A NEW IMAGE COMPRESSION ALGORITHM:
HIERARCHICAL PIXEL AVERAGING (HPA)

by

RICCARDO IUDICA

A THESIS SUBMITTED FOR THE DEGREE OF

MASTER IN AEROSPACE SCIENCE AND TECHNOLOGY

ADVISORS:
ENRIQUE GARCÍA–BERRO MONTILLA
JORDI PORTELL I DE MORA

Barcelona, February 2014
All that is gold does not glitter,
Not all those who wander are lost;
The old that is strong does not wither,
Deep roots are not reached by the frost.

## Contents

1 Introduction .................................................. 5  
1.1 Different kind of compression techniques ............... 5  
1.2 Data compression in Space ........................................ 6  
1.2.1 The HPA algorithm ............................................ 7  
1.3 Organization of this work ....................................... 7  

2 The Hierarchical Pixel Averaging algorithm ............... 9  
2.1 Subdivision of $16 \times 16$ pixel blocks ..................... 9  
2.2 Differential values ............................................ 11  
2.3 HPA coefficients and remainders ............................. 12  
2.4 Overhead of the HPA algorithm ............................... 13  
2.5 Limitations of HPA ............................................. 13  

3 HPA pre-processing algorithm ................................. 15  
3.1 HPA algorithm enabling ....................................... 15  
3.2 Algorithm for pixel rearrangement ......................... 16  
3.3 The HPA blocks processing ...................................... 18  
3.3.1 Calculation of the HPA coefficients ....................... 18  
3.3.2 Coefficient interleaving ..................................... 19  
3.3.3 Writing the HPA coefficients .............................. 22  
3.3.4 Writing the remainders ..................................... 23  
3.4 Histogram Balancing algorithm .............................. 23  
3.5 Sign bit processing ............................................ 27  

4 Inverse algorithm ............................................... 31  
4.1 The REMReading algorithm .................................... 32  
4.2 The inverse sign bit algorithm ............................... 33  
4.3 The inverse histogram balancing algorithm ............... 34  
4.4 The inverse HPA block array processing .................... 35  
4.4.1 Processing of the interleaved coefficients ............. 36  
4.4.2 Reading the HPA coefficients ............................. 36
Some years ago I was lost: I didn’t know exactly what to do with my life, I was doing a job that didn’t interest me, for I had lost the “path of Science”. Little by little, the disappointment I was feeling started to disappear and I began to ask myself what I really wanted, what has always interested and motivated me; what was that field which I would never feel too tired or bored to work on. The answer was crystal clear: the Aerospace field! It was hard to see at first but, thanks to some hints I collected in the way, I realized it.

That is how I got back to the path of Science and decided to study the Master in Aerospace Science and Technology. The way was found and I took it, trying to give it my best; sometimes it was hard, I had to divide myself between my job and the university. But, if you really want something, you just have to go and get it. And so I did.

First of all I would like to thank my tutors Enrique and Jordi (in the order I met them) for all the support they both gave me during this Master thesis. Working with them has been very motivating, the atmosphere was always nice and they always gave me interesting suggestions on the direction to take, especially when I was stuck in situations with not so promising (that is an euphemism for bad) results. With them I learnt how hard, funny and motivating at the same time can be a scientific investigation.

I have dis-learnt the definition of Sundays or holidays tough, being them the firsts to answer my mail at every time in every part of the year! And, of course, I replied them back as soon as I could. And I thank them for teaching me this, too :)

I have to thank Cintia, my girlfriend, for all the patience and the support she showed me in these months. She shared with me my successes and my failures, my happiness and my sadness and supported me in the difficult moments. She was precious.

My family, my parents and my sister, have always been by my side. They are used to my firm decisions and patiently support and help me whenever I need it; they have the gift, as almost every family does, to understand me even if I say nothing at all.

A special thank goes to two persons who, some years ago when I was lost, reminded me how much I love Science. It is thanks to them that I found my way back
to Science. They also reminded me that no matter how difficult things are, if we like them, we have to go and get them. So, many thanks go to Christoph and to Inma.

I cannot forget to mention all my friends, my colleagues at work or at the gym, who showed interest when I told them about the thesis and patience when I was, inevitably, talking too much. I am known for speaking sometimes too much; and they handled it perfectly :) So, many thanks to all you guys.

Walk with the dreamers, the believers, the courageous, the cheerful, the planners, the doers, the successful people with their heads in the clouds and their feet on the ground. Let their spirit ignite a fire within you to leave this world better than you found it.

Wilferd Peterson
# List of Figures

2.1 HPA block, formed by 16×16 pixels. ........................................ 10
3.1 Structure of an example image made of 32×32 pixels. ................. 16
3.2 Pseudo-coordinates of the $i$-th block. ................................. 17
3.3 Code that implements the identification of the pixels, given the $i$-th
block. ................................................................................. 17
3.4 HPA algorithm flow diagram. ................................................. 18
3.5 HPA algorithm flow diagram.................................................. 19
3.6 Code that implements the HPA coefficients calculation. ......... 20
3.7 Compression ratios for different sizes of HPAArray. ..................... 21
3.8 Code that implements the HPA arrays processing. .................. 22
3.9 Code that implements the buffer reduction algorithm. ............ 23
3.10 Code that implements the HPA values copying algorithm. ....... 23
3.11 Code that implements the interleaving and copying algorithm. .... 24
3.12 Original and balanced histograms of LevODif coefficients. ......... 25
3.13 Code that implements the histogram balancing algorithm. ........ 26
3.14 Code that calls the Sign Bit Detection algorithm. ................. 27
3.15 Code that performs the Extract Sign Bit function. ................. 28
4.1 Flow diagram representing the global HPA image compression and
decompression process. ....................................................... 31
4.2 Structure of the HPA data packet. ........................................... 32
4.3 Code that implements the RemSplitter function. .................... 33
4.4 The piece of code that implements the LevSRem remainders reading
stage. ...................................................................................... 33
4.5 The fragment of code that implements the inverse sign bit algorithm. 34
4.6 The piece of code that implements the inverse histogram balancing
algorithm. ........................................................................... 35
4.7 Flow diagram of the inverse HPA algorithm. ........................... 36
4.8 Part of the code that implements the inverse HPA array processing. 36
4.9 Part of the code that implements the buffer expansion algorithm. 37
4.10 Flow diagram of the inverse HPA algorithm. .......................... 38
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.11</td>
<td>Fragment of code that implements the inverse HPA algorithm.</td>
<td>38</td>
</tr>
<tr>
<td>4.12</td>
<td>Part of the code that implements the pixels reconstruct from the HPA coefficients.</td>
<td>39</td>
</tr>
<tr>
<td>4.13</td>
<td>Original and reconstructed image after block rearrangement.</td>
<td>39</td>
</tr>
<tr>
<td>4.14</td>
<td>The piece of code that implements the inverse block algorithm.</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>Sample image: Banyoles (1024×592 pixels, 8-bit grayscale).</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>Sample image: Catedral (1024×592 pixels, 8-bit grayscale).</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Sample image: Eixample (1024×592 pixels, 8-bit grayscale).</td>
<td>43</td>
</tr>
<tr>
<td>5.4</td>
<td>Sample image: Field (1024×592 pixels, 8-bit grayscale).</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Sample image: Mosaic10 (1024×1024 pixels, 16-bit grayscale).</td>
<td>44</td>
</tr>
<tr>
<td>5.6</td>
<td>From left to right and from top to bottom: image composed of the level 4, 3, 2 and 1 (zoomed) average coefficients.</td>
<td>45</td>
</tr>
<tr>
<td>5.7</td>
<td>Mixed representation scheme.</td>
<td>46</td>
</tr>
<tr>
<td>5.8</td>
<td>Mixed representation of the average coefficients from the different layers.</td>
<td>47</td>
</tr>
<tr>
<td>5.9</td>
<td>Images with ∆H&lt;sub&gt;0&lt;/sub&gt; centered in 0 (left panel) and centered in 128 (right panel).</td>
<td>47</td>
</tr>
<tr>
<td>5.10</td>
<td>Histograms with ∆H&lt;sub&gt;0&lt;/sub&gt; centered in 0 (left panel) and 128 (right panel).</td>
<td>48</td>
</tr>
<tr>
<td>5.11</td>
<td>Histogram of the original image.</td>
<td>48</td>
</tr>
<tr>
<td>5.12</td>
<td>Log of the comparison between the original files and the decoded ones.</td>
<td>49</td>
</tr>
<tr>
<td>5.13</td>
<td>Comparison of the FAPEC compression ratios with and without HPA.</td>
<td>50</td>
</tr>
<tr>
<td>5.14</td>
<td>Comparison of the FAPEC execution time with and without HPA.</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Structure of the lossless HPA data packet.</td>
<td>54</td>
</tr>
<tr>
<td>6.2</td>
<td>Implementation of the rounding of the ∆H&lt;sub&gt;0&lt;/sub&gt;.</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Structure of the Lossy HPA data packet.</td>
<td>55</td>
</tr>
<tr>
<td>6.4</td>
<td>Qualitative comparison of the entropy of the HPA coefficients for the lossy version (top panel) versus the lossless version of the compression algorithm (bottom panel).</td>
<td>55</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of the Lossy HPA prototype versus JPEG.</td>
<td>56</td>
</tr>
<tr>
<td>6.6</td>
<td>Execution time of the Lossy HPA+FAPEC image compression algorithm.</td>
<td>57</td>
</tr>
<tr>
<td>A.1</td>
<td>Representation of the removal of a coefficient in HPA Stopper.</td>
<td>69</td>
</tr>
<tr>
<td>A.2</td>
<td>Data structure of the HPA Stopper scheme.</td>
<td>70</td>
</tr>
<tr>
<td>A.3</td>
<td>Data structure of the HPA Sum scheme.</td>
<td>70</td>
</tr>
<tr>
<td>A.4</td>
<td>Data structure of the Dif HPA Sum scheme.</td>
<td>71</td>
</tr>
<tr>
<td>A.5</td>
<td>Data structure of the 3 levels HPA.</td>
<td>72</td>
</tr>
<tr>
<td>A.6</td>
<td>Data structure of the 2 levels HPA.</td>
<td>73</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Compression ratios obtained adopting different strategies for the remainders. .......................................................... 24
3.2 Number of coefficients exceeding the coding range after applying histogram balancing. ........................................... 26
Chapter 1

Introduction

One of the consequences in the current society of technological improvements is the increase in the amount of information transferred every day in every aspect of our life. It occurs from the personal images shared through social networks to the terabytes of data processed in scientific research. In order to ease the storage and the transfer of such a vast information, many different types of algorithms and standards for data compression have been developed, and more are under development at the time of writing this report.

1.1 Different kind of compression techniques

Data compression algorithms can be divided into different classes according to their characteristics. Specifically, we can classify them in two classes. The first of these classes is that of lossy methods. These methods transform the data in a way that implies an intentional and acceptable data loss (usually bounded by the user). The second one is that of lossless methods, in which the data integrity is maintained and can be completely recovered through a reversible process. The technique used normally depends on the type of application and the type of data. For example, in the case of images, video and audio applications the lossy technique is often preferred, since they involve a huge volume of data (and thus much higher compression ratios are required) while some level of quality loss can even be hardly noticeable, provided that the reconstructed data is close enough to the original. On the other hand, for science data, documents, and computer files in general, it is typically mandatory to recover the original data without any loss. In the case of scientific projects it also means taking advantage of the high precision that modern instruments can offer. Hence, a lossless approach is typically used.
1.2 Data compression in Space

Space exploration begun in the decade of the sixties and, since then, thousands of artificial satellites or probes have been sent around our Planet and throughout the Solar system. Every space mission is launched after many years of careful analysis and design. Normally, each space mission has stricts requirements in terms of available bandwidth for the data transmission down to Earth, available time for such transfer, and available computing resources onboard. Each of these resources is very expensive, so after these studies, compromise solutions are reached in order to maintain a good quality of the data produced while assuring a relatively small amount of data that needs to be sent.

A good compression algorithm is needed to reduce the cost of the mission or, alternatively, to improve the quality of the data or service delivered. Having less data volume to transfer means that either less bandwidth is needed or that the time required for the downlink gets shorter, thus helping the scheduling of the operations from the ground segment.

One of the most challenging space missions nowadays is Gaia, which was launched December the 20th, 2013. Gaia is the most ambitious astrometric space mission currently envisaged [1]. It was adopted within the scientific programme of the European Space Agency (ESA) in October 2000, and unanimously approved by its Science Programme Committee (SPC) in February 2006. It aims to measure the positions and proper motions of an extremely large number of stars and other types of objects with unprecedented accuracy, complemented with high-resolution photometry and medium-resolution spectroscopy. As a result, the most complete and accurate three-dimensional map of our Galaxy will be obtained, also including Solar System objects and extragalactic sources. This space observatory will be a technological challenge in all its aspects, from its instrumentation and onboard data handling to the on-ground data processing and analysis.

