Calculation of ϕ_D and ϕ_{sky}

Maurizio Tomasi

August 2014

Abstract

In this document I provide the implementation of two programs to calculate the value of ϕ_D and ϕ_{sky} as a function of time for a LFI-like survey. An overview of the theory is provided in the first part of the report. Then, the full source code of the two programs is presented and commented in detail. The two programs are written in Pascal and can be compiled using the Free Pascal compiler.

Contents

§ 1. Introduction

This document provides the complete implementation of two stand-alone programs to compute the impact of beam convolution in the estimation of the calibration constant and of the sky temperature measured by the Planck/LFI radiometers. Such analysis is important to compare the approximation of a Dirac delta beam in the calibration (the so-called "pencil-beam" approximation) with the exploitation the knowledge of the beam response over the full 4π solid angle (the so-called "4π calibration").

Both the pencil-beam approximation and the 4π calibration have been used in important full-sky CMB experiments. The WMAP and HFI teams have always used the pencil-beam approximation, as well as the LFI team for the 2013 data release (Collaboration, 2013). In 2014, the LFI team published the results of the analysis of the full Planck dataset using a new approach where the knowledge of the 4π beam has been taken into account during the calibration of the Time Ordered Data (TOD, a time series of voltages) into Time Ordered Information (TOI, a time series of thermodynamic temperatures).

The purpose of this note is to define two new quantities, ϕ_D and ϕ_{sky}, which quantify the impact of the two approaches to calibration on the TOIs, and to provide the implementation of two command-line programs which allow to measure such quantities out of the TOIs.

1.1. Definition of ϕ_{sky} and ϕ_D. The definition of the quantity ϕ_D, as well as an explanation of its meaning, is provided in Collaboration (2013). Consider the output of a differential radiometer:

$$V_{\text{out}}(t) = G(t) \times (B * (T_{\text{sky}} + D)) + M, \tag{1}$$

where $D(\theta, \varphi)$ is the temperature of the Doppler CMB dipole used for the calibration of the instrument, $B(\theta, \varphi)$ is the beam response along some direction, T_{sky} is the temperature of the sky (with the exception of the dipole), and M is an instrumental offset. The quantity ϕ_D is defined as

$$\phi_D = \frac{\partial_t \big(B_s * D\big)}{\partial_t \big(B_m * D\big)}. \tag{2}$$

and it is used to relate the estimation $\tilde{G}$ of the true calibration constant G (in K/V), since the following relation holds:

$$\tilde{G} = G(1 - f_{\text{sl}})(1 + \phi_D) \tag{3}$$

where f_{sl} is the fraction of the beam which is not within the main lobe. Eq. (3) only holds under the assumption of a pencil-beam approximation, because if a 4π beam calibration is employed (and B is known without error), then

$$\tilde{G} = G. \tag{4}$$

Let's now assume a perfect calibration ($\tilde{G} = G$). Then the measured temperature $\tilde{T}_{\text{sky}}(t)$ at some time t is

$$\tilde{T}_{\text{sky}}(t) = \big(B * T_{\text{sky}}\big)(t) + M = \big(B_m * T_{\text{sky}}\big)(t) + \big(B_s * T_{\text{sky}}\big)(t) + M, \tag{5}$$

where $B = B_m + B_s$ is the 4π beam divided into a "main" and a "side" part (the Planck/LFI convention is that $B_m(\vec{x}) = 0$ for any direction $\vec{x}$ farther than 5° from the beam axis), and $B * T_{\text{sky}}$ is the TOD of the sky temperature convolved with the beam (the beam moves with time, so this frame of reference is continuously changing). If we neglect angular scales smaller than the width of the main beam, then

$$B_m * T_{\text{sky}} \approx (1 - f_{\text{sl}}) T_{\text{sky}} \tag{6}$$

along any direction $\vec{x}$. Therefore, Eq. (5) becomes

$$\tilde{T}_{\text{sky}}(t) = (1 - f_{\text{sl}}) T_{\text{sky}}(t) + \big(B_s * T_{\text{sky}}\big)(t) + M. \tag{7}$$

Solving for $T_{\text{sky}}(t)$, we get

$$T_{\text{sky}}(t) = \frac{\tilde{T}_{\text{sky}}(t) - \big(B_s * T_{\text{sky}}\big)(t) - M}{1 - f_{\text{sl}}}. \tag{8}$$

Since we are interested in expressing Eq. (8) in the form $T_{\text{sky}} = \alpha \tilde{T}_{\text{sky}} + T_0$, we define the new quantity

$$\phi_{\text{sky}}(t) \equiv \frac{\big(B_s * T_{\text{sky}}\big)(t)}{\tilde{T}_{\text{sky}}(t)} = \frac{\big(B_s * T_{\text{sky}}\big)(t)}{\big(B * T_{\text{sky}}\big)(t) + M}, \tag{9}$$

so that

$$T_{\text{sky}}(t) = \frac{1 - \phi_{\text{sky}}}{1 - f_{\text{sl}}} \tilde{T}_{\text{sky}}(t) + T_0, \tag{10}$$

and $\alpha = 1 - \phi_{\text{sky}}$. The quantity ϕ_{sky} defined by Eq. (9) quantifies the impact of sidelobes in the measurement of the sky temperature T_{sky}, and it is the main subject of this note. Since it is difficult to estimate the value of M, we will use the first equality with the $\tilde{T}_{\text{sky}}$ term in the following.

1.2. Computational issues. There are two major problems in computing ϕ_{sky} using Eq. (9):

1. Computations must be done in time-domain, so that a lot of data must be processed. (Planck/LFI pointing data are 1 GB/yr for 30 GHz radiometers and 3 GB/yr for 70 GHz radiometers: considering all the 22 radiometers, the sum is 53 GB/yr.)
2. Eq. (9) requires the computation of a convolution over the 4π sphere. This is usually expensive to compute numerically.

The solution to the first problem is to massively parallelize the code. Fortunately, this is not difficult, as the algorithm is "embarassingly parallel": if $t_1 \neq t_2$, then $\phi_D(t_1)$ and $\phi_D(t_2)$ do not depend on each other and can be computed by separated processes (the same applies to ϕ_{sky}). We use the Message Passing Interface (MPI) to parallelize the code, assuming a distributed memory environment. About the second problem, there are two solutions:

1. Use the fast convolution algorithm described by Wandelt and Górski (2001), which greatly reduces the computation times by pre-computing a mathematical object, called a *ringset*, which is then used to estimate the value of the convolution at any point;
2. If one of the two terms of the convolution is the dipole, an approach that is even faster than ringsets is to use the so-called *convolution matrices*, first used by Collaboration (2013).

Several inputs are required to compute ϕ_D and ϕ_{sky}:

1. Pointing information. These can either be provided as FITS files or as files compressed using `squeezer`. (The second option is much faster, because compressed files are ~ 10 times smaller than FITS files and therefore reduce the time used for I/O.)
2. Temperatures (needed by ϕ_{sky} for the term $\tilde{T}_{\text{sky}}$ in Eq. 9). Like pointing informations, these can either be saved in FITS files or in compressed files produced by `squeezer`.
3. The calculation of ϕ_{sky} requires FITS files containing ringsets, i.e., a set of numbers which can be used to numerically estimate the convolution between a beam model and a sky signal;
4. The calculation of ϕ_D uses convolution matrices instead of ringsets.
5. Since ϕ_D requires to model the dipole D, which has a component due to the orbital motion of the spacecraft around the Sun, to apply Eq. (2) the program needs a file containing the speed of the spacecraft as a function of time.

Moreover, the user is expected to provide the parameters required for the computation in a *parameter file*, whose syntax is similar to the INI files used by old versions of Windows[1].

[1] See `http://en.wikipedia.org/wiki/INI_file`.

1.3. How numerical codes are implemented. In this document we will provide the complete source code of two programs, `phisky` and `phid`, which can be used to estimate the value of ϕ_{sky} and ϕ_D for the Planck/LFI radiometers. We use a programming technique called *literate programming*, where both the program and the documentation (this report) are written at the same time *in the same file*. We use the NoWeb system[2] by Norman Ramsey to implement this idea: two programs, called *tangler* and *weaver*, are then used to extract the source code to compile (`notangle`) and the LaTeX file used to produce a standalone document (`noweave`).

We chose not to implement both calculations in the same program, but to implement two separate programs. This leads inevitably to some code duplication, which however is mitigated by the fact that we are using literate programming techniques. The advantage of having two programs lies mainly in the fact that computing ϕ_D is much faster than computing ϕ_{sky}, and yet it is the first term which usually dominates (i.e., $\phi_D \gg \phi_{\text{sky}}$). So it is likely that in a variety of situations it is not required to compute both of them.

The two programs, `phisky` and `phid`, are written using a dialect of Pascal implemented by the Free Pascal[3] compiler (version 2.6.x or above). Here are the skeleton implementation of `phid`; we will fill all the details later:

4 ⟨*phid.pas* 4⟩≡

```
{ -*- mode: delphi -*- }
program CalcPhiD;

{$mode objfpc}{$h+}
uses Classes, SysUtils, INIFiles, DataTypes, Squeezer, Cfitsio, Healpix, Mpi,
     RotMatrix, SatelliteVelocities, ConvolvedParams;

{$linklib c} { Required by OpenMPI/MPICH }

const
    ProgramName = 'phid';
    ⟨General-purpose constants 31c⟩

type
    ⟨Basic type definitions (shared between phisky and phid) 25a⟩
    ⟨Type definitions used by phid 7a⟩

⟨Basic functions 32c⟩
⟨General functions (shared between phisky and phid) 10b⟩
⟨High-level functions for the phid program 16a⟩

var
    ⟨Variables used by phid and phisky in the main loop 13b⟩
    ⟨Variables used in the main loop of phid 10d⟩

begin
    Mpi.Init;
    try
        try
            ⟨Implementation of phid 6a⟩
        finally
            Mpi.Finalize;
        end;
    except
        on E : Exception do WriteLn('Error: ' + E.message);
    end;
end.
```

[2] `http://www.cs.tufts.edu/~nr/noweb/`.
[3] `http://www.freepascal.org/`.

(The units `Classes`, `SysUtils`, and `INIFiles` are part of the Free Pascal Standard Library.)
And here is the skeleton of `phisky`; it essentially shares the same structure:

5 ⟨*phisky.pas* 5⟩≡

```
{ -*- mode: delphi -*- }
program CalcPhiSky;

{$mode objfpc}{$h+}
uses Classes, SysUtils, INIFiles, DataTypes, Ringsets, Squeezer, Cfitsio, Healpix, Mpi;

{$linklib c} { Required by OpenMPI/MPICH }

const
    ProgramName = 'phisky';
    ⟨General-purpose constants 31c⟩

type
    ⟨Basic type definitions (shared between phisky and phid) 25a⟩
    ⟨Type definitions used by phisky 7b⟩

⟨Basic functions 32c⟩
⟨General functions (shared between phisky and phid) 10b⟩
⟨High-level functions for the phisky program 17⟩

var
    ⟨Variables used by phid and phisky in the main loop 13b⟩
    ⟨Variables used in the main loop of phisky 10e⟩

begin
    Mpi.Init;
    try
        try
            ⟨Implementation of phisky 6b⟩
        finally
            Mpi.Finalize;
        end;
    except
        on E : Exception do WriteLn('Error: ' + E.message);
    end;
end.
```

In the next sections we will implement each of the placeholders indicated in the code with ⟨. . .⟩. We will sometimes used the procedure `Log`, which prints some status message on the screen alongside with a timestamp. Knowing how this procedure works is not important for understanding the logic of the two programs, so we moved its implementation and description in Appendix A

§ 2. Numerical estimation of ϕ_{sky} and ϕ_D

2.1. The overall structure of the program. The basic steps followed by the two programs, `phid` and `phisky`, are quite the same. They can be summed up as follows:

1. Read a set of convolution matrices (`phid`) or ringsets (`phisky`;
2. Read the pointing information for each given OD (these include: the time, the direction θ, φ of the beam axis in the sky, and the orientation ψ around the beam axis); the program `phisky` must also read TOD containing the temperatures $\tilde{T}_{\mathrm{sky}}(t)$;
3. Apply Eq. (2) or Eq. (9) for each pointing sample and orientation to compute $\phi_D/\phi_{\mathrm{sky}}$.
4. Save the values of $\phi_D/\phi_{\mathrm{sky}}$ as a TOD (this is optional, as the files are going to be huge).

