Calibrating TODs using the Da Capo algorithm

Maurizio Tomasi

May 28, 2018

Contents

1 Presentation of the `Da Capo` algorithm **1**
- 1.1 Introduction . . . 1
- 1.2 Modelling the output of a detector scanning the sky . . . 1
- 1.3 How to install and build the code . . . 5
  - 1.3.1 Installing the code from source . . . 6
  - 1.3.2 Using Docker to describe the installation . . . 6
  - 1.3.3 Running the programs . . . 8

2 Mapping the input data **11**
- 2.1 What is an index file? . . . 12
- 2.2 How flagged data are removed . . . 13
- 2.3 Splitting the data into periods . . . 14
- 2.4 Configuring the program through a parameter file . . . 15
- 2.5 Implementing the main loop for `index.py` . . . 18
- 2.6 Writing the index file . . . 21

3 Implementing `Da Capo` **25**
- 3.1 Overall structure of the program . . . 25
- 3.2 Implementation of the main loop for `calibrate.py` . . . 26
- 3.3 The logging system . . . 27
- 3.4 Keeping track of the time spent by the program . . . 27
- 3.5 Configuring the program . . . 28
- 3.6 Sharing data among the MPI processes . . . 32
  - 3.6.1 Splitting lengths . . . 32
  - 3.6.2 Loading parts of TODs from FITS files . . . 35
- 3.7 Estimation of the dipole signal . . . 40
- 3.8 Implementation of the `Da Capo` algorithm . . . 43
- 3.9 Computational tricks . . . 46
- 3.10 Implementation of the conjugate gradient algorithm . . . 46
  - 3.10.1 Solving the problem using the conjugate gradient method . . . 47
  - 3.10.2 From instrumental parameters to TODs . . . 47
  - 3.10.3 From TODs to instrumental parameters . . . 49
  - 3.10.4 The gain hit map . . . 50
  - 3.10.5 From TODs to maps . . . 52
  - 3.10.6 From maps to TODs . . . 53
  - 3.10.7 More high-level calculations . . . 54
  - 3.10.8 The conjugate gradient algorithm . . . 56
- 3.11 Preconditioning the conjugate gradient algorithm . . . 58
  - 3.11.1 A simple example . . . 59
  - 3.11.2 Using preconditioners for error estimation . . . 59
  - 3.11.3 Row permutation in preconditioners . . . 61
  - 3.11.4 Full preconditioner . . . 61

3.11.5 Jacobi preconditioner . . . 63
3.11.6 Putting all together . . . 64
3.12 Implementation of the Da Capo algorithm . . . 65
3.13 Saving the results of the computation . . . 69
3.13.1 Gains and offsets . . . 69
3.13.2 Sky map . . . 70
3.13.3 Convergence information . . . 70
3.13.4 Flushing data and closing the file . . . 71

4 Validation of the code **73**
4.1 Unit tests . . . 73
4.1.1 Creating test data . . . 74
4.1.2 Implementing unit tests . . . 77
4.1.3 Testing other functions . . . 80
4.2 Integration tests . . . 81
4.2.1 Creation of simulated timelines . . . 81
4.2.2 Verifying the results . . . 87

A Index of symbols **89**

Introduction

This document presents the implementation of a Python 3 program that is able to calibrate Time Ordered Data (TOD) measured by a CMB space survey using the signal of the CMB dipole. The underlying assumption is that the signal of the dipole is strong enough to be used for a photometric calibration, and that the scanning strategy of the spacecraft is such that the full dynamic range of the dipole signal is explored regularly during the survey.

The algorithm used for the calibration is `Da Capo`, which was developed by Dr. E. Keihänen and is presented in Planck Collaboration (2015). The algorithm uses many concepts developed in the context of destriping (Burigana et al., 1999; Keihänen et al., 2004), and it is able to produce an estimate of three quantities at the same time:

1. A time-dependent calibration factor;
2. A timestream of offsets, used to model the $1/f$ part of the noise in the TOD;
3. An estimate of the sky map.

This implementation of the algorithm includes two Python 3 programs: the first program, `index.py`, scans the potentially large set of FITS files containing the TODs and creates an *index file*, which is used by the second program, `calibrate.py`, to decide how to split the workload among the MPI processes used in the computation; then, `calibrate.py` runs the `Da Capo` algorithm and output the set of products listed above. The most critical parts of the code have been written in Fortran 2003, in order to improve the execution speed; Python bindings have been generated automatically using `f2py`.

The source code of the program is provided in this document in its full form. It has been typeset using `noweb`[1], a *literate programming* tool which allows to produce source code files and LaTeX documentation at the same time. This document is meant to be read from the first to the last page, like a narrative: my hope is that it is readable enough for other people to understand a number of concepts related to High Performance Computing, i.e., the management of large quantities of data, the distribution of work among MPI processes, and the ways one can produce scientific results that are both reliable and reproducible.

My biggest thanks go to Elina Keihänen, which helped me a lot in understanding the details of the `Da Capo` algorithm and gave me a few suggestions that simplified my work considerably. I would also like to thank the CORE systematics working group for their suggestions in how to improve the scientific quality of the code's results: in particular, Jacques Delabrouille, Ted Kisner, Diego Molinari, and Paolo Natoli.

[1] `http://www.cs.tufts.edu/~nr/noweb/`.

Chapter 1

Presentation of the Da Capo algorithm

§ 1.1. Introduction

In this document we will provide the complete source code of a program which reads a set of Time Ordered Data (TOD) files and performs a photometric calibration of the data using the dipole signal. We use the `noweb` system[1] by Norman Ramsey to implement the code: two programs, called *tangler* and *weaver*, are used to extract the source code to compile (`notangle`) and the LaTeX file used to produce a standalone document (`noweave`) from the same document, which is a text file with extension `.nw`.

The code is written in Python 3, and it reads a set of FITS files containing Time-Ordered Information (TOI), i.e., tables containing the uncalibrated samples produced by a detector scanning the sky and the timing and pointing information. The program disentangles the noise, the dipolar signal caused by the Doppler effect induced by the motion of the spacecraft, and the sky signal, and it produces the following outputs:

1. A timestream of gain constants, used to perform the photometric calibration;
2. A calibrated sky map, in Kelvin;
3. A set of offsets, which give a rough approximation of the $1/f$ component of the noise.

The program uses the Da Capo algorithm developed by Dr. E. Keihänen and described in Planck Collaboration (2015). The algorithm is briefly presented in Sect. 1.2

§ 1.2. Modelling the output of a detector scanning the sky

In this work we consider a detector which measures the signal entering the optical system from a fixed direction of the sky. Each sample in the Time Ordered Data (TOD) produced by the detector can be modelled using the following equation:

$$V_i = G_k\big(T_i + D_i\big) + b_k + N_i, \tag{1.1}$$

where V_i is an uncalibrated sample ($[V_i] = \mathrm{V}$), G_k is the gain[2] ($[G] = \mathrm{V/K}$), T_i is the signal associated with Galactic emissions and the CMB ($[T_i] = \mathrm{K}$), D_i is the signal of the CMB dipole (it might already have been convolved with the beam response γ, i.e., $D_i \equiv \gamma * D$; in any case, $[D_i] = \mathrm{K}$), b_k is the offset used to model the slow $1/f$ noise variations ($[b_k] = \mathrm{V}$), and N_i is a white noise component ($[N_i] = \mathrm{V}$). For a realistic example of the behaviour in time of these components, see Fig. 1.1

The idea behind Eq. (1.1) is to consider the noise part as composed by a set of constant baselines b_k, which approximate the $1/f$ component of the noise, and a purely white noise component. This works if the period of each baseline is smaller than $1/f_{\mathrm{knee}}$, with f_{knee} being the knee frequency of the $1/f$ noise power spectrum. See Fig 1.2.

[1] http://www.cs.tufts.edu/~nr/noweb/.

[2] We assume from now on that the output of the detector is a voltage. However, the code runs fine as long as the detector's output is some quantity that is proportional to the temperature of the object within the main beam of the instrument.

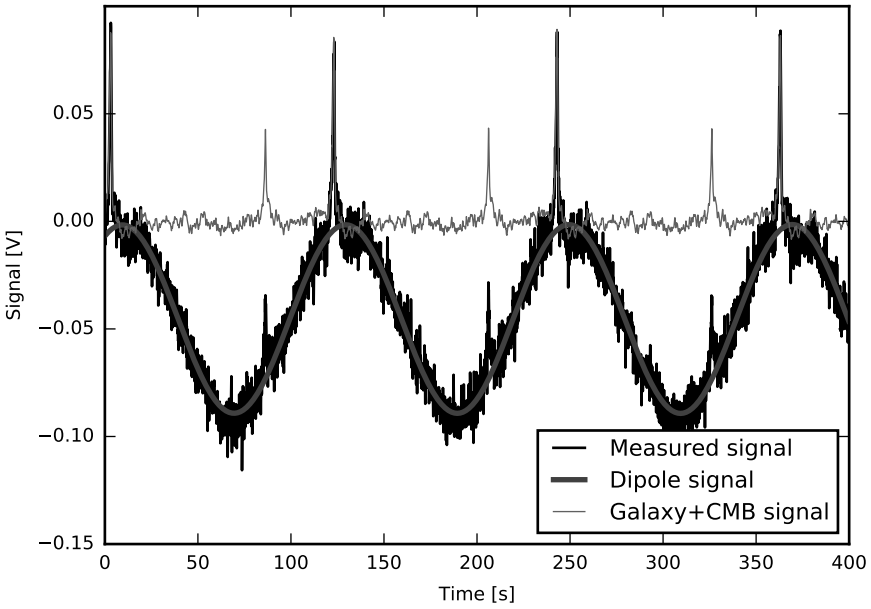


Figure 1.1: *Example of a TOD stream. The overall signal (black line) is mainly due to the sum of two components, the smooth component $G_k D_i$ (thick gray line) due to the dipole and the fast-changing $G_k T_i$ component due to the Galaxy and the CMB (thin gray line), plus a noise component due to $1/f$ fluctuations and white noise.*

The Da Capo algorithm was developed by E. Keihänen for the calibration of the Planck/LFI timelines released in 2015. Its purpose is to find an optimal estimate for the gains G_k and the offsets b_k. In the case of the 2015 Planck data release, both the gains and the offsets were calculated once every hour. I call such time frame a *calibration period*, and I index it using the symbol k: thus, G_k and b_k are the gain and offset to be used for all the samples acquired during the k-th calibration period. (Later, we will consider the more interesting case where the calibration period and the offset period are of different lengths.) The i symbol is reserved for indexing samples in a TOD. See Fig. 1.3 for an example.

The two unknowns of Eq. (1.1) are G_k and b_k, while all the other quantities are supposed either to be known (D_i, V_i) or unimportant for the computation (N_i, T_i if we apply a mask first). The value T_i is typically taken from a pixelized map, so

$$T_i = \sum_p P_{ip} m_p, \tag{1.2}$$

where P is the so-called *pointing matrix*, a rectangular matrix of size $M \times N$, where N is the number of elements in the TOD and M is the number of pixels in the map. The index p runs over all the M pixels in the map. The value of the element P_{ip} is 1 if the i-th sample in the TOD has been measured while the instrument was pointing towards pixel p, 0 otherwise.

An example of a pointing matrix is the following (assuming a TOD with length $N = 9$ and a map with $M = 3$ pixels):

$$\boldsymbol{P} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{1.3}$$

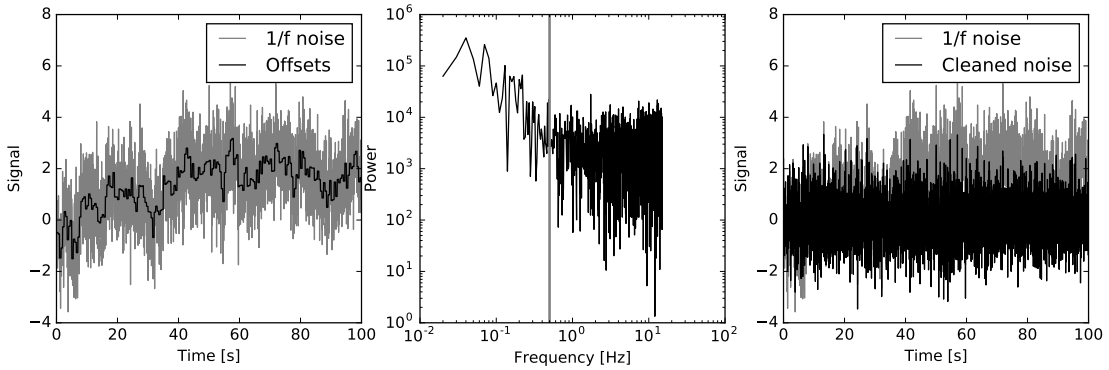


Figure 1.2: *How noise is modelled by* Da Capo*. A $1/f^2$ noise realization is approximated with a set of offsets, each calculated by averaging the noise samples over periods of 0.5 s (left). In this example, the $1/f^2$ noise knee frequency is $f_{knee} = 0.5\,Hz$, which corresponds to a period of $2\,s$: this is four times greater than the period of each offset (middle). The difference between the $1/f^2$ noise samples and the offsets reveals an almost pure white noise component (right).*

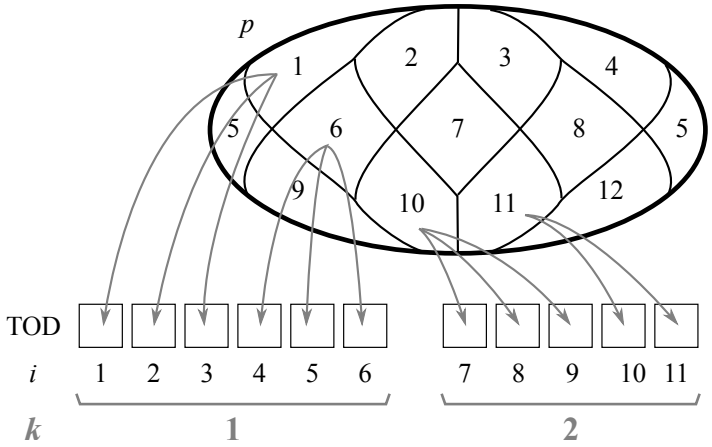


Figure 1.3: *How samples and calibration periods are indexed in this document. A minimal TOD containing $N = 11$ samples is shown here. It has been arbitrarily split into two calibration periods (indexed by k), containing 6 and 5 samples respectively. The first 6 samples are calibrated using the same gain $G_{k=1}$, the latter 5 using $G_{k=2}$. Note that, given some i, the value of k can be always derived unambiguously.*

An important property of this matrix is that the product $P^T P$ is a square diagonal matrix where each diagonal element at position pp is equal to the number of times pixel p has been observed. In our example:

$$\boldsymbol{P^T P} = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}. \tag{1.4}$$

To determine the gains, the Da Capo algorithm minimizes the quantity

$$\chi^2 = \sum_i \frac{\left(V_i - V_i^{\text{model}}\right)^2}{\sigma_i^2}, \tag{1.5}$$

where

$$V_i^{\text{model}} = G_k \left(\sum_p P_{ip} m_p + D_i \right) + b_k \tag{1.6}$$

is our model for the output of the detector, which supposes no white noise and considers a static sky

signal not dependent on time (thus, variable sources have to be masked before the application of this algorithm).

Since Eq. (1.5) is quadratic in its unknowns, Planck Collaboration (2015) linearizes it in the following way:

$$V_i^{\rm model} = G_k\left(D_i + \sum_p P_{ip} m_p^0\right) + G_k^0 \sum_p P_{ip}\left(m_p - m_p^0\right) + \left(\left(G_k - G_k^0\right)\sum_p P_{ip}\left(m_p - m_p^0\right)\right) + b_k \qquad (1.7)$$

(in the Planck 2015 paper there is a typo: the part $\sum_p P_{ip}$ is missing from the third term in the right side), where m^0 and G^0 are the results of the previous iteration. The algorithm iterates to converge towards the solution. Since the third term is of the second order, it can be dropped, and the model becomes

$$V_i^{\rm model} \approx G_k\left(D_i + \sum_p P_{ip} m_p^0\right) + G_k^0 \sum_p P_{ip}\tilde{m}_p + b_k, \qquad (1.8)$$

where

$$\tilde{m}_p = m_p - m_p^0. \qquad (1.9)$$

To further simplify Eq. (1.5), we rewrite part of the terms in Eq. (1.8) by introducing a new matrix F_{ij}, where j is an index which runs over all the available calibration periods, and a vector a which contains all the unknowns (G_k and b_k):

$$\sum_j F_{ij} a_j \equiv G_k\left(D_i + \sum_p P_{ip} m_p^0\right) + b_k. \qquad (1.10)$$

Taking as example the pointing matrix in Eq. (1.3), we imagine here that there are two calibration periods, with 5 and 3 samples respectively. Then,

$$\boldsymbol{F} = \begin{pmatrix} 1 & 0 & D_1 + m_1^0 & 0 \\ 1 & 0 & D_1 + m_1^0 & 0 \\ 1 & 0 & D_2 + m_2^0 & 0 \\ 1 & 0 & D_1 + m_1^0 & 0 \\ 1 & 0 & D_2 + m_2^0 & 0 \\ 0 & 1 & 0 & D_3 + m_3^0 \\ 0 & 1 & 0 & D_3 + m_3^0 \\ 0 & 1 & 0 & D_1 + m_1^0 \\ 0 & 1 & 0 & D_3 + m_3^0 \end{pmatrix}, \qquad \boldsymbol{a} = \begin{pmatrix} b_1 \\ b_2 \\ G_1 \\ G_2 \end{pmatrix}. \qquad (1.11)$$

Note that $\boldsymbol{F}$ contains both the dipole signal and the previous guess m^0 for the sky map. If we rewrite the second term in the right hand of Eq. (1.8) as

$$\tilde{\boldsymbol{P}}_{ip} = G_k^0 \boldsymbol{P}_{ip}, \qquad (1.12)$$

then Eq. (1.5) reaches its final form

$$\chi^2 = \left(\boldsymbol{V} - \tilde{\boldsymbol{P}}\,\tilde{\boldsymbol{m}} - \boldsymbol{F}\,\boldsymbol{a}\right)^T \boldsymbol{C}_n^{-1}\left(\boldsymbol{V} - \tilde{\boldsymbol{P}}\,\tilde{\boldsymbol{m}} - \boldsymbol{F}\,\boldsymbol{a}\right), \qquad (1.13)$$

with C_n being the noise covariance matrix for the noise component N_i. Equation (1.13) is easy to understand: it requires to minimize the difference between the measured vector $\boldsymbol{V}$ and the vector $\boldsymbol{Fa}$, which was created by scanning the sky map (dipole and Galaxy) and assuming some gains G_k and a fixed $1/f$ noise realization b_k. The term $\tilde{\boldsymbol{P}}\tilde{\boldsymbol{m}}$ takes into account the discrepancies between the map used in building $\boldsymbol{Fa}$ (produced during the previous iteration) and the new map being estimated now.

The calibration solution is found by minimizing Eq. (1.13) with respect to $\tilde{m}$ and then with respect to a. The solution a must satisfy the equation

$$\boldsymbol{Aa} = \boldsymbol{v}, \qquad (1.14)$$

where

$$\boldsymbol{A} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{F}, \tag{1.15}$$

$$\boldsymbol{v} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{V}, \tag{1.16}$$

$$\boldsymbol{Z} = \boldsymbol{I} - \tilde{\boldsymbol{P}} \left(\tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}} \right)^{-1} \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1}. \tag{1.17}$$

(The definition of $\boldsymbol{Z}$ in Planck Collaboration (2015) has a typo.)

Once a first estimate $\boldsymbol{a}_1$ for $\boldsymbol{a}$ is found, we can solve for $\tilde{\boldsymbol{m}}_1$:

$$\tilde{\boldsymbol{m}}_1 = \left(\tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}} \right)^{-1} \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \left(\boldsymbol{V} - \boldsymbol{F} \boldsymbol{a}_1 \right), \tag{1.18}$$

and then we iterate again, starting from $(\boldsymbol{a}_1, \tilde{\boldsymbol{m}}_1)$ to find an improved solution $(\boldsymbol{a}_2, \tilde{\boldsymbol{m}}_2)$ and so on until we reach convergence.

Planck Collaboration (2015) explains that the equations derived so far have serious degeneracies, that prevent them from being solved in any practical purpose. The most obvious one is the ambiguity between D_i and T_i: in the extreme case $T_i \propto D_i$ (a sky signal similar to the dipole), there would be a perfect degeneracy between G_k and the sky map. In a more realistic case, T_i would be the sum of a dipole-free component and a nonzero dipolar component, and the degeneracy would be smaller but still present. Therefore, the authors introduced a constraint to the equations, which is related to the peculiar way the dipole is used in the calibration. We only report their results here:

1. We need to introduce a new matrix, $\boldsymbol{m}_c$, which is a $M \times 2$ matrix containing the dipole and the monopole:
$$\boldsymbol{m}_c = \begin{pmatrix} D_1 & 1 \\ D_2 & 1 \\ D_3 & 1 \\ \cdots & \cdots \\ D_M & 1 \end{pmatrix}. \tag{1.19}$$
2. The definition of matrix $\boldsymbol{A}$ (Eq. 1.14) remains the same, but matrix $\boldsymbol{Z}$ becomes
$$\boldsymbol{Z} = \boldsymbol{I} - \tilde{\boldsymbol{P}} \left(\boldsymbol{M} + \boldsymbol{C}_m^{-1} \right)^{-1} \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1}, \tag{1.20}$$
with
$$\boldsymbol{M} = \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}}, \tag{1.21}$$
$$\left(\boldsymbol{M} + \boldsymbol{C}_m^{-1} \right)^{-1} = \boldsymbol{M}^{-1} - \boldsymbol{M}^{-1} \boldsymbol{m}_c \left(\boldsymbol{m}_c^T \boldsymbol{M}^{-1} \boldsymbol{m}_c \right)^{-1} \boldsymbol{m}_c^T \boldsymbol{M}^{-1}. \tag{1.22}$$
3. With these modifications, the solution has the property that there is no dipole in the map $\boldsymbol{m}$, in the sense that
$$\sum_p \boldsymbol{m}_p \boldsymbol{D}_p = 0. \tag{1.23}$$
Moreover, that the monopole of the map is zero, i.e.,
$$\langle \boldsymbol{m} \rangle = 0. \tag{1.24}$$

In the remainder of this document, I will show how the codes presented in this document can be used to perform the aforementioned calculations.

§ 1.3. How to install and build the code

Before presenting the implementation of the code, I show here how the source code can be obtained and compiled. The code has been developed on Linux machines, but it should be fairly portable to POSIX architectures (e.g., Mac OS X). Windows users might have some problems, as the code relies on the Healpy[3] library, which has not been ported to Windows at the time of writing (December 2016).

[3] https://github.com/healpy/healpy.

Variable	Default value	Notes
NOWEAVE	noweave	Part of Noweb
NOTANGLE	notangle	Part of Noweb
CPIF	cpif	Part of Noweb
TEX2PDF	lualatex	
BIBTEX	bibtex	
PYTHON	python3	
F2PY	f2py	Bundled with NumPy
AUTOPEP8	autopep8	You can use yapf as an alternative
DOCKER	sudo docker	You might want to remove sudo
MPIRUN	mpirun -n 2	Used for the integration tests
INKSCAPE	inkscape	Only used to build the PDF file

Table 1.1: *List of the variables that can be defined in* configuration.mk.

1.3.1. Installing the code from source. To install and compile the source code of this program, you will need the following dependencies:

1. Python 3.x (version 3.5 was used in the development; the code might work with earlier versions, but this has not been tested);
2. Noweb (http://www.cs.tufts.edu/~nr/noweb/), for creating the .py source codes and this documentation: you can install it under Ubuntu with the command apt install noweb;
3. NumPy (http://www.numpy.org/), for low-level vector/matrix computations;
4. SciPy (https://www.scipy.org/), for higher-level vector/matrix computations;
5. AstroPy (http://www.astropy.org/), for writing/reading FITS files;
6. MPI4py (https://pythonhosted.org/mpi4py/), for MPI communications;
7. Numba (http://numba.pydata.org/), to speed up a few Python routines;
8. Healpy (https://github.com/healpy/healpy), used to work with Healpix maps;
9. Click (http://click.pocoo.org/5/), for implementing a smart command-line interface;
10. Autopep8, to nicely re-indent the Python files produced by Noweb;
11. A fortran compiler that is supported by f2py (gfortran and Intel Fortran compiler are ok);

After you have all the requirements installed, you can download the source codes discussed in this text from the GitHub repository hosted at the following URL:

https://github.com/ziotom78/dacapo_calibration

To compile the program, just run make. If the defaults used by Makefile do not suit you, you can create a file named configuration.mk in the same directory where Makefile resides, which can provide new values for the variables listed in Table 1.1.

1.3.2. Using Docker to describe the installation. We provide here a Docker file which installs all the requirements in a virtual machine based on the miniconda3 image[4] provided by Continuum Analytics. Docker is a containerization platform that allows to spawn small Virtual Machines (VMs) running some specific task within a Linux system. Docker's advantage over other virtualization systems (e.g., QEmu, VirtualBox, VMWare...) is that such machines are really fast to start. They are typically used to run a service or a task that needs to be run in an isolated and controlled environment.

[4] https://hub.docker.com/r/continuumio/miniconda3/.

In our context, Docker is the ideal solution to test for missing dependencies in the environment we use. We describe an environment, called *image*, by means of one of Docker's preconfigured VMs available on the Docker's Hub website (https://hub.docker.com/). There are plenty of them, we'll pick one of the official Anaconda Python images, by Continuum Analytics. These virtual machines are based on well-known Linux distributions (Continuum Analytics uses Debian), and they come with a basic Python distribution already installed. For our purposes, Anaconda's VMs are not enough, as we need a few more packages not included there. So we specify a set of commands that have to be run in order to configure the VM properly.

The recipe to build a Docker image must be written in a text file, usually called `Dockerfile`. Our `Dockerfile` contains the name of the base Anaconda Python's VM, the name of the mantainer (myself), and a set of commands to be run from the command line which prepare the VM. In such commands (the lines beginning with `RUN`) we can use Debian's and Anaconda's standard tools to install programs and libraries:

7 ⟨*Dockerfile 7*⟩≡

```
FROM continuumio/miniconda3

MAINTAINER Maurizio Tomasi <maurizio.tomasi@unimi.it>

RUN apt-get -y update && \
    apt-get -y upgrade && \
    apt-get install -y \
    apt-utils \
    g++ \
    gcc \
    gfortran \
    git \
    inkscape \
    make \
    noweb \
    openmpi-bin \
    texlive-latex-base \
    texlive-latex-extra
RUN conda install \
    astropy \
    click \
    matplotlib \
    mpi4py \
    numba \
    numpy \
    pytest \
    scipy
RUN pip install \
    autopep8 \
    healpy \
    quaternionarray
CMD git clone https://github.com/ziotom78/dacapo_calibration && \
    cd dacapo_calibration && \
    make all && \
    make fullcheck
```

The last `CMD` command will be explained later.

To build the Docker image, you have to enter the `docker` directory within the source code repository and run `docker build`:

```
$ sudo docker build -t="ziotom78:dacapo" .
```

This task needs to be redone only when `Dockerfile` changes. It will take a while, since it needs to build the entire VM, running all the `RUN` command lines and downloading the necessary packages from Debian's, Anaconda's and PyPI's repositories: once this is done, subsequent runs will be much faster.

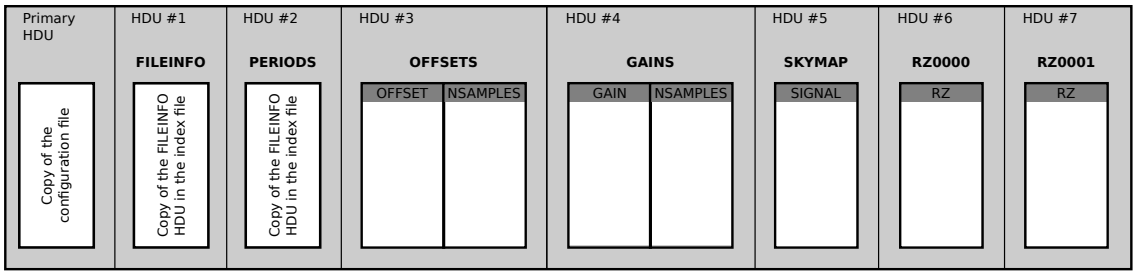


Figure 1.4: *Structure of a FITS file created by the program* `calibrate.py`. *The primary HDU and the first two extension HDU contain copies of the input files used to run the calculation; they are used to allow users to trace the history of the computation that produced the results saved in this file. HDUs #3, #4, and #5 contain the solution of the Da Capo problem, namely, the offsets and gains that are in* $\boldsymbol{a}$ *(Eq. 1.14) and the sky map* $\boldsymbol{m}$ *(Eq. 1.18). HDU #6 and following contain information about the rate of convergence of the algorithm; in this example we show just two of them, but the effective number depends on the details of the computation.*

Once the image has been built, you can run the associated *container*, which is basically an instance of the VM based on some image (refer to the Docker's documentation for more details about the terminology):

```
$ sudo docker run --name dacapo "ziotom78:dacapo"
```

When we use `docker run`, and only in this case, Docker will run the command on the line beginning with `CMD`, thus downloading the most up-to-date version of the source code presented in this document, compiling it and running the full set of tests.

The advantage of creating a Docker machine is that the full sequence of operations needed to configure a barebone system in order to install and run the code is fully documented.

1.3.3. Running the programs. The programs we are going to implement in the next chapters works on the following assumptions:

1. The user must provide one or more FITS files containing the TODs that must be used as an input for the calculation;
2. A program, `index.py` (implemented in Ch. 2), scans the FITS files one by one and counts the number of samples in each of them, optionally flagging unwanted data; the output of this program is an *index file*;
3. A second program, `calibrate.py` (Ch. 3), implements the Da Capo algorithm. It divide the TODs among the MPI processes according to the information read from the index file, it performs the calculations shown above, and it collects the results at the end of the computation. Results are saved in one FITS file, whose structure is shown in Fig. 1.4.

To see an example of how to invoke the code, suppose that you have a simulated/measured TOD saved in a set of files in some directory ($ indicates the command-line prompt, where you type commands):

```
$ ls /storage/my_data/tods/
tod_0001.fits
tod_0002.fits
tod_0003.fits
tod_0004.fits
```

Using CFITSIO's `listhead`[5] example program, we can see the structure of each file. Each of them contains one tabular HDU with a number of columns:

[5] `http://`.

```
$ listhead /storage/my_data/tods/tod_0001.fits
[...snip...]
TTYPE1  = 'TIME    '
TFORM1  = 'J       '
TUNIT1  = 's       '
TTYPE2  = 'PIXIDX  '
TFORM2  = 'J       '
TTYPE3  = 'SIGNAL  '
TFORM3  = 'D       '
TUNIT3  = 'V       '
[...snip...]
```

The first column (`TIME`) contains the time when each sample was acquired (in seconds); the second column (`PIXIDX`) contains the index of the pixels on the sky sphere that are associated with each sample; the third column contains the actual, uncalibrated samples measured by the detector. We assume that all the samples must be used in the computation, i.e., there is no need to flag the samples. (Our program will be able to discard bad data, but we are keeping this example as simple as possible.)

The first step to do is to run the program `index.py`, which read the input parameters from a file, named the *parameter file*. It is a text file which follows the traditional syntax of INI files; in our case, it is the following (file `examples/index.ini`):

```
[input_files]
path = /storage/my_data/tods
mask = tod_????.fits
hdu = 1
column = TIME

[periods]
length = 1536

[output_file]
file_name = index.fits
```

The section `input_files` specifies where to load the files, and which is the HDU and column containing timing information; this is needed by `index.py` to decide the length of each offset period. The `periods` section specifies how long each offset period must be, in seconds. Finally, the `output_file` section specifies the name of the file that will be produced by the program.

To run `index.py`, we must pass the INI file as the first and only argument on the command line:

```
python3 index.py examples/index.ini
```

The program will read the TOD files one by one and will create the file `index.fits`. We are now ready to run the `Da Capo` calibration.

The `calibrate.py` program is more complex than `index.py`, but it works in a similar way. It requires the user to specify input parameters through a parameter file, which in our example is the following:

```
[input_files]
index_file = index.fits
signal_hdu = 1
signal_column = SIGNAL
pointing_hdu = 1
pointing_columns = PIXIDX

[dacapo]
t_cmb_K = 2.72548
```

```
solsysdir_ecl_colat_rad = 1.7656131194951572
solsysdir_ecl_long_rad = 2.995889600573578
solsysspeed_m_s = 370082.2332
nside = 256
periods_per_cal_constant = 1
cg_stop_value = 1e-9

[output_file]
file_name = dacapo_results.fits
comment = Dummy
```

The `input_files` section specifies the location of the index file and the columns containing the pointing information[6] and the signal. The `dacapo` section is used to configure the way calculations are made. Finally the `output_file` section tells the program how to save the results of the computation.

