the output as part of the compressed image data (minimizing coefficient storage requirements). In Figure 2 we can see how the sequential DCT-based mode works.

Fig. 2. Sequential DCT-based mode

In our baseline sequential process we shall take into account that before computing the FDCT we need to apply a level shift by subtracting $2^{p-1} = 128$ (where $p$ is the precision parameter) and then apply the FDCT:

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{x,y} \cos \left( \frac{(2x+1)u\pi}{16} \right) \cos \left( \frac{(2y+1)v\pi}{16} \right)$$

where $C_u, C_v = \begin{cases} \frac{1}{\sqrt{2}} & u,v = 0 \\ 1 & \text{otherwise} \end{cases}$

The Quantizer The 64 values obtain after the FDCT are quantized using one of
1. THE JPEG COMPRESSION

64 corresponding values from a quantization table (obtained from the table specifications, this define the bits used to define each coefficient), these values are defined by applications in order to customize and optimize picture quality for the specific application. Then the quantized coefficients are prepared for entropy encoding by using the previous DC coefficient to predict the current one, having the difference encoded and organizing the 63 AC coefficients into a one-dimensional zig-zag sequence (without applying to them any differential encoding) as seen in Figure 3.

For each coefficient $S_{vu}$ we quantize using the parameter $Q_{vu}$ from the quantization table specified by $T_{qi}$ (frame parameter) as follows:

$$S_{qvu} = \text{round} \left( \frac{S_{vu}}{Q_{vu}} \right)$$

The Entropy Encoder

We can use one of two entropy coding procedures to compress the data further: the Huffman encoding (requiring the Huffman table specifications) or the Arithmetic encoding (where you can specify a conditioning table or use the default one). In the basic JPEG encoding we use the Huffman encoding with a maximum of two DC and two AC Huffman tables within one scan. As we are talking about a JPEG image (multiple-component) we have two options to encode these multiple-components. The non-interleaved method and the interleaved method. Let’s assume we have three components A, B and C. The non-interleaved method compresses first all the component A, then the component B and finally the component C while the interleaved method compresses a data unit (an 8x8 block) from A, one from B and one from C to come back to compress a data unit from A (or other sequences can be applied to interleave the components). From this, we can define the minimum coded unit (MCU), which is a single data unit for the non-interleaved method and the first round for the interleaved method.

The decoder takes a compressed image data and table specifications as input and outputs a digital reconstructed image data, again through a specified set of procedures. It basically inverses the process of encoding, from the compressed data we obtain the table specifications which we use for the entropy decoder and the dequantizer. Finally after dequantization the DCT coefficients are transformed to an 8x8 block of samples by the IDCT (inverse DCT). As we can see in Figure 4.

---

**Fig. 3.** Preparation of quantized coefficients for entropy encoding
We remove the normalization from the quantization process (although depending on the rounding used, the dequantized coefficient may be outside the foreseeable range):

\[ R_{vu} = S_{vu} \times Q_{vu} \]

We apply the IDCT:

\[
s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)
\]

with \( C_u, C_v \) defined as for the FDCT.

We add 128 to the result in order to undo the level shift obtaining the original information.

**Fig. 4.** Single-component image DCT-based decoder

### 1.2. The JFIF.

JFIF stands for JPEG File Interchange Format and is the most common file format for JPEG compressed images and it includes all table specifications used in the encoding (decoding) process and all the image data (compressed). However this is not the only option available to exchange compressed data. We can also have the abbreviated format for compressed image data (which differs from the JFIF as it may not include all the specification tables required) and the abbreviated format for table-specification data (containing only the table-specification, which might be useful if an application requires to reconstruct more than one image with the same tables). They follow the same requirement as the JFIF for the information available.

The goal of the JFIF is to be able to exchange images between applications environments (standards for data representation, communication or storage).

The data is described by a uniform structure and a set of parameters, special two-byte codes referred to as markers, identifying the various parts. This markers can be followed by particular sequences of parameters, called marker segments, which may give information such as length of the part, the components,... This marker segments are specific to each marker and will always give the same information.

For all methods the compressed data only contains the data for one image and, as
we have a baseline sequential DCT-based method, it will contain only one frame 
that will include one or more scans containing the complete encoding of one or 
more image components.

Going back to the previously seen example, the multi-component case with com-
ponents A,B and C, we will have three scans for the non-interleaved case one scan 
if all three components are interleaved together or two scans if we interleave two 
components and leave the third non-interleaved. To know about the interleaving 
system used we define the MCU (minimum coded unit). For a non-interleaved im-
age data the MCU is defined as one data unit. However, in an interleaved situation, 
the MCU contains at least one data unit from each component (it gives all the data 
units required to give the first round of interleaving pattern).

On our situation the markers are the following:

- **SOF**\textsubscript{0}: The marker is X'FFC0' and it identifies the Start of Frame while 
informing that the method is a baseline DCT.
- **DHT**: The marker is X'FFC4' and it identifies the start of the Huffman 
table(s).
- **SOI**: The marker is X'FFD8' and it identifies the Start of Image.
- **EOI**: The marker is X'FFD9' and it identifies the End of Image.
- **SOS**: The marker is X'FFDA' and it identifies the Start of Frame.
- **DQT**: The marker is X'FFDB' and it identifies the quantization frame(s).
- **DNL**: The marker is X'FFDC' and it identifies the number of lines.
- **DRI**: The marker is X'FFDD' and it identifies the restart interval.
- **DHP**: The marker is X'FFDE' and it identifies hierarchical progression.
- **EXP**: The marker is X'FFDF' and it expands reference component(s).
- **APP**\textsubscript{n}: The markers are X'FFE0' to X'FFEF' and they are reserved for ap-
lication segments.
- **JPEG**\textsubscript{n}: The markers are X'FFF0' to X'FFFD' and they are reserved for 
JPEG extensions.
- **COM**: The marker is X'FFDB' and it identifies a comment section.
- **RST**\textsubscript{m}*: The markers are X'FFD0' through X'FFD7' that refer to restart 
modulo 8.