5. As the user is not likely to want all the TODs saved to disk (because of their size), is useful to implement a few techniques to reduce the dimensionality of the TODs. This step produces so-called *reduced TODs* of the quantities ϕ_D/ϕ_{sky}.

6. Save the reduced TODs.

The analysis of the program's results is usually carried using the reduced TODs mentioned in step 5. In this case the code compresses the TODs so that they can be saved in files of acceptable size; see Sect. 2.4 for further details about this.

We can now implement the ⟨*Main program*⟩ placeholders seen in the code above. For `phid` the main program will follow these steps:

6a ⟨*Implementation of* `phid` 6a⟩≡ (4)
⟨*Check program options* 10a⟩
⟨*Subdivide the pointing files among the MPI processes* 14⟩
⟨*Read the* $\mathcal{M}$ *matrices and the satellite velocities* 12b⟩
⟨*Initialize the structures used to compress the TODs* 20b⟩
⟨*Apply Eq.* (2) *to the data in each pointing/temperature file* 21⟩
⟨*Compress the TODs and produce reduced TODs* 31a⟩
⟨*Save the reduced TODs* 32a⟩

The `phisky` program differs slightly, as its set of input files is somewhat different. However, a number of steps are exactly the same, so we are going to reuse their implementation:

6b ⟨*Implementation of* `phisky` 6b⟩≡ (5)
⟨*Check program options* 10a⟩
⟨*Subdivide the pointing files among the MPI processes* 14⟩
⟨*Read the ringsets* 12a⟩
⟨*Initialize the structures used to compress the TODs* 20b⟩
⟨*Apply Eq.* (9) *to the data in each pointing/temperature file* 22a⟩
⟨*Compress the TODs and produce reduced TODs* 31a⟩
⟨*Save the reduced TODs* 32a⟩

2.2. Reading the configuration file. We begin the discussion of the program source code by describing how the parameters of the computation are stored in memory. (Every other piece of the software is going to use these parameters, so it is good to discuss them first.)

How parameters are stored in memory. The most straightforward approach to keep the user's settings in memory would be to use many scattered global variables, but we prefer to keep everything within one structure, in order to pass such settings to procedures and functions more easily. The `phid` program keeps its configuration in the `TPhiDConfiguration` structure:

7a ⟨*Type definitions used by* `phid` 7a⟩≡ (4) 11a ▷

```
TPhiDConfiguration = record
    PointingFileNames : TStringArray;
    SlMatrixFileNames : TStringArray;
    MbMatrixFileNames : TStringArray;
    SatelliteVelocityFileName : String;
    DipoleParams : TDipole;

    Quantiles : TPercentageArray;
    QuantileTableFileName : String;

    Nside : Uint16;
    OutputMapFileName : String;

    case SaveTods : Boolean of
    True: (TodFilePath : ShortString); { We need a ShortString here! }
end;
```

Defines:
`TPhiDConfiguration`, used in chunks 10d, 36, and 38b.
Uses `TPercentageArray` 25a and `TStringArray` 25b.

(The type `TPercentageArray` is defined later.)
The `phisky` program uses a different structure:

7b ⟨*Type definitions used by* `phisky` 7b⟩≡ (5) 11b ▷

```
TPhiSkyConfiguration = record
    MbRingsetFileNames : TStringArray;
    SlRingsetFileNames : TStringArray;
    PointingFileNames : TStringArray;
    TemperatureFileNames : TStringArray;

    QualityFlagMask : UInt32;

    InterpolationOrder : Byte;
    Quantiles : TPercentageArray;
    QuantileTableFileName : String;

    Nside : Uint16;
    OutputMapFileName : String;

    case SaveTods : Boolean of
    True: (TodFilePath : ShortString); { We need a ShortString here! }
end;
```

Defines:
`TPhiSkyConfiguration`, used in chunks 10e, 37, and 39a.
Uses `TPercentageArray` 25a and `TStringArray` 25b.

A few notes about the parameters defined in the two structures:

1. The pair of structure members `MbMatrixFileNames`/`SlMatrixFileNames` (used by `phid`) and `MbRingsetFileNames`/`SlRingsetFileNames` (used by `phisky`) are arrays of strings. They

contain the names of the convolution matrices/ringsets that must be used in the computation of the convolutions between the beam and the dipole/sky signal in Eq. (2) and Eq. (9). More than one matrix/ringset is allowed, as T_{sky} is typically the sum of many contributions (e.g., the Galactic emission plus the CMB) and in the Planck/LFI collaboration it is customary[4] to provide beam models where B_{sl} is split into two parts: the "near-lobe" part and the "far-lobe" part.

2. The pointing files are listed in the `PointingFileNames` variable. At the time of writing, Planck/LFI records them into one file per operating day (OD).

3. The $\tilde{T}_{\text{sky}}$ files ("temperature") used by `phisky` are listed in the `TemperatureFileNames`. There is a one-to-one correspondence between each of these files and the pointing files.

4. We have mentioned in Sect. 2.1 that we need to reduce the dimensionality of the ϕ_D/ϕ_{sky} TODs. Since the input data is separated into many pointing files, one possible approach is to compute all the ϕ_D/ϕ_{sky} values from each pointing file, and then save only a bunch of statistical quantities per file. Such quantities should provide a fairly sufficient description of the distribution of these values: in our case, we save the number of samples, the minimum and maximum value, the average, plus an user-specified number of quantiles. The member `Quantiles` is a list of the quantiles requested by the user, which will be saved in a FITS file whose name is specified by the member `QuantileTableFileName`.

5. Another way to reduce the dimensionality of the data is to project all the ϕ_D/ϕ_{sky} values on a Healpix map. The `Nside` parameter specifies the resolution of the map, while the `OutputMapFileName` parameter is the name of the FITS file that will be created by the program.

Reading the parameters from a text file. The value of a `TPhiDConfiguration` and `TPhiSkyConfiguration` variable is initialized by reading a so-called INI file[5]. We are not going to provide here the details of the parsing of such files: see Appendix **??**

Here is an example of a configuration file for `phid`:

```
[Pointings]
template = /storage/pointings/LFI27M_%.4d.sqz
first_index = 91
last_index = 1604

[Main beam matrices]
file_name_1 = /storage/convmat/convmat_mb.dat

[Sidelobe matrices]
sidelobes = /storage/convmat/convmat_nl.dat
sidelobes = /storage/convmat/convmat_fl.dat

[Input]
satellite_velocity_file = /storage/planck_velocity.fits
dipole_dir_theta_ecl = 1.7656131194951572
dipole_dir_phi_ecl = 2.9958896005735780
dipole_speed_m_s = 370082.2332

[Output]
save_tods = true
```

[4]This is motivated by the fact that the numerical codes used to model the beam require different parameters in the two regions and uses different gridding schemes.

[5]`http://en.wikipedia.org/wiki/INI_file`.

```
tod_file_path = /storage/phid_tods/
quantiles = 25,50,75
quantiles_file_name = /storage/quantiles.fits
map_nside = 64
output_file_name = /storage/phid_maps/LFI27M_phid.fits
```

Similarly, a INI file accepted by `phisky` looks like the following:

```
[Pointings]
template = /storage/pointings/LFI27M_%.4d.sqz
first_index = 91
last_index = 1604

[Temperatures]
template = /storage/reduced/LFI27M_%.4d.sqz
first_index = 91
last_index = 1604

[Main ringsets]
file_name_1 = /storage/ringsets/main_beam/galaxy.fits
file_name_2 = /storage/ringsets/main_beam/cmb.fits

[Side ringsets]
file_name_1 = /storage/ringsets/near_sidelobes/galaxy.fits
file_name_2 = /storage/ringsets/near_sidelobes/cmb.fits
file_name_3 = /storage/ringsets/far_sidelobes/galaxy.fits
file_name_4 = /storage/ringsets/far_sidelobes/cmb.fits

[Input]
quality_flag_mask = 6111248

[Output]
save_tods = true
tod_file_path = /storage/phisky_tods/
interpolation_order = 5
quantiles = 25,50,75
quantiles_file_name = /storage/quantiles.fits
map_nside = 64
output_file_name = /storage/phisky_maps/LFI27M_phisky.fits
```

Each parameters is provided in the form `key = value`, and parameters are grouped in sections like `[Pointings]`, whose name is enclosed within square brackets. A few notes:

1. The convolution matrices and the ringsets used for the computation of the convolutions in Eq. (2) and Eq. (9) are listed in two separated sections (`[Main beam matrices]` and `[Sidelobe matrices]` for `phid`, and `[Main ringsets]` and `[Side ringsets]` for `phisky`). It is not mandatory to use names like `file_name_1`, `file_name_2`, …for the keys: the program will read any key/value entry within this section and assume that each specifies the path to a ringset file.

2. Pointings and temperature files are expressed by means of a template: the `%.4d` characters in the file name are substituted with a four-digit number (zero padded) which goes from `first_index` to `last_index`. The user can also list all the files explicitly:

   ```
   [Pointings]
   pointing_file_name0000 = /storage/pointings/LFI27M_0091.sqz
   ```

```
pointing_file_name0001 = /storage/pointings/LFI27M_0092.sqz
pointing_file_name0002 = /storage/pointings/LFI27M_0093.sqz
...
pointing_file_name1512 = /storage/pointings/LFI27M_1603.sqz
pointing_file_name1513 = /storage/pointings/LFI27M_1604.sqz
```

(Similarly for `[Temperatures]`.)

Neither `phid` nor `phisky` can run without a parameter file, which must be provided on the command line. Therefore, the first step is to verify that the user actually provided a parameter file:

10a ⟨*Check program options* 10a⟩≡ (6) 10c ▷

```
if ParamCount <> 1 then
begin
   PrintHelp;
   Exit;
end;
```

Uses `PrintHelp` 10b.

The `PrintHelp` function is extremely simple and only prints a lame one-liner, as it would be useless to carefully describe all the parameters accepted in the INI files in a error message to be printed on a terminal. Only the "master" MPI process is allowed to print the help on terminal, so that the user is not going to get N copies of the same text on the screen (with N being the number of MPI processes).

10b ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩≡ (4 5) 13a ▷

```
procedure PrintHelp;
begin
    if Mpi.CommRank(Mpi.World) = 0 then
        WriteLn(Format('Usage: %s PARAMETER_FILE', [ProgramName]));
end;
```

Defines:
`PrintHelp`, used in chunk 10a.

Once we are sure the user provided the name of a configuration file, we parse it using a yet-to-be-defined procedure `ReadConfiguration`. We print it on the screen immediately, so the user can check that everything is ok:

10c ⟨*Check program options* 10a⟩+≡ (6) ◁ 10a

```
ReadConfiguration(ParamStr(1), Configuration);
if Mpi.CommRank(Mpi.World) = 0 then
    PrintConfiguration(Configuration);
```

Uses `Configuration` 10d 10e, `PrintConfiguration` 38b 39a, and `ReadConfiguration` 36 37.

The two implementations of `ReadConfiguration` (one for `phid`, the other for `phisky`) are quite long but not too interesting, so we defer their presentation to Appendix **??**.

The `Configuration` variable is not a global variable, as it is visibile only within the main program's body. Of course, its type changes according to the program:

10d ⟨*Variables used in the main loop of* `phid` 10d⟩≡ (4) 11c ▷

```
Configuration : TPhiDConfiguration;
```

Defines:
`Configuration`, used in chunks 10c, 12, 14, 20–22, 28b, 31a, 32a, and 36–39.
Uses `TPhiDConfiguration` 7a.

10e ⟨*Variables used in the main loop of* `phisky` 10e⟩≡ (5) 11d ▷

```
Configuration : TPhiSkyConfiguration;
```

Defines:
`Configuration`, used in chunks 10c, 12, 14, 20–22, 28b, 31a, 32a, and 36–39.
Uses `TPhiSkyConfiguration` 7b.

This completes the part of the code which is devoted to the parsing of user-options. We move now to the implementation of the data-analysis tasks.

