Feasibility study of the PEC compressor in HDF5 file format

by

Carlos Estepa Sánchez

Advisor: Enrique García-Berro Montilla
Co-advisor: Jordi Portell i de Mora

September 2011
Ahhh, what an awful dream!

There were ones and zeroes everywhere! And I thought I saw a 2...

Bender (Futurama)
Acknowledgements

First of all, I have to say THANK YOU, from the deep of my soul, to Enrique García-Berro and Jordi Portell, for giving me their trust and their knowledge, encouraging me to follow each step on my way and making me a better person. All the time we spent together making this possible has been a pleasure, I will never forget it. I also want to give kind regards to other people in IEEC/UPC/UB who I had the pleasure to share my time with: Alberto González, Marcial Clotet and Javier Castañeda. Thank you all for your support.

I want to mention here all the people who deal with me day after day, and have been with me during the development of this project: all my workmates in Abelló Linde - and its dependent companies Linde Medicinal and INO Therapeutics -; my trainer Alejandro and all his pupils and workmates in Club Esportiu Paradís; my classmates at the UPC: Tamara, Silvia, Alberto, Raquel, David, etc.; my friends Eli, Hugo, Montse, Raúl and Manfred and, with special affection, my mother, my aunties and uncles, my grandparents and all my cousins. You are also (a little) part of this!

And, of course, this chapter of my life cannot be closed without making any references to the person who I am sharing it: Sandra, this is for you. Thank you for your patience. Thank you for being there in good and bad moments. Thank you for being the way you are. I love you.
Table of Contents

1. The Gaia Mission...............................................................................................................................5
   1.1 Overview ........................................................................................................................................5
   1.2 Gaia Data Processing .........................................................................................................................6
       1.2.1 Gaia Data Processing and Analysis Consortium (DPAC)......................................................6
       1.2.2 Data Simulations (CU2) ...........................................................................................................7
       1.2.3 Core Processing (CU3) .............................................................................................................8
       1.2.4 Initial Data Treatment (IDT) ......................................................................................................9
       1.2.5 Astrometric Core Solution (AGIS) ...........................................................................................9
       1.2.6 Intermediate Data Updating (IDU) ..........................................................................................9
   1.3 Objectives of this work ..................................................................................................................10

2. The Prediction Error Coder (PEC) ....................................................................................................13
   2.1 Introduction ....................................................................................................................................13
   2.2 Description of PEC ..........................................................................................................................13
   2.3 Modified version for HDF5 ..........................................................................................................14

3. The Java Native Interface ..................................................................................................................17
   3.1 What is the JNI? .............................................................................................................................17
   3.2 Important implications ....................................................................................................................17
   3.3 How JNI works ..............................................................................................................................18

4. The HDF5 file format ........................................................................................................................19
   4.1 What is HDF? ...............................................................................................................................19
   4.2 Which are the HDF Technologies? ...............................................................................................19
   4.3 What is HDF5? ............................................................................................................................19
   4.4 Using the HDF5 data model ........................................................................................................19
   4.5 The Abstract Data Model ............................................................................................................21
       4.5.1 File ..........................................................................................................................................21
       4.5.2 Groups ....................................................................................................................................22
       4.5.3 Datasets ..................................................................................................................................22
       4.5.4 Dataspace ................................................................................................................................22
       4.5.5 Datatype ..................................................................................................................................23
       4.5.6 Attribute .................................................................................................................................24
   4.6 HDF5 in this project .......................................................................................................................24
   4.7 Example of a HDF5 File .................................................................................................................24

5. The GbinToHdf5 Application ............................................................................................................29
   5.1 The GBin file format ......................................................................................................................29
   5.2 Classifying data ............................................................................................................................31
   5.3 Structure of the HDF5 File for AstroObservations .....................................................................32
   5.4 GBin File Read and HDF5 File Write .........................................................................................32
   5.5 Compound dataset for header .....................................................................................................34
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Storing SM and AF samples</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>Obtained HDF5 File</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Format</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Size</td>
<td>40</td>
</tr>
<tr>
<td>6.3</td>
<td>Processing Time</td>
<td>41</td>
</tr>
<tr>
<td>6.4</td>
<td>Statistics</td>
<td>42</td>
</tr>
<tr>
<td>6.5</td>
<td>Interpretation of results</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Obtaining data from the HDF5 File</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions</td>
<td>47</td>
</tr>
<tr>
<td>8.1</td>
<td>Forthcoming work</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>Annex</td>
<td>51</td>
</tr>
<tr>
<td>9.1</td>
<td>Rice Coder Overview</td>
<td>51</td>
</tr>
<tr>
<td>9.2</td>
<td>Setup of the environment – HDF5 and Eclipse</td>
<td>53</td>
</tr>
<tr>
<td>9.3</td>
<td>Setup of the environment – DPAC and Eclipse</td>
<td>56</td>
</tr>
<tr>
<td>9.4</td>
<td>DPAC API: Methods used</td>
<td>57</td>
</tr>
<tr>
<td>9.5</td>
<td>HDF5 API: Used methods</td>
<td>60</td>
</tr>
<tr>
<td>9.6</td>
<td>Study of the need to split AstroObservations into groups</td>
<td>66</td>
</tr>
<tr>
<td>10</td>
<td>Bibliography</td>
<td>69</td>
</tr>
</tbody>
</table>
Index of figures

Fig. 1.1: The Gaia focal plane.................................................................................................................................................5
Fig. 1.2: CU2 Breakdown into Development Units........................................................................................................8
Fig. 2.1: Overview of the Prediction Error Coder (PEC) and its three coding strategies.............................................14
Fig. 3.1: Programming steps for a JNI-based “Hello World” program........................................................................18
Fig. 4.1: HDF5 Abstract Model Modules..................................................................................................................20
Fig. 4.2: HDF5 Library Dependences..............................................................................................................................21
Fig. 4.3: Dataspace properties..............................................................................................................................................23
Fig. 4.4: HDF Viewer (‘about’ dialog)..........................................................................................................................25
Fig. 4.5: Open File window from HDFview..................................................................................................................25
Fig. 4.6: Example of a HDF5 File....................................................................................................................................26
Fig. 4.7: Left Bar...........................................................................................................................................................................26
Fig. 4.8: Information Bar.........................................................................................................................................................27
Fig. 4.9: Image Tools..............................................................................................................................................................27
Fig. 4.10: Table example..........................................................................................................................................................27
Fig. 4.11: File Properties.........................................................................................................................................................28
Fig. 5.1: Snapshot of the Gaia Main DB Dictionary Tool, illustrating the AstroObservations data model.......................................................................................................................................................................................29
Fig. 5.2: Calculating the header space...........................................................................................................................................31
Fig. 5.3: Hyperslab selection example........................................................................................................................................33
Fig. 6.1: Header dataset of an AstroObservation HDF5 file............................................................................................39
Fig. 6.2: AF and SMsamples dataset.......................................................................................................................................40
Fig. 9.1.1: Rice codes for values $n = 0$ to 16 and parameter $k = 0$ to 5 ........................................................................51
Fig. 9.1.2: Adaptive Rice coder specified in the CCSDS 121.0 recommendation...............................................................52
Fig. 9.2.1: HDF5 Libraries, sorted by architecture..................................................................................................................53
Fig. 9.2.2: HDF Libraries for Windows 32 bits.......................................................................................................................53
Fig. 9.2.3: Folder creation in our project.................................................................................................................................54
Fig. 9.2.4: HDF5 Jar files included in our project......................................................................................................................54
Fig. 9.2.5: DLL libraries required to manage HDF5 files......................................................................................................55
Fig. 9.3.1: Gaia-DPAC libraries needed to access AstroObservation files..............................................................................56
Fig. 9.6.1: File Size per AstroObservation number.............................................................................................................68
1. The Gaia Mission

1.1 Overview

Gaia is an ambitious space observatory, 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 astronomical objects (until the 20\textsuperscript{th} magnitude) with unprecedented accuracy. As a result, a three-dimensional chart of more than 1 billion stars of our Galaxy will be obtained, as well as solar system objects and extragalactic sources. The precision of the angular measurements will be about 20 micro-arc seconds (μas) at a moderate brightness (15th magnitude). In order to achieve this, Gaia will use two telescopes combined onto a single focal plane composed of 106 state-of-the-art Charge Coupled Devices (CCDs), the complete layout of which is illustrated in Fig. 1.1.

![Fig. 1.1: The Gaia focal plane](image)

The satellite will continuously scan the sky, allowing for about 86 transits of each star during 5 years of duration of the mission, each transit being composed of several measurements. Full sky coverage will be possible because of the spin of the satellite around its own axis, which itself precesses at a fixed angle of 45° with respect to the Sun. Besides the angular measurements, each star will also be measured by two photometers, and a sizeable number of them will be measured using the Radial Velocity Spectrometer, which allows measuring the radial component of the three-dimensional velocity. This will lead to the most complete and accurate map of the stars of our Galaxy.

Continuous measurement of stellar sources using CCDs implies a special operation, different than the typical shutter-based imaging. Time Delayed Integration (TDI) is the best option in this case. It is based on a continuous charge shift from one pixel row to the next, synchronized with the satellite spin motion. It will be done in each CCD, thus accumulating the charge
during its corresponding integration period. In this way, long exposure times can be achieved with minimal distortion and blur. The total transit time of a stellar source will be about 87 seconds in the focal plane, during which 9 astrometric measurements (among others), each with an exposure time of 4.4 seconds will be performed.

With a predicted launch in early 2013, Gaia will orbit around the L2 Lagrange point, 1.5 million kilometers from the Earth opposite to the Sun. The satellite will be seen from the Cebreros ground station (Spain) for less than 10 hours per day in average. A second ground station in Perth (Australia) will be used just a few days during the entire mission, when the instruments will generate the largest amounts of data while scanning along the Galactic Plane. Nevertheless, during normal operation a permanent link between Gaia and the ground station will not be available.

1.2 Gaia Data Processing

Measuring more than 1 billion objects several times with the highest resolution implies a technological challenge, not only for the predicted 100 TB data base of compressed raw data (on ground), but also, and specially, for the on board data handling. With a total sum of almost one gigapixel, reading the whole set of CCDs entirely would imply a video output of more than 3.2 Gbps, so a selective sampling method is mandatory. This will be done by detecting and selecting the most interesting sources to measure, and then reading only windows (sets of pixels) around detected sources. For a typical case of single faint stars, the astrometric instrument will make bins of 12 pixels in the across-scan direction for obtaining a sample, that is, a single flux value. Afterwards, six contiguous samples will be read in the along-scan direction, thus offering a pattern or shape of the Point Spread Function (PSF) of the star, also named line spread function or LSF. Using this sampling method and the baseline technical features, the video data output rate of the astrometric CCDs may be reduced to less than 100 Mbps in the worst of the cases, namely, when a very dense stellar field is observed. Although this value may still appear large, typical communication standards like Spacewire can implement this in the payload data bus comfortably. Besides, this value includes the full readout of the sky mappers, from which another window around the detected star will be the only data to transmit. After all the on-board data selection procedures, the effective data to transmit will be reduced to some 12 Mbps in the worst of the cases, while the average value is about 7 Mbps.

