

A Simulation Framework for the *LiteBIRD* Instruments

M. Tomasi,^{*1,2} L. Pagano,^{3,4,5} A. Anand,⁶ C. Baccigalupi,^{7,8,9}
 A. J. Banday,¹⁰ M. Bortolami,^{3,4} G. Galloni,^{3,6} M. Galloway,¹¹
 T. Ghigna,^{12,13} S. Giardiello,¹⁴ M. Gomes,¹⁵ E. Hivon,¹⁵
 N. Krachmalnicoff,^{7,8,9} S. Micheli,¹⁶ M. Monelli,¹³ Y. Nagano,¹⁷
 A. Novelli,¹⁶ G. Patanchon,^{18,19,13} D. Poletti,^{20,21} G. Puglisi,^{22,23,24}
 N. Raffuzzi,³ M. Reinecke,²⁵ Y. Takase,¹⁷ G. Weymann-Despres,^{26,27}
 D. Adak,²⁸ E. Allys,²⁹ J. Aumont,¹⁰ R. Aurvik,¹¹ M. Ballardini,^{3,4,30}
 R. B. Barreiro,³¹ N. Bartolo,^{32,33,34} S. Basak,³⁵ M. Bersanelli,^{1,2}
 A. Besnard,⁵ T. Brinckmann,³ E. Calabrese,¹⁴ P. Campeti,^{4,25,36}
 E. Carinos,¹⁰ A. Carones,^{7,8} F. J. Casas,³¹ K. Cheung,^{37,38,39,40}
 M. Citran,⁴¹ L. Clermont,⁴² F. Columbro,^{16,43} G. Coppi,²⁰
 A. Coppolecchia,^{16,43} F. Cuttaia,³⁰ P. Dal Bo,⁴⁴ P. de Bernardis,^{16,43}
 E. de la Hoz,⁴⁵ M. De Lucia,⁴⁴ S. Della Torre,²¹
 P. Diego-Palazuelos,²⁵ H. K. Eriksen,¹¹ T. Essinger-Hileman,⁴⁶
 C. Franceschet,^{1,2} U. Fuskeland,¹¹ M. Gerbino,⁴ M. Gervasi,^{20,21}
 C. Gimeno-Amo,³¹ E. Gjerløw,¹¹ A. Gruppuso,^{30,47}
 M. Hazumi,^{48,49,13,50} S. Henrot-Versillé,²⁷ L. T. Hergt,^{51,27} B. Jost,¹³
 K. Kohri,⁴⁸ L. Lamagna,^{16,43} T. Lari,⁴⁴ M. Lattanzi,⁴ C. Leloup,¹³
 F. Levrier,²⁹ A. I. Lonappan,⁵² M. López-Caniego,^{53,54} G. Luzzi,⁵⁵
 J. Macias-Perez,⁵⁶ B. Maffei,⁵ E. Martínez-González,³¹ S. Masi,^{16,43}
 S. Matarrese,^{32,33,34,57} T. Matsumura,¹³ L. Montier,¹⁰ G. Morgante,³⁰
 L. Mousset,^{29,10} R. Nagata,⁴⁹ F. Noviello,¹⁴ I. Obata,⁴⁸
 A. Occhiuzzi,¹⁶ A. Paiella,^{16,43} D. Paoletti,^{30,47}
 G. Pascual-Cisneros,^{13,31} F. Piacentini,^{16,43} M. Pinchera,⁴⁴
 G. Polenta,⁵⁵ L. Porcelli,⁵⁸ M. Remazeilles,³¹ A. Ritacco,⁵⁶
 A. Rizzieri,^{26,41} J. A. Rubiño-Martín,^{28,59} M. Ruiz-Granda,^{31,60}
 J. Sanghavi,^{61,62} V. Sauvage,⁵ M. Shiraishi,⁶³ G. Signorelli,^{64,44}
 S. L. Stever,^{5,17,13} R. M. Sullivan,¹¹ K. Tassis,^{65,66} L. Terenzi,³⁰
 L. Vacher,⁷ B. van Tent,²⁷ P. Vielva,³¹ I. K. Wehus,¹¹
 M. Zannoni,^{20,21} and Y. Zhou¹²
 LiteBIRD Collaboration.

*Corresponding author.

- ¹Dipartimento di Fisica, Università degli Studi di Milano, Via Celoria 16 - 20133, Milano, Italy
- ²INFN Sezione di Milano, Via Celoria 16 - 20133, Milano, Italy
- ³Dipartimento di Fisica e Scienze della Terra, Università di Ferrara, Via Saragat 1, 44122 Ferrara, Italy
- ⁴INFN Sezione di Ferrara, Via Saragat 1, 44122 Ferrara, Italy
- ⁵Université Paris-Saclay, CNRS, Institut d'Astrophysique Spatiale, 91405, Orsay, France
- ⁶Dipartimento di Fisica, Università di Roma Tor Vergata, Via della Ricerca Scientifica, 1, 00133, Roma, Italy
- ⁷International School for Advanced Studies (SISSA), Via Bonomea 265, 34136, Trieste, Italy
- ⁸INFN Sezione di Trieste, via Valerio 2, 34127 Trieste, Italy
- ⁹IFPU, Via Beirut, 2, 34151 Grignano, Trieste, Italy
- ¹⁰IRAP, Université de Toulouse, CNRS, CNES, UPS, Toulouse, France
- ¹¹Institute of Theoretical Astrophysics, University of Oslo, Blindern, Oslo, Norway
- ¹²International Center for Quantum-field Measurement Systems for Studies of the Universe and Particles (QUP), High Energy Accelerator Research Organization (KEK), Tsukuba, Ibaraki 305-0801, Japan
- ¹³Kavli Institute for the Physics and Mathematics of the Universe (Kavli IPMU, WPI), UTIAS, The University of Tokyo, Kashiwa, Chiba 277-8583, Japan
- ¹⁴School of Physics and Astronomy, Cardiff University, Cardiff CF24 3AA, UK
- ¹⁵Institut d'Astrophysique de Paris, CNRS/Sorbonne Université, Paris, France
- ¹⁶Dipartimento di Fisica, Università La Sapienza, P. le A. Moro 2, Roma, Italy
- ¹⁷Okayama University, Department of Physics, Okayama 700-8530, Japan
- ¹⁸ILANCE, CNRS – University of Tokyo International Research Laboratory, Kashiwa, Chiba 277-8582, Japan
- ¹⁹Université Paris Cité, F-75006 Paris, France
- ²⁰University of Milano Bicocca, Physics Department, p.zza della Scienza, 3, 20126 Milan, Italy
- ²¹INFN Sezione Milano Bicocca, Piazza della Scienza, 3, 20126 Milano, Italy
- ²²Dipartimento di Fisica e Astronomia, Università degli Studi di Catania, Via S. Sofia, 64, 95123, Catania, Italy
- ²³INAF, Osservatorio Astrofisico di Catania, via S. Sofia 78, I-95123 Catania, Italy
- ²⁴INFN, Sezione di Catania, via S. Sofia 64, I-95123, Catania, Italy
- ²⁵Max Planck Institute for Astrophysics, Karl-Schwarzschild-Str. 1, D-85748 Garching, Germany
- ²⁶Department of Physics, University of Oxford, Denys Wilkinson Building, Keble Road, Oxford OX1 3RH, UK
- ²⁷Université Paris-Saclay, CNRS/IN2P3, IJCLab, 91405 Orsay, France
- ²⁸Instituto de Astrofísica de Canarias, E-38200 La Laguna, Tenerife, Canary Islands, Spain
- ²⁹Laboratoire de Physique de l'École Normale Supérieure, ENS, Université PSL, CNRS, Sorbonne Université, Université de Paris, 75005 Paris, France
- ³⁰INAF - OAS Bologna, via Piero Gobetti, 93/3, 40129 Bologna, Italy
- ³¹Instituto de Física de Cantabria (IFCA, CSIC-UC), Avenida los Castros SN, 39005, Santander, Spain