The order of this markers is: the image is always start with **SOI**, following with 
tables (**DQT** and **DHT**) and misc. (**DRI**, **COM**, **APP**\textsubscript{n}), then the frame header 
(**SOF**), then the scans (also starting with possible tables and misc. and then adding 
the scan header (**SOS**)) and finally, in-between entropy encoded segments the **RST**\textsubscript{m} 
markers) and in-between scans the **DNL** segment.

### 1.3. The quality parameter.

Finally, when compressing an image to JPEG we 
can specify a quality parameter q. This parameter is not universally defined and 
it depends on the application used but the idea is the same for all applications, 
it refers to the length of the elements of the quantization table. However, as we 
include the quantization tables in the data, we can exchange the resulting JFIF 
between applications without worries (it’s only used in the encoding and not the 
decoding).
The problem is that this parameter is not defined, not even referred to, in the specification done by the JPEG committee which leads to several scaling options. The most common option is a scale from 1 to 100, used by IJG or Paint Shop Pro, but Apple software typically uses a 1 to 4 scale while Adobe Photoshop mostly used a "high", "medium" and "low" scale.

Also, amongst the 1 to 100 scale users those scales are not comparable, as some are 1 the lowest and 100 the highest quality possible while others are the opposite and more importantly, even if they are defined in the same direction, as they do not refer to percentages they are scaled differently. Meaning that with two different programs, using both 1 to 100 scales from lower to higher, a quality of 70 means a different quality for one and the other.

In our project the application used (PIL Library) has a 0 to 100 scale but we will only use the range [5, 95) as the limits are mathematically defined but not useful in practice and may be unstable.

2. The RAW file format

The raw format is a format that cannot be used per se as an image (not viewable) but contains all the information to create one. For such type of file each of the main manufacturers has developed it's own format (i.e. Sony has .ARW and Canon has .CRW and Nikon has .NEF amongst others) but the .TIFF format file, belonging to Adobe Systems Incorporated, is free of distribution which makes it the most extended one.

However the .TIFF file format differs from all other raw formats because it can be viewed as an image and because we can access the information regarding the encoding as it does not have commercial purposes, the copyright specifies that as long as there is no direct commercial advantage you can distribute the specifications freely.

In this project we will work with the .TIFF format file.

3. The TIFF file format

As in the JPEG situation we have the Baseline TIFF and possible expansions to that baseline. We will only treat with the baseline TIFF.

3.1. TIFF Structure. The maximum length for a TIFF file is $2^{32}$ bytes and they all must include an Image File Header (IFH), the Image File Directory (IFD) and image data. There might be more than one IFD in a TIFF file but that should not be the case in baseline TIFF (as baseline readers might not read beyond the first IFD).

**Image File Header** It’s an 8byte long image file header always at the beginning of the file containing the byte order used within the file, an arbitrary number and the offset of the first IFD in the following order.

The byte order can be "II" (4949 in hexadecimal) refers to the LITTLE-ENDIAN byte...
order (byte order is always from the least significant byte to the most significant one) or "MM" (4D4D in hexadecimal) refers to the big-endian byte order (from the most significant to the least significant byte), in both cases it’s irrelevant if we have 16-bit or 32-bit integers.

The arbitrary number is carefully chosen as 42 further identifying the file as TIFF, it’s order depends on the value of the byte order.

Finally the offset of the first IFD where the directory may be at any location after the header but must begin on a word boundary. The first byte of the file has an offset of 0, every other location is given respect to this first byte.

**Image File Directory** We have a 2-byte count of the number of entries, followed by that number of 12-byte entries and ending with a 4-byte offset of the next IFD. For the 4-byte offset it’s very important to remember add 0000 if we are at the last IFD (even if it’s also the first!).

The entries must be sorted in ascending order by Tag. Then, for each entry we have a Tag (2-bytes) to identify the field, followed by the Type (2-bytes again), then a 4-byte count (number of values of the indicated Type) and finally a 4-byte offset known as the Value offset.

As the previous offset, the Value is also expected to begin on a word boundary (making the offset an even number). In some situations (which can be determined by the Type and Count of the field) the Value fits into 4-bytes. In this situations, the Value Offset will contain the Value instead of the pointer. We need to take into account that if Value is shorter than 4 bytes we have it, left-justified, within the 4-byte Value Offset (even if we don’t need the space, the length is still 4 bytes).

About the Count we need to bear in mind that it’s not the total number of bytes but the number of values. For example, the 16-bit value SHORT has Count 1.