1.2.1 Gaia Data Processing and Analysis Consortium (DPAC)

The Gaia Data Processing and Analysis Consortium (DPAC) is a European-wide effort including ESA, the Gaia Science Operations Centre (SOC) – established at the European Space Astronomy Centre (ESAC) in Madrid – and a science and engineering community of over 300 individuals, distributed on more than 15 countries. It also includes six large Data Processing Centers (DPCs) which are organizing themselves to process the data that will
arrive from the satellite. The Consortium has carefully estimated the effort required and has united in a single organization the material, financial and human resources, plus appropriate expertise, needed to conduct this processing to its completion in around 2020. The Consortium is organized into nine Coordination Units, each responsible for a well-defined set of tasks in the Gaia data processing effort. Below is a list of the Coordination Units (CUs):

- CU1: System Architecture
- CU2: Data Simulations
- CU3: Core Processing
- CU4: Object Processing, including multiple stars, Solar System objects and extended objects
- CU5: Photometric Processing
- CU6: Spectroscopic Processing
- CU7: Variability Processing
- CU8: Astrophysical Parameters
- CU9: Catalogue Access (not activated yet)

While each individual CU independently coordinates its own activities, the Consortium as a whole is coordinated by the DPAC Executive (DPACE), charged with handling issues and defining policies which concern all CUs. The DPACE is composed of the eight CU managers, a Chair and a Deputy Chair, and a representative from the CNES Data Processing Center. The development of the DPAC tasks is scheduled in cycles of six months, allowing for an adaptive development and easing the coordination between different CUs. At the time of writing this report DPAC is already in Cycle 10. About three full cycles still remain until a complete and tested implementation is available and delivered to ESA for operations. The simulation and core processing units (CU2 and CU3) are further described hereafter, which are the ones that concern our work. For more information regarding Gaia, DPAC or other CUs see Ref. [1] and [2].

1.2.2 Data Simulations (CU2)

The main task of the Data Simulations Unit (CU2) is to develop a software system capable of covering the simulation needs of the Gaia DPAC. For this, CU2 needs a strong software engineering base, capable of handling the development of such a complex system in a professional way. However, software engineering competence alone is not sufficient for the task ahead. Instead, a strong scientific component is also needed to ensure that the system fulfills the scientific needs of DPAC. CU2, as most CUs, is mapped into Development Units (DU), as depicted in Fig. 1.2, with a clear definition of responsibilities and interfaces.
DU5 is in charge of the development of the GASS data generator. The Gaia System Simulator (or GASS) is designed to simulate the telemetry stream of the Gaia mission according to the design specifications, using simplified models of astronomical objects and instruments. Such simplifications are needed to make possible the generation of large amounts of telemetry data in an acceptable time. In our case, GASS has generated the simulated telemetry datasets needed to test the solution presented in this work in a realistic manner. For more information on GASS, see Ref. [2].

1.2.3 Core Processing (CU3)

The Core Processing Unit (CU3) covers the main processing pipeline, from the ingestion (MIT) and initial data treatment (IDT) of the raw telemetry arriving from the spacecraft, to the astrometric core solution (AGIS), which means the determination of the 5 astrometric parameters for all sources – namely, sky coordinates, proper motions and parallax. It also covers the instruments health monitoring, the several calibrations and verifications, and the coordination of the relativistic modeling and tests relevant for Gaia. Several of the tasks of CU3 require very close interaction with the photometric, spectroscopic and object processing CUs, both for the algorithm development and for the actual processing. The boundaries of CU3 are defined mainly by the interfaces to those other CUs, and by the activities of CU1. In particular, software modules for most of the above-mentioned daily calibrations are provided by other CUs.

The Gaia data processing consists of several iterative processes dealing with astrometry, photometry and spectroscopy. These iterations reflect the fact that, as a survey satellite reaching accuracy and completeness levels never obtained before, it has to be self-calibrated. The processing pipeline of Gaia is very complex and deserves further comments. The following just aims to summarize the main steps and to highlight their impact on the overall DPAC organization. The reader is, however, referred to Ref. [1] for a more detailed description.
1.2.4 Initial Data Treatment (IDT)

The Initial Data Treatment transforms the most recently arrived telemetry flow into a more convenient form, determines basic image parameters, links the new observations with previous observations (sources) in the main data base, and derives various auxiliary data. The first part of this process consists in uncompressing, rearranging and reformatting the data to create raw information ready for storage in the main database. This part is not strictly data processing since there is no change or addition to the information content. Since its input is the telemetry stream, the IDT input data interface must be the same than the one used for telemetry data. Moreover, since GASS is the simulator in charge of the telemetry data generation, its output must also agree with such interface definition.

To insert the observations into the Main Data Base (MDB) they must be properly identified with sources already observed and entered into the MDB at an earlier time. This process is referred to as cross-matching. Observations that have no corresponding MDB source are entered as new sources. This phase needs to have access to a low-precision attitude, which will be based on a smoothed version of the onboard attitude, and the one-dimensional or two-dimensional centroiding for each observation, which is equivalent to the transit time of the center of the image for every source on every CCD. This is the fundamental astrometric measurement. At the same time an estimation of the source brightness or flux is made for the photometry. The transit times (centroids) are transformed into local geometric coordinates and preliminary (rough) sky coordinates, using the available calibrations and attitude. Transit times and fluxes for each of the observations (amounting to roughly $10^{12}$ at the end of the mission) will be updated every six months as better calibration files become available, these forming part of a process called Intermediate Data Updating (IDU) and briefly described hereafter. The result of IDT is the insertion of the observations into the main data base, properly linked to already identified sources in the sky.

1.2.5 Astrometric Core Solution (AGIS)

The Astrometric Global Iterative Solution (or AGIS) is the backbone of the data processing, since it provides the calibration and the attitude solution needed for most of the other treatments, in addition to the astrometric solution of about 100 million primary sources. The astrometric solution is carried out by successive iterations looking for a convergent solution. The system is described, along with some testing results, in Ref. [2].

1.2.6 Intermediate Data Updating (IDU)

There is a final process called Intermediate Data Updating that is in charge of refining the solution of the observations as reduced data from other CUs (astrometry, photometry and object processing) and better calibrations are available. This process essentially repeats the
main IDT algorithms using such reduced data and improved calibrations. It also runs some additional algorithms that determine improved calibrations of the response of the instrument.

IDU will have to re-process the large amount of raw data received from the satellite using a very complex set of algorithms. While IDT has to run in near-real time (which is already a challenge), IDU must process the raw data as quickly as possible in order to make possible the overall iterative processing of DPAC in a reasonable time. Due to its huge processing requirements, IDU will run at the Mare Nostrum supercomputer in Barcelona. The features of this data processing system imply large data transfers from the central disk to the hundreds of processing nodes and vice versa, as well as large data transfers between the nodes.

1.3 Objectives of this work

In parallel with the DPAC tasks, during 2007 a data compression study for the Gaia mission took place. It was called GOCA, which stands for Gaia Optimum Compression Algorithm, a contract between IEEC/GTD and ESA/ESTEC in the form of a Technology Research Programme (TRP). One of its results was the Prediction Error Coder (PEC), a highly optimized data compression algorithm that performs extremely well with the data produced by Gaia. With an adequate pre-processing stage, this algorithm can easily reach compression ratios better than 2 with extremely low processing requirements.

The main objective of this project is to apply the PEC compressor (with a suitable pre-processing stage) to the input data files for IDU – at least for the so-called AstroObservations (containing raw samples from the Sky Mapper and the Astrometric Field of Gaia), which represent the largest input for IDU. They can amount roughly 50 TB (uncompressed) towards the end of the mission. The goal is to decrease the necessary input/output load needed for IDU processing, while keeping negligible the computing overhead due to data compression and decompression. Data transfers between nodes should also benefit from this approach, as long as the AstroObservations are transferred as files – the current baseline. In order to do this as efficient and robust as possible, the HDF5 file format has been chosen for the data storage, which is a standard used in High-Performance Computing environments that allows concurrent input/output operations. The combination of HDF5 with PEC should allow very quick read operations (even with concurrent accesses to the file), while keeping the file size small and, thus, reducing the saturation of the central disk and network systems of MareNostrum.

This project is organized as follows. Chapter 2 describes the PEC compressor and the minor adaptations needed for this work. The Java Native Interface (JNI), needed for the integration of PEC with HDF5, is described in Chapter 3. Chapter 4 describes the HDF5 file format. The core of this work, namely, the application developed, is described in Chapter 5, while Chapter 6 shows the tests executed and the results obtained. Indications on how to use the resulting HDF5+PEC file are listed in Chapter 7. Finally, Chapter 8 summarizes the work, elaborates our conclusions and proposes some forthcoming work. The annexes give additional
information on the Rice entropy coder (used in the current standard for space data compression) and on the setup of the development environment.
2. The Prediction Error Coder (PEC)

2.1 Introduction

The scientific payload of Gaia will generate a complex set of data. The data compression solutions initially proposed for the mission [3] are not applicable when considering the flight-realistic limitations of the mission. For example, the available processing power on-board is not enough for applying standard compressors (such as zip) to a data stream of about 5 Mbps or more, which is a requirement for the mission. Even a quick algorithm like the CCSDS 121.0 recommendation for lossless data compression (from the Consultative Committee for Space Data Systems) requires an excessive percentage of the on-board processor capacity. Moreover, a software implementation is mandatory in this case, so the efficient hardware implementations of CCSDS 121.0 [4] become useless here. An alternative is to use the Rice coder alone (the core of the 121.0 recommendation), with pre-calibrated values of the $k$ parameter for each set of samples. Nevertheless, this solution is not reliable at all, because the value of $k$ must be chosen very carefully in order to obtain good compression ratios for a given set of data. And most importantly, a single outlier in the data can lead to large expansion ratios [5,6]. Additional information about the Rice coder and its limitations can be found in the Annex of this document.

2.2 Description of PEC

Rice codes are the adequate solution when the data to be compressed follow a geometric (or Laplacian) statistical distribution, which often arise after an adequate pre-processing stage [6]. However, any deviation from these statistics can lead to a significant decrease of the final compression ratio. This weakness is solved by the so-called Prediction Error Coder (PEC). PEC is focused on the compression of prediction errors, thereby its name. Hence, a pre-processing stage based on a data predictor plus a differentiator (outputting signed values) is needed. PEC is a very fast and robust compression algorithm that yields good ratios under nearly any situation.