2.3. Raw data processing.

Loading convolution matrices and ringsets. The first step in the data processing part is to read the data used for the computation. We have two kind of datasets to load here: convolution matrices and ringsets (used to compute the convolutions in Eq 2 and Eq. 9), and pointing/temperature files. Convolution matrices and ringsets should be read once when the program starts, as they are used intensively during the execution. But pointing and temperature files should be loaded one by one, as it is a waste of space to keep them all in memory (each of them is going to be loaded and used by *one and only one* MPI process).

We aggregate the variables that will hold the data loaded when the program starts in two dedicated structures, `TPhiDInputData` and `TPhiSkyInputData`:

11a ⟨*Type definitions used by* `phid` 7a⟩+≡ (4) ◁7a 15a▷
```
TConvMatrixArray = array of TConvolutionMatrix;
TPhiDInputData = record
    MbMatrices : TConvMatrixArray;
    SlMatrices : TConvMatrixArray;
    SolSysVelEcl : TVector;
    SatelliteVelocity : TSatelliteVelocities;
end;
```
Defines:
`TPhiDInputData`, used in chunks 11c and 18.

11b ⟨*Type definitions used by* `phisky` 7b⟩+≡ (5) ◁7b 15b▷
```
TRingsetArray = array of TRingset;
TPhiSkyInputData = record
    MbRingsets : TRingsetArray;
    SlRingsets : TRingsetArray;
end;
```
Defines:
`TPhiSkyInputData`, used in chunks 11d, 19, and 43.

We need of course a variable in the main program with the appropriate type:

11c ⟨*Variables used in the main loop of* `phid` 10d⟩+≡ (4) ◁10d 15c▷
```
InputData : TPhiDInputData;
```
Uses `TPhiDInputData` 11a.

11d ⟨*Variables used in the main loop of* `phisky` 10e⟩+≡ (5) ◁10e 15d▷
```
InputData : TPhiSkyInputData;
```
Uses `TPhiSkyInputData` 11b.

We now provide an implementation of the code that initializes the `InputData` variables. Such code is called in the main program[6] and in the case of `phisky` is going to take some time (and eat much memory!). Therefore, `phisky` provides the user with an estimate of the memory needed by the ringsets using a new function, `MemoryUsed`, whose implementation is described in Appendix D.2.

12a ⟨*Read the ringsets* 12a⟩≡ (6b)

```
with Configuration do
begin
    Log('Loading the ringsets');
    LoadRingsets(MbRingsetFileNames, InterpolationOrder,
                 InputData.MbRingsets);
    LoadRingsets(SlRingsetFileNames, InterpolationOrder,
                 InputData.SlRingsets);

    Log(Format('Ringsets loaded, %s of memory currently used',
               [MemoryUsed(InputData)]));
end;
```

Uses `Configuration` 10d 10e, `LoadRingsets` 41b, `Log` 32c, and `MemoryUsed` 43.

In the case of `phid`, it is not likely we will end eating much memory, so no estimate is provided. But we must load satellite velocities as well:

12b ⟨*Read the* $\mathcal{M}$ *matrices and the satellite velocities* 12b⟩≡ (6a)

```
with Configuration do
begin
    Log('Loading the convolution matrices');

    LoadConvolutionMatrices(MbMatrixFileNames, InputData.MbMatrices);
    LoadConvolutionMatrices(SlMatrixFileNames, InputData.SlMatrices);

    Log('Convolution matrices loaded, loading the satellite velocities');

    LoadFromFile(SatelliteVelocityFileName, InputData.SatelliteVelocity);

    Log('Satellite velocities loaded');

    { Set up the velocity of the satellite wrt the Solar System }
    with InputData.SolSysVelEcl do
        Healpix.AnglesToVector(DipoleParams.DirTheta, DipoleParams.DirPhi, x, y, z);
    InputData.SolSysVelEcl :=
        ScaleVector(InputData.SolSysVelEcl, DipoleParams.SpeedMS);

end;
```

Uses `Configuration` 10d 10e, `LoadConvolutionMatrices` 40b, and `Log` 32c.

Splitting the work among the MPI processes. As we said in Sect. 1, our implementation of `phid` and `phisky` is going to use the MPI library, to allow the code to run on multiple-core clusters and thus to save the wall-clock time needed for the computation. From now on it is important to keep in mind that the code we are going to implement shall work on N different machines at the same time, with N being a number typically in the 10–100 range.

In Sect. 1 we defined the kind of job done by this program as “embarassingly parallel”. When the program runs in a parallel system, each process can work independently of the others (i.e.,

[6] Note that this initialization happens in *every* MPI process: thus, if N processes are running, every file needed to initialize `InputData` will be read – probably at the same time – N times. It might be wiser to change the code so that only one of the processes initializes `InputData` and sends it to the others. However, the implementation would be more complex because the predefined MPI functions only allow homogeneous vectors to be transferred among processes, while `InputData` is a complex data structure both in `phid` and in `phisky`.

no need to exchange information with other processes during the calculations). We are going to process a number of pointing files (each containing pointing information for one operational day of LFI), but each of them can be processed separatedly from the others. There are two possible approaches to this:

1. If the number of files to process is M, we can assign to each of the N MPI processes the analysis of M/N files (assuming that $M > N$). Of course, since M is not necessarily going to be a multiple of N, a few MPI processes might need to process one more file than the others.

2. The *master/slave* approach is surely the most efficient approach, but it is quite complex to implement. Basically, one of the MPI processes acts as a "master", which keeps a list of all the work that needs to be done (i.e., all the pointing files that must be processed). Any other MPI process is a "slave", which asks the master for new job (a pointing file to process), does it, and then asks the master again. Each time the master receive a request from a slave, checks what is the first job that has not been assigned yet, sends the job description to the slave, and then it marks the job as "assigned". When all the jobs have been assigned, the master has completed its task.

If the size of the collection of pointing files varied significantly, the master/slave approach would be the best, as the MPI processes lucky enough to get the smallest files would end doing more jobs than the other processes. However, in the case of LFI the pointing files are roughly all of the same size (this is a consequence of the fact that operational days have all roughly the same length, with only a few exceptions). Thus, in this version of the program we adopt the simplest approach of dividing the set of files into subsets with $\sim M/N$ elements: if $i = 0 \ldots N-1$ is the index ("rank") of a MPI process and f_k is the k-th pointing file to process, then the i-th process will analyze the subset of files given by

$$\left\{ f_{\lfloor iM/N \rfloor}, \ldots, f_{\lfloor (i+1)M/N \rfloor - 1} \right\}$$

This calculation is provided by the function `DivideMpiJobs` below (hint: in Pascal, `div` is the integer division):

13a ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁10b 26a▷

```
procedure DivideMpiJobs(MpiRank, MpiSize : Integer;
                        NumOfFiles : Integer;
                        out FirstIdx, LastIdx : Integer);
begin
    FirstIdx := (NumOfFiles * MpiRank) div MpiSize;
    LastIdx := (NumOfFiles * (MpiRank + 1)) div MpiSize - 1;
end;
```

Defines:
`DivideMpiJobs`, used in chunk 14.

On exit, the `DivideMpiJobs` function sets the value of `FirstIdx` and `LastIdx` to the index of the first and last (inclusive) element in an array of `NumOfFiles` which should be processed by the current MPI process. We must therefore define two variables, `FirstFileIdx` and `LastFileIdx`, in the main program, as well as an iterator that cycles over all the indexes in the range `FirstFileIdx...LastFileIdx`, and a couple of variables that will hold the index of the current MPI process and the number of running processes.

13b ⟨*Variables used by* `phid` *and* `phisky` *in the main loop* 13b⟩≡ (4 5) 20a▷

```
FirstFileIdx, LastFileIdx : Integer;
MpiRank, MpiSize : Integer;
Idx : Integer;
```

Defines:
`FirstFileIdx`, used in chunks 14 and 20–22.
`LastFileIdx`, used in chunks 14 and 20–22.

The procedure `DivideMpiJobs` assumes that the number of files M is larger than the number of MPI processes N (otherwise the equation for `LastIdx` might even produce negative numbers). Thus, in the main program we check that this condition holds: if it does not, then the variable `MpiSize` is redefined and the jobs with the highest rank will stop working (because they would have nothing to do).

14 ⟨*Subdivide the pointing files among the MPI processes* 14⟩≡ (6)

```
MpiRank := Mpi.CommRank(Mpi.World);
MpiSize := Mpi.CommSize(Mpi.World);
if MpiSize > Length(Configuration.PointingFileNames) then
begin
    MpiSize := Length(Configuration.PointingFileNames);
    if MpiRank >= MpiSize then
    begin
        Log(Format('Too many MPI processes (%d) and too few ' +
                   'files (%d): this process will stop',
                   [MpiSize, Length(Configuration.PointingFileNames)]));
        Exit;
    end;
end;

DivideMpiJobs(MpiRank, MpiSize, Length(Configuration.PointingFileNames),
              FirstFileIdx, LastFileIdx);

case LastFileIdx - FirstFileIdx + 1 of
0: Log('No files to load');
1: Log(Format('This MPI process will analyze pointing file %s',
              [Configuration.PointingFileNames[FirstFileIdx]]));
else Log(Format('This MPI process will analyze %d pointing files ' +
                    '(out of %d): %s … %s',
                 [LastFileIdx - FirstFileIdx + 1,
                  Length(Configuration.PointingFileNames),
                  Configuration.PointingFileNames[FirstFileIdx],
                  Configuration.PointingFileNames[LastFileIdx]]))
end;
```

Uses `Configuration` 10d 10e, `DivideMpiJobs` 13a, `FirstFileIdx` 13b, `LastFileIdx` 13b, and `Log` 32c.

Numerical calculation of ϕ_D and ϕ_{sky}. We need two new structures to hold the ϕ_D/ϕ_{sky} TODs. Each MPI process will use one variable of these types to hold the TOD for the pointing file that is currently processing.

The structure used by `phid` contains the following arrays:

1. The time of the observation, t (in `ObtTimes`);
2. The value of the dipole, D (in `D`);
3. The value of $B_s * D$ (in `BslD`);
4. The value of $B_m * D$ (in `BslD`);
5. The estimate for ϕ_D (in `PhiD`);
6. A flag that tells if the value estimated for ϕ_D is reliable or not (in `Valid`).

All these array have the same length, which is equal to the number of samples found in the pointing file being processed.

15a ⟨*Type definitions used by* `phid` 7a⟩+≡ (4) ◁11a

```
TPhiDTod = record
    ObtTimes : Array of Int64;
    D : Array of Double;
    BslD : Array of Double;
    BmD : Array of Double;
    PhiD : Array of Double;
    Valid : Array of Boolean;
end;
```

Defines:
`TPhiDTod`, used in chunks 15c, 16b, 18, and 23.

In the case of the `TPhiSkyTod` structure, the field `BslTsky` contains the value of $B_s * T_{\text{sky}}$, while `BmTsky` contains $B_m * T_{\text{sky}}$. The `Valid` field is `False` whenever the denominator in Eq. 2 is zero. The field `TskyMeas` contains $\tilde{T}_{\text{sky}}$, and it is copied from the temperature file.

15b ⟨*Type definitions used by* `phisky` 7b⟩+≡ (5) ◁11b

```
TPhiSkyTod = record
    ObtTimes : Array of Int64;
    TskyMeas : Array of Double;
    BslTsky : Array of Double;
    BmTsky : Array of Double;
    PhiSky : Array of Double;
    Valid : Array of Boolean;
end;
```

Defines:
`TPhiSkyTod`, used in chunks 15d, 17, 19, and 24.

A variable of the corresponding type is declared in the main block:

15c ⟨*Variables used in the main loop of* `phid` 10d⟩+≡ (4) ◁11c

```
PhiTod : TPhiDTod;
```

Defines:
`PhiTod`, used in chunks 21 and 22a.
Uses `TPhiDTod` 15a.

15d ⟨*Variables used in the main loop of* `phisky` 10e⟩+≡ (5) ◁11d

```
PhiTod : TPhiSkyTod;
```

Defines:
`PhiTod`, used in chunks 21 and 22a.
Uses `TPhiSkyTod` 15b.