Finally we have the following Types:

- **Byte**: 8-bit unsigned integer
- **Ascii**: 8-bit byte containing a 7-bit ascii code. The last byte has to be nul. We have that in this situation the Count does include the final Nul byte but does not include the pad byte (if needed). Also in a Ascii field we can have more than one string (separated by nul byte) and then the Count will be the number of bytes in all strings plus their terminating nul bytes. It’s important to notice that only one nul is allowed between strings.
- **Short**: 16-bit unsigned integer
- **Long**: 32-bit unsigned integer
- **Rational**: two Long where the first represents the numerator and the second the denominator of a fraction.
- **Sbyte**: 8-bit signed integer.
- **Undefined**: An 8-bit byte that may contain anything (depends on the field definition).
- **Sshort**: 16-bit signed integer.
- **Slong**: 32-bit signed integer.
- **Srational**: The same system as with Rational but with two Slong.
- **Float**: Single precision IEEE format.
- **Double**: Double precision IEEE format.

Each field has an associated Count and therefore is always treated like an array, even if it only has one single value.
Now for any TIFF file we can have four types of images: Bilevel, Grayscale Palette-color and RGB Full Color (also called Class R). In this project we will deal with the last.

### 3.2. RGB Full Color Images

For this situation we have that each pixel is made up of three components (red, green and blue) and there’s no ColorMap. We require the following fields:

<table>
<thead>
<tr>
<th>TagName</th>
<th>Decimal</th>
<th>Hex</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageWidth</td>
<td>256</td>
<td>100</td>
<td>SHORT/LONG</td>
<td></td>
</tr>
<tr>
<td>ImageLength</td>
<td>257</td>
<td>101</td>
<td>SHORT/LONG</td>
<td></td>
</tr>
<tr>
<td>BitsPerSample</td>
<td>258</td>
<td>102</td>
<td>SHORT</td>
<td>8,8,8</td>
</tr>
<tr>
<td>Compression</td>
<td>259</td>
<td>103</td>
<td>SHORT</td>
<td>1 or 327773</td>
</tr>
<tr>
<td>PhotometricInterpretation</td>
<td>262</td>
<td>106</td>
<td>SHORT</td>
<td>2</td>
</tr>
<tr>
<td>StripOffset</td>
<td>273</td>
<td>111</td>
<td>SHORT/LONG</td>
<td></td>
</tr>
<tr>
<td>SamplesPerPixel</td>
<td>277</td>
<td>115</td>
<td>SHORT</td>
<td>3 or more</td>
</tr>
<tr>
<td>RowsPerStrip</td>
<td>278</td>
<td>116</td>
<td>SHORT/LONG</td>
<td></td>
</tr>
<tr>
<td>StripByteCount</td>
<td>279</td>
<td>117</td>
<td>LONG/SHORT</td>
<td></td>
</tr>
<tr>
<td>XResolution</td>
<td>282</td>
<td>11A</td>
<td>RATIONAL</td>
<td></td>
</tr>
<tr>
<td>YResolution</td>
<td>283</td>
<td>11B</td>
<td>RATIONAL</td>
<td></td>
</tr>
<tr>
<td>ResolutionUnit</td>
<td>296</td>
<td>128</td>
<td>SHORT</td>
<td>1,2 or 3</td>
</tr>
</tbody>
</table>

Where the BitsPerSample apply only to the main image data and if we have ExtraSamples we also require the appropriate BitsPerSample values for those.

Finally we can also have some extra fields, some examples are:

- **Artist:** Tag 315 (13B in hex) and ascii Type. Person who created the image, for older TIFF files it can store the Copyright information.
- **Copyright:** Tag 33432 (8298 in hex)
- **DateTime:** Tag 306 (132 in hex) with ascii Type and N = 20 as it follows the format "YYYY:MM:DD HH:MM:SS" with the 24-hour clock.
- **ExtraSample:** Tag 338 (152 in hex) with Short Type and N = m which specifies that each pixel has m extra components that are usually used for non-color information as could be the opacity, and each of them can be either unspecified (0), associated alpha data (1) or unassociated alpha data (2).
- **HostComputer:** Tag 136 (13C in hex) and ascii Type referring to the Computer/OS in use when the image was created. It can also include Make, Model and Software.
Chapter 2
Methodology

1. Preliminaries

We start by taking photographs with a camera (a Sony α 390). We will take random pictures from any kind of situation (indoors, outdoors, moving or still environment etc.) but we will also make sure to take consecutive images from the same still objects (without moving the camera and with the smallest amount of time possible between them), we will refer to such collection of images as a group.

Once all the required images have been obtained, which for us means 151, we pass them to the computer obtaining .ARW files. As we need .TIFF files we convert them using the Image Data Converter Ver.4 (the official converting programme for the Sony brand).

In order to be able to run the code described below we need the 151 TIFF files recently obtained in a single folder (and that folder containing nothing but those files).

On the other hand we also want to be certain that the order of random differences we impose is lower (or of the same order) than images taken in real life. In order to do that, we will apply the code 4.3 to the images obtained sorted by groups.

2. Image comparison

To treat the images we used a python library called PIL (Python Imaging Library), although we will use only the Image module, which allows us to decompress the image into python readable objects and interact with those. Moreover, we will also need to install the Pillow library, which we will use as support for the PIL library, because otherwise the PIL library cannot deal with TIFF files. The other python libraries required are the os, random, xslwriter and time.

