Bachelor Thesis: Angular Distribution of Light in the Habitat of Cuttlefish

Author:
Jonas Håkansson
Department of Astronomy and Theoretical Physics, Lund University

Supervisor:
Björn Samuelsson
Division of Cell- and Organism Biology, Institution of Biology, Lund University
and
Department of Astronomy and Theoretical Physics, Lund University

Tuesday, June 01 2010

Abstract

This work presents a way of analysing photographs stored in Canon's CR2-format in order to analyse the light intensity profile of the area where the photographs were taken. A set of underwater photographs of the habitat of the Cuttlefish, *Sepia officinalis*, were provided for this thesis. The light intensity profile of the habitat of the cuttlefish is of interest because of the unusual shape of its pupils. Programs were developed for obtaining data from the provided photographs. The results show that the light conditions do vary greatly with angle.

1 Introduction

The eye of the cuttlefish, *Sepia officinalis*, is the target of a collaborative study headed by Dan Nilsson at the Division of Cell- and Organism Biology, Institution of Biology at Lund University and Lydia Mäthger at the Marine Biological Laboratory in Woods Hole Massachusetts. The cuttlefish is of interest in part because of the unusual shape of its pupils, see *Figure 1*. One of the aims of the study is to examine if the shape of the pupils can be explained as an adaptation to the general light intensity profile in the habitat of the creature.

To answer this question a clearer picture of how the light intensity in the habitat varies with angle is needed. The angle of interest is the one marked 'a' in picture *Figure 2*, and it will often be referred to as simply 'the angle' or similar in this thesis.

Figure 1: A photograph of the eye of the cuttlefish, *Sepia officinalis*, note the shape of the pupil.

I was provided with a set of photographs of the habitat. The photographs were taken at depths inhabited by the cuttlefish using a high-end digital camera with a fisheye lens. The aim of this thesis is thus to examine these picture and the structure of the file format they were stored in, in order to analyse the light conditions of the habitat of the cuttlefish.

The photographs had been stored in a raw image format, Canon's CR2. The

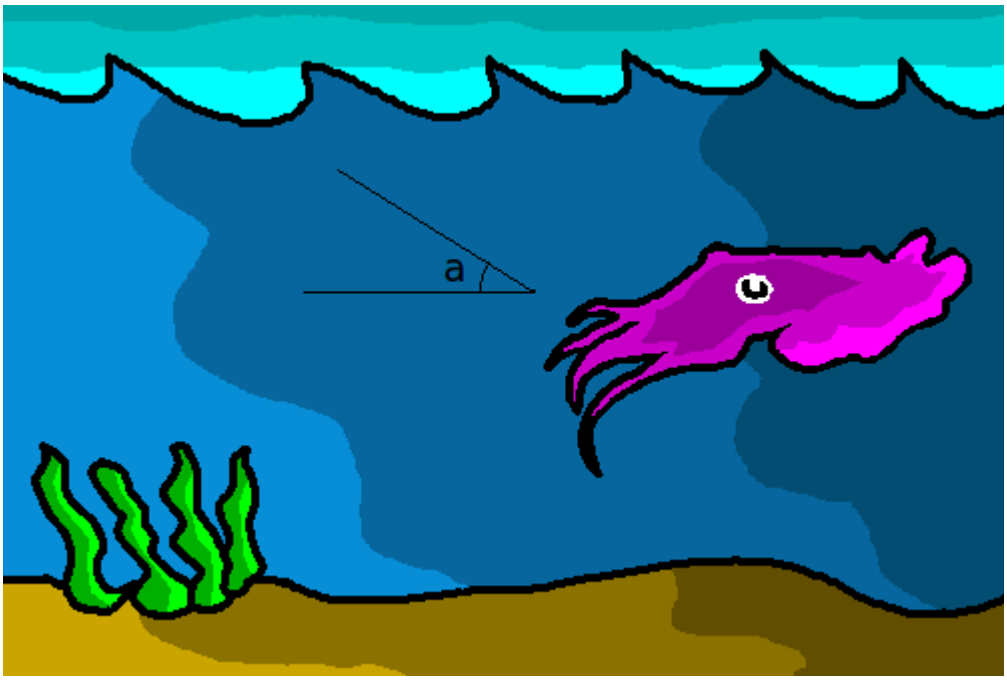

Figure 2: An illustration of the habitat in which to examine how the light intensity varies with the angle a.

advantage of raw image formats is that they store the actual sensor data, and the camera settings. When storing in non-raw formats the picture is transformed for better viewing and smaller file size (See for example Nykrog [2005]). This means that information is lost when storing in non-raw formats, and that's why the set of photographs on which to base this thesis were stored in Canon's CR2-format. One of the drawbacks with raw formats is that they tend to lack official documentation, and this is the case with the CR2-format.

Pupil shapes that deviate from the more known circular shape aren't unheard of in the animal kingdom. Hyrax, catfish, and horse are examples of animals with more exotic pupil shapes. Pupil shapes resembling that of the cuttlefish are believed to act as a sunshade and exclude strong light from above. It might also help camouflage the animal. Furthermore, it is believed to provide a more even retinal illumination than a circular pupil. [Land & Nilsson, 2002]

2 Materials and Methods

The data in this thesis is extracted from 33 photographs taken in the habitat of the cuttlefish, *Sepia officinalis*. The camera used was a *Canon EOS-1Ds Mark II* equipped with a *Sigma 8mm f/3.5 EX DG Circular Fisheye lens*. The opsins in the retina of *Sepia officinalis* are all strictly sensitive to green light, this result in the creature having monochromatic vision [Mäthger et al., 2006]. To accommodate for this, the pictures were taken with a bandpass filter centred at approximately 500 nm and with a spectral width of approximately 70 nm.

The format the images were stored in is Canon's CR2-format (this is covered in greater detail in the appendixes). Java programs were constructed in order to obtain information from the image files.

Two types of information were retrieved from the photographs — what is known as the metadata and the actual picture data. The metadata sections contain information about the camera settings, such as exposure time and the shutter opening. Parsing of the metadata sections of the photographs revealed that the camera settings didn't differ between the different photographs. Taking the photographs using different settings would have resulted in a set of photographs less suitable for comparison.

The image data was obtained in the form of a 5108×3349 matrix where each element was an integer between 0 and 3712. These elements corresponded to pixel values, with a low value corresponding to a low (dark) pixel value and a high value corresponding to a high (light) pixel value. The positions of the elements in the matrix directly corresponded to the positions of the pixels in the picture. The matrix contained information for both red, blue and green ligh, stored as explained in *Appendix B*, and illustrated in *Figure 14*. The red and blue data were stored using one colour channel each, and the green data was stored using two channels. The green stored using the first channel will be referred to as 'green1' and the green stored using the second channel will be referred to as 'green2'. The programs constructed for analysing the picture data were able to draw the pictures from the raw image data, the result is shown in *Figure 3*.

Gathering data for any part of the picture was a matter of parsing the corresponding part of the matrix containing the picture data, concentrating on the colour of interest, and simply counting how many times each of the different pixel values, ranging from 0 to 3712, appear. Data was gathered for 24 rectangles, 100 pixels high and 200 pixels wide, situated in the horizontal middle of the picture at different heights. The locations of these rectangles are shown in *Figure 4*. This was done for the whole set of pictures, 33 in total, so that 24 data files were constructed in total, each one containing sums of the number of times each pixel value appears in the current rectangle.

For each of these 24 rectangles, the mean and the standard deviation of the pixel values was calculated as following

$$\langle p \rangle = \frac{1}{N} \sum_{i=0}^{3712} i \cdot n_i \tag{1}$$

$$\sigma_p = \sqrt{\frac{1}{N} \sum_{i=0}^{3712} (i - \langle p \rangle)^2 \cdot n_i} \tag{2}$$

where $\langle p \rangle$ is the average pixel value, i is a variable ranging over the different possible pixel values (0 to 3712), n_i is the number of occurrences of the pixel value i, and N is the total number of pixels.