The term D in the TPhiDTod variable can be easily computed using the convolution matrix for a Dirac's delta:

16a ⟨*High-level functions for the* phid *program* 16a⟩≡ (4) 16b ▷

```
procedure ComputeD(const SolSysVelEcl : TVector;
                   const Pointings : TDetectorPointings;
                   const Vel : TSatelliteVelocities;
                   var D : TDoubleArray);
var
    Idx : Integer;

begin
    SetLength(D, Length(Pointings.Theta));
    for Idx := 0 to Length(Pointings.Theta) - 1 do
    begin
        with Pointings do
            D[Idx] := Convolve(DiracDelta, SolSysVelEcl, Vel,
                                ScetTimes[Idx], Theta[Idx], Phi[Idx], Psi[Idx]);
    end;
end;
```

Defines:
ComputeD, used in chunk 18.

Assuming that a TPhiDTod variable has had its members BslD and BmTd already initialized, the procedure CalculatePhiD computes ϕ_D using Eq. 2.

16b ⟨*High-level functions for the* phid *program* 16a⟩+≡ (4) ◁ 16a 18 ▷

```
procedure CalculatePhiD(var PhiDTod : TPhiDTod);
var
    Idx : Integer;
    BmDDiff : Double;

begin
    with PhiDTod do
    begin
        SetLength(PhiD, Length(BslD));
        SetLength(Valid, Length(BmD));
        for Idx := 1 to Length(PhiD) - 1 do
        begin
            BmDDiff := BmD[Idx] - BmD[Idx - 1];
            if BmDDiff <> 0.0 then
            begin
                PhiD[Idx] := (BslD[Idx] - BslD[Idx - 1]) / BmDDiff;
                Valid[Idx] := True;
            end else
                Valid[Idx] := False;
        end;
    end;
end;
```

Defines:
CalculatePhiD, used in chunk 18.
Uses TPhiDTod 15a.

The procedure `CalculatePhiSky` has a similar implementation:

17 ⟨*High-level functions for the* `phisky` *program* 17⟩≡ (5) 19 ▷

```
procedure CalculatePhiSky(const Flags : Array of UInt32;
                          QualityFlagMask : UInt32;
                          var PhiSkyTod : TPhiSkyTod);
var
    Idx : Integer;

begin
    with PhiSkyTod do
    begin
        SetLength(PhiSky, Length(BslTsky));
        SetLength(Valid, Length(BslTsky));
        for Idx := 0 to Length(PhiSky) - 1 do
        begin
            if (TskyMeas[Idx] <> 0.0) and ((Flags[Idx] and QualityFlagMask) = 0) then
            begin
                PhiSky[Idx] := BslTsky[Idx] / TskyMeas[Idx];
                Valid[Idx] := True;
            end else
                Valid[Idx] := False;
        end;
    end;
end;
```

Defines:
`CalculatePhiSky`, used in chunk 19.

Uses `TPhiSkyTod` 15b.

To use `CalculatePhiD` and `CalculatePhiSky`, we need to wrap their call in a function which loads the pointing and temperature files (the latter are needed by `phisky` only, of course, as they are used to get the term $\tilde{T}_{\mathrm{sky}}$ in Eq. 9). This is the purpose of `ProcessPointingFile` (used by `phid`) and `ProcessFilePair` (used by `phisky`), which save the TODs in two `out` variables (`PhiDTod` and `PhiSkyTod`) and calls a function `ProjectTodOntoMap`: the purpose of the latter will be explained later.

18 ⟨*High-level functions for the* `phid` *program* 16a⟩+≡ (4) ◁16b 23▷

```
{ Forward declaration }
procedure SumConvMatricesIntensities(const Matrices : TConvMatrixArray;
                                     const SolSysVelEcl : TVector;
                                     const SatVel : TSatelliteVelocities;
                                     const Pointings : TDetectorPointings;
                                     var DestVector : TDoubleArray); forward;

procedure ProcessPointingFile(const FileName : String;
                              const InputData : TPhiDInputData;
                              out PhiDTod : TPhiDTod;
                              var BinnedMap : THealpixMap;
                              var HitMap : THealpixMap);
var
    FileHeader : Squeezer.TFileHeader;
    Pointings : TDetectorPointings;

begin
    Log(Format('Reading pointing file %s…', [FileName]));
    ReadDetectorPointings(FileName, FileHeader, Pointings);
    Log(Format('…pointing file read, %d samples found',
               [Length(Pointings.ObtTimes)]));

    SetLength(PhiDTod.ObtTimes, Length(Pointings.ObtTimes));
    Move(Pointings.ObtTimes[0], PhiDTod.ObtTimes[0],
         Length(Pointings.ObtTimes) * SizeOf(Pointings.ObtTimes[0]));

    Log('Computing the term D…');
    ComputeD(InputData.SolSysVelEcl, Pointings,
             InputData.SatelliteVelocity, PhiDTod.D);
    Log('…term D computed');

    Log(Format('Calculating convolutions using %d+%d (MB/SL) convolution matrices…',
               [Length(InputData.MbMatrices), Length(InputData.SlMatrices)]));
    SumConvMatricesIntensities(InputData.MbMatrices, InputData.SolSysVelEcl,
                               InputData.SatelliteVelocity, Pointings, PhiDTod.BmD);
    SumConvMatricesIntensities(InputData.SlMatrices, InputData.SolSysVelEcl,
                               InputData.SatelliteVelocity, Pointings, PhiDTod.BslD);
    Log('…convolutions calculated');

    Log('Calculating _D…');
    CalculatePhiD(PhiDTod);
    Log('…_D calculated');

    ProjectTodOntoMap(Pointings, PhiDTod.PhiD, PhiDTod.Valid, BinnedMap, HitMap);
end;
```

Defines:
`ProcessPointingFile`, used in chunk 21.

Uses `CalculatePhiD` 16b, `ComputeD` 16a, `Log` 32c, `ProjectTodOntoMap` 28c, `SumConvMatricesIntensities` 41a, `TPhiDInputData` 11a, and `TPhiDTod` 15a.

The name ProcessFilePair refers to the fact that we need to load the temperature file as well as the pointing file. Its implementation is therefore slightly more complex than ProcessPointingFile. It uses a function, SumRingsetIntensities, which iterates over an array of ringsets and produces the sum of their intensities for a set of pointings; its implementation is trivial, and we defer its implementation to Appendix D.2.

19 ⟨*High-level functions for the* phisky *program* 17⟩+≡ (5) ◁17 24▷

```
{ Forward declaration }
procedure SumRingsetIntensities(const Ringsets : TRingsetArray;
                                const Pointings : TDetectorPointings;
                                var DestVector : TDoubleArray); forward;

procedure ProcessFilePair(const PntFileName, TskyFileName : String;
                          const InputData : TPhiSkyInputData;
                          QualityFlagMask : UInt32;
                          out PhiSkyTod : TPhiSkyTod;
                          var BinnedMap : THealpixMap;
                          var HitMap : THealpixMap);

var
    FileHeader : Squeezer.TFileHeader;
    Pointings : TDetectorPointings;
    Temperature : TDifferencedData;

begin
    Log(Format('Reading pointing file %s…', [PntFileName]));
    ReadDetectorPointings(PntFileName, FileHeader, Pointings);
    Log(Format('…pointing file read, %d samples found',
               [Length(Pointings.ObtTimes)]));

    Log(Format('Reading temperature file %s…', [TskyFileName]));
    ReadDifferencedData(TskyFileName, FileHeader, Temperature);
    Log(Format('…temperature file read, %d samples found',
               [Length(Pointings.ObtTimes)]));

    SetLength(PhiSkyTod.ObtTimes, Length(Pointings.ObtTimes));
    Move(Pointings.ObtTimes[0], PhiSkyTod.ObtTimes[0],
         Length(Pointings.ObtTimes) * SizeOf(Pointings.ObtTimes[0]));

    SetLength(PhiSkyTod.TskyMeas, Length(Temperature.SkyLoad));
    Move(Temperature.SkyLoad[0], PhiSkyTod.TskyMeas[0],
         Length(Temperature.SkyLoad) * SizeOf(Temperature.SkyLoad[0]));

    Log(Format('Calculating convolutions using %d+%d (MB/SL) ringsets…',
              [Length(InputData.MbRingsets), Length(InputData.SlRingsets)]));
    SumRingsetIntensities(InputData.MbRingsets, Pointings, PhiSkyTod.BmTsky);
    SumRingsetIntensities(InputData.SlRingsets, Pointings, PhiSkyTod.BslTsky);
    Log('…convolutions calculated');

    Log('Calculating _sky…');
    CalculatePhiSky(Temperature.Flags, QualityFlagMask, PhiSkyTod);
    Log('…_sky calculated');

    ProjectTodOntoMap(Pointings, PhiSkyTod.PhiSky, PhiSkyTod.Valid, BinnedMap, HitMap);
end;
```

Defines:
ProcessFilePair, used in chunk 22a.

Uses CalculatePhiSky 17, Log 32c, ProjectTodOntoMap 28c, SumRingsetIntensities 42, TPhiSkyInputData 11b, and TPhiSkyTod 15b.

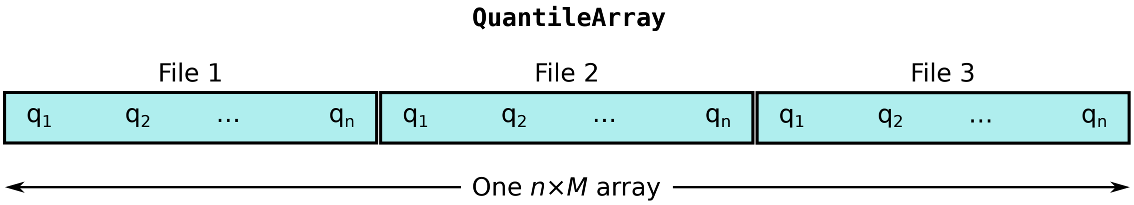

Figure 1: *How quantiles for each MPI process are stored in memory.*

In the main program we iterate through all the pointing/temperature filenames that this MPI process must analyze (these are the elements in `Configuration.PointingFileNames` and `Configuration.TemperatureFileNames` whose index falls within `FirstFileIdx`…`LastFileIdx`). If the user asked to save the raw TODs, we call the procedure `SavePhiTod`, which we will implement in the next section.

Let's now concentrate to the way quantiles are stored in memory. Each MPI process must analyze a set of files, which is a subset of all the files specified in the parameter file [REF XXX]. If the MPI process needs to process M files and extract n quantiles from each of them, then the most natural data structure in which to keep such quantiles is a $n \times M$ bi-dimensional array (matrix). However, in order to make things easier when we will implement the code to pass such quantiles through MPI processes, we choose to use monodimensional arrays that are indexed like a bi-dimensional one (see Fig. 1).

We declare two variables that will hold the quantiles: the first one keeps the quantiles calculated for the files processed by the current MPI process, while the second one will hold all the quantiles (this will be used by the root process only):

20a ⟨*Variables used by* `phid` *and* `phisky` *in the main loop* 13b⟩+≡ (4 5) ◁13b 22b▷

```
QuantileArray, OverallQuantileArray : TDoubleArray;
```

As we said above, the size of `QuantileArray` should be $n \times M$, with n the number of quantiles to compute and M the number of files processed by the current MPI process.

20b ⟨*Initialize the structures used to compress the TODs* 20b⟩≡ (6) 28b▷

```
SetLength(QuantileArray, (LastFileIdx - FirstFileIdx + 1) * Length(Configuration.Quantiles));
```

Uses `Configuration` 10d 10e, `FirstFileIdx` 13b, and `LastFileIdx` 13b.

We are now ready to implement the main loop of the application:

21 ⟨*Apply Eq.* (2) *to the data in each pointing/temperature file* 21⟩≡ (6a)

```
for CurFileIdx := FirstFileIdx to LastFileIdx do
begin
    with Configuration do
    begin
        try
            ProcessPointingFile(PointingFileNames[CurFileIdx],
                                InputData, PhiTod, BinnedMap, HitMap);

            AppendQuantiles(PhiTod.PhiD, PhiTod.Valid, Configuration.Quantiles,
                            QuantileArray, CurFileIdx - FirstFileIdx);

            if Configuration.SaveTods then
                SavePhiTod(ConcatPaths([TodFilePath,
                    ChangeFileExt(ExtractFileName(PointingFileNames[CurFileIdx]),
                                  '-phid.fits')]),
                           PhiTod);
        except
            on E : Exception do
                Log(Format('Unable to process file "%s" (%s), skipping…',
                           [PointingFileNames[CurFileIdx],
                            E.Message]));
        end;
    end;
end;
```

Uses `AppendQuantiles` 26b, `Configuration` 10d 10e, `FirstFileIdx` 13b, `LastFileIdx` 13b, `Log` 32c, `PhiTod` 15c 15d, `ProcessPointingFile` 18, and `SavePhiTod` 23.