The structure of the output file written by `calibrate.py` is complex: it contains the estimates for the unknowns in the Da Capo problem (gains G_k, offsets b_k, and the sky map T_i), but also many other ancillary information. This is a variation of an idea proposed by K. Riebe in a talk[7] given at the XXVI conference[8] of the Astronomical Data Analysis Software and Systems (ADASS): to have a "provenance model" that records the processing steps leading from the input raw data to the scientific products. This allows users to check the quality of the products, as well as to search for possible error sources. The structure of the FITS file shown in Fig. 1.4 allows the user to reconstruct the path from the input TODs to the estimates for G_k, b_k, and T_i.

[6] Pointing information can either be provided as a couple (colatitude, longitude), in radians, in which case the Healpix pixelization is assumed, or as a sequence of integer pixel indexes, like in this example.

[7] http://www.adass2016.inaf.it/index.php/participant-list/14-talk/124-riebe-kristin.

[8] http://www.adass2016.inaf.it/.

Chapter 2

Mapping the input data

Before implementing the algorithm described in Chapter 1, we need to decide which is the best way to load the input data to be used by `calibrate.py`, our implementation of the Da Capo algorithm. Our program is likely to be applied to large amount of data: considering one detector with a sampling frequency of $\sim 10^2$ Hz, in one year such detector will produce $\sim 10^9$ samples. Considering that each sample is 8 bytes wide and must be associated with ancillary information (e.g., pointing angles, flags), storing these data in memory is likely to require hundreds of GB of memory. Therefore, we are going to use MPI to implement a program which splits the necessary computations among a number of computing units; each unit will load only a subset of the whole data. (This approach works because, as we shall see, the operations described in Chapter 1 can be easily done concurrently.) In order to do this, we first need to devise a strategy to efficiently split the data among the MPI processes.

The subject of this chapter is the implementation of an ancillary program, `index.py`, which scans the input data and writes and *index file*. The contents of the index file will allow `calibrate.py` to quickly decide which file each MPI process needs to read.

This is the skeleton of the program, we'll discuss its implementation through this chapter:

11 ⟨*index.py* 11⟩≡

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-

from collections import namedtuple
from configparser import ConfigParser
from enum import Enum
from glob import glob
from typing import Any, List, Union
import logging as log
import os.path, sys
from astropy.io import fits
from numba import jit
import click
import numpy as np

⟨Flagging code 13a⟩
⟨Datatypes used by index.py 16⟩
⟨Functions to calculate the length of each period 15⟩
⟨Functions to load the parameters for index.py from an INI file 17a⟩
⟨Functions to write the index file to disk 24b⟩
⟨Other functions used by index.py 20c⟩

⟨Implementation of the index_main function 18b⟩
if __name__ == '__main__':
    index_main()
```

Uses `index_main` 18b.

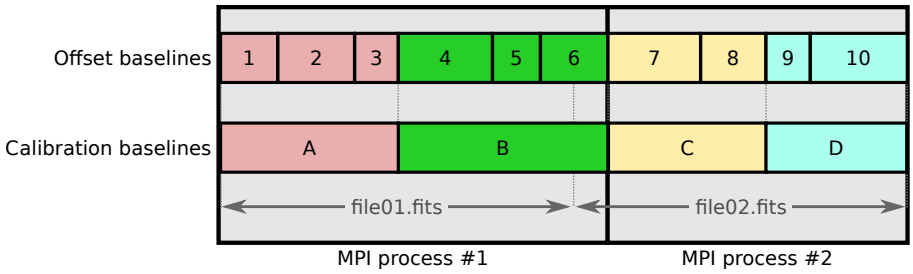


Figure 2.1: *Example of splitting a TOD into offset and calibration periods: the length of each offset is proportional to the number of samples, not on their time span. Shorter periods are likely to contain more flagged data than others. The length of an offset period (labeled as 1, 2, 3, ...) is arbitrary and variable, but calibration periods (labeled as A, B, C, D) must be made of an integer number of offset periods. Offset periods are shorter than calibration periods because they have to keep track of $1/f$ noise, which fluctuates faster than gain variations in typical situations. A further complication stems from the fact that the TOD might be split into FITS files (`file01.fits`, `file02.fits`) whose boundaries do not coincide with the boundaries of the period, as shown here. Nevertheless, we will implement* `calibrate.py` *with the constraint that each MPI process only loads an integer number of offset/calibration baselines.*

§ 2.1. What is an index file?

As said above, the amount of data to be read by our program is going to be quite relevant. It is unlikely that such data will be saved in one huge file: typically they will be split into many FITS files that will be loaded by the many MPI processes.

We introduce now a requirement on the way data are split into offset/calibration periods: we recall that a *calibration period* is a time interval for which the gain G is constant for all the samples acquired during the same interval, while an *offset period* is the same, but for the offset b (see Eq. 1.1). From now on, we allow the two periods to be different, with the constraint that a calibration period must be an integer number of offset periods. (This stems from the fact that offset periods need to keep track of $1/f$ noise, which typically show faster variations than gain changes.) See Fig. 2.1 for an example. In this way, when we will implement `calibrate.py` in Chapter 3, we will assign an integer number of calibration periods to each MPI process, and this will consequently imply that each process will get an integer number of offset periods as well.

This is the most logical way to split data in order to implement the Da Capo algorithm using MPI. However, it complicates data loading from disk, because the data might be split in files at boundaries that do not corrispond with the offset/calibration periods. The most efficient solution would be to make each MPI process load the data from disk (i.e., the first MPI process reads the first file, the second one reads the second file, and so on). Then, the processes should exchange data among them using MPI calls, until each process has the data it needs: in the example shown in Fig. 2.1, after MPI process #1 has read `file01.fits`, it should ask process #2 for the first samples in `file02.fits`. But this solution is complicated to implement: as each process must both send and receive data, the possibility of deadlocks[1] is high.

Therefore, we adopt a simpler, albeit slower, solution. We split our implementation in *two* programs:

1. The first program, `index.py`, reads the data from all the FITS files, removes any bad data (signaled by a flag column), counts the number of remaining samples in each file, and writes an *index file*;

2. The second program, `calibrate.py`, which implements the Da Capo algorithm, uses the index file to decide which files must be loaded by each MPI process.

The index file should also contain all the necessary information required for `calibrate.py` to

[1] A *deadlock* happens when some process A halts after having requested data from process B, while at the same time B is waiting for some data from A. The two process are therefore hanged, as none of them is able to continue.

remove the same bad ("flagged") data that were discarded by `index.py`: in other words, it must apply the same flagging algorithm used by `index.py`. We shall begin to implement this part of the code first.

§ 2.2. How flagged data are removed

We allow the possibility to have a column in the TOD FITS files which specifies the quality of the data, and which can therefore be used to discard bad data when necessary. Let's call this column the *flags column*. Our program allows the user to pick two methods to remove data using the flags column:

1. Data are discarded by comparing the values in the flags column with some predefined value. For instance, the flags column might contain booleans, and `index.py` could therefore be instructed to remove data whose flag is set to `FALSE` (or `TRUE`). This is a particularly simple situation, which is usually found in data generated by simulations.

2. In more realistic situations, a flag is a much richer structure than a simple true/false switch. Flags are usually unsigned integers of some bitwidth, where each bit represents a different flag. For instance, one bit might signal whether sample was acquired while the main beam was crossing some moving object (planets, etc.), while another bit could be set whenever the spacecraft is doing some nonstandard maneuver, etc. In such case, to determine if a sample must be used or not, one must use a bitmask, which is combined with the flag to check if the result is nonzero. This operation is called a *bitwise and*.

We represent the two types of flagging using an enumeration type (`Enum` is part of Python's standard library):

13a ⟨*Flagging code* 13a⟩≡ (11) 13b▷
```
class FlagType(Enum):
    equality = 0
    bitwise_and = 1
```
Defines:
`FlagType`, used in chunks 13d, 17b, and 23.

A constant source of ambiguity in implementing flagging algorithms is to decide whether to remove samples whose condition is `True` or `False`. For this reason, our code will ask explicitly which case applies. Thus, we need an additional enumeration type:

13b ⟨*Flagging code* 13a⟩+≡ (11) ◁13a 13c▷
```
class FlagAction(Enum):
    include = 0
    exclude = 1
```
Defines:
`FlagAction`, used in chunks 13d, 17b, and 23.

The `include` case specifies that only those samples for which the equality/bitwise-and condition returns `True` must be included, while the contrary applies for the `exclude` case.

Finally, we combine all the information required to properly remove flagged samples from a TOD into a new type, which combines `FlagType`, `FlagAction`, and the numerical value used in the equality/bitwise-and operation:

13c ⟨*Flagging code* 13a⟩+≡ (11) ◁13b 13d▷
```
Flagging = namedtuple('Flagging', 'flag_type flag_value flag_action')
```
Defines:
`Flagging`, used in chunks 13d, 17b, 21c, and 23b.

(If you do not know what a `namedtuple` is, refer to the Python documentation.) We are going to use this type both in `index.py` and in `calibrate.py`.

The function `flag_mask` takes a generic array as input, the array of flags, and a `Flagging` object, and it returns a mask of booleans which records the position of the "good" samples. The implementation is quite straightforward:

13d ⟨*Flagging code* 13a⟩+≡ (11) ◁13c
```
def flag_mask(flags: Any,
```

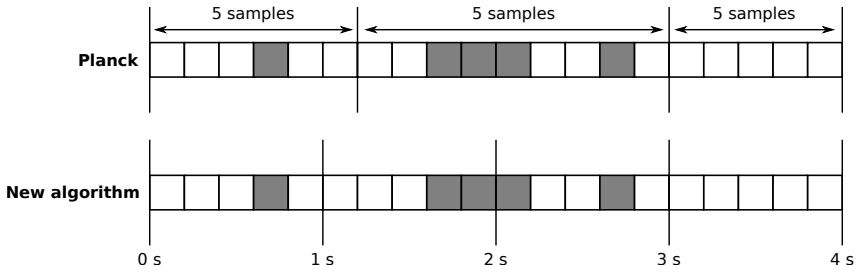


Figure 2.2: ***Top:*** *Example of the application of the algorithm used in the Planck/LFI pipeline to determine the length of a calibration/offset period. The number of samples in each period is fixed using the value for ν_{samp}, the sampling frequency (in this example, 5 samples). Thus, the effective time span covered by each period depends on the number of flagged data it contains.* ***Bottom:*** *In our implementation, we stick to the time; therefore, the number of samples in each period might change, but the time span is always the same.*

```
            flagging: Flagging) -> Any:
    if flagging.flag_type == FlagType.equality:
        mask = (flags == flagging.flag_value)
    elif flagging.flag_type == FlagType.bitwise_and:
        mask = (np.array(np.bitwise_and(flags, flagging.flag_value),
                         dtype='bool') != 0)
    else:
        raise ValueError('unknown FlagType in "flag_mask": {0}'
                         .format(flagging.flag_type))

    if flagging.flag_action == FlagAction.exclude:
        mask = np.logical_not(mask)

    return mask
```

Defines:
`flag_mask`, used in chunks 20b, 25, and 39b.
Uses `FlagAction` 13b, `Flagging` 13c, and `FlagType` 13a.

Note that we define the type for `flags` and for the return value to be `Any`: at the moment, Python's typing system is not powerful enough to deal with NumPy's arrays, so we'll use `Any` every time we expect the parameter to be a NumPy array.

§ 2.3. Splitting the data into periods

Now that we have implemented all the functions necessary to remove flagged data, let's turn to the computation of the length of each offset/calibration period. As said in Sect. 2.1, we assume that a calibration period is made by an integer number of offset periods. Therefore, offset periods are the minimal unit of time we're going to use when splitting data among the MPI processes. It is therefore crucial to record the length of each offset period in the index file.

There are many ways to compute how many samples should be in each offset period. The most simple formula considers the sampling frequency ν_{samp} of the data (typical values fall in the range 10–100 Hz), and assigns to each period a number of samples N equal to

$$N = \nu_{\text{samp}} \times \Delta t, \tag{2.1}$$

where Δt is the timescale of an offset period, e.g., 5 s. This is the approach followed by `Madam` (Keihänen et al., 2005). This formula has the advantage of being really simple, as each period has the same number of samples, but it has the drawback that if many data have been flagged, the actual interval of time spanned by the periods will vary widely and will generally be greater than Δt.

We adopt a slightly more complex algorithm. For each period, we consider the maximum number N of elements $\{x_i\}_{i=1}^{N}$ such that $x_N - x_i \leq \Delta t$. With this algorithm, each period is never longer than

Δt. (Compare this with the algorithm used by `Madam`, where each period is never *shorter* than Δt.) See Fig. 2.2 for a comparison between the two approaches.

We implement this algorithm in the function `split_into_periods`, which is quite simple. We require the caller to pre-allocate an array that will contain the number of elements for each period (the `period`) argument. Although the number of periods cannot be estimated easily before the actual computation of the lengths of each period, an upper bound to the number of periods n_p is given by the formula

$$n_p = \left\lfloor \frac{t_N - t_i}{\Delta t} \right\rfloor + 1, \tag{2.2}$$

which follows from the fact that our algorithm guarantees that each period will not be longer than Δt.

We use two counters, `sample_idx` and `period_idx`, which are used to cycle over the samples in `data` and over the output array `periods`, respectively. Since the number of samples is potentially large, we use the `Numba`[2] package to speed up the running time. Just putting the `@jit` decorator in front of the definition of the function will make Numba compile the function before executing it the first time, thus reducing its running time.

15 ⟨*Functions to calculate the length of each period* 15⟩≡ (11)

```
@jit
def split_into_periods(time_array, period_length, periods):
    "'Decide the length (in samples) of the destriping periods,
    according to the timing of each sample. Both "time_array" and
    "period length" must be expressed using the same measure unit
    (e.g., seconds, clock ticks...). The array "periods" must have
    been sized before calling this function."'

    periods[:] = 0
    sample_idx = 0
    period_idx = 0
    while sample_idx < len(time_array):
        start_time = time_array[sample_idx]
        while (sample_idx < len(time_array)) and \
              (time_array[sample_idx] - start_time < period_length):
            sample_idx += 1
            periods[period_idx] += 1

        period_idx += 1
```

Defines:
`split_into_periods`, used in chunk 21a.

As an example, consider the following script:

```
from index import split_into_periods
import numpy as np

result = np.array([0, 0, 0], dtype='int')
split_into_periods([1.0, 2.0, 3.0, 8.0, 9.0, 11.0], 3.0, result)

print(result)
```

The output is:

```
[3 2 1]
```

§ 2.4. Configuring the program through a parameter file

The operations done by `index.py` are not complex, but there are many options available to the user: the type of flagging algorithm to use, the location of the FITS files containing the data, etc. Our aim

[2] http://numba.pydata.org/.

is to make `index.py` as much versatile as possible. When the user wants to run the program, it must provide the path to a text file which describes the way the program must execute its operations. Keeping the parameters in a file ensures that it is possible to re-run the same analysis and get the same output (compare this with those programs which get such parameters from command line switches, such as `-num-of-iterations=10`, which are lost in the user's shell history).

Python provides the convenient module `configparser`, which implements a INI file reader. INI files are text files which associate a *key name* to each parameter (value). Here is an example of INI file that we want `index.py` to parse:

```
[flagging]
type = bitwise_and
value = 1
action = exclude
hdu = 1
column = FLAGS

[input_files]
path = /storage/tods
mask = tod_*.fits
hdu = 1
column = TIME

[periods]
length = 3.5

[output_file]
file_name = ./output.fits
```

As the example shows, INI files are split into sections marked by `[·]`, e.g., `[flagging]`. Each sections contains key/parameter pairs in the form `key = parameter`. The `configparser` library implements a class, `ConfigParser`, which parses this kind of files and offers a dictionary-like access to the keys.

The example above shows the way one can specify all the information needed to `index.py` to perform its job: the `flagging` section specifies which samples in the input files need to be discarded, the `input_files` section specifies the paths of the TOD files and the location of the column containing the timing of each samples (needed to properly compute the length of each offset period), and so on.

Instead of directly accessing a `ConfigParser` object during the computation, we use it to initialize a dedicated object of type `IndexConfiguration`, which contains all the fields read from the INI file. In this way, should we switch to some other library than `configparser` in the future, the number of places in the code to upgrade to the API of the new library will be limited. The object containing the configuration needed by `index.py` is defined as follows:

16 ⟨*Datatypes used by* `index.py` 16⟩≡ (11) 20a▸

```
IndexConfiguration = namedtuple('IndexConfiguration',
                                ['flagging',
                                 'flag_hdu',
                                 'flag_column',
                                 'input_path',
                                 'input_mask',
                                 'input_hdu',
                                 'input_column',
                                 'period_length',
                                 'output_file_name'])
```

Defines:
`IndexConfiguration`, used in chunks 17a and 24b.

All the fields have a one-to-one correspondence with the keys in the INI file shown above.

We need now to implement `read_index_conf_file`, the function that will parse the INI file and return a `IndexConfiguration` object.

17a ⟨*Functions to load the parameters for* `index.py` *from an INI file* 17a⟩≡ (11) 17c▷

```
def read_index_conf_file(file_name: str) -> IndexConfiguration:
    conf_file = ConfigParser()
    conf_file.read(file_name)

    try:
        ⟨Parse the flagging section and initialize local variables 17b⟩
        ⟨Parse all the other sections and initialize local variables 18a⟩
    except ValueError as e:
        log.error('invalid value found in one of the entries in "%s": %s',
                  file_name, e)
        sys.exit(1)

    return IndexConfiguration(flagging=flagging,
                              flag_hdu=flag_hdu,
                              flag_column=flag_column,
                              input_path=input_path,
                              input_mask=input_section.get('mask'),
                              input_hdu=time_hdu,
                              input_column=time_column,
                              period_length=period_length,
                              output_file_name=output_file_name)
```

Defines:
`read_index_conf_file`, used in chunk 19b.
Uses `IndexConfiguration` 16.

The reason why the parsing of the `flagging` section is separated from the others in the implementation above stems from the fact that `index.py` makes flagging an optional step. There are many cases (primarily in simulations) where flagging is not used, and all the input samples must be kept. In this case, `input.py` allows the user to drop the `flagging` section from the INI file entirely. Here is the code which parses this section:

17b ⟨*Parse the* `flagging` *section and initialize local variables* 17b⟩≡ (17a)

```
try:
    flag_section = conf_file['flagging']
except KeyError:
    flag_section = flagging = flag_hdu = flag_column = None

if flag_section is not None:
    flagging = Flagging(flag_type=FlagType[flag_section.get('type')],
                        flag_value=flag_section.getint('value'),
                        flag_action=FlagAction[flag_section.get('action')])
    flag_hdu = int_or_str(flag_section.get('hdu'))
    flag_column = int_or_str(flag_section.get('column'))
```

Uses `FlagAction` 13b, `Flagging` 13c, `FlagType` 13a, and `int_or_str` 17c.

If `conf_file['flagging']` fails because of the lack of the `flagging` section in the INI file, a `KeyError` is raised and all the variables used for flagging are set to `None`. Otherwise, the methods provided by `flag_section` (`get` and `getint`) are used to retrieve the value of the parameters.

Note the use of the `int_or_str` function to initialize `flag_hdu` and `flag_column`. The user might want to specify these values through their ordinal number or their full name (in the case of HDUs, this is the `EXTNAME`, while for columns it is simply the column's name). The function `int_to_str` assumes that the value specified by the user is a number if it can be parsed as an integer, otherwise it is a name:

17c ⟨*Functions to load the parameters for* `index.py` *from an INI file* 17a⟩+≡ (11) ◁17a

```
def int_or_str(x: str) -> Union[int, str]:
    'Convert "x" into an integer if possible. Otherwise, return it unmodified.'
    try:
```

```
            int_value = int(x)
            return int_value
        except ValueError:
            return x  # A string
```

Defines:
int_or_str, used in chunks 17b, 18a, 23b, 25, and 30a.

Thus, the user can indicate the HDU to load either by means of its position in the FITS file, as in `hdu = 1`, or by its `EXTNAME`, as in `hdu = FLAGS`.

Parsing the other sections (`input_files`, `output_file`, and `periods`) is done similarly to `flagging`. Of course, since the formers are required, the code prints an error and exit immediately if it does not find them:

18a ⟨*Parse all the other sections and initialize local variables* 18a⟩≡ (17a)

```
try:
    input_section = conf_file['input_files']
    output_section = conf_file['output_file']
    period_section = conf_file['periods']

    input_path = input_section.get('path', fallback='.')
    time_hdu = int_or_str(input_section.get('hdu', fallback=1))
    time_column = int_or_str(input_section.get('column', fallback=1))
    period_length = period_section.getfloat('length')
    output_file_name = output_section.get('file_name')
except KeyError as e:
    log.error('section/key %s not found in the configuration file "%s"',
              e, file_name)
    sys.exit(1)
```

Uses int_or_str 17c.

As above, we use `int_or_str` to initialize the variable containing the HDU and the column specified by the user.

§ 2.5. Implementing the main loop for `index.py`

Now that we have implemented `IndexConfiguration`, it's time to implement the main loop of the program. We start with the skeleton of the `main` function, which is wrapped using the `click`[3] library. Using `click` allows to build sophisticated command-line interfaces with little effort; using it for `index.py`, which just accepts the name of the INI file on the command line, might sound like an overkill; but it's just too convenient here. (Among other things, it provides an help message if the user forgets to specify the path to the INI file when running `index.py`, and it automatically prints an helpful error message if the INI file does not exist or is not readable.)

After having initialized a number of libraries and variables, the routine enters the main loop, reading all the FITS files containing the TODs are read sequentially and saving the information needed to create the index file. After all the files have been read, the results are saved by means of the function `write_output`, which is implemented later.

18b ⟨*Implementation of the* `index_main` *function* 18b⟩≡ (11)

```
@click.command()
@click.argument('configuration_file')
def index_main(configuration_file):
    ⟨Initialize index.py 19a⟩

    list_of_file_info = []  # type: List[TODFileInfo]
    periods = np.array([], dtype='int64')
    prev_times = None
    prev_flags = None
    prev_last_time = None
```

[3] http://click.pocoo.org/5/.

```
    for idx, file_name in enumerate(list_of_file_names):
        log.info('processing file "%s" (%d/%d)',
                 file_name, idx + 1, len(list_of_file_names))

        ⟨Extract basic information from the FITS file and save them into list_of_file_info 20b⟩
        ⟨Determine the length of each period 21a⟩
        ⟨Save the periods into periods 21b⟩

        log.info('file "%s" processed successfully', file_name)

    write_output(file_name=configuration.output_file_name,
                 info_list=list_of_file_info,
                 periods=periods,
                 configuration=configuration)
```

Defines:
index_main, used in chunk 11.
Uses TODFileInfo 20a and write_output 24b.

The variables prev_times and prev_flags are used when determining the length of the periods; they will be explained in due time. The variable prev_last_time is used to check that the FITS files have their time columns sorted properly.

In the initialization, we want to set up the logging system (implemented using the standard Python module logging, which we rename to log when importing it). The default output format used by logging is quite lame, so we specify a custom format which includes the date and time when the message was produced:

19a ⟨*Initialize* index.py 19a⟩≡ (18b) 19b▷

```
log.basicConfig(level=log.INFO,
                format='[%(asctime)s %(levelname)s] %(message)s')
```

After the logging system is ready, we can parse the INI file provided by the user via the command line (and passed as a parameter to index_main by the click library):

19b ⟨*Initialize* index.py 19a⟩+≡ (18b) ◁19a 19c▷

```
log.info('reading configuration file "%s"', configuration_file)
configuration = read_index_conf_file(configuration_file)
log.info('configuration file read successfully')
```

Uses read_index_conf_file 17a.

Recalling the definition for read_index_conf_file, the object configuration will be of type IndexConfiguration.

The INI file requires to specify the list of FITS files containing the TOD via two parameters, the path and a file name mask, which can use the widely known wildcards * and ?. (Refer to Sect. 2.4.) We use Python's standard function glob (from the glob module) to expand these wildcards into a list of file names, and expanduser to expand ~ into the home directory path:

19c ⟨*Initialize* index.py 19a⟩+≡ (18b) ◁19b

```
list_of_file_names = \
    sorted(glob(os.path.join(os.path.expanduser(configuration.input_path),
                             configuration.input_mask)))
```

The use of sorted here is extremely important. We scan the TOD files one by one; since the code that determines the length of each period (provided below) assumes that the time of each samples increases monotonically, we must be sure that the files are read in the right order. However, the order of the elements in the list returned by glob is not specified. We therefore assume that the name of the FITS files has a fixed form with an increasing index, so that a lexicographical sort of the names will provide the right ordering. (A more robust, although more complicated, approach would be to scan the files once to determine the time of the first sample in each file, and then to sort them before reading their data.)

Within the for loop, the first task to do is to load the column containing the timing of each sample, filter the data, and save the basic information about this file and its data in a dedicated data structure,

TODFileInfo. This structure is defined as follows:

20a ⟨*Datatypes used by* index.py 16⟩+≡ (11) ◁16 21c▷
```
TODFileInfo = namedtuple('TODFileInfo',
                         ['file_name',
                          'mod_time',
                          'num_of_samples',
                          'num_of_unflagged_samples'])
```
Defines:
TODFileInfo, used in chunks 18b, 20b, 23–25, and 37a.

We now enter the for loop. Remember that file_name is the variable used in the loop to hold the name of the file to be loaded during the current iteration. Here is the code which creates TODFileInfo objects:

20b ⟨*Extract basic information from the FITS file and save them into* list_of_file_info 20b⟩≡ (18b)
```
times = read_column(file_name=file_name,
                    hdu=configuration.input_hdu,
                    column=configuration.input_column)

if (prev_last_time is not None) and (times[0] < prev_last_time):
    log.error(('column {0} in HDU {1} of the FITS files is not '
               'sorted in ascending order ({2} > {3})')
              .format(configuration.input_column,
                      configuration.input_hdu,
                      times[0], prev_last_time))
    sys.exit(1)

prev_last_time = times[-1]
num_of_samples = len(times)

if configuration.flagging:
    flags = read_column(file_name=file_name,
                        hdu=configuration.flag_hdu,
                        column=configuration.flag_column)

    mask = flag_mask(flags=flags,
                     flagging=configuration.flagging)
    times = times[mask]

file_info = TODFileInfo(file_name=os.path.abspath(file_name),
                        mod_time=os.path.getmtime(file_name),
                        num_of_samples=num_of_samples,
                        num_of_unflagged_samples=len(times))
list_of_file_info.append(file_info)
```
Uses flag_mask 13d, read_column 20c, and TODFileInfo 20a.

In order to be useful, the path to the input FITS files should be absolute, i.e., paths such as ../data/file1.fits must have their .. part be expanded. This is the reason behind the call to os.path.abspath. We record the file last modification time (through os.path.getmtime) because in this way it is easy to catch stale index files (i.e., situations that arises when one updates the input TOD files but forgets to recreate the index file).

The code above makes use of the function read_column, which is just a tiny wrapper around a few astropy.io.fits' functions:

20c ⟨*Other functions used by* index.py 20c⟩≡ (11)
```
def read_column(file_name: str,
                hdu: Union[int, str],
                column: Union[int, str]):
    with fits.open(file_name) as f:
        return f[hdu].data.field(column)
```

Defines:
`read_column`, used in chunk 20b.

Note that `astropy.io.fits` allows to specify HDUs and column names either by means of one-based integers (positional indexes) or strings (names). Thus, our use of `int_or_str` to parse HDUs and columns from the INI files is justified.

The following code performs the harder task to determine the length of each period within the current file. As shown in Fig. 2.1, it is usually the case that the boundary between two FITS files falls within a period: the period marked as "6" has one half which falls in `file01.fits` and the other half which falls in `file02.fits`). Therefore, we need special care for those periods which fall at the file boundaries. Here is the code:

21a ⟨*Determine the length of each period* 21a⟩≡ (18b)

```
if prev_times is not None:
    times = np.concatenate((prev_times, times))
    if configuration.flagging is not None:
        flags = np.concatenate((prev_flags, flags))

num_of_periods = int((times[-1] - times[0]) // configuration.period_length) + 1
cur_periods = np.zeros(num_of_periods, dtype='int64')
split_into_periods(times, configuration.period_length, cur_periods)
cur_periods = cur_periods[cur_periods > 0]

if idx + 1 < len(list_of_file_names):
    last_period_len = cur_periods[-1]
    prev_times = times[-last_period_len:]
    if configuration.flagging is not None:
        prev_flags = flags[-last_period_len:]

    cur_periods = cur_periods[0:-1]
```

Uses `split_into_periods` 15.

Since it is impossible to tell if period "6" is complete before reading `file02.fits`, we save those samples in `times` and `flags` which belong to the last period into `prev_times` and `prev_flags`, and we ignore them in the current iteration (thus cutting the last element from the list `cur_periods`). In the next iteration, we call `np.concatenate` to prepend the tail of the previous file to `times` and `flags`.

Finally, we can append `cur_periods` to the overall list of periods:

21b ⟨*Save the periods into* `periods` 21b⟩≡ (18b)

```
periods = np.concatenate((periods, cur_periods))
```

This completes the implementation of `index.py`'s main loop: at the end of the `for` loop, the variables `file_info_list` and `periods` contain all the information needed to write the index file to disk. This last task is the subject of the next section.

§ 2.6. Writing the index file

We now turn to the implementation of the code that reads and write index information in FITS files. We implement a class, `IndexFile`, whose purpose is to keep together all the information needed and to provide a couple of methods to read and write them to files:

21c ⟨*Datatypes used by* `index.py` 16⟩+≡ (11) ◁20a

```
class IndexFile:
    def __init__(self,
                 input_hdu: Union[int, str]=1,
                 input_column: Union[int, str]=0,
                 flag_hdu: Union[int, str]=1,
                 flag_column: Union[int, str]=1,
                 flagging: Flagging=None):
        self.flagging = flagging
        self.tod_info = []
```

```
            self.periods = None
            self.period_length = 0.0

            self.input_hdu = input_hdu
            self.input_column = input_column
            self.flag_hdu = flag_hdu
            self.flag_column = flag_column

        def store_in_hdus(self) -> List:
            ⟨Create file_info_hdu, containing file information 22a⟩
            ⟨Create period_hdu, containing the length of each period 22b⟩
            ⟨Set the header of the two HDUs 23a⟩
            return [file_info_hdu, period_hdu]

        def load_from_fits(self, file_name: str, first_idx=-1, last_idx=-1):
            with fits.open(file_name) as f:
                fileinfo_hdu = f['FILEINFO']
                ⟨Load the FILEINFO HDU from fileinfo_hdu 23b⟩

                periods_hdu = f['PERIODS']
                ⟨Load the PERIODS HDU from periods_hdu 24a⟩
```

Defines:
IndexFile, used in chunks 24b, 25, 32b, 38b, and 40a.
Uses Flagging 13c.

Let's discuss the implementation of `IndexFile.store_in_hdus` first. The first data HDU contains a table with the name of each FITS file read in the main loop, the last modification time of the file, the overall number of samples, and the number of samples that were kept after removing bad (flagged) data. Since we need to specify the number of characters in FITS columns which contain string, we must first compute the maximum length of the paths to be written in the column: this is the purpose of the variable `fname_format`:

22a ⟨*Create* `file_info_hdu`, *containing file information* 22a⟩≡ (21c)

```
fname_format = '{0}A'.format(max([len(x.file_name) for x in self.tod_info]))
file_info_columns = [fits.Column(name='FILENAME',
                                 format=fname_format,
                                 array=[x.file_name for x in self.tod_info]),
                     fits.Column(name='MODTIME',
                                 format='1D',
                                 array=[x.mod_time for x in self.tod_info]),
                     fits.Column(name='NSAMPLES',
                                 format='1K',
                                 array=[x.num_of_samples
                                        for x in self.tod_info]),
                     fits.Column(name='NUNFLAG',
                                 format='1K',
                                 array=[x.num_of_unflagged_samples
                                        for x in self.tod_info])]
file_info_hdu = fits.BinTableHDU.from_columns(file_info_columns)
file_info_hdu.name = 'FILEINFO'
```

The PERIODS HDU has a much simpler structure:

22b ⟨*Create* `period_hdu`, *containing the length of each period* 22b⟩≡ (21c)

```
period_columns = [fits.Column(name='NSAMPLES',
                              format='1K',
                              array=self.periods)]
period_hdu = fits.BinTableHDU.from_columns(period_columns)
period_hdu.name = 'PERIODS'
```

Before writing the two HDUs in a FITS file, we set a number of header keywords. The most important keywords are those that specify the flagging, as `calibrate.py` will need to redo the flagging

exactly as it was done when creating the index file.

23a ⟨*Set the header of the two HDUs* 23a⟩≡ (21c)

