INAF
LFI Project System Team

Planck LFI

TITLE: **A tool to compress LFI detector pointings and differenced scientific data**

DOC. TYPE: Technical manual
PROJECT REF.: PL-LFI-xxx-xx-xxx
ISSUE/REV.: 1.0

DATE: December 2, 2013

Prepared by	Maurizio Tomasi	December 2, 2013
Agreed by	M. Bersanelli LFI Instrument Scientist C.R. Butler LFI Program Manager	???
Approved by	N. Mandolesi LFI Principal Investigator	???

CHANGE RECORD

Issue	Date	Sheet	Description of change	Release
0.1	5 June, 2013	All	First draft issue of document	0.1

DISTRIBUTION LIST

Recipient	Company/Institute	E-mail address	Sent
N. Mandolesi	INAF-IASF Bologna	reno@bo.iasf.cnr.it	Yes
C. Butler	INAF-IASF Bologna	butler@bo.iasf.cnr.it	Yes
M. Bersanelli	Univ. di Milano	marco.bersanelli@fisica.unimi.it	Yes
A. Mennella	Univ. di Milano	daniele.mennella@fisica.unimi.it	Yes
M. Tomasi	INAF-IASF Milano	tomasi@lambrate.inaf.it	Yes
A. Zacchei	INAF-OAT Trieste	zacchei@oats.inaf.it	Yes

Contents

Abstract

In this document we describe the hypothesis and the usage of `squeezer`, a program which compresses/decompresses LFI detector pointings and differenced data. The program applies a number of compression techniques and is able to obtain compression ratios of the order of 5-10. Given that such data typically requires several Terabytes of storage, this translates into a significant advantage for situations in which such pointings need to be transferred.

This document describes the compression algorithms used by the program, their implementation and the usage of `squeezer`. The performance of the program is discussed using a number of examples.

1 Introduction

Pointing information are among the most important information available in the *Planck*/LFI Level 1 pipeline. They are needed to produce maps, and they are therefore required whenever any member of the *Planck* needs to study calibration schemes, simulate systematic effects of thermal and optical origin, etc.

Because of their importance, it would be desirable to have them available on the many computational facilities available within the *Planck* community. Examples of such facilities are: the NERSC supercluster in the US, Sisu in Finland, the Erebor cluster in Italy... Unfortunately, it is difficult to transfer such data because of their volume (several Terabytes).

The *Planck*/LFI DPC adopted the following strategy to make pointing information available to the collaboration: pointing information are generated at the LFI DPC using the full knowledge of the instrument beams, spacecraft distortions, etc., but they are not transferred elsewhere. Rather, Attitude History Files are transferred to other computing centres, which then reconstruct the pointings using the Level S pipeline or other tools. Once the pointings have been generated, the consistency between them and the ones generated by the LFI DPC is measured by various means, e.g., by comparing hit maps.

This process has worked pretty well, but it requires considerable effort: the Level S pipeline must be ported and installed on every new computing facility, and its consistency must be measured after any upgrade to the software. Moreover, the software itself needs to be upgraded whenever the pipeline used to recreate pointing information at the DPC changes.

This document describes the implementation of `squeezer`, a program which efficiently compresses *Planck*/LFI pointing information, allowing it to be copied and stored using little disk space and bandwidth.

2 Installation of the program

To obtain the sources of the program, you must use the Git version control system to connect to Github:

```
$ git clone git@github.com:ziotom78/squeezer.git
```

This will create a directory named `squeezer` that contains the source code of the program. To compile the program, you must have the following libraries installed:

1. CFITSIO;
2. The GNU Scientific Library.

Compile the program with the following commands:

```
$ cd squeezer
$ mkdir build && cd build
& ../configure
& make
& sudo make install
```

The above commands assume you have `sudo` powers, which is not probably true if you are installing `squeezer` on a cluster. In this case specify a different prefix when calling `configure`:

```
$ ../configure --prefix=$HOME/usr
```

If the script has troubles in finding the CFITSIO or GSL libraries, you can specify the path of their include files using `CPPFLAGS` and the path of the libraries themselves using `LDFLAGS`:

```
$ ../configure CPPFLAGS="-I /path/to/include" LDFLAGS="-L /path/to/libraries"
```

Usually, you can find the directory where to find CFITSIO include files by looking at the environment variable `CFITSIO_INC`, or something like.