3 Results

The angular scale shown in *Figure 4* and used in the plots was obtained by taking a picture of a scene in which known angles were marked, using a camera

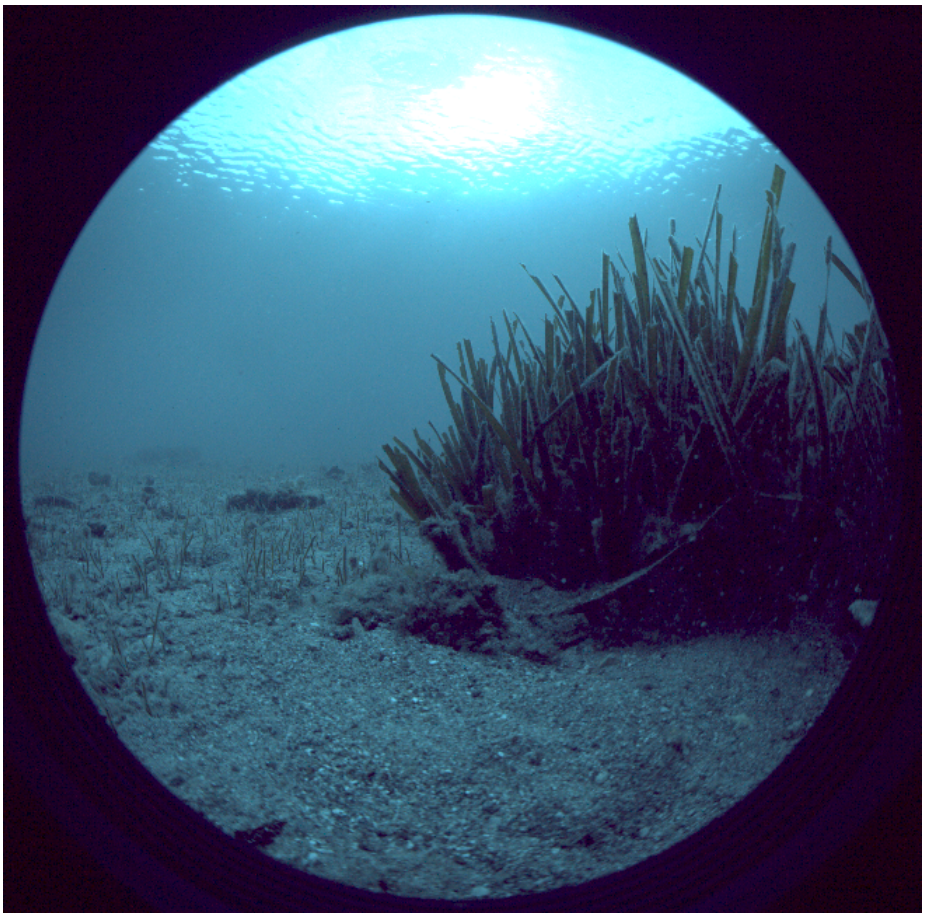

Figure 3: The result of using the constructed Java programs to "develop" one of the pictures.

equipped with the same fisheye lens as the one used when taking the 33 pictures on which this thesis is based.

Because the photographs were taken using a fisheye lens, only a circular region in the middle of the picture contain actual photography — the rest is black. Examining a rectangle in a black area of the picture reveals that the mean pixel value in a totally black area is about 130, for each of the four colours. This background noise had to be taken into consideration for the examination of the data for the 24 rectangles. The plots for the standard deviation and the means of the pixel values without background subtraction are shown in *Figure 5* and *Figure 6*.

To make the intensity distribution profiles more accessible, the data was binned. The pixel values were first lowered by 130 to account for the background noise and the binned using 20 bins per decade on a logarithmic scale. The leftmost bin was used for collecting pixel values which were lowered below zero by the background reduction. The result is shown in *Figure 7* for the green1 data, in *Figure 8* for the green2 data, in *Figure 9* for the blue data and in *Figure*

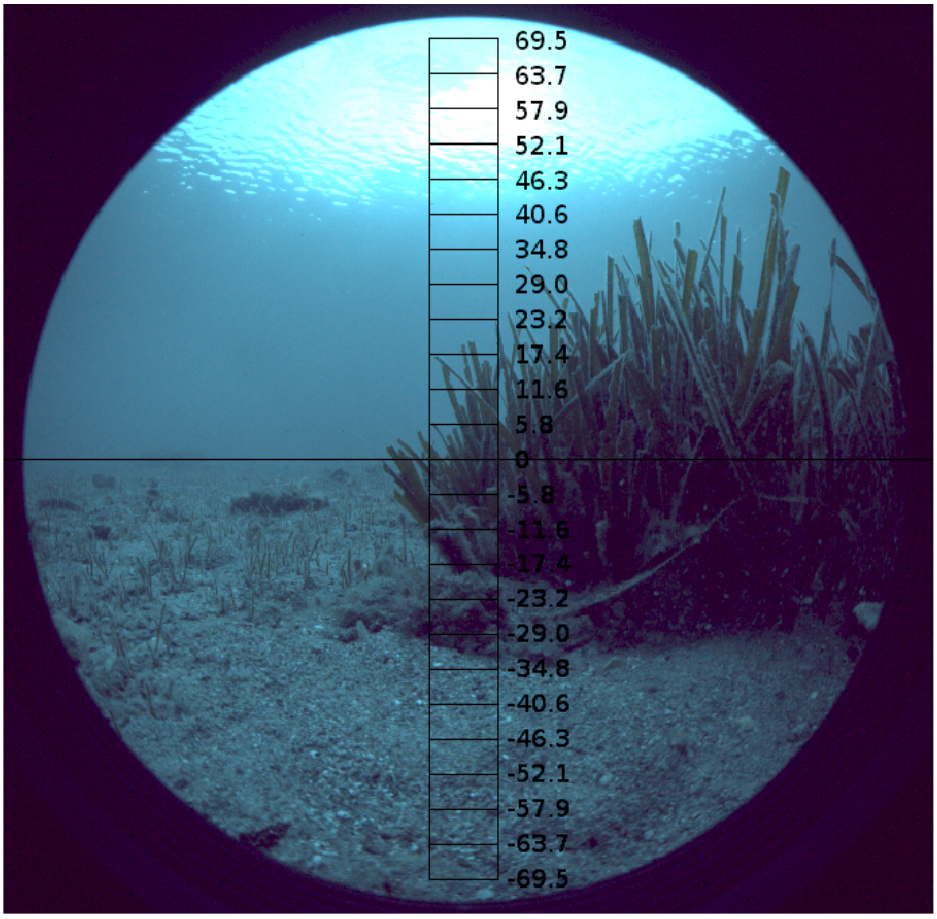

Figure 4: The 24 rectangles analysed. The numbers are the angles corresponding to the rectangles measured in degrees.

10 for the red data.

4 Discussion

From the plot of the average sensor counts, *Figure 6*, it's clear that the light intensity is highest when looking up at the surface and lowest when looking down at the bottom. A local minimum in the light intensity is situated at about $-20°$ followed by a local maxima somewhere around $-40°$. Neither of the two extremal points are very pronounced, and they might therefore be of little importance.