The program `phisky` uses a quite similar loop. Of course, in this case we need to consider temperature files as well.

22a ⟨*Apply Eq.* (9) *to the data in each pointing/temperature file* 22a⟩≡ (6b)

```
for CurFileIdx := FirstFileIdx to LastFileIdx do
begin
    with Configuration do
    begin
        try
            ProcessFilePair(PointingFileNames[CurFileIdx],
                            TemperatureFileNames[CurFileIdx],
                            InputData, Configuration.QualityFlagMask,
                            PhiTod, BinnedMap, HitMap);

            AppendQuantiles(PhiTod.PhiSky, PhiTod.Valid, Configuration.Quantiles,
                            QuantileArray, CurFileIdx - FirstFileIdx);

            if Configuration.SaveTods then
                SavePhiTod(ConcatPaths([TodFilePath,
                    ChangeFileExt(ExtractFileName(TemperatureFileNames[CurFileIdx]),
                                  '-phisky.fits')]),
                             PhiTod);
        except
            on E : Exception do
                Log(Format('Unable to process files "%s" and "%s" (%s), skipping…',
                           [PointingFileNames[CurFileIdx],
                            TemperatureFileNames[CurFileIdx],
                            E.Message]));
        end;
    end;
end;
```

Uses `AppendQuantiles` 26b, `Configuration` 10d 10e, `FirstFileIdx` 13b, `LastFileIdx` 13b, `Log` 32c, `PhiTod` 15c 15d, `ProcessFilePair` 19, and `SavePhiTod` 23.

The variable `CurFileIdx` used in the two `for` loops above is declared within the scope of the main block:

22b ⟨*Variables used by* `phid` *and* `phisky` *in the main loop* 13b⟩+≡ (4 5) ◁20a 28a▷

```
CurFileIdx : Integer;
```

Defines:
`CurFileName`, used in chunks 40b and 41b.

Saving the ϕ_D and ϕ_{sky} TODs. The two programs save the information in the `TPhiDTod` and `TPhiSkyTod` variable into a FITS file. In both cases, the file contains just one binary table HDU and is created by the procedure `SavePhiTod`, whose implementation of course differ between `phid` and `phisky`.

23 ⟨*High-level functions for the* `phid` *program* 16a⟩+≡ (4) ◁18 36▷

```
procedure SavePhiTod(const FileName : String;
                     const PhiDTod : TPhiDTod);
const
    FileColumns : Array[1..5] of Cfitsio.TColumn =
        ((Name: 'OBTTIME'; Count: 1; DataType: FitsTypeDouble;  UnitStr: ''),
         (Name: 'BSLD';    Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'BMD';     Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'PHID';    Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'VALID';   Count: 1; DataType: FitsTypeLogical; UnitStr: ''));

var
    F : TFitsFile;

begin
    Log(Format('Saving %d values of _D into file %s…',
               [Length(PhiDTod.ObtTimes), FileName]));
    try
        EnsurePathExists(FileName);
        F := Cfitsio.CreateFile(FileName, Overwrite);
        try
            Cfitsio.CreateTable(F, BinaryTable, 0, FileColumns, 'PHID');
            Cfitsio.WriteComment(F, 'File created by the phid program');
            Cfitsio.WriteComment(F, 'Phid was compiled on ' +
                                    {$i %date} + ' ' + {$i %time});

            Cfitsio.WriteColumn(F, 1, 1, 1, PhiDTod.ObtTimes);
            Cfitsio.WriteColumn(F, 2, 1, 1, PhiDTod.BslD);
            Cfitsio.WriteColumn(F, 3, 1, 1, PhiDTod.BmD);
            Cfitsio.WriteColumn(F, 4, 1, 1, PhiDTod.PhiD);
            Cfitsio.WriteColumn(F, 5, 1, 1, PhiDTod.Valid);

            Log(Format('…done, file %s has been saved', [FileName]));
        finally
            Cfitsio.CloseFile(F);
        end;
    except
        on E : EFitsError do Log(Format('Unable to write file %s: %s',
                                        [FileName, E.message]));
    end;
end;
```

Defines:
  `SavePhiTod`, used in chunks 21, 22a, and 24.
Uses `EnsurePathExists` 33a, `Log` 32c, and `TPhiDTod` 15a.

The function `EnsurePathExists` is implemented in Appendix B, and it creates the path needed to save the file specified in its argument if needed (e.g., a call to `EnsurePathExists('/path/test.fits')` will create the directory `/path` if it does not exist).

24 ⟨*High-level functions for the* `phisky` *program* 17⟩+≡ (5) ◁19 37▷

```
procedure SavePhiTod(const FileName : String;
                     const PhiSkyTod : TPhiSkyTod);
const
    FileColumns : Array[1..6] of Cfitsio.TColumn =
        ((Name: 'OBTTIME'; Count: 1; DataType: FitsTypeDouble;  UnitStr: ''),
         (Name: 'SLCONV';  Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'MBCONV';  Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'TSKY';    Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'PHISKY';  Count: 1; DataType: FitsTypeFloat;   UnitStr: 'K_CMB'),
         (Name: 'VALID';   Count: 1; DataType: FitsTypeLogical; UnitStr: ''));

var
    F : TFitsFile;

begin
    Log(Format('Saving %d values of _sky into file %s',
               [Length(PhiSkyTod.ObtTimes), FileName]));
    try
        EnsurePathExists(FileName);
        F := Cfitsio.CreateFile(FileName, Overwrite);
        try
            Cfitsio.CreateTable(F, BinaryTable, 0, FileColumns, 'PHISKY');
            Cfitsio.WriteComment(F, 'File created by the phisky program');
            Cfitsio.WriteComment(F, 'Phisky was compiled on ' +
                                    {$i %date} + ' ' + {$i %time});
            Cfitsio.WriteColumn(F, 1, 1, 1, PhiSkyTod.ObtTimes);
            Cfitsio.WriteColumn(F, 2, 1, 1, PhiSkyTod.BslTsky);
            Cfitsio.WriteColumn(F, 3, 1, 1, PhiSkyTod.BmTsky);
            Cfitsio.WriteColumn(F, 4, 1, 1, PhiSkyTod.TskyMeas);
            Cfitsio.WriteColumn(F, 5, 1, 1, PhiSkyTod.PhiSky);
            Cfitsio.WriteColumn(F, 6, 1, 1, PhiSkyTod.Valid);
        finally
            Cfitsio.CloseFile(F);
        end;
    except
        on E : EFitsError do Log(Format('Unable to write file %s: %s',
                                    [FileName, E.message]));
    end;
end;
```

Uses `EnsurePathExists` 33a, `Log` 32c, `SavePhiTod` 23, and `TPhiSkyTod` 15b.

2.4. Compressing and saving the results. The $\phi_{\rm sky}$ TODs generated by the program require much disk space (of the same order of magnitude as the pointings, which means roughly 20 GB/radiometer if saved in a compressed format) Therefore, the code reduces the dimensionality of the TODs in two ways:

- Statistical quantities are computed for each OD, and only these are saved. This is still a TOD, but instead of having one row per sample, we have one row per OD.
- The values of $\phi_{\rm sky}$ are projected (binned) on a Healpix map.

The `TPercentageArray` type used in the definition of `Quantiles` is an open array of integer numbers (`TPercentage`) representing a percentage. We use this custom type instead of a `Integer` because in this way its range is automatically checked by the compiler:

25a ⟨*Basic type definitions (shared between* `phisky` *and* `phid`*)* 25a⟩≡ (4 5) 25b ▷

```
TPercentage = 0..100;
TPercentageArray = array of TPercentage; { Open array }
```

Defines:
`TPercentage`, never used.
`TPercentageArray`, used in chunks 7, 26b, 30, 35, and 40a.

We add a few useful new types as well:

25b ⟨*Basic type definitions (shared between* `phisky` *and* `phid`*)* 25a⟩+≡ (4 5) ◁ 25a

```
TStringArray = array of String;  { Open array }
TDoubleArray = array of Double;  { Open array }
TBoolArray   = array of Boolean; { Open array }
```

Defines:
`TStringArray`, used in chunks 7, 33b, 34, and 39–41.

25c ⟨*Compress the $\phi_{\rm sky}$ TODs and produce reduced TODs* 25c⟩≡

```
{ No code here }
```

Producing statistics for each OD. To compute the quantiles, we need a sorting procedure. Unfortunately, Free Pascal does not provide a general-purpose sorting routine, so we provide here our own implementation of the "quick sort" algorithm:

26a ⟨*General functions (shared between* phisky *and* phid*)* 10b⟩+≡ (4 5) ◁13a 26b▷

```
procedure InPlaceQuickSort(var A : Array of Double; FirstIdx, LastIdx : Integer);
Var
    i, j : LongInt;
    tmp, pivot : Double;

Begin
    i := FirstIdx;
    j := LastIdx;
    pivot := A[(FirstIdx + LastIdx) div 2];
    repeat
        while pivot > A[i] do Inc(i);
        While pivot < A[j] do Dec(j);
        if i <= j then begin
            tmp := A[i];
            A[i] := A[j];
            A[j] := tmp;
            Inc(i);
            Dec(j);
        end;
    until i > j;
    if FirstIdx < j then InPlaceQuickSort(A, FirstIdx, j);
    if i < LastIdx then InPlaceQuickSort(A, i, LastIdx);
End;
```

Defines:
`InPlaceQuickSort`, used in chunk 27b.

To save memory, the `InPlaceQuickSort` (as its name suggests) performs an in-place sorting, so that at the end of the call the original ordering of the double array `A` is lost.

Before applying `InPlaceQuickSort`, we need to filter out those elements of the ϕ_{sky} array that contain invalid values. Therefore, our implementation of the `AppendQuantiles` function has the following structure:

26b ⟨*General functions (shared between* phisky *and* phid*)* 10b⟩+≡ (4 5) ◁26a 28c▷

```
procedure AppendQuantiles(const A : TDoubleArray;
                          const Valid : TBoolArray;
                          const Quantiles : TPercentageArray;
                          var QuantileArray : TDoubleArray;
                          ChunkIdx : Integer);
var
    ValidValues : TDoubleArray;
    Idx, ValidIdx, QuantIdx : Integer;
    ValuesStr : String;
begin
    Assert(Length(A) = Length(Valid));
    Log(Format('Computing %d quantiles out of an array of %d elements…',
               [Length(Quantiles), Length(A)]));
    ⟨Pick the valid values from A and store them into ValidValues 27a⟩
    ⟨Sort ValidValues and compute the quantiles from it 27b⟩
    Log(Format('…the quantiles are %s.', [ValuesStr]));
end;
```

Defines:
`AppendQuantiles`, used in chunks 21 and 22a.
Uses `Log` 32c and `TPercentageArray` 25a.

The purpose of the variable `ValidValues` is to hold a subset of the `A` array which corresponds to those values whose twin element in `Valid` (an array of Boolean values) is `True`:

27a ⟨*Pick the valid values from* `A` *and store them into* `ValidValues` 27a⟩≡ (26b)

```
SetLength(ValidValues, Length(A));
ValidIdx := Low(ValidValues);
for Idx := Low(A) to High(A) do
begin
    if Valid[Idx] then
    begin
        ValidValues[ValidIdx] := A[Idx];
        Inc(ValidIdx);
    end;
end;
if ValidIdx = Low(ValidValues) then
begin
    Log('…no valid values found for this OD, skipping the computation of quantiles');
    Exit;
end;

SetLength(ValidValues, ValidIdx);  { Truncate the tail of ValidValues }
Log(Format('…%d valid values found…', [Length(ValidValues)]));
```

Uses `Log` 32c.

Once `ValidValues` is initialized, computing the quantiles is a matter of picking the right indexes in the array. We do not check for those cases where the quantile falls in the middle between two values (the `div` operation is an integer division and throws away any remainder), because we feel that the loss of precision due to this choice is negligible but makes the code considerably simpler.