- ³²Dipartimento di Fisica e Astronomia “G. Galilei”, Università degli Studi di Padova, via Marzolo 8, I-35131 Padova, Italy
- ³³INFN Sezione di Padova, via Marzolo 8, I-35131, Padova, Italy
- ³⁴INAF, Osservatorio Astronomico di Padova, Vicolo dell’Osservatorio 5, I-35122, Padova, Italy
- ³⁵School of Physics, Indian Institute of Science Education and Research Thiruvananthapuram, Maruthamala PO, Vithura, Thiruvananthapuram 695551, Kerala, India
- ³⁶Excellence Cluster ORIGINS, Boltzmannstr. 2, 85748 Garching, Germany
- ³⁷Jodrell Bank Centre for Astrophysics, Alan Turing Building, Department of Physics and Astronomy, School of Natural Sciences, The University of Manchester, Oxford Road, Manchester M13 9PL, UK
- ³⁸University of California, Berkeley, Department of Physics, Berkeley, CA 94720, USA
- ³⁹University of California, Berkeley, Space Sciences Laboratory, Berkeley, CA 94720, USA
- ⁴⁰Lawrence Berkeley National Laboratory (LBNL), Computational Cosmology Center, Berkeley, CA 94720, USA
- ⁴¹Université Paris Cité, CNRS, Astroparticule et Cosmologie, F-75013 Paris, France
- ⁴²Centre Spatial de Liège, Université de Liège, Avenue du Pré-Aily, 4031 Angleur, Belgium
- ⁴³INFN Sezione di Roma, P.le A. Moro 2, 00185 Roma, Italy
- ⁴⁴INFN Sezione di Pisa, Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
- ⁴⁵CNRS-UCB International Research Laboratory, Centre Pierre Binétruy, UMI2007, Berkeley, CA 94720, USA
- ⁴⁶NASA Goddard Space Flight Center, Greenbelt, MD 20771, USA
- ⁴⁷INFN Sezione di Bologna, Viale C. Berti Pichat, 6/2 – 40127 Bologna, Italy
- ⁴⁸Institute of Particle and Nuclear Studies (IPNS), High Energy Accelerator Research Organization (KEK), Tsukuba, Ibaraki 305-0801, Japan
- ⁴⁹Japan Aerospace Exploration Agency (JAXA), Institute of Space and Astronautical Science (ISAS), Sagamihara, Kanagawa 252-5210, Japan
- ⁵⁰The Graduate University for Advanced Studies (SOKENDAI), Miura District, Kanagawa 240-0115, Hayama, Japan
- ⁵¹Department of Physics and Astronomy, University of British Columbia, 6224 Agricultural Road, Vancouver, BC V6T1Z1, Canada
- ⁵²University of California, San Diego, Department of Physics, San Diego, CA 92093-0424, USA
- ⁵³Aurora Technology for the European Space Agency, Camino bajo del Castillo, s/n, Urbanización Villafranca del Castillo, Villanueva de la Cañada, Madrid, Spain
- ⁵⁴Universidad Europea de Madrid, 28670, Madrid, Spain
- ⁵⁵Space Science Data Center, Italian Space Agency, via del Politecnico, 00133, Roma, Italy
- ⁵⁶Université Grenoble Alpes, CNRS, LPSC-IN2P3, 53, avenue des Martyrs, 38000 Grenoble, France
- ⁵⁷Gran Sasso Science Institute (GSSI), Viale F. Crispi 7, I-67100, L’Aquila, Italy
- ⁵⁸Istituto Nazionale di Fisica Nucleare–Laboratori Nazionali di Frascati (INFN–LNF), Via E. Fermi 40, 00044, Frascati, Italy
- ⁵⁹Departamento de Astrofísica, Universidad de La Laguna (ULL), E-38206, La Laguna, Tenerife, Spain

⁶⁰Dpto. de Física Moderna, Universidad de Cantabria, Avda. los Castros s/n, E-39005 Santander, Spain

⁶¹Universitäts-Sternwarte, Fakultät für Physik, Ludwig-Maximilians Universität München, Scheinerstr.1, 81679 München, Germany

⁶²GRAPPA, Institute for Theoretical Physics Amsterdam, University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands

⁶³Suwa University of Science, Chino, Nagano 391-0292, Japan

⁶⁴Dipartimento di Fisica, Università di Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy

⁶⁵Institute of Astrophysics, Foundation for Research and Technology – Hellas, Vasilika Vou-
ton, GR-70013 Heraklion, Greece

⁶⁶Department of Physics and ITCP, University of Crete, GR-70013, Heraklion, Greece

E-mail: maurizio.tomasi@unimi.it

Abstract. *LiteBIRD*, the Lite (Light) satellite for the study of *B*-mode polarization and Inflation from cosmic background Radiation Detection, is a space mission focused on primordial cosmology and fundamental physics.

In this paper, we present the *LiteBIRD* Simulation Framework (LBS), a Python package designed for the implementation of pipelines that model the outputs of the data acquisition process from the three instruments on the *LiteBIRD* spacecraft: LFT (Low-Frequency Telescope), MFT (Mid-Frequency Telescope), and HFT (High-Frequency Telescope). LBS provides several modules to simulate the scanning strategy of the telescopes, the measurement of realistic polarized radiation coming from the sky (including the Cosmic Microwave Background itself, the Solar and Kinematic dipole, and the diffuse foregrounds emitted by the Galaxy), the generation of instrumental noise and the effect of systematic errors, like pointing wobbling, non-idealities in the Half-Wave Plate, *et cetera*.

Additionally, we present the implementation of a simple but complete pipeline that showcases the main features of LBS. We also discuss how we ensured that LBS lets people develop pipelines whose results are accurate and reproducible.