The distributions of the sensor counts show a clear difference in the width of the intensities covered between the different angles. The distribution width is approximately 1, 2, and 2.5 orders of magnitude at the angles 48°, 3°, and $-39°$, respectively. *Figure 8*, for instance, also shows a spike at the rightmost bin. This is explained by the camera sensor being saturated. A sensor capable of covering a greater spectrum of intensities would thus probably have shown

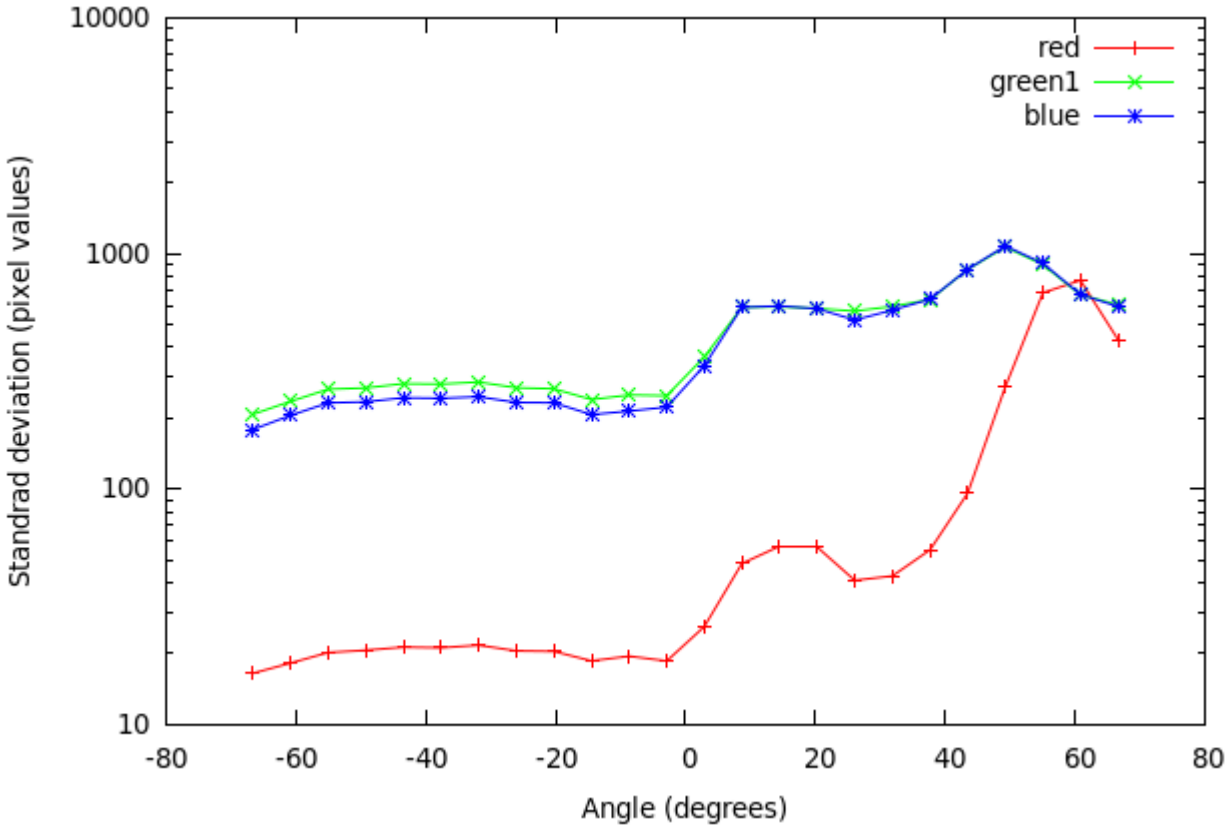

Figure 5: The standard deviations for the different rectangles. The red data is represented by the red line, the blue data by the blue line and the green1 data by the green line.

the width of the intensities covered to be greater at the larger angles.

The cuttlefish is known to have polarisation vision [Land & Nilsson, 2002]. For future experiments, photographs taken with polarising filter would therefore be of interest.

Underwater photographs of an angular scale, taken with the same camera and lens, would make direct calibration of the angles possible. This would also allow use of larger areas in the pictures to gather data from, and one wouldn't be confined to gather data from the horizontal middle of the picture. However, such photographs were not included in the set which this thesis is based on.

The *Sigma 8mm f/3.5 EX DG Circular Fisheye lens* is supposed to cover 180° [Sigma, 2010], but the manner in which the photographs were taken allowed only for an angle of view of 145°.

Most of the time put into this project was spent understanding different parts of the CR2-format and constructing Java programs to obtain data from these parts. But the programs could still be improved. For one thing they could be optimized to run faster. But more importantly they could be generalized as to be able to obtain information from CR2-files from different cameras.

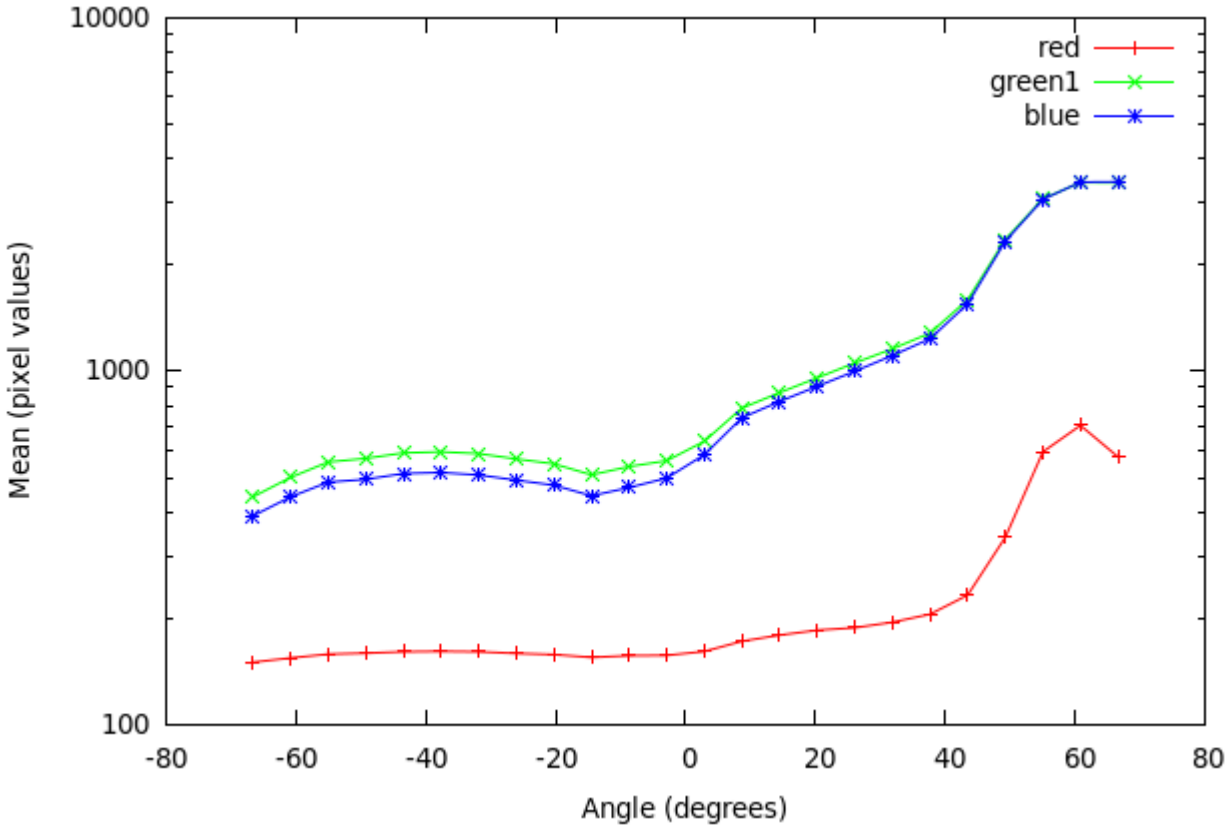

Figure 6: The means for the different rectangles. The red data is represented by the red line, the blue data by the blue line and the green1 data by the green line.