27b ⟨*Sort* `ValidValues` *and compute the quantiles from it* 27b⟩≡ (26b)

```
                InPlaceQuickSort(ValidValues, Low(ValidValues), High(ValidValues));
Log(Format('…values have been sorted, their range is [%.3e, %.3e]…',
           [ValidValues[Low(ValidValues)], ValidValues[High(ValidValues)]]));
Idx := ChunkIdx * Length(Quantiles);
ValuesStr := '';
for QuantIdx := Low(Quantiles) to High(Quantiles) do
begin
    QuantileArray[Idx + QuantIdx] :=
        ValidValues[(Quantiles[QuantIdx] * Length(ValidValues)) div 100];
    if ValuesStr <> '' then ValuesStr := ValuesStr + ', ';
    ValuesStr := ValuesStr + Format('%.3e (%d%%)',
                                    [QuantileArray[Idx + QuantIdx], Quantiles[QuantIdx]]);
end;
```

Uses `InPlaceQuickSort` 26a and `Log` 32c.

Projecting ϕ_D and ϕ_{sky} on a Healpix map. Another way to condensate the amount of information enclosed in a set of $\phi_D/\phi_{\mathrm{sky}}$ TODs is to project their value on a map. Both `phid` and `phisky` take advantage of a set of a Free Pascal implementation of the Healpix pixelisation scheme to produce a FITS file containing the binned map of ϕ_D and ϕ_{sky} values.

The two programs produce the maps via the following steps:

1. Each time a pointing file is processed and the $\phi_D/\phi_{\mathrm{sky}}$ TOD has been calculated, it is immediately projected onto a pair of "local" maps. The first map keeps track of the sum of the $\phi_D/\phi_{\mathrm{sky}}$ values that have "hit" that pixel, while the second one keeps track of the number of hits per pixel. These maps are qualified as "local", as each MPI process keeps its own pair of maps.

2. When all the pointing files have been processed by all the MPI processes, a "reduce" operation is performed by the root process (with rank #0), and all the binned maps and hit maps are summed together.

3. Dividing each pixel of the binned map by the corresponding value in the hit map produces a map of $\phi_D/\phi_{\rm sky}$ values.

Each process has two pairs of binned/hit maps: the first one is actually used by each process (`BinnedMap` and `HitMap`), while the second pair is only used by the root process to collect the result of all the maps.

28a ⟨*Variables used by* `phid` *and* `phisky` *in the main loop* 13b⟩+≡ (4 5) ◁22b 32b▷

```
BinnedMap, HitMap : THealpixMap;
OverallBinnedMap, OverallHitMap : THealpixMap;
```

These variables are initialized before the loop over the pointing files starts. We initialize `OverallBinnedMap` and `OverallHitMap` in every MPI process, even if only the root process will actually use it, because in this way we turn off a few warnings that might be issued by the Free Pascal compiler:

28b ⟨*Initialize the structures used to compress the TODs* 20b⟩+≡ (6) ◁20b

```
InitMap(Configuration.Nside, Ring, BinnedMap);
InitMap(Configuration.Nside, Ring, HitMap);

InitMap(Configuration.Nside, Ring, OverallBinnedMap);
InitMap(Configuration.Nside, Ring, OverallHitMap);
```

Uses `Configuration` 10d 10e.

The function that is used to construct the binned and hit maps is `ProjectTodOntoMap`. It assumes that the `NSIDE` value used by `BinnedMap` and `HitMap` is the same.

28c ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁26b 29▷

```
procedure ProjectTodOntoMap(const Pointings : TDetectorPointings;
                            const Tod : TDoubleArray;
                            const Valid : TBoolArray;
                            var BinnedMap : THealpixMap;
                            var HitMap : THealpixMap);
var
    Idx : Integer;
    PixelIdx : Cardinal;

begin
    Log(Format('Projecting %d samples into a NSIDE=%d map…',
               [Length(Tod), BinnedMap.Resolution.Nside]));
    for Idx := Low(Tod) to High(Tod) do
    begin
        if Valid[Idx] then
        begin
            PixelIdx := AnglesToPix(BinnedMap, Pointings.Theta[Idx], Pointings.Phi[Idx]);
            BinnedMap.Pixels[PixelIdx] := BinnedMap.Pixels[PixelIdx] + Tod[Idx];
            HitMap.Pixels[PixelIdx] := HitMap.Pixels[PixelIdx] + 1.0;
        end;
    end;
    Log('…projection completed.');
end;
```

Defines:
`ProjectTodOntoMap`, used in chunks 18 and 19.
Uses `Log` 32c.

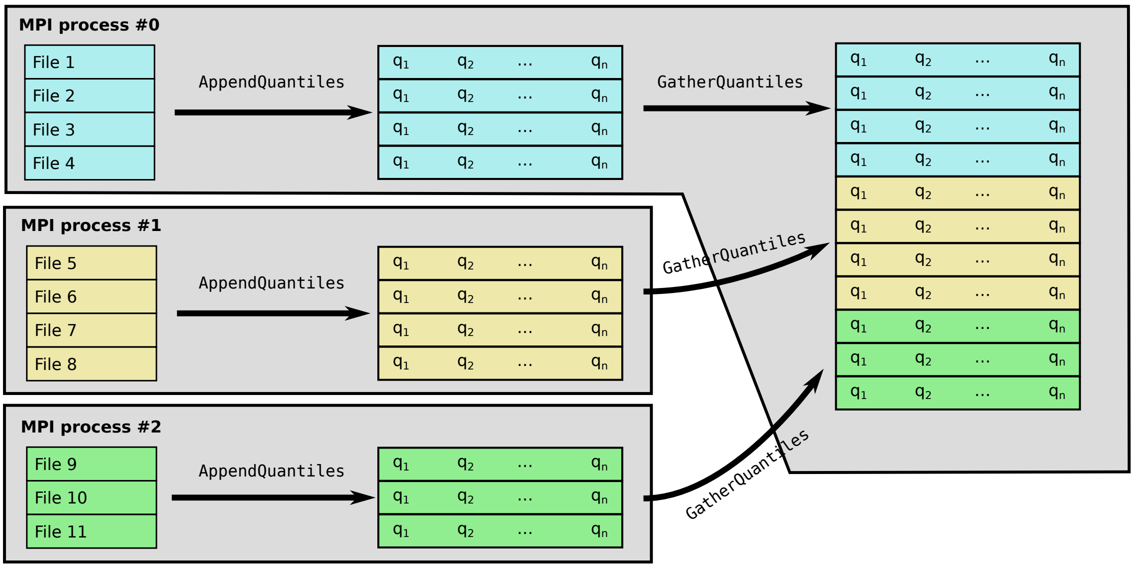

Figure 2: *How quantiles are gathered together by the root MPI process. In this simple example we consider a run where a total of 11 files must be processed by 3 MPI processes. Each MPI process computes its quantiles by calling* `AppendQuantiles` *iteratively. At the end of the process, all the MPI processes but the first send their results to the first process (the "root") by means of the procedure* `GatherQuantiles`.

Saving the reduced TODs and the maps. So far, the quantiles have been calculated by each MPI process for its own files. It is time to gather them together and produce just one file with all the quantiles (one row per file). This process is done by the procedure `GatherQuantiles`. Its role with respect to `AppendQuantiles` is sketched in Fig. 2. The implementation of `GatherQuantiles` relies on the MPI functions `Gather` and `Gatherv`.

29 ⟨*General functions (shared between* phisky *and* phid*)* 10b⟩+≡ (4 5) ◁28c 30▷

```
procedure GatherQuantiles(const LocalQuantiles : Array of Double;
                          out OverallQuantiles : TDoubleArray);
var
    Idx : Integer;
    BufLengths : Array of Integer;
    Displacements : Array of Integer;
    LocalBufLength : Array[1..1] of Integer;

begin
    { Retrieve the number of quantiles computed by each MPI process }
    SetLength(BufLengths, Mpi.CommSize(Mpi.World));
    LocalBufLength[1] := Length(LocalQuantiles);
    Mpi.Gather(LocalBufLength, BufLengths, 0, Mpi.World);

    { Set up the array of displacements so that no holes will be left
      in OverallQuantiles }
    SetLength(Displacements, Length(BufLengths));
    Displacements[0] := 0;
    for Idx := Low(BufLengths) to High(BufLengths) - 1 do
        Displacements[Idx + 1] := Displacements[Idx] + BufLengths[Idx];

    { Gather the quantiles from each MPI process to the root process }
    SetLength(OverallQuantiles,
              Displacements[High(Displacements)] + BufLengths[High(BufLengths)]);
    Mpi.Gatherv(LocalQuantiles, OverallQuantiles, BufLengths,
                Displacements, 0, Mpi.World);
```

```
end;
```

Defines:
`GatherQuantiles`, used in chunk 31a.

30 ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁29 33b▷

```
procedure SaveQuantileTOD(const FileName : String;
                          const Quantiles : TPercentageArray;
                          const QuantileArray : TDoubleArray);
const
    PercentTableDef : Array[1..1] of Cfitsio.TColumn =
        ((Name: 'PERCENT'; Count: 1; DataType: FitsTypeShort; UnitStr: 'Percentage'));

var
    QuantileTableDef : Array of Cfitsio.TColumn;
    QuantIdx, Idx : Integer;
    F : TFitsFile;
    CurQuantiles : TDoubleArray;

begin
    SetLength(QuantileTableDef, Length(Quantiles));
    for QuantIdx := Low(QuantileTableDef) to High(QuantileTableDef) do
    begin
        with QuantileTableDef[QuantIdx] do
        begin
            Name := Format('Q%.3d', [Quantiles[QuantIdx]]);
            Count := 1;
            DataType := FitsTypeFloat;
            UnitStr := '';
        end;
    end;

    try
        Log(Format('Saving quantiles into FITS file "%s"…', [FileName]));
        F := Cfitsio.CreateFile(FileName, Overwrite);
        try
            CreateTable(F, BinaryTable, 0, PercentTableDef, 'PERCENTAGES');
            WriteColumn(F, 1, 1, 1, Quantiles);

            CreateTable(F, BinaryTable, 0, QuantileTableDef, 'QUANTILES');
            SetLength(CurQuantiles, Length(QuantileArray) div Length(Quantiles));
            for QuantIdx := 0 to Length(Quantiles) - 1 do
            begin
                for Idx := 0 to Length(CurQuantiles) - 1 do
                    CurQuantiles[Idx] := QuantileArray[Idx * Length(Quantiles) + QuantIdx];
                WriteColumn(F, 1 + QuantIdx, 1, 1, CurQuantiles);
            end;
        finally
            Cfitsio.CloseFile(F);
        end;
    except
        on E : Exception do Log(Format('Unable to write file "%s": %s',
                                       [FileName, E.Message]));
    end;
    Log(Format('…file "%s" saved.', [FileName]));
end;
```

Uses `Log` 32c and `TPercentageArray` 25a.

31a ⟨*Compress the TODs and produce reduced TODs* 31a⟩≡ (6) 31b ▷

```
GatherQuantiles(QuantileArray, OverallQuantileArray);
if MpiRank = 0 then
begin
    EnsurePathExists(Configuration.QuantileTableFileName);
    SaveQuantileTOD(Configuration.QuantileTableFileName,
                    Configuration.Quantiles, OverallQuantileArray);
end;
```

Uses `Configuration` 10d 10e, `EnsurePathExists` 33a, and `GatherQuantiles` 29.

To produce the Healpix map containing the binned values of ϕ_D and ϕ_{sky}, we need to collect all the N maps produced by each of the N MPI processes running. It is just a matter of calling the MPI `reduce` function on the pixels of the local binned and hit maps, and then normalize the binned map using the hit map:

31b ⟨*Compress the TODs and produce reduced TODs* 31a⟩+≡ (6) ◁ 31a

```
Log(Format('Reducing %d binned values…', [Length(BinnedMap.Pixels)]));
Mpi.Reduce(BinnedMap.Pixels, OverallBinnedMap.Pixels, MPI_SUM, 0, Mpi.World);
Log(Format('Reducing %d hit count pixels…', [Length(BinnedMap.Pixels)]));
Mpi.Reduce(HitMap.Pixels, OverallHitMap.Pixels, MPI_SUM, 0, Mpi.World);

if MpiRank = 0 then
begin
    for Idx := 0 to Length(BinnedMap.Pixels) - 1 do
    begin
        if OverallHitMap.Pixels[Idx] > 0 then
            OverallBinnedMap.Pixels[Idx] :=
                OverallBinnedMap.Pixels[Idx] / OverallHitMap.Pixels[Idx]
        else
            OverallBinnedMap.Pixels[Idx] := Healpix.Unseen;
    end;
end;
```

Uses `Log` 32c.