```
if self.flagging is not None:
    list_of_flag_types = ', '.join(['"{0}"'.format(x.name)
                                    for x in FlagType])
    list_of_flag_actions = ', '.join(['"{0}"'.format(x.name)
                                      for x in FlagAction])

    file_info_hdu.header['FTYPE'] = (self.flagging.flag_type.name,
                                     list_of_flag_types)
    file_info_hdu.header['FVALUE'] = (self.flagging.flag_value,
                                      'Value used for flagging')
    file_info_hdu.header['FACTION'] = (self.flagging.flag_action.name,
                                       'Flag action ({0})'
                                       .format(list_of_flag_actions))
    file_info_hdu.header['FHDU'] = (self.flag_hdu,
                                    'Flags column HDU')
    file_info_hdu.header['FCOL'] = (self.flag_column,
                                    'Flags column number/name')

file_info_hdu.header['INPHDU'] = (self.input_hdu,
                                  'Time column HDU')
file_info_hdu.header['INPCOL'] = (self.input_column,
                                  'Time column number/name')
file_info_hdu.header['NSAMPLES'] = (sum([x.num_of_samples
                                         for x in self.tod_info]),
                                    'Total number of samples')
file_info_hdu.header['NUNFLAG'] = (sum([x.num_of_unflagged_samples
                                        for x in self.tod_info]),
                                   'Total number of unflagged samples')

period_hdu.header['LENGTH'] = (self.period_length, 'Length of a period')
```

Uses `FlagAction` 13b and `FlagType` 13a.

We can now turn to the implementation of `IndexFile.load_from_fits`. Starting from version 1.1, this function allows to specify the first and last index (both inclusive) of the `TODFileInfo` array, in order to load the definition of a subset of the whole TOD: this is useful for debugging. Here is the code that reads back the `FILEINFO` HDU:

23b ⟨*Load the* `FILEINFO` *HDU from* `fileinfo_hdu` 23b⟩≡ (21c)

```
fileinfo_hdr = fileinfo_hdu.header
if 'FTYPE' in fileinfo_hdr:
    self.flagging = \
                    Flagging(flag_type=FlagType[fileinfo_hdr['FTYPE']],
                             flag_action=FlagAction[fileinfo_hdr['FACTION']],
                             flag_value=fileinfo_hdr['FVALUE'])
    self.flag_hdu = int_or_str(fileinfo_hdr['FHDU'])
    self.flag_column = int_or_str(fileinfo_hdr['FCOL'])
else:
    self.flagging = self.flag_hdu = self.flag_column = None

self.tod_info = [TODFileInfo(file_name=x[0].decode(),
                             mod_time=x[1],
                             num_of_samples=x[2],
                             num_of_unflagged_samples=x[3])
                 for x in fileinfo_hdu.data.tolist()]
if first_index < 0:
    _first = 0
else:
```

```
        _first = first_index

    if last_index < 0:
        _last = len(self.tod_info) - 1
    else:
        _last = min(len(self.tod_info) - 1, last_index)

    self.tod_info = self.tod_info[first_index:(last_index + 1)]
    self.input_hdu = fileinfo_hdu.header['INPHDU']
    self.input_column = fileinfo_hdu.header['INPCOL']
```

Uses FlagAction 13b, Flagging 13c, FlagType 13a, int_or_str 17c, and TODFileInfo 20a.

Note that if first_index and last_index are not provided, the code reads all the TODs (as it was the case in version 1.0).

And here is the code which reads the PERIODS HDU:

24a ⟨*Load the* PERIODS *HDU from* periods_hdu 24a⟩≡ (21c)

```
    self.periods = periods_hdu.data.field('NSAMPLES')
    self.period_length = periods_hdu.header['LENGTH']
```

To save the results in index.py, we use the IndexFile class we have just defined. The following function is a simple wrapper around IndexFile.store_in_hdu:

24b ⟨*Functions to write the index file to disk* 24b⟩≡ (11)

```
    def write_output(file_name: str,
                     info_list: List[TODFileInfo],
                     periods: Any,
                     configuration: IndexConfiguration):
        log.info('writing file "%s"', file_name)

        index_file = IndexFile(input_hdu=configuration.input_hdu,
                               input_column=configuration.input_column,
                               flag_hdu=configuration.flag_hdu,
                               flag_column=configuration.flag_column,
                               flagging=configuration.flagging)
        index_file.tod_info = info_list
        index_file.periods = periods

        hdu_list = [fits.PrimaryHDU()] + index_file.store_in_hdus()
        fits.HDUList(hdu_list).writeto(file_name, clobber=True)
        log.info('file "%s" written successfully', file_name)
```

Defines:
write_output, used in chunk 18b.
Uses IndexConfiguration 16, IndexFile 21c, and TODFileInfo 20a.

Chapter 3

Implementing Da Capo

In this chapter we provide a full implementation of the algorithm described in Chapter . The purpose is to read a potentially large number of FITS files containing the input TODs, and to fit the data with a map containing the dipole signal in order to extract a set of offsets and gains. We will call this program `calibrate.py`, and we are going to implement it in Python and Fortran (the latter being used for the most number-crunching intensive tasks).

§ 3.1. Overall structure of the program

Here is the skeleton of the program; we will fill all the details later:

25 ⟨*calibrate.py* 25⟩≡

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-

import io
import sys
from copy import copy
from collections import namedtuple
from configparser import ConfigParser
from datetime import datetime

import click
from typing import List, Any
import numpy as np, scipy
import healpy
import logging as log
from numba import jit
from index import int_or_str, flag_mask, TODFileInfo, IndexFile
from astropy.io import fits
from mpi4py import MPI
import ftnroutines

⟨Miscellaneous functions 27b⟩
⟨Datatypes used by calibrate.py 28⟩
⟨Functions to divide the TOD among the MPI processes 33a⟩
⟨Functions to set up calibrate.py 29⟩
⟨Estimation of the dipole signal 41a⟩
⟨Matrix/vector multiplication functions 49a⟩
⟨Implementation of the conjugate gradient method 56c⟩
⟨CG preconditioners and error estimation 60⟩
⟨Implementation of the Da Capo algorithm 65c⟩
```

⟨*Implementation of* `calibrate_main` 26b⟩

```
if __name__ == '__main__':
    calibrate_main()
```

Uses `flag_mask` 13d, `IndexFile` 21c, `int_or_str` 17c, and `TODFileInfo` 20a.

Note the line `import ftnroutines as ftn`. We are going to implement a few number-crunching routines in Fortran, in order to run faster (*way* faster!). We will use `f2py`, an awesome tool provided by the NumPy package which allows to seamlessy call Fortran routines within Python. We therefore need to have a fortran file, where to put these low-level routines. Here is the skeleton:

26a ⟨*ftnroutines.f90* 26a⟩≡

```
! Compile this file using the following command:
!
!    f2py -c -m ftnroutines -f90flags=-std=f2003 ftnroutines.f90
```

⟨*Fortran routines used by* `calibrate.py` 35a⟩

§ 3.2. Implementation of the main loop for `calibrate.py`

To implement the main function, we make use of the excellent `click` library. We specify that the executable takes only one required argument, the configuration file. There are a few switches that tune the way logging messages are produced:

- The `-debug` flag increases the verbosity of the code, and it is useful for debugging;
- By default, only the MPI process with rank #0 prints messages on the screen, unless something critical happens. The `-full-log` makes all the MPI process write their own log messages: this is useful to debug deadlocks. If this flag is used, the messages are no longer printed on the console, but in a separate file. The name of the file is derived from the variable `DEFAULT_LOGFILE_MASK`, which has the string `%04d` which is filled with the zero-based rank number of the current MPI process: therefore, each process writes its own log file.
- The user can provide a custom file name for the log files, thus overriding `DEFAULT_LOGFILE_MASK`. This can be done using the `-logfile` flag. The file name must have an escape sequence like `%d`, otherwise all the MPI processes will write into the same file, with unforeseeable consequences.

The skeleton of the function lists the actions that we will implement in the rest of this chapter.

26b ⟨*Implementation of* `calibrate_main` 26b⟩≡ (25)

```
DEFAULT_LOGFILE_MASK = 'calibrate_%04d.log'

@click.command()
@click.argument('configuration_file')
@click.option('-debug/-no-debug', 'debug_flag',
              help='Print more debugging information during the execution')
@click.option('-full-log/-no-full-log', 'full_log_flag',
              help='Make every MPI process write log message to files'
              ' (use -logfile to specify the file name)')
@click.option('-i', '-index-file', 'indexfile_path', default=None, type=str,
              help='Specify the path to the index file to use '
              '(overrides the one specified in the parameter file).')
@click.option('-logfile', 'logfile_mask', default=DEFAULT_LOGFILE_MASK,
              help='Prints (a subset of) logging messages on the screen'
              ' (default is "{0}")'.format(DEFAULT_LOGFILE_MASK))
def calibrate_main(configuration_file: str, debug_flag: bool,
                   full_log_flag: bool, logfile_mask: str,
                   indexfile_path: str):
    mpi_comm = MPI.COMM_WORLD
```

```
mpi_size = mpi_comm.Get_size()
mpi_rank = mpi_comm.Get_rank()

⟨Initialize the logging system for calibrate.py 27a⟩
⟨Read the configuration file and build a CalibrateConfiguration object 32a⟩
⟨Load the index file 32b⟩
⟨Determine how input data should be split into baselines 34⟩
⟨Load the part of the TOD that is assigned to this MPI process into the tod variable 40b⟩
⟨Build the monopole and dipole maps 41b⟩
⟨Configure the preconditioner 65b⟩

da_capo_results = da_capo(mpi_comm,
                          voltages=tod.signal, pix_idx=tod.pix_idx,
                          samples_per_ofsp=local_samples_per_ofsp,
                          samples_per_gainp=local_samples_per_gainp,
                          mc=mc,
                          mask=mask,
                          threshold=configuration.dacapo_stop,
                          max_iter=configuration.dacapo_maxiter,
                          max_cg_iter=configuration.cg_maxiter,
                          cg_threshold=configuration.cg_stop,
                          pcond=pcond)

⟨Gather the results from the MPI processes and save them in coll_offsets and coll_gains 68c⟩
if mpi_rank == 0:
    ⟨Save the results of the calibration in the output file 69a⟩
```

Uses da_capo 66b.

§ 3.3. The logging system

We use Python's `logging` module, which we rename to `log` in order to short a bit function calls. We do not want too much output, so we enable logging only for the root MPI process (which is going to do more stuff than the others). Keep this in mind when you find `log.info` calls elsewhere in the code.

27a ⟨*Initialize the logging system for* `calibrate.py` 27a⟩≡ (26b)

```
if debug_flag:
    log_level = log.DEBUG
else:
    log_level = log.INFO

log_format = '[%(asctime)s %(levelname)s MPI#{0:04d}] %(message)s'.format(mpi_rank)
if full_log_flag:
    log.basicConfig(level=log_level, filename=(logfile_mask % mpi_rank),
                    filemode='w', format=log_format)
else:
    if mpi_rank == 0:
        log.basicConfig(level=log_level, format=log_format)
    else:
        log.basicConfig(level=log.CRITICAL)
```

§ 3.4. Keeping track of the time spent by the program

It is usually a good idea to keep track of the time spent in doing calculations. This allows to see if any change in the codebase improves computation times or introduces a performance regression. We will use a very simple class, `Profile`, to keep track of how much time is spent during the execution:

27b ⟨*Miscellaneous functions* 27b⟩≡ (25) 68b▹

```
class Profiler:
```

```
    def __init__(self):
        self.start_time = None
        self.tic()

    def tic(self):
        "'Record the current time."'
        self.start_time = datetime.now()

    def toc(self):
        "'Return the elapsed time (in seconds) since the last call to tic/toc."'
        now = datetime.now()
        diff = now - self.start_time
        self.start_time = now
        return diff.total_seconds()
```

Defines:
Profiler, used in chunk 66b.
tic, never used.
toc, used in chunk 66b.

The idea of the class is the following: before running some long computation, the `tic` method is called. After the computation, the `toc` method will return the number of seconds elapsed since the last call to `tic`. Calling `toc` one more time will return the time elapsed since the last call to `toc`, and so on.

§ 3.5. Configuring the program

The program `calibrate.py` is going to be considerably more complex than `index.py`, and therefore it is going to require a large number of parameters to start. As we did for `index.py` (see Sect. 2.4), we use Python's `configparser` module to implement the ability to read input parameters from INI files. The set of parameters required by the program is listed in Table 3.1; we are not going to explain them all here, but we will refer many times to this table in the following sections.

As we did for `index.py` (see `IndexConfiguration`), we define a data structure that holds all the settings loaded from an INI file:

28 ⟨*Datatypes used by* `calibrate.py` 28⟩≡ (25) 44▷

```
CalibrateConfiguration = namedtuple('CalibrateConfiguration',
                                    ['index_file',
                                     'first_tod_index', 'last_tod_index',
                                     'signal_hdu', 'signal_column',
                                     'pointing_hdu', 'pointing_columns',
                                     't_cmb_k', 'solsys_speed_vec_m_s',
                                     'frequency_hz',
                                     'nside', 'mask_file_path',
                                     'periods_per_cal_constant',
                                     'cg_stop', 'cg_maxiter',
                                     'dacapo_stop', 'dacapo_maxiter',
                                     'pcond', 'output_file_name', 'save_map',
                                     'save_convergence',
                                     'comment',
                                     'parameter_file_contents'])
```

Defines:
CalibrateConfiguration, used in chunks 29, 31d, 38b, and 40a.

We are now to implement a function, `read_calibrate_conf_file`, which has the purpose of reading an INI file and build a `CalibrateConfiguration` object; the path to the INI file is the only argument to the function. We are going to use `configparser`'s functions like `getint` and `getfloat`, which raise a `ValueError` exception if the data type of the key in the INI file is not what we expected (for instance, a line like "`nside = 16.32`", see Table 3.1 for the expected types). Therefore, after we create a `ConfigParser` object and read the INI file, we wrap all the routines that extract the

Section	Parameter
input_files	index_file
	first_tod_index]& \texttt{int}& 0-based index of the first TOD file to read, or -1 to read from the f
	signal_column
	pointing_columns
dacapo	t_cmb_k
	solsysdir_ecl_colat_rad
	solsysdir_ecl_long_rad
	solsysspeed_m_s
	frequency_hz]& \texttt{float}& Frequency (in Hertz) to use for the calcula
	mask
	periods_per_cal_constant
	cg_stop_value
	cg_max_iterations
	dacapo_stop_value
	dacapo_max_iterations
	pcond
output	file_name
	save_map
	save_convergence_information
	comment

Table 3.1: *Parameters to be specified in the INI file read by* `calibrate.py`.

parameters in a `try...except` block and provide human-readable error messages for this kind of errors (remember that `log.error` only prints an error on the root MPI process, see Sect 3.3):

29 ⟨*Functions to set up* `calibrate.py` 29⟩≡ (25)

```
def read_calibrate_conf_file(file_name: str) -> CalibrateConfiguration:
    log.debug('entering read_calibrate_conf_file')
    conf_file = ConfigParser()
    conf_file.read(file_name)

    try:
        ⟨Read the INI section index_file 30a⟩
        ⟨Read the INI section dacapo 30b⟩
        ⟨Read the INI section output 31b⟩
    except ValueError as e:
        log.error('invalid value found in one of the entries in "%s": %s',
                  file_name, e)

    ⟨Build and return a CalibrateConfiguration object 31c⟩
```

Defines:
`read_calibrate_conf_file`, used in chunk 32a.
Uses `CalibrateConfiguration` 28.

The first section we interpret in the INI file is `index_file`. For each key under `index_file`, we initialize a local variable with the value found in the INI file. At the end of `read_calibrate_conf_file`, we will use them to build a `CalibrateConfiguration` object. We allow the user to specify either the name or the number of the HDUs/table columns, as we did in the implementation of `index.py`. Therefore, we reuse the function `int_or_str` we implemented before (see the list of imports at the beginning of `calibrate.py`):

30a ⟨*Read the INI section* `index_file` 30a⟩≡ (29)

```
input_sect = conf_file['input_files']

index_file = input_sect.get('index_file', None)
first_tod_index = input_sect.get('first_tod_index', -1)
last_tod_index = input_sect.get('last_tod_index', -1)
signal_hdu = int_or_str(input_sect.get('signal_hdu'))
signal_column = int_or_str(input_sect.get('signal_column'))
pointing_hdu = int_or_str(input_sect.get('pointing_hdu'))
pointing_columns = [int_or_str(x.strip())
                    for x in input_sect.get('pointing_columns').split(',')]
```

Uses `int_or_str` 17c and `split` 33b.

We do not force the user to provide an `index_file` line in the configuration file, as this information can be provided from the command line using the `-index-file` switch. We'll check later if an index file has been actually provided in one way or in the other. In order to preserve compatibility with version 1.0 of this code, we do not need the user to specify `first_tod_index` and `last_tod_index`: in this case, the code will analyze all the TOD files.

We allow the user to provide TOD files containing pointing information either as colatitude/longitude pairs or pixel numbers, and the number of elements in the list will allow us to discriminate between these two cases. However, we do not make this distinction here: for the moment, we are allowing `pointing_columns` to be a list of integer/strings of any length. We will check how many columns have been actually specified later.

The next section we read is `dacapo`. Here we do not store the direction of the solar system velocity and its speed, but we convert it directly into a NumPy 3-element vector:

30b ⟨*Read the INI section* `dacapo` 30b⟩≡ (29) 31a▷

```
dacapo_sect = conf_file['dacapo']

t_cmb_k = dacapo_sect.getfloat('t_cmb_k')
solsysdir_ecl_colat_rad = dacapo_sect.getfloat('solsysdir_ecl_colat_rad')
solsysdir_ecl_long_rad = dacapo_sect.getfloat('solsysdir_ecl_long_rad')
solsysspeed_m_s = dacapo_sect.getfloat('solsysspeed_m_s')
solsys_speed_vec_m_s = solsysspeed_m_s * \
    np.array([np.sin(solsysdir_ecl_colat_rad) * np.cos(solsysdir_ecl_long_rad),
              np.sin(solsysdir_ecl_colat_rad) * np.sin(solsysdir_ecl_long_rad),
              np.cos(solsysdir_ecl_colat_rad)])

freq_str = dacapo_sect.get('frequency_hz', fallback=None)
if freq_str.lower in ['', 'none', 'nan', 'no']:
    frequency_hz = None
else:
    frequency_hz = float(freq_str)

nside = dacapo_sect.getint('nside')
mask_file_path = dacapo_sect.get('mask', fallback=None)
periods_per_cal_constant = dacapo_sect.getint('periods_per_cal_constant')
cg_stop = dacapo_sect.getfloat('cg_stop_value', 1e-9)
cg_maxiter = dacapo_sect.getint('cg_max_iterations', 100)
dacapo_stop = dacapo_sect.getfloat('dacapo_stop_value', 1e-9)
dacapo_maxiter = dacapo_sect.getint('dacapo_max_iterations', 20)
pcond = dacapo_sect.get('pcond').lower()
```

The default values we provide for `cg_stop`, `cg_maxiter`, `dacapo_stop`, and `dacapo_maxiter` are rought estimates; we will come back to this topic later, when we will present the implementation of the conjugate gradient algorithm.

After we read the parameters, we check the meaningfulness of a few of them:

31a ⟨*Read the INI section* `dacapo` 30b⟩+≡ (29) ◁30b

```
try:
    if not healpy.isnsideok(nside):
        raise ValueError('invalid NSIDE = {0}'.format(nside))
    if cg_stop < 0.0:
        raise ValueError('cg_stop_value ({0:.3e}) should not be negative'
                         .format(cg_stop))
    if dacapo_stop < 0:
        raise ValueError('dacapo_stop_value ({0:.3e}) should not be negative'
                         .format(dacapo_stop))
    if periods_per_cal_constant < 1:
        raise ValueError('periods_per_cal_constant ({0}) should be greater than zero'
                         .format(periods_per_cal_constant))
    if cg_maxiter < 0:
        raise ValueError('cg_maxiter (%d) cannot be negative'.format(cg_maxiter))
    if dacapo_maxiter < 0:
        raise ValueError('dacapo_maxiter (%d) cannot be negative'.format(dacapo_maxiter))
except ValueError as e:
    log.error(e)
    sys.exit(1)
```

We raise an exception and catch it immediately so that we do not duplicate the calls to `log.error` and `sys.exit` for each case.

The last part of the INI file includes the paths of the files that are to be created by `calibrate.py`. Nothing new here:

31b ⟨*Read the INI section* `output` 31b⟩≡ (29)

```
output_sect = conf_file['output']

output_file_name = output_sect.get('file_name')
save_map = output_sect.getboolean('save_map', fallback=True)
save_convergence = output_sect.getboolean('save_convergence_information', fallback=True)
comment = output_sect.get('comment', fallback=None)
```

We apply the idea discussed in Chapter 1 of keeping a "provenance model" for scientific products by saving a copy of the parameter file in the `Configuration` object, with the purpose of saving a copy in the output files. We decode the string in an array of bytes and keep them in a NumPy array:

31c ⟨*Build and return a* `CalibrateConfiguration` *object* 31c⟩≡ (29) 31d▷

```
param_file_contents = io.StringIO()
conf_file.write(param_file_contents)
param_file_contents = np.array(list(param_file_contents.getvalue().encode('utf-8')))
```

We are now ready to build a `CalibrateConfiguration` object. It is just a matter of putting together all the local variables initialized before:

31d ⟨*Build and return a* `CalibrateConfiguration` *object* 31c⟩+≡ (29) ◁31c

```
return CalibrateConfiguration(index_file=index_file,
                              first_tod_index=first_tod_index,
                              last_tod_index=last_tod_index,
                              signal_hdu=signal_hdu,
                              signal_column=signal_column,
                              pointing_hdu=pointing_hdu,
                              pointing_columns=pointing_columns,
                              t_cmb_k=t_cmb_k,
                              solsys_speed_vec_m_s=solsys_speed_vec_m_s,
                              frequency_hz=frequency_hz,
                              nside=nside,
```

```
                                    mask_file_path=mask_file_path,
                                    periods_per_cal_constant=periods_per_cal_constant,
                                    cg_stop=cg_stop,
                                    cg_maxiter=cg_maxiter,
                                    dacapo_stop=dacapo_stop,
                                    dacapo_maxiter=dacapo_maxiter,
                                    pcond=pcond,
                                    output_file_name=output_file_name,
                                    save_map=save_map,
                                    save_convergence=save_convergence,
                                    comment=comment,
                                    parameter_file_contents=param_file_contents)
```

Uses CalibrateConfiguration 28.

We can now turn back to the implementation of `calibrate_main`. We must take into account the possibility that the user specified an index file through the command line, so we update `configuration.index_file` if needed:

32a ⟨*Read the configuration file and build a* `CalibrateConfiguration` *object* 32a⟩≡ (26b)

```
log.info('reading configuration file "%s"', configuration_file)
configuration = read_calibrate_conf_file(configuration_file)
log.info('configuration file read successfully')

if indexfile_path is not None:
    configuration.index_file = indexfile_path

if configuration.index_file is None:
    log.error('error: you must specify an index file, either in the '
              'parameter file or using the -index-file switch')
    sys.exit(1)
```

Uses read_calibrate_conf_file 29.

To load the index file, we use the `IndexFile` class we implemented for `index.py`, passing the first and last index of the TOD files to be loaded by the code:

32b ⟨*Load the index file* 32b⟩≡ (26b)

```
index = IndexFile()
index.load_from_fits(configuration.index_file,
                     first_index=configuration.first_tod_index,
                     last_index=configuration.last_tod_index)
log.info('%d files are going to be loaded', len(self.tod_info))
```

Uses IndexFile 21c.

§ 3.6. Sharing data among the MPI processes

In this section we implement the algorithms used by `calibrate.py` to split the TODs read from FITS files among the MPI processes. It is useful to have a fresh read of Ch. 2, as we are going to reuse many concepts from that discussion.

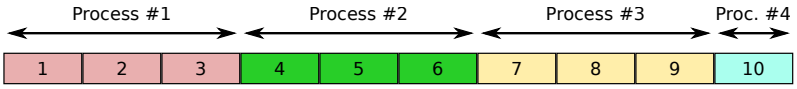
It is easy to explain the problem we are going to solve here by referring to Fig. 2.1 (page 12). Suppose we have running using two MPI processes, and that the TOD we are going to analyze has been split in two FITS files, `file01.fits` and `file02.fits`. We suppose that `index.py` (implemented in Ch. 2) has already ran, so we know that there are 10 offsets baselines, and that the sixth one is shared between the two FITS files `file01.fits`. What we need to do now is (1) to determine the length of each calibration baseline, and (2) to split the baselines between the two MPI processes.

3.6.1. Splitting lengths. The length of each calibration baseline is easy to compute, in principle, as we have asked the user to provide the length of such baselines in terms of the number of offset baselines (see the parameter `periods_per_cal_constant` in Table 3.1, page 29). The point is that not every way of splitting is equivalent in terms of performance. Consider for instance the case where we want

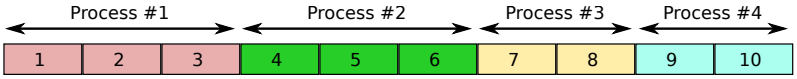
to fit $n = 10$ baselines into $m = 4$ MPI processes. We could do this if we assign to the first $m-1 = 3$ processes a number of baselines equal to

$$\left\lfloor \frac{n}{m-1} \right\rfloor = 3,$$

(we use $\lfloor \cdot \rfloor$ operation because we want to handle only an integer number of baselines), and if we leave the remaining baselines (1) to the last process. However, this is not optimal, as the fourth process has 1/3 of the data to process, and it is therefore likely to complete its tasks more quickly than the others:



The best solution is to evenly distribute the data among the processes, something like in the following figure:



Achieving this is only slightly more complicated than the previous formula:

33a ⟨*Functions to divide the TOD among the MPI processes* 33a⟩≡ (25) 33b▹

```
def split_into_n(length: int, num_of_segments: int):
    log.debug('entering split_into_n')
    assert (num_of_segments > 0), \
        "num_of_segments={0} is not positive".format(num_of_segments)
    assert (length >= num_of_segments), \
        "length={0} is smaller than num_of_segments={1}".format(length, num_of_segments)

    start_positions = np.array([int(i * length / num_of_segments)
                                for i in range(num_of_segments + 1)],
                               dtype='int')
    return start_positions[1:] - start_positions[0:-1]
```

Defines:
`split_into_n`, used in chunks 33b and 34.

If we test `split_into_n` with the following script:

```
from calibrate import split_into_n
print(split_into_n(10, 4))
```

then the output is:

```
[2 3 2 3]
```

which is slightly different from the figure above because it follows the pattern 2+3+2+3 instead of 3+3+2+2, but it is perfectly equivalent in terms of evenness of the distribution.

The function `split_into_n` is exactly what we need in order to divide the calibration baselines among the MPI processes, but we have some more little work to do here. Remember that a calibration baseline is made up by an integer number of offset baselines: the task is quite similar, but we cannot use `split_into_n` for this task too. The user does not specify the *overall number* of calibration baselines (like it is the case for the overall number of MPI processes, which is the parameter `num_of_segments` in `split_into_n`), but rather the number of offset baselines that must fit in each calibration baseline. Therefore, we need another splitting function to do this task. Fortunately, it is quite easy to wrap `split_into_n` into another function which does what we want:

33b ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◃33a 35b▹

```
def split(length, sublength: int):
    log.debug('entering split')
    assert (sublength > 0), "sublength={0} is not positive".format(sublength)
```

```
    assert (sublength < length), \
        "sublength={0} is not smaller than length={1}".format(sublength, length)

    return split_into_n(length=length,
                        num_of_segments=int(np.ceil(length / sublength)))
```

Defines:
  split, used in chunks 30a, 34, and 40d.
Uses split_into_n 33a.

The return value must be interpreted in the same way as for split_into_n. But note that sublength is just an approximation, as not every segment will end up having this number of baselines. For instance, if we want to split 10 offset baselines in a set of calibration baselines, each having roughly 4 of them, the script

```
from calibrate import split
print(split(10, 4))
```

produces

```
[3 3 4]
```

which is the most even distribution possible with these constraints: two calibration baselines of length 3 and the last one of length 4.

We are now able to implement the code in calibrate_main. Remember that at the beginning of that function we have defined mpi_rank to be the zero-based index of the current MPI process, and that configuration is an instance of the CalibrateConfiguration class which contains the key/value pairs loaded from the parameter file (see Sect. 3.5 and Table 3.1). Here is the code:

34 ⟨*Determine how input data should be split into baselines* 34⟩≡ (26b)

```
samples_per_ofsp = index.periods

gainp_lengths = split(length=len(samples_per_ofsp),
                      sublength=configuration.periods_per_cal_constant)
samples_per_gainp = ftnroutines.sum_subranges(samples_per_ofsp,
                                              gainp_lengths)

log.info('number of offset periods: %d; number of gain periods: %d',
         len(samples_per_ofsp), len(gainp_lengths))

gainp_per_process = split_into_n(length=len(samples_per_gainp),
                                 num_of_segments=mpi_size)
samples_per_process = ftnroutines.sum_subranges(samples_per_gainp,
                                                gainp_per_process)

gainp_idx_start = sum(gainp_per_process[0:mpi_rank])
gainp_idx_end = gainp_idx_start + gainp_per_process[mpi_rank]

ofsp_idx_start = sum(gainp_lengths[0:gainp_idx_start])
ofsp_idx_end = ofsp_idx_start + sum(gainp_lengths[gainp_idx_start:gainp_idx_end])

local_samples_per_ofsp = samples_per_ofsp[ofsp_idx_start:ofsp_idx_end]
local_samples_per_gainp = samples_per_gainp[gainp_idx_start:gainp_idx_end]
```

Uses split 33b, split_into_n 33a, and sum_subranges 35a.

We make use here of sum_subranges, the first of the many Fortran functions we will find in this chapter. The purpose of this function is to group the elements in array according to the parameter subrange_lengths, and then to add all the elements belonging to the same group. In Python, this function would be implemented in the following way:

```
def sum_subranges(array, subrange_lengths):
    log.debug('entering sum_subranges')
```

```
    array_idx = 0
    result = np.zeros(len(subrange_lengths), dtype='int')
    for subrange_idx, cur_length in enumerate(subrange_lengths):
        for i in range(cur_length):
            result[subrange_idx] += array[array_idx + i]
        array_idx += cur_length

    return result
```

However, cycling over large arrays is deadly slow. A Fortran implementation is very similar, but it works at far larger speeds:

35a ⟨*Fortran routines used by* `calibrate.py` 35a⟩≡ (26a) 48a▷

```
  subroutine sum_subranges(array, subrange_lengths, output)
    integer(kind=8), dimension(:), intent(in) :: array
    integer(kind=8), dimension(:), intent(in) :: subrange_lengths
    integer(kind=8), dimension(size(subrange_lengths)), intent(out) :: output

    integer(kind=8) :: array_idx, subrange_idx
    integer(kind=8) :: i

    output = 0
    array_idx = 0

    do subrange_idx = 1, size(subrange_lengths)
       do i = 1, subrange_lengths(subrange_idx)
          output(subrange_idx) = output(subrange_idx) + array(array_idx + i)
       enddo
       array_idx = array_idx + subrange_lengths(subrange_idx)
    enddo

  end subroutine sum_subranges
```

Defines:
`sum_subranges`, used in chunk 34.

The powerfulness of `f2py` is that it is able to parse the definition of routines like `sum_subranges` and produce a NumPy-friendly interface. The following script tests `sum_subranges`:

```
import ftnroutines as ftn
print(ftn.sum_subranges(array=[1, 2, 3, 4, 5], subrange_lengths=[3, 2]))
```

The call asks to return a vector of two elements, containing the sum of the first three $(1+2+3)$ and last two $(4+5)$ elements of array. The output is:

```
[6 9]
```

In the next section we will tackle the problem of loading the data from the set of FITS files specified in the index file.

3.6.2. Loading parts of TODs from FITS files. As we stated before, we do not expect the FITS files containing the TOD to have their boundaries coincident with our offset baselines. Each MPI process in our code must therefore be able to read samples from many FITS files and to assemble them together in one big data structure. This "big data structure" is a TOD, and we represent it using a class:

35b ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◁33b 36▷

```
  class TOD:
      def __init__(self, signal, pix_idx, num_of_pixels):
          self.signal = signal
          self.pix_idx = pix_idx
          self.num_of_pixels = num_of_pixels
```

Defines:
`TOD`, used in chunks 38b and 40.

We avoid using named tuples, as this is not going to be a read-only class (as `namedtuples` are): these objects will be built iteratively, each time loading a new chunk of data from a different FITS file.

We need also a structure which identifies the chunk of data in a FITS file that must be loaded by the current process:

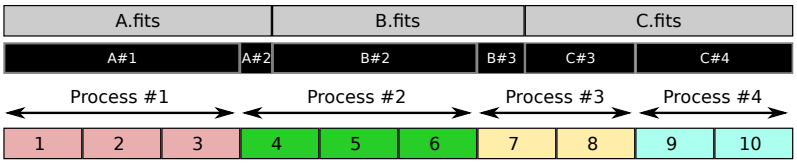
36 ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◂35b 37a▸

```
TODSubrange = namedtuple('TODSubrange',
                         ['file_info',
                          'first_idx',
                          'num_of_samples'])
```

Defines:
`TODSubrange`, used in chunks 37, 38, and 40a.