A full end-to-end pipeline has been developed using LBS to characterize the scientific performance of the *LiteBIRD* experiment. This pipeline and the results of the first simulation run are presented in Puglisi et al. (2025).

Contents

1	Introduction	2
2	Requirements	3
3	Overall design	4
3.1	Supported platforms	4
3.2	Memory layout	6
3.2.1	Scientific samples	6
3.2.2	Pointing information and Half-Wave Plate angles	8
3.2.3	I/O	9
3.2.4	Other data	9
3.3	Provenance tracking	10
4	Modules implemented in LBS	11
4.1	Simulation modules	11
4.1.1	Simulation of input maps	11
4.1.2	Scanning strategy	11
4.1.3	Instrumental noise	12
4.1.4	Solar dipole	13
4.1.5	Ideal HWPs	13
4.2	Data-reduction modules	14
5	Validation	14
5.1	Accuracy	14
5.2	Reliability	16
5.3	Reproducibility	17
5.4	Performance	18
6	A full example	18
6.1	Setting up the simulation	19
6.2	Accessing the IMo	20
6.3	Instantiating observations	23
6.4	Simulation of input maps	24
6.5	Scanning strategy	25
6.6	An HWP	25
6.7	Map scanning	25
6.8	CMB dipole	26
6.9	Noise generation	27
6.10	Map-making	27
6.11	TOD and map saving	28
7	Conclusions	28
A	Using Numba to optimize intensive computations	28

1 Introduction

LiteBIRD—the Lite (Light) satellite for the study of B -mode polarization and Inflation from cosmic background Radiation Detection—is a large-class satellite mission proposed by the Institute of Space and Astronautical Science (ISAS), Japan Aerospace Exploration Agency (JAXA), whose launch is planned for the 2030s. It will observe the full sky in 15 frequency bands from 34 to 448 GHz for 3 years, with effective polarization sensitivity of $2.2 \mu\text{K} \cdot \text{arcmin}$ and angular resolution of 31 arcmin (at 140 GHz), employing 4508 detectors sampling at 19.1 Hz [1]. Its main goal is to observe the polarization of the Cosmic Microwave Background (CMB) radiation, with the aim of testing the validity of the inflationary paradigm. The target sensitivity of *LiteBIRD* is $\delta r \leq 10^{-3}$, where δr is the uncertainty of the tensor-to-scalar ratio, r . This sensitivity will let us test major single-field inflation models.

LiteBIRD will host three instruments onboard the spacecraft: the Low-Frequency Telescope (LFT), the Mid-Frequency Telescope (MFT), and the High-Frequency Telescope (HFT). All the instruments employ Transition-Edge Sensor (TES) bolometers to measure the intensity and the polarization of the radiation from the sky, which will be focused on the focal planes of each instrument employing refractive and reflective telescopes.

To validate the design of the instruments onboard the spacecraft, the *LiteBIRD* collaboration has identified the need for a framework, called LBS, that provides a set of modules to model several aspects of the data acquisition process of the instruments, including the most relevant systematic effects. This framework is a Python library the collaboration has used to develop an end-to-end (E2E) *simulation pipeline* that is described in [2]; the purpose of the pipeline is to simulate the production of the time-ordered data that will be acquired by the actual instrument once deployed in space, to determine whether the design of the experiment can achieve its scientific objectives. Another important purpose of this pipeline is to produce output data that can be used as input by data-reduction pipelines; this is the case of the work described in [3], where the authors process the output of the E2E simulations using `Commander3` to estimate the amount of resources needed to perform a Bayesian end-to-end analysis of the *LiteBIRD* data. In this work, we describe the framework itself.

We decided to base the simulation pipeline on a framework instead of coding the whole E2E pipeline directly because this ensures a few advantages:

- The design phase of *LiteBIRD* needs several pipelines: apart from the E2E pipeline, the team requires specific pipelines to simulate targeted effects like Half-Wave Plate (HWP) systematics or the observation of transient sources, as well as simpler pipelines that produce approximated results in a fraction of the time needed by full simulations. Implementing a common framework speeds up the development, because each pipeline can be built by joining several ready-made building blocks.
- Reusing the same framework for many pipelines ensures consistency between them, particularly concerning the mathematical models used to describe the hardware and the file formats used to load input data and to save the results of the simulations.

The structure of this work is the following. In Section 2 we describe the requirements that have driven the implementation of LBS, including the memory layout and the need to properly track the provenance of the inputs to ensure that the results produced using LBS are reproducible. In Section 3 we show how LBS was implemented and what are its main characteristics. Section 4 provides a description of the most important modules that

are available in LBS 0.11.0, the version used to run the E2E tests described in [2]. We explain how we validated the implementation of LBS in Section 5. The last part of the paper, Section 6, provides the full implementation of a simple pipeline and showcases the main features of LBS that have been presented in the previous sections.

2 Requirements

The *LiteBIRD* Simulation Team was established in 2019 to develop a simulation framework for the collaboration. The following core requirements were established since the beginning:

- **Development of pipelines.** The framework must facilitate the development of simulation pipelines that can generate realistic timelines and maps. These outputs are crucial for validating the design of the scientific instruments and the overall mission architecture.
- **Computational efficiency.** Due to the limited computational resources available on high-performance computing (HPC) clusters, the framework must exhibit high performance in terms of memory utilization and execution time.
- **Availability of fundamental simulation components.** The framework must provide a comprehensive set of building blocks that can be assembled to build the simulation pipelines needed by the *LiteBIRD* collaboration.
- **Language Familiarity.** The framework must be implemented in a programming language that is widely adopted within the collaboration, as this minimizes the barrier to entry for new developers.
- **Ease of contribution.** The architecture of the framework and the development practices must be structured to enable reasonably straightforward contributions from collaboration members.
- **Comprehensive documentation.** This ensures maintainability and enables users to utilize the framework’s capabilities fully.
- **Reproducibility of results.** The framework must incorporate tools and mechanisms that enhance the ability to reproduce simulation results.
- **Integration with an Instrument Model database (IMo).** The framework must use a well-established way to retrieve its inputs from a common database containing a description of the instrument, which is handled by the *LiteBIRD* Instrument Model team (IMo team).

We now define the meaning of an *end-to-end (E2E) pipeline*, because it helps to understand the design of our framework better and puts this work in the context of what is described in [2].

The main scientific goal of *LiteBIRD* is to set an upper limit to the value δr , the uncertainty on the scalar-to-ratio parameter r . Thus, E2E simulations should go through the following steps:

1. Start from a reasonable estimate of the sky signal;

2. Model the data acquisition process of the detectors, considering the way the instruments are built and mounted within the optical system, the movements of the spacecraft, and other details of the mission;
3. The result of the simulation of data acquisition is a set of timelines, which are used to produce sky maps at different frequencies in the range 34–448 GHz;
4. Combine and process the sky maps using component-separation codes to produce estimates of the CMB as well as other signals (dust, synchrotron emission, etc.);
5. Compute power spectra from the CMB map;
6. Estimate r from the timelines, maps, and power spectra.