PEC has three coding options, namely, Low Entropy (LE), Double-Smoothed (DS) and Large Coding (LC). LC makes use of unary prefix codes, while LE and DS rely on the “minus zero” feature of signed prediction errors – that is, an unused code that can be used as an escape sequence without requiring additional bits. As already mentioned, it is worth emphasizing that a pre-processing stage is required when using PEC, and that separate sign bits are used to avoid a mapping stage. In Figure 2.1 a schematic view of the PEC operation and its three coding strategies is shown.
The Prediction Error Coder (PEC) assumes that the data values are close to zero. When this is true, coding parameters must be chosen in a way that the first segments are significantly smaller than the original symbol size, while the last segments are slightly larger. This obviously leads to a compressed output, while the ratio will be determined by the probability density function of the data combined with the selected coding table. Another advantage is that PEC is flexible enough to process data distributions with probability peaks far from zero. With an adequate choice of its freely adjustable parameters, good compression ratios can still be reached when such distributions are processed. Finally, it is worth mentioning that PEC limits the maximum code length to twice the symbol size in the worst of the cases.

An adequate coding table and coding option must be selected for the operation of PEC. In order to easily determine the best configuration for each case, an automated PEC calibrator was developed, which just requires a representative histogram of the values to be coded. It analyzes such histogram and determines the optimum configuration of PEC. This is done by testing each of the possible PEC configurations of the histogram of values and selecting the one offering the highest compression ratio. In the case of space missions like Gaia, the calibrator must be run on-ground with simulated data prior to launch. PEC is robust enough to offer good compression ratios despite of variations in the statistics of the data. Nevertheless, the calibration should be repeated periodically during the mission, re-configuring PEC to guarantee the best results.

## 2.3 Modified version for HDF5

PEC has been adapted to perform best when used in combination with HDF5 for the IDU system. In particular, only the Large coding strategy has been considered. This can be considered as a simplification for the first prototype implementation. On the other hand, another modification has been introduced. Specifically, due to the massive number of zeroes

---

### Figure 2.1: Overview of the Prediction Error Coder (PEC) and its three coding strategies

<table>
<thead>
<tr>
<th>Segment</th>
<th>Low Entropy</th>
<th>Double-Smoothed</th>
<th>Large Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st range:</td>
<td>± X</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>2nd range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>3rd range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>4th range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Segment</th>
<th>Low Entropy</th>
<th>Double-Smoothed</th>
<th>Large Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st range:</td>
<td>± X</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>2nd range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>3rd range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
<tr>
<td>4th range:</td>
<td>- 0</td>
<td>f</td>
<td>l</td>
</tr>
</tbody>
</table>
found in the IDU data, we have implemented a special flag which is activated when a given number of consecutive zeroes are received in the compressor. After such flag, the number of zeroes received is coded. We call this version “IDU-Tailored PEC”. This version of the software has been coded in C language. Finally, we mention that a Java-compatible solution is necessary for being integrated in IDU. The implementation of such solution will be described in subsequent chapters, where we will explain how the PEC source was adapted to Java.
3. The Java Native Interface

As mentioned in the previous chapter, it was necessary to adapt the C implementation of PEC to the Java HDF5 project. There were two options to do it:

- Creating a new Java implementation, that is, converting the C PEC algorithm to Java and making the necessary corrections, taking special care of the bit-level input/output routines.
- Using the Java Native Interface (JNI) for directly calling the PEC algorithm from Java.

3.1 What is the JNI?

The Java Native Interface [7] is a powerful tool to execute code not programmed in Java but in other programming languages (such as C, C++ or assembler, that is, the native program), simply calling it in the same execution of a Java program (invocation). The JNI calls a native method, which calls implemented functions on native libraries, while the main program is implemented on the Java environment.

There is also the possibility to reverse the invocation and embed a JVM within the native code. This links to a native library that implements the JVM and executes code implemented with Java. However, this approach is not useful in our case.

3.2 Important implications

Using JNI means that two important benefits of the Java programming language will not be applicable to the project:

- Java applications that depend of the JNI cannot be run on multiple host environments, so the portability of Java is reduced. That is, the native library will have to be compiled for each of the environments where it shall run.
- A bad implementation of the native methods could corrupt the entire application, so stability, safety and security of Java is affected.
3.3 How JNI works

To execute a C-based code in a Java application it is necessary to follow these steps:

1. Create a class where the native method is declared.
2. Compile the Java program (with javac).
3. Obtain the .h file (the C header) from the Java class (with javah).
4. Write the implementation of the native method.
5. Compile the native method and obtain the system library (.dll in Windows, .so in Linux).
6. Execute the Java program using the system library.

Figure 3.1 illustrates these steps, which we have followed closely.

Fig. 3.1: Programming steps for a JNI-based “Hello World” program
4. The HDF5 file format

4.1 What is HDF?

The Hierarchical Data Format technology [8] is a library and a multi-object file format specifically designed to transfer large amounts of graphical, numerical or scientific data between computers. It was developed by the NCSA, but it is currently maintained by the HDF Group. This technology addresses problems of how to manage, preserve and allow maximum performance of data which has an enormous growth in size and complexity. It is developed and maintained as open source projects, so HDF is available and supported to users free of charge.

4.2 Which are the HDF Technologies?

At present, these technologies are included in two file format libraries (HDF4 and HDF5), a modular data editor, associated tools and utilities, and a conversion library. Both HDF4 and HDF5 were designed to be a general scientific format, adaptable to virtually any scientific or engineering application.

4.3 What is HDF5?

HDF5 [9] is a unique technology suite particularly good at dealing with data where complexity and scalability are important. Data of virtually any type or size can be stored in HDF5, including complex data structures and data types. Other features of HDF5 are:

- Portable: It can run on most operating systems and machines.
- Scalable: It works well in high-end computing environments, and can accommodate data objects of almost any size or multiplicity.
- Efficient: It provides a fast access to data, including parallel I/O. It can also store large amounts of data efficiently, since it has built-in compression, or applications can also provide their own special-purpose compression system.
- Open Source: the HDF5 data model, file format, API, library, and tools are open and distributed free of charge.

4.4 Using the HDF5 data model

HDF5 implements a model for managing and storing data:
- It includes a data model, an abstract storage data model and libraries to implement it and to map it with different storage mechanisms.
- It provides a programming interface for a correct implementation of the abstract models.
- It implements a model of data transfer and an efficient data movement.

The HDF5 Library calls the Operating System or other Storage Management software to store and retrieve persistent data. The HDF5 Library may also link to other software, such as filters for compression. The HDF5 Library is linked to an application program, which may be written in C, C++, Fortran 90 or Java. The application program implements problem specific algorithms and data structures, and calls the HDF5 Library to store and retrieve data. Figure 4.1 shows the different modules which are part of HDF5 format.

![Fig. 4.1: HDF5 Abstract Model Modules](image)

The Abstract Data Model is the conceptual model of data. It includes the data types used and the data organization, and it is independent of the storage media or the programming environment. The Storage Model is the representation of objects in the Abstract Data Model. Finally, the Programming Model is the model of the computing environment, which includes many platforms, from small single systems to large multiprocessors and clusters. This module manipulates, instantiates, populates, and retrieves objects from the Abstract Data Model.

The HDF5 Library is a C module that implements the Programming Model and Abstract Data Model. Figure 4.2 shows the dependencies of these modules.
The Stored Data is the actual implementation of the Storage Model. The Storage Model is mapped to several storage mechanisms, including single disk files, multiple files (family of files), and memory representations. Finally, the HDF5 API represents all the programming functions provided with HDF5, which are called by the Library when necessary.

4.5 The Abstract Data Model

The HDF5 Abstract Data Model includes concepts for defining and describing complex data stored in files. It is a very general model which is designed to cover many specific data models. Many different kinds of data can be mapped to objects of the HDF5 ADM, and therefore be stored and retrieved using HDF5. The ADM is not a model of any particular problem or application, as the final users need to map their data to the concepts of the ADM. The key concepts of the HDF5 are described below.

4.5.1 File

An HDF5 File is a container for an organized collection of objects. The objects are groups, datasets and other objects as defined below. The objects are organized as a rooted, directed graph. Every HDF5 file has at least one object, the root group. All objects are members of the root group or descendents of the root group. In other words, the HDF5 data files are very similar to the UNIX file system: groups are like folders, and datasets are like files. There is always the primary group, called root, and all of the data is stored as datasets or included in another folder/group. For example:

```
/           root group.
/foo/       foo group inside root group.
/foo/zoo    zoo dataset inside foo group, inside root group.
/foo/woo/   woo group inside foo group, inside root group.
```
HDF5 objects have a unique identity within a single HDF5 file, and can be accessed only by its names within the hierarchy of the file. HDF5 objects in different files do not necessarily have unique identities. When the file is created, the File Creation Properties specify settings for the file. They include version information and parameters of global data structures. When the file is opened, the File Access Properties specify settings for the current access to the file. File Access Properties include parameters for storage drivers and parameters for caching and garbage collection. The File Creation Properties are permanent for the life of the file, but the File Access Properties can be changed by closing and reopening the file.

### 4.5.2 Groups

An HDF5 Group is analogous to a UNIX file system directory.

- A group can contain zero or more objects (except for the root group).
- Every object must be a member at least one group (except the root group).

The root group is a special case, it may not be a member of any other group, and if it contains zero objects, the HDF5 file is empty.

### 4.5.3 Datasets

An HDF5 Dataset is a multidimensional (32 maximum) array of Data Elements. The shape of the array (number of dimensions, size of each dimension) is described by the Dataspace object (see below). A Data Element is a single unit of data which may be a number, a character, an array of numbers or characters, or a record of heterogeneous data elements. A Data Element is a set of bits; the layout of the bits is described by the Datatype (see below).

The Dataspace and Datatype are set when the Dataset is created, but they cannot be changed for the life of the Dataset. The Dataset Creation Properties are set when the Dataset is created. They also include the fill value and storage properties such as chunking and compression. These properties cannot be changed after the Dataset is created. The Dataset object manages the storage and access to the data. While the data is conceptually a contiguous rectangular array, it is physically stored and transferred in different ways depending on the storage properties and the storage mechanism used.