In this case, we use a simple `namedtuple` because we do not plan to modify a `TODSubrange` variable once it has been created.

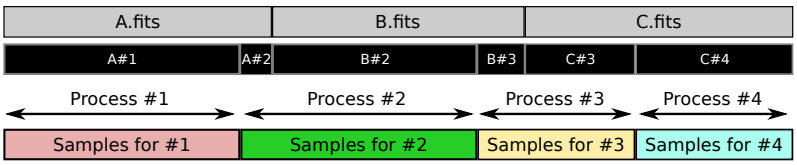
We have defined the two data structures that we need to implement the loading of data from the FITS files. What we want to do now is explained by the following example. We have a set of three files, named `A.fits`, `B.fits`, and `C.fits`, each containing a section of the complete TOD. We are running using 4 MPI processes, and we have judiciously split the 10 calibration baselines among them. What we need now is to build a function which determines which sections of which files each MPI process must load. The following sketch shows a possible solution:



The first gray row shows the boundaries of the FITS files containing the TOD, while in the bottom line we have the usual calibration baselines, distributed among the four MPI process. Our task is to determine the solution shown by the row of black rectangles, each of them symbolizing a `TODSubrange` object. Each rectangle is a section of a FITS file that is loaded by an MPI process:

1. Process #1 loads `A#1`, which is part of `A.fits`;
2. Process #2 loads `A#2` from `A.fits` and `B#2` from `B.fits`;
3. Process #3 loads `B#3` from `B.fits` and `C#3` from `C.fits`;
4. Process #4 loads `C#4` from `C.fits`.

Note that the boundaries of each `TODSubrange` object coincide either with the boundary of a FITS file or with the boundary of a gain baseline. Also, the algorithm does not need to know the boundary for *every* gain baseline, but only for those which are at the boundary between two MPI processes. In other words, the algorithm only needs to know the overall number of samples that each MPI process must load. See this sketch:



We implement this algorithm in the function `assign_files_to_processes`, which has the purpose of producing a nested list of `TODSubrange` objects. Each element in the list is a nested list of `TODSubrange` objects that must be read by a particular MPI process; so, in pseudocode we want to have the following behaviour for the case shown in the plot above:

```
>>> print(assign_files_to_processes(...))
[ [<A#1>], [<A#2>, <B#2>], [<B3>, <C3>], [<C4>] ]
```

The result is a list of four elements because there are four MPI processes; each element in the list is the list of TODSubrange objects to be read by the process itself.

The function accepts two arguments: the first one is the overall number of samples that each process must load, and it is simply the sum of the number of samples in each of the gain baseline "owned" by the process. The second argument is a list of TODFileInfo objects, which contains the information about each of the TOD FITS files that were listed in the index file (see Sect. 2.5, when we implemented TODFileInfo for index.py).

Our implementation of the algorithm works by running a for loop for each of the MPI processes. Within each loop, it iterates over the TOD files in tod_info until it "fills" the MPI process with the right number of samples:

37a ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◁36 38b▷

```
def assign_files_to_processes(samples_per_process: Any,
                              tod_info: List[TODFileInfo]) -> List[List[TODSubrange]]:
    log.debug('entering assign_files_to_processes')

    result = []  # type: List[List[TODSubrange]]
    file_idx = 0
    file_sample_idx = 0
    samples_in_file = tod_info[file_idx].num_of_unflagged_samples
    for samples_for_this_MPI_proc in samples_per_process:
        samples_left = samples_for_this_MPI_proc
        MPI_proc_subranges = []
        while (samples_left > 0) and (file_idx < len(tod_info)):
            if samples_in_file > samples_left:
                ⟨Process cases like A#1, B#2, C#3 37b⟩
            else:
                ⟨Process cases like A#2, B#3, C#4 38a⟩

            if samples_in_file == 0:
                if file_idx + 1 == len(tod_info):
                    break  # No more files, exit the while loop

                file_sample_idx = 0
                file_idx += 1
                samples_in_file = tod_info[file_idx].num_of_unflagged_samples

        result.append(MPI_proc_subranges)

    return result
```

Defines:
assign_files_to_processes, used in chunk 40b.
Uses TODFileInfo 20a and TODSubrange 36.

Cases like A#1, B#2, and C#3 happen when the current file has more samples than are needed to fill the current MPI process. In this case we reset samples_left in order to make the while loop terminate:

37b ⟨*Process cases like* A#1, B#2, C#3 37b⟩≡ (37a)

```
MPI_proc_subranges.append(TODSubrange(file_info=tod_info[file_idx],
                                      first_idx=file_sample_idx,
                                      num_of_samples=samples_left))
file_sample_idx += samples_left
samples_in_file -= samples_left
samples_left = 0
```

Uses TODSubrange 36.

The opposite case (as for A#2, B#3, and C#4) is when the FITS file is not large enough to provide the current MPI process with the required number of samples. In this case, we set samples_in_file

to zero, so that `file_idx` will be increased before the next `for` iteration:

38a ⟨*Process cases like* `A#2, B#3, C#4` 38a⟩≡ (37a)

```
MPI_proc_subranges.append(TODSubrange(file_info=tod_info[file_idx],
                                      first_idx=file_sample_idx,
                                      num_of_samples=samples_in_file))
samples_left -= samples_in_file
samples_in_file = 0
```

Uses `TODSubrange` 36.

How does our implementation perform against the example case we presented above? Let's test it with the following code:

```
from index import TODFileInfo
from calibrate import assign_files_to_processes
files = [TODFileInfo(name, 0, 12, 12) for name in ('A.fits',
                                                   'B.fits',
                                                   'C.fits')]
result = assign_files_to_processes([10, 10, 8, 8], files)
for mpi_idx, proc in enumerate(result):
    for subrange in proc:
        print('Process #{0}: {1}, {2:2d} |{3}|'
              .format(mpi_idx + 1,
                      subrange.file_info.file_name,
                      subrange.first_idx,
                      '-' * subrange.num_of_samples))
```

The code builds three fake `TODFileInfo` objects, each with the same number of samples in it (12). Then, it asks to assign the $12 + 12 + 12 = 36$ samples to 4 MPI processes, such that each of them has 10, 10, 8, and 8 samples, respectively. Finally, for each subrange it prints the start index in the FITS file and some ASCII art to represent the length of the data to load. If you have a look at the figure at page 36, you can see the match. The output of this script is the following:

```
Process #1: A.fits,  0 |----------|
Process #2: A.fits, 10 |--|
Process #2: B.fits,  0 |--------|
Process #3: B.fits,  8 |----|
Process #3: C.fits,  0 |----|
Process #4: C.fits,  4 |--------|
```

This shows that our implementation works as expected.

We need a function that loads one subrange and returns a `TOD` object. Our idea, as we already said, is to iteratively call this function and concatenate all the TODs it returns into one huge TOD. We check that the characteristics of the file match the information loaded from the index file. (However, in the current implementation we do not check if the modification time embedded in the index file matches the FITS file we are loading: the implementation of this feature is left to a next release.)

38b ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◃37a 40a▹

```
def load_subrange(subrange: TODSubrange,
                  index: IndexFile,
                  configuration: CalibrateConfiguration) -> TOD:
    log.debug('entering load_subrange')

    with fits.open(subrange.file_info.file_name) as f:
        signal = f[configuration.signal_hdu].data.field(configuration.signal_column)
        if len(signal) != subrange.file_info.num_of_samples:
            log.error('expected %d samples in file "%s", but %d found: '
                      'you should rebuild the index file',
                      subrange.file_info.num_of_samples,
```

```
                    subrange.file_info.file_name,
                    len(signal))
            sys.exit(1)

        ⟨Load the pixel index of each sample 39a⟩
        ⟨If necessary, remove flagged data 39b⟩

    if len(signal) != subrange.file_info.num_of_unflagged_samples:
        log.error('expected %d unflagged samples in file "%s", but %d found: '
                  'you should rebuild the index file',
                  subrange.file_info.num_of_unflagged_samples,
                  subrange.file_info.file_name,
                  len(signal))
        sys.exit(1)

    start, end = (subrange.first_idx,
                  (subrange.first_idx + subrange.num_of_samples))
    signal = signal[start:end]
    pix_idx = pix_idx[start:end]

    return TOD(signal=signal, pix_idx=pix_idx,
               num_of_pixels=healpy.nside2npix(configuration.nside))
```

Defines:
load_subrange, used in chunk 40a.
Uses CalibrateConfiguration 28, IndexFile 21c, TOD 35b, and TODSubrange 36.

Loading the pixel index of each sample requires some work. As you might remember, we allow the user to keep either the pixel index in the FITS files, or the colatitude/longitude of the pointing direction (see Table 3.1, page 29). When we loaded the parameter file (Sect. 3.5), we did not discriminate between the two possibilities, allowing `configuration.pointing_columns` to be either a 1-element or 2-element list. Therefore, in the implementation of `load_subrange` we need to take care of both cases:

39a ⟨*Load the pixel index of each sample* 39a⟩≡ (38b)

```
pix_idx = [f[configuration.pointing_hdu].data.field(x)
           for x in configuration.pointing_columns]
if len(pix_idx) == 2:
    theta, phi = pix_idx
    pix_idx = healpy.ang2pix(configuration.nside, theta, phi)
    del theta, phi
elif len(configuration.pointing_columns) == 1:
    pix_idx = pix_idx[0]
else:
    log.error('one or two columns are expected for the pointings (got %s)',
              ', '.join([str(x) for x in configuration.pointing_columns]))
    sys.exit(1)
```

Since unwanted data might be present in the TODs, we have to remove any of them. In this case we reuse function `flag_mask`, which we implemented for `index.py`, to remove any flagged sample[1] from `signal` and `pix_idx`.

39b ⟨*If necessary, remove flagged data* 39b⟩≡ (38b)

```
if index.flagging is not None:
    flags = f[index.flag_hdu].data.field(index.flag_column)
    mask = flag_mask(flags, index.flagging)

    signal = signal[mask]
```

[1] Note that there is a potential inefficiency here: `pix_idx` might have been computed from colatitude/longitude pairs using `healpy.ang2pix` (see above); in this case, part of the result of that computation is discarded here and might have therefore not been computed in the first place.

```
    pix_idx = pix_idx[mask]
```
Uses flag_mask 13d.

We have all the pieces needed to implement load_tod, which loads all the TODSubrange objects in a list and build a single, potentially huge TOD object. This function is going to be called just once by every MPI process:

40a ⟨*Functions to divide the TOD among the MPI processes* 33a⟩+≡ (25) ◁38b
```
def load_tod(tod_list: List[TODSubrange],
             index: IndexFile,
             configuration: CalibrateConfiguration) -> TOD:
    log.debug('entering load_tod')

    result = TOD(signal=np.array([], dtype='float'),
                 pix_idx=np.array([], dtype='int'),
                 num_of_pixels=healpy.nside2npix(configuration.nside))
    for cur_subrange in tod_list:
        cur_tod = load_subrange(cur_subrange, index, configuration)
        result.signal = np.concatenate((result.signal, cur_tod.signal))
        result.pix_idx = np.concatenate((result.pix_idx, cur_tod.pix_idx))

    return result
```
Defines:
load_tod, used in chunk 40b.
Uses CalibrateConfiguration 28, IndexFile 21c, load_subrange 38b, TOD 35b, and TODSubrange 36.

We call load_tod in the main function, calibrate_main:

40b ⟨*Load the part of the TOD that is assigned to this MPI process into the* tod *variable* 40b⟩≡ (26b) 40c▷
```
files_per_process = assign_files_to_processes(samples_per_process, index.tod_info)
tod = load_tod(files_per_process[mpi_rank], index, configuration)
```
Uses assign_files_to_processes 37a and load_tod 40a.

As a test of the correctness of the index, the data, and the program, we check that the number of samples in the TOD we have just loaded matches the sum of the samples in each offset period and calibration period that this MPI process will analyze:

40c ⟨*Load the part of the TOD that is assigned to this MPI process into the* tod *variable* 40b⟩+≡ (26b) ◁40b 40d▷
```
assert len(tod.signal) == sum(local_samples_per_ofsp)
assert len(tod.signal) == sum(local_samples_per_gainp)
```

Finally, we print some information about the number of samples loaded by all the MPI processes. In order to do this, we use the MPI function allreduce:

40d ⟨*Load the part of the TOD that is assigned to this MPI process into the* tod *variable* 40b⟩+≡ (26b) ◁40c
```
overall_num_of_samples = mpi_comm.allreduce(len(tod.signal), op=MPI.SUM)
log.info('elements in the TOD: %d (split among %d processes)',
         overall_num_of_samples, mpi_size)
```
Uses split 33b and TOD 35b.

All the code needed to load the input TOD files has been presented. Now we turn to the most mathematical part of the code, the one where we apply the Da Capo calibration code.

§ 3.7. Estimation of the dipole signal

To calibrate a stream of raw measurements, we need to have some function to estimate the amplitude of the dipole signal given some pointing direction towards the sky. The formula which relates the temperature of the dipole D to the speed of the spacecraft rest frame at time t along direction $\boldsymbol{x}$ is

$$D(\boldsymbol{x},t) = T_{\text{CMB}}\left(\frac{1}{\gamma(t)\big(1-\boldsymbol{\beta}(t)\cdot\boldsymbol{x}\big)}-1\right), \tag{3.1}$$

where $\boldsymbol{\beta}(t) = \boldsymbol{v}(t)/c$ is the speed of the rest frame, and $\gamma(t) = \left(1-\beta^2(t)\right)^{-1/2}$. However, this formula does not consider frequency-dependent relativistic corrections due to the fact that detectors usually measure brightness around some frequency ν instead of the temperature D itself (Quartin and Notari, 2015). A more precise formula for the dipole signal would thus be the following:

$$D(\boldsymbol{x}, t, \nu) = T_{\mathrm{CMB}}\left(\beta \cdot \boldsymbol{x} + Q(\nu) \cdot \left(\beta \cdot \boldsymbol{x}\right)^2\right), \tag{3.2}$$

where $Q(\nu)$ is defined as

$$Q(\nu) = \frac{h\nu}{2k_B T_{\mathrm{CMB}}} \coth\left(\frac{h\nu}{2k_B T_{\mathrm{CMB}}}\right) = \frac{h\nu}{2k_B T_{\mathrm{CMB}}} \frac{\exp(h\nu/k_B T_{\mathrm{CMB}})+1}{\exp(h\nu/k_B T_{\mathrm{CMB}})-1}. \tag{3.3}$$

In our case, we ignore the velocity of the spacecraft and take into account the velocity of the Solar System with respect to the CMB rest frame only. Consider that in the case of a spacecraft orbiting the L_2 point, this speed is of the order of 30 km/s, thus accounting for roughly 10 % of the speed with respect to the CMB rest frame (370 km/s).

We implement the function `get_dipole_temperature`, which takes the value of T_{CMB} (`t_cmb_k`), the velocity of the Solar System $\boldsymbol{v}(t)$ (`solsys_speed_vec_m_s`, a 3-element NumPy array), and an array of directions $\{\boldsymbol{x}_i\}$ (`directions`), and it returns an array of temperature (one for each direction $\boldsymbol{x}_i$). Since version 1.1, the caller can optionally specify a frequency: in this case, Eq. (3.2) will be used instead of Eq. (3.1).

41a ⟨*Estimation of the dipole signal* 41a⟩≡ (25)

```
SPEED_OF_LIGHT_M_S = 2.99792458e8
PLANCK_H_MKS = 6.62606896e-34
BOLTZMANN_K_MKS = 1.3806504e-23

def get_dipole_temperature(t_cmb_k: float, solsys_speed_vec_m_s, directions, freq=None):
    "'Given one or more one-length versors, return the intensity of the CMB dipole

    The vectors must be expressed in the Ecliptic coordinate system.
    If "freq" (frequency in Hz) is specified, the formulation will use the
    quadrupolar correction.
    "'
    log.debug('entering get_dipole_temperature')

    beta = solsys_speed_vec_m_s / SPEED_OF_LIGHT_M_S
    if freq:
            fact = PLANCK_H_MKS * freq / (BOLTZMANN_K_MKS * t_cmb_k)
            expfact = np.exp(fact)
            q = (fact / 2) * (expfact + 1) / (expfact - 1)
            dotprod = np.dot(beta, directions)
            return t_cmb_k * (dotprod + q * dotprod**2)
    else:
            gamma = (1 - np.dot(beta, beta))**(-0.5)

            return t_cmb_k * (1.0 / (gamma * (1 - np.dot(beta, directions))) - 1.0)
```

Defines:
`BOLTZMANN_K_MKS`, never used.
`get_dipole_temperature`, used in chunks 42a, 80c, 81, and 83b.
`PLANCK_H_MKS`, never used.
`SPEED_OF_LIGHT_M_S`, never used.

We can now implement the part of the main function `calibrate_main` where we build the monopole and dipole maps $\boldsymbol{m}_c$ (Eq. 1.19). Since the monopole map needs to take any mask into account, we load the mask the user might have specified in the parameter file (through the `mask` parameter in the `dacapo` section, see Table 3.1, page 29):

41b ⟨*Build the monopole and dipole maps* 41b⟩≡ (26b) 42a▸

```
if configuration.mask_file_path is not None:
```

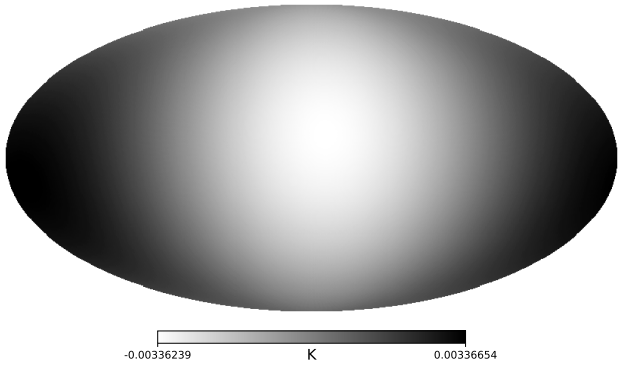


Figure 3.1: *Amplitude of the Doppler signal caused by the motion of the Solar System with respect to the CMB rest frame. See Eq.* (3.1).

```
        mask = healpy.read_map(configuration.mask_file_path, verbose=False)
        mask = np.array(healpy.ud_grade(mask, configuration.nside), dtype='int')
    else:
        mask = None
```

Converting the result of `healpy.ud_grade` truncates any floating-point number to its lowest integer; thus, pixels in the mask with any value between 0 and 1 will be converted to zero. (This is likely to be the case if the mask has an higher value for `NSIDE` than the value in `configuration.nside`, which has been specified by the user in the configuration file.)

After having loaded the mask, we can generate the dipole map:

42a ⟨*Build the monopole and dipole maps* 41b⟩+≡ (26b) ◁41b 42b▷

```
directions = healpy.pix2vec(configuration.nside,
                            np.arange(healpy.nside2npix(configuration.nside)))
dipole_map = get_dipole_temperature(t_cmb_k=configuration.t_cmb_k,
                                    solsys_speed_vec_m_s=configuration.solsys_speed_vec_m_s,
                                    directions=directions,
                                    freq=configuration.frequency_hz)
```

Uses `get_dipole_temperature` 41a.

Finally, we create a `MonopoleAndDipole` class, which will be needed to run the `Da Capo`algorithm:

42b ⟨*Build the monopole and dipole maps* 41b⟩+≡ (26b) ◁42a

```
mc = MonopoleAndDipole(mask=mask, dipole_map=dipole_map)
```

Uses `MonopoleAndDipole` 54c.

The implementation of `MonopoleAndDipole` map will be discussed in Sect. 3.10.7.

As a reference, the following code produces the map shown in Fig. 3.1:

```
import matplotlib
matplotlib.use('Agg')  # Non-GUI backend
import healpy
import matplotlib.pylab as plt
import numpy as np
from calibrate import get_dipole_temperature
nside = 64
solsys_speed_vec_m_s = np.array([-359234.3, 52715.1, -71643.0])
directions = healpy.pix2vec(nside, np.arange(healpy.nside2npix(nside)))
dipole_map = get_dipole_temperature(t_cmb_k=2.72548,
                                    solsys_speed_vec_m_s=solsys_speed_vec_m_s,
                                    directions=directions)
healpy.mollview(dipole_map, title='', unit='K', cmap='gray_r')
plt.savefig('test_dipole_temperature.pdf', bbox_inches='tight')
```

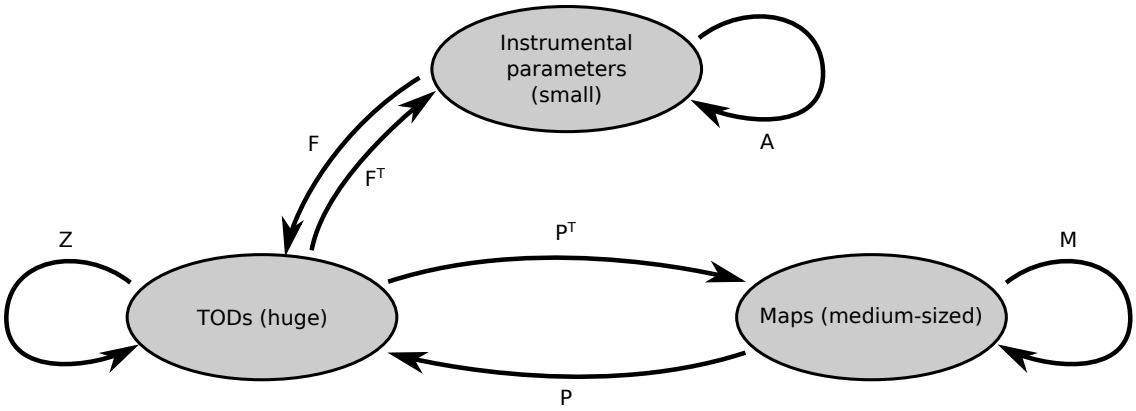


Figure 3.2: *Domains and codomains of the matrix operators used in Da Capo. Matrix $\tilde{\boldsymbol{P}}$ is not shown, as it is structurally identical to matrix $\boldsymbol{P}$. For each domain, a qualitative estimate of its size is provided; for a 1 year mission, a detector with $1/f$ knee frequency 20 mHz which samples data at 50 Hz and needs to be recalibrated each 6 hours requires the following amount of memory: 5 MB for the instrumental parameters, 100 MB for a Healpix map with* NSIDE = 1024*, and 50 GB for the TODs.*

The velocity of the Solar System (`solsys_speed_vec_m_s`) is the one measured by Planck Collaboration (2015), and it is expressed in the Ecliptic coordiante system.

§ 3.8. Implementation of the Da Capo algorithm

In this chapter we are going to implement the equations presented in Chapter 3, which are the core of the Da Capo algorithm. Let's first recap them shortly.

Our model for the output of a detector is

$$V_i = G_k\big(T_i + D_i\big) + b_k + N_i, \qquad \text{(1.1 revisited)}$$

where V_i is the output of the detector, G_k is the gain factor, T_i is the temperature of the sky (Galaxy and CMB), D_i is the dipole, b_k is a constant offset, and N_i is a zero-mean, Gaussian random variable.

In order to estimate the $1/f$ baselines b_k (offsets), the gains G_k, and the sky map $\boldsymbol{m}$, we start from a guess for G and $\boldsymbol{m}$, and then we solve the following equation:

$$\boldsymbol{A}\boldsymbol{a} = \boldsymbol{v}, \qquad \text{(1.14 revisited)}$$

where $\boldsymbol{a}$ is the vector which concatenates the offsets b_k and the gains G_k, and

$$\boldsymbol{A} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{F}, \qquad \text{(1.15 revisited)}$$

$$\boldsymbol{v} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{V}, \qquad \text{(1.16 revisited)}$$

$$\boldsymbol{Z} = \boldsymbol{I} - \tilde{\boldsymbol{P}}\big(\boldsymbol{M} + \boldsymbol{C}_m^{-1}\big)^{-1} \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1}, \qquad \text{(1.20 revisited)}$$

$$\boldsymbol{M} = \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}}, \qquad \text{(1.21 revisited)}$$

$$\big(\boldsymbol{M} + \boldsymbol{C}_m^{-1}\big)^{-1} = \Big(\boldsymbol{I} - \boldsymbol{M}^{-1}\boldsymbol{m}_c\big(\boldsymbol{m}_c^T \boldsymbol{M}^{-1} \boldsymbol{m}_c\big)^{-1} \boldsymbol{m}_c^T\Big)\boldsymbol{M}^{-1}, \qquad \text{(1.22 revisited)}$$

and $\boldsymbol{m}_c$ is a $N \times 2$ matrix which contains the map of the monopole (first column, with all elements equal to one) and of the dipole (second column). Once we have an estimate for $\boldsymbol{a}$, we get a better estimate of the sky map $\boldsymbol{m}$ if we add the map

$$\tilde{\boldsymbol{m}}_1 = \big(\tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}}\big)^{-1} \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1}\big(\boldsymbol{V} - \boldsymbol{F}\boldsymbol{a}\big) \qquad \text{(1.18 revisited)}$$

to $\boldsymbol{m}$. Then, we can solve again Eq. (1.14) until the solution is good enough.

Since the equations seem daunting, we provide here a few clues about how to implement these calculations. Refer also to Fig. 3.2.

First of all, if we explicit $\boldsymbol{A}$ and $\boldsymbol{v}$ in Eq. (1.14), we get

$$\boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z}(\boldsymbol{F}\boldsymbol{a}) = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z}(\boldsymbol{V}). \tag{3.4}$$

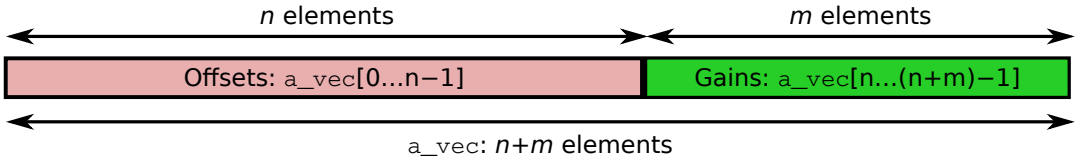
This representation of Eq. (1.14) makes a few facts self-evident:

1. Once vector $\boldsymbol{a}$ is multiplied by $\boldsymbol{F}$, the result has the shape of a TOD of voltages, exactly like $\boldsymbol{V}$ (the TOD passed as input to the algorithm). This TOD is our best approximation of the input TOD, based on the values of the offsets b, the gains G, the dipole $\boldsymbol{D}$, and the map $\boldsymbol{m}$.

2. The two TODs are passed to the same sequence of operators, namely, $\boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z}$. From the equation above, it follows that Da Capo requires that the results of the two operations be equal, if $\boldsymbol{a}$ contains the "right" offsets b_k and gains G_k.

We now turn to describe the meaning of the operator $\boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z}$...

We need a datatype to represent vectors in the group "Instrumental parameters" (Fig. 3.2), like $\boldsymbol{a}$. Such vectors must contain both the offsets b_k and the gains G_k, are represented using a new data structure, `OfsAndGains`. We keep them together in one NumPy vector, `a_vec`, using the following memory layout:



It is much handier to have them both in the same array than to keep two separate arrays, as we shall see soon.

The definition of `OfsAndGains` is the following. Note that we provide two `@property` functions to access the offsets and the gains as if they were separate arrays:

44 ⟨*Datatypes used by* `calibrate.py` 28⟩+≡ (25) ◁28 45b▷

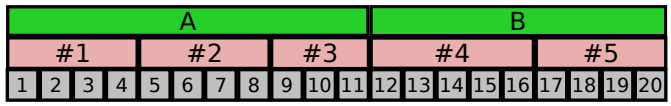
```
class OfsAndGains:
    def __init__(self, offsets, gains, samples_per_ofsp, samples_per_gainp):
        self.a_vec = np.concatenate((offsets, gains))
        self.samples_per_ofsp = np.array(samples_per_ofsp, dtype='int')
        self.samples_per_gainp = np.array(samples_per_gainp, dtype='int')

        self.ofsp_per_gainp = OfsAndGains.calc_ofsp_per_gainp(samples_per_ofsp,
                                                              samples_per_gainp)

    def __copy__(self):
        return OfsAndGains(offsets=np.copy(self.offsets),
                           gains=np.copy(self.gains),
                           samples_per_ofsp=np.copy(self.samples_per_ofsp),
                           samples_per_gainp=np.copy(self.samples_per_gainp))

    def __repr__(self):
        return 'a: {0} (offsets: {1}, gains: {2})'.format(self.a_vec,
                                                          self.offsets,
                                                          self.gains)

    @property
    def offsets(self):
        return self.a_vec[0:len(self.samples_per_ofsp)]

    @property
```

```
    def gains(self):
        return self.a_vec[len(self.samples_per_ofsp):]
```

⟨*Implementation of* `OfsAndGains.calc_ofsp_per_gainp` 45a⟩

Defines:
`OfsAndGains`, used in chunks 45b, 49a, 50, 52–57, 62a, 63b, 66b, 68a, 73, 75b, 78e, and 79b.
Uses `OfsAndGains.calc_ofsp_per_gainp` 45a.

We use `OfsAndGains.calc_ofsp_per_gainp` in the constructor. This is a static method to compute the number of offset periods within each gain period, and it is a complementary information to `samples_per_ofsp` (number of raw samples in each offset period) and `samples_per_gainp` (number of raw samples in each gain period). To explain why this is useful, consider the following example, where a TOD with $N = 20$ samples has already been divided into 5 offset periods and 2 gain periods:



The variable `samples_per_ofsp` is a list containing the numbers `[4, 4, 3, 5, 4]` (samples in each offset period), and the list in `samples_per_gainp` contains `[11, 9]` (samples in each gain period). The purpose of calling `OfsAndGains.calc_ofsp_per_gainp` is to produce a third list, `ofsp_per_gainp`, which contains the number of offset periods per gain period, in this case the list `[3, 2]`. This information will be used by the preconditioner used to speed up the Conjugate Gradient algorithm.

The implementation of `OfsAndGains.calc_ofsp_per_gainp` is trivial:

45a ⟨*Implementation of* `OfsAndGains.calc_ofsp_per_gainp` 45a⟩≡ (44)

```
  @staticmethod
  def calc_ofsp_per_gainp(samples_per_ofsp, samples_per_gainp):
      log.debug('entering calc_ofsp_per_gainp')

      ofsp_per_gainp = []

      cur_ofsp_idx = 0
      for samples_in_cur_gainp in samples_per_gainp:
          ofsp_in_cur_gainp = 0
          sample_count = 0
          while sample_count < samples_in_cur_gainp:
              sample_count += samples_per_ofsp[cur_ofsp_idx]
              cur_ofsp_idx += 1
              ofsp_in_cur_gainp += 1

          assert sample_count == samples_in_cur_gainp
          ofsp_per_gainp.append(ofsp_in_cur_gainp)

      return ofsp_per_gainp
```

Defines:
`OfsAndGains.calc_ofsp_per_gainp`, used in chunks 44 and 62a.

We have implemented the `__copy__` method because we are going to make copies of `OfsAndGains` objects in the implementation of the conjugate gradient algorithm. The following helper function creates a new `OfsAndGains` object with the same number of offset/gain periods as `source`:

45b ⟨*Datatypes used by* `calibrate.py` 28⟩+≡ (25) ◁44 54c▷

```
  def ofs_and_gains_with_same_lengths(source: OfsAndGains, a_vec):
      result = copy(source)
      result.a_vec = np.copy(a_vec)
      return result
```

Defines:
`ofs_and_gains_with_same_lengths`, used in chunks 57c, 62a, and 63b.
Uses `OfsAndGains` 44.

The typical usage of this function is to create a new vector $\boldsymbol{a}$ from an existing one without the annoyance of recomputing the length of every gain/offset period.