There are other dependencies that are optional (i.e., their absence do not prevent `squeezer` from being built, but it will have reduced functionality):

1. If the TOODI library is available, then `squeezer` will be able to load pointing information from the DMC database. This is currently possible only on the Ironthrone cluster.

2. If the CppUnit is available, you can run `make check` to check that the code runs correctly on your machine.

3. If the installation script finds an installed copy of HPixLib, it will generate an executable called `hit_map`, which can be used to produce hit maps from pointing files.

3 Usage of the program

The executable `squeezer` can be used to compress and decompress pointings. Run `squeezer help` to get help.

4 Principles of data compression

4.1 Types of compression schemes

A compression scheme is an algorithm which takes as input an ordered list of N symbols and produces a new list of symbols, which makes its output, with the objective that the space needed to store the output is smaller that the space needed for the input. The input and output symbols do not need to be part of the same set: for instance, the input symbols might be bytes, while the output symbols might be words or floating-point numbers. The *compression ratio* c_r is defined as the ratio between the number of bytes necessary to encode the input and the number of bytes necessary to encode the compressed stream. For efficient compressors, $c_r \gg 1$.

In many cases, in order to make the compression more efficient, the input data needs to be transformed in some way. For instance, the sequence $x_i = 2i + 1$ with $i = 0 \ldots N$, whose first elements are

$$1, 3, 5, 7, 9, 11, \ldots,$$

is not easily compressed by traditional algorithms. But if we transform the sequence x_i into the sequence $y_i = x_{i+1} - x_i$, then the sequence becomes

$$2, 2, 2, 2, 2, \ldots,$$

and such sequence can be trivially compressed.

Compression algorithms can be divided into two broad families, depending on the type of output $\mathbf{x}'$ produced by the sequential application of the compressor and decompressor through the following pipeline:

$$\mathbf{x} \xrightarrow{\text{Compression}} \mathbf{y} \xrightarrow{\text{Decompression}} \mathbf{x}'. \tag{1}$$

Lossless compressors guarantee that $\mathbf{x}' = \mathbf{x}$, while *lossy compressors* degrade the quality of the data so that the result of the decompression is not identical to the input given to the compressor. The difference $\varepsilon = \|\mathbf{x} - \mathbf{x}'\|$ between the input and the product of the decompression is called *decompression error*, and it depends on the choice of the norm $\|\cdot\|$: in this document we will use the well-known norm

$$\|\mathbf{v}\|_\infty = \max_{i=1}^{N} |v_i| \,. \tag{2}$$

Thus, if we compress a vector of angles θ_i, the compression error ε_θ will be defined as the maximum discrepancy between the input angles and the compressed angles.

4.2 Truncation

The most simple way to lossly compress data is to truncate them to some form. For instance, instead of saving an integer number j, one might save the value of $\lfloor \log_{10} j \rfloor$ (largest power of ten that is smaller than j). Since for every integer number $\lfloor \log_{10} j \rfloor < j$, this means that we save some bytes in trasmitting the number: if j takes 32 bits, $\lfloor \log_{10} j \rfloor$ only takes 4 bits (because $\lfloor \log_{10} 2^{32} \rfloor = 9$, and 4 bits can encode all the numbers between 0 and 9).

All the floating-point values used in the memorization of pointing information and differenced data are in double precision. This is however seldom needed, as the relative error in converting a double-precision number into a single-precision number is less than 10^{-7} for machines using one of the the IEEE-754 standards, which is the case of machines equipped with Intel and AMD 32/64 bit processors (Handbook of Floating-Point Arithmetic, Müller et al., Birkhäuser 2010).