### 4.5.4 Dataspace

The HDF5 Dataspace describes the layout of the elements of a multidimensional array. Conceptually, the array is a hyper-rectangle with one to 32 dimensions. HDF5 Dataspaces can be extendable. Therefore, each dimension has a current and maximum size, but the
maximum size can be unlimited. The Dataspace describes this hyper-rectangle: it is a list of dimensions, with the current and maximum (or unlimited) size:

![Dataspace properties](image)

Dataspace objects are also used to describe hyperslab selections from a dataset. Any subset of the elements of a Dataset can be selected for read or write by specifying a set of hyperslabs. A non-rectangular region can be selected by the union of several (rectangular) Dataspaces.

### 4.5.5 Datatype

The HDF5 Datatype object describes the layout of a single data element. A data element is a single element of the array. It can be a single number, a character, or an array of numbers or other kind of data. The Datatype object describes the storage layout of this data. There are 11 classes of Datatypes, organized in 2 types:

- **Atomic Datatypes**: These datatypes are indivisible; they must be a single object: a number, a string, etc. Atomic Datatypes are:
  - Time: Time references.
  - Bitfield: String of bits.
  - String: Array of 1-byte character encoding.
  - Reference: Reference to object or region within the HDF5 file.
  - Opaque: Uninterpreted data.
  - Integer: Twos complement integers.
  - Float: Floating Point numbers.

- **Compound Datatypes**: They are composed of multiple elements of atomic Datatypes:
  - Enumeration: A list of discrete values, with symbolic names in the form of strings.
  - Array: Array (1-4 dimensions) of data elements.
  - Variable Length: A variable length, 1-dimensional array of data elements.
  - Compound: A Datatype composed of a sequence of Datatypes.

In addition to these standard types, users can define additional Datatypes. A Datatype object can be stored in the HDF5 file. The Datatype is linked into a group, and therefore given a name. A Named Datatype can be opened and used in any way that a Datatype object can be used.
4.5.6 Attribute

Any HDF5 Named Data Object (Group, Dataset, or Named Datatype) may have zero or more user defined Attributes. Attributes are used to document the object. The Attributes of an object are stored with the object. An HDF5 Attribute has a name and data. The data is described analogously to the Dataset: the Dataspaces defines the layout of an array of Data Elements, and the Datatype defines the storage layout and interpretation of the elements.

An Attribute is very similar to a Dataset with the following limitations:

- An attribute can only be accessed via the object; attribute names are significant only within the object. Attributes cannot be shared.
- For practical reasons, an Attribute should be a small object (no more than 1000 bytes).
- The data of an Attribute must be read or written in a single access (selection is not allowed).
- Attributes do not have Attributes.

There are many more features that can be looked up in the HDF5 user’s guide. However, those described so far are enough to understand why we use HDF5 to compress and store the Gaia AstroObservations.

4.6 HDF5 in this project

In the present study we have programmed using a Windows XP 32-bits system, using the Eclipse Europa Platform (Version 3.3.2) as the programming interface and, because the Gaia DPAC software is developed in Java, we used this programming language. In particular, the latest version of the Java Developement Kit was used. The version used here is 6 Update 13, which can be found in the Sun Microsystem’s Java webpage:

http://java.sun.com/javase/downloads/index.jsp

The latest release of the Eclipse Europa Platform was downloaded from the Eclipse Project’s main page:


4.7 Example of a HDF5 File

In this section, the features of the HDF5 will be shown using an example file. We use HDFView from the HDF Group:
This software can open both HDF4 or HDF5 files and shows their content in a graphical user interface. It can be downloaded from


Once installed, it can be tested using a demo file:

The program shows the group hierarchy on the left, from which each dataset can be opened in a way similar to that of the windows folder explorer. A window with the content of the dataset will be opened. Depending of the type of data stored in the dataset, we will obtain one or another window interface. For example:
The HDF5 file format

The previous picture shows a jpeg image stored in the HDF5 file. The program opens a new window with several tools according to the type of data. Additionally, in this snapshot the several bars or sub-windows with the HDF5 file information and access tools can be seen. Starting with the left bar, it shows the contents (groups and datasets) stored in the HDF5 file. Three distinct groups (arrays, datatypes and images) and several datasets inside of them, classified by type in each group can be seen. It is also possible to see a text dataset inside the root group. Double-click on them opens, like in the Windows File Explorer the corresponding action.
Continuing with the lower bar, when a dataset is opened, the characteristics and HDF5 Attributes of the Dataset can be seen. Specifically, the name of the Dataset, the data type, the pixel size, and the number of attributes are listed, as shown in Fig. 4.8.

![Fig. 4.8: Information Bar](image)

On the upper part of the dataset window see some useful tools for displaying the dataset are available. In the case of a picture, the tools are shown in Fig. 4.9.

![Fig. 4.9: Image Tools](image)

They include bright control and zoom buttons, which do not appear in a table dataset, for example, where we only have a plot button:

![Fig. 4.10: Table example](image)

It is also interesting to have a look of the HDF5 File Properties that we can get when right-clicking on the file in Windows:
Fig. 4.11: File Properties

Fig. 4.11 illustrates the power of this file format. All the data is completely indexed and stored in just 1,69 MB.
5. The \textit{GbinToHdf5} Application

5.1 The GBin file format

First of all, we must know and understand what kind of data we have to deal with, in order to define the HDF5 model and store all the fields in a HDF5 file. One of the tools available for Gaia DPAC is the so-called Gaia Main Database Dictionary Tool, a snapshot of which can be seen in Figure 5.1. This tool allows the user to navigate through the complex Data Model of the Gaia DPAC systems.

![Fig. 5.1: Snapshot of the Gaia Main DB Dictionary Tool, illustrating the AstroObservations data model](image)

The table we are most interested in is the AstroObservation table, which holds the Sky Mapper (SM) and Astrometric Field (AF) samples, which represents the major input to the IDU system. An AstroObservation, in its definition of cycle 6 (following the DPAC nomenclature), is a group of the following set of data:

- \texttt{solutionId}: Solution identifier, helping with the versioning of the MDB.
- \texttt{creationDate}: Creation date of the data.
- \texttt{transitId}: Unique transit identifier, including the observation time among other attributes.
- \texttt{resRefTime}: Residual (high-resolution) observation time.
- \texttt{healPixFov}: Spatial index for the Field Of View (FOV) of the observation at detection time, based on HealPix.
- \texttt{healPixOpp}: Spatial index of the opposite FOV.
- \texttt{gClass}: Window class used in SM, AF and XP, indicating the resolution of the measurement.
- **objectType**: Object type as determined by the on-board instrument.
- **gPriority**: Priority of the observation for its transmission from Gaia.
- **acqFlags**: Acquisition flags, indicating truncation of acquisition windows, availability of BP/RP data, etc.
- **af1Centring**: Along-scan position of the AF1 window with respect to the nominal position.
- **xpOffset**: Along-scan position of BP/RP windows within a reference macrosample.
- **gMag**: G magnitude (brightness) estimated by the on-board instrument (the VPU).
- **af1Gate**: Gate information for AF1, indicating a reduction in the integration time of samples.
- **af29Gate**: Gate information for AF2-9.
- **af29GateRelease**: Gate release information for AF2-9, indicating if some sample may be contaminated by residual charge.
- **acWinCoord**: Across-scan coordinate of transmitted SM and AF windows.
- **alWinCoord**: Along-scan coordinate of transmitted SM and AF windows.
- **aocsUpdate**: Information on across-scan attitude updates occurred in AF2-9, revealing if some windows are not rectangular.
- **shapeAf**: Shape (truncation) information for AF2-9 windows.
- **activeCtr**: Active Centring information for AF2-9 windows.
- **smSamples**: Samples from the Sky Mapper, each from 2x2 or 4x4 pixels depending on the sampling scheme.
- **afSamples**: Samples from the Astrometric Fields (AFs), with 6x1 to 18x12 samples each depending on the sampling scheme.
- **distToLastCi**: Distance (in TDI periods) to the last Charge Injection.

The detailed meaning of each of these fields is beyond the scope of this work and can be found in the adequate Gaia documentation [2].

Each of the above mentioned fields has a different size and multiplicity, as indicated in the following table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Total bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SolutionId</td>
<td>long</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>creationDate</td>
<td>double</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>transitId</td>
<td>long</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>resRefTime</td>
<td>int</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>healPixFov</td>
<td>int</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>healPixOpp</td>
<td>int</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>gClass</td>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>objectType</td>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gPriority</td>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>acqFlags</td>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>af1Centring</td>
<td>byte</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
All these data fields are stored in the so-called GBin file format, which stands for Gaia Binary. It is simply a Java serialized object with Zip compression, which is very limited in terms of indexing capabilities, although the access through Gaia Java applications is rather good. Also, due to the fact of having Zip compression, it will be a very good reference in terms of file size. That is, our goal in this work is to obtain file sizes similar to the original GBin ones—which would mean that we are able to level Zip compression with PEC.

### 5.2 Classifying data

Due to the different nature of the several AstroObservation fields, the best option is to create one dataset for all the fields having a fixed length (which we will call the header), plus two datasets with variable lengths for the SM and AF samples, storing all the data types independently. First, enough space for the header must be allocated. Figure 5.2 shows the sizes to be considered. For the adequate decompression of SM and AF samples, the lengths of the SM and AF arrays are also stored in header.

```plaintext
/* SCALAR VECTOR LENGTHS */
--- ------ ----- -----
2x long = 2x8 = 16 byte[10] = 10x1 = 10 int[4] = 4x1 = 4
1x double = 1x8 = 8 short[10] = 10x2 = 20 int[4] = 4x1 = 4
5x int = 5x4 = 20 byte[3] = 3x1 = 3
1x short = 1x2 = 2 byte[4] = 4x1 = 4
5x byte = 5x1 = 5 short[3] = 9x1 = 9
SUBTOTAL = 47 SUBTOTAL = 60 SUBTOTAL = 6
--- -----------------------------------------------
TOTAL HEADER = 115 bytes
*/
```

Fig. 5.2: Calculating the header space.

\(^1\) Not implemented yet.
5.3 Structure of the HDF5 File for AstroObservations

As explained before, all data from the AstroObservations GBin Files will be stored by type on the HDF5 File. The main issue is to control the number of indexes in the HDF5. The final size of the file strongly depends on them. Storing all AstroObservations in only one single dataset increases the final HDF5 File size due to the larger indexes needed. To avoid this, AstroObservations from the GBin file are split into groups of 400, each with the 3 datasets mentioned (header, AfSamples and SmSamples) inside of each group. More details about the split can be found in Annex 9.6.

5.4 GBin File Read and HDF5 File Write

The procedure for bringing the data from GBin to HDF5 is very simple, as we just have to read the data using the routines available from the DPAC GaiaTools library, reading AstroObservation by AstroObservation, field by field. The following simple sentence reads sequentially from a previously opened AstroObservation file:

```c
if (gt.next()) ao = astrobsFact.getObject(gt);
```

Once the object with all the contents of one observation are read, the data fields can be directly accessed, manipulated as necessary and finally stored in the adequate destination.