This list prompts us to make two crucial remarks. First, the data acquisition process is modeled only in the first *two steps* of the procedure: technically speaking, steps 3–6 are *data reduction tasks*, which are the same for simulated and real data. However, to ensure that the results of E2E simulations can be accurately interpreted and fed into the next stages of the analysis, we had to include a few data-reduction modules (map-makers) in our simulation framework, as explained in Section 4.2. The second remark is that N Monte Carlo realizations are needed to estimate error bars for the scientific parameters produced by E2E simulations. This increases the processing power required to run the simulations, necessitating the framework to be optimized for both speed and memory.

E2E simulations are not the only kind of simulations needed for an experiment of the scale of *LiteBIRD*. For instance, to characterize the ability to perform in-flight calibrations, the *LiteBIRD* collaboration has developed dedicated pipelines to simulate the observation of bright objects (planets). Thus, the list of modules to be included in the framework is not exhausted once all the modules needed in a E2E pipeline simulating nominal operations are implemented.

3 Overall design

Now that we have listed the requirements, in this section we present the architectural design of LBS and describe the elements that enable the simulation of the three instruments onboard the *LiteBIRD* spacecraft. The features listed in this section are used by all the simulation modules provided by LBS (see Section 4).

3.1 Supported platforms

LBS is implemented in Python and can be used on 64-bit Unix machines¹ like Linux and Mac OS X. In each release, we support Python versions whose End of Life (EoL) is more than one year in the future and that are supported by NumPy [4] and Numba [5]. We must support a range of versions, as the *LiteBIRD* collaboration runs simulations on many HPC clusters, and each of them might support different versions of the Python interpreter. Table 1 shows the list of versions that have been officially released at the time of writing. When selecting version numbers, we follow the rules of semantic versioning. Until version 1.0 is released, we

¹We are not able to support Windows systems natively because our code relies on Healpy, which is currently not available under this operating system (see <https://github.com/healpy/healpy/issues/25>). One can however install LBS within the Windows Subsystem for Linux (WSL).

increment the second number whenever changes in the codebase introduce new features, and we increment the third number if the new release only contains bug fixes. (This has been the case with versions 0.2.1 and 0.15.1.)

Table 1: List of LBS versions that have been officially released.

Version	Release date	Supported Python versions
0.15.1	June 2025	3.10-3.13
0.15.0	June 2025	3.10-3.13
0.14.0	February 2025	3.9-3.13
0.13.0	June 2024	3.9-3.12
0.12.0	March 2024	3.9-3.12
0.11.0	November 2023	3.9-3.12
0.10.0	June 2023	3.9-3.12
0.9.0	February 2023	3.7.1-3.9
0.8.0	October 2022	3.7.1-3.9
0.7.0	September 2022	3.7.1-3.9
0.6.0	July 2022	3.7.1-3.9
0.5.0	June 2022	3.7.1-3.9
0.4.0	December 2021	3.7.1-3.9
0.3.0	September 2021	3.6.1-3.8
0.2.1	March 2022	3.6-3.8
0.2.0	February 2022	3.6-3.8
0.1.0	September 2020	3.6-3.8

In this paper, we describe version 0.11.0, which is the version used to implement the E2E script described in [2].

We develop LBS on a public GitHub repository² and release it under an open-source license (GPL3³); we chose open-source solutions due to their advantages in terms of accessibility and cost. The documentation is hosted publicly⁴, and the manual is *complete*: the policy of the development team is to merge contributions only if they contain appropriate additions/modifications to the User’s manual and all the public functions and classes have their own docstrings. Moreover, the directory `notebooks`⁵ contains several Jupyter notebooks that show how to use the library to develop realistic pipelines.

We implemented LBS in Python, striking a balance between development efficiency and performance, as it offered a familiar and widely used language within the *LiteBIRD* collaboration. While other choices, such as C++, Fortran, or Julia, might offer some performance advantages, Python has so far enabled faster prototyping and broader accessibility across the team, while providing adequate performance. Our code heavily uses NumPy [4], to handle arrays and matrices, and employs Numba [5] for those most CPU-intensive tasks where we measured a distinct advantage over NumPy. Numba has proven crucial in maximizing the performance while keeping the amount of memory allocated by modules like the pointing

²https://github.com/litebird/litebird_sim

³<https://www.gnu.org/licenses/gpl-3.0.en.html>

⁴<https://litebird-sim.readthedocs.io/en/master/index.html>

⁵https://github.com/litebird/litebird_sim/tree/master/notebooks

simulator reasonable, as explained in Section A. Moreover, Numba makes code deployment straightforward, as one does not need to ensure the availability of a C/C++/Fortran compiler on the host system. We use AstroPy [6–8] to track time and perform coordinate conversions and Healpy [9] to handle sky maps in HEALPix format [10].

Since LBS is a library and not an executable, the user willing to run a simulation must first write a script that uses the library to perform the computations. Importing the library is easy, as it is listed on the Python Package Index⁶; thus, LBS can be installed using the command

```
pip install litebird_sim==0.11.0
```

where we specified version 0.11.0 because it is the subject of this paper. (Avoiding the version specification will install the most recent, which is 0.15.1 at the time of writing.)

3.2 Memory layout

LBS is designed to model the continuous data output of *LiteBIRD*’s onboard instruments over its nominal three-year duration. LBS does not simulate low-level hardware functionalities such as the response of the optical system to its components (mirrors, struts, etc.) or the propagation of thermal instabilities within the mechanical structures. By focusing on the timelines of scientific samples rather than hardware-level operations, LBS tries to achieve a balance between simulation fidelity and computational feasibility.

The most significant feat in simulating the output of a three-year space mission with thousands of detectors is to allocate sufficient RAM for all the data structures. In this section, we describe the way the framework manages data.

3.2.1 Scientific samples

A key functionality provided by LBS is the allocation of the so-called Time-Ordered Data⁷ (TOD). A TOD is stored as a matrix where each row represents the simulated output timeline of a single detector. Storing this matrix requires substantial memory, potentially exceeding the capacity of typical computer systems when simulating multiple detectors at once: each detector samples the input signal from the telescope at a frequency of 19 Hz using 32-bit floating-point numbers, thus requiring more than 6 GB per detector.

To overcome potential memory limitations and accelerate calculations, LBS fully supports MPI for distributing the TODs across multiple processes. Considering a TOD matrix of shape $N \times M$, where N is the number of detectors and M the number of samples, there are several ways to split it among k MPI processes:

- Splitting along the time axis results in each process holding a matrix of shape $N \times (M/k)$. This layout is optimal when the simulation needs to work on the timelines of multiple detectors at once. For instance, noise correlations among detectors on the same wafer can be simulated efficiently using a correlation matrix and Cholesky decomposition. This data layout avoids inter-process communication overhead if no wafer hosts more than M/k processes.