The decision to use the PIL Image module and not create a code also to decompress the image, comes from the fact that PIL is reliable and has been tested by many other users that can testify that it works properly while this would not be true from a homemade code (even if maybe it could have been a little bit faster as it would have been more specific to our goals) and the aim of the project is to compare, and
be able to rely on the comparison, a number of images.
For every image we do the following: we open the same image twice, we then
apply random changes to one of both versions, and save the exact percentage of
difference between both. Then for every q between 5 and 95 we compress both
images to JPEG with quality = q and compare and save the absolute differences
and the percentage. We finally record our data into an excel file.
There are two comments required here: we use a 5-95 scale (instead of the defined
0-100) because the limits of the interval are not stable nor recommended for use
in this scale. Also, the comparison is between pixels, which are given by a triplet
data (representing the red, green and blue values of that pixel), and we do not
take into account if the difference is from (3,4,5) to (3,3,5) or to (100,190,9), it all
counts as different.
Finally for each image we do the following:

(1) Open the image twice, as im1 and im2.
(2) Randomly modify im2 so that the differences between both images are of order
   p (which we can modify at the start of the code).
(3) Then for every q from 5 to 95:
   (a) Save both files as JPEG with quality=q.
   (b) Open both files again.
   (c) Compare both files.
(4) Save the obtained data.

To open the image we use the function available in PIL Image.open('filename')
which obtains an image object (which only contains the header information). Once
we have this object, if we apply the load() function, we obtain a pixel matrix for
each image that we can compare and modify, this is how we will, in the second step,
access the pixels. Therefore in our code we will open the same twice but only load
one of them (im2).
To do step (2) we will access the pixel by using the objects obtained in the pre-
vious step and simply doing im1[i,j] (where im1 is the loaded image). Then the
only question is how to randomly implement the desired changes. For that we will
use the random library already included in python, specifically the function ran-
dom.randrange(0,total) with the following condition: if random.randrange(0,total)
¡ percentage we apply a change. We have the total and percentage because ran-
drange returns an integer and we might want to have a 0.1% difference for example.
Then the change itself requires to create a triplet (random.randrange(0,255), ran-
dom.randrange(0,255), random.randrange(0,255)) which we will compare with the
original pixel and exchange (we compare because in the very improbable situation
where the random numbers obtained are the same as the numbers we originally
had, we do not want to count that one as different).
For step (a) we make use of the function save available at PIL. We need to specify
as follows: im1.save("Image\path\name + .jpg","JPEG", quality = q) this will
compress our image to the required quality.
For step (b) we open the just now saved images (we cannot overwrite them on
the original image as we need that one in every iteration!) using again the load() function.
For step (c) we could have used the function getdata() which gives us all the data as
a one-dimensional array but as we were dealing with 14.000.000 pixels we obtained
memory errors when applying the list() function required to obtain the array. Finally for step (4) we require the library xlslwriter which allows us to save the data on an excel file after creating a workbook and a worksheet (workbook = xlslwriter.Workbook(Name) and worksheet = workbook.add_worksheet()) by simply using worksheet.write(row, 0, data).
We can see the code in the annex, in code 4.1.

3. Comments

There are two situations that should be commented. The use of xlslwriter instead of csv and the use of the load function instead of the draw module.
The choice to use the xlslwriter was purely about comfort, as this is the excel python library and it already saves the data in an excel file. Also the fact that it doesn’t require to open the file, just create it, and that if the situation requires it we can not only choose the raw but also the column where we place the information.
On the other hand, the choice of using the load function was made after comparing how much it took for both methods, load and draw, to apply the random changes for a total of 50 images. The results are not really relevant because many uncontrollable things can affect the time (such as internal computer processes) but the same computer (without any user action) was used to obtain the times. We obtained the following:

<table>
<thead>
<tr>
<th></th>
<th>Load function</th>
<th>Draw module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>17,57414</td>
<td>17,68164</td>
</tr>
<tr>
<td>Std dev</td>
<td>0,8482</td>
<td>1,964224</td>
</tr>
<tr>
<td>Var</td>
<td>0,719444</td>
<td>3,858177</td>
</tr>
<tr>
<td>Max</td>
<td>19,81</td>
<td>25,064</td>
</tr>
<tr>
<td>Min</td>
<td>15,74</td>
<td>15,601</td>
</tr>
</tbody>
</table>

This results are not reliable enough to say that the load function is a better method to use than the draw module but it shows that the draw module is not better than the load function. Therefore, as the load function requires less modules from PIL and it’s much simpler it was the chosen system.
Chapter 3
Results

It is interesting to note that while the .tiff file format has a size of over 40.000kb the JPEG compressions have much lower sizes, ranging from a little over 200kb for some images compressed with quality \( q = 5 \) to under 6.000 for images with a quality \( q = 94 \). Therefore the information contained in one or another picture will vary greatly.

We have been able to see the visual effects of this information decay in Figure 1 in Introduction.

1. Preliminary comparisons

We can see the full table of comparisons in the PDF "Comparacions", from that data we obtain the following Table 1:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>96.6182%</td>
<td></td>
</tr>
<tr>
<td>Std dev</td>
<td>9.8556</td>
<td></td>
</tr>
<tr>
<td>Var</td>
<td>97.1341</td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>0.001%</td>
<td></td>
</tr>
<tr>
<td>Min (2)</td>
<td>51.6545</td>
<td></td>
</tr>
</tbody>
</table>

Therefore we can see that there is one case where the images are very close and we will have to talk about it on its own, but for all the other images we have that two very similar images, once analysed by pixels are quite different as shown in Figure 1. We need to see for which photos, and also analyse why, the differences are 0.001% and 51.6545%. This happens for pictures DSC06675 and DSC06736 and DSC06802 and DSC06803 respectively and we can see them on Figure 2. For the first situation we have that DSC06675 and DSC06736 are both camera malfunctions resulting in black pictures while DSC06802 and DSC06803 have a large area that is completely white due to light. We analyse the pixels (of those four images and the images from the other groups) to see if we can find any difference between the four photos and the rest.