Data in a HDF5 file is stored in tables. As previously mentioned, groups of 400 (NUM_ELEMENTS) AstroObservations are made, but it is not possible to store the complete table at once. Instead, the correct procedure is the following:

1. Read all the fields from the AstroObservation.
2. Copy them into a temporary HDF5 Dataset which has the same size.
3. Use hyperslab selection feature of HDF5 to select the temporary dataset and the adequate row.
4. Copy contents.

The hyperslab selection is a unique feature of HDF5 which allows manipulating a part of the dataset without modifying the rest of the table. This also allows concurrent write access from several processes. The selection is always a rectangular-shaped form, although several selections with the same dimensions can be combined. This selection requires 4 parameters to be set:

- start: vector that indicates the start position to select in the table.
- block: size of the dataspace for one selection. If this is NULL, it assumes just one element per dimension.
- count: number of blocks to select along each dimension.
- stride: number of elements in each dimension between the first element of each block. If this is set to NULL, there is only selected one block.

An example can be seen in Figure 5.3, where the crosses illustrate a hyperslab selection with the following parameters:

- start = (0,1)
- block = (3,2)
- count = (2,4)
- stride = (4,3)

Fig. 5.3: Hyperslab selection example

Fortunately, the hyperslab selection required in this project is simpler than that shown in the example. In our case, we just need start and count:

```c
H5.H5Sselect_hyperslab(DS,HDF5Constants.H5S_SELECT_SET,start,null,count,null);
start[0] = i;
H5.H5Sselect_hyperslab(datas[0][1],HDF5Constants.H5S_SELECT_SET,start,null, count,null);
```

The first sentence marks the temporary dataset, whereas the second one marks the definitive one. The only difference is the starting point:
Finally, the data is copied to its final location. The temporary dataset remains until the HDF5 file is completed and the program ends.

### 5.5 Compound dataset for header

Most fields of the AstroObservation are going to be stored into a single dataset. This means that this dataset cannot be a single-type or common type dataset. Instead, it has to be a Compound dataset of 400 rows (one per AstroObservation per group), and a total of 59 columns:

- One per Scalar Field (SolutionID, CreationDate, TransitID, ResRefTime, HealPixFov, HealPixOpp, GClass, ObjectType, GPriority, AcqFlags, AFlCentring, XpOffset, GMag, AFlGate, Af29Gate and Af29GateRelease). This is, 16 scalar types.
- One per each vector dimension for Vector Fields with fixed length (AcWinCoord x 10, AlWinCoord x 10, AocsUpdate x 8, ActiveCtr x 4, DistToLastCi x 9). That is, 41 values.
- One per AF and SM sample array length. That is, 2 additional values.

These 59 columns have a total length of 115 bytes, being the length per field as calculated before. The lengths for AF and SM sample arrays will be stored as ints. Now the main issue here is how to create this compound datatype. First of all, it is necessary to allocate the necessary space:

```c
//Create the Dataspace
datasp[0][1] = H5.H5Screeate_simple (RANK, dim, null); 
```

where RANK is 2 (the dimension of a table) and dim is the vector that indicates the length per dimension. The latter indicates just one column – as we will store AstroObservations independently, concatenating them one under another in a matrix – and the necessary number of elements as rows – which is 400 except the last group, which may hold a smaller number of AstroObservations until the end of the file. This operation will be made for each group to be created in the HDF5 File. Finally, the datatype in itself is created (as a Compound), and afterwards we add the name, size and position of each of the data fields within the 115 bytes of the compound:

```c
//create the Memory Data type
datatype_id = H5.H5Screeate(HDF5Constants.HST_COMPOUNO, LENGTH);
H5.H5Screeate (datatype_id, "ObjectType", 37, H5.J2C (HDF5Constants.HST_NA TIVE_ INT8));
```
Feasibility study of the PEC compressor in HDF5 file format

Then the dataset is created with this sentence:

```
//Creation of the compound dataset
datas[0][0] = H5.H5Dcreate (file_id, "/G0" + (elmsCounter - 1) + "/HEADER", datatype_id, datas[0][1], HDF5Constants.H5P_DEFAULT);
```

Finally, we access the GBin file contents and copy its fields to the dataset making a conversion to byte. For example:

Read:  
```
SolutionID[0] = ao.getSolutionId();
```

Conversion:  
```
SolutionIDec = HDFNativeData.longToByte(0,1,SolutionID);
```
Write: `System.arraycopy(SolutionIDec, 0, datal, 0, 8);`

When we are done, the dataset is closed. Then, we create a new one in another group, and the storing operation is repeated until all AstroObservations of the GBin file are read and stored.

### 5.6. Storing SM and AF samples

These two fields represent the largest amount of data in either the GBin or HDF5 files, so they will be compressed using our Tailored PEC for reducing the necessary space. Before that, some considerations must be taken into account. Firstly, before creating a dataset we need to know its exact dimensions. When compressing data, the resulting size cannot be known beforehand. Thus, we have to execute the compressor twice: first for just obtaining the total lengths of the compressed AF and SM samples, and second, once the dataset is created, for filling it with the actually compressed data. This is obviously a very simplistic approach that could be optimized by storing the compressed PEC output in a temporary array and finally copying it to its final destination. Nevertheless, for this prototype we will follow this simplistic approach. Secondly, all the AF and SM samples of all the AstroObservations in a group are stored together, since we can clearly retrieve them owing to their array lengths stored separately. Every AF and SM array is read, compressed and concatenated one after another in the vector. Thus, we have three datasets per group: HEADER (59 fields x 400 elements, or less if it is the last group), AFsamples (one row with a variable number of elements), and SMsamples (one row with a variable number of elements). Finally, to use the PEC compressor we have to compile its native code adequately, depending on the computer platform where we intend to run. In the case of Windows, the best option is to use Microsoft Visual C++ 2008 Express Edition [10], which can be downloaded for free from the Microsoft webpage:

[link](http://www.microsoft.com/express/Downloads/#2008-Visual-CPP)

Once installed, we only need the command prompt for Visual C++, available from the Start Menu Programs Microsoft Visual C++ 2008 Express Edition Command Prompt for Visual Studio 2008. With this tool, we will be able to compile the C code and to generate the required dll library for JNI by issuing the following command:

```
cl -I "C:\Archivos de Programa\Java\jdk1.6.0_12\include" -I "C:\Archivos de Programa\Java\jdk1.6.0_12\include\win32" /LD libPecCompressor.c
```

Options are:

- `cl`: Compiler of C/C++.
- `-I`: Include. This is used to locate necessary files to work with Java.
- `/LD`: Generate System Library.
Finally, the library libPecCompressor.dll file can be either copied into the Java main Path (C:\Archivos de programa\Java\jre6\bin in a Spanish Windows), included in the command-line when invoking our program, or linked from the Eclipse environment. With this, the project is ready to be executed.
6. Obtained HDF5 File

6.1 Format

The format of the HDF5 File is as expected, that is, in agreement with the design fixed during the development stage:

- From the root group, there are one or more main groups named G00, G01, G02, etc. Each of these groups contains 400 AstroObservations (except the last one).
- Each of these groups has 3 datasets inside: HEADER, AFSamples and SMSamples.
- The HEADER dataset has the scalar and fixed length vector fields in a table.
- The AFsamples and SMsamples datasets have only one row each.

Figures 6.1 and 6.2 illustrate the contents of an AstroObservation HDF5 file, opened with the HDFView utility.

![TableView - HEADER](image)

*Fig. 6.1: Header dataset of an AstroObservation HDF5 file*
During the development of this project, 8 GBIN files were tested and converted. They come from a “Cycle 6” GASS dataset (in the Gaia DPAC nomenclature), processed by the Initial Data Treatment (IDT) system – which actually generated the AstroObservation files from the simulated telemetry data.

For each file we need to calibrate adequately the PEC compressor in order to get the best compression ratio. The following table shows the sizes and coding tables for each GBIN file, as well as the obtained HDF5 File size. It also shows the so-called “G classes” of the AstroObservations contained in the file. Class 0 corresponds to the brightest stars, whereas class 3 corresponds to the faintest ones:

<table>
<thead>
<tr>
<th>File ID</th>
<th>G classes in file</th>
<th>GBIN Size</th>
<th>HDF5 Size</th>
<th>PEC Calibration Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>171</td>
<td>0, 1</td>
<td>2.91 MB</td>
<td>2.40 MB</td>
<td>(3,6,10)</td>
</tr>
<tr>
<td>172</td>
<td>1, 2</td>
<td>2.55 MB</td>
<td>2.66 MB</td>
<td>(4,8,12)</td>
</tr>
<tr>
<td>173</td>
<td>2, 3</td>
<td>440 KB</td>
<td>553 KB</td>
<td>(4,8,12)</td>
</tr>
<tr>
<td>174</td>
<td>3</td>
<td>423 KB</td>
<td>537 KB</td>
<td>(4,6,10)</td>
</tr>
<tr>
<td>175</td>
<td>3</td>
<td>314 KB</td>
<td>432 KB</td>
<td>(4,6,10)</td>
</tr>
<tr>
<td>176</td>
<td>3</td>
<td>223 KB</td>
<td>330 KB</td>
<td>(3,4,8)</td>
</tr>
<tr>
<td>177</td>
<td>3</td>
<td>176 KB</td>
<td>269 KB</td>
<td>(3,4,6)</td>
</tr>
<tr>
<td>178</td>
<td>3</td>
<td>9.47 MB</td>
<td>11.9 MB</td>
<td>(3,4,9)</td>
</tr>
</tbody>
</table>
6.3 Processing Time

One of the main goals of the HDF5 format is to allow for an efficient data access, with small read/write times. Thus, a mandatory test for the application developed here is to compare how much time is used by the processor to access both file formats, namely, GBIN and HDF5. The test was done using the largest file (with the identifier 178). We have run the Converter and Decoder programs over a variety of platforms, in order to obtain conclusive results. To get coherent results and minimize the effects of other processes that may be running in the system, we have repeated the test 5 times in each platform. These are the performance results (in seconds) for each platform:

Platform 1: Windows 7 64-bits, Intel Core i5-2400 (3.1 GHz)

<table>
<thead>
<tr>
<th></th>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Time (HDF5)</td>
<td>4.088</td>
<td>4.169</td>
<td>3.996</td>
<td>4.486</td>
<td>4.268</td>
<td>4.194</td>
</tr>
<tr>
<td>Compression Time</td>
<td>1.449</td>
<td>1.337</td>
<td>1.152</td>
<td>0.997</td>
<td>1.218</td>
<td>1.279</td>
</tr>
<tr>
<td>Read Time (HDF5)</td>
<td>4.446</td>
<td>4.165</td>
<td>4.197</td>
<td>4.181</td>
<td>4.212</td>
<td>4.237</td>
</tr>
</tbody>
</table>