⁶https://pypi.org/project/litebird_sim

⁷LBS can also be used to run map-based simulations that do not need to simulate timelines. In fact, it has been used to develop map-based pipelines used internally by the *LiteBIRD* collaboration. These simulations are orders of magnitude faster to run but produce less realistic results.

- Splitting along the detector axis leads to a TOD matrix with a $(N/k) \times M$ shape. This layout is typically employed when the simulation pipeline needs to compute Fourier transforms of the timelines.
- More generally, one might want to split the number of detectors N and the number of time samples M simultaneously.

LBS lets the user to specify two parameters⁸: `n_blocks_det` and `n_blocks_time`, which define the number of splits for detectors and time samples, respectively. The default setting for both is 1, the only possible choice for serial applications where MPI is not employed. A visual depiction of the behavior of these two parameters is shown in Figure 1, which is taken from the LBS User’s Manual.

Providing the ability to specify the split layout during the initial allocation of the TODs might not be enough for complex simulations that require running modules in sequence with different requirements. For instance, users might want to simulate the presence of correlations between detectors *and* time correlations ($1/f$ noise) in the same script: in this case, there is no optimal split layout that can work with both simulation modules. To accommodate these cases, LBS lets the user change the data splits *after the TODs have been allocated*; this is done through the method `Observation.set_n_blocks()`, which accepts new values for `n_blocks_det` and `n_blocks_time` and reshuffles the samples in every TOD matrix across different MPI processes.

Another feature provided by LBS is the ability to work with multiple TOD matrices at once. When modeling data acquisition, it is often advisable to keep different signal components separate, such as the scientific signal and various noise contributions. Moreover, the sky signal itself typically consists of multiple astrophysical components, including the CMB, thermal emission from interstellar dust, and synchrotron emission from charged particles, among others. To track each component, LBS can create multiple 2D matrices per MPI process, ensuring that distinct contributions are represented individually, at the expense of increased memory usage.

Apart from the samples in a TOD, simulation modules often need additional information about the detectors being simulated. LBS can store an arbitrary number of attributes in memory; examples include the name of the wafer hosting the detector, the white noise level, the slope and knee frequency of the $1/f$ noise. When the user configures LBS to distribute the N detectors across separate MPI processes, the framework ensures that the relevant attributes for each detector are also distributed to their corresponding process, enabling each process to have the necessary data for simulations and calculations available without duplication or the need for inter-process communication. For instance, if two processes #1 and #2 simulate four detectors A, B, C, and D, setting `n_blocks_det=2` and `n_blocks_time=1` will make the parameters for detectors A and B available only on process #1. In contrast, the parameters for C and D will be present on process #2. The method `Observation.set_n_blocks()`, which we described above and lets to change the split layout of the TOD matrices, automatically redistributes the attributes too.

⁸Starting from version 0.14.0, LBS offers more sophisticated grouping options for N , enabling detectors to be grouped based on their hosting wafer or other attributes. However, this feature was not used in [2] and will not be discussed further here.

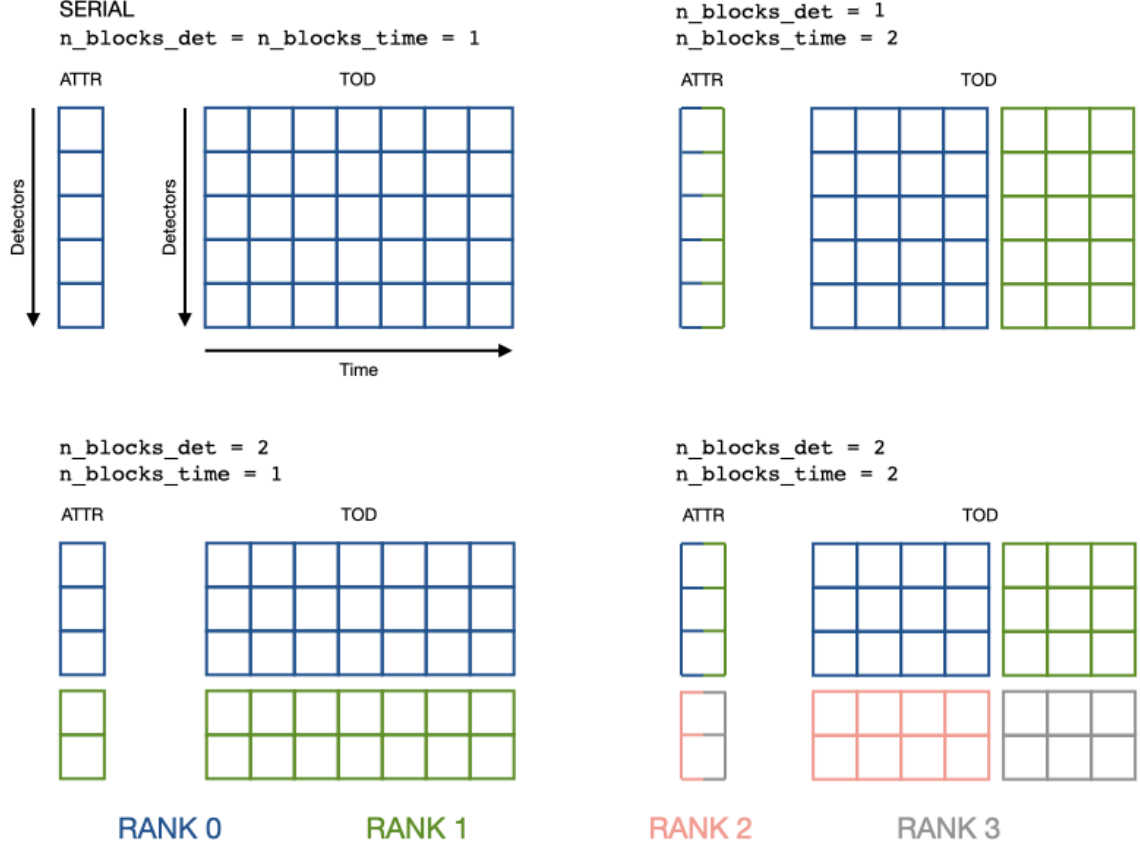


Figure 1. The way LBS can split a 2D matrix containing a TOD, as well as any other attribute associated with each detector, like the noise level, the FWHM of the beam, etc. The first row illustrates how splits are made when the code is run in serial mode, i.e., without MPI. The parameters `n_blocks_det` and `n_blocks_time` specify the number of splits between the detectors and along the time axis, respectively. The bottom row shows how TODs are split when there are four MPI processes. Attributes can be copied over many MPI processes if these processes handle the same detectors. (The latter is the case of the two panels on the right.)