Gaia obviously needs an excellent compression algorithm on board. A Technological Research Programme of ESA, called GOCA (Gaia Optimum Compression Algorithm), led to the design and implementation of a new compression algorithm called PEC (Prediction Error Coder). PEC offers better results than the current standard for lossless data compression in space (CCSDS 121.0), which was developed in the mid nineties. CCSDS 121.0 is based on the Rice coder method, which is concieved for noiseless data following geometric distributions. Its performance rapidly degrades in the presence of outliers. PEC, on the contrary, is much more resilient in front of unexpected data, providing good compression efficiencies under almost any situation. An adaptive layer was implemented later, which brought us FAPEC, the Fully Adaptive Prediction Error Coder [2]. In this work we elaborate on a new pre-processing stage for FAPEC specifically designed for lossless or lossy image compression.
1.2.1 The HPA algorithm

FAPEC requires an adequate pre-processing stage decorrelating the input data and generating prediction errors with a lower entropy level. By default, FAPEC just provides basic pre-processing stages such as delta pre-processing, that is, outputting the differences between consecutive values. Although this approach already provides good results, the ratios obtained can be further improved if advantage of the inter-pixel correlations typically present in an image is taken into account.

Another functionality that would make FAPEC more versatile is lossy image compression, specially if different quality levels can be selected by the user. Even beyond that, we are interested in an algorithm that could provide a progressive image decompression from a given image file. That is, allowing to decompress the same image file at different quality levels depending on the needs of the user. This is something that we continuously experience nowadays when browsing images in a smartphone, or when browsing many high-resolution pictures in a computer and wish to see some picture in its full resolution. Providing such an “in-built thumbnail” capability would obviously be extremely interesting for the current information technologies.

To provide FAPEC with this new functionality, in this work we have developed a new algorithm called Hierarchical Pixel Averaging (HPA). The HPA algorithm analyzes hierarchically the image into small areas (or blocks) of just $16 \times 16$ pixels which, in turn, are subdivided into smaller blocks. After calculating the average values at each level, differential coefficients between the average of contiguous levels are extracted which substitute the original pixel values. This approach, avoiding any wavelet or transformation algorithm, should also avoid the typical “artifacts” that can be seen in typical lossy compression algorithms (such as JPEG) when the quality is largely reduced. The particularity of the differential coefficients is that they have a lower entropy than the original pixels. Given the fact that the compression ratio achievable with FAPEC depends on the entropy of the input data, such a pre-processing stage can improve the overall ratios. The decoding (or post-processing) process must obviously be considered as well, which just has to invert these operations and will be used to assess the correct reconstruction of the images.

The whole HPA algorithm, together with some routines for image processing, will be implemented in C language for a Linux platform. Some of the features implemented will be described throughout this report.

1.3 Organization of this work

This report is structured in such a way that it covers the timeline of the development of the HPA algorithm. Chapter 2 describes the logic behind the HPA algorithm, illustrating the basic $16 \times 16$ pixels block in which the image is decomposed and from which the average and differential coefficients are calculated. The description
of the implementation of the coding algorithm is discussed in chapter 3. In chapter 4 we explain the inverse algorithm, how it differs from the coding algorithm, and how it was implemented. Chapter 5 presents the results, including the compression ratios achieved and the execution times, compared to the plain FAPEC compression. The tests performed also allow to simulate the whole flow of compression-decompression, up to the reconstruction of the original image. Finally, in the last chapter we draw our conclusions and describe some possibilities for future work that can be done on the HPA algorithm to exploit its full potential.
Chapter 2
The Hierarchical Pixel Averaging algorithm

In this chapter we present the concept of HPA (Hierarchical Pixel Averaging), an image data processing algorithm that has been designed as a pre-processing stage for FAPEC [3] or other data compressors [4, 5, 6, 7]. The goal is twofold: on one hand, we intend to achieve better compression ratios after the application of this algorithm. On the other hand, its design should allow to progressively introduce controlled losses in the compression, further increasing the compression ratios at the expense of a reasonable quality loss in the reconstructed image. In order to allow the usage of HPA in space missions, the computation complexity of HPA has been kept to a minimum, and its operation is efficient with small image blocks [8].

2.1 Subdivision of 16×16 pixel blocks

One of the most typical approaches in image data compression is to transform the image by means of some kind of mathematical formulation [9], such as the discrete wavelet transform (DWT) — which was used used in Refs. [10] and [11] — or the cosine transform — which is used in the JPEG standard [9]. HPA, on the other hand, avoids such complex transformations, and just applies simple operations on the space and brightness domains of the image — mainly just doing averages. By remaining in the space domain we intend to avoid the artifacts that can be seen in compressors such as JPEG when reducing too much the quality in lossy compression.

HPA works in small image blocks of just 16×16 pixels. Let us consider one of such square portions and let us give them a number, from 0 to 255, as shown in Fig. 2.1. Let us call the entire block a Level-4 block. It can be divided into 4 smaller areas, each one with an 8-pixels side, we call them Level-3 blocks. In the same way, we can define Level-2 blocks (with 4-pixels sides) and Level-1 blocks (with 2-pixels sides). Finally, the Level-0 consists of the 256 individual pixels. Summarizing, for
each 256 pixels block, we have defined the following hierarchy:

- 256 level 0 pixels
- 64 level 1 blocks, each made of 2×2 pixels
- 16 level 2 block, each made of 4×4 pixels
- 4 level 3 block, each made of 8×8 pixels
- 1 level 4 block, made of 16×16 pixels

A consequence of such a hierarchy is that each block of level \(n+1\) is composed of 4 blocks of level \(n\). Another characteristic is that the average value of the \(n+1\) level block is equal to the mean of the average values of the 4 sub-blocks of level \(n\). This relation holds for the four upper levels. For instance, for level 1 blocks, the average value is equal to the mean of the single four pixels that belongs to it. We can describe these relations as follows:

\[
\overline{H_{n+1}(A)} = \frac{H_n(B) + H_n(C) + H_n(D) + H_n(E)}{4} \tag{2.1}
\]

where \(H_{n+1}(A)\) is the average of block A from level \((n+1)\), \(H_n(B)\) is the average of the block B from level \(n\). For the last level, the following relation holds:

\[
\overline{H_1(A)} = \frac{H_0(B) + H_0(C) + H_0(D) + H_0(E)}{4} \tag{2.2}
\]

where \(H_1(A)\) is the average of block A from level 1, and the four \(H_0\) values are the individual pixels in that level 1 block. Summarizing, in each block of 16×16 pixels the following values exist:

Figure 2.1: HPA block, formed by 16×16 pixels.
• Level 0: 256 individual pixels.

• Level 1: 64 $H_1$ coefficients (sometimes indicated as $lev_1Avg$), each being the average of a 2×2 pixels block — see Eq. (2.2).

• Level 2: 16 $H_2$ coefficients, each being the average of a 4×4 pixels block. It can be calculated from the 4 $H_1$ coefficients contained in it, as indicated in Eq. (2.1).

• Level 3: 4 $H_3$ coefficients, each being the average of an 8×8 pixels block. It can be calculated from the 4 $H_2$ coefficients contained in it — using Eq. (2.1).

• Level 4: 1 $H_4$ coefficient, average of the complete 16×16 pixels block. It can be calculated from the 4 $H_3$ coefficients contained in it — using again Eq. (2.1).

Note that, conceptually speaking, all these averages must be done considering the necessary decimals. We will describe hereafter how to implement it in practice, avoiding any floating-point operation or decimal value.

### 2.2 Differential values

So far we have seen how a 16×16 pixels block is hierarchically divided into smaller ones with decreasing size (i.e., 8×8, 4×4, 2×2, and finally reaching the individual pixels). The basic concept behind the HPA algorithm is to extract differential coefficients from the average values and to transmit them instead of the original pixels. As it will be explained later, using this approach a reduced entropy is obtained, due to the narrower range of possible values that the differential coefficients can take when compared to the original values. In the end, that will obviously lead to better compression ratios. The relation between the differential coefficient and its original value is the following:

$$\Delta H_n(i) = H_n(i) - H_{n+1}(A)$$ \hspace{1cm} (2.3)

where $H_n(i)$ and $\Delta H_n(i)$ are the average and differential coefficients of any given block $i$ from level $n$, respectively, and $H_{n+1}(A)$ is the average coefficient of their parent block $A$ from level $n + 1$. For each block of 16×16 pixels, the following coefficients are calculated:

• Level 0: 256 $\Delta H_0$, calculated by subtracting, to each individual pixel, the $H_1$ value of its 2×2 parent block;

• Level 1: 64 $\Delta H_1$, calculated by subtracting, to each $H_1$, the $H_2$ value of its 4×4 parent block;

• Level 2: 16 $\Delta H_2$, calculated by subtracting, to each $H_2$, the $H_3$ value of its 8×8 parent block;
• Level 3: $4 \Delta H_3$, calculated by subtracting, to each $H_3$, the $H_4$ value;

• Level 4: $1 H_4$. This remains unaltered.

We can now calculate the number of coefficients that are needed in order to reconstruct each block of $16 \times 16$ pixels. As previously explained, for each block we now have $256 \Delta H_0$, $64 \Delta H_1$, $16 \Delta H_2$, $4 \Delta H_3$ and $1 H_4$ coefficients. The total number of coefficients is now 341, which is 33% larger than the original number of pixels (256). Nevertheless, the actual number of coefficients to be coded can be reduced, as explained below.

### 2.3 HPA coefficients and remainders

At first, it may seem that using the HPA algorithm actually increases the amount of data. To overcome this problem let us consider the detailed definition of an average value. As previously shown, the following relation exists:

$$H_{n+1}(A) = \frac{H_n(B) + H_n(C) + H_n(D) + H_n(E)}{4} \tag{2.4}$$

Let us define $H_n(A)$ as the sum of all the pixels contained in block $A$ of level $n$, and $\rho_n(A)$ as the remainder of its division by 4 (that is, the remainder of $H_n(A)/4$. It allows redefining the equation as follows:

$$H_n(B) + H_n(C) + H_n(D) + H_n(E) = 4 \times H_{n+1}(A) + \rho_{n+1}(A) \tag{2.5}$$

From this equation, it is evident that one of the four children coefficients can actually be calculated from its 3 brothers, combined with its parent information. Therefore, it is enough to code only three out of the four values for the lower level — that is, $H_n(B)$, $H_n(C)$ and $H_n(D)$ — and the fourth one can be computed as follows:

$$H_n(E) = 4 \times H_{n+1}(A) + \rho_{n+1}(A) - H_n(B) - H_n(C) - H_n(D) \tag{2.6}$$

At first sight it may seem that we have not actually reduced the number of coefficients to code, since we have just replaced $H_n(E)$ by $\rho_{n+1}(A)$. However, as it will be shown below, the $\rho$ remainders can be coded in a much more efficient manner than the $H_n$ coefficients.

Following the procedure described by Eq. (2.6), the number of $H$ coefficients for each $16 \times 16$ block can be reduced to the following values:

• Level 0: 192 $\Delta H_0$

• Level 1: 48 $\Delta H_1$

• Level 2: 12 $\Delta H_2$
2.4 Overhead of the HPA algorithm

The HPA algorithm converts the original pixels into a set of coefficients that, typically, will take smaller values. The cost of this operation is a coding overhead, caused by the previously introduced remainders ($\rho$), coming from the computations of all the average values. Considering that they all come from division by 4, each remainder can be coded in just 2 bits. The detailed implementation of the division will be explained in depth in the next chapter. For now, we just need to know the actual overhead caused by these remainders:

- Level 1: 64 $\rho$ remainders, that is, 128 bits. They can be stored in 16 bytes.
- Level 2: 16 $\rho$ remainders, that is, 32 bits. They can be stored in 4 bytes.
- Level 3: 4 $\rho$ remainders, that is, 8 bits. They can be stored in 1 byte.
- Level 4: 1 $\rho$ remainder, that is, 2 bits.

The total amount of data is 21 bytes + 2 bits. This represents an overhead of 0.66 bits per pixel. In the next chapter we will present different studies done for the optimum processing of the $\rho$ remainders and their effect on the overall entropy and compression performance.

2.5 Limitations of HPA

All the analyses presented so far are based on the assumption that an image can be divided in an integer number of square blocks of 16 pixels side. This is an important limitation of the HPA algorithm, because it initially prevents the application of the HPA to images which have some dimension not multiple of 16 pixels. On the other hand, the typical size of the sensors in normal cameras or space telescopes is often a multiple of 16. That is the reason why, in this first implementation, we decided to accept the limitation on the image size. It will be certainly one of the first improvements to be done in future versions, to allow applying the HPA algorithm to any image of any size.
Chapter 3

HPA pre-processing algorithm

This chapter explains in detail the image processing that has to be done before sending the data to the FAPEC coder. Some pre-processing is necessary before the HPA algorithm itself is used to generate a sequence of HPA coefficients that are ultimately sent to the compressor.