The results show the light intensity to vary greatly with the angle. A non-circular pupil could therefore provide an animal in the habitat with a more even retinal illumination, than would a circular pupil. A pupil shape like that of cuttlefish could therefore be imagined to allow an animal to view objects on different angles without moving its eyes.

5 Appendix A - Overall file structure

The file format covered here is the CR2-format, but raw image file formats vary with camera model, so that the CR2-format for one camera isn't necessarily identical to that of another camera. The CR2-images I worked with were captured using a *Canon EOS-1Ds Mark II*. I'd also like to point out that the CR2-format contains a lot of information, most of it which is far from easy to locate and understand. Fully examining the CR2-format, even the one specific to the camera used, is beyond the scope of this thesis. When unable to extract necessary information from the files manually I relied on the program *ExifTool* by Phil Harvey [Harvey, 2010] to do it for me.

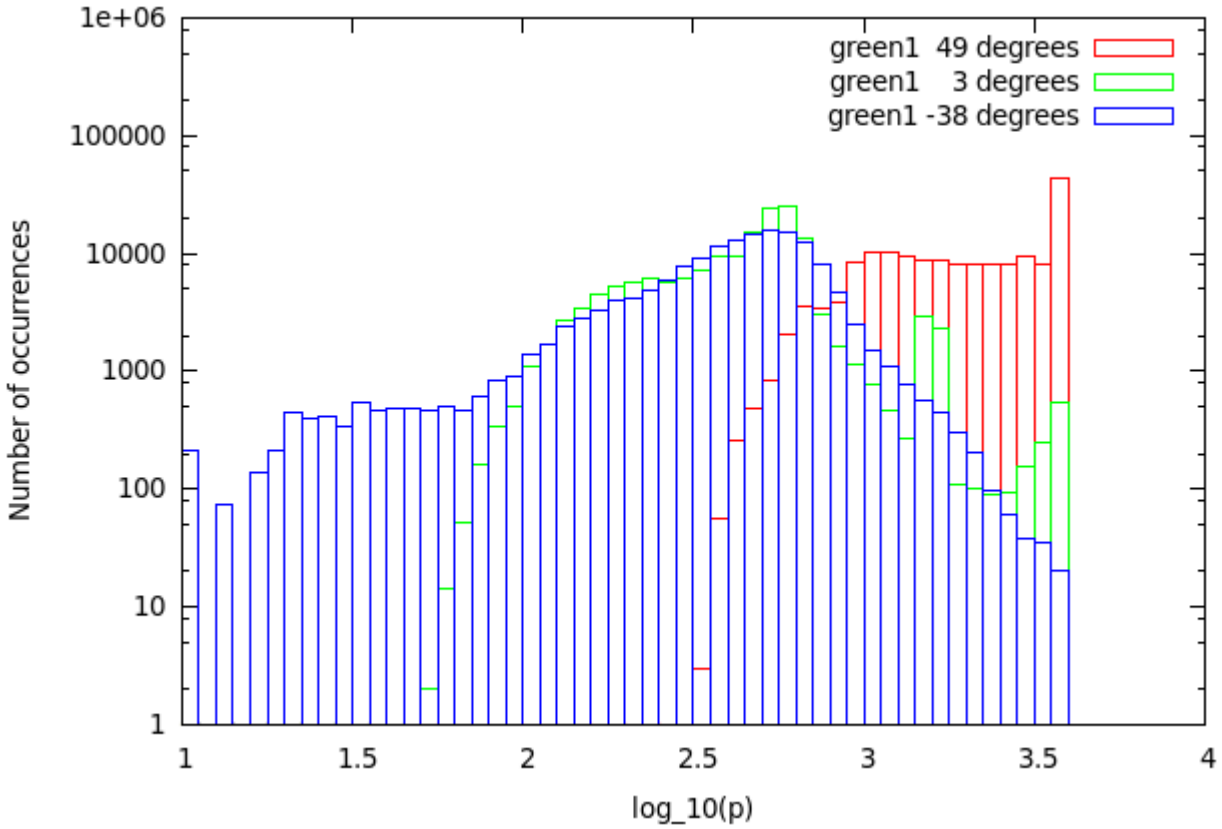

Figure 7: Plot of the binned distribution of number of occurrences per pixel value, (p), for the green1 data. The red bins represents the rectangle at 48°, the green bins represent the rectangle at 3° and the blue bins represent the rectangle at −38°.

CR2-files are what is known as binary files. A binary file, in contrast to a plain text file, is a file in which data is stored in a way suitable for reading by a program but far from suitable for reading by a human. In this appendix I will try to explain how to interpret parts of the CR2-format. The prefix '0x' is used to indicate hexadecimal numbers and '0b' to indicate binary numbers.

Using a hex editor to view a CR2-file the data is represented by two digit hexadecimal numbers. Each two digit hexadecimal number is one byte. Two useful notations are 'unsigned short', which is two bytes meant to be understood as a positive number and 'unsigned long',which is four bytes meant to be understood as a larger positive number. A signed short, or long, is capable of representing both negative and positive numbers, here we will however focus on unsigned numbers and I will use 'short' to refer to an unsigned short and 'long' to refer to an unsigned long. Keep in mind that it is common to use a notation in which 'short' refers to a signed short and 'long' refers to a signed long. As an example of the notation used here, 'cafe babe' interpreted as a long would be 0xcafebabe or 3405691582, and '0472' interpreted as a short would be

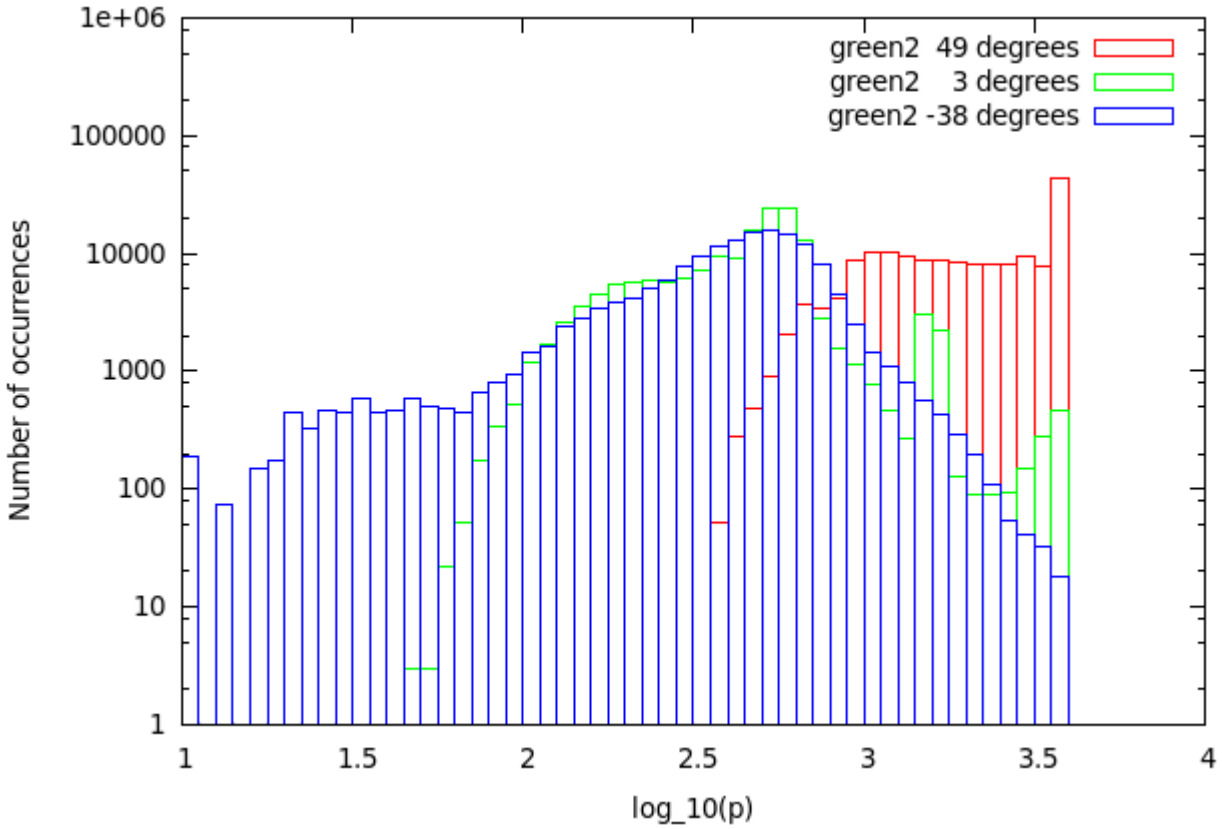

Figure 8: Plot of the binned distribution of number of occurrences per pixel value, (p), for the green2 data. The red bins represents the rectangle at 49°, the green bins represent the rectangle at 3° and the blue bins represent the rectangle at −38°.