The program `squeezer` truncates the following data streams from double-precision numbers to single-precision numbers, thus obtaining a constant compression ratio $c_r = 2$:

1. SCET times. In order to prevent excessive loss of precision (the order of magnitude of the SCET time t_i is $\sim 10^{15}$ ms, which means that truncating from double precision to single precision produces a relative error of $\sim 10^{-7}$, i.e., a few seconds), the values that are encoded are
$$\tau_i = t_i^{\text{SCET}} - \frac{t_N^{\text{SCET}} - t_1^{\text{SCET}}}{t_N^{\text{OBT}} - t_1^{\text{OBT}}} t_i^{\text{OBT}}, \tag{3}$$
i.e., the difference between the SCET value itself and a linear conversion of the OBT time into SCET units. In this way, τ_i is of the order of unity, and therefore the truncation error becomes negligible.

2. The angles that describe a pointing direction (θ, φ);

3. The beam rotation angle (ψ);

4. Differenced data.

4.3 Run-length compression

Run-length compressors are effective when there are long sequences of repeated symbols in the input. The idea is to replace each sequence with a pair associating each symbol with the number of times it was repeated. Therefore, the sequence

5	5	5	6	6	4	4	3	2

is encoded as

3	5	2	6	2	4	1	3	1	2

Run-length compression is *lossless*: no information is thrown away during the compression/decompression process. In the context of LFI, this kind of compression is very effective with the data quality flags saved that are stored in differenced data objects, as they tend to repeat very often.

Sometimes run-length compression is more effective if one applies some transforms to the data. This applies to LFI On-Board Times (OBT), which usually increment in regular intervals. The idea is that if t_i is a sequence of on-board times, one can apply run-length compression to the series

$$d_i = t_{i+1} - t_i,$$

where the first value t_1 must of course be encoded separatedly. This proves to be very effective, as an entire OD of OBT times compresses down to a dozen of bytes ($c_r \sim 10^6$).

4.4 Polynomial compression

Polynomial compression is a simple *lossy* compression scheme which is effective when the data to compress vary very smoothly and have no noise. The idea is to encode the data by approximating them with smooth functions that need less information to be encoded: the program `squeezer` uses polynomials, but in principle any other family of smooth functions can be used.

Polynomial compression works fairy well with *Planck*/LFI pointing information, i.e., the orientation of the beam and the direction of its axis. Such information is fully described by three angles: two of them (θ and φ) identify the position of the beam centre in the sky, and the third (ψ) characterizes the rotation of the beam with respect to some reference frame.

All of these angles vary very smoothly with time. The idea is that if a set of N points are fitted reasonably well by a m-degree polynomial, then one can save to disk the m coefficients of the polynomial instead of the N points. This is illustrated in Fig. 2, where we show how the compression works on a set of $N = 23$ points. Data are split into "frames", each containing the same number of points (in the figure there are 10 points per frame), with the possible exception of the last one (in the figure, it has 3 points). Within each frame, the points are interpolated using a m-degree polynomial, whose coefficients are saved instead of the points: the higher m, the better the fit. Depending on the number of points left in the last (rightmost) frame, the code might choose not to calculate any interpolation, but rather to save the points as they are.

The interpolation formula for a set of angles $\{\phi_i\}_{i=1}^{N}$ assumes that the points to be fitted have coordinates (i, ϕ_i), and that the fit minimizes the function

$$\chi^2 = \sum_{i=0}^{N-1} \left(\phi_i - \sum_{j=0}^{m} a_j i^j \right)^2 \tag{4}$$