As explained in the previous chapter, the HPA algorithm is applied to a part of the image consisting of a block of 16 by 16 pixels. The dimensions of an image are generally of the order of several hundreds (even thousands) of pixels and the entire image can contain millions of pixels. Hence, before applying the HPA algorithm, some preparatory steps must be followed:

- Dividing the image into a sequence of blocks.
- Applying the HPA algorithm to each block and store the output coefficients.
- Eventually, reorganize the HPA coefficients of the entire image.
- Send all the HPA coefficients to FAPEC.

In this chapter we will describe each of these functions and we will present some details of the developed code that implements them.

3.1 HPA algorithm enabling

The starting point of this work is the existing C code that implements FAPEC®. The goal is to extend such code, including the HPA pre-processing option for image data. The first modification required for this is the addition of the command-line parameters which enable the image processing through the HPA algorithm. We have called the first parameter image. When it is set to 1, the functions for the pre-processing and the calculation of the HPA coefficients are enabled. Together with it, also the image dimensions must be given as input to FAPEC. They are stored in
3.2 Algorithm for pixel rearrangement

An image is a bi-dimensional data structure. Nevertheless, in general the raw data coming from a sensor is stored in a one-dimensional array. Throughout this work we consider the raw data as an input file. To give an example, an image that has the dimension of 32×32 pixels is normally stored in a one-dimensional array of 1024 pixels. The pixels come ordered according to their position in the image: first, the ones that belong to the first row of the image, then, the ones from the second row, and so on. Fig. 3.1 shows a representation of this arrangement.

Thus, before using HPA the pixels have to be rearranged in 16×16 blocks. In this way the input sequence is prepared to be sent to the HPA: first, we will have the pixels that belong to the first block, then the ones that belong to the second, and so on. The rearrangement is done using a BlockCreation function, which is the first function that is called when the image processing features are enabled. As previously stated, the image dimensions are stored in the imageWidth and imageHeight parameters. The first operation to be done is the calculation of the dimension of the image in terms of 16×16 pixels blocks. To do so, we simply divide the original dimensions by 16 and obtain the number of horizontal and vertical blocks. In particular we have the following:

```plaintext
nHorBlocks = imageWidth / 16;
nVerBlocks = imageHeight / 16;
```

The total number of blocks is then calculated as:
nBlocks = nHorBlocks * nVerBlocks;

Each block occupies a specific position within the image. The generic i-th block is identified by calculating its pseudo-coordinates as shown in Fig. 3.2:

\[
iBlockRow = \text{ceil} \left( \frac{i}{nHorBlocks} \right);
iBlockColumn = i \mod nHorBlocks;
\]

if (iBlockColumn == 0) iBlockColumn = 4;

for (iBlock=1;iBlock<=nBlocks;iBlock++)
{
  // First pixel index calculation... two parts
  // Last pixel of the previous big line (i_BlockRow-1)
  nPixel = (iBlockRow-1)*nHorBlocks*256;
  for (iPixelRowWithinBLock=1;iPixelRowWithinBLock<=16;iPixelRowWithinBLock++)
  {
    // Calculation of the first pixel of the current line (iPixelRowWithinBLock)
    int fPixel = nPixel + (iPixelRowWithinBLock-1)*nHorBlocks*16 + (iBlockColumn-1)*16;
    for (iPixelColumnWithinBLock = 0; iPixelColumnWithinBLock< 16; iPixelColumnWithinBLock++)
    {
      // Get sample instructions
      absPixelPos = iPixel + iPixelColumnWithinBLock;
      imageSample = MI_Get32(PD->mi, SYM_SIZE);
      // Store sample instructions - alternatives:
      MI_IncreasePointer(PD->mi, absPixelPos*SYM_SIZE/8);
      // Restore PD->mi pointer
      MI_RestoreState(PD->mi);
    }
  }
}

The code shown in Fig. 3.3 implements the pixel identification. The first part is used to identify the position of the i-th block, as explained before, while the second is used to find the pixels that belong to each row of the block. At the end of this
phase, the pixels are stored in a different memory interface (essentially, a memory buffer) and are ordered according to the block they belong to and, within the block, to the row they are located in.

3.3 The HPA blocks processing

So far, we have seen how the pixels are reordered on a block-basis. As explained in the previous chapter, for each block, the following coefficients are generated:

- $192 \Delta H_0$
- $48 \Delta H_1$
- $12 \Delta H_2$
- $3 \Delta H_3$
- $1 \overline{H}_4$
- $85 \rho$ remainders, divided among the four levels.

The second part of the code developed in this work processes the image at a block-level. It provides each block to the HPA and, afterwards, collects and reorganizes the output coefficients. The function that executes these functionalities is called HPAArrayProcessing. Fig. 3.4 shows the flow diagram of the implemented code.

3.3.1 Calculation of the HPA coefficients

Inside the HPAArrayProcessing function, each $16 \times 16$ block is passed as input to another function, called HPAValuesCalculation, in which the actual calculation of the HPA coefficients is done. As previously stated, the 256 pixels compose a $16 \times 16$ block, and are numbered as in Fig. 2.1. Looking carefully at the level 1 blocks, it can be seen that they are composed of four pixels at the following positions: $h, h + 1, h + 16$ and $h + 17$, where $h$ is the position of the upper left pixel of the block. In a similar way, given a level 2 block, the first pixels of their associated level 1 blocks have the following coordinates: $i, i + 2, i + 32$ and $i + 34$. The same consideration
3.3 The HPA blocks processing

can be done for the first pixels of the associated level 2 blocks \((j, j + 4, j + 64, j + 68)\) and for the level 3 blocks \((k, k + 8, k + 128, k + 136)\). Since these offsets remain constant, we decided to create four arrays with Look-Up Tables (LUT), which allow a fast and correct identification of the pixels. They are the following:

- \(\text{int } \text{hOffset}[ ] = \{0, 1, 16, 17\}\), with the offsets for the \(2 \times 2\) blocks.
- \(\text{int } \text{iOffset}[ ] = \{0, 2, 32, 34\}\), with the offsets for the \(4 \times 4\) blocks.
- \(\text{int } \text{jOffset}[ ] = \{0, 4, 64, 68\}\), with the offsets for the \(8 \times 8\) blocks.
- \(\text{int } \text{kOffset}[ ] = \{0, 8, 128, 136\}\), with the offsets for the \(16 \times 16\) block.

![Figure 3.5: HPA algorithm flow diagram.](image)

Fig. 3.5 shows the flow diagram for the calculation of the HPA coefficients. There are four iteration levels, each one nested in the previous one. First, the pixels at level 0 are summed to calculate the \(\text{Lev0Sum}\) (that is, \(H_0\)). This value is then used to calculate the \(\text{Lev1Avg}\) \((\overline{H_1})\) and \(\text{Lev1Rem}\) \((\rho_1)\). Successively, \(\text{Lev1Avg}\) is used to compute the \(\text{Lev0Dif}\) \((\Delta H_0)\) using the code that implements Eq. (2.3):

\[
\text{Lev0Dif} = \text{Lev0Avg} - \text{Lev1Avg}
\]  

(3.1)

The same procedure is repeated for each level. Fig. 3.6 shows the part of the developed code that processes a \(16 \times 16\) block to extract its HPA Coefficients. The instructions to calculate the \(\text{Lev#Dif}\) are inside the \text{HistogramBalancing} function, which will be explained later in this chapter.

3.3.2 Coefficient interleaving

An entropy coder (FAPEC, in our case) performs best when receiving values with uniform statistics. To achieve this, the HPA coefficients must be output doing an
for (kIndex = 0; kIndex < 4; kIndex++)
    {
        for (jIndex = 0; jIndex < 4; jIndex++)
            {
                for (iIndex = 0; iIndex < 4; iIndex++)
                    {
                        for (hIndex = 0; hIndex < 4; hIndex++)
                            {
                                inPixelPos = kOffset[kIndex] + jOffset[jIndex] + iOffset[iIndex] + hOffset[hIndex];
                                MI_IncreasePointer(mInOut, inPixelPos*SYM_SIZE/8);
                                imageSample = MI_Get32(mInOut, SYM_SIZE);
                                MI_RestoreState(mInOut);
                                lev1sum[i] += imageSample;
                                lev0diftemp[hIndex] = imageSample;
                                outPixelPos++;
                            }
                (*lev1rem)[(hpaArrayPos*64) + i] = lev1sum[i] & 0x03;
                (*lev1avg)[(hpaArrayPos*64) + i] = lev1sum[i] >> 2;
                lev1avgtemp[i] = lev1sum[i] >> 2;
                // Histogram Balancing function - lev 0 dif
                HistogramBalancing((*lev1avg)[(hpaArrayPos*64) + i], lev0dif, lev0diftemp,
                                    (hpaArrayPos*256) + outPixelPos -4, 0, 4);
                lev2sum[j] += (*lev1avg)[(hpaArrayPos*64) + i];
                i++;
            }
        (*lev2rem)[(hpaArrayPos*16) + j] = lev2sum[j] & 0x03;
        (*lev2avg)[(hpaArrayPos*16) + j] = lev2sum[j] >> 2;
        lev2avgtemp[j] = lev2sum[j] >> 2;
        // Histogram Balancing function - lev 1 dif
        HistogramBalancing((*lev2avg)[(hpaArrayPos*16) + j], lev1dif, lev1avgtemp,
                            (hpaArrayPos*256), i - 4, 4);
        lev3sum[k] += (*lev2avg)[(hpaArrayPos*16) + j];
        j++;
    }

Figure 3.6: Code that implements the HPA coefficients calculation.

interleaving. That is, the coefficients of the same level are put together, starting from
the higher level down to LevODif. At the same time, considering the limitations of
some communications channels (such as in space communications) but also to reduce
the latency in the image compression, the image should be compressed in blocks —
not the complete image at once. For example, we may compress the images in
blocks covering all its width and 16 pixels high, that is, a row of HPA blocks, and
then output the resulting HPA coefficients interleaved, as previously said. In this
work, the array containing one of these image blocks containing a given number of
HPA blocks (each with 16 × 16 pixels) will be called HPAArray. In general, its size
will typically be different (smaller) from that of the original image. To make the
code more versatile, we decided to define its size using the arraySize parameter.

In order to find the best value, we have compared the overall compression ratio
achieved with different values for the size of the HPAArray, for a single HPA block, a
“line” of HPA blocks (covering all the image width), two “lines” of HPA blocks, and
finally the entire image. Fig. 3.7 shows the compression ratio achieved, normalized
to the best result found. The results show that, except for some particular files,
the compression ratio normally increases with the size of HPAArray. An interesting
result, and probably the best compromise, is found when the size of the HPA Array is equal to \( n \) blocks, that is, a complete “line” of 16×16 HPA blocks. It offers good enough results (less than 1% below the best case), and gives more versatility and reliability to the compression. Furthermore, the procedure of processing a whole line of an image can be easily applied to pushbroom sensors, like the ones found in some Earth observation satellites.

![Figure 3.7: Compression ratios for different sizes of HPAArray.](image)

Fig. 3.8 shows the code used to handle the \texttt{HPAArray} and to get the HPA coefficients from it. Note that some lines have been removed for the sake of clarity and conciseness. The first part initializes the dynamic arrays used to store the HPA coefficients. They are sized according to the size of the \texttt{HPAArray}. After that, the number of arrays in which the image is divided is calculated and, for each of these arrays, the blocks that belong to it are identified. Finally, the commands that call the HPA algorithm for each block and save the output coefficients in the dynamic arrays can be found. These arrays are:

- \texttt{lev0dif}, \texttt{lev1dif}, \texttt{lev2dif} and \texttt{lev3dif}, used for the differential coefficients;
- \texttt{lev1rem}, \texttt{lev2rem}, \texttt{lev3rem} and \texttt{lev4rem}, used for the remainders;
- \texttt{lev4avg}, \texttt{lev1avg}, \texttt{lev2avg} and \texttt{lev3avg}, used for the average coefficients.
3.3.3 Writing the HPA coefficients

This is the easiest part of the developed code: the function which copies the HPA coefficients into the memory interface that will be given as input to FAPEC. We remind that only three out of the four average and differential coefficients per HPA block are copied. At the beginning of the `HPAValuesCopying` function there is a call to the `reduceBuf` function, being its aim the goal to remove the third out of every four elements. The code that removes the coefficients is shown in Fig. 3.9.

The `HPAValuesCopying` function takes as input the dynamic arrays and the size of the HPA array. It then calculates the number of coefficients that must be copied for each level and type. Fig. 3.10 shows the part of the code that copies the `lev3dif`
3.4 Histogram Balancing algorithm

In order to reduce further the entropy of the HPA coefficients we have developed a new algorithm, called *Histogram Balancing*. Let us consider the following situation.

```c
for (i = 0; i < length; i++)
{
    if ((i+1)%4 != 1)
    {
        tempBuf[p] = (*buf)[i];
        p++;
    }
}
```

Figure 3.9: Code that implements the buffer reduction algorithm.

coefficients into the memory interface.

```c
// element from 1 to 3 - LEVEL 3 diff
for (b = 0; b < 3*arraySize; b++)
{
    MI_Put(mOut, (*lev3dif)[b], SYM_SIZE);
}
```

Figure 3.10: Code that implements the HPA values copying algorithm.