\(^1\)We define Min(2) as the minimum of the full table once the original minimum is excluded
3. RESULTS

Fig. 1. Classification of the differences between pictures

The code used can be seen in Annex, Code 4.7, and we obtain that DSC06675 and DSC06736 both have a 99.999% of pixels filled with color (0,0,0) (black) while DSC06802 and DSC06803 both have over 50% of pixels filled with color (255,255,255) (white). However, if we analyse the rest of the pictures (belonging to a group) the next percentage of repeated pixel color drops down to under 15%. Therefore we can suspect that having a pixel highly repeated will affect the results for this project.

For DSC06801 and DSC06802 we can see in Figure 3 that the differences between the compressed versions remain very close to 50%. Then if we consider the JPEG compression as a hash function we can be certain that a compression of one cannot be used as a collision for the other. The same will be true for all other groups of photos (except DSC06675 and DSC06736) as the original tiff images are already over 80% different (in most cases over 95%) so comparing these two "similar" images and comparing two random images would be the same.

The problem appears when we see the results for DSC06675 and DSC06736, observed in Figure 4. In this situation we have collision, for 71 qualities available (out of 90) it is an absolute zero and only for 2 the difference is slightly higher than the original tiff difference. This means that in order to say that the JPEG compression properly works as a hash function we need to make sure that we are not going to apply it to a camera malfunction (or any other photo that has over 90% of the picture filled by the same colour pixel.

\(^2\)The images are in JPEG format with quality 94 (highest possible) as \texttt{HyTeX} does not support .TIFF
**Fig. 2.** Groups that have the lowest differences

(a) G1: DSC06675 and DSC06736 (0.001% diff)

(b) G2: DSC06802 and DSC06803 (51.6545% diff)

**Fig. 3.** Differences for the compressed versions of DSC06801 and DSC06802
2. Results for a differential random difference

In Figure 4, Figure 9 and Figure 10 we can see the results obtained for our 151 figures with original random differential differences of order 0.5%, 0.1% and 0.01%. First we will talk about the outliers, which all have clear smaller differences. There is a total of 5 outliers, although it might seem as there is only 4 because two of them are really close (the lowest orange line is covering a blue line). The outliers are DSC06675, DSC06736, DSC06801, DSC06802 and DSC06803 and can be seen in Figure 5.

As we can see four of the five images where the ones we talked about in the previous section. For the fifth one (DSC06801), which was not mentioned before because it did not belong to any group, if we look at the pixels we can also see that it has 68.5% of white ((255,255,255)) pixels. This means that we could argue that having very homogeneous photos supposes a problem, not only towards having a bigger chance of two pictures being closer but also towards having closer compressions even if the original difference is the same. The reason for this might be that random changes can be interpreted as irrelevant and smoothed by the homogeneous surrounding pixels.

Now that we have commented on the outliers (and we have found a way to detect them and try to remove them if they are like DSC06675 and DSC06736) we focus on the rest. We can see in Figure 6 the average differences for the different differential random differences. As we can see, in average we will have that the JPEG compression is a good hash function, even from the beginning.

In Figure 7 we can see that the ratio between the difference between images when compressed and the original difference is quite large, in numbers we have that the
Fig. 5. The photos whose differential difference when compressed is clearly below the rest. 

minimum ratio is 3.96050 (for 0.5% difference), 2.64060 (for 0.1% difference) and 2.32592 (for a 0.01% difference) meaning that, even in the worst case scenario, the difference doubles. This means that for any image with enough color variation we can be certain that the JPEG compression is a good hash function. Now we need to define when is there enough color variation.

3Once the outliers we have commented are removed, and the minimum in every case, they might not be for the same picture
3. RESULTS

Fig. 6. Graphic showing the average differences when compressed for the different original differences

Fig. 7. Graphic showing the relation between the minimum difference and the original difference
Fig. 8. Graphic of the differences for all quality $q$, between a photo and a random modification with a difference of order 0,5\%
Fig. 9. Graphic of the differences for all quality $q$, between a photo and a random modification with a difference of order 0.1%
Fig. 10. Graphic of the differences for all quality $q$, between a photo and a random modification with a difference of order $0.01\%$.
Chapter 4
Conclusions and further work

From the results obtained we can see that once we remove the homogeneity of the picture the differences obtained after compressing with JPEG at least double the original differences which allows us to say that there is no collision. Then if we have a homogeneity of 50% we can say that the results point towards the difference being maintained by the compression but 3 photographs is not a large enough sample. However if we obtained a larger sample and the results obtained now still proved to be true, we would not have collisions either as the compressions would be as close as the originals.

Finally, for the camera malfunctions we have that if that situation is to arise, we will have collisions with high probability as any two camera malfunctions are likely to be a collision for most qualities $q$.

Before being able to use the photographs as keys we should find a way to remove highly homogeneous photographs. The most obvious way would be to go through all the pixels and obtain the most repeated pixel’s percentage, however it would be interesting to try to find a more efficient way as we have to keep in mind that this must be done by a smartphone.