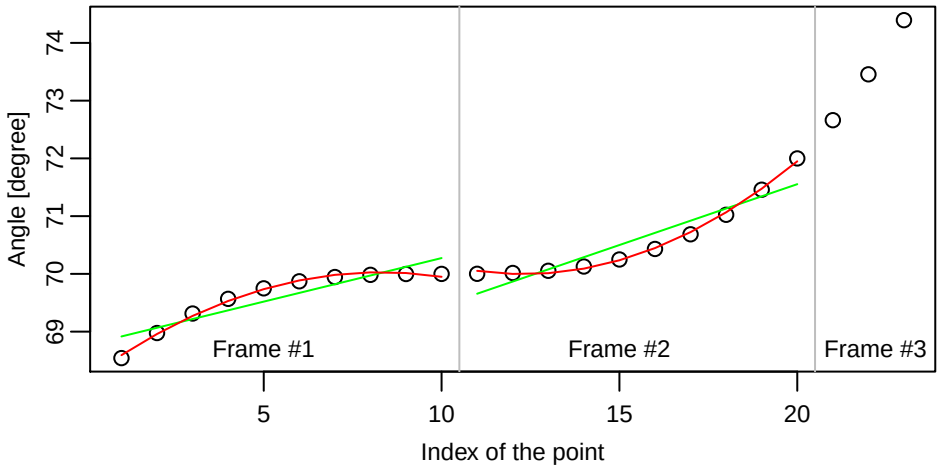

Figure 1: Implementation of the polynomial compression. Green lines are first-order least-squares polynomials ($y = a_0 + a_1 x$), red lines are second-order polynomials ($y = a_0 + a_1 x + a_2 x^2$). Refer to the text for more details.

$$\begin{aligned}
\textstyle\sum_{i=1}^{n} i &= \tfrac{1}{2}\left(n^2+n\right)\\
\textstyle\sum_{i=1}^{n} i^2 &= \tfrac{1}{6}\left(2n^3+3n^2+n\right)\\
\textstyle\sum_{i=1}^{n} i^3 &= \tfrac{1}{4}\left(n^4+2n^3+n^2\right)\\
\textstyle\sum_{i=1}^{n} i^4 &= \tfrac{1}{30}\left(6n^5+15n^4+10n^3-n\right)\\
\textstyle\sum_{i=1}^{n} i^5 &= \tfrac{1}{12}\left(2n^6+6n^5+5n^4-n^2\right)\\
\textstyle\sum_{i=1}^{n} i^6 &= \tfrac{1}{42}\left(6n^7+21n^6+21n^5-7n^3+n\right)\\
\textstyle\sum_{i=1}^{n} i^7 &= \tfrac{1}{24}\left(3n^8+12n^7+14n^6-7n^4+2n^2\right)\\
\textstyle\sum_{i=1}^{n} i^8 &= \tfrac{1}{90}\left(10n^9+45n^8+60n^7-42n^5+20n^3-3n\right)\\
\textstyle\sum_{i=1}^{n} i^9 &= \tfrac{1}{20}\left(2n^{10}+10n^9+15n^8-14n^6+10n^4-3n^2\right)\\
\textstyle\sum_{i=1}^{n} i^{10} &= \tfrac{1}{66}\left(6n^{11}+33n^{10}+55n^9-66n^7+66n^5-33n^3+5n\right)
\end{aligned}$$

Table 1: Faulhaber's polynomials up to the tenth degree.

(least-square fitting). It is trivial to prove that χ^2 has a global minimum at the point $\mathbf{a} = \left(a_0, a_1, \ldots, a_m\right)$ which satisfies the equation

$$\mathbf{A}\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \ldots \\ a_m \end{pmatrix} = \begin{pmatrix} \sum \phi_i \\ \sum i\phi_i \\ \sum i^2\phi_i \\ \ldots \\ \sum i^m\phi_i \end{pmatrix}, \tag{5}$$

with

$$\mathbf{A} = \begin{pmatrix} n & \sum i & \sum i^2 & \sum i^3 & \ldots & \sum i^m \\ \sum i & \sum i^2 & \sum i^3 & \sum i^4 & \ldots & \sum i^{m+1} \\ \sum i^2 & \sum i^3 & \sum i^4 & \sum i^5 & \ldots & \sum i^{m+2} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \sum i^m & \sum i^{m+1} & \sum i^{m+2} & \sum i^{m+4} & \ldots & \sum i^{2m} \end{pmatrix}, \tag{6}$$

where we assumed that all the sums run from $i = 1$ to $N - 1$.

The matrix **A** does not depend on the value of the points ϕ_i, and therefore it needs to be inverted only once. Moreover, the quantity $\sum_i i^p$ can be computed efficiently using Faulhaber's formula:

$$\sum_{i=1}^{k} i^p = \frac{1}{p+1} \sum_{j=0}^{p} (-1)^j \binom{p+1}{j} B_j n^{p+1-j}, \tag{7}$$

where B_j is the j-th Bernoulli number ($B_1 = 1/2$). Table 1 reports the formulae for $p = 1 \ldots 10$.