§ 3.9. Computational tricks

The application of $\boldsymbol{A}$ to the vector of unknowns $\boldsymbol{a}$ is quite trivial and does not really need to build any matrix in memory at all:

1. We can assume that $\boldsymbol{C}_n = \sigma^2 \boldsymbol{I}$ (i.e., the white noise of the detector remains constant in time), so that all the $\boldsymbol{C}_n$ terms in the equation above are simplified out;
2. Matrix $\tilde{\boldsymbol{P}}$ is slightly more complicated than $\boldsymbol{P}$. First, elements in the former are expressed in V/K, while elements in the latter are pure numbers. Therefore, $\boldsymbol{Pm}$ scans a map into a TOD expressed in the same measure unit as $\boldsymbol{m}$ (temperature), while $\tilde{\boldsymbol{P}}\boldsymbol{m}$ produces a TOD of voltages.
3. Applying matrix $\boldsymbol{P}^T$ to a TOD produces a binned map of the TOD itself. Applying $\tilde{\boldsymbol{P}}^T$ to a TOD of temperatures produces a map of binned voltages.
4. Matrix $\boldsymbol{M} \equiv \tilde{\boldsymbol{P}}^T \tilde{\boldsymbol{P}}$ has a particularly simple structure. As an example, we consider the example in Eq. (1.3) and two calibration periods with gains G_1 and G_2 which cover 5 and 3 samples respectively. Then
$$\tilde{\boldsymbol{P}} = \begin{pmatrix} G_1 & 0 & 0 \\ G_1 & 0 & 0 \\ 0 & G_1 & 0 \\ G_1 & 0 & 0 \\ 0 & G_1 & 0 \\ 0 & 0 & G_2 \\ 0 & 0 & G_2 \\ G_2 & 0 & 0 \\ 0 & 0 & G_2 \end{pmatrix}, \tag{3.5}$$
and
$$\boldsymbol{M} \equiv \tilde{\boldsymbol{P}}^T \tilde{\boldsymbol{P}} = \begin{pmatrix} 3G_1^2 + G_2^2 & 0 & 0 \\ 0 & 2G_1^2 & 0 \\ 0 & 0 & 3G_2^2 \end{pmatrix}. \tag{3.6}$$
Therefore, the diagonal elements of matrix $\boldsymbol{M}$ represent a map where each pixel is the sum of the squared gains of each sample falling within the pixel itself.
5. $\boldsymbol{Z}$ is a complex matrix, which has the purpose of "cleaning" a TOD from the part of the signal which does not depend on noise.
6. $\boldsymbol{F}$ converts the parameters $\boldsymbol{b}$ and $\boldsymbol{G}$ into a TOD containing an estimate of the *uncalibrated* signal (in Volt);
7. $\boldsymbol{F}^T$ takes a TOD as input and produces a shorter vector divided in two halves: the first half contains the sum of the signal within each calibration period, the second half contains the sum of the *uncalibrated* sky signal $\boldsymbol{D}+m$ (remember that the G constant decalibrates a signal, converting it from Kelvin to Volt).

§ 3.10. Implementation of the conjugate gradient algorithm

In this long section we will slowly implement all the matrix/vector functions used by the conjugate gradient algorithm and by `Da Capo`. We begin by introducing the algorithm of the conjugate gradient and its variant with a preconditioner (sect. 3.10.1). Then, in sections 3.10.2 and following, we will implement the algorithm in Python.

3.10.1. Solving the problem using the conjugate gradient method. Equation (1.14) on page 4 can be solved for $\boldsymbol{a}$ without inverting matrix $\boldsymbol{A}$ if we use the conjugate gradient method (Strikwerda, 2004). We do not spend too much time describing the method, as there are plenty of explanations[2] available. The interest of this method lies in the fact that it does not calculate matrix $\boldsymbol{A}^{-1}$ explicitly, because it only computes products of $\boldsymbol{A}$ with other vectors. (In our case, matrix $\boldsymbol{A}$ would be too big to be stored and inverted in memory.)

It is possible to devise an iterative algorithm which converges to the solution of Eq. (1.14); the outline is the following (the symbol $\leftarrow$ denotes variable assignment):

$\boldsymbol{r}_0 \leftarrow \boldsymbol{v} - \boldsymbol{A}\boldsymbol{a}_0$
$\boldsymbol{p}_0 \leftarrow \boldsymbol{r}_0$
$k \leftarrow 0$
loop
  $\gamma_k \leftarrow \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k}$
  $\boldsymbol{a}_{k+1} \leftarrow \boldsymbol{a}_k + \gamma_k \boldsymbol{p}_k$
  $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k - \gamma_k \boldsymbol{A} \boldsymbol{p}_k$
  if $\boldsymbol{r}_{k+1}$ is small enough **then**
    exit the loop
  end if
  $\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{r}_{k+1} + \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k} \boldsymbol{p}_k$
  $k \leftarrow k + 1$
end loop

The condition which makes the loop end is that the residual $\boldsymbol{r}_k$ is small enough (had the algorithm converged to the exact solution, each element of $\boldsymbol{r}_k$ would be identically zero).

It is often the case that the algorithm above converges slowly towards the solution. This usually happens because matrix $\boldsymbol{A}$ has a bad conditioning number, i.e., the ratio between the largest and the smallest eigenvalues of $\boldsymbol{A}$ is large: this implies that numerical roundoff errors significantly affects the evaluation of $\boldsymbol{A}$. It is possible to improve the convergence rate if one is able to derive a positive symmetric definite matrix $\boldsymbol{M}$, called the *preconditioner*, which has the property that

$$\boldsymbol{M} \approx \boldsymbol{A}^{-1}. \tag{3.7}$$

In this case, the algorithm becomes the following:

$\boldsymbol{r}_0 \leftarrow \boldsymbol{v} - \boldsymbol{A}\boldsymbol{a}_0$
$\boldsymbol{z}_0 \leftarrow \boldsymbol{M}^{-1} \boldsymbol{r}_0$
$\boldsymbol{p}_0 \leftarrow \boldsymbol{r}_0$
$k \leftarrow 0$
loop
  $\gamma_k \leftarrow \frac{\boldsymbol{z}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k}$
  $\boldsymbol{a}_{k+1} \leftarrow \boldsymbol{a}_k + \gamma_k \boldsymbol{p}_k$
  $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k - \gamma_k \boldsymbol{A} \boldsymbol{p}_k$
  if $\boldsymbol{r}_{k+1}$ is small enough **then**
    exit the loop
  end if
  $\boldsymbol{z}_{k+1} \leftarrow \boldsymbol{M}^{-1} \boldsymbol{r}_{k+1}$
  $\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{z}_{k+1} + \frac{\boldsymbol{z}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{z}_k^T \boldsymbol{r}_k} \boldsymbol{p}_k$
  $k \leftarrow k + 1$
end loop

It is not unusual to see wall clock times reduced by one order of magnitude with the introduction of a preconditioner. We will implement two kinds of preconditioners in Sect. 3.11.

[2] A quite good introduction is available on Wikipedia: https://en.wikipedia.org/wiki/Conjugate_gradient_method.

3.10.2. From instrumental parameters to TODs. The first function we implement is `apply_f`, which calculates the product between matrix $\boldsymbol{F}$ (Eq 1.11). This operation transforms a set of instrumental parameters (gains and offsets) into a TOD (see Fig. 3.2).

Since this function must be iterated over the number of samples in the TOD, and since this is a potentially large number, we implement it in Fortran. Here is the code:

48a ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁35a 49b▷

```
subroutine apply_f(offsets, gains, samples_per_ofsp, samples_per_gainp, &
     pix_idx, dipole_map, sky_map, output)

  real(kind=8), dimension(:), intent(in) :: offsets
  real(kind=8), dimension(:), intent(in) :: gains
  integer(kind=8), dimension(size(offsets)), intent(in) :: samples_per_ofsp
  integer(kind=8), dimension(size(gains)), intent(in) :: samples_per_gainp
  integer(kind=8), dimension(:), intent(in) :: pix_idx
  real(kind=8), dimension(:), intent(in) :: dipole_map
  real(kind=8), dimension(size(dipole_map)), intent(in) :: sky_map
  real(kind=8), dimension(size(pix_idx)), intent(out) :: output

  integer(kind=8) :: cur_ofsp_idx, cur_gainp_idx
  integer(kind=8) :: samples_in_ofsp, samples_in_gainp
  integer(kind=8) :: i

  cur_ofsp_idx = 1
  cur_gainp_idx = 1
  samples_in_ofsp = 0
  samples_in_gainp = 0

  do i = 1, size(output)

     output(i) = offsets(cur_ofsp_idx) +&
          (dipole_map(pix_idx(i) + 1) + sky_map(pix_idx(i) + 1)) * gains(cur_gainp_idx)

     ⟨Increment samples_in_ofsp 48b⟩
     ⟨Increment samples_in_gainp 48c⟩
  enddo

end subroutine apply_f
```

Defines:
`ftnroutines.apply_f`, used in chunk 49a.
Uses `apply_f` 49a.

Once we have calculated the value of the i-th sample, we need to advance to the next sample. While doing this, we check if we must advance to the next offset/gain value in the arrays `offsets` and `gains`. This is the code for updating the offset indexes:

48b ⟨*Increment* `samples_in_ofsp` 48b⟩≡ (48a 49b)

```
samples_in_ofsp = samples_in_ofsp + 1

if (samples_in_ofsp .ge. samples_per_ofsp(cur_ofsp_idx)) then
   cur_ofsp_idx = cur_ofsp_idx + 1
   samples_in_ofsp = 0
endif
```

We update the index to the pixel and the index to the period. The same must be done for the gain as well:

48c ⟨*Increment* `samples_in_gainp` 48c⟩≡ (48a 49b 53)

```
samples_in_gainp = samples_in_gainp + 1

if (samples_in_gainp .ge. samples_per_gainp(cur_gainp_idx)) then
   cur_gainp_idx = cur_gainp_idx + 1
```

```
    samples_in_gainp = 0
  endif
```

We nicely wrap this function in a Python function, which has the only purpose of extracting all the parameters from the `OfsAndGains` object:

49a ⟨*Matrix/vector multiplication functions* 49a⟩≡ (25) 50▷

```
def apply_f(a: OfsAndGains, pix_idx, dipole_map, sky_map):
    log.debug('entering apply_f')
    return ftnroutines.apply_f(a.offsets, a.gains,
                               a.samples_per_ofsp, a.samples_per_gainp,
                               pix_idx, dipole_map, sky_map)
```

Defines:
`apply_f`, used in chunks 48a, 56a, 68a, 73, and 77c.
Uses `ftnroutines.apply_f` 48a and `OfsAndGains` 44.

Note that we do not use MPI explicitly in the application of $\boldsymbol{F}$. However, remember that the overall TOD (which might well be one full year of data) is split among the MPI processes. The way `aply_F` is called is such that the result is a TOD which covers the same time span as the chunk of data currently loaded by the current MPI process: `a` only includes the offsets and gains owned by the process, and `pixidx` lists only the pixels "seen" by the samples loaded by the process.

Since this is a quite important concept to grasp, let's see it from another point of view. The case here is to compute a matrix-vector multiplication of the form $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$. We write this operation in full form:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \dots & a_{1m} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \tag{3.8}$$

Suppose now that vector $\boldsymbol{x}$ is split in many sub-vectors, and that each of them belongs to a different MPI process. For instance, process 1 has $(x_1\ x_2)$, process 2 has $(x_3\ x_4\ x_5)$, and so on. Suppose also that multiplication by $\boldsymbol{A}$ can be implemented algorithmically, i.e., without the need of keeping the whole matrix $\boldsymbol{A}$ in memory. If process 1 computes the quantity

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1m} \\ a_{21} & a_{22} & a_{23} & \dots & a_{1m} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \tag{3.9}$$

and all the other MPI processes do the same with their sub-vectors of $\boldsymbol{x}$. Then, the coefficients of $\boldsymbol{y}$ are equal to the result that would have been obtained using Eq. (3.8) directly. Therefore, in order to compute all the coefficients for vector $\boldsymbol{y}$, it is enough for each MPI process to compute the multiplication between a sub-matrix of $\boldsymbol{A}$ and a sub-vector of $\boldsymbol{x}$, and then to concatenate together all the results from each MPI process. This explains why we have implemented `apply_F` without caring for MPI parallelization.

3.10.3. From TODs to instrumental parameters. Now that we have implemented the multiplication by $\boldsymbol{F}$, let's turn ourselves to $\boldsymbol{F}^T$. As it was the case before, the amount of data that are typically involved require us to implement the calculation in Fortran. The input data (a TOD) is passed through the argument `vector`; the other arguments should be self-explanatory:

49b ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁48a 52a▷

```
subroutine apply_ft(vector, offsets, gains, samples_per_ofsp, samples_per_gainp, &
     pix_idx, dipole_map, sky_map, output)
  real(kind=8), dimension(:), intent(in) :: vector
  real(kind=8), dimension(:), intent(in) :: offsets
  real(kind=8), dimension(:), intent(in) :: gains
  integer(kind=8), dimension(size(offsets)), intent(in) :: samples_per_ofsp
  integer(kind=8), dimension(size(gains)), intent(in) :: samples_per_gainp
  integer(kind=8), dimension(size(vector)), intent(in) :: pix_idx
  real(kind=8), dimension(:), intent(in) :: dipole_map
```

```
    real(kind=8), dimension(size(dipole_map)), intent(in) :: sky_map
    real(kind=8), dimension(size(offsets) + size(gains)), intent(out) :: output

    integer(kind=8) :: cur_ofsp_idx, cur_gainp_idx
    integer(kind=8) :: samples_in_ofsp, samples_in_gainp
    integer(kind=8) :: i

    cur_ofsp_idx = 1
    cur_gainp_idx = 1
    samples_in_ofsp = 0
    samples_in_gainp = 0

    output = 0

    do i = 1, size(vector)
       output(cur_ofsp_idx) = output(cur_ofsp_idx) + vector(i)
       output(size(offsets) + cur_gainp_idx) = output(size(offsets) + cur_gainp_idx) + &
            vector(i) * (dipole_map(1 + pix_idx(i)) + sky_map(1 + pix_idx(i)))

       ⟨Increment samples_in_ofsp 48b⟩
       ⟨Increment samples_in_gainp 48c⟩
    enddo

  end subroutine apply_ft
```

Defines:
ftnroutines.apply_ft, used in chunk 50.
Uses apply_ft 50.

The output is a plain array: the first `size(offsets)` elements contain the running sum of the corresponding elements of `vector` within each offset period, while the end tail of `output` contains the running sum of the monopole and dipole observed while measuring the i-th sample, multiplied by that sample.

Note that we reuse the same code used in the implementation of `ftnroutines.apply_f` to increment `samples_in_ofsp` and `samples_in_gainp`, as the logic and the variable names used here are the same.

The Python wrapper for `ftnroutines.apply_ft` is trivial to implement:

50 ⟨Matrix/vector multiplication functions 49a⟩+≡ (25) ◁49a 51▷

```
  def apply_ft(vector, a: OfsAndGains, pix_idx, dipole_map, sky_map):
      log.debug('entering apply_ft')
      return ftnroutines.apply_ft(vector, a.offsets, a.gains,
                                  a.samples_per_ofsp, a.samples_per_gainp,
                                  pix_idx, dipole_map, sky_map)
```

Defines:
apply_ft, used in chunks 49b, 56, 73, and 77d.
Uses ftnroutines.apply_ft 49b and OfsAndGains 44.

Note that the result is not a `OfsAndGains` object, but rather a plain NumPy array: this might be assigned to the `a_vec` component of a `OfsAndGains` object, as the Fortran routine builds it using the same memory layout we presented above (first offsets, then gains). Also, the calculation is parallelized using the same principle used to implement `apply_F`: each MPI process produces a `OfsAndGains` object which only contains those offset/gain periods owned by the process itself, as the input TOD is a local subset of the full TOD.

3.10.4. The gain hit map. Let's now concentrate in the multiplication by matrix $\boldsymbol{M} = \tilde{\boldsymbol{P}}^T \boldsymbol{C}_n^{-1} \tilde{\boldsymbol{P}}$ (Eq. 1.21), with the $\boldsymbol{C}_n^{-1}$ term dropped according to the hypotheses presented in Sect. 3.9. Matrix $\boldsymbol{M}$ is a diagonal matrix, where the p diagonal coefficients are the pixel values of a map which is similar to hit maps. (To get a proper hit map, one would consider the diagonal elements of the matrix $\boldsymbol{P}^T \boldsymbol{P}$.)

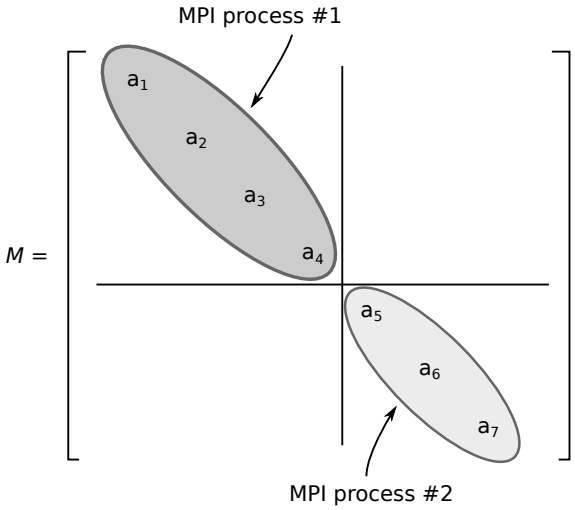


Figure 3.3: *How the diagonal coefficients of matrix $\boldsymbol{M}$ are computed, in the case of a 7×7 matrix and two MPI processes running concurrently. All the non-diagonal terms are zero and are not showed here.*

We need to be careful in implementing this calculation, as it is not as easy to parallelize as it was the case for `apply_F` and `apply_FT`. The point is that $\boldsymbol{M}$ is a diagonal matrix whose coefficients depend on *all* the samples in the full TOD, not only on the samples of the local TOD. Each diagonal coefficient m_{pp} contains the sum of the squared values of the gains associated with each sample of the full TOD which falls within pixel p. Therefore, to correctly consider the contribution from all the MPI processes we simply need to split the sum among the processes:

$$m_{pp} = \underbrace{\sum_i \sum_k G_k^2 \delta_{ki}^G \delta_{ip}^P}_{\text{overall sum}} = \underbrace{\sum_i \sum_k G_k^2 \delta_{ki}^G \delta_{ip}^P}_{\text{MPI process \#1}} + \underbrace{\sum_i \sum_k G_k^2 \delta_{ki}^G \delta_{ip}^P}_{\text{MPI process \#2}} + \cdots , \tag{3.10}$$

where δ_{ki}^G is nonzero only if gain G_k must be applied to the i-th sample of the TOD, and δ_{ip}^P is nonzero only if the i-th sample of the TOD falls within pixel p. Therefore, we allow each MPI process to compute the sum on its own, and then we collect all the results from the processes and sum them together. At the end, each MPI process will compute a subset of the diagonal coefficients for $\boldsymbol{M}$, as shown in Fig. 3.3.

Since we are going to assemble together many sums from the processes, we implement the algorithm described above in the function `sum_local_results`:

51 ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◂50 52b▸

```
def sum_local_results(mpi_comm, function, **arguments):
    log.debug('entering sum_local_results')
    result = function(**arguments)

    if mpi_comm:
        totals = np.zeros_like(result)
        mpi_comm.Allreduce(sendbuf=result, recvbuf=totals, op=MPI.SUM)
        return totals
    else:
        return result
```

Defines:
`sum_local_results`, used in chunks 52c and 54b.

This function takes as arguments a Python function and a list of arguments. It runs the function, collect the result and then coordinates with all the MPI processes to sum it together: this is the purpose of `mpi_comm.Allreduce`. On exit, each MPI process has the overall sum available in the variable `totals`.

We can finally turn to the calculation of $\boldsymbol{M}$. Since it is a diagonal matrix, we just keep in memory

its diagonal elements. The following Fortran function computes one partial ("local") sum of the ones in Eq. (3.10):

52a ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁49b 53a▷

```
subroutine compute_diagm_locally(gains, samples_per_gainp, pix_idx, output)
  real(kind=8), dimension(:), intent(in) :: gains
  integer(kind=8), dimension(size(gains)), intent(in) :: samples_per_gainp
  integer(kind=8), dimension(:), intent(in) :: pix_idx
  real(kind=8), dimension(:), intent(inout) :: output

  integer(kind=8) :: cur_gainp_idx, samples_in_gainp
  integer(kind=8) :: i

  output = 0
  cur_gainp_idx = 1
  samples_in_gainp = 0

  do i = 1, size(pix_idx)
     output(pix_idx(i) + 1) = output(pix_idx(i) + 1) + gains(cur_gainp_idx)**2
     samples_in_gainp = samples_in_gainp + 1

     if (samples_in_gainp .ge. samples_per_gainp(cur_gainp_idx)) then
        cur_gainp_idx = cur_gainp_idx + 1
        samples_in_gainp = 0
     endif
  enddo

end subroutine compute_diagm_locally
```

Defines:
`ftnroutines.compute_diagm_locally`, used in chunk 52b.
Uses `compute_diagm_locally` 52b.

The Python wrapper is the following:

52b ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁51 52c▷

```
def compute_diagm_locally(a: OfsAndGains, pix_idx, num_of_pixels: int):
    log.debug('entering compute_diagm_locally')
    result = np.empty(num_of_pixels, dtype='float')
    ftnroutines.compute_diagm_locally(a.gains, a.samples_per_gainp, pix_idx, result)
    return result
```

Defines:
`compute_diagm_locally`, used in chunk 52.
Uses `ftnroutines.compute_diagm_locally` 52a and `OfsAndGains` 44.

Computing the full diagonal for $\boldsymbol{M}$ is just a matter of combining `ftnroutines.compute_diagm_locally` with `sum_local_results`:

52c ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁52b 53b▷

```
def compute_diagm(mpi_comm, a: OfsAndGains, pix_idx, num_of_pixels: int):
    log.debug('entering compute_diagm')
    return sum_local_results(mpi_comm, function=compute_diagm_locally,
                             a=a,
                             pix_idx=pix_idx,
                             num_of_pixels=num_of_pixels)
```

Defines:
`compute_diagm`, used in chunks 55a, 68a, 73, and 77e.
Uses `compute_diagm_locally` 52b, `OfsAndGains` 44, and `sum_local_results` 51.

3.10.5. From TODs to maps. We now implement the operation described by matrix $\tilde{\boldsymbol{P}}$ (Eq. 1.12). This operation is similar to the application of $\boldsymbol{P}$, the pointing matrix, which scans a map into a TOD; however, $\tilde{\boldsymbol{P}}$ requires each sample in the TOD to be multiplied by a gain value. As it was the case for

`apply_f`, we can trivially split this computation among MPI processes by using each of them to produce one chunk of the overall TOD. Since the samples to be processed are going to be a huge number, we implement the core of the calculation in Fortran:

53a ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁52a 53c▷

```
subroutine apply_ptilde(map_pixels, gains, samples_per_gainp, pix_idx, output)
  real(kind=8), dimension(:), intent(in) :: map_pixels
  real(kind=8), dimension(:), intent(in) :: gains
  integer(kind=8), dimension(size(gains)), intent(in) :: samples_per_gainp
  integer(kind=8), dimension(:), intent(in) :: pix_idx
  real(kind=8), dimension(size(pix_idx)), intent(out) :: output

  integer(kind=8) :: cur_gainp_idx, samples_in_gainp
  integer(kind=8) :: i

  output = 0
  cur_gainp_idx = 1
  samples_in_gainp = 0

  do i = 1, size(pix_idx)
     output(i) = output(i) + map_pixels(pix_idx(i) + 1) * gains(cur_gainp_idx)
     ⟨Increment samples_in_gainp 48c⟩
  enddo

end subroutine apply_ptilde
```

Defines:
`ftnroutines.apply_ptilde`, used in chunk 53b.
Uses `apply_ptilde` 53b.

The Python wrapper to `ftnroutines.apply_ptilde` is trivial to implement:

53b ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁52c 54a▷

```
def apply_ptilde(map_pixels, a: OfsAndGains, pix_idx):
    log.debug('entering apply_ptilde')
    return ftnroutines.apply_ptilde(map_pixels, a.gains, a.samples_per_gainp, pix_idx)
```

Defines:
`apply_ptilde`, used in chunks 53a, 55a, 73, and 78a.
Uses `ftnroutines.apply_ptilde` 53a and `OfsAndGains` 44.

3.10.6. From maps to TODs. The opposite operation of $\tilde{\boldsymbol{P}}$ is $\tilde{\boldsymbol{P}}^T$, which bins a TOD into a map after having multiplied each sample by the associated gain factor. In this case, we make each MPI process produce a map, and then we sum all the maps together; the reason why this work is the same as for the computation of $\boldsymbol{M}$ (Sect. 3.10.4).

The following routine returns in `output` the map, given the TOD in `vector` and the gains in `gains`. Note that `output` must already have been allocated.

53c ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁53a 55b▷

```
subroutine apply_ptildet_locally(vector, gains, samples_per_gainp, pix_idx, output)
  real(kind=8), dimension(:), intent(in) :: vector
  real(kind=8), dimension(:), intent(in) :: gains
  integer(kind=8), dimension(size(gains)), intent(in) :: samples_per_gainp
  integer(kind=8), dimension(:), intent(in) :: pix_idx
  real(kind=8), dimension(:), intent(inout) :: output

  integer(kind=8) :: cur_gainp_idx, samples_in_gainp
  integer(kind=8) :: i

  output = 0
  cur_gainp_idx = 1
  samples_in_gainp = 0
```

```
    do i = 1, size(pix_idx)
       output(pix_idx(i) + 1) = output(pix_idx(i) + 1) + vector(i) * gains(cur_gainp_idx)
       ⟨Increment samples_in_gainp 48c⟩
    enddo
  end subroutine apply_ptildet_locally
```

Defines:
ftnroutines.apply_ptildet_locally, used in chunk 54a.
Uses apply_ptildet_locally 54a.

The Python wrapper to `ftnroutines.apply_ptildet_locally` allocates space for the result using `np.empty`:

54a ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁53b 54b▷

```
def apply_ptildet_locally(vector, a: OfsAndGains, pix_idx, num_of_pixels: int):
    log.debug('entering apply_ptildet_locally')
    result = np.empty(num_of_pixels, dtype='float')
    ftnroutines.apply_ptildet_locally(vector, a.gains,
                                      a.samples_per_gainp,
                                      pix_idx, result)
    return result
```

Defines:
apply_ptildet_locally, used in chunks 53c and 54b.
Uses ftnroutines.apply_ptildet_locally 53c and OfsAndGains 44.

Finally, to apply $\tilde{\boldsymbol{P}}^T$, we need to collect all the maps from the MPI processes and sum them together. We can use `sum_local_results`, which we originally used to sum real numbers, as it is generic enough to accept vectors as well:

54b ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁54a 55a▷

```
def apply_ptildet(mpi_comm, vector, a: OfsAndGains, pix_idx,
                  num_of_pixels: int):
    log.debug('entering apply_ptildet')
    return sum_local_results(mpi_comm, function=apply_ptildet_locally,
                             vector=vector,
                             a=a,
                             pix_idx=pix_idx,
                             num_of_pixels=num_of_pixels)
```

Defines:
apply_ptildet, used in chunks 55a, 68a, 73, and 78b.
Uses apply_ptildet_locally 54a, OfsAndGains 44, and sum_local_results 51.

3.10.7. More high-level calculations. We now turn to the implementation of matrix multiplication with $\boldsymbol{Z}$ (Eq. 1.20) and $\boldsymbol{A}$ (Eq. 1.14).

Multiplication by matrix $\boldsymbol{Z}$ takes a TOD as input and produces a TOD as output. Its purpose is to "clean" the TOD from the sky signal and the offset baselines, in order to make the residuals due to errors in the baselines/gains and white noise to show up.

As the implementation of this kind of functions need to process both monopole and dipole maps, we implement a new datatype that keeps them together in a consistent way:

54c ⟨*Datatypes used by* `calibrate.py` 28⟩+≡ (25) ◁45b

```
class MonopoleAndDipole:
    def __init__(self, mask, dipole_map):
        if mask is not None:
            self.monopole_map = np.array(mask, dtype='float')
        else:
            self.monopole_map = np.ones_like(dipole_map)

        self.dipole_map = np.array(dipole_map) * np.array(self.monopole_map)
```

Defines:
MonopoleAndDipole, used in chunks 42b, 55–57, 62a, 63b, 66b, 73, and 74b.

If a mask is provided, pixels in the monopole map are equal to one only if the corresponding pixel in the mask nonzero, otherwise they are set to zero. (If no mask is present, all the pixels are set to 1.) The dipole map is multiplied by the monopole map so that zero pixels are copied from the latter to the former.

Multiplication by $\boldsymbol{Z}$ is done by applying the definition:

$$\boldsymbol{Z} = \boldsymbol{I} - \tilde{\boldsymbol{P}}\left(\boldsymbol{M} + \boldsymbol{C}_m^{-1}\right)^{-1}\tilde{\boldsymbol{P}}^T\boldsymbol{C}_n^{-1}, \qquad \text{(1.20 revisited)}$$

We have all the functions needed to implement this calculation available, so it is just a matter of combining them in the proper way:

55a ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁54b 56a▷

```
def apply_z(mpi_comm, vector, a: OfsAndGains, pix_idx, mc: MonopoleAndDipole):
    log.debug('entering apply_z')
    binned_map = apply_ptildet(mpi_comm, vector, a, pix_idx,
                               len(mc.dipole_map))
    diagM = compute_diagm(mpi_comm, a, pix_idx, len(mc.dipole_map))

    nonzero_hit_mask = diagM != 0
    inv_diagM = np.zeros_like(diagM)
    inv_diagM[nonzero_hit_mask] = 1.0 / diagM[nonzero_hit_mask]

    binned_map = np.multiply(binned_map, inv_diagM)
    monopole_dot = np.dot(mc.monopole_map, binned_map)
    dipole_dot = np.dot(mc.dipole_map, binned_map)

    # Compute (m_c^T M^-1 m_c)
    small_matr = np.array([[np.dot(mc.dipole_map,
                                   np.multiply(inv_diagM, mc.dipole_map)),
                            np.dot(mc.dipole_map, inv_diagM)],
                           [np.dot(mc.monopole_map,
                                   np.multiply(inv_diagM, mc.dipole_map)),
                            np.dot(mc.monopole_map, inv_diagM)]])
    small_matr_prod = np.linalg.inv(small_matr) @ np.array([dipole_dot,
                                                            monopole_dot])

    ftnroutines.clean_binned_map(inv_diagM, mc.dipole_map, mc.monopole_map,
                                 small_matr_prod, binned_map)

    return vector - apply_ptilde(binned_map, a, pix_idx)
```

Defines:
apply_z, used in chunks 56, 73, and 78c.
Uses apply_ptilde 53b, apply_ptildet 54b, compute_diagm 52c, ftnroutines.clean_binned_map 55b, MonopoleAndDipole 54c, and OfsAndGains 44.

The call to `ftnroutines.clean_binned_map` is required in order to speed up the code. The Fortran implementation of the function is the following:

55b ⟨*Fortran routines used by* `calibrate.py` 35a⟩+≡ (26a) ◁53c

```
subroutine clean_binned_map(inv_diagM, dipole_map, monopole_map, small_matr_prod, binned_map)
  real(kind=8), dimension(:), intent(in) :: inv_diagM
  real(kind=8), dimension(size(inv_diagM)), intent(in) :: dipole_map
  real(kind=8), dimension(size(inv_diagM)), intent(in) :: monopole_map
  real(kind=8), dimension(2), intent(in) :: small_matr_prod
  real(kind=8), dimension(size(inv_diagM)), intent(inout) :: binned_map

  binned_map = binned_map - inv_diagM * (dipole_map * small_matr_prod(1) + &
       monopole_map * small_matr_prod(2))

end subroutine clean_binned_map
```

Defines:
ftnroutines.clean_binned_map, used in chunk 55a.

and it is equivalent to the following Python code:

```
binned_map -= inv_diagM * (mc.dipole_map * small_matr_prod[0] +
                           mc.monopole_map * small_matr_prod[1])
```

However, benckmarks on real-world sized simulations revealed that the Fortran routines makes the overall speed of the code roughly 10 % faster. This is probably due to the fact that the Python version must allocate a number of temporary vectors, and that each vector operation is implemented by means of its own loop. In Fortran, only one loop is used, and no temporary memory is needed at all.

Let's now turn to the implementation of the multiplication by $\boldsymbol{A} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{F} = \boldsymbol{F}^T \boldsymbol{Z} \boldsymbol{F}$ (dropping $\boldsymbol{C}_n^{-1}$ as usual):

56a ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁55a 56b▷

```
def apply_A(mpi_comm, a: OfsAndGains, sky_map, pix_idx,
            mc: MonopoleAndDipole, x: OfsAndGains):
    log.debug('entering apply_A')

    vector1 = apply_f(x, pix_idx, mc.dipole_map, sky_map)
    vector2 = apply_z(mpi_comm, vector1, a, pix_idx, mc)
    return apply_ft(vector2, a, pix_idx, mc.dipole_map, sky_map)
```

Defines:
apply_A, used in chunks 57c, 73, and 78e.
Uses apply_f 49a, apply_ft 50, apply_z 55a, MonopoleAndDipole 54c, and OfsAndGains 44.

Vector $\boldsymbol{v}$ (Eq 1.16) is implemented in the same way, as its definition is very similar to $\boldsymbol{A}$:

56b ⟨*Matrix/vector multiplication functions* 49a⟩+≡ (25) ◁56a

```
def compute_v(mpi_comm, voltages, a: OfsAndGains, sky_map, pix_idx,
              mc: MonopoleAndDipole):
    log.debug('entering compute_v')
    vector = apply_z(mpi_comm, voltages, a, pix_idx, mc)
    return apply_ft(vector, a, pix_idx, mc.dipole_map, sky_map)
```

Defines:
compute_v, used in chunks 57c, 73, and 78d.
Uses apply_ft 50, apply_z 55a, MonopoleAndDipole 54c, and OfsAndGains 44.