3.2.2 Pointing information and Half-Wave Plate angles

Beyond TODs, the collection of pointing information also requires consistent memory allocations. Pointing information specifies the direction and orientation of each detector’s beam axis at the time when each sample was measured, and it is typically expressed through three angles: colatitude, longitude, and orientation (tilt). Consequently, the pointing matrix for N detectors and M time samples is a $N \times M \times 3$ array. For MPI applications, LBS stores⁹ these matrices using the same distribution strategy as the TOD matrices.

⁹As it is the case for CMB space surveys, the nominal scanning strategy is a composition of rotations with constant angular speed. In simulation codes, the typical approach is to encode these rotations as lowly-sampled quaternions, e.g., one per second or even less, depending on the angular speeds, and then use `slerp` operations to compute pointing angles at the same sampling frequency as scientific data. Since version 0.13.0, LBS can compute pointings for each detector on the fly from quaternions, thus avoiding the need to store the pointing matrices and significantly reducing memory usage.

In LBS 0.11.0, we used a simple scheme to encode the direction and orientation of the main beam of each detector as well as the angle of the HWP:

- The direction of the main beam was encoded by two angles (colatitude and longitude);
- The orientation and the Half-Wave Plate angle were summed together and saved as an angle, called the “polarization angle”.

The polarization angle is enough to simulate the behavior of a pencil beam and an ideal HWP, as under these assumptions, the map-maker has enough information to solve for I, Q, and U in each pixel. However, this approach provides insufficient information to simulate the systematic effects of non-ideal, asymmetric beams and realistic HWPs. For this reason, after version 0.11.0, we progressively implemented a more realistic model for pointings and HWP angles, which will be used in future simulation runs:

- Since version 0.12.0, LBS stores the HWP angle and the orientation as two distinct angles;
- As the previous change increased the memory occupation, since version 0.13.0, pointings can be computed on-the-fly;
- Starting from version 0.14.0, the polarization angle is no longer considered to be part of the orientation of the beam. This change is motivated by the integration of the 4π beam convolution code provided by the Ducc¹⁰ library (see Section 6.7), as 4π beams produced using electromagnetic simulation codes already encode the orientation of the polarization plane. (If no beam convolution is used in the code, it is still possible to tell LBS to add the polarization angle to the orientation angle when doing map-making, as it was the case before.)

3.2.3 I/O

One of the purposes of LBS is to produce simulated data to be fed into data reduction pipelines, thereby validating the latter. To enable this interaction, LBS provides tools that save the timelines and the pointing information in HDF5 files. In version 0.11.0, LBS ensures that every MPI process saves its TODs and pointings in separate files; thus, a program running on N MPI processes will save N files. To save disk space, the user can use a flag to instruct LBS to store the low-sampled quaternions used for computing the pointings instead of the full $N \times M \times 3$ pointing matrix.

HDF5 metadata are used extensively to save ancillary information; examples include the list of names of the simulated detectors and their nominal sampling rates, their angular resolution, and a human-readable description of the simulation, among others.

Saving of HDF5 files has been crucial to interface LBS with the Commander pipeline, as explained in [3].

3.2.4 Other data

Simulation modules often need to create new objects in memory, like sky maps or Fourier-transformed timelines. For MPI applications, modules usually allocate only those objects

¹⁰<https://gitlab.mpcdf.mpg.de/mtr/ducc>

relevant to the data chunk hosted by their TOD matrix. A notable exception is the map-making module: as the maps are typically created from the samples acquired by several detectors and the resolution¹¹ of *LiteBIRD*'s beams is not as high as for other CMB ground experiments, each MPI process receives a full copy of the final map.

3.3 Provenance tracking

Any simulation of a real scientific instrument needs several inputs that describe how the instrument is made and how it operates: what is the sampling frequency of the detectors, what are the characteristics of the noise profile, how many bits are used to measure one sample, what are the parts of the celestial sphere that are observed by the instruments during the nominal data acquisition, etc. This description is typically stored in a so-called Instrument Model (IMo), which is a database containing the details of each detector, optical system, hardware component, and so on. The *LiteBIRD* collaboration has implemented the IMo in a versioned database, where each quantity is referred through a path and a version number. The implementation of the database is provided by InstrumentDB¹², a Python program that manages information via a SQLite [11] database. InstrumentDB is more than a plain database, as it provides a programmatic interface over the web that can be accessed by software written in any language. InstrumentDB is kept on a remote server hosted at the ASI-SSDC¹³ (Agenzia Spaziale Italiana Space Science Data Center), whose access is restricted both for its web interface, which *LiteBIRD* members commonly access, and for its RESTful HTTP interface, which is usually used by scripts.

LBS can use the web interface to retrieve information from the database. However, as calls to a remote server can easily be a bottleneck for large-scale simulations that need a large number of inputs, InstrumentDB permits exporting the database to a folder, which can be copied to another computer and accessed locally by LBS itself. The latter is the preferred method for the LBS modules to fetch their inputs; in fact, this is the *only* way¹⁴ the E2E scripts described in [2] can operate.

The *LiteBIRD* collaboration maintains a comprehensive description of the design of the spacecraft and of the instruments in the SSDC database server, which is restricted to members of the *LiteBIRD* collaboration. However, the source code of LBS includes a reduced version of the database containing the basic design parameters that have already been published in [1]. This allows users outside the collaboration to use LBS to run simulations, with the caveat that these may not represent the most up-to-date design of the experiment. For instance, the reference to the table containing general facts about the LFT instrument might be stored in the object whose path is

```
/releases/v10.3/satellite/LFT/instrument_info
```

(assuming a hypothetical version 10.3 of the IMo), while the corresponding information taken from [1] is stored in the publicly available object with path

```
/releases/vPTEP/satellite/LFT/instrument_info
```

¹¹The highest angular resolution of *LiteBIRD* channels is 17.9 arcmin at 402 GHz, and the typical resolution of Healpix maps used in the *LiteBIRD* collaboration is NSIDE=512, which leads to 36 MB of storage space for a collection of three maps (I/Q/U).

¹²<https://github.com/ziotom78/instrumentdb>.

¹³<https://www.ssdsc.asi.it/>

¹⁴We purposefully prevented E2E scripts from querying the remote database to avoid saturating the network bandwidth at SSDC. Scripts based on LBS can access the remote database only if they send few requests per minute, otherwise the SSDC webserver will throttle the connection.

4 Modules implemented in LBS

To simulate complex instruments like those onboard the *LiteBIRD* spacecraft, multiple modules are required. Considering the task of simulating the nominal acquisition of the instruments, the following components are needed at a minimum:

- A flight simulator that models the way the spacecraft moves in space as time passes;
- An optical simulator that considers the way photons propagate from celestial sources (CMB, ISM, point sources, ...) to the telescope;
- An electronic simulator that simulates the way the signal captured by the optical system is measured and digitized by detectors;
- A noise simulator that injects realistic instrumental noise in the samples measured by the detectors;
- Etc.