0x0472 or 1138. The order in which to read the bytes isn't always from the left to the right, but more on that later. Some bytes are meant to be understood as characters, a plain text interpretation of these bytes will reveal which characters they are meant to be understood as.

A position in the file is referred to by offset, that is, the position in the file measured in the number of bytes from the beginning of the file so that the zeroth (it's common to start counting at zero) byte has the offset 0x0000, the sixteenth byte has the offset 0x0010 and so on. When referring to the position of a byte in a specific section of the file it is not unusual to express the position as the offset from the start of that section. It is important not to mistake the offset from the beginning of the file for the offset from the beginning of the section and vice versa.

The CR2-file format is based on the TIFF file format [Clevy, 2010]. The first thing one encounters when parsing the file is what is known as the file header. The first two bytes contain two characters indicating the byte order used in the file, in the case of the images analysed for this thesis the byte order is little

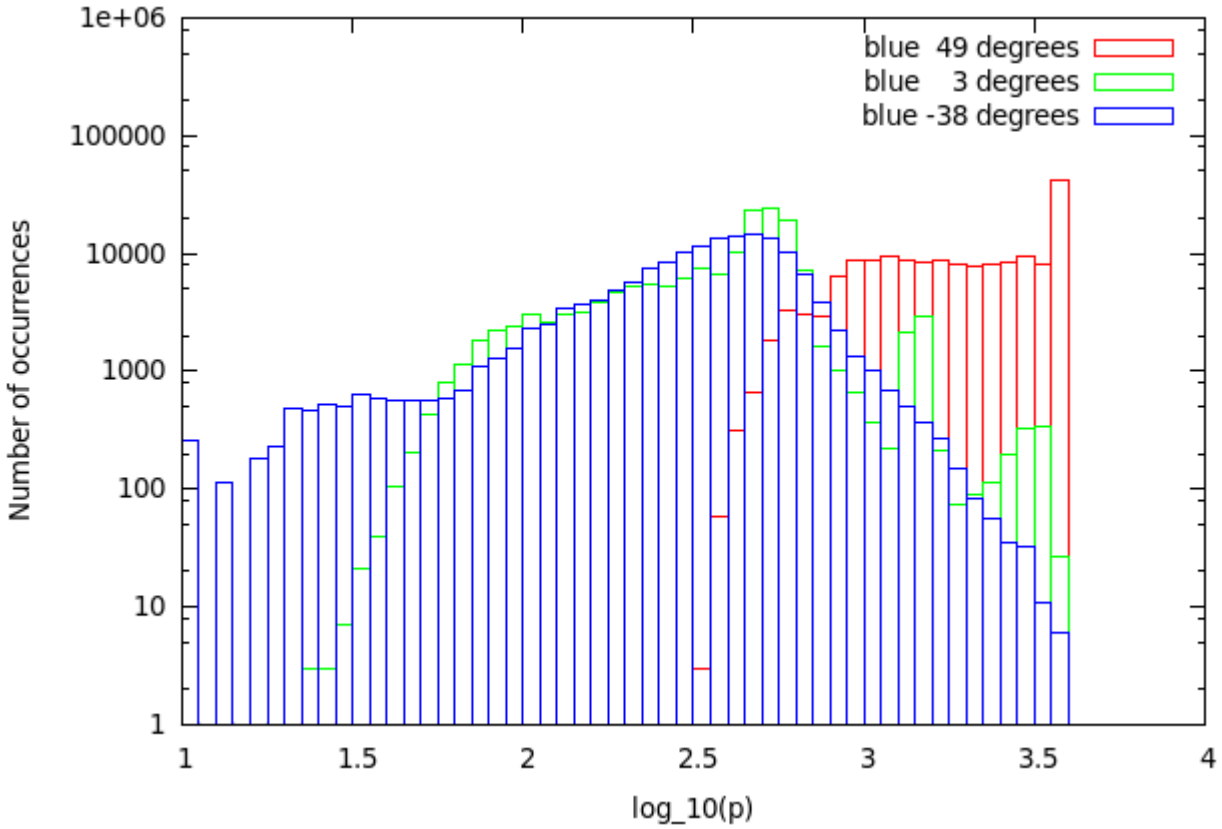

Figure 9: Plot of the binned distribution of number of occurrences per pixel value, (p), for the blue data. The red bins represents the rectangle at 48°, the green bins represent the rectangle at 3° and the blue bins represent the rectangle at −39°.

endian (also known as Intel byte order), as opposed to big endian (also knows as Motorola byte order), and this is indicated by the two characters being 'II' as opposed to 'MM'.

The byte order of the file indicates how one is supposed to read the data. The byte order being little endian means that a data type longer than one byte is to be read from the right to the left. For instance, consider the following text: '0200 9006 d406'. If this text was to be understood as three unsigned shorts (expressed using hexadecimal digits, of course), a short being two bytes long, the first short would be 0x0002 or 2, the second short would be 0x0690 or 1680 and the third short would be 0x06d4 or 1748.

We've now covered the two first bytes of the file, the next byte is a number identifying the file as being TIFF-structured, 0x2a or 42. The next four bytes, '1000 0000', constitute a long 0x0000 0010, which is to be interpreted as the offset to the beginning of the first IFD, IFD0. Notice how the four bytes has to be read using little endian byte order when interpreting them as a long.