3.10.8. The conjugate gradient algorithm. In this section we will implement the Conjugate Gradient (CG) algorithm, to compute the solution for the equation

$$\boldsymbol{A}\boldsymbol{a} = \boldsymbol{v}. \qquad \text{(1.14 revisited)}$$

The algorithm has been presented in Sect. 3.10.1, and it requires to compute the product between $\boldsymbol{A}$ and a vector containing the "instrumental parameters" mentioned in Fig. 3.2 (page 43), as well as a number of dot products.

Our code makes each MPI process run the main iteration of the CG algorithm concurrently. Our implementation for apply_A is already able to handle calculations distributed among many MPI processes; however, we still miss a function which computes the dot product in the same context.

The dot products used in the CG algorithm always involve instrumental parameters, i.e., the a_vec field of a OfsAndGains object. Each MPI process contains only those offset/gains period that "belong" to it. Therefore, to compute the global dot product between two vectors $\boldsymbol{a}$ and $\boldsymbol{a}'$

$$\boldsymbol{a} \cdot \boldsymbol{a}' = \sum a_i \cdot a_i', \qquad (3.11)$$

we split the sum among the MPI processes as we did in the computation of $\boldsymbol{M}$ (Eq. 3.10 on page 51). This is really easy to do:

56c ⟨*Implementation of the conjugate gradient method* 56c⟩≡ (25) 57c▷

```
def mpi_dot_prod(mpi_comm, x, y):
```

```
    log.debug('entering mpi_dot_prod')

    local_sum = np.dot(x, y)

    if mpi_comm is not None:
        return mpi_comm.allreduce(local_sum, op=MPI.SUM)
    else:
        return local_sum
```

We are going to implement the gradient algorithm presented in Sect. 3.10.1 on page 47, with an optional preconditioner passed in the argument `pcond`. Note that if `pcond == None`, the algorithm is plain CG. The difference between plain CG and preconjugate CG is in the term $\boldsymbol{r}$ becoming $\boldsymbol{M}^{-1}\boldsymbol{r}$. We can incorporate both cases by always using a variable z (`z`) instead of r (`r`), which is either set to $\boldsymbol{M}^{-1}\boldsymbol{r}$ or to $\boldsymbol{r}$, depending on the parameter `pcond` being either `None` or not:

57a ⟨*Set* `z` *by optionally applying* `pcond` *to* `r` 57a⟩≡ (57c)

```
if pcond is not None:
    z = pcond.apply_to(r)
else:
    z = copy(r)
```

Preconditioners are Python objects which implement the method `apply_to`. We are going to provide a few preconditioners later, in Sect. 3.11.

The function `conjugate_gradient` returns a 2-tuple containing the solution $\boldsymbol{a}$ for Eq. (1.14) and a list of the *stopping factors* $\{s_k\}$, i.e., the values computed at each iteration which quantify if $\boldsymbol{r}_k$ is "small enough" or not. In our case, the stopping factor s_k for $\boldsymbol{r}_k$ is defined as

$$s_k \equiv \sqrt{\boldsymbol{r}_k \cdot \boldsymbol{r}_k}. \tag{3.12}$$

Here is the code that updates `stopping_factor` and checks if convergence has been reached:

57b ⟨*Update* `stopping_factor` *using* `r.a_vec` 57b⟩≡ (57c)

```
stopping_factor = np.sqrt(mpi_dot_prod(mpi_comm, r.a_vec, r.a_vec))
log.info('conjugate_gradient: iteration %d/%d, stopping criterion: %.5e',
         k, max_iter, stopping_factor)

list_of_stopping_factors.append(stopping_factor)
if stopping_factor < threshold:
    return a, list_of_stopping_factors
```

The iteration ends at step k iff $s_k <$ `threshold`. Since it is not granted that the sequence of s_k is monotonically decreasing, the code keeps the lowest value for s_k and the corresponding `OfsAndGains` object into two variables, `best_stopping_factor` and `best_a`. If the loop ends because the maximum number of iterations (`max_iter`) has been reached, this is the configuration that will be returned to the caller.

57c ⟨*Implementation of the conjugate gradient method* 56c⟩+≡ (25) ◁56c

```
def conjugate_gradient(mpi_comm, voltages, start_a: OfsAndGains, sky_map,
                       pix_idx, mc: MonopoleAndDipole, pcond=None,
                       threshold=1e-9, max_iter=100):

    log.debug('entering conjugate_gradient')

    a = copy(start_a)
    residual = (compute_v(mpi_comm, voltages, a, sky_map, pix_idx, mc) -
                apply_A(mpi_comm, a, sky_map, pix_idx, mc, a))
    r = ofs_and_gains_with_same_lengths(source=start_a,
                                        a_vec=residual)

    k = 0
    list_of_stopping_factors = []
    ⟨Update stopping_factor using r.a_vec 57b⟩
```

⟨*Set* `z` *by optionally applying* `pcond` *to* `r` 57a⟩

```
old_r_dot = mpi_dot_prod(mpi_comm, z.a_vec, r.a_vec)
p = copy(z)
best_stopping_factor = stopping_factor
best_a = a

while True:
    k += 1
    if k >= max_iter:
        return best_a, list_of_stopping_factors

    Ap = apply_A(mpi_comm, start_a, sky_map, pix_idx, mc, p)
    gamma = old_r_dot / mpi_dot_prod(mpi_comm, p.a_vec, Ap)
    a.a_vec += gamma * p.a_vec
    r.a_vec -= gamma * Ap

    ⟨Update stopping_factor using r.a_vec 57b⟩
    if stopping_factor < best_stopping_factor:
        best_stopping_factor, best_a = stopping_factor, copy(a)

    ⟨Set z by optionally applying pcond to r 57a⟩
    new_r_dot = mpi_dot_prod(mpi_comm, z.a_vec, r.a_vec)
    p.a_vec = z.a_vec + (new_r_dot / old_r_dot) * p.a_vec

    old_r_dot = new_r_dot
```

Defines:
`conjugate_gradient`, used in chunks 66b, 73, and 79b.
Uses `apply_A` 56a, `compute_v` 56b, `MonopoleAndDipole` 54c, `ofs_and_gains_with_same_lengths` 45b, and `OfsAndGains` 44.

§ 3.11. Preconditioning the conjugate gradient algorithm

In this section we provide a few preconditioners for the CG algorithm implemented in Sect. 3.10.8. Preconditioners are a tool to speed up the convergence of the CG algorithm; they work by providing an approximation for $\boldsymbol{A}^{-1}$, the matrix to be inverted. In this section, we are going to implement two kinds of preconditioners:

1. A traditional Jacobi preconditioner (Sect. 3.11.5);
2. A more refined preconditioner, which we pompously call the *full preconditioner* (Sect. 3.11.4).

The full preconditioner should be closer to the value of $\boldsymbol{A}^{-1}$, but it might require considerably more memory. The Jacobi preconditioner uses far less memory.

There is no general rule that allows to choose the "right" preconditioner. You should run a few tests using each of the preconditioners, as well as one test using no preconditioner at all, in order to determine which is the most efficient solution. Keep in mind that any preconditioner should not affect the quality of the result, but only the speed of convergence. The only notable exception is when the convergence is so slow that `conjugate_gradient` quits before s_k having reached `threshold`, because the number of iterations reached `max_iter`: in this case, a preconditioner might help to get a better solution without the need to increase `max_iter`.

Since inverting matrix $\boldsymbol{A}$ is prohibitive (otherwise, why should we have written all the code presented so far?!), in the two preconditioners we are going to implement we use the following approximation:

$$\boldsymbol{A} = \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{Z} \boldsymbol{F} \approx \boldsymbol{F}^T \boldsymbol{C}_n^{-1} \boldsymbol{F}, \tag{3.13}$$

i.e., we drop the $\boldsymbol{Z}$ term entirely. (Remember that the purpose of this term subtract the part of a TOD that is due to the sky signal only and leaves only the noise part.)

3.11.1. A simple example. To see what is the meaning of the assumptions made in the previous paragraph, we introduce here a small example. We consider a TOD containing just 8 samples, which must be split into three offset periods and two gain periods. The three offset periods cover 3, 2, and 3 samples respectively; the gain periods cover 2 and 1 gain periods. Therefore, matrix $\boldsymbol{F}$ has the following shape:

$$\boldsymbol{F} = \begin{pmatrix} 1 & 0 & 0 & D_1 & 0 \\ 1 & 0 & 0 & D_2 & 0 \\ 1 & 0 & 0 & D_3 & 0 \\ 0 & 1 & 0 & D_4 & 0 \\ 0 & 1 & 0 & D_5 & 0 \\ 0 & 0 & 1 & 0 & D_6 \\ 0 & 0 & 1 & 0 & D_7 \\ 0 & 0 & 1 & 0 & D_8 \end{pmatrix} \tag{3.14}$$

The first three columns encode the information about the three offset periods, while the latter two columns are related to the two gain periods.

Our approximation for $\boldsymbol{A}$ is therefore

$$\boldsymbol{F}^T\boldsymbol{F} = \begin{pmatrix} 3 & 0 & 0 & \sum_{i=1}^{3} D_i & 0 \\ 0 & 2 & 0 & \sum_{i=4}^{5} D_i & 0 \\ 0 & 0 & 3 & 0 & \sum_{i=6}^{8} D_i \\ \sum_{i=1}^{3} D_i & \sum_{i=4}^{5} D_i & 0 & \sum_{i=1}^{5} D_i^2 & 0 \\ 0 & 0 & \sum_{i=6}^{8} D_i & 0 & \sum_{i=6}^{8} D_i^2 \end{pmatrix}. \tag{3.15}$$

We shall use this example in the next sections to illustrate a few implementation quirks in the code for our preconditioners.

3.11.2. Using preconditioners for error estimation. Preconditioners are useful for a topic that is apparently disconnected from their purpose of speeding up CG convergence: they allow to quantify the error in the gain and offset estimates produced by `Da Capo`. In explaining this concept, I will use a few ideas from the theory of destriping algorithms for map-making, see e.g. Kurki-Suonio et al. (2009). The theory developed in that paper can be easily adapted to our context.

The fundamental observation is that Eq. (1.13) implies that matrix $\boldsymbol{A}$ is the Fisher matrix of the offset/gains in $\boldsymbol{a}$, i.e.

$$\boldsymbol{A}_{ij} = \frac{\partial \chi^2}{\partial a_i \partial a_j}, \tag{3.16}$$

Thus, once the code finds the solution $\boldsymbol{a}$ of the `Da Capo` problem and updates all the elements of $\boldsymbol{A}$ with the solution (offsets, gains, and the sky map), element ij of the *inverse* of $\boldsymbol{A}$ is the covariance between coefficient a_i and a_j. This is a fundamental observation, because we have just built two preconditioners that estimate the value of $\boldsymbol{A}^{-1}$. We can use them to estimate the error associated with each offset/gain, with the following approximation: if $\boldsymbol{M} \approx \boldsymbol{A}^{-1}$ is the preconditioner, then

$$\sigma_i \gtrsim \sqrt{\boldsymbol{M}_{ii}}. \tag{3.17}$$

I used $\gtrsim$ in the equation above, because equality only holds if all the offsets and gains are independent: we are discarding the fact that off-diagonal elements in $\boldsymbol{A}^{-1}$ are nonzero. Thus, the left side of Eq. (3.17) only provides a lower bound for the offset and gain errors.

With the example in Eq. (3.15) at hand, we can also provide a simple interpretation for Eq. (3.17). Suppose that matrix $\boldsymbol{F'}^T\boldsymbol{F'}$ in Eq. (3.22) is diagonal, i.e., we neglect the off-diagonal terms: we'll see that this is a reasonable assumption once we suppose that there is no correlation[3] among the offsets

[3] This would be the case, had we decided to estimate each baseline/gain by considering only the samples within one period, instead of solving all the baselines and gains at the same time. It is a pedagogically interesting case, because it helps in understanding the reasoning that follows.

and gains. If the k-th element of vector $\boldsymbol{a}$ corresponds to an offset (this is the case for $k = 1, 2, 4$), then the error is

$$\sigma_k^{\text{ofs}} = \sqrt{M_{kk}} = \frac{1}{\sqrt{\left(\boldsymbol{F}'^T \boldsymbol{C}_n^{-1} \boldsymbol{F}\right)_{kk}}} = \frac{\sigma^{\text{noise}}}{\sqrt{N}}, \tag{3.18}$$

where σ^{noise} is the noise of the data (supposed constant), and N is the number of pixel hit count associated with the baseline. This means that, the more the pixel is observed during the offset period, the better the estimate of the offset: a reasonable assumption.

Similarly, if the k-th element of vector $\boldsymbol{a}$ corresponds to a gain (in our example, $k = 3, 5$), then

$$\sigma_k^{\text{gain}} = \sqrt{M_{kk}} = \frac{1}{\sqrt{\left(\boldsymbol{F}'^T \boldsymbol{C}_n^{-1} \boldsymbol{F}\right)_{kk}}} = \frac{\sigma^{\text{noise}}}{\sum_i D_i^2}. \tag{3.19}$$

In this case, the goodness of the estimate depends also on the strongness of the dipole signal: again, this is reasonable. (You can also check that measure units are consistent both in Eq. 3.18 and 3.19.)

The fact that in the examples above we assumed that matrix $\boldsymbol{F}'^T \boldsymbol{F}'$ be block-diagonal implies that the covariance σ_{kl} is zero everywhere: in other words, we are assuming that the error associated with any offset/gain does not depend on the others. This is clearly only an approximation.

It is evident that, in order to derive the value of σ_k, we need to stop neglecting the $\boldsymbol{C}_n$ term in equations like (1.20). (It is $\boldsymbol{C}_n$'s duty to make σ^{noise} appear in equations 3.18 and 3.19.) We are however going to follow a shortway: we assume that $\boldsymbol{C}_n$ is diagonal again, and we assume that $1/f$ noise is completely captured by the sequence of offsets. In this way, we can estimate the value of σ^{noise} to be used in equations 3.18 and 3.19 easily from the stream of voltages in the TOD.

To compute the RMS level of the data, we need to disentangle the white noise part from the sky signal and the $1/f$ noise. We can do it easily by means of the following trick. Consider two measurements made by a detector that have been taken at times t_0 and $t_0 + \delta t$, where $\delta t = 1/\nu_{\text{samp}}$ is the time interval between two consecutive readings of the Analog-to-Digital Converter in the detector. Then from Eq. (1.1) it follows that

$$\begin{aligned} V(t_0) &= G\big(T + D\big) + b + N, \\ V(t_0 + \delta t) &= G\big(T' + D'\big) + b + N', \end{aligned}$$

where the gain G and the offset b are the same, and $T \approx T'$, $D \approx D'$. Since the PDF of N is

$$P(N) \propto \exp\left(-\frac{N^2}{2\sigma_n^2}\right), \tag{3.20}$$

it follows that the variable

$$V(t_0 + \delta t) - V(t_0) \approx N' - N \tag{3.21}$$

does not depend on the sky signal nor on the 1/f noise characteristics, and whose PDF is a Gaussian with zero mean and RMS equal to $2\sigma_n$. Therefore, we can get an estimate of the RMS of the white noise by computing the variance between consecutive differences of the samples in the TOD. Here is a Python function which implements the idea and which will be used in the implementation of the two preconditioners:

60 ⟨*CG preconditioners and error estimation* 60⟩≡ (25) 62a▸

```
@jit
def compute_rms(signal: Any, samples_per_period: List[int]) -> Any:
    result = np.empty(len(samples_per_period))
    start_idx = 0
    for i, cur_samples in enumerate(samples_per_period):
        subarray = signal[start_idx:start_idx + cur_samples]
        if cur_samples % 2 > 0:
            result[i] = 0.5 * (np.var(subarray[1::2] - subarray[0:-1:2]))
        else:
```

```
            result[i] = 0.5 * (np.var(subarray[1::2] - subarray[0::2]))

        start_idx += cur_samples

    return result
```

Defines:
compute_rms, used in chunks 62a, 63b, 73, and 80b.

Note the presence of `@jit`: we use Numba to speed it up, as it might need to run the iteration on a large set of periods.

3.11.3. Row permutation in preconditioners. The shape of $\boldsymbol{F}^T\boldsymbol{F}$, our approximation for $\boldsymbol{A}$, is quite sparse, but it is however not easy to invert, as you can see from Eq. (3.15). We can simplify the problem if we apply a permutation matrix[4] $\boldsymbol{\pi}$ so that each gain column follows the corresponding offset columns:

$$\boldsymbol{F'}^T\boldsymbol{F'} = \begin{pmatrix} 3 & 0 & \sum_{i=1}^{3} D_i & 0 & 0 \\ 0 & 2 & \sum_{i=4}^{5} D_i & 0 & 0 \\ \sum_{i=1}^{3} D_i & \sum_{i=4}^{5} D_i & \sum_{i=1}^{5} D_i^2 & 0 & 0 \\ 0 & 0 & 0 & 3 & \sum_{i=6}^{8} D_i \\ 0 & 0 & 0 & \sum_{i=6}^{8} D_i & \sum_{i=6}^{8} D_i^2 \end{pmatrix}, \tag{3.22}$$

which is block-diagonal. This transformation $\boldsymbol{F'}^T\boldsymbol{F'} = \boldsymbol{\pi}\boldsymbol{F}^T\boldsymbol{F}$ is possible with the following permutation matrix:

$$\boldsymbol{\pi} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.23}$$

Now, what's the point of using $\boldsymbol{\pi}$? The interesting fact about permutation matrices is that they are orthogonal, i.e., $\boldsymbol{\pi}^T\boldsymbol{\pi} = \boldsymbol{I}$. Therefore,

$$\left(\boldsymbol{F}^T\boldsymbol{F}\right)\boldsymbol{a} = \left(\boldsymbol{\pi}^T\boldsymbol{F'}^T\boldsymbol{F'}\boldsymbol{\pi}\right)\boldsymbol{a},$$

which is intuitive: we can either use $\boldsymbol{F}$ or $\boldsymbol{F'}$, but in the latter case we must scramble the elements in $\boldsymbol{a}$ using $\boldsymbol{\pi}$ and the elements in the result using $\boldsymbol{\pi}^T$, if we want to get the same result. From the fact that

$$\boldsymbol{F}^T\boldsymbol{F} = \boldsymbol{\pi}^T\boldsymbol{F'}^T\boldsymbol{F'}\boldsymbol{\pi} \tag{3.24}$$

we can write its inverse as

$$\left(\boldsymbol{F}^T\boldsymbol{F}\right)^{-1} = \left(\boldsymbol{\pi}^T\boldsymbol{F'}^T\boldsymbol{F'}\boldsymbol{\pi}\right)^{-1} = \tag{3.25}$$

$$= \left(\boldsymbol{\pi}^T\right)^{-1}\left(\boldsymbol{F'}^T\boldsymbol{F'}\right)^{-1}\left(\boldsymbol{\pi}\right)^{-1} = \tag{3.26}$$

$$= \boldsymbol{\pi}\left(\boldsymbol{F'}^T\boldsymbol{F'}\right)^{-1}\boldsymbol{\pi}^T. \tag{3.27}$$

Applying $\boldsymbol{\pi}$ and $\boldsymbol{\pi}^T$ does not require to keep the full form of matrix $\boldsymbol{\pi}$ in memory: it is enough to properly swap the terms in the `for` loops which implement the calculation. This means that we can write a Python function which multiplies $\left(\boldsymbol{F'}^T\boldsymbol{F}\right)^{-1}$ by $\boldsymbol{a}$ with no intermediate passages.

3.11.4. Full preconditioner. We implement a class, `FullPreconditioner` which stores the inverse of the matrix $\boldsymbol{F'}^T\boldsymbol{F'}$ like the one in Eq. (3.22). Since the inverse of a block-diagonal matrix of the form

$$\begin{pmatrix} \boldsymbol{A}_{11} & 0 & \dots & 0 \\ 0 & \boldsymbol{A}_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \boldsymbol{A}_{nn} \end{pmatrix} \tag{3.28}$$

[4] A *permutation matrix* is a square matrix whose effect on a matrix $\boldsymbol{A}$ is to swap the order of the columns/rows in $\boldsymbol{A}$. The coefficients of $\boldsymbol{\pi}$ are either 1 or 0, with only one nonzero coefficient per row and column.

is again a block-diagonal matrix, in the form

$$\begin{pmatrix} \boldsymbol{A}_{11}^{-1} & 0 & \dots & 0 \\ 0 & \boldsymbol{A}_{22}^{-1} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \boldsymbol{A}_{nn}^{-1} \end{pmatrix}, \tag{3.29}$$

we need to compute the inverse of each sub-matrix along the diagonal. This is the reason why we anticipated that this kind of preconditioner might require a significant amount of memory; the size of a sub-matrix is equal to $(n+1)^2$, where n is the number of offset baselines per gain period. Since the inverse is not necessarily sparse, we need to keep every coefficient in memory. For situations[5] where there are many offset periods per gain period, this number might get huge.

The `FullPreconditioner` class we are going to implement stores the inverse of the sub-diagonal matrices related to those offset/gain periods that are "owned" by the MPI process.

62a ⟨*CG preconditioners and error estimation* 60⟩+≡ (25) ◁60 63b▷

```
class FullPreconditioner:
    def __init__(self, mc: MonopoleAndDipole, pix_idx,
                 samples_per_ofsp, samples_per_gainp):
        assert sum(samples_per_ofsp) == len(pix_idx)

        self.samples_per_ofsp = samples_per_ofsp
        self.samples_per_gainp = samples_per_gainp
        self.ofsp_per_gainp = \
            OfsAndGains.calc_ofsp_per_gainp(samples_per_ofsp, samples_per_gainp)

        assert sum(self.ofsp_per_gainp) == len(self.samples_per_ofsp)

        self.matrices = []
        ⟨Compute the inverse of each diagonal sub-matrix and save them in self.matrices 62b⟩

    def apply_to(self, a: OfsAndGains) -> OfsAndGains:
        cur_ofsp_idx = 0
        cur_gainp_idx = 0
        a_vec = np.empty(len(a.a_vec))
        result = ofs_and_gains_with_same_lengths(a, a_vec)

        ⟨Multiply offsets/gains by the sub-diagonal inverse matrices and save the result in result 63a⟩
        return result

    def compute_offset_errors(self, voltages, samples_per_ofsp):
        rms = compute_rms(voltages, samples_per_ofsp)
        return np.sqrt(rms * np.array([np.diag(x)[:-1] for x in self.matrices]).flatten())

    def compute_gain_errors(self, voltages, samples_per_gainp):
        rms = compute_rms(voltages, samples_per_gainp)
        return np.sqrt(rms * np.array([np.diag(x)[-1] for x in self.matrices]))
```

Defines:
`FullPreconditioner`, used in chunks 65a, 73, and 80a.
Uses `compute_rms` 60, `MonopoleAndDipole` 54c, `ofs_and_gains_with_same_lengths` 45b, `OfsAndGains` 44, and `OfsAndGains.calc_ofsp_per_gainp` 45a.

In the `FullPreconditioner.__init__` method, we calculate each of the $\boldsymbol{A}_{ii}^{-1}$ matrices in Eq. (3.29):

62b ⟨*Compute the inverse of each diagonal sub-matrix and save them in* `self.matrices` 62b⟩≡ (62a)

```
cur_ofsp_idx = 0
cur_sample_idx = 0
```

[5] This was not the case of the Da Capo implementation used in the Planck 2015 data release, as there was always just one offset period per gain period: in this case, the sub-diagonal matrices have always shape 2×2 and can be easily stored in memory.

```
for ofsp_in_cur_gainp in self.ofsp_per_gainp:
    cur_matrix = np.zeros((ofsp_in_cur_gainp + 1,
                           ofsp_in_cur_gainp + 1))

    first_sample = cur_sample_idx
    for i, cur_ofsp in enumerate(samples_per_ofsp[cur_ofsp_idx:(cur_ofsp_idx +
                                                                ofsp_in_cur_gainp)]):
        cur_monopole = mc.monopole_map[pix_idx[cur_sample_idx:(cur_sample_idx + cur_ofsp)]]
        cur_dipole = mc.dipole_map[pix_idx[cur_sample_idx:(cur_sample_idx + cur_ofsp)]]
        cur_matrix[i, i] = np.sum(cur_monopole)
        cur_matrix[ofsp_in_cur_gainp, i] = cur_matrix[i, ofsp_in_cur_gainp] = \
                np.sum(cur_dipole)

        cur_sample_idx += cur_ofsp

    cur_matrix[ofsp_in_cur_gainp, ofsp_in_cur_gainp] = \
        np.sum(mc.dipole_map[pix_idx[first_sample:cur_sample_idx]]**2)

    # If the determinant is not positive, the matrix is not positive definite!
    assert np.linalg.det(cur_matrix) > 0

    self.matrices.append(np.linalg.inv(cur_matrix))
```

In the `FullPreconditioner.apply_to` method, the calculation is a bit more involved than a simple call to `@`, Python's matrix multiplication operator. Remember that we need to implement the equivalent of the π permutation, so we have to jump back-and-forth between offsets and gains:

63a ⟨*Multiply offsets/gains by the sub-diagonal inverse matrices and save the result in* `result` 63a⟩≡ (62a)

```
gains = a.gains
offsets = a.offsets
for cur_gainp_idx, num_of_ofsp in enumerate(self.ofsp_per_gainp):
    # y = M^-1 x   for each block in F^T F
    x = np.empty(num_of_ofsp + 1)
    x[0:num_of_ofsp] = offsets[cur_ofsp_idx:(cur_ofsp_idx + num_of_ofsp)]
    x[num_of_ofsp] = gains[cur_gainp_idx]
    y = self.matrices[cur_gainp_idx] @ x

    result.offsets[cur_ofsp_idx:(cur_ofsp_idx + num_of_ofsp)] = y[0:num_of_ofsp]
    result.gains[cur_gainp_idx] = y[num_of_ofsp]
    cur_ofsp_idx += num_of_ofsp
    cur_gainp_idx += 1
```

3.11.5. Jacobi preconditioner. The Jacobi preconditioner approximates $\boldsymbol{A}^{-1}$ by considering only its diagonal terms, using the following approximation:

$$\boldsymbol{A}^{-1} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{12} & \dots & a_{2n} \\ a_{31} & a_{12} & \dots & a_{3n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}^{-1} \approx \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{nn} \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{a_{11}} & 0 & \dots & 0 \\ 0 & \frac{1}{a_{22}} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \frac{1}{a_{nn}} \end{pmatrix} \tag{3.30}$$

Clearly, this approximation works well if matrix $\boldsymbol{A}$ is diagonal-dominant. It is a considerably simpler preconditioner than the one implemented in Sect. 3.11.4, but it has the advantage of requiring an amount of memory which is considerably smaller.

63b ⟨*CG preconditioners and error estimation* 60⟩+≡ (25) ◁62a 65a▷

```
class JacobiPreconditioner:
    def __init__(self, mc: MonopoleAndDipole, pix_idx,
                 samples_per_ofsp, samples_per_gainp):
```

```
        self.diagonal = OfsAndGains(offsets=np.zeros(len(samples_per_ofsp)),
                                    gains=np.zeros(len(samples_per_gainp)),
                                    samples_per_ofsp=samples_per_ofsp,
                                    samples_per_gainp=samples_per_gainp)

        ⟨Compute the inverses of the hit counts and save them in self.diagonal 64a⟩
        ⟨Compute the inverses of the squared dipole amplitudes and save them in self.diagonal 64b⟩

    def apply_to(self, a: OfsAndGains) -> OfsAndGains:
        return ofs_and_gains_with_same_lengths(a, a.a_vec * self.diagonal.a_vec)

    def compute_offset_errors(self, voltages, samples_per_ofsp):
        rms = compute_rms(voltages, samples_per_ofsp)
        return np.sqrt(rms * self.diagonal.offsets)

    def compute_gain_errors(self, voltages, samples_per_gainp):
        rms = compute_rms(voltages, samples_per_gainp)
        return np.sqrt(rms * self.diagonal.gains)
```

Defines:
JacobiPreconditioner, used in chunks 65a, 73, and 80a.
Uses compute_rms 60, MonopoleAndDipole 54c, ofs_and_gains_with_same_lengths 45b, and OfsAndGains 44.

In the case of the Jacobi preconditioner, there is no need to play with permutation matrices, so we can process offsets and gains separatedly. The part of the preconditioning matrix which depends on the offsets is simply an "hit count" of the number of samples that fall within each offset period. We need to consider any mask the user might have provided, so we go through `monopole_map`:

64a ⟨*Compute the inverses of the hit counts and save them in* `self.diagonal` 64a⟩≡ (63b)

```
cur_sample_idx = 0
offsets = self.diagonal.offsets
for row_idx, cur_sample_num in enumerate(samples_per_ofsp):
    cur_sum = np.sum(mc.monopole_map[pix_idx[cur_sample_idx:(cur_sample_idx +
                                                             cur_sample_num)]])
    if cur_sum != 0.0:
        offsets[row_idx] = 1.0 / cur_sum
    else:
        offsets[row_idx] = 1.0

    cur_sample_idx += cur_sample_num
```

The gain part requires us to access `dipole_map`, which MonopoleAndDipole sets to zero in each pixel that was masked. So the following code automatically takes into account the presence of masks:

64b ⟨*Compute the inverses of the squared dipole amplitudes and save them in* `self.diagonal` 64b⟩≡ (63b)

```
cur_sample_idx = 0
gains = self.diagonal.gains
for row_idx, cur_sample_num in enumerate(samples_per_gainp):
    cur_sum = np.sum(mc.dipole_map[pix_idx[cur_sample_idx:(cur_sample_idx +
                                                           cur_sample_num)]]**2)
    if cur_sum != 0.0:
        gains[row_idx] = 1.0 / cur_sum
    else:
        gains[row_idx] = 1.0

    cur_sample_idx += cur_sample_num
```

3.11.6. Putting all together. We now turn to the initialization of the preconditioner class in the body of the main function `configure_main`. We need a global variable which contains a dictionary, PCOND_DICT: this will be used to determine which preconditioner class to initialize, depending on the

string provided by the user in the configuration file.

65a ⟨*CG preconditioners and error estimation* 60⟩+≡ (25) ◁63b
```
PCOND_DICT = {'none': None,
              'full': FullPreconditioner,
              'jacobi': JacobiPreconditioner}
```
Defines:
PCOND_DICT, used in chunk 65b.
Uses FullPreconditioner 62a and JacobiPreconditioner 63b.

The keys for this dictionary are the strings the user must provide in the configuration file.

We can now implement the code in `calibrate_main` that initialize the preconditioner:

65b ⟨*Configure the preconditioner* 65b⟩≡ (26b)
```
try:
    pcond_class = PCOND_DICT[configuration.pcond]
except KeyError:
    log.error('Unknown preconditioner "%s", valid choices are: %s',
              configuration.pcond,
              ', '.join(['"{0}"'.format(x) for x in PCOND_DICT.keys()]))
    sys.exit(1)

if pcond_class is not None:
    pcond = pcond_class(mc=mc,
                        pix_idx=tod.pix_idx,
                        samples_per_ofsp=local_samples_per_ofsp,
                        samples_per_gainp=local_samples_per_gainp)
else:
    pcond = None
```
Uses PCOND_DICT 65a.

§ 3.12. Implementation of the Da Capo algorithm

Let's move to the most juicy part of the work: the implementation of the actual Da Capo algorithm. It might be a good idea to have a look at Sect. 1.2 again, before continuing.

Since Da Capo is an iterative procedure that improves the gain/offset solution, we need a starting guess for the gains G_k and offsets b_k in Eq. (1.1), as well as the sky map $\tilde{\boldsymbol{m}}$ (Eq. 1.18). We are going to adopt the shortest route for b_k and $\tilde{\boldsymbol{m}}$ by setting them all equal to zero, but for G_k we pick a more elaborate[6] choice. Specifically, we take Eq. (1.1) and consider the approximation

$$V_i = G_k\big(T_i + D_i\big) + b_k + N_i \approx G_k\, D_i, + \xi \tag{3.31}$$

which means that we suppose that $T_i \ll D_i$, and that the noise part reduces to a constant offset ξ. In this case, G_k is the slope of the least-squares solution of the fit between D_i and V_i. We implement this model in the following function:

65c ⟨*Implementation of the Da Capo algorithm* 65c⟩≡ (25) 66a▷
```
def guess_gains(voltages, pix_idx, dipole_map, samples_per_gainp):
    log.debug('entering guess_gains')

    result = np.empty(len(samples_per_gainp), dtype='float')
    cal_start = 0
    for gainp_idx, gainp_len in enumerate(samples_per_gainp):
        cal_end = cal_start + gainp_len
        cur_fit = scipy.polyfit(x=dipole_map[pix_idx[cal_start:cal_end]],
                                y=voltages[cal_start:cal_end],
                                deg=1)
        result[gainp_idx] = cur_fit[0]
```

[6] In theory, any positive, nonzero number for G_k would allow the algorithm to converge. However, picking a more accurate solution makes the code converge faster.

```
            cal_start += gainp_len

        return result
```

Defines:
guess_gains, used in chunk 66b.