5 Implementation details

The `squeezer` program saves files in a binary format which can be read/written sequentially, i.e., without jumping back and forth. This is important, as it allows to pipe the input/output of the program; an example is the following:

```
$ squeezer compress foo.fits - | ssh remotehost squeezer decompress - foo.fits
```

This command uses `squeezer` to compress some FITS file and tunnel it through SSH to `remotehost`, where another copy of `squeezer` (which must have been already installed on `remotehost`) decompresses it. Note the usage of `-` to denote input/output through the console.

In order to allow the above code to work, a number of tricks have been implemented in the structure of the compressed file created by `squeezer`. First, the information stored in the j-th byte in the file can be interpreted by only using the $j-1$ bytes before it (i.e., no seek-forward read operations are necessary). Second, the program does not need to keep all the $j-1$ bytes in memory, but only a subset of them (this limits the amount of memory needed for the decompression).

5.1 File format

The file format used by `squeezer` is binary and is made by a *header* and a sequence of *chunks*. The header contains general information about the data in the file, while each chunk memorizes one vector of data. The number of chunks depends on the type of file:

1. Files containing compressed detector pointings have always 5 chunks: OBT times, SCET times, and the three angles θ, φ, and ψ;

2. Files containing differenced data have 4 chunks: OBT times, SCET times, differenced data and quality flags.

The only restriction about the order of the chunks in the file is that SCET times need to appear after OBT times.

5.1.1 The file header

Table 2 lists the order of the elements found in the header of a file created by `squeezer`. Most of the fields have obvious meanings. A few notes:

1. The first four bytes in the file allow to determine if the file has really been created by `squeezer` or not. The string `PDP#0` (i.e., "PDP" followed by a null byte) is used for files containing pointing information, the string `PDD#0` is used for differenced data (either calibrated or uncalibrated).

Bytes	Size	Type	Description
0-3	4	String	Either PDP or PDD, followed by a null byte.
4-12	8	Double-precision f.p.	The constant $2.1325 \times 10^5 = (1/2+3/4+5/8+7/16) \times 10^5$
13-14	2	Word	Version of the file specification. Currently it is 0x100
15-16	2	Word	Year when the file was created (4-digit number)
17	1	Byte	Month (1...12)
18	1	Byte	Day of the monty (1...31)
19	1	Byte	Hour (0...23)
20	1	Byte	Minute (0...59)
21	1	Byte	Second (0...59)
22	1	Byte	Number of the horn (18...28)
23	1	Byte	Number of the radiometer (either 0 or 1)
24-25	2	Word	Number of the operational day
26-33	8	Double-precision f.p.	OBT of the first sample in the file
34-40	8	Double-precision f.p.	OBT of the last sample in the file
41-48	8	Double-precision f.p.	SCET of the first sample (in ms) in the file
49-56	8	Double-precision f.p.	SCET of the last sample (in ms) in the file
57-60	4	Double-word	Number of chunks in the file

Table 2: Structure of a file header.

Bytes	Size	Type	Description
0-3	4	String	The string CNK, followed by a null byte
4-11	8	Quad-word	Number of bytes in the chunk (excluding the header)
12-15	4	Double-word	Number of data samples in the chunk (N)
16-19	4	Double-word	Type of data encoded in the chunk (see Table 4)
20-27	8	Double-precision f.p.	$\min_{i=1}^{N} \lvert\epsilon_i\rvert$
28-35	8	Double-precision f.p.	$\max_{i=1}^{N} \lvert\epsilon_i\rvert$
36-43	8	Double-precision f.p.	$\langle\lvert\epsilon_i\rvert\rangle_{i=1}^{N}$
44-51	8	Double-precision f.p.	$\langle\epsilon_i\rangle_{i=1}^{N}$

Table 3: Structure of a chunk header. The symbol ϵ_i denotes the compression error for the i-th sample.

2. Bytes 4-12 contain a constant which has the purpose of helping decompression codes in deciding if they interpret multibyte fields using the correct ordering (i.e., big/little endian). The idea is that any decompression code should read the eight bytes, cast them into a double precision floating-point number and compare this number with the value listed above (which has an exact binary representation).