The next two bytes contain two characters, 'CR', indicating that the file is

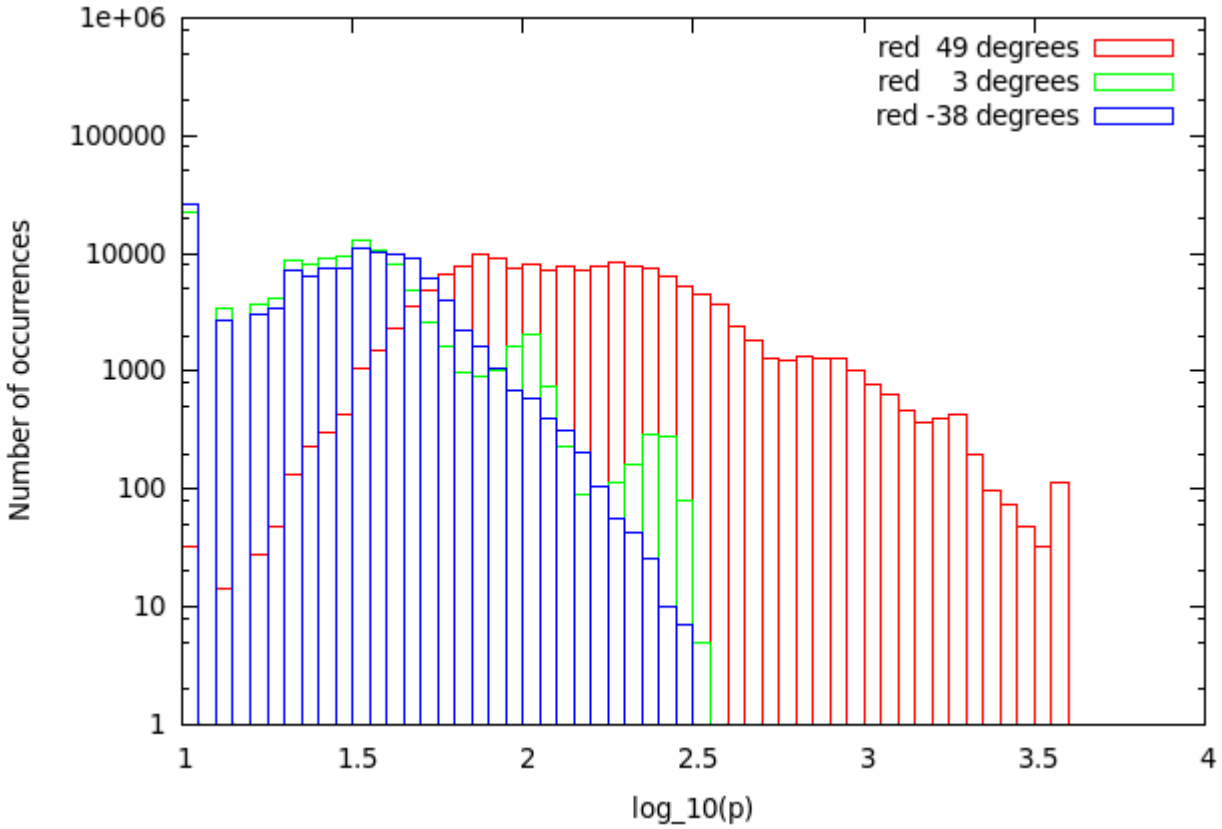

Figure 10: Plot of the binned distribution of number of occurrences per pixel value, (p), for the red data. The red bins represents the rectangle at 48°, the green bins represent the rectangle at 3° and the blue bins represent the rectangle at −39°.

a CR2-file. There are a few more bytes before the end of the file header and the beginning of the first IFD, but they are of little interest to us.

The IFDs can be thought of as directories storing a certain representation of the actual image [Clevy, 2010], these representations can be thumbnail pictures, preview pictures, or smaller greyscale pictures. In the first IFD, IFD0, however, in addition to a small RGB version of the actual picture, a pointer to what is known as the EXIF sub directory is found, but more on this later. The structure of an IFD is fairly straightforward (see for example Adobe [1992] and Clevy [2010]). It contains a number of twelve byte entries which in turn contain either a small volume of data or an offset to a larger volume of data. The number of entries is described by the first two bytes in the IFD. The last four bytes of the IFD contains the offset to the next IFD, if the current IFD is the last IFD, which in our case is IDF3, then the last four bytes simply point to the start of the entire file, that is offset 0x0000 0000. The structure of the IFDs is illustrated in *Table 1.*

The structure of the entries in the IFDs is also fairly straightforward (see

Offset	Content
0	Number of entries, X
2	Entry 0
14	Entry 1
...	...
$2 + X \times 12$	Offset to next IFD

Table 1: The structure of the image file directories (IFDs).

Type	Indicator	Length (bytes)
String	2	2 (bytes) per character
Unsigned short	3	2
Unsigned long	4	4
Unsigned rational	5	8

Table 2: The types encountered in the entries of the IFDs.

for example Adobe [1992] and Clevy [2010]). The first two bytes is simply a tag by which to recognize the entry. The next two bytes is a type indicator, that is a number associated by a way in which to interpret the information stored, or linked to, in the entry. The types encountered in the entries and their corresponding type indicators are summarized in *Table 2*. The next four bytes contain a number indicating how many of the indicated data type the entry contains or points to. The next four bytes either contain the data or an offset to it. The structure of the entries is illustrated in *Table 3*.

The IFDs of interest to us are the first IFD, IDF0 and the fourth and last IFD, IFD3. IFD0 contains a pointer to the EXIF subdirectory which in turn contains a link to the Makernote subdirectory. The EXIF and Makernote subdirectories are interesting to us because they contain information about the camera settings, such as exposure time and shutter opening. It is important to be aware of the camera settings since photographs meant to gather data from should preferably be taken with the same settings.

Unfortunately I was unable to get a clear picture of how to read the information stored in the EXIF and Makernote subdirectories. I ended up using *ExifTool* by Phil Harvey [Harvey, 2010] to extract the EXIF and Makernote data. The data extracted revealed that the photographs were all taken with the same camera settings.

The last IFD, IFD3 is the one of greatest interest to us, it contains the raw image data compressed to a lossless JPEG. The second entry of IFD3, marked with the tag '0x0111' contains a pointer to the raw image data. The sixth entry, marked with the tag '0xc640' contains what is known as the slices or submatrices of the image data.

6 Appendix B - Decoding the raw image data

Decoding the actual image data is done in several steps. The first step is to locate the raw image data. The first entry in IFD3, marked with the tag '0x0111', points to the start of the image data which is marked with ther tag '0xffd8'. This tag is recognized as the start of image (SOI) marker (for a list of different

Offset	Content	Length (bytes)
0	Tag	2
2	Type	2
4	Count	4
8	Data or pointer to data	4

Table 3: The structure of the entries in the IFDs.

'0xff'-markers, see for example ITU [1992]).

The file sections covered here, the ones marked with a tag composed of '0xff' followed by another byte, are not to be read using the little endian byte order, that is, we are now supposed to read from the left to the right.

Continuing on, the first two bytes encountered after the SOI marker is another marker, this time it is '0xffc4' which is the define Huffman table (DHT) marker. This section will be covered in detail later. The two next bytes encountered inform us on the length of this section, 62 bytes. This length includes the two bytes used to describe the length of the section. If we for now ignore the next DHT section and jump 60 bytes forward, we reach the start of frame (SOF) marker, '0xffc3'. As with the DHT section, the first two bytes encountered after the marker tell us the length of this section, 14. Ignoring the SOF section for now and jumping 12 bytes forward takes us to the start of scan (SOS) marker '0xffda'. The two first bytes of the section once again tell the length of the current section, 14. Jumping 12 bytes forward we reach the first byte of the actual raw image data. We have thus successfully located the raw image data.

The raw image data looks like an extremely long string of bytes. Until now the hexadecimal representation of bytes have been the most convenient, but when dealing with the raw image data it is much more useful to use a binary representation of the bytes.

What one now must take into consideration is that the number 0xff, 0b11111111, in binary, followed by another number (like 0xc4, 0xc3 or 0xda) usually marks the beginning of a section. This could potentially cause problem every time the number is needed for storing image data. The solution is to add what is known as padding bytes every time the number is used. The number chosen to use as padding bytes is 0x00, or 0b00000000. So when parsing the image data, if one encounters the number 0xff00, or 0b11111111 00000000 this should be interpreted as 0xff, or 0b11111111. So if encountering 0xff, or 0b11111111, followed by anything other than 0x00, or 0b00000000, one can be certain that one is no longer parsing the raw image data. The end of the image data is actually marked with a tag known as the end of scan (EOS) tag, '0xffd9', or '0b11111111 11011001'.

Before covering the DHT section and using it to decode the raw image data, some attention will be devoted to the SOF and SOS sections. I'll start with the SOF section. The first byte after the two bytes describing the length of the section informs us about the bit precision of the image, 0x0c. A bit precision of twelve means that we can use twelve bits to describe a pixel. This is important when trying to make sense of the DHT section since the number of Huffman keys per channel used to encode the image into raw image data is one more than the bit precision. The concepts 'Huffman keys' and 'channels' are defined shortly. The next two bytes inform us on the height of the image, which is 3349

pixels, and the two bytes after that inform us on the width of the image, which is 5108 pixels. The next byte inform us on the number of channels which is 2.