Platform 2: Linux Gentoo 64-bit, Intel Core i5-2400 (3.1GHz)

<table>
<thead>
<tr>
<th></th>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Time (HDF5)</td>
<td>2.466</td>
<td>2.510</td>
<td>2.461</td>
<td>2.439</td>
<td>2.557</td>
<td>2.486</td>
</tr>
<tr>
<td>Compression Time</td>
<td>1.199</td>
<td>1.158</td>
<td>1.184</td>
<td>1.204</td>
<td>1.170</td>
<td>1.183</td>
</tr>
</tbody>
</table>

Platform 3: Windows XP 32-bit, Intel Core2 Duo T7100 (1.8GHz)

<table>
<thead>
<tr>
<th></th>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression Time</td>
<td>5.042</td>
<td>5.344</td>
<td>5.036</td>
<td>5.214</td>
<td>5.270</td>
<td>5.178</td>
</tr>
<tr>
<td>Read Time (HDF5)</td>
<td>11.718</td>
<td>11.656</td>
<td>11.735</td>
<td>11.484</td>
<td>11.672</td>
<td>11.652</td>
</tr>
</tbody>
</table>
Platform 4: Linux Fedora 32-bit, Intel Core2 Duo 6400 (2.13GHz)

<table>
<thead>
<tr>
<th></th>
<th>TEST 1</th>
<th>TEST 2</th>
<th>TEST 3</th>
<th>TEST 4</th>
<th>TEST 5</th>
<th>MEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Time (HDF5)</td>
<td>8.389</td>
<td>8.326</td>
<td>8.412</td>
<td>8.357</td>
<td>8.313</td>
<td>8.359</td>
</tr>
<tr>
<td>Compression Time</td>
<td>4.213</td>
<td>4.562</td>
<td>4.369</td>
<td>4.311</td>
<td>4.322</td>
<td>4.352</td>
</tr>
</tbody>
</table>

6.4 Statistics

It is also interesting to check the contents of each file when converting from GBIN to HDF5. Every file contains a different kind of data, mostly indicated by the G Class of the AstroObservations – which have a strong influence in the contents and sizes of the SM and AF samples. The final statistics for all the files are shown below:

mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.171.gbin
Number of AstroObservations in file = 927
Number of AstroObs per group = 400
Number of Groups = 3
GClass = 0 --> 175 (18.878101%)
GClass = 1 --> 752 (81.121895%)
GClass = 2 --> 0 (0.0%)
GClass = 3 --> 0 (0.0%)

mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.172.gbin
Number of AstroObservations in file = 6730
Number of AstroObs per group = 400
Number of Groups = 17
GClass = 0 --> 0 (0.0%)
GClass = 1 --> 8 (0.11887073%)
GClass = 2 --> 6722 (99.881134%)
GClass = 3 --> 0 (0.0%)

mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.173.gbin
Number of AstroObservations in file = 1652
Number of AstroObs per group = 400
Number of Groups = 5
GClass = 0 --> 0 (0.0%)
GClass = 1 --> 0 (0.0%)
GClass = 2 --> 36 (2.1791768%)
GClass = 3 --> 1616 (97.82082%)

mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.174.gbin
Number of AstroObservations in file = 1710
Number of AstroObs per group = 400
Number of Groups = 5
GClass = 0 --> 0 (0.0%)
GClass = 1 --> 0 (0.0%)
GClass = 2 --> 0 (0.0%)
GClass = 3 --> 1710 (100.0%)

mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.175.gbin
Number of AstroObservations in file = 1387
Number of AstroObs per group = 400
Number of Groups = 4
GClass = 0 --> 0 (0.0%)
6.5 Interpretation of results

First of all, it is important to note that this application is just an initial prototype. There are large margins of improvement, either in the pre-processing and compression of the samples, in the execution of the conversion (and thus in the performance of the application), and also in the actual data model definition for the HDF5 AstroObservations format. Nevertheless, as already shown, the inclusion of the PEC compressor in HDF5 is completely feasible, as well as the storage of Gaia AstroObservations in such format.

Section 6.2 shows that the resulting HDF5 files are similar, yet typically larger, than the original GBIN files. It can be easily seen that the GC classes 0 and 1 are better compressed than classes 2 or 3. Better compression results might be achieved with an adaptive layer in the compressor, or even by simply using a more elaborated pre-processing stage. It is worth noting that the HDF5 file could also be compressed in ZIP or RAR, so the file size could be further reduced – although we would then lose the efficient access that the HDF5 format offers. Chapter 8 summarizes the possible improvements envisaged for this application.

The processing time, shown in Section 6.3, is definitely the most important achievement of HDF5+PEC over GBIN. In all the platforms the reading time is significantly faster when the combination HDF5+PEC is adopted than when GBIN is used. The worst case is the 32-bit Linux, where our application is 25% faster than the GBIN Gaia library, whereas in the 64-bit Linux case we double the GBIN performance. It is worth noting that such platform (64-bit
Obtained HDF5 File

Linux) is the target of our application, as this is the Operative System of the MareNostrum supercomputer – where the Gaia IDU system will run during operations. Regarding the writing performance, our application also offers excellent results, being such operation always quicker than the GBIN reading – although writing is a typically a more costly operation than reading.

We should also notice other details of these results. Both 64-bits tests have been executed in the same computer, but it is clear that the results are better in Linux than in Windows. It is most probably caused by a more efficient implementation of the HDF5 libraries in such platform. Also, the 64-bits version of the application obviously provides a much better performance than the 32-bits versions, although in this case it is more difficult to compare due to the different computers where the tests have run.

As a final remark, it must be noted that the format of the file has been defined as simple as possible. Only AF and SM samples are compressed with PEC in this implementation. Other solutions combining the data, classifying AstroObservations or even compressing other fields could be considered. For example, the header dataset could also be compressed with PEC, or the several fields of the AstroObservations could be stored in a more efficient manner in the HDF5 file. In any case, the characteristics of the HDF5 file – mainly the size and the access time – must be considered. Also the GClass plays an important role in the storage and compression of the obtained file. GClass is closely correlated with the contents and sizes of the other fields, so other compression scenarios are possible. Thus, several alternative HDF5 formats could be considered, studied and optimized.
7. Obtaining data from the HDF5 File

An essential part in the application development is to provide to the final users with an easy interface which will allow obtaining all the data stored in the HDF5 files. Several different options could be tackled, but when considering the profile of the final user, we decided to provide the same kind of functions typically given by DPAC libraries – but in the HDF5 version. The program developed was named “Decoder”. These implemented functions in Decoder are:

- **static int openFile (String filename)**: Opens an existing HDF5 files in read-only mode. The Decoder program does not need to write on the HDF5 file, so this access mode guarantees the faster way to obtain the data. This method needs the file path as a String and returns the file identifier.

- **static void closeFile (int ID)**: Closes a previously opened HDF5 file, passing the file identifier to the method.

- **static int getNumRows (int ID)**: Returns the Number of AstroObservations stored in the file. This is a very useful function for testing or control purposes. This value is stored as a small dataset (int) in the HDF5 File, as indicated in previous sections.

- **static int getNumElements (int ID)**: Returns the Number of Elements stored per group. This parameter can be different in each HDF5 File, but we consider 400 as the best option in this development (see Annex 9.6).

- **static void close()**: Resets the number of AstroObservation read as -1, so with this method it is possible to restart the reading of the file without closing and reopen it.

- **static boolean next()**: This method increases the current reading index in 1 unit and, if this is done successfully (i.e., if there are still further records in the file), returns true. If the Number of Elements of the group has been reached, then it automatically points to the next group. If the last AstroObservation stored in the HDF5 file has been read already, it then returns false.

- The scalar fields and the arrays with fixed length are read only once per group. A limitation of HDF5 is that it does not allow to obtain only one element of a stored array, so when the group is opened the first time, all NUM_ELEMENTS² field (for example, all SolutionID stored in the first group) are read and stored in a temporary memory table. While the current read index does not point to the last AstroObservation stored in the group, the program just returns the necessary value of the array. In any other case, all the arrays of the group are read and the temporary table is filled again. These are the methods³:
  - **static long getSolutionID()**
  - **static double getCreationDate()**
  - **static double getTransitID()**
  - **static int getResRefTime()**

² NUM_ELEMENTS = 400 on the present development.
³ The method for returning ShapeAf field was not implemented since this field was NULL in all the GBIN files provided.
- Obtaining data from the HDF5 File

- static int getHealPixFov()
- static int getHealPixOpp()
- static byte getGClass()
- static byte getObjectType()
- static byte getGPriority()
- static byte getAcqFlags()
- static byte getAf1Centring()
- static byte getXpOffset()
- static short getGMag()
- static byte getAf1Gate()
- static byte getAf29Gate()
- static byte getAf29GateRelease()
- static byte[] getAcWinCoord()
- static short[] getAlWinCoord()
- static byte[] getAocsUpdate()
- static byte[] getActiveCtr()
- static short[] getDistToLastCi()

- Something similar happens to the non-constant array fields. As in the previous case, the first time the group is read, all data from these fields (in this case, all the SMSamples and AFSamples) is read; then, the PEC Compressor is started, and a temporary table is created and filled for both fields. The lengths information stored in the HEADER dataset is also read, because it is necessary to uncompress. This means that there is a performance penalty when the program has to show the first SMsample or AFSample on the first AstroObservation of the group, whereas for other AstroObservations it will be quicker – as the PEC compressor has to uncompress all the NUM_ELEMENTS AstroObservations the first time the group is accessed, whereas for the other AstroObservations the program only shows the necessary row of the temporary table. These functions are:
  - static int[] getSmSamples()
  - static int[] getAfSamples()

All these implementations and the necessary HDF5 Java Libraries to access HDF5 files are provided in a single JAR File, which can be included in a new program development.
8. Conclusions

The training period needed for the development of the application presented here has been very significant, due to the many concepts involved in this study. First of all, it was mandatory to understand the Java-based processing model and environment of the Gaia DPAC systems. It includes the overall organization, the IDU system (on which our application is targeted), the Main Database layout, and the data access approach and routines. We also had to understand the Prediction Error Coder (PEC), designed during the Gaia Optimum Compression Algorithm study, and requiring an adequate calibration. Next, it was necessary to learn the HDF5 file format, its data model, and what was necessary for building the present solution. An adequate HDF5 implementation of the raw Gaia data had to be devised. Finally, the PEC compressor had to be integrated. Since the code available for this compressor was in C, the simplest approach was to use the Java Native Interface to use it directly from Java. In order to obtain reasonable compression levels, the PEC compressor was adapted to the raw Gaia data and to the storage model. A very simple differential pre-processing stage was introduced, and finally it was calibrated with a realistic reference dataset.