3.3.4 Writing the remainders

As explained in the previous chapter, the $\rho$ remainders (2 bits each) are put together in sets of 4 to compose a byte. The procedure is a simple left shift, followed by a logic “OR”. From level 0 up to level 3, the number of $\rho$ remainders is a multiple of 4, and hence the number of iterations is easily calculated. This is not always true for the Lev4Rem, the number of which cannot be known a priori. To overcome this, we count the number of Lev4Rem at the beginning of the HPAValuesCopying. Fig. 3.11 shows the calculation of the number of Lev4Rem remainders and the part that combines them together in bytes. We have tested two different configurations:

- The $\rho$ remainders are sent to FAPEC together with the HPA coefficients.
- The $\rho$ remainders are put into a different memory interface, bypassing FAPEC.

We have done some tests using different files. Table 3.1 shows the comparison of the final compression ratio in the two cases. As can be seen, putting the $\rho$ remainders in a separated memory interface improves the compression ratio in the majority of the cases. Hence, we have decided to leave the $\rho$ remainders outside FAPEC and just copy them, after the compression, at the end of the output file.
// calculation of the number of bytes to be used for lev4rem
int nByte = ceil((float) arraySize / 4);
// calculation of the number of rem for the last byte
int nRem = arraySize % 4;
if (nRem == 0)
{
    nRem = 4;
}
for (b = 0; b < nByte; b++)
{
    if (b != (nByte-1) )
    {
        char elem1 = (*((lev4rem)[b*4+3] << 6) | (*((lev4rem)[b*4+2] << 4) |
                        (*((lev4rem)[b*4+1] << 2) | (*((lev4rem)[b*4] & 0x03);
        MI_Put(mRem, elem1, 8);
    } // else it's the last byte. the count should be fine, because the last rems
    // starts from 4b until 4b+nRem (both included)
    else
    {
        char elem1 = (*((lev4rem)[b*4] & 0x03;
        for (c = 1; c < nRem; c++)
        {
            elem1 = elem1 << 2;
            elem1 = elem1 | (*((lev4rem)[b*4+c;]
        }
        MI_Put(mRem, elem1, 8);
    }
}

Figure 3.11: Code that implements the interleaving and copying algorithm.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Performance</th>
<th>(\rho) bypassing FAPEC</th>
<th>(\rho) within FAPEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>banyoles-1024x592.raw</td>
<td>1.2842</td>
<td>1.2733</td>
<td></td>
</tr>
<tr>
<td>field-1024x592.raw</td>
<td>1.2323</td>
<td>1.2223</td>
<td></td>
</tr>
<tr>
<td>mosaic10.raw</td>
<td>2.3688</td>
<td>2.4369</td>
<td></td>
</tr>
<tr>
<td>eixample-1024x592.raw</td>
<td>1.1341</td>
<td>1.1256</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Compression ratios obtained adopting different strategies for the remainders.

We have to code the difference between a pixel \(P\) and a reference level \(R\). Let us call \(D\) this difference. Depending on the value of \(R\), \(D\) will be limited in range and will not cover all the available dynamic range (e.g., +/- 128 in the case of 8 bits). More specifically, if we call \(m\) and \(M\) the minimum and maximum values that the pixels can assume (i.e., \(m = 0\) and \(M = 255\) for an 8-bit image), then the maximum absolute value that the difference can have in both signs (which we will call \(T\)) is given by the following equation:

\[
T = \min(R - m, M - R)
\]  

(3.2)
This means that there will be other values of $D$ for which its histogram will be unbalanced on one sign. In order to balance the values that $D$ can take (and thus obtain “balanced” coded values, which we call $C$), we can process the difference as follows. If $\text{abs}(D) \leq T$, then we leave $D$ unaltered and thus $C = D$. Else, we proceed as follows. For $C = (T + 1) + (\left( (D - T) \gg 1 \right)$, if $((D - T) & 1)$, then $C = -C$.

In this way we reduce the maximum absolute value that the difference $D$ can reach. Fig. 3.12 shows the histograms of the lev0dif coefficients for one example file (example-1024x592.raw), as obtained from HPA (left panel) and after the applying histogram balancing procedure (right panel). Although it is difficult to see the difference at first sight, we have reduced the entropy of these coefficients from 5.30 bits/sample to 5.27 bits/sample.

Throughout our software, the histogram balancing algorithm is called several times and, each time, the number of coefficients that needs to be processed is different. For this reason we decided to design the function so that it takes as input parameter the number of coefficients to process. We called this parameter elCount. Moreover, the algorithm takes as input the array with the coefficients to process, and another array in which the processed differential coefficients will be put. We called these two arrays lowLevAvg and lowLevDif respectively. Another input is the reference level, which is given in the refAvg parameter. The last two input coefficients are two offsets which are used to identify the correct position of the elements to be processed. These offsets are:

- offset1, which is used to identify the position, within the output array, in which the values will be stored. This is needed because the lowLevDif array may contain the coefficients of an HPAArray, hence belonging to many blocks.
- offset2, which is used for the input and output arrays, because we may need to process only some of the coefficients in a HPA block. An example is the processing of the Lev2Dif: they are 16 for each block, but when we apply the
// Threshold identification
if ( (refAvg-m) < (M-refAvg) )
    threshold = refAvg-m;
else
    threshold = M-refAvg;

// loop through the lowLevAvg array
for (i = 0; i < elCount ; i++)
{
    // calculation of the difference
    dif = lowLevAvg[offset2 + i] - refAvg;
    // comparing the difference with the threshold
    if (abs(dif) <= threshold)
        (*lowLevDif)[offset1 + offset2 + i] = dif;
    else
    {
        tempValue = (threshold + 1) + ((abs(dif) - threshold) >> 1);
        if ( (abs(dif) - threshold) & 1)
            tempValue = -tempValue;
        (*lowLevDif)[offset1 + offset2 + i] = tempValue;
    }
}

Figure 3.13: Code that implements the histogram balancing algorithm.

Histogram Balancing, we apply it only to 4 of them at the time; hence we must give the starting position to the function.

As an example, we can consider the following instruction:

HistoBalancing(lev2avg[j], lev1dif, lev1avg, (hpaArrayPos*64), i-4, 4)

In this part of code, the elements to be processed belong to the lev1avg array. They start from index i - 4 and are 4. They will be put in the lev1dif array starting from position hpaArrayPos * 64 + i - 4. A similar instruction is used in the code to process the lev0dif, lev2dif and the lev3dif. Fig. 3.13 shows a relevant part of the code of the histogram balancing function.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Lev3Dif</th>
<th>Lev2Dif</th>
<th>Lev1Dif</th>
<th>Lev0Dif</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mosaic10.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>banyoles-1024x592.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>catedral-1024x592.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>eixample-1024x592.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>field-1024x592.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pirineus-1024x592.raw</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2: Number of coefficients exceeding the coding range after applying histogram balancing.
3.5 Sign bit processing

void HPAValuesCalculation (...) {
    // HPA parameters calculation
    // outer loop, loop 4
    for (kIndex = 0; kIndex < 4; kIndex++) {
        // HPA Sign Bit detection algorithm
        HPASignBitDetector (lev0dif, byteHPASignBitSelector, hpaArrayPos);
    }
}

void HPASignBitDetector (int *lev0dif[], int *byteHPASignBitSelector[], int arrayPos)
{
    int i, lowerBound, upperBound;
    // Definition of the upper and lower boundaries
    lowerBound = - (int) pow(2, SYM_SIZE - 1);
    upperBound = (int) pow(2, SYM_SIZE - 1) - 1;
    // check of the lev0dif
    for (i=0; i<256; i++) {
        // control of the element with the permitted range
        if ( ( (*lev0dif)[arrayPos*256 + i] < lowerBound ) ||
            ( (*lev0dif)[arrayPos*256 + i] > upperBound ) )
        {
            // when it founds an element that is beyond the permitted range,
            // it sets the flag to 1 and returns
            (*byteHPASignBitSelector)[arrayPos] = (*byteHPASignBitSelector)[arrayPos] | 1;
            return;
        }
    }
}

Figure 3.14: Code that calls the Sign Bit Detection algorithm.

We have already commented on how the histogram balancing algorithm improves the performance of HPA by reducing the entropy of the differential coefficients. Nevertheless, there is a drawback in using this algorithm, which is related to the notation used in the coding. Given the fact that the differential coefficients can adopt values either positive or negative, we decided to use the “two’s complement” notation. Namely, given a symbol size (SYM_SIZE) of 8 bits, they can take values in the range between $-128$ and $+127$. Using the histogram balancing algorithm a possibility exists that a coefficient adopts the value $2^{SYM\_SIZE-1}$ (that is, $+128$ for 8-bit images or $+32768$ for 16-bit images). We performed some tests with different images to check if this really happens, and if so, how many times occur. The results of this test are shown in Table 3.2.

The results reveal that the only coefficients affected by this problem are the Lev0Dif ones. In particular, the problematic values taken were $+128$, as they are
void HPASignBitCalculation(int *byteHPASignBitSelector[], int *curCounterHPASignBit, int arraySize, int *lev0 dif[], int *signBitArray[])
{
    int i;
    // loop through the element of byteHPASignBitSelector
    for (i = 0; i < arraySize; i++)
    {
        // If it’s 1, then I need to extract the bytes
        if ( (*byteHPASignBitSelector)[i] == 1 )
        {
            // function that extract the sign bit from the lev0 dif, at block i, and puts
            // the bits in the signBitArray, at position curCounterHPASignBit
            ExtractHPASignBit(lev0 dif, i, signBitArray, (*curCounterHPASignBit));
            *curCounterHPASignBit = *curCounterHPASignBit + 1;
        }
    }
}

void ExtractHPASignBit(int *lev0 dif[], int arrayPos, int *signBitArray[], int curCounterHPASignBit)
{
    int i, upperBound;
    // definition of the upper boundary, which depends on the value of SYM_SIZE
    upperBound = (int) pow(2, SYM_SIZE - 1);
    for (i = 0; i < 192; i++)
    {
        if ((*lev0 dif)[arrayPos*192 + i] == upperBound)
        {
            (*signBitArray)[curCounterHPASignBit*192 + i] = 1;
        }
    }
}

Figure 3.15: Code that performs the Extract Sign Bit function.

8-bit images. They are really few in number, but obviously fix this issue must be
fixed in order to get a really lossless compressor. However, one possible simple
solution exists. We first note that if these coefficients are to be coded, it would not
be enough to use the same number of bits. Instead, one extra bit is needed with
the two’s complement notation. To solve this problem, we have designed a protocol
to mark which ones of the Lev0Dif coefficients have this special value. It consists
of using a bit (per HPA block) set to 1 to mark a block containing a special value.
For example, a 1024×592 image, which is composed of 64×37 blocks, needs 2368
bits. We put these bits in an array, called byteHPASignBitSelector, whose final
size in bytes is given by the number of 16×16 blocks divided by 8. In the previous
example, the array can be coded into 296 bytes. Once the byteHPASignBitSelector
array is set, another processing is conducted within the blocks, in order to find which
coefficients fall beyond the allowed range. As demonstrated by the previous tests,
these coefficients can adopt only the value $2^{SYM_SIZE-1}$. Hence, only one bit is enough
to identify them. The bits are saved in another array, called signBitArray. The
size of this array depends on the number of blocks that have the bit set to 1 in the byteHPASignBitSelector. For each of these blocks, in theory, 256 bits are needed. Actually, this has been optimized as follows: since only 3 out of 4 Lev0dif coefficients are transmitted, only 192 bits will be needed per block. Hence, the original size, in bits, of the signBitArray is given by the number of blocks that have the bit set to 1 in the byteHPASignBitSelector multiplied by 192. As happened for the byteHPASignBitSelector, after putting together 8 bits in one byte the size can be reduced. The final size in bytes is equal to the number of blocks that have the bit set to 1 in the byteHPASignBitSelector multiplied by 192/8. In the previous example, the tests conducted revealed that 4 Lev0dif coefficients are equal to +128 and they belong to four different 16×16 blocks. Hence, the size of signBitArray is 4×192/8 = 4×24 = 96 bytes. The reduction of the coefficients from 256 to 192 brings a more subtle optimization. That is, if the coefficient that falls beyond the permitted range is the one that is not transmitted, then its bit will not be set to one. This reduces the final entropy and improves the compression ratio achievable with FAPEC.