We are not going to provide an exhaustive list of the modules implemented by LBS, as this list is still growing; the interested reader can refer to the User’s manual available at <https://litebird-sim.readthedocs.io>, which includes the documentation of *all* the modules¹⁵ implemented in LBS.

In this section we will describe some of the modules implemented in LBS 0.11.0; we will then use these modules in Section 6 to implement a simple pipeline.

For the sake of clarity, we group the modules in two sets: *simulation modules* and *data-reduction modules*. The formers have the task to simulate a process or the behaviour of some hardware, while the latters will be eventually replaced by the code that will be used to process the data acquired by the real instrument.

4.1 Simulation modules

4.1.1 Simulation of input maps

LBS provides the tools necessary to produce synthetic maps of the CMB and foreground sky at a specific observation frequency. These maps can then be observed through a simulation of the way the spacecraft spins, as discussed in more detail in Section 6.5. The production of synthetic sky maps is performed through the PySM3 [12, 13] library, which is conveniently wrapped by the `Mbs` module, whose acronym stands for Map-Based Simulation. Therefore, LBS supports all the models provided by PySM3.

4.1.2 Scanning strategy

LBS can simulate the orbit of the spacecraft and compute the directions where each detector is looking at the sky as a function of time, i.e., the *pointing information*. The *LiteBIRD* spacecraft will scan the sky spinning around its spin axis while precessing around the Sun-Earth direction and orbiting around the Second Lagrangian point of the Sun-Earth system. The details of the scanning strategy are described in [1]. Here, we provide some basic facts that help to understand how LBS produces the pointing information.

¹⁵As stated in Section 3, the development team has the rule that no new module is integrated in the codebase if it is not fully documented in the User’s Manual.

The most important thing to stress is that *LiteBIRD* will perform a *survey* of the sky, which means that the spacecraft will perform a set of periodic, continuous movements instead of pointing the telescope at several targets in sequence. The movements simulated by LBS are the composition of a set of rotations:

- The spacecraft rotates around its spin axis at a constant angular speed;
- The spin axis of the spacecraft rotates around the Sun-Earth axis at a constant angular speed;
- The Sun-Earth axis rotates around the Sun with a period of 365 days.

Of the three rotations, only the last one is not performed at a constant angular speed; instead, LBS uses AstroPy [8] to model the true revolution of the Earth around the Sun accurately¹⁶. To accelerate pointing generation, the code encodes rotations using quaternions sampled at a tunable frequency that is typically lower than the detectors' sampling frequency, and then it employs a *slerp* operation to compute the timelines of pointings at the nominal sampling rate.

4.1.3 Instrumental noise

A realistic simulation of any instrument must include a noise component. Currently, LBS permits simulations of white noise and correlated noise with a $1/f$ shape. There is no facility yet to introduce correlations in the noise between different detectors; however, this can be quickly implemented by multiplying the matrix containing the noise timelines by a proper correlation matrix. The power $P(f)$ of the noise for each detector as a function of the frequency f is modeled by the following equation:

$$P(f) = \sigma^2 \left(1 + \left(\frac{f_k}{f + f_{\min}} \right)^\alpha \right), \quad (4.1)$$

where σ^2 is the power associated with white noise, f_k is the so-called *knee frequency* of the correlated noise, α is the *slope* of the $1/f$ component, and $f_{\min} \ll f_{\text{samp}}$ is a parameter that avoids the singularity at $f = 0$. The noise simulation module provided by LBS requires the values of the three parameters σ^2 , f_k , and α to be provided for each detector.

Creating realistic $1/f$ noise is technically challenging, as the amount of power associated with this component increases with the inverse of the frequency, and this means that there are correlations between samples separated by long time intervals. However, this prevents MPI-based simulations from splitting TODs along the time axis, unless some compromises are made on the correlation of the simulated noise timelines. There are noise generators that can produce noise with long correlations. Still, LBS 0.11.0 uses a simpler approach where white noise goes through a high-pass filter before being added to each $N \times M$ matrix containing the scientific signal. This means that the noise coherence is not preserved across MPI processes. This approach has been employed by other CMB-oriented frameworks, such

¹⁶LBS can simulate a constant-speed revolution of the Earth around the Sun, for those simulations where accuracy is not as important as the speed of execution of the code. Moreover, the module can compute the expected velocity of the spacecraft with respect to the Sun. This can be used to simulate the kinetic dipole resulting from the spacecraft's motion with respect to the Sun. For this purpose, LBS implements an orbit simulator that simulates a Lissajous orbit, similar to the one followed by other spacecrafts orbiting L_2 like WMAP and Planck.

as [14], and has been shown to be sufficient for many applications, as the slowest noise terms are typically suppressed¹⁷ as part of the pre-processing. The problem of simulating $1/f$ noise is a telling example of the difference between a mathematical model and a numerical model: even though the model expressed by (4.1) is simple, a precise numerical implementation is not. We foresee that in the future we will probably need to implement a more sophisticated noise generator that avoids this problem.

4.1.4 Solar dipole

LBS provides tools to simulate the signal associated with the relative velocity of the Solar System with respect to the CMB, i.e., the *solar dipole*¹⁸. The CMB dipole is caused by a Doppler shift of the frequencies observed while looking at the CMB blackbody spectrum, according to the formula

$$T(\vec{\beta}, \hat{n}) = \frac{T_0}{\gamma(1 - \vec{\beta} \cdot \hat{n})}, \quad (4.2)$$

where T_0 is the temperature in the rest frame of the CMB, $\vec{\beta} = \vec{v}/c$ is the dimensionless velocity vector, \hat{n} is the direction of the line of sight, and $\gamma = (1 - \vec{\beta} \cdot \vec{\beta})^{-1/2}$.

CMB experiments usually employ the linear thermodynamic temperature definition, where temperature differences $\Delta_1 T$ are related to the actual temperature difference ΔT by the relation

$$\Delta_1 T = \frac{T_0}{f(x)} \left(\frac{\text{BB}(T_0 + \Delta T)}{\text{BB}(T_0)} - 1 \right) = \frac{T_0}{f(x)} \left(\frac{\exp x - 1}{\exp \left(x \frac{T_0}{T_0 + \Delta T} \right) - 1} - 1 \right), \quad (4.3)$$

where $x = h\nu/k_B T$,

$$f(x) = \frac{x e^x}{e^x - 1}, \quad (4.4)$$

and $\text{BB}(\nu, T)$ is the spectral radiance of a black-body according to Planck's law:

$$\text{BB}(\nu, T) = \frac{2h\nu^3}{c^2} \frac{1}{e^{h\nu/k_B T} - 1} = \frac{2h\nu^3}{c^2} \frac{1}{e^x - 1}. \quad (4.5)$$

LBS implements several simplifications of the formula that we call “dipole models”; they are reported¹⁹ in the User's manual.