The map is now ready to be saved. We use the following format:

31c ⟨*General-purpose constants* 31c⟩≡ (4 5)

```
    MapColumns : Array[1..2] of Cfitsio.TColumn =
        ((Name: 'AVGPHI'; Count: 1; DataType: FitsTypeFloat; UnitStr: ''),
         (Name: 'HITS';   Count: 1; DataType: FitsTypeLong;  UnitStr: ''));
```

Defines:
`MapColumns`, used in chunk 32a.

Now to the code for saving the map. The Healpix unit already provides a WriteHealpixMap procedure. However, this can be used only for saving *one* column, while here we're interested in squeezing two columns in the same table. Fortunately, we do not have to write low-level code, as the function WriteHealpixKeywords already takes care of the burden of writing all the keywords needed by the Healpix standard (e.g., NSIDE) in the current HDU:

32a ⟨*Save the reduced TODs* 32a⟩≡ (6)

```
if MpiRank = 0 then
begin
    Log(Format('Writing binned map in %s…', [Configuration.OutputMapFileName]));
    try
        EnsurePathExists(Configuration.OutputMapFileName);
        FitsFile := Cfitsio.CreateFile(Configuration.OutputMapFileName, Overwrite);
        try
            Cfitsio.CreateTable(FitsFile, BinaryTable, 0, MapColumns, 'PHIMAP');
            Healpix.WriteHealpixKeywords(FitsFile, OverallBinnedMap);
            Cfitsio.WriteColumn(FitsFile, 1, 1, 1, OverallBinnedMap.Pixels);
            Cfitsio.WriteColumn(FitsFile, 2, 1, 1, OverallHitMap.Pixels);
        finally
            Cfitsio.CloseFile(FitsFile);
        end;
        Log('…done, map has been written successfully');
    except
        on E : Exception do Log('…error, unable to write the map: ' + E.Message);
    end;
end;
```

Uses Configuration 10d 10e, EnsurePathExists 33a, Log 32c, and MapColumns 31c.

The variable FitsFile is used both by phid and phisky, so we declare it in the common section of the var block:

32b ⟨*Variables used by* phid *and* phisky *in the main loop* 13b⟩+≡ (4 5) ◁28a

```
FitsFile : TFitsFile;
```

§ A. Logging support

32c ⟨*Basic functions* 32c⟩≡ (4 5) 33a▷

```
procedure Log(const Message : String);
var
    MpiRank : Integer;

begin
    MpiRank := Mpi.CommRank(Mpi.World);
    WriteLn(
                Format('[%s #%d] %s',
                       [FormatDateTime('YYYY/MM/DD tt', Now),
                        MpiRank,
                        Message]));
end;
```

Defines:
Log, used in chunks 12, 14, 18, 19, 21–24, 26–28, 30–32, and 40–42.

§ B. File utilities

A very handy function to have when the programs need to save a file is `EnsurePathExists`. It creates any missing directory in the path of its argument, which can either be a file name or a path itself.

33a ⟨*Basic functions* 32c⟩+≡ (4 5) ◁32c

```
procedure EnsurePathExists(const FileName : String);
var
    Path : String;

begin
    Path := ExtractFilePath(ExpandFileName(FileName));
    if not ForceDirectories(Path) then
        raise Exception.CreateFmt('Unable to create the path "%s" ' +
                                  '(needed to save file "%s")',
                                  [Path, FileName]);
end;
```

Defines:
`EnsurePathExists`, used in chunks 23, 24, 31a, and 32a.

§ C. Reading INI files

??

To implement the procedure `ReadConfiguration`, we must have several pieces of code at hand first. Free Pascal's large standard library provides the `TIniFile` class[7], which is a good starting point. But we also need some other low-level functions. First, let's implement the procedure `GetAllValuesFromSection`, which will be used to retrieve the list of ringsets from the sections `[Main ringsets]` and `[Side ringsets]`. There is no method in `TIniFile` to retrieve a list of values, so we use `TIniFile.ReadSection` to read all the *keys*, and then cycle over them to save its value into `ValueList`.

33b ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁30 34▷

```
procedure GetAllValuesFromSection(IniFile : TIniFile;
                                  SectionName : String;
                                  var ValueList : TStringArray);
var
    KeyList  : TStringList;
    Idx      : Integer;

begin
    KeyList := TStringList.Create;
    try
        IniFile.ReadSection(SectionName, KeyList);
        SetLength(ValueList, KeyList.Count);
        for Idx := 0 to KeyList.Count - 1 do
            ValueList[Idx] := (IniFile.ReadString(SectionName,
                                                  KeyList[Idx], ''));
    finally
        KeyList.Free;
    end;
end;
```

Defines:
`GetAllValuesFromSection`, used in chunks 34, 36, and 37.
Uses `TStringArray` 25b.

[7] `http://www.freepascal.org/docs-html/fcl/inifiles/tinifile.html`.

Parsing the list of pointing/temperature files is trickier, as in this case the user might either list all the file names (in this case we rely on `GetAllValuesFromSection`) or use the shorthand provided by `first_index/last_index`.

34 ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁33b 35▷

```
procedure GetSequenceOfFiles(IniFile : TIniFile;
                             const SectName : String;
                             var FileNames : TStringArray);
var
    FirstIdx, LastIdx, Idx : Integer;
    Template : String;

begin
    if IniFile.ValueExists(SectName, 'first_index') and
       IniFile.ValueExists(SectName, 'last_index') and
       IniFile.ValueExists(SectName, 'template') then
    begin
        FirstIdx := IniFile.ReadInteger(SectName, 'first_index', 0);
        LastIdx := IniFile.ReadInteger(SectName, 'last_index', 0);
        Template := IniFile.ReadString(SectName, 'template', '');

        SetLength(FileNames, LastIdx - FirstIdx + 1);
        for Idx := FirstIdx to LastIdx do
            FileNames[Idx - FirstIdx] := Format(Template, [Idx]);
    end else
        GetAllValuesFromSection(IniFile, SectName, FileNames);
end;
```

Defines:
`GetSequenceOfFiles`, used in chunks 36 and 37.

Uses `GetAllValuesFromSection` 33b and `TStringArray` 25b.

Another piece of code we need is a procedure which parses the comma-separated list of percentages associated to the `quantiles` key (under the `[Output]` section).

35 ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁34 38a▷

```
procedure GetPercentages(const InputStr : String;
                         out Percentages : TPercentageArray);
var
    PercStrList : TStringList;
    Code : Word;
    Idx : Integer;

begin
    PercStrList := TStringList.Create;
    try
        ExtractStrings([','], [' ', #9], PChar(InputStr), PercStrList);
        SetLength(Percentages, PercStrList.Count);
        for Idx := 0 to PercStrList.Count - 1 do
        begin
            Val(PercStrList[Idx], Percentages[Idx], Code);
            if Code <> 0 then
                raise ERangeError(Format('"%s" is not a percentage',
                                         [PercStrList[Idx]]));
        end;
    finally
        PercStrList.Free;
    end;
end;
```

Defines:
`GetPercentages`, used in chunks 36 and 37.
Uses `TPercentageArray` 25a.

Now we are ready to implement the function `ReadConfiguration` for `phid`. If the parameters of the solar CMB dipole are not provided, then the Planck 2014 parameters will be used.

36 ⟨*High-level functions for the* `phid` *program* 16a⟩+≡ (4) ◁23 38b▷

```
procedure ReadConfiguration(const FileName : String;
                            out Configuration : TPhiDConfiguration);
var
    IniFile : TIniFile;
    ListOfPercentages : String;
begin
    IniFile := TIniFile.Create(FileName);
    try
        with Configuration do
        begin
            GetAllValuesFromSection(IniFile, 'Main beam matrices', MbMatrixFileNames);
            GetAllValuesFromSection(IniFile, 'Sidelobe matrices', SlMatrixFileNames);

            GetSequenceOfFiles(IniFile, 'Pointings', PointingFileNames);

            SatelliteVelocityFileName :=
                IniFile.ReadString('Input', 'satellite_velocity_file', '');
            DipoleParams.DirTheta :=
                IniFile.ReadFloat('Input', 'dipole_dir_theta_ecl', 1.7656131194951572);
            DipoleParams.DirPhi :=
                IniFile.ReadFloat('Input', 'dipole_dir_phi_ecl', 2.9958896005735780);
            DipoleParams.SpeedMS :=
                IniFile.ReadFloat('Input', 'dipole_speed_m_s', 370082.2332);

            ListOfPercentages :=
                IniFile.ReadString('Output', 'quantiles', '25,50,75');
            GetPercentages(ListOfPercentages, Quantiles);
            QuantileTableFileName := IniFile.ReadString('Output', 'quantiles_file_name', '');
            Nside := IniFile.ReadInteger('Output', 'map_nside', 64);
            if not Healpix.IsNsideValid(Nside) then
                raise Exception.CreateFmt('Invalid NSIDE value (%d)',
                                          [Nside]);

            OutputMapFileName :=
                IniFile.ReadString('Output', 'output_file_name', 'phi_d.fits');

            SaveTods := ReadBoolean(IniFile, 'Output', 'save_tods', False);
            if SaveTods then
                TodFilePath :=
                    IniFile.ReadString('Output', 'tod_file_path', './');
        end;
    finally
        IniFile.Free;
    end;
end;
```

Defines:
`ReadConfiguration`, used in chunk 10c.

Uses `Configuration` 10d 10e, `GetAllValuesFromSection` 33b, `GetPercentages` 35, `GetSequenceOfFiles` 34, `ReadBoolean` 38a, and `TPhiDConfiguration` 7a.

And here follows the same procedure for `phisky`:

37 ⟨*High-level functions for the* `phisky` *program* 17⟩+≡ (5) ◁24 39a▷

```
procedure ReadConfiguration(const FileName : String;
                            out Configuration : TPhiSkyConfiguration);
var
    IniFile : TIniFile;
    ListOfPercentages : String;
begin
    IniFile := TIniFile.Create(FileName);
    try
        with Configuration do
        begin
            GetAllValuesFromSection(IniFile, 'Main ringsets', MbRingsetFileNames);
            GetAllValuesFromSection(IniFile, 'Side ringsets', SlRingsetFileNames);

            GetSequenceOfFiles(IniFile, 'Pointings', PointingFileNames);
            GetSequenceOfFiles(IniFile, 'Temperatures', TemperatureFileNames);
            if Length(PointingFileNames) <> Length(TemperatureFileNames) then
                raise Exception.CreateFmt('# of pointing/temperature files differ (%d/%d)',
                                          [Length(PointingFileNames),
                                           Length(TemperatureFileNames)]);

            QualityFlagMask := IniFile.ReadInteger('Input', 'quality_flag_mask', 6111248);

            InterpolationOrder :=
                IniFile.ReadInteger('Output', 'interpolation_order', 5);
            ListOfPercentages :=
                IniFile.ReadString('Output', 'quantiles', '25,50,75');
            GetPercentages(ListOfPercentages, Quantiles);
            QuantileTableFileName := IniFile.ReadString('Output', 'quantiles_file_name', '');
            Nside := IniFile.ReadInteger('Output', 'map_nside', 64);
            if not Healpix.IsNsideValid(Nside) then
                raise Exception.CreateFmt('Invalid NSIDE value (%d)',
                                          [Nside]);

            OutputMapFileName :=
                IniFile.ReadString('Output', 'output_file_name', 'phi_sky.fits');

            SaveTods := ReadBoolean(IniFile, 'Output', 'save_tods', False);
            if SaveTods then
                TodFilePath :=
                    IniFile.ReadString('Output', 'tod_file_path', './');
        end;
    finally
        IniFile.Free;
    end;
end;
```

Defines:
`ReadConfiguration`, used in chunk 10c.