The byteHPASignBitSelector array is processed in the HPAValuesCalculation function as shown in Fig. 3.14. The structure of the HPASignBitDetector function is also shown there. The signBitArray is calculated in the HPASignBitCalculation function, which is called within the HPAValuesCopying right before writing in the memory interface. Fig. 3.15 shows a part of the code that implements the sign bit calculation.
Chapter 4

Inverse algorithm

In this chapter we describe the algorithm developed for the post-processing in the decoding phase, required for the reconstruction of the original image. The input now is a memory interface which contains the HPA coefficients of the image which are decompressed by the FAPEC decoder. Fig. 4.1 shows the complete flow diagram of the coding-decoding process. As seen in the previous chapter, the coefficients come in a precise order, as shown in Fig. 4.2. In order to process them correctly, the same steps used in the coding phase must be repeated, but in reverse order. Hence, we have the following steps now:

- Read the HPA coefficients (H) and REM remainders ($\rho$).
- Rearrange the HPA coefficients in HPA arrays.
- Apply the inverse HPA algorithm to each block to reconstruct the original pixels.
- Rearrange the pixels to their original position.

Figure 4.1: Flow diagram representing the global HPA image compression and decompression process.
In the image-preprocessing phase, the rem coefficients are put in a separate memory interface. We do this to bypass FAPEC, so these coefficients are copied at the end of the compressed file. During the decoding, first of all, the compressed file is read and divided again into two memory interfaces, one for each type of coefficients.

In the HPA algorithm, for each $16 \times 16$ pixels block, there are 85 $\rho$ remainders, distributed in different levels as follows:

- 1 level 4 $\rho$ remainder, which is stored in a $\text{lev4rem}$ array.
- 4 level 3 $\rho$ remainders, which are stored in a $\text{lev3rem}$ array.
- 16 level 2 $\rho$ remainders, which are stored in a $\text{lev2rem}$ array.
- 64 level 1 $\rho$ remainders, which are stored in a $\text{lev1rem}$ array.

Each one of them has 2 bits, and hence the entire set of 85 coefficients of a single $16 \times 16$ pixels block has 21 bytes plus 2 bits. Once the compressed file is read, the total size of the $\rho$ remainders must be calculated. This can be done by knowing the size of the original image. To give an example, let us consider the Example image, which is made of $1024 \times 592$ pixels, that is, 2368 blocks. It needs $2368 \times 21.25 \text{ bytes} = 50320 \text{ bytes}$ to store them all. The division in the two memory interfaces is done using the REMSplitter function, before FAPEC processes the file. The size of the $\rho$ remainders, previously calculated, is used to derive when the $H$ coefficients end, and when the $\rho$ remainders start. The code shown in Fig. 4.3 is part of the REMSplitter function that implements these actions. On the other hand, the code shown in Fig. 4.4 is an example of how the Lev3Rem remainders are read and stored into the dynamic array.
4.2 The inverse sign bit algorithm

In the previous chapter we discussed why we used two arrays, byteHPASignBitSelector and signBitArray, to account for the Lev0Dif coefficients. We remind that these coefficients reach the value $2^{(SYM\_SIZE) - 1}$. To correctly decode these coefficients, the following actions are performed:

- First, the byteHPASignBitSelector is read and processed. Its elements are checked and the bits equal to 1 are counted in the HPASignBitSelectorCounter function.

- That information is used to calculate the size of the signBitArray to be read from the memory interface. For each bit equal to 1 in byteHPASignBitSelector, there are 24 bytes in the signBitArray, and each bit of those 24 bytes corresponds to a differential coefficient of a 16×16 pixels block.

- The correction of the Lev0Dif coefficients in themselves is done within the InvHPASignBitCalculation function.

```c
// number of bytes needed, for each HPAArray, for the lev4Rem
int nByte = ceil((float) arraySize / 4);
// size of the Rem coefficients (in terms of bytes) for each HPAArray
int hpaArrayRemSize = 21 * arraySize + nByte;
// total size of the rem coefficients, in terms of bytes
int totalHPAArrayRemSize = hpaArrayRemSize * nHPAArrays;
// position of the first rem coefficient
int remPos = fileSize - totalHPAArrayRemSize;
// first part - copy of the coded image in another memInterface
while (i < remPos)
{
    MI_Put(&mAdditional, MI_Get32(mIn, 8), 8);
    ++i;
}
// second part - copy of the rem coefficients in the rem memory interface
while (i < fileSize)
{
    MI_Put(mRem, MI_Get32(mIn, 8), 8);
    ++i;
}
```

Figure 4.3: Code that implements the RemSplitter function.

```c
for (b = 0; b < arraySize; b++)
{
    char elem5 = MI_Get32(mRem, 8);
    for (c = 0; c < 4; c++)
    {
        (*lev3rem)[b*4+c] = (elem5 >> 2*c) & 0x03;
    }
}
```

Figure 4.4: The piece of code that implements the Lev3Rem remainders reading stage.
Fig. 4.5 shows a part of the code which sets the coefficient to the value $2^{(\text{SYM\_SIZE}-1)}$.

```c
void InvHPASignBitCalculation(...) {
  // loop through the element of byteHPASignBitSelector
  for (i = 0; i < arraySize; i++) {
    // If it's 1, then I need to extract the bytes
    if ( (*byteHPASignBitSelector)[i] == 1 ) {
      InvExtractHPASignBit(lev0dif, i, signBitArray, curCounterHPASignBit);
      curCounterHPASignBit++;
    }
  }
}
void InvExtractHPASignBit(...) {
  int i, upperBound = (int) pow(2, SYM\_SIZE - 1);
  for (i = 0; i < 192; i++) {
    if ( (*signBitArray)[curCounterHPASignBit*192 + i] == 1 ) {
      (*lev0dif)[arrayPos*192 + i] = upperBound;
    }
  }
}
```

Figure 4.5: The fragment of code that implements the inverse sign bit algorithm.

### 4.3 The inverse histogram balancing algorithm

In the coding algorithm, histogram balancing is used when computing the differential values at each level, either from the original pixels or from the $\overline{H}$ coefficients. The procedure used to invert it is the following:

- First, the threshold $T$ is calculated as the minimum between the reference level minus $m$ and $M$ minus the reference level, where $m$ and $M$ are, respectively, 0 and the maximum possible value (for example 255 for 8 bit images).

- Then, the coded value $C$ is compared, in absolute value, to the threshold. If the absolute value is smaller, then the differential coefficient is taken as equal to the coded value. Otherwise, it is modified as follows:

$$\Delta H = 2 \times |C|T - 2$$

- After that, the sign of the differential value is checked. If the coded value $C$ is negative, we subtract 1 to $\Delta H$.

- If the reference value is placed in the second half of the range $[m, M]$, the sign of the differential value is changed.
Finally, the original value is calculated by adding the reference level to the differential value.

The piece of code shown in Fig. 4.6 is part of the `InvHistogramBalancing` function which implements the inverse histogram balancing algorithm.

```c
// variable initialization
int m = 0, M = pow(2, SYM_SIZE) - 1, threshold, dif, i, codedValue;
// Threshold identification
if ( (refAvg-m) < (M-refAvg) )
    threshold = refAvg-m;
elsethreshold = M-refAvg;
// loop through the lowLevAvg array
for (i = 0; i < elCount ; i++)
{
    // reading the coded value
    codedValue = lowLevDif[offet1 + offset2 + i];
    // if the abs of the coded value is smaller than the threshold,
    // it is the difference as it is
    if ( abs(codedValue) <= threshold )
        dif = codedValue;
    else
    {
        // calculation of the abs of the dif
        dif = 2*abs(codedValue) - threshold - 2;
        // this is to account for the ref level being in the second half
        if (refAvg > pow(2, SYM_SIZE-1) )
            dif = -dif;
    }
    // reconstruction of the original value
    (*lowLevAvg)[offset2 + i] = dif + refAvg;
}
```

Figure 4.6: The piece of code that implements the inverse histogram balancing algorithm.

As an example of how the function behaves, let us consider the following call:

```c
InvHistogramBalancing(lev2avg[j],&lev1avg,*lev1dif,arrayPos*64,j*4,4);
```

This call will process 4 elements of the `lev1dif` array, starting from position `arrayPos*64` + `j*4`, using `lev2avg[j]` as the reference level, and will store the results in the `lev1avg` array, starting from position `j*4`.

### 4.4 The inverse HPA block array processing

In the coding phase, the coefficients are interleaved in a number \( n \) of 16×16 pixels blocks. In the decoding phase we follow the same flow diagram, but this time, backwards. Fig. 4.7 shows how the flow diagram looks like in this phase.
4.4.1 Processing of the interleaved coefficients

The coefficients come interleaved in an \texttt{HPAArray}. Hence, first there are all the \texttt{Lev4Avg} coefficients, which are followed by the \texttt{Lev3Diff}, \texttt{Lev2Diff}, \texttt{Lev1Diff} coefficients. Finally the \texttt{Lev0Diff} can be found. The function \texttt{InvHPAArrayProcessing}, and its counterpart \texttt{HPAArrayProcessing} in the coding algorithm, takes care of the identification of the blocks that belong to each \texttt{HPAArray} and, consequently, modifies the pointer position in the memory interfaces that contain the $H$ coefficients and the $\rho$ remainders in order for them to be read by the \texttt{HPAValuesReading} and \texttt{RemReading} functions respectively. Fig. 4.8 shows part of the code that processes each \texttt{HPAArray}.

```c
for (iBlock=1; iBlock<=nBlocks; iBlock++)
  // call to the function for HPA processing, passing the array and the position i
  for (i=0; i<=hpaArrayCounter; i++)
  {
    // calculation of the position of the first element of the block i in the hpaArray
    absPixPos = (hpaArray[i] -1 ) * 256;
    MI_IncreasePointer(&mArrayProcessing, absPixPos*SYM_SIZE/8);
    InvHPAValuesCalculation (&mArrayProcessing, i, &lev4avg, &lev4rem, &lev3dif,
                            &lev3rem, &lev2dif, &lev2rem, &lev1dif, &lev1rem, &lev0dif);
    // restore state of the mInOut memin
    MI_RestoreState(&mArrayProcessing);
  }
```

Figure 4.8: Part of the code that implements the inverse HPA array processing.

4.4.2 Reading the HPA coefficients

The coefficients are read from the first memory interface and put in several dynamic arrays, one for each type. The arrays are generated in the \texttt{InvHPAArrayProcessing} function and passed to \texttt{HPAValuesReading}. After reading them, the arrays are expanded. That is, before each group of three elements, a fourth void element is added, in order to account for the coefficients that are not transferred in the coding phase. Fig. 4.9 shows how the buffers expansion is implemented.

The reading phase of the $\rho$ values has been covered previously. After reading, all
4.4 The inverse HPA block array processing

```c
for (i = 0; i < (length*4/3); i++)
{
    if (((i+1)%4) != 1)
        tempBuf[i] = (*buf)[p];
    p++;
}
```

Figure 4.9: Part of the code that implements the buffer expansion algorithm.

the $H$ coefficient and $\rho$ remainder arrays are sent from the `InvHPAArrayProcessing` to the `InvHPACalculation` function.