Finally it would be interesting to further study the influence of pixel homogeneity in the results. In order to do that we should be able to apply the same code for a larger range of images while making sure that there are more homogeneous pictures and with different degrees of homogeneity. Generating ourselves the homogeneous pictures would also be an option but it would be hard to obtain real options, the best way to do that would be to take a picture already homogeneous and either add to that homogeneity or remove from it so that the overall homogeneity will still resemble one that we could obtain naturally.
References

Annex: The code

Listing 4.1. Comparing the image with its randomly modified version

```python
from PIL import Image
import os
import random
import xlsxwriter
import time

path = "C:\Users\Laia\Documents\Universitat\TFG"
image_path = path + "\Images4"
compre_path = path + "\Compressions"
change_path = path + "\Modified_Compressions"

percentage= 1; #Tan per mil
total = 10000; #El val maxim del randrange

#Making sure all necessary folders exist!
if not os.path.exists(path):
    os.makedirs(path)

if not os.path.exists(image_path):
    os.makedirs(image_path)
    print('This program will fail because it does not have the images where they are needed')

if not os.path.exists(compre_path):
    os.makedirs(compre_path)

if not os.path.exists(change_path):
    os.makedirs(change_path)

images = os.listdir(image_path)
```
# List all the images we want to convert

# Create the workbook where we will write the results
workbook = xlsxwriter.Workbook('Results_0_01.xlsx')
workbook2 = xlsxwriter.Workbook('Temps_normal.xlsx')
worksheet = workbook.add_worksheet()
worksheet2 = workbook2.add_worksheet()

# First line of worksheets - t:
nums = ['Name', 'Original total', 'Original differences', 'Original %']
for i in range(0, 100):
    nums.append('Compression level ' + str(i))
    nums.append('CL ' + str(i))
worksheet.write_row(0, 0, nums)

first = True  # To check that the needed folder exists only for the first image!
count = 1
leng = len(images)
begin_time = time.time()
for im in images:
    dir = {}
    # We take an image
    im1 = Image.open(image_path + '\' + im)
    im2 = Image.open(image_path + '\' + im)
    ima = im1
    ims = im2
    name = ''
    name_aux = im.split('.')[1]
    if len(name_aux) == 2:
        name = name_aux[0]
    else:
        for i in range(0, len(name_aux) - 2):
            name += name_aux[i]
    # We now modify the image slightly
    print 'We modify the image'
    mida = im1.size
    start_time = time.time()
    ims1 = ims.load()
    dir['original'] = 0
    for i in range(0, mida[0]):
        for j in range(0, mida[1]):
            if random.randrange(0, total) < percentage:
                aux = ims1[i, j]
                ims1[i, j] = (random.randrange(0, 255),
                              random.randrange(0, 255),
                              random.randrange(0, 255))
if aux != ims1[i, j]:
    dir["original"] += 1

print "It took us %s" %(time.time() - start_time)
print "And the percentage change is %f" 
%(dir["original"] * 100. / (mida[0] * mida[1]))
worksheet2.write_row(count, 0, [time.time() - start_time])
midaa = mida[0] * mida[1]
# Now we compress it for every possible option
llista = [im]
llista.append(mida[0] * mida[1])
llista.append(dir["original"]) 
llista.append(dir["original"] * 100. / (mida[0] * mida[1]))
print "We save both images"
for i in range (5, 95):
    if first:
        if not os.path.exists(compres_path + '\Quality_' + str(i)):
            os.makedirs(compres_path + '\Quality_' + str(i))
        if not os.path.exists(change_path + '\Quality_' + str(i)):
            os.makedirs(change_path + '\Quality_' + str(i))
    ima.save(compres_path + '\Quality_' + str(i) + '\' + name + '\
    .jpg', 'JPEG', quality = i)
    ims.save(change_path + '\Quality_' + str(i) + '\' + name + '\
    .jpg', 'JPEG', quality = i)
# Here we do a comparison between them
imat1 = Image.open(compres_path + '\Quality_' + str(i) +\
    '\' + name + '.jpg')
imat2 = Image.open(change_path + '\Quality_' + str(i) +\
    '\' + name + '.jpg')
imat1 = Image.open(compres_path + '\Quality_' +\
    str(i) + '\' + name + '.jpg').load()
imat2 = Image.open(change_path + '\Quality_' +\
    str(i) + '\' + name + '.jpg').load()
    dir[i] = 0
for j in range(0, mida[0]):
    for k in range(0, mida[1]):
        if imat1[j, k] != imat2[j, k]:
            dir[i] += 1

if i%5 == 0:
    print "For a level of compression %f we have a change of %f" 
    %(i, dir[i] * 100. / midaa)
llista.append(dir[i])
llista.append(dir[i] * 100. / midaa)
first = False
worksheet.write_row(count, 0, llista)

print 'We have done image %s of %s' % (str(count), str(leng))
count += 1
workbook.close()
workbook2.close()
print "The program took a total amount of %s seconds" % (time.time() - begin_time)
ANNEX: THE CODE