That there are two channels simply means that we have to construct two define Huffman tables. These are the tables used to decode the raw image data. The first value of the first pixel is decoded using the first define Huffman table and which table to use the alternates between the first and the second one. Pixel values obtained for one channel, using one define Huffman table, is to be kept separate from pixel values obtained for another channel, using another define Huffman table.

Huffman coding isn't something I have a full understanding of, but for our intents and purposes it's enough to know how to extract the keys for decoding the Huffman compressed image data and then how to use these keys to actually decode it. The DHT section is easier to understand if presented byte for byte, as in *Table 4*.

Offset	Data			Explanation
0x0000	0xffc4			DHT marker.
0x0002	0x003e			Length of this section, in our case it is 62 bytes.
0x0004	0x00			Marker telling us that the first define Huffman table is beginning.
0x0005	0x00	0x02	0x02	These numbers are informing us on how to build the Huffman keys for the first define Huffman table. More on this later.
	0x02	0x03	0x01	
	0x01	0x01	0x01	
	0x00	0x00	0x00	
	0x00	0x00	0x00	
	0x00			
0x0015	0x04	0x05	0x03	These numbers are informing us on what the first set of Huffman keys actually mean. More on this later.
	0x06	0x07	0x02	
	0x08	0x00	0x01	
	0x09	0x0a	0x0c	
	0x0b			
0x0022	0x01			Marker telling us that the second define Huffman table is beginning.
0x0023	0x00	0x03	0x01	These numbers are informing us on how to build the Huffman keys for the second define Huffman table. More on this later.
	0x01	0x01	0x01	
	0x01	0x01	0x01	
	0x01	0x01	0x01	
	0x00	0x00	0x00	
	0x00			
0x0033	0x00	0x01	0x02	These numbers are informing us on what the second set of Huffman keys actually mean. More on this later.
	0x03	0x04	0x05	
	0x06	0x07	0x08	
	0x09	0x0a	0x0b	
	0x0c			

Table 4: The 0xffc4 section of the file explained.

Constructing the Huffman keys is fairly simple. Each key is composed of a number of bits, in our case '0b111110' is one key, '0b100' is another. To find these numbers for the first define Huffman table we first look at the numbers provided above, the ones at offset 0x0005. The first number tells us how many keys have a length of one digit, the second one tells us how many have a length of two digits, and so one. The keys are then constructed by starting at zero, that is 0b00. This is our first key. The second key is then constructed by increasing the number of the first key with one, our second key is thus 0b01. Keeping in mind that we only want two keys of length two, constructing the third key is somewhat trickier, we increase the number of the previous key with one, reaching 0b10 and then we put a zero at the end of the number reached this way, the third key is thus 0b100.

The algorithm used to construct the Huffman keys can be summarized as this: The first key is 0b00. Constructing a key (other than the first one) is done by adding one to the previous key. If the quota of keys with the same length as the previous key isn't filled by the previous key, the number reached this way is the desired key, otherwise we must also put a zero at the end of this number for it to be the key we're seeking.

The number described as the meaning of the keys, the ones at offset 0x0015 and 0x0023 in *Table 4* are to be paired with the keys found using the algorithm described above. The first number is to be understood as the meaning of the first key, the second number is to be understood as the meaning of the second key and so on. The whole procedure is shown in *Table 5* for the first define Huffman table and in *Table 6* for the second define Huffman table.

Length of key	Key	Meaning of key
2	0b00	0x04
	0b01	0x05
3	0b100	0x03
	0b101	0x06
4	0b1100	0x07
	0b1101	0x02
5	0b11100	0x08
	0b11101	0x00
	0b11110	0x01
6	0b111110	0x09
7	0b1111110	0x0a
8	0b11111110	0x0c
9	0b111111110	0x0b

Table 5: Define Huffman Table 1

Although I've decided to call the numbers in the rightmost columns the meanings of the keys, there's more to them than small numbers. In the end, we want the keys to be able to find pixel values, and to describe a pixel we need more than a number between zero and twelve.

The image data is, as described earlier, represented by an enormous string of ones and zeros. Decoding the data consist of reading this string, starting from the beginning, until a combination of ones and zeros similar to one of the keys in *Table 5* or *Table 6* is found. When a key is recognized, its value is interpreted

Length of key	Key	Meaning of key
2	0b00	0x00
	0b01	0x01
	0b10	0x02
3	0b110	0x03
4	0b1110	0x04
5	0b11110	0x05
6	0b111110	0x06
7	0b1111110	0x07
8	0b11111110	0x08
9	0b111111110	0x09
10	0b1111111110	0x0a
11	0b11111111110	0x0b
12	0b111111111110	0x0c

Table 6: Define Huffman Table 2

as how many additional digits we are supposed to read, these additional digits are then to be understood as the value for a pixel.

But understanding these additional digits isn't as simple as interpreting the digits as a binary number. The first thing we do when decoding this string of ones and zeros is to look at the first digit, if it is a one, the string is to simply be read as a binary number and this number then constitute the value of the pixel in question. If however the first digit is a zero, the value of the pixel is to be understood as minus the binary number obtained by replacing every one in the string with a zero, and every zero with a one [WT, 2010]. Which of the two define Huffman tables we are supposed to use is simple, we start with the first one and then alternate between the two.

To further illustrate the process of decoding the image data, I provide an example. Look at the string of digits in *Figure 11*. We start by reading the string from the beginning, looking for a key from the first define Huffman table, *Table 5*. After reading five digits, marked a, we recognize "0b11100" as the key for "0x08", this means that we are to read the next eight digits, marked b. Doing that gives us the number 0b10011010, since the first digit is a one we can conclude that the value of the first pixel is 154. Now we start over, reading from where we stopped and looking for a key from the second define Huffman table, *Table 6*. After reading four digits, marked c, we recognize "0b1110" as the key for "0x04", this means that we are to read the next four digits, marked d. Doing this gives us the number 0b0010, since the first digit is a zero we haver to do more than simply interpret this number as the value of the pixel. We start by replacing every zero with a one and every one with a zero. This gives the number 0b1101, which we understand as 13. This means that the value for the second pixel is -13.

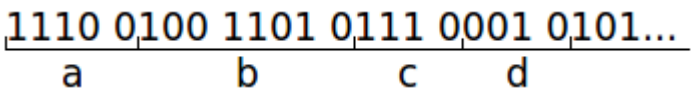

Figure 11: An example of a string of ones and zeros to decode.

The pixel values now reached aren't final, they are to be altered further. Before we start to alter the values of the pixels we must first place the pixels in an appropriate matrix. As mentioned earlier in this the width of the images is 5108 pixels and the height is 3349 pixels. This means that we are supposed to place the pixels in a 5108×3349 matrix starting at the top left corner of the matrix and then fill the matrix row after row, like illustrated in *Table 7*.

1	2	...	5107	5108
5109	5110	...	10215	10216
...	...	...	...	...
17101585	17101586	...	17106691	17106692

Table 7: How to place the pixels in the 5108×3349 matrix, the numbers are meant to be understood as the pixel numbers.

Having the pixel values neatly placed in a matrix we are now ready to use these values to obtain the actual pixel values, the values suitable to correspond to actual light intensity. Every pixel is to be understood as the difference between the actual value of the current pixel and the actual value from the previous pixel, that is, you get the actual value of a pixel by adding it's current value to the value of the actual value of the previous pixel. Here as always, we are to keep the two channels separated, 'previous pixel' is thus meant to be understood as the previous pixel belonging to the same channel as the current pixel.