3. The last field is a count of the chunks that follow in the file.

5.1.2 Chunks

Following the file header illustrated in Table 2, there is a sequence of "chunks" (their number can be accessed by the field at bytes 57-60 in the file header). Each chunk starts with a "chunk header" which has the structure described in Table 3.

The field in bytes 16-19 establish the kind of data encoded in the chunk, and therefore the compression algorithm used. See Table 4 for a list of types.

Data	Code	Compression	Lossy?	Compression error	Reference
OBT times	10	Run-length	No	N.A.	Sect. 5.3
SCET times	11	Truncation	Yes	10^{-6} s	Sect. 5.2
Colatitude (θ)	12	Polynomial	Yes	User-configurable	Sect. 5.4
Longitude (φ)	13	Polynomial	Yes	User-configurable	Sect. 5.4
Beam rotation (ψ)	14	Polynomial	Yes	User-configurable	Sect. 5.4
Scientific data	15	Truncation	Yes	10^{-6} V	Sect. 5.2
Data quality flags	16	Run-length	No	N.A.	Sect. 5.3

Table 4: Types of compression applied to pointing/differenced data streams. The column "Code" lists the values to be used in the "Chunk type" field of the chunk header, see Table 3.

The order of the chunks is arbitrary. The only requirement is that SCET times must come after OBT times (because to decompress the formers one needs the latters).

5.2 Truncation

5.3 Run-length compression

5.4 Polynomial compression

Refer to Sect. 4.4 for a general description of the polynomial compression.

For an example of the error induced by this compression, see Fig. 2. The oscillatory behaviour of the error in the right side of the three plots is due to the size of each "frame", i.e., each sequence of data which is fitted to the same polynomial. Lines where the error is equal to zero are due to the fact that the compressor was not able to find a polynomial which fitted the angles with the desired accuracy (and therefore the angles in that frame were saved in the file with no compression).

Fig. 3 shows the result of several runs of `squeezer` on the same dataset, using different frame sizes and error bounds for the angles. The results show that for an error bound of 1 arcsec (blue lines), the best compression ratio ($c_r \approx 6$) can be achieved by using polynomials of the 4th order (third plot) and frames with $N \approx 35$. This corresponds to the following invocation:

```
$ squeezer compress -n 35 -p 4 -s 1 ...
```