Listing 4.2. Comparing groups of images and their JPEG compressions

```
.. Author:: 'Laia G. Grifell'

from PIL import Image
from PIL import ImageDraw
import os
import random
import xlsxwriter
import time

path = "C:\Users\Laia\Documents\Universitat\TFG\Imatges"
compres_path = path + "\JPEG"

percentage = 1; #Tan per mil

#Making sure all necessary folders exist!
if not os.path.exists(path):
    os.makedirs(path)

if not os.path.exists(compres_path):
    os.makedirs(compres_path)

#Create the workbook where we will write the results
workbook = xlsxwriter.Workbook('Results_Compar.xlsx')
worksheet = workbook.add_worksheet()

#First line of worksheet:
ums = ["Name_1", "Name_2", "Original_total", "Original_differences"]
for i in range(0, 100):
    nums.append("Compression_level" + str(i))

for i in range(0, len(groups)):
    group = groups[i]
    image_path = path + "\\" + group
    photos = os.listdir(image_path)
    gpath = compres_path + "\\" + group

    if not os.path.exists(gpath):
        os.makedirs(gpath)

#We save all the images as jpg
mida1 = [0, 0]
```
print "We_save_the_jpeg_versions"
for i in range(0, len(photos)):
    imla = Image.open(image_path + '\ ' + photos[i])
    im1 = imla.load()
    name = photos[i].split('.')[0]
    for j in range(5, 95):
        cpath = gpath + '\ ' + str(j)
        if i == 0 and not os.path.exists(cpath):
            os.makedirs(cpath)
            imla.save(cpath + '\ ' + name + '.jpg', 'JPEG', quality=j)

    if i == 0:
        mida1 = imla.size
        del imla

    # We compare the images
for i in range(0, len(photos)):
    name1 = photos[i].split('.')[0]
for j in range(i+1, len(photos)):
    btime = time.time()
    im1 = Image.open(image_path +"\n"+photos[i]).load()
    im2 = Image.open(image_path +"\n"+photos[j]).load()
    name2 = photos[j].split('.')[0]
    line = [name1, name2]
    # We compare the originals
    dif = 0;
    for k in range(0, mida1[0]):
        for l in range(0, mida1[1]):
            if im1[k,l] != im2[k,l]:
                dif +=1
    mida = mida1[0]*mida1[1]
    line.append(mida)
    line.append(dif*100./mida)
    del im1, im2

for m in range(5, 95):
    cpath = gpath + "\n" + str(m)
    im1 = Image.open(cpath+"\n"+\name1 + '.jpg').load()
    im2 = Image.open(cpath+"\n"+\name2 + '.jpg').load()
    dif = 0;
    for k in range(0, mida1[0]):
        for l in range(0, mida1[1]):
            if im1[k,l] != im2[k,l]:
                dif +=1
    line.append(dif*100./mida)
del im1, im2
line.append(time.time() - btime)
count += 1
worksheet.write_row(count, 0, line)
print "We have done comparison %i" %count

workbook.close()
Listing 4.3. Comparing groups of images

..Author..= 'Laia Grifell'

```

from PIL import Image
import os
import xlsxwriter

path = "C:\Users\Laia\Documents\Universitat\TFG\Images4"

if not os.path.exists(path):
    os.makedirs(path)
    print "The program will fail because there are no photos to compare!"

grups = os.listdir(path)
min_dif = 1000;
total_dif = []

#Create the workbook where we will write the results
workbook = xlsxwriter.Workbook('Comparisions.xlsx')
worksheet = workbook.add_worksheet()
worksheet.write_row(0,0,[ 'Foto_1', 'Foto_2', 'Diferences'])

count = 0;
grouping = 0;
tgrouping = len(grups)
for group in grups:
    group_path = path + "\\" + group
    try:
        fotos = os.listdir(group_path)
        for i in range(0,len(fotos)):
            im1 = Image.open(group_path+'\\'+fotos[i])
            ima1 = im1.load()
            mida1 = ima1.size
            for j in range(i+1,len(fotos)):
                dif = 0
                im2 = Image.open(group_path+'\\'+fotos[j])
                ima2 = im2.load()
                mida2 = ima2.size
                if mida1[0] != mida2[0] or mida1[1] != mida2[1]:
                    print "We have a problem with sizes!"
                    dif = 1000
                else:
                    d = mida1[0] - mida2[0];
                    d = sqrt(d)
                    if d < min_dif:
                        min_dif = d
                        total_dif = [ i, j]
            if min_dif < total_dif[0]:
                worksheet.write_row(count,0,[fotos[i], fotos[j], min_dif])
                count += 1
```
for k in range(0, mida1[0]):
    for l in range(0, mida1[1]):
        if ima1[k, l] != ima2[k, l]:
            dif += 1;

        total_dif.append(dif)
        float(dif) / float(mida1[1] * mida1[0])
        if min_dif > dif:
            min_dif = dif

    count += 1
    worksheet.write_row(count, 0, \[ fotos[i], fotos[j], dif \])

except:
    print "We have something that is not a folder of images in the path! It's %s" % group
    grouping += 1
    print "We have done group %i of %i" % (grouping, tgrouping)
workbook.close()

Listing 4.4. The open function
def open(fp, mode="r"):
    ""
    Opens and identifies the given image file.

    This is a lazy operation; this function identifies the file, but the file remains open and the actual image data is not read from the file until you try to process the data (or call the :py:meth:`PIL.Image.Image.load` method). See :py:func:`PIL.Image.new`.