### 4.4.3 The inverse HPA algorithm

This part of the code is in charge of reconstructing the average coefficients from the higher levels and, afterwards, the original pixels that compose the image. The algorithm works as follows. Let us consider that we already have three average coefficients from level $n$ and the average coefficients and remainders from level $n+1$. Let us call $A$, $B$ and $C$ the first three average coefficients, $D$ the missing one and $E$ the coefficient from the upper level. The following relation stands:

$$H_{n+1}(E) = \frac{H_n(A) + H_n(B) + H_n(C) + H_n(D)}{4} + \rho_{n+1}(E) \tag{4.1}$$

Hence, the missing coefficient is computed as follows:

$$H_n(D) = 4 \times (H_{n+1}(E) - \rho_{n+1}(E)) - H_n(A) - H_n(B) - H_n(C) \tag{4.2}$$

The reverse HPA algorithm is applied to all the coefficients that are not transferred during the coding, which, for each 16×16 pixels group, are the following:

- 1 Lev3Dif
- 3 Lev2Dif
- 12 Lev1Dif
- 48 Lev0Dif

There are four levels of iteration, each one nested in the previous one. First, the Lev4Avg, the Lev4Rem and the three Lev3Dif are combined with the inverse HPA algorithm to calculate the 4 Lev3Avg. After that, in each level 3 block, the Lev3Avg and the corresponding Lev3Rem are used to compute the 16 Lev2Avg coefficients. The same procedure is repeated until all the 256 original pixels are calculated. Fig. 4.10 illustrates this flow diagram, and Fig. 4.11 shows the piece of code that implements the inverse HPA algorithm.
After the reconstruction of the missing coefficients, also a first reordering of the pixels is done, and the final order is on a 16×16 block basis. This means that the output is a sequence of pixels ordered according to the 16×16 pixels block to which they belong. Namely, the first 256 pixels are the ones that belong to the first block, and so on. The logic used for the rearrangement is the same as the one used during the HPA calculation. There are four fixed arrays which contain the offsets needed to calculate the beginning of the blocks of level 3, 2 and of the elements of the blocks of level 1:

- int kOffset[ ] = 0, 8, 128, 136;
- int jOffset[ ] = 0, 4, 64, 68;
- int iOffset[ ] = 0, 2, 32, 34;
- int hOffset[ ] = 0, 1, 16, 17;

The logic consists of four for loops that allow to swap through these arrays and compute the right position in which the reconstructed element will be stored (in a similar way as in the coding algorithm). Fig. 4.12 shows part of the code for the pixel reconstruction.
4.5 The inverse block

The last operation of the decoding process consists of a pixel rearrangement. The pixels arrive ordered on a 16×16 pixels block basis and must be reordered according to the position they occupy globally in the image. The operation is carried out within the InvBlockCreation function in two operations. First, the current block position is identified in terms of block row and block column. The row and column are calculated as follows:

```c
for (iIndex = 0; iIndex < 4; iIndex++)
{
    InvHistogramBalancing (lev2avg[i], &lev1avg, *lev1dif, arrayPos*64, i*4, 4);
    InvHPAAlogorithm (lev2avg[i], (*lev2rem)[arrayPos*16 + i], &lev1avg, i*4, 0);
    for (jIndex = 0; jIndex < 4; jIndex++)
    {
        InvHistogramBalancing (lev1avg[j], &lev0diftemp, *lev0dif, arrayPos*256 + j*4, 4);
        InvHPAAlogorithm (lev1avg[j], (*lev1rem)[arrayPos*64 + j], &lev0diftemp, 0, 0);
        for (hIndex = 0; hIndex < 4; hIndex++)
        {
            outPixelPos = kOffset[kIndex] + jOffset[jIndex] + iOffset[iIndex] + hOffset[hIndex];
            lev0pixel[outPixelPos] = lev0diftemp[hIndex];
            MI_IncreasePointer(miOut, outPixelPos*SYM_SIZE/8);
            MI_Put(miOut, lev0diftemp[hIndex], SYM_SIZE);
            MI_RestoreState(miOut);
        }
        j++;
    }
    i++;
}
```
for (iBlock=1;iBlock<=nBlocks;iBlock++)
{
    // first pixel index calculation.. two parts
    // last pixel of the previous big line (i_BlockRow-1)
    lPixel = (iBlockRow-1)*nHorBlocks*256;
    for (iPixelRowWithinBLock = 1; iPixelRowWithinBLock <= 16; iPixelRowWithinBLock++)
    {
        // calculation of the first pixel of the current line (iPixelRowWithinBLock)
        int fPixel=lPixel + (iPixelRowWithinBLock-1)*nHorBlocks*16 +
            (iBlockRow-1)*16;
        for (iPixelColumnWithinBLock = 0; iPixelColumnWithinBLock< 16; iPixelColumnWithinBLock++)
        {
            // absPixelPos is the position of the final uncompressed array, place to save
            absPixelPos = fPixel + iPixelColumnWithinBLock+compressedPosCorrection;
            // position of the pixel in the compressed array. It's given by number of
            compressedPos= (iBlock-1)*256 + (iPixelRowWithinBLock-1)*16 +
                iPixelColumnWithinBLock + compressedPosCorrection;
            MI_IncreasePointer(mOutput, compressedPos*SYM_SIZE/8);
            imageSample = MI_Get32(mOutput, SYM_SIZE);
            MI_IncreasePointer(&mAdditional, absPixelPos*SYM_SIZE/8);
            MI_Put(&mAdditional, imageSample, SYM_SIZE);
            MI_RestoreState(mOutput);
            MI_RestoreState(&mAdditional);
        }
    }
}

Figure 4.14: The piece of code that implements the inverse block algorithm.

iBlockRow = ceil ( i / nHorBlocks );
iBlockCol = i % nHorBlocks;
if (iBlockCol == 0) iBlockCol = nHorBlocks;

where iBlock is the number of the current block and nHorBlocks is the width of the
image expressed in blocks number. Afterwards, the position of the single pixels in the
current block is calculated, in terms of pixel row and column in the original image,
using the data of the first step. Fig. 4.13 displays the representation of a 32×32
pixels image, before (left panel) and after (right panel) the final pixel reordering,
whereas Fig. 4.14 shows the code that implements the part of the algorithm for the
pixel identification and reordering.
Chapter 5

Results

In this chapter we present the main results of this work. We describe here the data used and the system configuration. Afterwards, we present the performance of the HPA lossless compression algorithm. Other analyses, like a visual representation of the different hierarchical levels and histograms, are shown as well. Finally, as a demonstration of the correctness of the coding algorithm, we run the HPA decoder to verify that the reconstructed image is identical to the original one.

5.1 System configuration and test data

As described in the previous chapters, the algorithm was developed in C language. The Eclipse IDE was used to edit and organize the code. The GNU Compiler Collection (GCC) was used to compile the code to binary files. The code has been developed using the original FAPEC code as basis, so the starting point was the usual FAPEC environment. That is, the code has been compiled and tested in a Linux operative system. More specifically, Fedora 18 and Debian 3.2.46 based on Linux 3.2.0-4 and Linux 3.6.10-4.fc18.1686 respectively. The architectures on which both the operative systems are installed are Intel i686 and i686-64bit.

The data samples used as input consist on a set of different images in raw format. The reason for choosing this format is to better simulate the behavior of the coding-decoding process as if the data was coming directly from the sensor that takes the pictures. The images used are the following:

- **banyoles-1024x592.raw**: 8 bit grayscale image of 1024×592 pixels.
- **catedral-1024x592.raw**: 8 bit grayscale image of 1024×592 pixels.
- **eixample-1024x592.raw**: 8 bit grayscale image of 1024×592 pixels.
- **field-1024x592.raw**: 8 bit grayscale image of 1024×592 pixels.
• mosaic10.raw: 16 bit (big endian) grayscale image of 1024×1024 pixels.

Note that we have intentionally selected image dimensions which are multiple of the basic 16×16 pixels HPA blocks. Figs. 5.1 to 5.5 show the images used.

Figure 5.1: Sample image: Banyoles (1024×592 pixels, 8-bit grayscale).

Figure 5.2: Sample image: Catedral (1024×592 pixels, 8-bit grayscale).
5.1 System configuration and test data

Figure 5.3: Sample image: Eixample (1024×592 pixels, 8-bit grayscale).

Figure 5.4: Sample image: Field (1024×592 pixels, 8-bit grayscale).
5.2 Visualization of HPA coefficients

The HPA algorithm divides and analyzes the images in different hierarchical levels, extracting average values from each one of them. In order to give a visual representation of the algorithm, we decided to generate several images, each one made of the coefficients extracted from each level. Two groups of pictures were generated: a first one, containing the images made of the average coefficients, and a second one, made of the differential coefficients of the level 0. The two cases are discussed separately.

5.2.1 Individual visualization of the average coefficients

The pictures of the first group are made of the average coefficients of the different levels. As said, from each 16×16 pixels block, the following coefficients are extracted:

- 1 $h_4$ coefficient;
- 4 $h_3$ coefficients, distributed in two columns and two rows;
- 16 $h_2$ coefficients, distributed in four columns and four rows;

Figure 5.5: Sample image: Mosaic10 (1024×1024 pixels, 16-bit grayscale).
5.2 Visualization of HPA coefficients

Figure 5.6: From left to right and from top to bottom: image composed of the level 4, 3, 2 and 1 (zoomed) average coefficients.

- 64 $H_1$ coefficients, distributed in eight columns and eight rows.

Let us consider the $H_1$ coefficients. For each block of 256 pixels, 64 coefficients are generated. This means that, if we replace the $16 \times 16$ block of original pixels with the $8 \times 8$ block of $H_1$, we will obtain another image which has half the dimensions of the original image. We repeated the same calculation for the different levels and, finally, obtained the following images:

- An image made of the $H_4$ coefficients: its dimensions are $1/16^{th}$ the original image;

- An image made of the $H_3$ coefficients: its dimensions are $1/8^{th}$ the original image;

- An image made of the $H_2$ coefficients: its dimensions are $1/4^{th}$ the original image;

- An image made of the $H_1$ coefficients: its dimensions are half the original image.

As an example of the aboved mentioned procedure we used the Example image. A detail of the four images is shown for comparison in Fig. 5.6.
5.2.2 Mixed visualization of the average coefficients

From the previous images it is clear that, going up to the upper levels, the images composed by the average coefficients can be considered as the original image scaled down to a lower resolution. Besides the representation shown in the previous paragraph, we made another image composed by the average coefficients from the four levels mixed together. This image is computed using the image composed by the $H_1$ as starting point, as described in the previous section, and thus its dimensions are half the original image. The new representation can be used, once again, to compare the resolution achievable with the different hierarchical levels. A scheme that explains how the average values are copied and repeated is shown in Fig. 5.7.

The rules followed are these:

- The $16 \times 16$ pixels block placed on the upper left corner is reconstructed using the $H_4$ coefficient repeated all over the block.

- The 3 blocks placed within the second row and second column are reconstructed using their $H_3$: each coefficient is repeated within the $8 \times 8$ pixel block it belongs to.

- The 12 blocks placed within the third and fourth row and the third and fourth column are reconstructed using their $H_2$: each coefficient is repeated within the $4 \times 4$ pixels block it belongs to.

- the rest of the image consists of the $H_1$ coefficients.

Fig. 5.8 shows the image reconstructed according to this procedure. Only the upper left corner is shown.
5.2 Visualization of HPA coefficients

5.2.3 Visualization of the level 0 differential coefficients

As explained in Chapter 2, the $\Delta H_0$ coefficients represent the difference between the actual pixels and the average coefficients of level 1. By construction, these coefficients assume values in a narrower range than the original pixels. In order to show this, we decided to produce an image made of the $\Delta H_0$ only and analyze its histogram. Two versions of this image were built. In the first one with the $\Delta H_0$ coefficients as they are: the histogram is centered in zero, but when displaying it as unsigned integers the negative values are mirrored in the higher part of the available range. In the second one we subtracted $2^{SYM\_SIZE - 1}$ (which is equal to 128 for the 8 bits image) to the $\Delta H_0$ coefficients. In this way the histogram is now centered in 128 and it assumes only positive values, so no distortion takes place when displaying it as unsigned values.

Fig. 5.9 shows the images built in this way. The image used for this test was
again Example. As can be seen in this figure, the shape of the original image is preserved. The histograms of the two images are shown in Fig. 5.10.

![Histograms with $\Delta H_0$ centered in 0 (left panel) and 128 (right panel).](image)

A comparison with the original image is done in Fig. 5.11, which shows the original histogram. It can be seen how the mean value is different from zero and that the data are distributed in a broader range: the median value is 70 against the median value of 0 or 128 seen in Fig. 5.10, and the standard deviation is $\sigma = 63.2$ which has to be compared with $\sigma = 23.2$ obtained in Fig. 5.10.

![Histogram of the original image.](image)

5.3 Coding-decoding process

We have implemented the inverse HPA algorithm in the decoder, which gives us the possibility to prove the correctness of the algorithm implemented in this work. The
images used for this test are the ones previously listed. In particular, we remind that we used 4 8-bit images, namely, Banyoles-1024x592.raw, Catedral-1024x592.raw, Eixample-1024x592.raw, Fields-1024x592.raw, and 1 16-bit image: Mosaic10.raw.