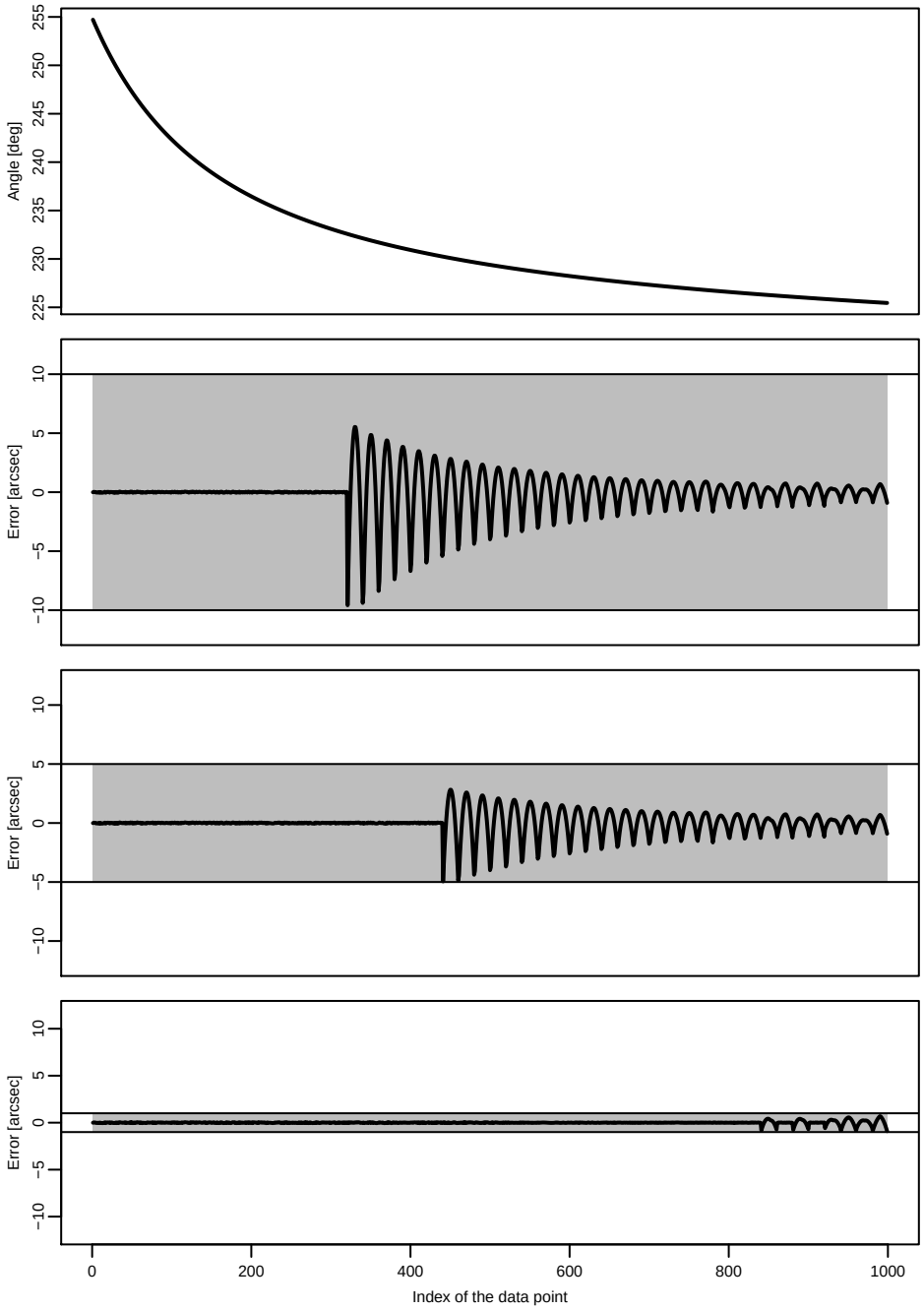

Figure 2: Compression errors. The topmost plot shows the angle to be compressed. The three plots below show the compression error and the preset tolerance (10 arcsec, 5 arcsec, 1 arcsec). The period of the oscillations depends on the length of the frames.

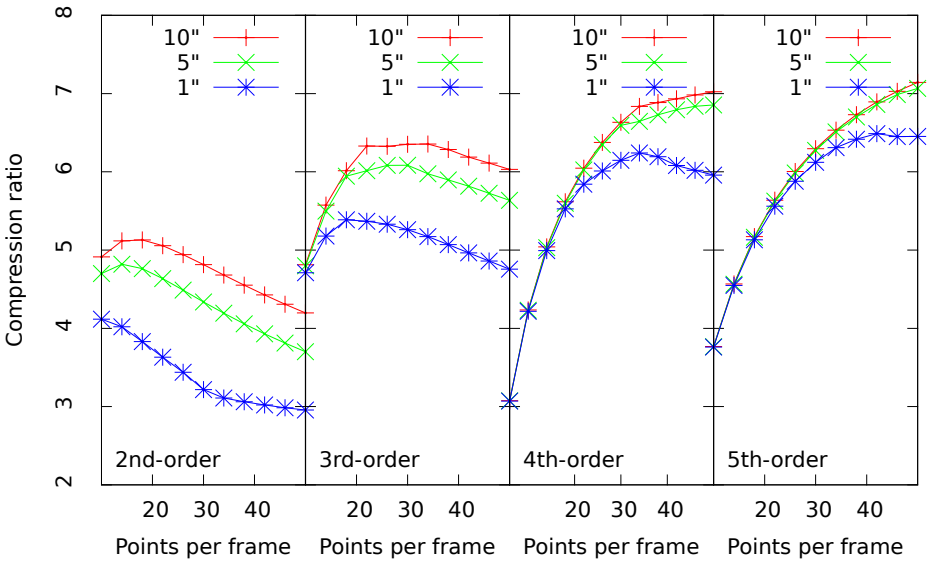

Figure 3: Compression rates as a function of the compressor parameters. See the text for further details.