Note that we do not consider any Galactic mask the user might have provided in the parameter file. After all, our guess for the set of G_k is just a starting point.

The function we are now going to implement, da_capo, implements the Da Capo algorithm and returns a variety of data structures. Instead of returning them as an anonymous tuple, we create a named tuple for the sake of clarity:

66a ⟨*Implementation of the Da Capo algorithm* 65c⟩+≡ (25) ◁65c 66b▷

```
DaCapoResults = namedtuple('DaCapoResults',
                           ['ofs_and_gains',
                            'sky_map',
                            'list_of_cg_rz',
                            'list_of_dacapo_rz',
                            'cg_wall_times',
                            'converged',
                            'dacapo_wall_time'])
```

We are now ready to implement da_capo, an implementation of the Da Capoalgorithm. The way this function split the job among the MPI processes is similar to conjugate_gradient: each process only "sees" the gain/offset periods that belong to it, and it keeps in memory only part of the TOD. The compute_map_corr function will be presented later; its purpose is to update the current estimate for the sky map.

66b ⟨*Implementation of the Da Capo algorithm* 65c⟩+≡ (25) ◁66a 67▷

```
def da_capo(mpi_comm, voltages, pix_idx, samples_per_ofsp, samples_per_gainp,
            mc: MonopoleAndDipole, mask=None, pcond=None, threshold=1e-9, max_iter=10,
            cg_threshold=1e-9, max_cg_iter=100) -> DaCapoResults:
    log.debug('entering da_capo')

    dacapo_prof = Profiler()

    sky_map = np.zeros_like(mc.dipole_map)
    start_gains = guess_gains(voltages, pix_idx, mc.dipole_map, samples_per_gainp)
    old_a = OfsAndGains(offsets=np.zeros(len(samples_per_ofsp)),
                        gains=start_gains,
                        samples_per_ofsp=samples_per_ofsp,
                        samples_per_gainp=samples_per_gainp)

    iteration = 0
    cg_wall_times = []
    list_of_cg_rz = []
    list_of_dacapo_rz = []
    while True:
        log.info('da_capo: iteration %d/%d', iteration + 1, max_iter)

        cg_prof = Profiler()
        new_a, rz = conjugate_gradient(mpi_comm, voltages, old_a, sky_map, pix_idx,
                                       mc, pcond=pcond, threshold=cg_threshold,
                                       max_iter=max_cg_iter)
        list_of_cg_rz.append(rz)
        cg_wall_times.append(cg_prof.toc())
        sky_map_corr = compute_map_corr(mpi_comm, voltages, old_a, new_a,
                                        pix_idx, mc.dipole_map, sky_map)
        sky_map += sky_map_corr
```

```
        stopping_factor = mpi_abs_max(mpi_comm, new_a.a_vec - old_a.a_vec)
        list_of_dacapo_rz.append(stopping_factor)
        log.info('da_capo: stopping factor %.3e (threshold is %.3e)',
                 stopping_factor, threshold)

        if stopping_factor < threshold:
            log.info('da_capo: convergence reached after %d steps', iteration)
            return DaCapoResults(ofs_and_gains=new_a,
                                 sky_map=sky_map,
                                 list_of_cg_rz=list_of_cg_rz,
                                 list_of_dacapo_rz=list_of_dacapo_rz,
                                 converged=True,
                                 cg_wall_times=cg_wall_times,
                                 dacapo_wall_time=dacapo_prof.toc())

        old_a = new_a
        iteration += 1

        if iteration >= max_iter:
            log.info('da_capo: maximum number of iterations reached (%d)', max_iter)
            return DaCapoResults(ofs_and_gains=new_a,
                                 sky_map=sky_map,
                                 list_of_cg_rz=list_of_cg_rz,
                                 list_of_dacapo_rz=list_of_dacapo_rz,
                                 converged=False,
                                 cg_wall_times=cg_wall_times,
                                 dacapo_wall_time=dacapo_prof.toc())
```

Defines:
da_capo, used in chunks 26b, 73, and 79b.
Uses compute_map_corr 68a, conjugate_gradient 57c, guess_gains 65c, MonopoleAndDipole 54c, mpi_abs_max 67, OfsAndGains 44, Profiler 27b, and toc 27b.

The da_capo function uses the helper function mpi_abs_max, which computes the maximum absolute value of a vector which is distributed[7] on the MPI processes:

67 ⟨*Implementation of the Da Capo algorithm* 65c⟩+≡ (25) ◁66b 68a▷

```
def mpi_abs_max(mpi_comm, vec):
    log.debug('entering mpi_abs_max')

    local_max = np.max(np.abs(vec))

    if mpi_comm is not None:
        return mpi_comm.allreduce(local_max, op=MPI.MAX)
    else:
        return local_max
```

Defines:
mpi_abs_max, used in chunk 66b.

The compute_map_corr function applies Eq. (1.18) in order to derive a better estimate for the sky map $\tilde{\boldsymbol{m}}$. (Remember that, during the first iteration, we set this map to zero identically.) The intuitive meaning of the following operations is the following:

1. We destripe the TOD by means of $\boldsymbol{F}$, and save the destriped TOD in diff_tod;

[7] An historical anecdote: the first versions of this code avoided calling allreduce on the result of np.max. The bug went unnoticed during the development, but once the code was used in production, it hanged in some repeatable circumstances. It turns out that if you forgot to call allreduce, then it happened sometimes that some MPI processes reach convergence before the others (because their new_a.a_vec was approximately equal to old_a.a_vec). They quit the loop within da_capo, while the other processes kept running it and hanged once one of the many allreduce calls used by conjugate_gradient was reached: they were waiting wait an answer from the "lucky" processes forever. During development, I used to use very small values for max_iter, so that the convergence was never fully reached in my tests.

2. We create the map using `apply_ptildet`; this function creates a map using a simple binning algorithm, which is the right thing to do here since we suppose that `diff_tod` no longer contains $1/f$ noise.

68a ⟨*Implementation of the Da Capo algorithm* 65c⟩+≡ (25) ◁67

```
def compute_map_corr(mpi_comm, voltages, old_a: OfsAndGains, new_a: OfsAndGains,
                     pix_idx, dipole_map, sky_map):
    log.debug('entering compute_map_corr')

    diff_tod = voltages - apply_f(new_a, pix_idx, dipole_map, sky_map)
    map_corr = apply_ptildet(mpi_comm, diff_tod, old_a, pix_idx, len(dipole_map))
    normalization = compute_diagm(mpi_comm, old_a, pix_idx, len(dipole_map))
    result = np.ma.array(map_corr, mask=(np.abs(normalization) < 1e-9),
                         fill_value=0.0)
    return (result / normalization).filled()
```

Defines:
`compute_map_corr`, used in chunks 66b, 73, and 79a.
Uses `apply_f` 49a, `apply_ptildet` 54b, `compute_diagm` 52c, and `OfsAndGains` 44.

Once the `da_capo` call has finished, the main loop in `calibrate_main` needs to gather all the offsets and gains spread among the MPI processes, in order to allow them to be saved (see Sect. 3.13). We must use some care in doing this task, as each MPI process has its own set of gains and offsets, whose lengths might differ from those held by the other processes. We implement a simple wrapper around MPI4Py's function `Gatherv`, which has the purpose of efficiently collecting NumPy arrays of varying lengths from MPI processes[8]). We first need to run `mpi_comm.gather` once to retrieve the length of each process' array, and then `mpi_comm.Gatherv` to send the actual array's data:

68b ⟨*Miscellaneous functions* 27b⟩+≡ (25) ◁27b

```
def gather_arrays(mpi_comm, array : Any, root=0) -> Any:
    lengths = mpi_comm.gather(len(array), root=root)
    if mpi_comm.Get_rank() == root:
        recvbuf = np.empty(sum(lengths), dtype=array.dtype)
    else:
        recvbuf = None

    mpi_comm.Gatherv(sendbuf=array, recvbuf=(recvbuf, lengths), root=root)
    return recvbuf
```

Defines:
`gather_arrays`, used in chunk 68c.

Note that only one of the MPI processes (traditionally called the "root process") will get the full sequence of offsets and gains. This is the reason why we initialized `recvbuf` only for the root process.

We can now use `gather_arrays` to collect the gains and offsets. If the user specified a preconditioner, we retrieve errors as well, otherwise we set all the errors to zero:

68c ⟨*Gather the results from the MPI processes and save them in* `coll_offsets` *and* `coll_gains` 68c⟩≡ (26b)

```
coll_gains = gather_arrays(mpi_comm, da_capo_results.ofs_and_gains.gains)
coll_offsets = gather_arrays(mpi_comm, da_capo_results.ofs_and_gains.offsets)
if pcond is not None:
    coll_gain_errors = gather_arrays(mpi_comm,
                                     pcond.compute_gain_errors(tod.signal,
                                                               local_samples_per_gainp))

    coll_offset_errors = gather_arrays(mpi_comm,
                                       pcond.compute_offset_errors(tod.signal,
                                                                   local_samples_per_ofsp))

else:
    log.warning('no preconditioner used, all offset/gain errors will be set to zero')
```

[8] MPI4Py also provides a simpler function, `Gather`, which assumes that all the processes send the same amount of samples.

```
    coll_offset_errors = np.zeros_like(coll_offsets)
    coll_gain_errors = np.zeros_like(coll_gains)
```

Uses gather_arrays 68b.

Now that we have all the results, it is time to save them to disk.

§ 3.13. Saving the results of the computation

We are at the end of the implementation. What is left off is the record of the results into a FITS file. Note that the following code is going to be executed only by the MPI process with rank #0. (If this surprises you, go back and take a look at the definition of `calibrate_main`).

3.13.1. Gains and offsets. The first step is to save offsets and gains. As we explained in Sect. 3.5, we are going to save a copy of the parameter file in the output files. We use a nice trick learned during the "Birds of a Feather" FITS session held during ADASS 2016, when M. Taylor explained how to embed metadata in the primary header of a FITS file. The idea is to convert textual information in a one-dimensional vector of raw bytes, which is saved as a $n \times 1$ bitmap image in the primary header of a FITS file. We use this trick to save the contents of the configuration file (kept in the `parameter_file_contents` of `CalibrateConfiguration` objects).

69a ⟨*Save the results of the calibration in the output file* 69a⟩≡ (26b) 69b▷

```
primary_hdu = fits.PrimaryHDU(data=configuration.parameter_file_contents)
```

Extracting such information from a fits file requires to convert the bytes back into characters and join them together to get a string. The following code shows how to print this information on the terminal:

```
with fits.open('file.fits') as f:
    print(''.join([chr(x) for x in f[0].data]))
```

We add a few more information in the header of the FITS file. Nearly all of these are already in the parameter file encoded in the primary HDU data, but having them available in the FITS metadata can be handy:

69b ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁69a 70a▷

```
primary_hdu.header.add_comment(configuration.comment)
primary_hdu.header['WTIME'] = (da_capo_results.dacapo_wall_time,
                               'Wall clock time [s]')
primary_hdu.header['MPIPROC'] = (mpi_comm.Get_size(),
                                 'Number of MPI processes used')
primary_hdu.header['CONVERG'] = (da_capo_results.converged,
                                 'Has the DaCapo algorithm converged?')
primary_hdu.header['GPPEROP'] = (configuration.periods_per_cal_constant,
                                 'Number of ofs periods per each gain period')
primary_hdu.header['CGSTOP'] = (configuration.cg_stop, 'Stopping factor for CG')
primary_hdu.header['CGMAXIT'] = (configuration.cg_maxiter, 'Maximum number of CG iterations')
primary_hdu.header['DCSTOP'] = (configuration.dacapo_stop, 'Stopping factor for DaCapo')
primary_hdu.header['DCMAXIT'] = (configuration.dacapo_maxiter, 'Maximum number of DaCapo iterations')
primary_hdu.header['PCOND'] = (configuration.pcond, 'Kind of preconditioner')
primary_hdu.header['NSIDE'] = (configuration.nside, 'Resolution of the map used by DaCapo')

if mask is None:
    fsky = 100.0
else:
    fsky = len(mask[mask > 0]) * 100.0 / len(mask)
primary_hdu.header['FSKY'] = (fsky, 'Fraction of the sky used by DaCapo')
```

We store the primary HDU in a list, which we extend immediately with the HDU containing a copy of the index file. The fact that they are the very first HDUs in the file allow this kind of files to be used as index files, i.e., we can pass the file containing the results of a computation in the line `index_file` of a parameter file to be passed to another call to `calibrate.py`. This allows to easily re-run computations with slightly different parameters for Da Capo.

70a ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁69b 70b▷

```
hdu_list = [primary_hdu] + index.store_in_hdus()
```

We now add two HDUs containing the offsets b_k and the gains G_k, respectively:

70b ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁70a 70c▷

```
cols = [fits.Column(name='OFFSET', array=np.array(coll_offsets).flatten(), format='1D'),
        fits.Column(name='ERR', array=np.array(coll_offset_errors).flatten(), format='1D'),
        fits.Column(name='NSAMPLES', array=samples_per_ofsp, format='1J')]
hdu_list.append(fits.BinTableHDU.from_columns(cols, name='OFFSETS'))

cols = [fits.Column(name='GAIN', array=np.array(coll_gains).flatten(), format='1D'),
        fits.Column(name='ERR', array=np.array(coll_gain_errors).flatten(), format='1D'),
        fits.Column(name='NSAMPLES', array=samples_per_gainp, format='1J')]
hdu_list.append(fits.BinTableHDU.from_columns(cols, name='GAINS'))
```

We do not save these HDUs immediately, as the user might have specified to save some other information in the file as well.

3.13.2. Sky map. If the user wants to save the sky map in the output file, we need to add another HDU to `hdu_list`. We are not using Healpy's `write_map` function, as this function always creates a new, stand-alone FITS file containing the map. Since we want the map to be saved together with the gains and the offsets, we build the HDU on our own using the basic functions provided by `astropy.io.fits`, and append the HDU to the list of HDUs to be saved.

70c ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁70b 70d▷

```
if configuration.save_map:
    col = fits.Column(name='SIGNAL', format='D', array=da_capo_results.sky_map)
    hdu = fits.BinTableHDU.from_columns([col], name='SKYMAP')

    hdu.header['PIXTYPE'] = ('HEALPIX', 'HEALPIX pixelisation')
    hdu.header['ORDERING'] = ('RING', 'Pixel ordering scheme, either RING or NESTED')
    hdu.header['NSIDE'] = (configuration.nside, 'Healpix"s resolution parameter')
    hdu.header['FIRSTPIX'] = (0, 'First pixel # (0-based)')
    hdu.header['LASTPIX'] = (healpy.nside2npix(configuration.nside) - 1,
                             'Last pixel # (0-based)')
    hdu.header['INDXSCHM'] = ('IMPLICIT', 'Indexing: IMPLICIT or EXPLICIT')
    hdu.header['OBJECT'] = ('FULLSKY', 'Sky coverage, either PARTIAL or FULLSKY')

    hdu_list.append(hdu)
```

3.13.3. Convergence information. The last chunk of data to save contains the information about the convergence of each Conjugated Gradient iteration. We save each iteration in a separate tabular HDU and append them to the output file.

70d ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁70c 71▷

```
if configuration.save_convergence is not None:
    for idx, cur_cg_rz_list, cur_dacapo_rz, cg_wall_time \
        in zip(range(len(da_capo_results.list_of_dacapo_rz)),
               da_capo_results.list_of_cg_rz,
               da_capo_results.list_of_dacapo_rz,
               da_capo_results.cg_wall_times):

        col = fits.Column(name='RZ', array=cur_cg_rz_list, format='1D')
        hdu = fits.BinTableHDU.from_columns([col], name='RZ{0:04d}'.format(idx))
```

```
hdu.header['DACAPORZ'] = (cur_dacapo_rz, 'DaCapo stopping factor')
hdu.header['WTIME'] = (cg_wall_time, 'CG wall clock time [s]')
hdu_list.append(hdu)
```

3.13.4. Flushing data and closing the file. Now that every HDU is in `hdu_list`, we save the file to disk. The `clobber=True` option tells Astropy not to complain if it must overwrite an existing file.

71 ⟨*Save the results of the calibration in the output file* 69a⟩+≡ (26b) ◁70d

```
fits.HDUList(hdu_list).writeto(configuration.output_file_name,
                               clobber=True)
log.info('gains and offsets written into file "%s"',
         configuration.output_file_name)
```

Chapter 4

Validation of the code

In this chapter, we implement a set of tests that verify the correctness of the functions implemented in `calibrate.py`. We want these tests to be as much automated as possible: we want the computer to check the correctness of what we have implemented in the previous chapters, with almost no use intervention. In this way, we can safely change our code and restructure it, with (almost) no fear of breaking it unknowingly, provided that we run such tests regularly.

There are two kinds of tests that are usually implemented in software projects:

1. *Unit tests* aim to test the behaviour of small routines, which usually do not depend on a complex environment and can be executed pretty fast.
2. *Integration tests* stress the whole software architecture, by running an end-to-end execution of the software and checking the overall results. They can take considerable time to run.

We are going to implement both kinds of tests in this chapter.

§ 4.1. Unit tests

We are going to use Python's `unittest` module to implement both of them. It is a module from the standard library which implements a nice way of organizing unit tests in "test cases", i.e., groups of tests doing similar things.

We implement here only unit tests for those routines that implement the mathematical model of the Da Capo algorithm. The idea is to build fake TODs containing a handful of samples, and then to build matrices like $\boldsymbol{F}$, $\boldsymbol{A}$, $\boldsymbol{Z}$ in their full form, and compare the result of a direct matrix computation with the results returned by functions like `apply_f`, `apply_A`, and `apply_z`.

Here is the skeleton of the `dacapo-test.py` script. It can be run from the command line with the command `make check`:

73 ⟨*dacapo-test.py* 73⟩≡

```
# -*- encoding: utf-8 -*-

import unittest as ut

import numpy as np
from calibrate import OfsAndGains, MonopoleAndDipole, \
    apply_f, apply_ft, compute_diagm, apply_ptilde, apply_ptildet, \
    apply_z, apply_A, compute_v, conjugate_gradient, compute_map_corr, \
    da_capo, JacobiPreconditioner, FullPreconditioner, compute_rms

⟨Helper test functions 74a⟩

class TestDaCapo(ut.TestCase):
    def setUp(self):
```

⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩

⟨*Test* `apply_f` 77c⟩
⟨*Test* `apply_ft` 77d⟩
⟨*Test* `compute_diagm` 77e⟩
⟨*Test* `apply_ptilde` 78a⟩
⟨*Test* `apply_ptildet` 78b⟩
⟨*Test* `apply_z` 78c⟩
⟨*Test* `apply_A` 78e⟩
⟨*Test* `compute_v` 78d⟩
⟨*Test* `compute_map_corr` 79a⟩
⟨*Test* `get_dipole_temperature` 80c⟩
⟨*Test the preconditioners* 79b⟩

```
class TestMiscellanea(ut.TestCase):
    ⟨Test compute_rms 80b⟩
```

Uses `apply_A` 56a, `apply_f` 49a, `apply_ft` 50, `apply_ptilde` 53b, `apply_ptildet` 54b, `apply_z` 55a, `compute_diagm` 52c, `compute_map_corr` 68a, `compute_rms` 60, `compute_v` 56b, `conjugate_gradient` 57c, `da_capo` 66b, `FullPreconditioner` 62a, `JacobiPreconditioner` 63b, `MonopoleAndDipole` 54c, and `OfsAndGains` 44.

The way we test things is the following: provided that we implemented the code for a matrix operation like the product

(4.1)

we compare the result of the call to the Python function with the result we get by carrying the full matricial calculation. This is a suitable approac because the TOD we are going to use has only a few samples and can therefore all the matrices we use are small enough to be kept entirely in memory.

To compare the results between full calculations and the call to the Python function we implemented in the previous chapters, we introduce a helper function:

74a ⟨*Helper test functions* 74a⟩≡ (73) 76a▸

```
def check_vector_match(name, vector1, vector2, rtol=1e-05):
    try:
        assert np.allclose(vector1, vector2, rtol=rtol)
        print('test "{0}" passed'.format(name))

    except:
        # This catches both failed assertions and mismatches in the
        # shapes of vec1/vec2
        print('ERROR: test "{0}" failed'.format(name))
        print('       vectors {0} and {1} are not the same'
              .format(vector1, vector2))
        raise
```

Defines:
`check_vector_match`, used in chunks 77–80.

The purpose of this function is to verify that all the elements in the two vectors `vector1` and `vector2` are "close enough" (we should not expect them to be exactly equal, because of the unavoidable rounding errors). If this does not happen, the code warns the user that the condition has not been satisfied and re-raises the exception. The warning message prints the two vectors, so that the user can see where the problem is. The `rtol` parameter quantifies the concept of "close enough": refer to the NumPy documentation of `allclose` for a precise explanation.

To verify the behaviour of the functions defined in the previous chapters, we must have some fake input data first. In the next section, we will instruct the code about how to create them.

4.1.1. Creating test data. The method `TestDaCapo.setUp` is called when the tests are going to start. Its purpose is to build the data structures that will be used in the actual tests. We begin by building a sky map:

74b ⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩≡ (73) 75a▸

```
self.num_of_pixels = 3
self.sky_map = np.array([-0.4, 0.2, 0.2])
self.D = np.sin(2 * np.pi * np.array([0, 1/3, 2/3]))
self.mc = np.array([self.D, [1, 1, 1]]).T
self.mon_and_dip = MonopoleAndDipole(mask=[1, 1, 1], dipole_map=self.D)
self.signal_sum_map = self.D + self.sky_map
```

Uses `MonopoleAndDipole` 54c.

Because of the way we defined functions in Chapter 3, it is not strictly needed to encode maps in Healpix format: it is enough for the code to know how to associate a direction in the sky with a pixel number. This is handy in our situation, as Healpix' lowest resolution is `NSIDE=1`, which has 12 pixels: too many for the modest requirements we have here! We build a sky map made by just three pixels, in `self.sky_map`. Note that the dot product between the map and the dipole `self.D` is zero, and that the average value of `self.sky_map` is zero: this was done in order to satisfy the constraints in Eq. 1.23 and Eq. 1.24.

Now that we have a map, we build a TOD and we specify the length of the offset and calibration periods. This is the time stream of the pixels seen at times $t_1, t_2, \dots t_{11}$:

75a ⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩+≡ (73) ◁74b 75b▷

```
self.pix_idx = np.array([0, 0, 1, 0, 1, 2, 2, 2, 0, 1, 0])
```

This corresponds to the pointing matrix

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}. \tag{4.2}$$

Next, we specify the offsets and gains: we choose here to use the same length for the two offset/gain periods, namely, 6 and 5 samples:

75b ⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩+≡ (73) ◁75a 77a▷

```
self.ofs_and_gains = OfsAndGains(offsets=np.array([10.0, 10.5]),
                                 gains=np.array([4.1, 4.2]),
                                 samples_per_ofsp=[6, 5],
                                 samples_per_gainp=[6, 5])

self.a_vec = self.ofs_and_gains.a_vec
self.Gext = np.repeat(self.ofs_and_gains.gains,
                      self.ofs_and_gains.samples_per_gainp)
self.bext = np.repeat(self.ofs_and_gains.offsets,
                      self.ofs_and_gains.samples_per_ofsp)
```

Uses `OfsAndGains` 44.

We save the `a_vec` member of `ofs_and_gains` in a new variable, because we are going to use it many times: this saves typing. The members `self.Gext` and `self.bext` contain the same offsets and gains in `self.ofs_and_gains`, but they have been repeated in order to make these two vectors of the same length as `self.pix_idx`:

$$\texttt{self.Gext} = \big(\underbrace{10.0\ 10.0\ 10.0\ 10.0\ 10.0\ 10.0}_{6\text{ times}}\ \underbrace{10.5\ 10.5\ 10.5\ 10.5\ 10.5}_{5\text{ times}}\big), \tag{4.3}$$

$$\texttt{self.Bext} = \big(\underbrace{4.1\ 4.1\ 4.1\ 4.1\ 4.1\ 4.1}_{6\text{ times}}\ \underbrace{4.2\ 4.2\ 4.2\ 4.2\ 4.2}_{5\text{ times}}\big). \tag{4.4}$$

Having members like `self.Gext` and `self.pixidx` is very handy for doing calculations, but if we want to check the correctness of the implementation, we should have matrices like $\boldsymbol{P}$, $\boldsymbol{F}$, etc. in their full form. The following function takes the data we have just defined and creates the matrices we need:

76a ⟨*Helper test functions* 74a⟩+≡ (73) ◁74a

```
def build_test_matrices(num_of_samples, num_of_pixels, pix_idx,
                        mc, signal_sum_map, ofs_and_gains):
    ⟨Build matrix P from pix_idx 76b⟩
    ⟨Build matrix F from ofs_and_gains 76c⟩
    ⟨Build matrices M, Z, and A 76d⟩

    return P, F, ptilde, M, Z, A
```

Defines:
`build_test_matrices`, used in chunks 77a and 79b.

Matrix $\boldsymbol{P}$ is built using `pix_idx`. The result will be like the matrix shown in Eq. (4.2):

76b ⟨*Build matrix* $\boldsymbol{P}$ *from* `pix_idx` 76b⟩≡ (76a)

```
P = np.zeros((len(pix_idx), num_of_pixels), dtype='int')
for i, pixel in enumerate(pix_idx):
    P[i][pixel] = 1
```

Next, we build $\boldsymbol{F}$. Our aim is to get a matrix like Eq. (1.11) in page 4, which in our example would be

$$\boldsymbol{F} = \begin{pmatrix} 1 & 0 & \texttt{self.D[0]} + \texttt{self.sky\_map[0]} & 0 \\ 1 & 0 & \texttt{self.D[0]} + \texttt{self.sky\_map[0]} & 0 \\ 1 & 0 & \texttt{self.D[1]} + \texttt{self.sky\_map[1]} & 0 \\ 1 & 0 & \texttt{self.D[0]} + \texttt{self.sky\_map[0]} & 0 \\ 1 & 0 & \texttt{self.D[1]} + \texttt{self.sky\_map[1]} & 0 \\ 1 & 0 & \texttt{self.D[2]} + \texttt{self.sky\_map[2]} & 0 \\ 0 & 1 & 0 & \texttt{self.D[2]} + \texttt{self.sky\_map[2]} \\ 0 & 1 & 0 & \texttt{self.D[2]} + \texttt{self.sky\_map[2]} \\ 0 & 1 & 0 & \texttt{self.D[0]} + \texttt{self.sky\_map[0]} \\ 0 & 1 & 0 & \texttt{self.D[1]} + \texttt{self.sky\_map[1]} \\ 0 & 1 & 0 & \texttt{self.D[0]} + \texttt{self.sky\_map[0]} \end{pmatrix}, \tag{4.5}$$

for the pointing matrix shown in Eq. (4.2). We build the matrix using two `for` loops: the first one builds the offset part (first 2 columns in the example above), the second one build the sky part.

76c ⟨*Build matrix* $\boldsymbol{F}$ *from* `ofs_and_gains` 76c⟩≡ (76a)

```
F = np.zeros((num_of_samples, len(ofs_and_gains.offsets) + len(ofs_and_gains.gains)))
start_idx = 0
for col, ofs_len in enumerate(ofs_and_gains.samples_per_ofsp):
    F[start_idx:(start_idx + ofs_len), col] = 1
    start_idx += ofs_len

start_idx = 0
for col, gain_len in enumerate(ofs_and_gains.samples_per_gainp):
    F[start_idx:(start_idx + gain_len), len(ofs_and_gains.offsets) + col] = \
        signal_sum_map[pix_idx[start_idx:(start_idx + gain_len)]]
    start_idx += gain_len
```

Finally, we build matrices $\boldsymbol{M}$ (Eq. 1.21, on page 5), $\boldsymbol{Z}$ (Eq. 1.20), and $\boldsymbol{A}$ (Eq. 1.15) using the definitions (remember that in Python 3 the `@` operator is matrix multiplication):

76d ⟨*Build matrices* $\boldsymbol{M}$, $\boldsymbol{Z}$, *and* $\boldsymbol{A}$ 76d⟩≡ (76a)

```
Gext = np.repeat(ofs_and_gains.gains, ofs_and_gains.samples_per_gainp)
ptilde = np.array([P[i] * Gext[i] for i in range(len(Gext))])
M = ptilde.T @ ptilde
invM = np.linalg.inv(M)
mat2x2 = mc.T @ invM @ mc
```

```
MCminv = invM - invM @ mc @ np.linalg.inv(mat2x2) @ mc.T @ invM
Z = np.eye(num_of_samples) - ptilde @ MCminv @ ptilde.T
A = F.T @ Z @ F
```

This completes the implementation of `build_test_matrices`. Let's turn back to the implementation of `TestDaCapo.setUp`, and use `build_test_matrices` to create the matrices and save them into `self` members:

77a ⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩+≡ (73) ◁75b 77b▷

```
self.P, self.F, self.ptilde, self.M, self.Z, self.A = \
    build_test_matrices(len(self.bext), self.num_of_pixels, self.pix_idx,
                        self.mc, self.signal_sum_map, self.ofs_and_gains)
```

Uses `build_test_matrices` 76a.

Finally, we build the TOD:

77b ⟨*Build the data structures containing the test TOD and save them in* `self` 74b⟩+≡ (73) ◁77a

```
self.tod = self.Gext * (self.P @ self.signal_sum_map) + self.bext
```

Defines:
`TestDaCapo.setUp`, never used.

4.1.2. Implementing unit tests. Now that `self` has been populated with fake input data, we can turn to the verification of the functions implemented in Chapter 3. The first function we test is `apply_f`, which performs a multiplication between matrix $\boldsymbol{F}$ (Eq. 1.11) and a vector of offsets and gains:

77c ⟨*Test* `apply_f` 77c⟩≡ (73)

```
def testApplyF(self):
    check_vector_match('apply_f',
                       self.F @ self.a_vec,
                       apply_f(self.ofs_and_gains, self.pix_idx,
                               self.D, self.sky_map))
```

Defines:
`TestDaCapo.testApplyF`, never used.
Uses `apply_f` 49a and `check_vector_match` 74a.

The code is really easy to follow: we use `check_vector_match` to test that the result of the matrix multiplication between `self.F` and `self.a_vec` matches the result of the call `apply_f` (which, as you might remember, does not build any matrix in memory).

Next, we test `apply_ft`:

77d ⟨*Test* `apply_ft` 77d⟩≡ (73)

```
def testApplyFT(self):
    check_vector_match('apply_ft',
                       self.F.T @ self.tod,
                       apply_ft(self.tod, self.ofs_and_gains, self.pix_idx,
                                self.D, self.sky_map))
```

Defines:
`TestDaCapo.testApplyFT`, never used.
Uses `apply_ft` 50 and `check_vector_match` 74a.

Next, we test `compute_diagm`:

77e ⟨*Test* `compute_diagm` 77e⟩≡ (73)

```
def testComputeDiagM(self):
    check_vector_match('compute_diagm',
                       np.diag(self.M),
                       compute_diagm(None, self.ofs_and_gains,
                                     self.pix_idx, self.num_of_pixels))
```

Defines:
`TestDaCapo.testComputeDiagM`, never used.
Uses `check_vector_match` 74a and `compute_diagm` 52c.

Next, we test apply_ptilde; nothing new here:

78a ⟨*Test* apply_ptilde 78a⟩≡ (73)
```
def testApplyPtilde(self):
    check_vector_match('apply_ptilde',
                       self.ptilde @ self.sky_map,
                       apply_ptilde(self.sky_map, self.ofs_and_gains,
                                    self.pix_idx))
```
Defines:
TestDaCapo.testApplyPtilde, never used.
Uses apply_ptilde 53b and check_vector_match 74a.

In testing apply_ptildet, we face a problem. The function calls MPI functions, but requiring to start N MPI processes to run an unit test is an overkill: remember that unit tests are meant to be quick and easy to run. Therefore, we pass None to the mpi_comm parameter: if you remember how we implemented this function, you will recall that in this case the function completely avoids MPI. Now you can understand one of the reasons behind that choice.

78b ⟨*Test* apply_ptildet 78b⟩≡ (73)
```
def testApplyPtildet(self):
    check_vector_match('apply_ptildet',
                       self.ptilde.T @ self.tod,
                       apply_ptildet(None, self.tod, self.ofs_and_gains,
                                     self.pix_idx, self.num_of_pixels))
```
Defines:
TestDaCapo.testApplyPtildet, never used.
Uses apply_ptildet 54b and check_vector_match 74a.

In testing apply_z, we use the same trick of setting mpi_comm to None:

78c ⟨*Test* apply_z 78c⟩≡ (73)
```
def testApplyZ(self):
    check_vector_match('apply_z',
                       self.Z @ self.tod,
                       apply_z(None, self.tod, self.ofs_and_gains,
                               self.pix_idx, self.mon_and_dip))
```
Defines:
TestDaCapo.testApplyZ, never used.
Uses apply_z 55a and check_vector_match 74a.