ricky@rDebian:~/workspace/FAPEC/src/code$ ls -l *.raw
-rw-r--r-- 1 ricky ricky 606208 Aug 26 14:33 banyoles-1024x592.raw
-rw-r--r-- 1 ricky ricky 606208 Aug 26 14:33 catedral-1024x592.raw
-rw-r--r-- 1 ricky ricky 606208 Aug 26 12:49 eixample-1024x592.raw
-rw-r--r-- 1 ricky ricky 606208 Aug 26 14:33 field-1024x592.raw
-rw-r--r-- 1 ricky ricky 2097152 Aug 26 15:06 mosaic10.raw

ricky@rDebian:~/workspace/FAPEC/src/code$ ls -l *.fapec
ls: cannot access *.fapec: No such file or directory

ricky@rDebian:~/workspace/FAPEC/src/code$ ls -l *.fapec.rst
ls: cannot access *.fapec.rst: No such file or directory

ricky@rDebian:~/workspace/FAPEC/src/code$ ./fapec banyoles-1024x592.raw -m fp 8 -img 1024 592 -np tc > comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./fapec catedral-1024x592.raw -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./fapec eixample-1024x592.raw -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./fapec field-1024x592.raw -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./fapec mosaic10.raw -m fp 16 -img 1024 1024 -np tc >> comp.log

ricky@rDebian:~/workspace/FAPEC/src/code$ ./decoder banyoles-1024x592.raw.fapec -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./decoder catedral-1024x592.raw.fapec -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./decoder eixample-1024x592.raw.fapec -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./decoder field-1024x592.raw.fapec -m fp 8 -img 1024 592 -np tc >> comp.log
ricky@rDebian:~/workspace/FAPEC/src/code$ ./decoder mosaic10.raw.fapec -m fp 16 -img 1024 1024 -np tc >> comp.log

ricky@rDebian:~/workspace/FAPEC/src/code$ ./workspace/FAPEC/src/code$ ls -l *.fapec
-rw-r--r-- 1 ricky ricky 606208 Nov 9 21:31 banyoles-1024x592.raw.fapec.rst
-rw-r--r-- 1 ricky ricky 606208 Nov 9 21:31 catedral-1024x592.raw.fapec.rst
-rw-r--r-- 1 ricky ricky 606208 Nov 9 21:32 eixample-1024x592.raw.fapec.rst
-rw-r--r-- 1 ricky ricky 606208 Nov 9 21:32 field-1024x592.raw.fapec.rst
-rw-r--r-- 1 ricky ricky 2097152 Nov 9 21:35 mosaic10.raw.fapec.rst

ricky@rDebian:~/workspace/FAPEC/src/code$ diff banyoles-1024x592.raw banyoles-1024x592.raw.fapec.rst
ricky@rDebian:~/workspace/FAPEC/src/code$ diff catedral-1024x592.raw catedral-1024x592.raw.fapec.rst
ricky@rDebian:~/workspace/FAPEC/src/code$ diff eixample-1024x592.raw eixample-1024x592.raw.fapec.rst
ricky@rDebian:~/workspace/FAPEC/src/code$ diff field-1024x592.raw field-1024x592.raw.fapec.rst
ricky@rDebian:~/workspace/FAPEC/src/code$ diff mosaic10.raw mosaic10.raw.fapec.rst

Figure 5.12: Log of the comparison between the original files and the decoded ones.
In order to produce a clean log, we started from scratch, from a situation in which there are only the original images and no compressed files. We then called FAPEC with the HPA pre-processing to compress all the input files. We redirected the log to a file, so the output to the terminal keeps clean. When compressing the files, FAPEC adds the .fapec suffix to the original filename extension. The same thing happens with the decoder: it adds the .rst suffix to the compressed image filename.

![Figure 5.13: Comparison of the FAPEC compression ratios with and without HPA.](image)

After the reconstruction, we use the `diff` command-line Linux tool to compare the original image with the reconstructed one. This tool only generates output when the two images differ. The terminal output of this test is shown in Fig. 5.12: the images are correctly compressed and decompressed and the reconstructed images are identical to the original ones. It demonstrates the correct operation of lossless HPA+FAPEC image compression.

### 5.4 Performance of FAPEC with HPA

Here we compare the performance of FAPEC with the HPA image pre-processing algorithm against FAPEC alone (with a plain differential pre-processing stage). We have used the same images previously described in order to have results for both 8 bits and 16 bits images.

The results in terms of compression ratio are shown in Fig. 5.13. As can be seen, the ratios are very similar for both FAPEC alone and FAPEC with HPA.
5.4 Performance of FAPEC with HPA

Figure 5.14: Comparison of the FAPEC execution time with and without HPA.

pre-processing. It is actually a very good result, considering that the current implementation of HPA is just a prototype, and that FAPEC alone is already providing excellent ratios on images. As we will see in the following section, the advantage of HPA is the possibility of introducing controlled losses in the image compression. Having already good ratios with the lossless option means a very good starting point for the lossy case.

We have also compared the execution time for both HPA+FAPEC and FAPEC. The result, in microseconds, is shown in Fig. 5.14. Again, the results are very similar for both cases, although in the case of FAPEC with HPA the values are slightly smaller — that is, FAPEC seems to be quicker when running with the HPA pre-processing. This surprising result can probably be explained by the way in which FAPEC receives the HPA coefficients, that is, in blocks of values with quite similar statistics. That means that FAPEC almost always uses the same coding option for that block, which can then be efficiently handled by the internal processor cache.
Chapter 6

Lossy version of HPA

The last goal of this work was to develop a first prototype of a lossy version of the HPA algorithm. Such a prototype will undoubtedly increase its applicability, since there are many applications that require a high compression ratio provided that the quality of the image, typically expressed in terms of Peak Signal to Noise Ratio (PSNR), is high enough. Here we explain the strategy followed to modify the algorithm, describe which part of information is lost, and finally present the results.

6.1 Logic

The strategy used to implement the Lossy version is quite simple. We start from the HPA data packet, which we remind to the reader in Fig. 6.1. The evolution to a lossy version of the algorithm is simple and consists in removing some of the least significant bits (LSB) from the $\Delta H$ or the $\rho$ coefficients. We opted to allow different configurations depending on the quality level required. In particular, in our strategy the number of the LSB removed from the $\Delta H$ coefficients varies from 0 to 8 and the number of LSB removed from the $\rho$ varies from 0 to 2. Following this strategy, we define the Quality Levels (hereafter $QL$) of lossy HPA as follows:

- $QL\;0$: it is the lossless compression (no LSB removal).
- $QL\;0.5$: we remove 1 bit from the $\rho_1$ coefficients.
- $QL\;1$: we remove 1 bit from $\Delta H_0$ and 1 bit from $\rho_1$. In practice, this is equivalent to remove 1 bit per pixel.
- $QL\;2$: 2 bits from $\Delta H_0$ and 2 bits from $\rho_1$ are removed. This means that we remove $\rho_1$ completely.
- $QL\;3$: we discard 3 bits from $\Delta H_0$, 1 bit from $\Delta H_1$, 2 bits from $\rho_1$ and 1 bit from $\rho_2$. 
- **QL 4**: 4 bits from $\Delta H_0$, 2 bits from $\Delta H_1$, 2 bits from $\rho_1$ and 2 bits from $\rho_2$ are discarded.
- **QL 5**: we eliminate 5 bits from $\Delta H_0$, 3 bits from $\Delta H_1$, 1 bit from $\Delta H_2$, 2 bits from $\rho_1$, 2 bits from $\rho_2$ and 1 bit from $\rho_3$.
- **QL 6**: 6 bits from $\Delta H_0$, 4 bits from $\Delta H_1$, 2 bit from $\Delta H_2$, and $\rho_1$, $\rho_2$ and $\rho_3$ are eliminated.
- **QL 7**: 7 bits from $\Delta H_0$, 5 bits from $\Delta H_1$, 3 bits from $\Delta H_2$, 1 bits from $\Delta H_3$, 2 bits from $\rho_1$, $\rho_2$ and $\rho_3$, and 1 bit from $\rho_4$ are dropped.
- **QL 8**: we drop 8 bits from $\Delta H_0$ (that is, this coefficient is completely removed), 6 bits from $\Delta H_1$, 4 bits from $\Delta H_2$, 2 bits from $\Delta H_3$, and all $\rho$ remainders.

In reception, when the $\rho$ coefficients are not sent (let us take as an example $\rho_1$), they are considered equal to zero and the reconstruction of $\mathcal{H}_1$ will be approximated as given in Eq. (6.1):

$$\mathcal{H}_1 = 4 \times H_1 + \rho_1 \cong 4 \times H_1 \quad (6.1)$$

From this we have that each fourth pixel that is reconstructed with the InvHPAAlgorithm is approximated by the expression given in Eq. (6.2):

$$\mathcal{H}_0' \cong 4 \times H_1 - 3 \times \mathcal{H}_0 \quad (6.2)$$

Regarding the $\Delta H$, we decided to remove a number $n$ of LSB from each coefficient by performing a right shift of $n$ bits. In the decoding process, to approximate the original values, a left shift of $n$ bits is performed to the $\Delta H$ coefficients. This is equivalent to divide by $2^n$ the $\Delta H$ coefficient. This approach does not approximate the coefficients accurately. To overcome this, we developed a function, called quickDivAndRound, which rounds the $\Delta H$ to the nearest number multiple of $2^n$. Fig. (6.2) shows the part of the function that implements this approach. With these modifications, the data packet sent by FAPEC is different for each level. To give an example, the data packet sent with QL 2 is represented in Fig. 6.3. Continuing with the QL 2 example, the consequence of dividing the $\Delta H_0$ by $2^2 = 4$ is that it reduces the entropy and,
consequently, FAPEC obviously leads to higher compression ratios. In Fig. (6.4) we compare the entropy in the lossless against the lossy implementation.

```c
int mask = 0x00000001 << (s-1);
if (n & mask) {
    return ((n >> s) + 1);
} else {
    return (n >> s);
}
```

Figure 6.2: Implementation of the rounding of the $\Delta H_0$

Figure 6.3: Structure of the Lossy HPA data packet.

Figure 6.4: Qualitative comparison of the entropy of the HPA coefficients for the lossy version (top panel) versus the lossless version of the compression algorithm (bottom panel).
6.2 Results

6.2.1 Peak Signal to Noise Ratio

To show the performance of this first prototype of the lossy HPA we calculate the compression ratio ($C_R$) and the PSNR of $QL_{0.5}$ to $QL_8$ with respect to the original image using the Example image as test data. We also decided to compare the performance achieved with our algorithm against the one achieved with JPEG, a very well known standard used worldwide for image compression. The JPEG standard allows to choose the quality desired during the compression: a higher quality corresponds to a higher PSNR at the cost of a lower $C_R$, and vice versa. In order to do a better comparison, we plot the values of $C_R$ and PSNR of the JPEG standard at different quality levels: from 100% to 70% with a 5% step and from 70% to 40% with a 10% step. Fig. (6.5) shows that for PSNR > 15 dB our algorithm performs very similar to JPEG.

6.2.2 Execution time

We have compared the execution time of FAPEC with HPA pre-processing with the execution time of Imagemagick, a command-line Linux tool designed for batch processing of images, when coding a raw image in JPEG. Once again we plot the values for different the different $C_R$ obtained. We took as test image the Big_building.pgm
Figure 6.6: Execution time of the Lossy HPA+FAPEC image compression algorithm.

file from www.imagecompression.info. The reason why we choose this picture is mainly for its size: 7216×5408 pixels (around 40 MB). As can be seen in Fig. (6.6) the execution time of FAPEC with HPA is still somewhat larger than that of the Imagemagick-JPEG. However, we judge this is something otherwise expected, since the code is still just a first prototype and it must be optimized.
Chapter 7

Conclusions

FAPEC is the fully adaptive version of PEC (Prediction Error Coder), an entropy coding algorithm that requires an adequate pre-processing stage. It can be as simple as a differentiator, just predicting each new sample as being equal to the previous sample. Nevertheless, more elaborated pre-processing stages providing better predictions will obviously generate smaller prediction errors and, thus, FAPEC will obtain better compression ratios. In this work we have developed a new pre-processing stage specific for images. We called it HPA, which stands for “Hierarchical Pixel Averaging”. The goal was to obtain an efficient image compression algorithm allowing to progressively move from lossless compression to lossy compression with different quality levels.

Here we have described the logic behind the HPA algorithm, which is different to other image compression algorithms based on wavelets or other kind of transformations. HPA processes the images by dividing them in different hierarchical levels, from blocks of $2 \times 2$ pixels (or Level 1 blocks) up to blocks of $16 \times 16$ pixels (or Level 4 blocks). Such Level 4 blocks conform the basic unit of the HPA algorithm, thus being able to operate in very small parts of the images — which is a clear advantage for applications such as satellite compression. Afterwards, HPA extracts average values and remainders from each level, leading to a total of 256 $H$ coefficients plus 170 bits for the $\rho$ remainders for each $16 \times 16$ block. This means that we add an overhead of about 0.66 bits per pixel, but that is compensated by the smaller entropy levels achieved in the $H$ coefficients. We must also note that we have taken this approach (instead of the typical wavelet-based or transformation-based approaches) in order to avoid the “artifacts” that can be seen in the restored images when they are lossy compressed with very low quality levels.

In this report we have described the software developed for the coding and decoding HPA algorithms, which has been implemented in C and tested on a Linux platform. We have presented the results and performance of the HPA algorithm on a variety of images — mainly from Earth observation satellites covering different
scenarios. With this approach, the entropy, in terms of bits/pixel, was significantly lowered. The whole HPA + FAPEC system was able to offer similar lossless ratios and compression times than FAPEC alone. This is a good result considering that FAPEC is able to provide excellent performances already. Having such a good starting point will obviously allow to achieve excellent lossy results. The correct lossless operation of HPA has been assessed by comparing the restored images against the original ones, confirming that not a single bit was lost in the process.

Finally, a first prototype of the lossy version of the HPA algorithm has been developed and tested successfully. We have compared its performance against JPEG — one of the most used image compression algorithm worldwide nowadays. The results in terms of compression ratios versus PSNR (that is, image quality) are extremely similar to those obtained with JPEG for a wide range of quality levels.