Uses `Configuration` 10d 10e, `GetAllValuesFromSection` 33b, `GetPercentages` 35, `GetSequenceOfFiles` 34, `ReadBoolean` 38a, and `TPhiSkyConfiguration` 7b.

To parse booleans, `TIniFile` provides the `ReadBool` method, which is however quite limited (it only accepts `0`, which stands for `false`, and `1`). So we provide a friendlier alternative in the `ReadBoolean` function, which we used above to parse the value of the key `save_tods` (in the `[Output]` section):

38a ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁35 39b▷

```
function ReadBoolean(const IniFile : TIniFile;
                     const SectName : String;
                     const KeyName : String;
                     DefaultValue : Boolean) : Boolean;
var
    DefaultValStr : String;
    Value : String;

begin
    if DefaultValue then DefaultValStr := 'true' else DefaultValStr := 'false';

    Value := UpperCase(IniFile.ReadString(SectName, KeyName,
                                          DefaultValStr));
    if (Value = 'TRUE') or (Value = 'YES') or (Value = 'ON') then
        Exit(True)
    else if (Value = 'FALSE') or (Value = 'NO') or (Value = 'OFF') then
        Exit(False);

    { Unable to understand what the user wants, so stick with the default }
    Exit(DefaultValue);
end;
```

Defines:
`ReadBoolean`, used in chunks 36 and 37.

C.1. Printing the configuration on the terminal. In this section we implement the function `PrintConfiguration` for both `phid` and `phisky`.

38b ⟨*High-level functions for the* `phid` *program* 16a⟩+≡ (4) ◁36 40b▷

```
procedure PrintConfiguration(const Configuration : TPhiDConfiguration);
begin
    with Configuration do
    begin
        WriteFileNames('[Pointings]', PointingFileNames);
        WriteFileNames('[Main beam matrices]', MbMatrixFileNames);
        WriteFileNames('[Sidelobe matrices]', SlMatrixFileNames);

        WriteLn('[Input]');
        WriteLn('satellite_velocity = ', SatelliteVelocityFileName);
        WriteLn;

        WriteLn('[Output]');
        WriteLn('nside = ', Nside);
        WriteLn('output_file_name = ', OutputMapFileName);
        WriteQuantiles('quantiles', Quantiles);
        WriteLn('quantiles_file_name', QuantileTableFileName);
        WriteLn('save_tods = ', SaveTods);
        if SaveTods then
            WriteLn('tod_file_path = ', TodFilePath);
    end;
end;
```

Defines:
`PrintConfiguration`, used in chunk 10c.
Uses `Configuration` 10d 10e, `TPhiDConfiguration` 7a, `WriteFileNames` 39b, and `WriteQuantiles` 40a.

39a ⟨*High-level functions for the* phisky *program* 17⟩+≡ (5) ◁37 41b▷

```
procedure PrintConfiguration(const Configuration : TPhiSkyConfiguration);
begin
    with Configuration do
    begin
        WriteFileNames('[Pointings]', PointingFileNames);
        WriteFileNames('[Temperatures]', TemperatureFileNames);
        WriteFileNames('[Main ringsets]', MbRingsetFileNames);
        WriteFileNames('[Side ringsets]', SlRingsetFileNames);

        WriteLn('[Output]');
        WriteLn('interpolation_order = ', InterpolationOrder);
        WriteLn('nside = ', Nside);
        WriteLn('output_file_name = ', OutputMapFileName);
        WriteQuantiles('quantiles', Quantiles);
        WriteLn('quantiles_file_name', QuantileTableFileName);
        WriteLn('save_tods = ', SaveTods);
        if SaveTods then
            WriteLn('tod_file_path = ', TodFilePath);
    end;
end;
```

Defines:
  PrintConfiguration, used in chunk 10c.
Uses Configuration 10d 10e, TPhiSkyConfiguration 7b, WriteFileNames 39b, and WriteQuantiles 40a.

The two implementations of PrintConfiguration use WriteFileNames to write a list of file names on the screen and WriteQuantiles to write the list of quantiles to use in producing the reduced TODs. The implementation of the former is fairly trivial:

39b ⟨*General functions (shared between* phisky *and* phid*)* 10b⟩+≡ (4 5) ◁38a 40a▷

```
procedure WriteFileNames(const SectName : String;
                         const NameList : TStringArray);
var
    Idx : Integer;

begin
    WriteLn(SectName);
    for Idx := 0 to Length(NameList) - 1 do
        WriteLn(Format('file_name_%.4d = %s',
                       [Idx, NameList[Idx]]));

    WriteLn;
end;
```

Defines:
  WriteFileNames, used in chunks 38b and 39a.
Uses TStringArray 25b.

The procedure `WriteQuantiles` writes a nicely formatted list of comma-separated percentages on the standard output. It is just called once, but moving this code out of `PrintConfiguration` improves the readability of the latter.

40a ⟨*General functions (shared between* `phisky` *and* `phid`*)* 10b⟩+≡ (4 5) ◁39b

```
procedure WriteQuantiles(const KeyName : String;
                         const Quantiles : TPercentageArray);
var
    Idx : Integer;

begin
    Write(KeyName, ' = ');
    for Idx := 0 to Length(Quantiles) - 1 do
    begin
        Write(Quantiles[Idx]);
        if Idx < High(Quantiles) then
            Write(', ')
        else
            WriteLn;
    end;
end;
```

Defines:
`WriteQuantiles`, used in chunks 38b and 39a.
Uses `TPercentageArray` 25a.

§ D. Other helper functions

D.1. Loading sets of convolution matrices. Here is a straightforward implementation of the procedure `LoadConvolutionMatrices`.

40b ⟨*High-level functions for the* `phid` *program* 16a⟩+≡ (4) ◁38b 41a▷

```
procedure LoadConvolutionMatrices(const FileNames : TStringArray;
                                  out Matrices : TConvMatrixArray);
var
    Idx : Integer;
    CurFileName : String;

begin
    SetLength(Matrices, Length(FileNames));
    for Idx := 0 to Length(FileNames) - 1 do
    begin
        CurFileName := FileNames[Idx];
        Log(Format('Reading file %s', [CurFileName]));
        LoadConvolutionMatrix(CurFileName, Matrices[Idx]);
    end;
end;
```

Defines:
`LoadConvolutionMatrices`, used in chunk 12b.
Uses `CurFileName` 22b, `Log` 32c, and `TStringArray` 25b.

41a ⟨*High-level functions for the* phid *program* 16a⟩+≡ (4) ◁40b

```
procedure SumConvMatricesIntensities(const Matrices : TConvMatrixArray;
                                     const SolSysVelEcl : TVector;
                                     const SatVel : TSatelliteVelocities;
                                     const Pointings : TDetectorPointings;
                                     var DestVector : TDoubleArray);
var
    MatIdx, Idx : Integer;

begin
    SetLength(DestVector, Length(Pointings.Theta));
    for Idx := 0 to Length(DestVector) - 1 do
        DestVector[Idx] := 0.0;

    for MatIdx := 0 to Length(Matrices) - 1 do
    begin
        Log(Format('Applying convolution matrix %d/%d', [Idx + 1, Length(Matrices)]));
        for Idx := 0 to Length(Pointings.Theta) - 1 do
            DestVector[Idx] := DestVector[Idx] +
                Convolve(Matrices[MatIdx], SolSysVelEcl, SatVel,
                         Pointings.ObtTimes[Idx],
                         Pointings.Theta[Idx], Pointings.Phi[Idx], Pointings.Psi[Idx]);
    end;
end;
```

Defines:
`SumConvMatricesIntensities`, used in chunk 18.
Uses `Log` 32c.

D.2. Loading sets of ringsets. Here we provide a straightforward implementation of the procedure `LoadRingsets`.

41b ⟨*High-level functions for the* phisky *program* 17⟩+≡ (5) ◁39a 42▷

```
procedure LoadRingsets(const FileNames : TStringArray;
                       InterpolationOrder : Integer;
                       out Ringsets : TRingsetArray);
var
    Idx : Integer;
    CurFileName : String;

begin
    SetLength(Ringsets, Length(FileNames));
    for Idx := 0 to Length(FileNames) - 1 do
    begin
        CurFileName := FileNames[Idx];
        Log(Format('Reading file %s', [CurFileName]));
        LoadRingsetFromFile(CurFileName,
                            InterpolationOrder,
                            Ringsets[Idx]);
    end;
end;
```

Defines:
`LoadRingsets`, used in chunk 12a.
Uses `CurFileName` 22b, `Log` 32c, and `TStringArray` 25b.

We need a function which sums all the intensities of a set of ringsets, too. This is provided by the `SumRingsetIntensities` procedure. It uses a callback, `UpdateUserAboutStatus`, to show how much of the TOD has been processed so far. This is useful, as the function `SumRingsetIntensities` is going to take a lot of time to complete when `phisky` is run on real data!

42 ⟨*High-level functions for the* `phisky` *program* 17⟩+≡ (5) ◁41b 43▷

```
procedure UpdateUserAboutStatus(Idx, NumOfElements : Integer; Data : Pointer);
var
    Percentage : Real;

begin
    if NumOfElements > 0 then
        Percentage := (Idx * 100.0) / (NumOfElements - 1)
    else
        Percentage := 100.0;

    Log(Format('   Ringset application: %d/%d (%.1f%%)',
               [Idx + 1, NumOfElements, Percentage]));
end;

procedure SumRingsetIntensities(const Ringsets : TRingsetArray;
                                const Pointings : TDetectorPointings;
                                var DestVector : TDoubleArray);
const
    LogUpdateDelayInMs = 30000.0; { = 30 s }
var
    Idx : Integer;
begin
    SetLength(DestVector, Length(Pointings.Theta));
    for Idx := 0 to Length(DestVector) - 1 do
        DestVector[Idx] := 0.0;

    for Idx := 0 to Length(Ringsets) - 1 do
    begin
        Log(Format('Applying ringset %d/%d', [Idx + 1, Length(Ringsets)]));
        GetRingsetIntensities(Ringsets[Idx], Pointings.Theta,
                              Pointings.Phi, Pointings.Psi,
                              DestVector, AddElements,
                              @UpdateUserAboutStatus, LogUpdateDelayInMs, nil);
    end;
end;
```

Defines:
`SumRingsetIntensities`, used in chunk 19.
Uses `Log` 32c.

We provide here an implementation of `MemoryUsed`, which is used by `phisky` to quantify how much memory is used by the ringsets. The unit `Ringsets` already provides a function `BytesUsed`, which returns the number of bytes used by a `TRingset` record. We wrap this into a more user-frendly function, which calculates the memory needed by *all* the ringsets in a `TPhiSkyInputData` record and the $T_{\rm sky}$ map, and formats the result using the proper measure unit.

43 ⟨*High-level functions for the* `phisky` *program* 17⟩+≡ (5) ◁42

```
function MemoryUsed(const InputData : TPhiSkyInputData) : String;
const
    Units : Array[1..4] of ShortString = ('bytes', 'KB', 'MB', 'GB');

var
    Size : Real = 0.0;
    FormattedNum : String;
    Idx, UnitIdx : Integer;

begin
    with InputData do
    begin
        for Idx := 0 to Length(MbRingsets) - 1 do
            Size := Size + BytesUsed(MbRingsets[Idx]);

        for Idx := 0 to Length(SlRingsets) -  1 do
            Size := Size + BytesUsed(SlRingsets[Idx]);
    end;

    UnitIdx := Low(Units);
    while (Size > 10 * 1024) and (UnitIdx < High(Units)) do
    begin
        Size := Size / 1024;
        Inc(UnitIdx);
    end;

    Str(Size:0:1, FormattedNum); { Just one digit after the separator }
    Result := FormattedNum + ' ' + Units[UnitIdx];
end;
```

Defines:
  `MemoryUsed`, used in chunk 12a.
Uses `TPhiSkyInputData` 11b.

References

Planck Collaboration. Planck 2013 results. V. LFI calibration. *ArXiv e-prints*, March 2013.

Benjamin D. Wandelt and Krzysztof M. Górski. Fast convolution on the sphere. *Phys. Rev. D*, 63:123002, May 2001. doi: 10.1103/PhysRevD.63.123002. URL `http://link.aps.org/doi/10.1103/PhysRevD.63.123002`.

§ E. 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.