As presented in section 1.3, all these steps go in the direction of our final goal, which is to improve the data manipulation and data processing methods for the Gaia IDU system. The goal is to decrease the system load caused by input/output processes, while at the same time keeping negligible the computing overhead due to data compression and decompression. More generally, this study intends to reveal a new and efficient data format for the storage and transfer of large data files, and how this new format can be used in the case of the Gaia mission.

An important feature of the present prototype application is that HDF technologies offer Java implementations of their HDF5 libraries, which allows for a full compliance of the Gaia DPAC software engineering guidelines. Also, the HDF5 file format is completely adaptable to large amounts of scientific data as those generated in space missions such as Gaia. The concurrent file access offered by the HDF5 libraries gives an important advantage to this format.

The file size is the main attribute that we wish to decrease with this solution over the original GBIN format. It is closely related to the time needed for the data transfers between computing nodes and a central disk. Nevertheless, this is not the only feature to consider. The actual performance of HDF5 data access routines is important as well – especially in such a high-performance computing system such as IDU. As seen in Chapter 6, both the file size and the access performance have been studied in detail. While the file sizes have not been significantly reduced with this prototype, the access times are much better than when using the original GBIN data format.

By means of the realistic reference data obtained from the GASS simulator, our tests have revealed that the HDF5+PEC solution offers similar file sizes, yet sometimes larger, when compared to the solution currently used by most DPAC systems – based on ZIP and
Conclusions

therefore much more complex than PEC. It is important to emphasize this, because such result has been achieved with a very simple compressor, a fixed calibration, and an extremely simple pre-processing stage. On the other hand, when it comes to the file access times, our solution is clearly better than the current GBIN file format. As seen in Chapter 6, the read time in HDF5 is at least 25% quicker than GBIN on any computing platform – which includes the PEC decompression times. Implementing the PEC compressor as a standard embedded filter in HDF5, or alternatively, implementing PEC in Java (thus avoiding the JNI overhead), would lead to even better access times. Summarizing, the HDF5+PEC solution is clearly competitive for the storage and transfer of raw Gaia data, and therefore it could be used on the operational environment of the Gaia IDU system, in the MareNostrum supercomputer.

8.1 Forthcoming work

This feasibility study opens a large range of possible improvements and options for the prototype application:

- The PEC compression routine has to be run twice for generating the HDF5 file: first, we obtain the SM and AF samples length, and then, the datasets are actually created and filled. Therefore, a first immediate improvement is to avoid such repetition, e.g., by storing the output of the first compression execution in a temporary array.
- A possible performance improvement could be obtained by implementing a Java version of the PEC compressor. This would avoid the JNI overhead in the application. Alternatively, PEC could be implemented as an embedded filter in HDF5, which would open the possibility of including PEC as a standard filter for this file format, useable in other projects and environments.
- An improved pre-processing stage would lead to an immediate improvement in the compression ratios, as well as the usage of coding tables adapted to the type of data to be compressed. This might be done by considering the GClass of each AstroObservation, for example.
- Another possible improvement is to redistribute the way in which AstroObservation fields are stored in the HDF5 file. This means that other groups or datasets could be created, and this containers can be filled with the data in a different way that the one which is presented here. Besides this, the PEC compression could be applied to other fields, not only AF and SM samples.
- HDF5 is a file format which is being continuously improved. Newer versions add new functions and improve previous methods, so a performance improvement could be obtained with newer HDF5 versions. Also, the temporary dataset for copying the data between the GBIN file and the HDF5 file might be avoided in a new release, recovering the file space that it occupies.
- Recent releases of the HDF5 libraries include an SZIP compression mechanism embedded in the format, which is based on the CCSDS 121.0 standard for space data.
compression. Comparative tests between PEC and SZIP could be run. It might also be possible to consider a combination of PEC+SZIP for an optimal file size.

- Finally, it is important to test the scalability of the application. HDF5 gives a concurrent access capability, so the access times could be compared when different threads or processes access the same HDF5 file. It is worth noting that this will be the case for the Gaia IDU system.
9. Annex

9.1 Rice Coder Overview

The Consultative Committee for Space Data Systems (CCSDS) recommends the use of a data compression algorithm based on Rice codes. These codes are optimal for discrete Laplacian distributions, which are expected to occur under the premises of the CCSDS, that is, after an adequate pre-processing stage. This assumes a correct operation of the predictor. However, in a realistic case, noisy samples might appear and modify the expected distribution.

Rice coders are a special case of Golomb coders, where the parameter m is a power of 2 (m = 2^k, with k ≥ 0). Rice coders have 2k variable-length codes, starting with a minimum length of k + 1, using a very simple coding algorithm [4, 11]. The length of a Rice code for an integer n coded using a parameter k can be easily computed as

\[ 1 + k + \left\lfloor \frac{n}{2^k} \right\rfloor. \]

Fig. 9.1.1 shows some Rice codes for small values of k:

<table>
<thead>
<tr>
<th>n</th>
<th>k = 0</th>
<th>k = 1</th>
<th>k = 2</th>
<th>k = 3</th>
<th>k = 4</th>
<th>k = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>111</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>11111</td>
<td>1111</td>
<td>111</td>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>111111</td>
<td>11111</td>
<td>1111</td>
<td>111</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1111111</td>
<td>111111</td>
<td>11111</td>
<td>1111</td>
<td>1111</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>11111111</td>
<td>1111111</td>
<td>111111</td>
<td>11111</td>
<td>11111</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>111111111</td>
<td>11111111</td>
<td>1111111</td>
<td>111111</td>
<td>111111</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1111111111</td>
<td>111111111</td>
<td>11111111</td>
<td>1111111</td>
<td>1111111</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>11111111111</td>
<td>1111111111</td>
<td>111111111</td>
<td>11111111</td>
<td>11111111</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>111111111111</td>
<td>11111111111</td>
<td>1111111111</td>
<td>111111111</td>
<td>111111111</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1111111111111</td>
<td>111111111111</td>
<td>11111111111</td>
<td>1111111111</td>
<td>1111111111</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 9.1.1: Rice codes for values n = 0 to 16 and parameter k = 0 to 5

The parameter k of the coder must be chosen carefully in order to obtain the expected compression ratios for a given set of data. There is a rapid increase in code length for small values of k. It might occur that the expected data set only has low values and, under this condition, receiving a single high value would lead to an output code of some hundreds or thousands of bits. Fortunately, the CCSDS 121.0 recommendation introduces an adaptive layer to automatically select the best k parameter for each data block. This is illustrated in...
Figure 9.2.1, where the Zero Block (ZB), Second Extension (SE) and Fundamental Sequence (FS) options, which are used for very low entropy data blocks, can also be seen.

*Fig. 9.1.2: Adaptive Rice coder specified in the CCSDS 121.0 recommendation*
9.2 Setup of the environment – HDF5 and Eclipse

In order to be able to use the HDF5 functions when developing Java applications using Eclipse it is necessary to follow the steps described hereafter.

Obtaining the HDF libraries

First of all, it is necessary to download the HDF5 libraries from the HDF Group web page. They can be found on ftp://ftp.hdfgroup.org/HDF5/hdf-java/bin/. To develop this project we use Windows XP 32 bits, so win32 is the folder that must be opened:

Fig. 9.2.1: HDF5 Libraries, sorted by architecture

Once inside on it, the necessary libraries which are compressed inside the file hdf-java-2.5-bin.tar we can be downloaded:

Fig. 9.2.2: HDF Libraries for Windows 32 bits.

These files must be uncompressed and placed in the development folder of the project.
Configuring the Eclipse Project

The next step is to add the required libraries to the Java Build Path. Then, all the necessary functions will be available to use, even if the entire project is moved to another computer. First, a new folder, named “lib”, has to be created in it:

![Folder creation in our project](image1)

We can then use the Windows Explorer to copy all the HDF Jar Archives to this folder:

![HDF5 Jar files included in our project](image2)

Then, we have to go to the view “Package explorer”, which is next to the Navigator. Alternatively, we can go to menu “Window” → “Show view” → “Package Explorer” – select all of the libraries and, with a right-click, open the contextual Menu → “Build path” → “Add to Build Path”.

Adding the system libraries

The third and last step is to add to the JRE the system files which are necessary to manage HDF5 files. They must be inside the first folder of the system PATH. This path is, for a
Spanish Windows OS, on C:\Archivos de programa\Java\jre6\bin. We have to copy the three dll from hdf-java-2.5-bin.tar/lib/win to this folder:

![jhsrb.dll](image1.png) ![j hdf5.dll](image2.png) ![j hdf.dll](image3.png)

*Fig. 9.2.5:* DLL libraries required to manage HDF5 files
9.3 Setup of the environment – DPAC and Eclipse

The method for configuring Eclipse with the DPAC libraries to access GBIN files is the same as for the HDF5 case. These are the jars needed:

![GaiaMdbDm](image1)  
![GaiaDpcDm](image2)  
![GaiaToolsComplete](image3)

*Fig. 9.3.1: Gaia-DPAC libraries needed to access AstroObservation files*
9.4 DPAC API: Methods used

`gaia.cu1.tools.util.props.PropertyLoader.load`

public static void load()
    throws GaiaConfigurationException
Load configuration from default configuration file. The value of the default configuration file can be set using the system property gaia.cu1.tools.util.prop.configuration set either from the command-line or in the ant build file. It will only execute once. If you explicitly need to reload properties use load(fileName, true).

**Throws:**
GaiaConfigurationException - if the default property file cannot be loaded.

public static void load(String configFile)
    throws GaiaConfigurationException
Load configuration from user-specified configuration file.

**Parameters:**
configFile -

**Throws:**
GaiaConfigurationException - if the property file cannot be loaded.

`gaia.cu1.tools.util.Logger.info`

void info(Object msg)
Log a message with info logging level.

**Parameters:**
msg - Message to be logged.

`gaia.cu1.tools.util.props.PropertyLoader.getArray`

public static String[] getArray(String property)
Returns an array of property values from the Properties. Note that this method assumes that the property values are separated by one or more spaces. Multiple spaces in between property values are equivalent to a single space. A property value corresponding to one or more spaces is not possible.

**Parameters:**
property - Property to query

**Returns:**
Property values.
gaia.cu1.tools.dal.file.GaiaRootGaiaTable.getNumRows

public final int getNumRows()
    throws GaiaDataAccessException

Gets the number of rows in a GaiaTable.