But how are we to move through the matrix when adding pixel values together, that is, which path are we to follow? The answer is that there are several paths to follow. First of all, we construct the two first pixels. The first to pixels are located in the top left corner of the matrix. Constructing them is done by adding 2048 to each of them. To obtain the final values for the rest of the pixels we start by moving from the top left corner, down to the bottom left corner, adding the previous pixel value to the current one. There being two channels means that we have to do this twice, once for the first channel, starting at the first element of the first row and once for the second channel, starting at the second element of the first row. When this is done, we have start values for each line. Now, for every line we start at the second pixel and go from left to right, calculating every actual pixel value by adding its current value to the actual value of the previous pixel value. This is illustrated in *Figure 12*.

With every actual pixel value now calculated one would expect the image to be properly decoded. This, however, is not the case. As mentioned in *Appendix A* the sixth entry in IFD3 contains information about the so called slices, or submatrices, that the image are stored in. Following the link provided in this entry we find the following information "2, 1680, 1748". The first number, 2, is the number of slices of width 1680. The third number, 1748, is the width of the last slice. It is hence clear that we have three slices, and the all have the same height as the matrix containing the whole image data.

To fill these slices we start by numbering the pixels in the whole matrix. We start counting at the top left corner, going from left to right, and when we reach the end of the first row we simply continue counting from the leftmost pixel in the row below. We then place the first $1680 \cdot 3349$ pixels in the first slice, starting from the top left corner and then going from left to right simply starting at the leftmost pixel on the row below when we reach the end of the

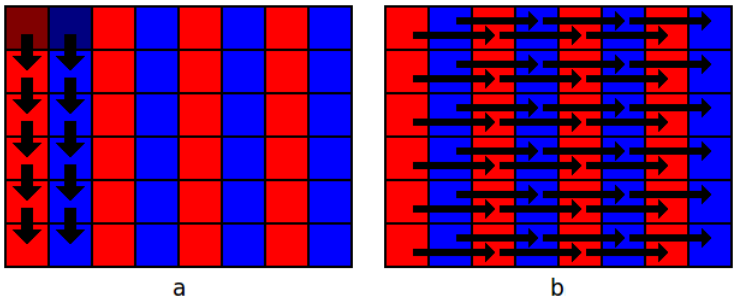

Figure 12: An illustration of which path to follow when constructing the actual pixel values from the first (difference) pixel values. The red cells represent pixels decoded using the first define Huffman table and the blue represent pixels decoded using the second define Huffman table. The arrows in a indicate which paths to follow when creating start values for each row, and the arrows in b indicate which path to the follow to calculate the actual values for the pixels on every row.

current row. We do the same with the next $1680 \cdot 3349$ pixels, but this time placing them in the second slice. Next we do the same thing with the next $1740 \cdot 3349$ pixels, but this time placing them in the third slice. The final matrix is then simply constructed by placing the three slices in a 5108×3349 matrix, side by side, starting with the first one and ending with the last one. This is illustrated in *Figure 13*.

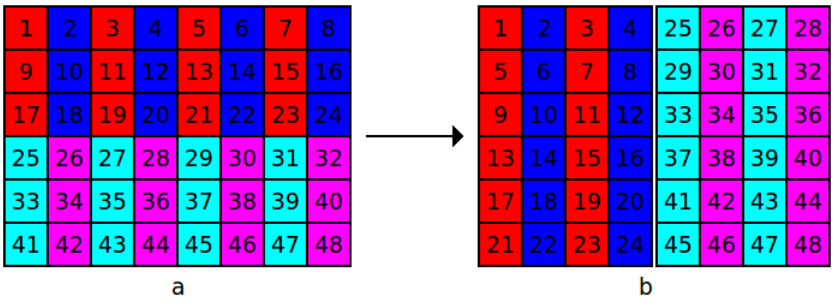

Figure 13: An illustration of how to rearrange the calculated pixel values to obtain the actual picture. The red and blue cells represent channel 1 respective channel 2 pixels belonging to the first slice, which in this example is 4 pixels wide and 6 pixels high. The turquoise and purple cells represent channel 1 respective channel 2 pixels belonging to the second slice, which in this example is 4 pixels wide and 6 pixels high. The numbers are the pixel numbers. There were actually three slices, two were 1680 pixels wide and 3349 pixels high, and the third one was 1740 pixels high and 3349 pixels high.

When this is done, we have a 5108×3349 matrix. Each element will be an integer, which it turned out lies between 0 and 3712. We now have the entire data which is to be analysed, but it's the data for four colours. The colours are red, green and blue, but green is captured by both channels since it is considered the more important of the colours since the human eye is more

sensitive to green than to red and blue. We'll refer to the green captured by the first channel as 'green1' and to the green captured by the second channel as 'green2' and treat them as if they were different colours. The four colours are stored in the matrix quite orderly [Clevy, 2010]. Every even numbered element of every even numbered row (and we start counting at zero, the top left element is thus the zeroth element of the zeroth row) contains data for a green1 pixel, every odd numbered element of every even numbered row contains data for a blue pixel, every even numbered element of every even odd numbered row contains data for a red pixel and every odd numbered element of every odd numbered row contains data for a green2 pixel. This is illustrated in *Figure 14*.

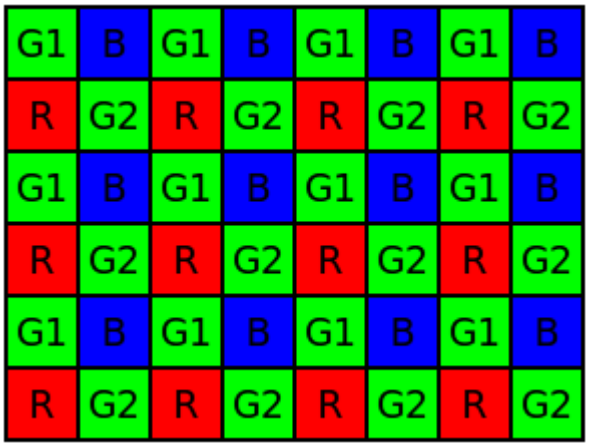

Figure 14: An illustration of how the different colours are stored in the picture matrix.

7 References

[Adobe, 1992] Adobe Developers Association (1992), TIFF - Revision 6.0, `http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf` retrieved June 02 2010

[Clevy, 2010] Laurent Clevy, Understanding What is stored in a Canon RAW .CR2 file, How and Why, `http://lclevy.free.fr/cr2/`, retrieved June 02 2010

[Harvey, 2010] Phil Harvey, exiftool Application Documentation, `http://www.sno.phy.queensu.ca/~phil/exiftool/exiftool_pod.html`, retrieved June 02 2010
software available at: `http://www.sno.phy.queensu.ca/~phil/exiftool/`

[ITU, 1992] International Telecommunication Union (1992), Recommendation T.81: Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines (1992), `http://www.w3.org/Graphics/JPEG/itu-t81.pdf`, retrieved June 02 2010

[Land & Nilsson, 2002] Land, M.F., Nilsson D.-E., Animal Eyes. Oxford University Press 2002

[Mäthger et al., 2006] Mäthger, L.M., Barbosa, A., Miner, S., Hanlon, R.T., Color blindness and contrast perception in cuttlefish (Sepia officinalis) determined by a visual sensorimotor assay. Vision Reasearch 46 (2006) 1746-1753

[Nykrog 2005] Nykrog, T, Digitalfoto. ICA bokförlag 2005

[Sigma, 2010] Sigma, `http://www.sigmaphoto.com/shop/8mm-f35-ex-dg-circular-fisheye-sigma`, retrieved Monday, June 07 2010

[WT, 2010] Wildtramper.com, Canon's CR2 Raw File Format Specification, `http://wildtramper.com/sw/cr2/cr2.html`, retrieved June 02 2010