    :param mode: The mode. If given, this argument must be "r".
    :exception IOError: If the file cannot be found, or the image cannot be opened and identified.
    ""

if mode != "r":
    raise ValueError("bad_mode", % mode)

if isPath(fp):

filename = fp
fp = builtins.open(fp, "rb")
else:
    filename = 

try:
    fp.seek(0)
except (AttributeError, io.UnsupportedOperation):
    fp = io.BytesIO(fp.read())

prefix = fp.read(16)
preinit()

for i in ID:
    try:
        factory, accept = OPEN[i]
        if not accept or accept(prefix):
            fp.seek(0)
            im = factory(fp, filename)
            _decompression_bomb_check(im.size)
            return im
    except (SyntaxError, IndexError, TypeError, struct.error):
        # import traceback
        # traceback.print_exc()
        pass

if init():

    for i in ID:
        try:
            factory, accept = OPEN[i]
            if not accept or accept(prefix):
                fp.seek(0)
                im = factory(fp, filename)
                _decompression_bomb_check(im.size)
                return im
        except (SyntaxError, IndexError, TypeError, struct.error):
            # import traceback
            # traceback.print_exc()
            pass

raise IOError("cannot identify image file %r"
              % (filename if filename else fp))
Listing 4.5. The load function

```python
def load(self):
    
    """
    Allocates storage for the image and loads the pixel data.
    In normal cases, you don’t need to call this method, since the Image
class automatically loads an opened image when it is accessed for the first time. This method
will close the file associated with the image.
    """

    if self.im and self.palette and self.palette.dirty:
        # realize palette
        self.im.putpalette(*self.palette.getdata())
        self.palette.dirty = 0
        self.palette.mode = "RGB"
        self.palette.rawmode = None
        if "transparency" in self.info:
            if isinstance(self.info["transparency"], int):
                self.im.putpalettealpha(self.info["transparency"], 0)
            else:
                self.im.putpalettealphas(self.info["transparency"])
                self.palette.mode = "RGBA"

    if self.im:
        if HAS_CFFI and USE_CFFI_ACCESS:
            if self.pyaccess:
                return self.pyaccess
        from PIL import PyAccess
        self.pyaccess = PyAccess.new(self, self.readonly)
        if self.pyaccess:
            return self.pyaccess
        return self.im.pixel_access(self.readonly)
```

Listing 4.6. The save function

```python
def save(self, fp, format=None, **params):
    """
    Saves this image under the given filename. If no format is specified,
    the format to use is determined from the filename extension, if possible.

    Keyword options can be used to provide additional instructions to
    the writer. If a writer doesn’t recognise an option, it is
```
silently ignored. The available options are described in the
:doc:`image format documentation <../handbook/image-file-formats>` for each writer.

You can use a file object instead of a filename. In this case, you must always specify the format. The file object must
implement the `'seek'`, `'tell'`, and `'write'` methods, and be opened in binary mode.

:param fp: File name or file object.
:param format: Optional format override. If omitted, the
    format to use is determined from the filename extension.
    If a file object was used instead of a filename, this
    parameter should always be used.
:param options: Extra parameters to the image writer.
:returns: None
:exception KeyError: If the output format could not be determined
    from the file name. Use the format option to solve this.
:exception IOError: If the file could not be written.

The file
    may have been created, and may contain partial data.

""

if isPath(fp):
    filename = fp
else:
    if hasattr(fp, "name") and isPath(fp.name):
        filename = fp.name
    else:
        filename = ""

# may mutate self!
self.load()

self.encoderinfo = params
self.encoderconfig = ()

preinit()

ext = os.path.splitext(filename)[1].lower()

if not format:
    try:
        format = EXTENSION[ext]
    except KeyError:
init()
try:
    format = EXTENSION[ext]
except KeyError:
    raise KeyError(ext)  # unknown extension

try:
    save_handler = SAVE[format.upper()]
except KeyError:
    init()
    save_handler = SAVE[format.upper()]  # unknown format

if isPath(fp):
    fp = builtins.open(fp, "wb")
close = 1
else:
close = 0

try:
    save_handler(self, fp, filename)
finally:
    # do what we can to clean up
    if close:
        fp.close()

Listing 4.7. Analysing the pixels frequency
____Author____="Laia_Grifell"

from PIL import Image
import os
import time
import xlsxwriter

path = "C:\Users\Laia\Documents\Universitat\TFG\Images2\AUXI"
path2 = "C:\Users\Laia\Documents\Universitat\TFG\Pixels"

if not os.path.exists(path2):
    os.makedirs(path2)

images = os.listdir(path)
print images
count1 = 0
for im in images:
try:
    im2 = Image.open(path + '\\' + im).getdata()
    im1 = list(im2)
    count = 1
    dicc = {}
    for i in range(0, len(im1)):
        if im1[i] not in dicc:
            dicc[im1[i]] = 1
        else:
            dicc[im1[i]] += 1
    workbook = xlsxwriter.Workbook(path2+'\\'+im.split('.')[0] + '.xlsx')
    P1 = workbook.add_worksheet()
    for key in dicc.keys():
        P1.write(count, 0, str(key))
        P1.write(count, 1, dicc[key])
        count += 1
    workbook.close()
    print "We have done image %i of %i" %(count1, len(images))
except:
    """im2 = Image.open(path + '\\' + im).getdata()
    im1 = list(im2)
    count1 += 1
    workbook = xlsxwriter.Workbook(path2+'\\'+im.split('.')[0] + '.xlsx')
    P1 = workbook.add_worksheet()
    for i in range(0, len(im1)):
        P1.write(i, 0, str(im1[i]))
    workbook.close()
    print "We have done image %i of %i" %(count1, len(images))"""