These excellent results broaden the different applications and fields in which HPA+FAPEC can be used. The aerospace field is a clear candidate, in which the bandwidth is very expensive and thus a good compression algorithm would lower the cost — or improve the quality at the same cost. Another possible application is imaging medicine — especially telemedicine, which is becoming more and more frequent. A good image compression algorithm may allow to transfer images with the best quality in shorter times or with smaller requirements in terms of bandwidth. Finally, the nature of HPA also allows to seamlessly perform progressive image decompression at different quality levels using the same image file.

7.1 Future work

There are many possibilities for future improvements to the HPA. First of all, the work done so far is limited to a single-band image, that is, images in grayscale. The first step should be to support color images or, in general, multi-band images. That is, simple color images (such as those composed of the three typical RGB channels), or 3D images, or even multispectral or hyperspectral data. As far as the images are provided in raw format, the implementation of the multichannel processing can be done quite easily. At a later stage we may investigate how to make best use of the inter-band correlation to achieve better ratios.

Another possibility is to change the interpolating algorithm used for the calculation of the fourth coefficient that is not transmitted. Different kinds of interpolation algorithms could be studied (linear, bilinear, quadratic, . . .) to find the one that performs better. This will probably depend on the image to be processed, since for a smoother image it could be enough to use the linear interpolation, while for a more variable image the bilinear or quadratic interpolation could be the best solution. Therefore, some tests will be needed here. A possible implementation is to add a pre-processing stage in which the different schemes are applied and the best one is automatically chosen.
7.1 Future work

The implementation of HPA should be optimized in order to achieve even better execution speeds. That would obviously provide even more advantage with respect to other compression algorithms.

Additional tests must also be done, not only on a wider variety of images (including color and multi-band images, when HPA is able to handle them), but also including other competitors such as CCSDS 122.0, DWTFAPEC, JPEG-LS and JPEG2000.

The exact HPA file format should be revised, as well as the lossy compression and (especially) decompression approach, in order to provide the previously mentioned feature of progressive image decoding at incremental quality levels. That would be an extremely interesting feature for mobile applications, for example.

Finally, in Appendix A we present some alternative schemes analyzed during the development of this work. In these schemes, the hierarchical approach is modified and applied in a different way. It could be worthwhile to further develop and analyze them in order to see if they can provide better results than the original HPA, specially for the lossless case.
# Acronyms

The following table has been generated from the on-line Gaia acronym list:

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>CCSDS</td>
<td>Consultative Committee for Space Data Systems</td>
</tr>
<tr>
<td>DWT</td>
<td>Discrete Wavelet Transform</td>
</tr>
<tr>
<td>DWTFAPEC</td>
<td>Discrete Wavelet Transform FAPEC</td>
</tr>
<tr>
<td>ESA</td>
<td>European Space Agency</td>
</tr>
<tr>
<td>FAPEC</td>
<td>Fully Adaptive Prediction Error Coder</td>
</tr>
<tr>
<td>FITS</td>
<td>Flexible Image Transport System</td>
</tr>
<tr>
<td>FLP</td>
<td>Frame-Loss Probability</td>
</tr>
<tr>
<td>GAIA</td>
<td>Global Astrometric Interferometer for Astrophysics (obsolete; now spelled as Gaia)</td>
</tr>
<tr>
<td>HB</td>
<td>Histogram Balancing</td>
</tr>
<tr>
<td>HPA</td>
<td>Hierarchical Pixel Averaging</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organisation for Standardisation (Geneva, Switzerland)</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JPL</td>
<td>Jet Propulsion Laboratory (DE ephemerides)</td>
</tr>
<tr>
<td>NDA</td>
<td>Non-Disclosure Agreement</td>
</tr>
<tr>
<td>PFL</td>
<td>Probability of Frame Loss (also known as FLP)</td>
</tr>
<tr>
<td>REM</td>
<td>Abbreviation for “remainder”</td>
</tr>
<tr>
<td>SPIE</td>
<td>Society of Photo-optical Instrumentation Engineers</td>
</tr>
<tr>
<td>TRP</td>
<td>Technology Research Programme (ESA)</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>USA</td>
<td>United States of America</td>
</tr>
</tbody>
</table>
Bibliography


Appendix A

Alternative coding schemes

This appendix presents some of the alternative coding schemes that were analyzed in this work. Because of their dubious performance in terms of compression ratio, these schemes were not pursued further. Nevertheless, given the simple but rather powerful ideas behind them, it is worth to present them here and to keep them in mind for future opportunities that may appear, for example creating a more adaptive coding scheme that tries different alternatives on the image to be compressed before choosing the best one.

A.1 HPA Stopper algorithm

Many images contain quite uniform areas covering the majority of the image. An example may be a typical astronomical picture containing a dark background and very few stars. For this kind of images, the decomposition in four hierarchical levels may be unnecessary. In the best case, let us consider to have all the 256 pixels very similar each other — and therefore very similar to the lev4Avg value (i.e., $H_4$ coefficient). In that case, instead of calculating the differential coefficients $\Delta H_0$ w.r.t. $H_1$, we may actually obtain better results (and with less operations) if we calculate them directly w.r.t. $H_4$. This means that, for each lev0Pixel[i] (or $\Delta H_0(i)$), the following equation applies:

$$\Delta H_0(i) = H_0(i) - H_4(A)$$  \hspace{1cm} (A.1)

In this case, the only transmitted coefficients will be $H_4$ and its 255 associated $\Delta H_0$.

We have implemented the similarity check between the single pixels and the $H_4$ coefficient by comparing the most significant bits (or MSB). To make the code more versatile, the number of bits to be compared is set in a configurable parameter. When the result of the check is negative (that is, when the individual pixels are not similar enough to $H_4$), another step follows: the 16×16 pixels block is divided into four 8×8 blocks and the same procedure is repeated with the $H_3$. That is, in each
8×8 block, all the pixels are compared with the corresponding $H_3$. If the result of the comparison is positive, then the $\Delta H_0$ of that 8×8 block will be calculated w.r.t. the $H_3$. This means that, for each $\Delta H_0(i)$, the following equation stands:

$$\Delta H_0(i) = H_0(i) - H_3(A) \quad (A.2)$$

If, for an 8×8 block, even this comparison is negative (i.e., the pixels are not similar enough to $H_3$), then that block will be coded according to the normal HPA scheme.

The overhead of this approach w.r.t. the original HPA scheme is 5 bits for each 16×16 pixels block, set in this way:

- 10000: this bit sequence means that the first comparison gave positive results. In this case, the other comparisons are not performed. Instead, the HistogramBalancing function is directly called and new values of $\Delta H_0$ are generated from the direct subtraction of lev4Avg.

- 0XXXX: if the first comparison is negative, the first bit is set to 0 and the remaining 4 bits are set according to the comparison of the pixels of a 8×8 block with the correspondent $H_3$. When a bit is set to 1, the HistogramBalancing function is called and new values of $\Delta H_0$ are generated from the subtraction with $H_3(i)$.

One important consideration is that only in the first case the entire 16×16 pixels block is processed at once, while for the other scheme an iteration is done to process each 8×8 pixels block separately.

The overhead introduced by this scheme can be calculated as follows. For every 256 pixels there are 5 bits of overhead — which is 0.24% of the original size in the case of 8 bit images, or 0.12% in the case of 16 bits images. These bits are written in an array, byteHPAStopper, which is transferred to FAPEC in order to allow, in the reception, the correct decoding. Another characteristic of this scheme is that the number of $H_n$, with $n = 1, 2, 3$, is variable and must be calculated when sending and receiving the coefficients.

To describe the implementation of this scheme, let us call HPA mode 1 the scheme in which $\Delta H_0$ are calculated w.r.t. the $H_4$, and HPA mode 2 the scheme in which $\Delta H_0$ are calculated w.r.t. the corresponding $H_3$. Then we have the following counters:

- mD1 is the number of blocks processed with the mode 1 scheme.
- no-mD1 is the number of blocks for which we could not select mode 1 (that is, nblocks - mD1).
- mD2 is the number of sub-blocks processed according to the mode 2 scheme.
A.2 HPA SUM algorithm

The idea in this scheme is to apply the HPA algorithm but, instead of sending the average values to the memory interface, send the sum of all the pixels belonging to each level. In order to do so, we modified our code to take advantage of the logic...
already existing. To save the values of the sum coefficients, four new arrays were defined: \(\text{lev4sumGlobal}, \text{lev3sumGlobal}, \text{lev2sumGlobal}, \text{lev1sumGlobal}\). They are passed to the \(\text{HPAValuesCalculation}\) and, from there, to the \(\text{HPASumCalculation}\), which is where their value is calculated. The \(\text{HPASumCalculation}\) works at the level of a 16×16 pixels block: it loops through the \(\text{lev1sum}\) array and updates the four new arrays as follows. Note that \(\text{clusterPos}\) is the index of the current block:

- \(\text{lev1sumGlobal }[\text{clusterPos} \times 64 + i] = \text{lev1sum}[i]\), where \(i\) assumes values from 0 to 64, in order to swap through the 64 lev1sum coefficients.

- \(\text{lev2sumGlobal }[\text{clusterPos} \times 16 + j] = \text{lev2sumGlobal }[\text{clusterPos} \times 16 + j] + \text{lev1sum}[i]\), where \(j = i / 4\), so it assumes values from 0 to 16.

- \(\text{lev3sumGlobal }[\text{clusterPos} \times 4 + k] = \text{lev3sumGlobal }[\text{clusterPos} \times 4 + k] + \text{lev1sum}[i]\)
  where \(k = j / 4\), so it assumes values from 0 to 4.

- \(\text{lev4sumGlobal}[\text{clusterPos}] = \text{lev4sumGlobal}[\text{clusterPos}] + \text{lev1sum }[i]\)

In this implementation, the \(\rho\) remainders are not used and so is the \(\text{mRem}\) memory interface. The size of the \(\text{mOut}\) memory interface is calculated considering the number of additional bits required by the sum coefficients. It is done as follows:
A.3 Differential HPA SUM algorithm

Instead of transmitting all the sum coefficients, the same logic of the HPA of transmitting three out of four coefficients was used. Hence, at the end, the following coefficients will be used:

- 1 lev4sumGlobal, which means 8 bits.
- 3 lev3sumGlobal, which means 18 bits.
- 12 lev2sumGlobal, which means 48 bits.
- 48 lev1sumGlobal, which means 96 bits.

This is equal to 22 bytes for each 16 × 16 block. Fig. A.3 shows the data structure sent to FAPEC in this processing scheme.

A.3 Differential HPA SUM algorithm

This scheme is an alternative version of the previous one. The sum of all the pixel belonging to each level are calculated, but this time they are provided differentially to FAPEC. The reference levels used in this scheme are the fourth part of the global sum coefficients from the upper level, calculated as in the previous scheme. The differential coefficients are calculated in the HPASUMDIFFCalculation function as follows:
• $\Delta H_3(k) = \text{lev3sumGlobal}(k) - \text{lev4sumGlobal} \gg 2$, with $k$ adopting values between 0 and 3.

• $\Delta H_2(j) = \text{lev2sumGlobal}(j) - \text{lev3sumGlobal}(k) \gg 2$, with $k = j / 4$ and $j$ taking values between 0 and 15.

• $\Delta H_1(i) = \text{lev1sumGlobal}(i) - \text{lev2sumGlobal}(j) \gg 2$, with $j = i / 4$ and $i$ taking values between 0 and 63.

• $\Delta H_0(p) = \text{lev0pixel}(p) - \text{lev1sumGlobal}(i) \gg 2$, with $i = p / 4$ and $p$ taking values between 0 and 255.

Fig. A.4 shows the data structure sent to FAPEC in this processing scheme.

### A.4 3 levels HPA algorithm

![HPA data](image)

Figure A.5: Data structure of the 3 levels HPA.

This scheme is very similar to the normal HPA, with the exception that the image is not divided hierarchically in 4 levels, but only in 3 levels. The consequence is that for each 16×16 blocks all the $H_3$ are transmitted, instead of sending the $H_4$ with only 3 $\Delta H_3$. Also, as expected, the number of total coefficients remains the same: 256 per block.

• Level 0: 192 $\Delta H_0$.

• Level 1: 48 $\Delta H_1$.

• Level 2: 12 $\Delta H_2$.

• Level 3: 4 $\Delta H_3$.

In order to implement a first version, the code was developed as a modification of the HPA. The HPAArrayProcessing and HPAValuesCalculation function remain almost the same. What changes is that the $H_4$ are not calculated here. Fig. A.5 shows a representation of the data structure of the 3 levels HPA scheme.
A.5 2 levels HPA algorithm

This scheme is similar to the previous one, but this time the extraction of the coefficients stops at level 2. As in the 3 levels HPA algorithm, the coefficients count is 256 again, grouped in the following way:

- Level 0: 192 $\Delta H_0$.
- Level 1: 48 $\Delta H_1$.
- Level 2: 16 $\Delta H_2$.

Fig. A.6 shows a representation of the data structure of the 2 levels HPA scheme.