Specified by:
getNumRows in interface GaiaTable

Returns:
the num rows

Throws:
GaiaDataAccessException - the gaia data access exception

See Also:
GaiaTable.getNumRows()

---

gaia.cu1.tools.dal.file.GaiaRootGaiaTable.close

public final void close()

Just sets the current row index to -1.

Specified by:
close in interface GaiaTable

See Also:
GaiaTable.close()

---

gaia.cu1.tools.dal.file.GaiaRootGaiaTable.next

public final boolean next()
    throws GaiaDataAccessException

Next.

Specified by:
next in interface GaiaTable

Returns:
ture, if next

Throws:
GaiaDataAccessException - the gaia data access exception

See Also:
GaiaTable.next()
long getSolutionId()

Get the SolutionId corresponding to the data currently loaded in the DataServer.

Returns:
solutionId

---

4 All the AstroObservation fields have a function to get the data from the loaded row (getCreationData, getHealPixOpp, etc...) and the functionality is exactly the same as getSolutionId.
9.5 HDF5 API: Used methods

**H5Fcreate**

```java
public static int H5Fcreate(java.lang.String name, int flags, int create_id, int access_id)
    throws HDF5LibraryException, java.lang.NullPointerException
```

H5Fcreate is the primary function for creating HDF5 files.

**Parameters:**
- name - Name of the file to access.
- flags - File access flags. Possible values include:
  - H5F_ACC_RDWR Allow read and write access to file.
  - H5F_ACC_RDONLY Allow read-only access to file.
  - H5F_ACC_TRUNC Truncate file, if it already exists, erasing all data previously stored in the file.
  - H5F_ACC_EXCL Fail if file already exists.
  - H5F_ACC_DEBUG Print debug information.
  - H5P_DEFAULT Apply default file access and creation properties.
- create_id - File creation property list identifier, used when modifying default file metadata. Use H5P_DEFAULT for default access properties.
- access_id - File access property list identifier. If parallel file access is desired, this is a collective call according to the communicator stored in the access_id (not supported in Java). Use H5P_DEFAULT for default access properties.

**Returns:**
- a file identifier if successful

**Throws:**
- HDF5LibraryException - Error from the HDF5 Library.
- java.lang.NullPointerException - name is null.

**H5Gcreate**

```java
public static int H5Gcreate(int loc_id, java.lang.String name, int lcpl_id, int gcpl_id, int gapl_id)
    throws HDF5LibraryException, java.lang.NullPointerException
```

H5Gcreate creates a new group with the specified name at the specified location, loc_id.

**Parameters:**
- loc_id - IN: The file or group identifier.
- name - IN: The absolute or relative name of the new group.
- lcpl_id - IN: Identifier of link creation property list.
- gcpl_id - IN: Identifier of group creation property list.
- gapl_id - IN: Identifier of group access property list. (No group access properties have been implemented at this time; use H5P_DEFAULT.)

**Returns:**
a valid group identifier

**Throws:**
Feasibility study of the PEC compressor in HDF5 file format

HDF5LibraryException - - Error from the HDF-5 Library.
java.lang.NullPointerException - - name is null.

**H5Gclose**

public static int H5Gclose(int group_id)
    throws HDF5LibraryException

H5Gclose releases resources used by a group which was opened by a call to H5Gcreate() or H5Gopen().

**Parameters:**
group_id - Group identifier to release.

**Returns:**
a non-negative value if successful

**Throws:**
HDF5LibraryException - - Error from the HDF-5 Library.

**H5Gopen**

public static int H5Gopen(int loc_id, java.lang.String name, int gapl_id)
    throws HDF5LibraryException,
    java.lang.NullPointerException

H5Gopen opens an existing group, name, at the location specified by loc_id.

**Parameters:**
loc_id - IN: File or group identifier specifying the location of the group to be opened.
name - IN: Name of group to open.
gapl_id - IN: Identifier of group access property list. (No group access properties have been implemented at this time; use H5P_DEFAULT.)

**Returns:**
a valid group identifier if successful

**Throws:**
HDF5LibraryException - - Error from the HDF-5 Library.
java.lang.NullPointerException - - name is null.

**H5Screate_simple**

public static int H5Screate_simple(int rank, byte[] dims, byte[] maxdims)
    throws HDF5Exception, java.lang.NullPointerException

Deprecated. use H5Screate_simple(int rank, long[] dims, long[] maxdims)

**Throws:**
HDF5Exception
java.lang.NullPointerException
H5Tcreate
public static int H5Tcreate(int type, long size)
        throws HDF5LibraryException
H5Tcreate creates a new datatype of the specified class with the specified number of bytes.

Parameters:
- type: Class of datatype to create.
- size: The number of bytes in the datatype to create.

Returns:
datatype identifier

Throws:
HDF5LibraryException - Error from the HDF-5 Library.

H5Tinsert
public static int H5Tinsert(int type_id, java.lang.String name, long offset, int field_id)
        throws HDF5LibraryException, java.lang.nullPointerException
H5Tinsert adds another member to the compound datatype type_id.

Parameters:
- type_id: Identifier of compound datatype to modify.
- name: Name of the field to insert.
- offset: Offset in memory structure of the field to insert.
- field_id: Datatype identifier of the field to insert.

Returns:
a non-negative value if successful

Throws:
HDF5LibraryException - Error from the HDF-5 Library.
java.lang/nullPointerException - name is null.

H5Dcreate
public static int H5Dcreate(int loc_id, java.lang.String name, int type_id, int space_id, int create_plist_id)
        throws HDF5LibraryException, java.langNullException
H5Dcreate creates a data set with a name, name, in the file or in the group specified by the identifier loc_id.

Parameters:
- loc_id: Identifier of the file or group to create the dataset within.
- name: The name of the dataset to create.
- type_id: Identifier of the datatype to use when creating the dataset.
- space_id: Identifier of the dataspace to use when creating the dataset.
- create_plist_id: Identifier of the set creation property list.

Returns:
a dataset identifier if successful

Throws:
HDF5LibraryException - - Error from the HDF-5 Library.
java.lang.NullPointerException - - name is null.

**H5Dopen**
Deprecated
public static int H5Dopen(int loc_id, java.lang.String name)
    throws HDF5LibraryException, java.lang.NullPointerException
H5Dopen opens the existing dataset specified by a location identifier and name, loc_id and name, respectively.

**Parameters:**
- loc_id - IN: Location identifier
- name - IN: Dataset name

**Returns:**
a dataset identifier if successful

**Throws:**
- HDF5LibraryException - - Error from the HDF-5 Library.
- java.lang.NullPointerException - - name is null.

**H5Dclose**
public static int H5Dclose(int dataset_id)
    throws HDF5LibraryException
H5Dclose ends access to a dataset specified by dataset_id and releases resources used by it.

**Parameters:**
- dataset_id - Identifier of the dataset to finish access to.

**Returns:**
a non-negative value if successful

**Throws:**
- HDF5LibraryException - - Error from the HDF-5 Library.

**H5Sselect_hyperslab**
public static int H5Sselect_hyperslab(int space_id, int op, byte[] start, byte[] stride, byte[] count, byte[] block)
    throws HDF5LibraryException, java.lang.NullPointerException,
            java.lang.IllegalArgumentException
H5Sselect_hyperslab selects a hyperslab region to add to the current selected region for the dataspace specified by space_id. The start, stride, count, and block arrays must be the same size as the rank of the dataspace.

**Parameters:**
- space_id - IN: Identifier of dataspace selection to modify
- op - IN: Operation to perform on current selection.
- start - IN: Offset of start of hyperslab
stride - IN: Hyperslab stride.
count - IN: Number of blocks included in hyperslab.
block - IN: Size of block in hyperslab.

**Returns:**
a non-negative value if successful

**Throws:**
HDF5LibraryException - - Error from the HDF5 Library.
java.lang.NullPointerException - - an input array is null.
java.lang.NullPointerException - - an input array is invalid.
java.lang.IllegalArgumentException

**H5Tclose**

public static int H5Tclose(int type_id)
    throws HDF5LibraryException

H5Tclose releases a datatype.

**Parameters:**
type_id - IN: Identifier of datatype to release.

**Returns:**
a non-negative value if successful

**Throw:**
HDF5LibraryException - - Error from the HDF5 Library.

**H5Sclose**

public static int H5Sclose(int space_id)
    throws HDF5LibraryException

H5Sclose releases a dataspace.

**Parameters:**
space_id - Identifier of dataspace to release.

**Returns:**
a non-negative value if successful

**Throw:**
HDF5LibraryException - - Error from the HDF5 Library.

**H5Gunlink**

public static int H5Gunlink(int loc_id, java.lang.String name)
    throws HDF5LibraryException, java.lang.NullPointerException

H5Gunlink removes an association between a name and an object.

**Parameters:**
loc_id - Identifier of the file containing the object.
name - Name of the object to unlink.

**Returns:**
a non-negative value if successful

Throws:
HDF5LibraryException - Error from the HDF5 Library.
java.lang.NullPointerException - name is null.

H5Fclose
public static int H5Fclose(int file_id)
    throws HDF5LibraryException
H5Fclose terminates access to an HDF5 file.
Parameters:
file_id - Identifier of a file to terminate access to.
Returns:
a non-negative value if successful
Throws:
HDF5LibraryException - Error from the HDF5 Library.
9.6 Study of the need to split AstroObservations into groups

An interesting feature of HDF5 is that all the data inside a file is completely indexed. That means all the values stored inside can be accessible independently from different locations and times. The concurrency is always guaranteed. The number of indexes is the most important way to control the file size, above the kind of data stored or the number of these data. Thus, the best option is to split the AstroObservations into groups and store these groups independently on the HDF5 File. The following study shows how the file size increases or decreases due to this grouping. For this, the following GBIN file is used:

File name: mdbcu3idtrawastroobservation.0011.00216-0011.00432.0000.178.gbin
File Size: 9,708 KB
Number of AstroObservations stored (N Astro): 43,553

The following figures are made creating the file with a fixed number of AstroObservations (as if we were truncating the file), modifying the number of elements of each group and noting the resulting file size. The X axis is Number of elements per group, while the Y axis is File Size in Bytes.
Feasibility study of the PEC compressor in HDF5 file format

\[ \text{N\text{Astro} = 7000} \]

\[ \text{N\text{Astro} = 10500} \]

\[ \text{N\text{Astro} = 15000} \]
As can be seen, the final file size indeed varies with the number of AstroObservations per group. In all of the cases, having all the AstroObservations stored in a single group results in the largest file, so that is not recommended at all. There are always several minimums on each plot, but in general, the size is smaller when a smaller number of elements per group is adopted. Considering the large files typically handled in Gaia, the best option when creating HDF5 Files seems to be \textbf{400 AstroObservations per group}.
10. Bibliography