The same for compute_v:

78d ⟨*Test* compute_v 78d⟩≡ (73)
```
def testComputeV(self):
    check_vector_match('compute_v',
                       self.F.T @ self.Z @ self.tod,
                       compute_v(None, self.tod, self.ofs_and_gains, self.sky_map,
                                 self.pix_idx, self.mon_and_dip))
```
Defines:
TestDaCapo.testComputeV, never used.
Uses check_vector_match 74a and compute_v 56b.

Let's now discuss function apply_A. This function implements the matrix multiplication by $\boldsymbol{A}$, where the vector that is to be multiplied is a sequence of offsets and gains. However, remember that the value of the coefficients in $\boldsymbol{A}$ depend on the value of the offsets and the gains we are using within the Da Capo main loop (see the implementation of the da_capo function). If we want to properly test apply_A, we should multiply it by a vector that it does not coincide with the gains and offsets used to build $\boldsymbol{A}$ itself, because this would be a corner case, not very representative of the general case. We therefore build a new OfsAndGains object, ofs_gains_guess, which contains offsets and gains that are unrelated to the ones in self.ofs_and_gains. Of course, self.ofs_and_gains and ofs_gains_guess share the same structure (two periods of 6 and 5 samples each):

78e ⟨*Test* apply_A 78e⟩≡ (73)
```
def testApplyA(self):
```

```
        ofs_gains_guess = OfsAndGains(offsets=np.zeros(2),
                                      gains=np.ones(2),
                                      samples_per_ofsp=self.ofs_and_gains.samples_per_ofsp,
                                      samples_per_gainp=self.ofs_and_gains.samples_per_gainp)
        check_vector_match('apply_A',
                           self.A @ ofs_gains_guess.a_vec,
                           apply_A(None, self.ofs_and_gains, self.sky_map,
                                   self.pix_idx, self.mon_and_dip,
                           ofs_gains_guess))
```

Defines:
`TestDaCapo.testApplyA`, never used.
Uses `apply_A` 56a, `check_vector_match` 74a, and `OfsAndGains` 44.

We apply the same trick in testing `compute_map_corr`, building a new TOD (`fake_tod`) which contains the numbers in the range $0 \ldots 10$:

79a ⟨*Test* `compute_map_corr` 79a⟩≡ (73)

```
    def testComputeMapCorr(self):
        fake_tod = np.arange(len(self.tod))
        check_vector_match('compute_map_corr',
                           np.linalg.inv(self.ptilde.T @ self.ptilde) @
                           self.ptilde.T @ (fake_tod - self.F @ self.a_vec),
                           compute_map_corr(None, fake_tod, self.ofs_and_gains,
                                            self.ofs_and_gains,
                                            self.pix_idx, self.D, self.sky_map))
```

Defines:
`TestDaCapo.testComputeMapCorr`, never used.
Uses `check_vector_match` 74a and `compute_map_corr` 68a.

We want now to test the implementation of the two CG preconditioners, `JacobiPreconditioner` and `FullPreconditioner`. Since the kind of test we are going to apply is the same for the two objects, we implement the test in a generic method of the `TestDaCapo` class. This method accepts a `precObj` as a parameter, which is the class to be used in initializing the preconditioner. The test exercises the `conjugate_gradient` method, which supposes that the offsets and gains are unknown. We must therefore not use objects like `self.A`, `self.Z` and so on, because they were built using the exact solution of the CG problem! We build a new guess for the offsets and gains in `ofs_gains_guess`, and we call `build_test_matrices` to build the matrices we need from these guesses:

79b ⟨*Test the preconditioners* 79b⟩≡ (73) 80a▷

```
    def precondition_check(self, precObj):
        ofs_gains_guess = OfsAndGains(offsets=np.zeros(2),
                                      gains=np.ones(2),
                                      samples_per_ofsp=self.ofs_and_gains.samples_per_ofsp,
                                      samples_per_gainp=self.ofs_and_gains.samples_per_gainp)

        P, F, ptilde, M, Z, A = build_test_matrices(len(self.bext),
                                                    self.num_of_pixels, self.pix_idx,
                                                    self.mc, self.D,
                                                    ofs_gains_guess)

        if precObj:
            pcond = precObj(mc=self.mon_and_dip,
                            pix_idx=self.pix_idx,
                            samples_per_ofsp=self.ofs_and_gains.samples_per_ofsp,
                            samples_per_gainp=self.ofs_and_gains.samples_per_gainp)
        else:
            pcond = None

        cg_a, _ = conjugate_gradient(None, self.tod, ofs_gains_guess,
                                     np.zeros_like(self.D),
                                     self.pix_idx, self.mon_and_dip, pcond=pcond)
```

```
    check_vector_match('conjugate_gradient',
                       np.linalg.inv(A) @ F.T @ Z @ self.tod, cg_a.a_vec)

    result = da_capo(mpi_comm=None,
                     voltages=self.tod,
                     pix_idx=self.pix_idx,
                     samples_per_ofsp=self.ofs_and_gains.samples_per_ofsp,
                     samples_per_gainp=self.ofs_and_gains.samples_per_gainp,
                     mc=self.mon_and_dip,
                     pcond=pcond)
    check_vector_match('da_capo (offsets)',
                       self.ofs_and_gains.offsets, result.ofs_and_gains.offsets)
    check_vector_match('da_capo (gains)',
                       self.ofs_and_gains.gains, result.ofs_and_gains.gains)
    check_vector_match('da_capo (map)',
                       self.sky_map, result.sky_map)
```

Defines:
  TestDaCapo.testPreconditioner, never used.
Uses build_test_matrices 76a, check_vector_match 74a, conjugate_gradient 57c, da_capo 66b, and OfsAndGains 44.

Note that we test both conjugate_gradient and da_capo.

We can now check the code using the two preconditioners and no preconditioner at all:

80a ⟨*Test the preconditioners* 79b⟩+≡ (73) ◁79b

```
  def testNoPreconditioner(self):
      self.precondition_check(None)

  def testJacobiPreconditioner(self):
      self.precondition_check(JacobiPreconditioner)

  def testFullPreconditioner(self):
      self.precondition_check(FullPreconditioner)
```

Uses FullPreconditioner 62a and JacobiPreconditioner 63b.

4.1.3. Testing other functions. There are a few other functions we should test. One of them is compute_rms, which we used in Sect. 3.11.2 to estimate the error of the offset/gain estimates produced by Da Capo. Here we generate a number of random samples which follows the "normal" distribution, i.e., a Gaussian with zero mean and RMS equal to one, and we check that compute_rms is close enough to the expected value:

80b ⟨*Test* compute_rms 80b⟩≡ (73)

```
  def testComputeRMS(self):
      chunks = [40000, 70000, 50000]
      samples = np.random.randn(np.sum(chunks))
      check_vector_match('compute_rms',
                         np.repeat(1.0, len(chunks)),
                         compute_rms(samples, chunks),
                         rtol=5e-2)
```

Uses check_vector_match 74a and compute_rms 60.

Another function to test is get_dipole_temperature. We test it both with and without the relativistic frequency-dependent correction:

80c ⟨*Test* get_dipole_temperature 80c⟩≡ (73)

```
  def testDipoleTemperature(self):
      solsysspeed = healpy.ang2vec(1.76560508, 2.97323038) * 370082.2332
      tcmb = 2.72548

      # No quadrupolar correction
      assert np.allclose(get_dipole_temperature(tcmb, solsysspeed, [0, 0, 1]),
                         -0.0006532168171509646)
```

```
    assert np.allclose(get_dipole_temperature(tcmb, solsysspeed, [0, 0, 1], 30e9),
                       -0.0006511369998784711)

    assert np.allclose(get_dipole_temperature(tcmb, solsysspeed, [0, 0, 1], 44e9),
                       -0.0006511328936482864)

    assert np.allclose(get_dipole_temperature(tcmb, solsysspeed, [0, 0, 1], 70e9),
                       -0.0006511213786323892)

    assert np.allclose(get_dipole_temperature(tcmb, solsysspeed, [0, 0, 1], 143e9),
                       -0.0006510659240198485)
```

Uses `get_dipole_temperature` 41a.

§ 4.2. Integration tests

In this section we will build a more complex test, which will exercise all the functionality provided by `index.py` and `calibrate.py`. In other words, we are going to implement an *integration test* that run an end-to-end simulation and check the validity of the results.

Our integration test consists of the following parts:

1. A program which produces a simulated TOD and saves it in FITS files;
2. A set of scripts that instruct `index.py` and `calibrate.py` about how to read the file created in the previous step;
3. A program which loads the results saved by `calibrate.py` and compare them with the expected values.

4.2.1. Creation of simulated timelines. We are going to simulate the observation of the sky made by one detector onboard a spacecraft which observes a substantial part of the sky.

Modelling the sky signal. We use a model of the sky that only contains the CMB (as measured by Planck) and the dipole. The first signal is read from a FITS file, while the dipole is generated on-the-fly using `get_dipole_temperature`; this is the task of a new Python function, `create_dipole_map`:

81 ⟨*Implementation of* `create_dipole_map` 81⟩≡ (83b)

```
def create_dipole_map(nside: int):
    "'Create a Healpix map containing the CMB dipole signal in K_CMB"'
    t_cmb_k = 2.7
    solsys_colat_rad = 1.76560508
    solsysdir_long_rad = 2.97323038
    solsysspeed_m_s = 370082.2332
    solsys_speed_vec_m_s = solsysspeed_m_s * \
        np.array([np.sin(solsys_colat_rad) * np.cos(solsysdir_long_rad),
                  np.sin(solsys_colat_rad) * np.sin(solsysdir_long_rad),
                  np.cos(solsys_colat_rad)])
    return get_dipole_temperature(
        t_cmb_k=t_cmb_k,
        solsys_speed_vec_m_s=solsys_speed_vec_m_s,
        directions=healpy.pix2vec(nside, np.arange(healpy.nside2npix(nside))))
```

Defines:
`create_dipole_map`, used in chunk 84b.
Uses `get_dipole_temperature` 41a.

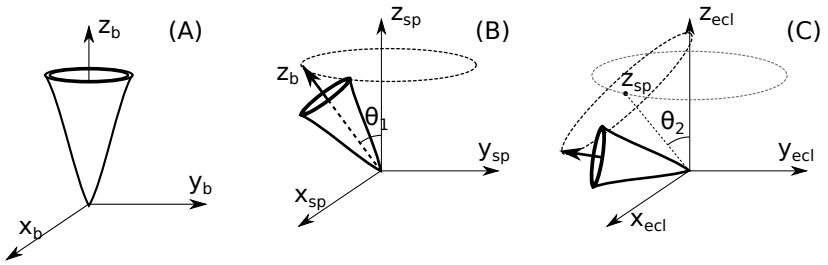


Figure 4.1: *Geometry of the scanning strategy for the long test described in Sect. 4.2.1. The code creates the scanning strategy for the detector considered in the simulation by going through a series of coordinate systems. (A) shows the beam reference frame, with the z_b axis aligned with the main beam of the detector. (B) shows that the z_b axis of the beam reference frame and the z_{sp} axis of the spacecraft are separated by an angle θ_1; the spacecraft is spinning around its z_{sp} axis, so that the main beam of the detector describes circles in the sky. (C) refers to the Ecliptic reference frame, where the z_{sp} axis of the spacecraft is separated by the Celestial North Pole by an angle θ_2. The spacecraft axis rotates around the North Pole, as shown by the gray circle.*

We need to create a TOD containing *uncalibrated* samples, but the CMB map and the dipole map measure the signal using thermodynamic temperatures. Therefore, we need a function which takes a TOD in Kelvin and applies Eq. 1.1 (page 1) to the temperatures: this is the purpose of `decalibrate_tod`:

82 ⟨*Implementation of* `decalibrate_tod` 82⟩≡ (83b)

```
def decalibrate_tod(tod, offsets, gains, samples_per_ofsp, samples_per_gainp):
    start_idx = 0
    cur_ofsp = 0
    cur_gainp = 0
    samples_in_gainp = 0
    while start_idx < len(tod):
        tod[start_idx:start_idx + samples_per_ofsp] = \
            tod[start_idx:start_idx + samples_per_ofsp] * gains[cur_gainp] + \
            offsets[cur_ofsp]

        start_idx += samples_per_ofsp
        cur_ofsp += 1
        samples_in_gainp += samples_per_ofsp
        if samples_in_gainp >= samples_per_gainp:
            cur_gainp += 1
            samples_in_gainp = 0
```

Defines:
`decalibrate_tod`, used in chunk 85c.

Creating the pointing information. The scanning strategy used by our imaginary experiment is shown in Fig. 4.1. The spacecraft spins around its z axis, and which in turns performs a rotation on the sky. This allows to scan the sky in large circles, which is the best solution if one plans to use the dipole to calibrate the data. The code uses quaternions to model rotations; the four quaternions `quat1`, `quat2`, `quat3`, and `quat4` correspond to the following transformations:

1. `quat1` rotates the reference frame of the beam into the reference frame of the spacecraft (from panel A to panel B of Fig. 4.1); this is achieved as a rotation around the x axis;
2. `quat2` implements the rotation around axis z_{sp} (panel B), with the number of rotations per minute specified in the parameter `rpm1`;
3. `quat3` rotates the spacecraft's reference frame into the Ecliptic reference frame (from panel B to panel C);
4. `quat4` implements the rotation around axis z_{ecl}, with the number of rotations per minute specified in the parameter `rpm1`.

Here is the code; the `times` parameter is an array containing the times of each sample in the output, and it is measured in seconds.

83a ⟨*Implementation of* `generate_pointings` 83a⟩≡ (83b)

```
def generate_pointings(rpm1: float, rpm2: float, times: Any, num_of_samples:
                       int):
    quat1 = np.tile(
        qa.rotation([1., 0, 0], np.pi / 2.), num_of_samples).reshape(-1, 4)
    quat2 = qa.rotation([0, 0, 1.], 2. * np.pi * rpm1 / 60.0 * times)
    quat3 = np.tile(
        qa.rotation([1., 0, 0], np.pi / 3.), num_of_samples).reshape(-1, 4)
    quat4 = qa.rotation([0, 0, 1.], 2. * np.pi * rpm2 / 60.0 * times)

    fullquat = qa.mult(quat4, qa.mult(quat3, qa.mult(quat2, quat1)))
    vectors = qa.rotate(fullquat, np.array([0., 0., 1.]))
    return healpy.vec2ang(vectors[:, 0:3])
```

Defines:
`generate_pointings`, used in chunk 85a.

Running the simulation. After having implemented the helper functions discussed in the previous paragraphs, we now turn to the program that will actually create the TOD. The executable requires the output path where to write the simulated TOD to be specified on the command line; as usual, we use the `click` library for this:

83b ⟨*create-test-files.py* 83b⟩≡

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-
"'Create a set of FITS files to be used as test input for the DaCapo
calibration codes.
"'

import logging as log
import os.path
from typing import Any

from astropy.io import fits
import numpy as np
import healpy
import click
from calibrate import get_dipole_temperature
import quaternionarray as qa

⟨Implementation of create_dipole_map 81⟩
⟨Implementation of write_simulated_tod 86b⟩
⟨Implementation of generate_pointings 83a⟩
⟨Implementation of decalibrate_tod 82⟩

@click.command()
@click.argument('outdir',
                type=click.Path(exists=True,
                                dir_okay=True,
                                file_okay=False))
def main(outdir: str):
    log.basicConfig(level=log.INFO,
                    format='[%(asctime)s %(levelname)s] %(message)s')

    ⟨Set up the parameters for the simulation 84a⟩
    ⟨Create the model map of the sky signal 84b⟩
    ⟨Create the pointing information and save the hit map 85a⟩
```

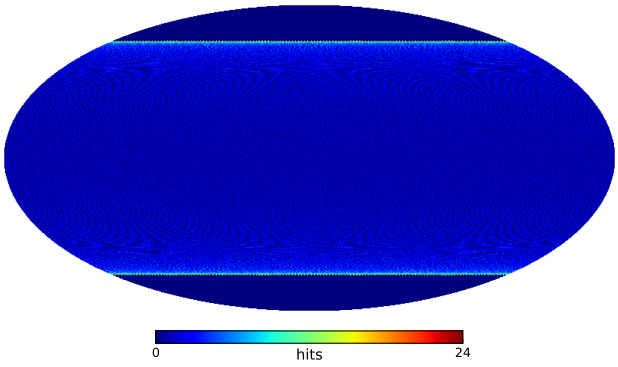


Figure 4.2: *Hit map of the pointings generated by the program* `create-test-files.py`.

⟨*Build the calibrated TOD* 85b⟩
⟨*Create a set of offsets/gains and decalibrate the TOD* 85c⟩
⟨*Save the decalibrated TOD to disk* 85d⟩

⟨*Create a parameter file for* `index.py` 85e⟩
⟨*Create a set of parameter files for* `calibrate.py` 86a⟩

```
if __name__ == '__main__':
    main()
```

Uses `get_dipole_temperature` 41a.

The steps in the `main` function should be straightforward to understand from their description. We now implement them one by one.

First, we define a few parameters to be used in the simulation, namely (1) the number of samples per each offset period, (2) the number of samples per each gain period, and (3) the number of samples in the whole TOD. The following numbers are arbitrary and can be changed according to one's own taste:

84a ⟨*Set up the parameters for the simulation* 84a⟩≡ (83b)

```
samples_per_ofsp = 5000  # Number of samples
samples_per_gainp = samples_per_ofsp * 10
num_of_samples = samples_per_gainp * 20

assert num_of_samples % samples_per_ofsp == 0
assert num_of_samples % samples_per_gainp == 0
assert samples_per_gainp % samples_per_ofsp == 0
```

We turn now to the creation of the sky model. We use a CMB map loaded from a FITS file and a generated dipole map (using `create_dipole_map`); we compute the peak-to-peak amplitude of the dipole and store it in `dipole_amplitude`, because we will need it later:

84b ⟨*Create the model map of the sky signal* 84b⟩≡ (83b)

```
galaxy_map = healpy.read_map(
    os.path.join('maps', 'COM_CMB_IQU-commander_256_ecliptic.fits'),
    verbose=False)
nside = healpy.npix2nside(len(galaxy_map))

dipole_map = create_dipole_map(nside=nside)
dipole_amplitude = np.max(dipole_map) - np.min(dipole_map)
```

Uses `create_dipole_map` 81.

Now we create the pointing information. The `times` variable is an array containing the time of each sample in the output simulation; as above, the value used for the angular speed of the two

transformations discussed in Sect. 4.2.1 is arbitrary and can be changed at one's wishes:

85a ⟨*Create the pointing information and save the hit map* 85a⟩≡ (83b)

```
times = np.linspace(0, 86400., num_of_samples)
theta, phi = generate_pointings(rpm1=1.,
                                rpm2=1. / (24. * 60.),
                                times=times,
                                num_of_samples=num_of_samples)
pixidx = healpy.ang2pix(nside, theta, phi)
healpy.write_map(os.path.join(outdir, 'long_test_hits.fits.gz'),
                 np.bincount(pixidx, minlength=healpy.nside2npix(nside)),
                 overwrite=True)
```

Uses generate_pointings 83a.

We save the hit count map in a FITS file, because it is useful for debugging purposes. An image of the map is shown in Fig. 4.2.

The calibrated TOD contains a white noise component that is scaled according to the dipole's peak-to-peak amplitude: in this way, we are sure that the S/N ratio of the calibrator is always good enough to permit a good calibration:

85b ⟨*Build the calibrated TOD* 85b⟩≡ (83b)

```
tod = (galaxy_map[pixidx] + dipole_map[pixidx] + np.random.randn() *
       dipole_amplitude * 1e-5)
```

We now turn tod into a decalibrated sequence of voltages:

85c ⟨*Create a set of offsets/gains and decalibrate the TOD* 85c⟩≡ (83b)

```
offsets = np.random.randn(num_of_samples //
                          samples_per_ofsp) * np.sqrt(np.var(dipole_map))
gains = (np.random.randn(num_of_samples // samples_per_gainp) + 50.0)
decalibrate_tod(tod, offsets, gains, samples_per_ofsp, samples_per_gainp)
```

Uses decalibrate_tod 82.

We are now ready to save the TOD to disk:

85d ⟨*Save the decalibrated TOD to disk* 85d⟩≡ (83b)

```
tod_file_name = 'long_test_tod.fits'
write_simulated_tod(file_name=os.path.join(outdir, tod_file_name),
                    time=times,
                    theta=theta,
                    phi=phi,
                    tod=tod,
                    offsets=offsets,
                    gains=gains,
                    samples_per_ofsp=samples_per_ofsp,
                    samples_per_gainp=samples_per_gainp)
log.info('file "%s" written', tod_file_name)
```

Uses write_simulated_tod 86b.

Having the TOD is however not enough to run an integration test. We need to have parameter files for index.py and calibrate.py, and we need them to be consistent with the assumptions we have used in creating the TOD. Therefore, the code now creates the parameter files from scratch. Let's start from the file for index.py:

85e ⟨*Create a parameter file for* index.py 85e⟩≡ (83b)

```
index_file_name = os.path.join(outdir, 'long_test_index.fits')
with open(os.path.join(outdir, 'long_test_index.ini'), 'wt') as f:
    f.write("'[input_files]
path = {path}
mask = {tod_file_name}
hdu = 1
column = TIME

[periods]
```

```
length = {samples_per_ofsp}

[output_file]
file_name = {index_file_name}
"'.format(path=outdir,
          tod_file_name=tod_file_name,
          index_file_name=index_file_name,
          samples_per_ofsp=times[samples_per_ofsp]))
```

Regarding the parameter file for `calibrate.py`, there are many possible choices for the kind of analysis done by the code. We choose to exercise all the possibilities provided by `calibrate.py` in terms of preconditioners, so we generate a set of calibration files, each with a different setting for `pcond`:

86a ⟨*Create a set of parameter files for* `calibrate.py` 86a⟩≡ (83b)

```
for pcond in ('none', 'jacobi', 'full'):
    ini_file_name = os.path.join(
        outdir, 'long_test_calibrate_{0}.ini'.format(pcond))
    output_file_name = os.path.join(
        outdir, 'long_test_results_{0}.fits'.format(pcond))

    with open(ini_file_name, 'wt') as f:
        f.write("'[input_files]
index_file = {index_file_name}
signal_hdu = 1
signal_column = SIGNAL
pointing_hdu = 1
pointing_columns = THETA, PHI

[dacapo]
t_cmb_K = 2.72548
solsysdir_ecl_colat_rad = 1.7656131194951572
solsysdir_ecl_long_rad = 2.995889600573578
solsysspeed_m_s = 370082.2332
nside = {nside}
periods_per_cal_constant = {gainp_per_ofsp}
cg_stop_value = 1e-9
cg_max_iterations = 100
dacapo_stop_value = 1e-9
dacapo_max_iterations = 20
pcond = {pcond}

[output]
file_name = {output_file_name}
save_map = yes
save_convergence = yes
comment = "Long duration test"
"'.format(index_file_name=index_file_name,
          gainp_per_ofsp=samples_per_gainp // samples_per_ofsp,
          output_file_name=output_file_name,
          pcond=pcond,
          nside=nside))

log.info('file "%s" written', ini_file_name)
```

Writing the TOD on disk. We have still to implement `write_simulated_tod`; we save the input values of the gains and the offset in the TOD, so that it will be easy to check the validity of the estimates produced by `calibrate.py` at the end of the integration test:

86b ⟨*Implementation of* `write_simulated_tod` 86b⟩≡ (83b)

```
def write_simulated_tod(file_name: str, time: Any, theta: Any, phi: Any, tod: Any,
              offsets: Any, gains, samples_per_ofsp: int, samples_per_gainp:
              int):
    hdu1 = fits.BinTableHDU.from_columns([
        fits.Column(name='TIME', array=time, format='J', unit='s'),
        fits.Column(name='THETA', array=theta, format='E', unit='rad'),
        fits.Column(name='PHI', array=phi, format='E', unit='rad'),
        fits.Column(name='SIGNAL', array=tod, format='D', unit='V')])

    hdu2 = fits.BinTableHDU.from_columns([
        fits.Column(name='OFFSET', array=offsets, format='D', unit='V')],
                                       name='OFFSETS')
    hdu2.header['OFSSAMP'] = (samples_per_ofsp, 'Samples per offset period')

    hdu3 = fits.BinTableHDU.from_columns([
        fits.Column(name='GAIN', array=gains, format='D', unit='V/K')],
                                       name='GAINS')
    hdu3.header['GAINSAMP'] = (samples_per_gainp, 'Samples per gain period')
    hdu3.header['GAINSOFS'] = (samples_per_gainp // samples_per_ofsp,
                               'Offset periods per gain period')

    hdulist = fits.HDUList([fits.PrimaryHDU(), hdu1, hdu2, hdu3])
    hdulist.writeto(file_name, clobber=True)
```

Defines:
write_simulated_tod, used in chunk 85d.

4.2.2. Verifying the results. Once we have the simulated TOD and the parameter files for `index.py` and `calibrate.py`, we are supposed to run the programs and examine the results. Therefore, we need some tool to verify that the results produced by `calibrate.py` match the expectations; in our case, we want to compare the gains estimated by `calibrate.py` with the gains used as input by `create-test-files.py`. This is easy to do, as `create-test-files.py` saves the input gains in a separate tabular HDU of the TOD FILE, named `GAINS`, and this is the same name used by `calibrate.py` for the HDU in the output file which contains the estimated gains. The following script loads the `GAIN` column from the `GAINS` HDU in two FITS files and compare them, returning nonzero if the two gains have a significant mismatch:

87 ⟨*check-gains.py* 87⟩≡

```
#!/usr/bin/env python3
# -*- encoding: utf-8 -*-
"'Usage: {basename} FILE1 FILE2

Check the consistency between the gains in FILE1 and FILE2. The two
files can have been created either using create-test-files.py or
calibrate.py.
"'

import os.path
import sys
from astropy.io import fits
import numpy as np


def main():
    if len(sys.argv) != 3:
        print(__doc__.format(basename=os.path.basename(sys.argv[0])))
        sys.exit(1)

    file1_name, file2_name = sys.argv[1:3]
```

```
    with fits.open(file1_name) as f:
        file1_gains = f['GAINS'].data.field('GAIN')

    with fits.open(file2_name) as f:
        file2_gains = f['GAINS'].data.field('GAIN')

    assert np.allclose(file1_gains, file2_gains,
                       rtol=2e-2), 'Gains do not match'


if __name__ == '__main__':
    main()
```

The `Makefile` included in the source code implements a command, `fullcheck`, which calls in turn `create-test-files.py`, `index.py`, `calibrate.py`, and `check-gains.py`, and verifies the correct execution of the code for all the preconditioners implemented by `calibrate.py`. It is ran automatically by the Docker container presented in Sect. 1.3.2.

Appendix A

Index of symbols

Here we provide a list of the symbols used in the code. Each reference is of the form nL, where n is the number of the page and L a letter specifying the code chunk within that page starting from "a". Underlined references point to the definition of the symbol.

apply_A: 56a, 57c, 73, 78e
apply_f: 48a, 49a, 56a, 68a, 73, 77c
apply_ft: 49b, 50, 56a, 56b, 73, 77d
apply_ptilde: 53a, 53b, 55a, 73, 78a
apply_ptildet: 54b, 55a, 68a, 73, 78b
apply_ptildet_locally: 53c, 54a, 54b
apply_z: 55a, 56a, 56b, 73, 78c
assign_files_to_processes: 37a, 40b
BOLTZMANN_K_MKS: 41a
build_test_matrices: 76a, 77a, 79b
CalibrateConfiguration: 28, 29, 31d, 38b, 40a
check_vector_match: 74a, 77c, 77d, 77e, 78a, 78b, 78c, 78d, 78e, 79a, 79b, 80b
compute_diagm: 52c, 55a, 68a, 73, 77e
compute_diagm_locally: 52a, 52b, 52c
compute_map_corr: 66b, 68a, 73, 79a
compute_rms: 60, 62a, 63b, 73, 80b
compute_v: 56b, 57c, 73, 78d
conjugate_gradient: 57c, 66b, 73, 79b
create_dipole_map: 81, 84b
da_capo: 26b, 66b, 73, 79b
decalibrate_tod: 82, 85c
flag_mask: 13d, 20b, 25, 39b
FlagAction: 13b, 13d, 17b, 23a, 23b
Flagging: 13c, 13d, 17b, 21c, 23b
FlagType: 13a, 13d, 17b, 23a, 23b
ftnroutines.apply_f: 48a, 49a
ftnroutines.apply_ft: 49b, 50
ftnroutines.apply_ptilde: 53a, 53b
ftnroutines.apply_ptildet_locally: 53c, 54a
ftnroutines.clean_binned_map: 55a, 55b
ftnroutines.compute_diagm_locally: 52a, 52b
FullPreconditioner: 62a, 65a, 73, 80a
gather_arrays: 68b, 68c
generate_pointings: 83a, 85a

get_dipole_temperature: 41a, 42a, 80c, 81, 83b
guess_gains: 65c, 66b
index_main: 11, 18b
IndexConfiguration: 16, 17a, 24b
IndexFile: 21c, 24b, 25, 32b, 38b, 40a
int_or_str: 17b, 17c, 18a, 23b, 25, 30a
JacobiPreconditioner: 63b, 65a, 73, 80a
load_subrange: 38b, 40a
load_tod: 40a, 40b
MonopoleAndDipole: 42b, 54c, 55a, 56a, 56b, 57c, 62a, 63b, 66b, 73, 74b
mpi_abs_max: 66b, 67
ofs_and_gains_with_same_lengths: 45b, 57c, 62a, 63b
OfsAndGains: 44, 45b, 49a, 50, 52b, 52c, 53b, 54a, 54b, 55a, 56a, 56b, 57c, 62a, 63b, 66b, 68a, 73, 75b, 78e, 79b
OfsAndGains.calc_ofsp_per_gainp: 44, 45a, 62a
PCOND_DICT: 65a, 65b
PLANCK_H_MKS: 41a
Profiler: 27b, 66b
read_calibrate_conf_file: 29, 32a
read_column: 20b, 20c
read_index_conf_file: 17a, 19b
SPEED_OF_LIGHT_M_S: 41a
split: 30a, 33b, 34, 40d
split_into_n: 33a, 33b, 34
split_into_periods: 15, 21a
sum_local_results: 51, 52c, 54b
sum_subranges: 34, 35a
TestDaCapo.setUp: 77b
TestDaCapo.testApplyA: 78e
TestDaCapo.testApplyF: 77c
TestDaCapo.testApplyFT: 77d
TestDaCapo.testApplyPtilde: 78a
TestDaCapo.testApplyPtildet: 78b
TestDaCapo.testApplyZ: 78c
TestDaCapo.testComputeDiagM: 77e
TestDaCapo.testComputeMapCorr: 79a
TestDaCapo.testComputeV: 78d
TestDaCapo.testPreconditioner: 79b
tic: 27b
toc: 27b, 66b
TOD: 35b, 38b, 40a, 40d
TODFileInfo: 18b, 20a, 20b, 23b, 24b, 25, 37a
TODSubrange: 36, 37a, 37b, 38a, 38b, 40a
write_output: 18b, 24b
write_simulated_tod: 85d, 86b

Bibliography

C. Burigana, M. Malaspina, N. Mandolesi, L. Danse, D. Maino, M. Bersanelli, and M. Maltoni. A preliminary study on destriping techniques of planck/lfi measurements versus observational strategy. *ArXiv Astrophysics e-prints*, June 1999.

E. Keihänen, H. Kurki-Suonio, T. Poutanen, D. Maino, and C. Burigana. A maximum likelihood approach to the destriping technique. *A&A*, 428:287–298, December 2004. doi: 10.1051/0004-6361:200400060.

E. Keihänen, H. Kurki-Suonio, and T. Poutanen. Madam – a map-making method for cmb experiments. *MNRAS*, 360:390–400, June 2005. doi: 10.1111/j.1365-2966.2005.09055.x.

H. Kurki-Suonio, E. Keihänen, R. Keskitalo, T. Poutanen, A.-S. Sirviö, D. Maino, and C. Burigana. Destriping cmb temperature and polarization maps. *A&A*, 506:1511–1539, November 2009. doi: 10.1051/0004-6361/200912361.

Planck Collaboration. Planck 2015 results. v. LFI calibration. *Astronomy & Astrophysics*, dec 2015. doi: 10.1051/0004-6361/201526632. URL http://dx.doi.org/10.1051/0004-6361/201526632.

M. Quartin and A. Notari. Improving Planck calibration by including frequency-dependent relativistic corrections. *JCAP*, 9:050, September 2015. doi: 10.1088/1475-7516/2015/09/050.

John C. Strikwerda. *Finite Difference Schemes and Partial Differential Equations, Second Edition*. Society for Industrial & Applied Mathematics (SIAM), jan 2004. doi: 10.1137/1.9780898717938. URL http://dx.doi.org/10.1137/1.9780898717938.