4.1.5 Ideal HWP

LiteBIRD implements Half-Wave Plates (HWPs) for each of its three instruments to rotate the polarization angle of the radiation entering the optical systems. In version 0.11.0, LBS simulates an ideal HWP, whose only effect is to alter the orientation angle of the detector, as described in Section 3.2.2. (More accurate simulators have been implemented in LBS 0.15.0.)

¹⁷Several techniques can be used to reduce low-frequency correlated noise, including relative gain calibration and map-making algorithms.

¹⁸LBS is able to simulate the kinematic dipole too. We leave the interested reader to the relevant chapter in the User's Manual: <https://litebird-sim.readthedocs.io/en/latest/dipole.html>.

¹⁹<https://litebird-sim.readthedocs.io/en/latest/dipole.html>

4.2 Data-reduction modules

Being a *simulation* framework, LBS should not include data-reduction modules. Nevertheless, we implemented a few map-makers in LBS, as there are cases where maps are often more straightforward to analyze than timelines: their most significant advantage is that they take significantly less space and are easier to visualize.

The LBS map-makers save maps in FITS files using the Healpix²⁰ [9, 10] pixelization scheme. The following map-makers can be used with LBS:

- An internal binner, i.e., a simple map-maker that assumes that only uncorrelated noise is present in the timelines.
- An internal destriper, i.e., a more advanced map-maker that can remove the effect of correlated instrumental noise from the timelines before producing a map.
- A wrapper around the TOAST2 destriper [14], which is an optional dependency: this map-maker is available only if the user installed TOAST2 alongside LBS.
- A function that exports the TOD to FITS files whose format is compatible with the Madam mapmaker [15].

5 Validation

We describe here the problem of *validation*, i.e., how to ensure that the pipeline produces correct outputs.

To validate a simulation software, it is necessary to ensure that the following requirements are met:

- The software must be *accurate*, i.e., the results must match the expected within some reasonable threshold;
- The software should be *reliable*: if the input parameters are wrong, it should signal it, and in any case it should warn the user about unexpected features in the simulated data.
- The results produced by the software should be *reproducible*, i.e., anybody who has access to the source code and to the input parameters should be able to get the same outputs.
- The code should be *performant*, both in terms of speed and resource occupation (memory, disk space, etc.).

5.1 Accuracy

LBS is not a *simulation software* but a framework. Thus, it is hard to determine whether its implementation is accurate enough or not, as an accuracy target is typically set for a specific pipeline. However, the modules in LBS can be validated so that their expected accuracy is enough for the kind of applications the framework is currently used in the collaboration. There are cases where we have implemented more than one algorithm for the same task:

²⁰<http://healpix.sourceforge.net>

(1) we have two pointing generators that trade between speed and accuracy, and (2) we implemented several ways to produce a map from a set of timelines.

In LBS, we ensure that algorithms are accurate by means of several sets of automatic tests:

- *Unit tests*, which test the correct behaviour and accuracy of single functions;
- *Integration tests*, which test the correctness and accuracy of single modules;
- *E2E tests*, which exercise multiple modules at once.

We discuss E2E tests in the companion paper [2], so in this section we will describe only the unit tests and integration tests.

Unit tests ensure that single functions work correctly. They typically call the function to test with some easy-to-understand parameters and check that the result is what is expected. (These tests are good for documentation purposes as well.) The following example shows the tests for the function `compute_pointing_and_polangle` (in `tests/test_scanning.py`), used to compute the pointing direction and the polarization angle out of a quaternion representing the orientation of the boresight of a detector:

```
def test_compute_pointing_and_polangle():
    quat = np.array(lbs.quat_rotation_y(np.pi / 2))
    result = np.empty(3)
    lbs.compute_pointing_and_polangle(result, quat)
    assert np.allclose(result, [np.pi / 2, 0.0, -np.pi / 2])

    # We stay along the same pointing, but we're rotating the detector
    # by 90°, so the polarization angle is the only number that
    # changes
    lbs.quat_left_multiply(quat, *lbs.quat_rotation_x(np.pi / 4))
    lbs.compute_pointing_and_polangle(result, quat)
    assert np.allclose(result, [np.pi / 2, 0.0, -np.pi / 4])
```

One of the limitations of unit tests is that the set of test inputs is necessarily far smaller than the overall set of possible inputs. Therefore, singularities and catastrophic cancellations that appear for non-trivial inputs can go undetected. A notable example in literature is [16], which discovered that in specific cases, the geometric algorithm implemented in [17] produced incorrect results due to catastrophic cancellation; this error went undetected because, on average, this cancellation produced detectably wrong results only occasionally, and this is the reason why the problem went unnoticed in the original paper. An effective way to detect this kind of errors is to implement random testing, which calls the same function repeatedly over many random inputs; these tests are effective when verifying result correctness is straightforward (as in the case explained by [17]). In LBS, an example is found in `tests/test_mapmaking.py`, where the function `test_cholesky_and_solve_random()` tests that the function `solve_cholesky` returns the solution of an equation of the form $Ax = v$, where A is a 3×3 Cholesky matrix²¹ stored in a custom format used by the map-makers.

²¹Cholesky matrices are used by the internal destriper to solve for the three Stokes components I , Q , and U of each pixel. Cholesky-based algorithms are advantageous in this context due to their low memory storage

The test compares the result with the solution of `numpy.linalg.solve` for a large number (1000 for LBS 0.11.0) of random Cholesky matrices.

Of course, unit tests are not enough to ensure that a simulation code is accurate and reliable, because errors can occur when combining several low-level functions into larger blocks. Integration tests verify that the simulation modules implemented in LBS work as expected by comparing the results expected by the analytical model with the actual results of the code. An example is found in `test/test_destriper.py`, which tests that the results of the destriper map-maker are accurate. The destriper implements the model presented in [18], where the destripping operation is represented through a linear operator that can be written in matrix form; however, the actual code avoids to build these matrices as they get huge in typical computations, so the numerical implementation of these equations must rely on more compact data structures. The tests implemented for the LBS destriper consider a minimal dataset consisting of only seven samples and observing a sky map of just 2 pixels; this case can be written in full matrix notation, and the test code checks that the results of the destriper with these inputs match the result obtained by simply inverting the destripping matrix. The test code is pretty long, as it amounts to roughly 1000 lines of code, because it includes unit tests, integration tests, and the code that builds the theoretical matrices and invert them.

5.2 Reliability

We have implemented several tests to ensure that the framework’s foundations behave as expected in various situations. Dedicated tests for MPI processes are implemented in a separate script (`test/test_mpi.py`), which checks the consistency of TOD/pointing matrices and attributes when different split layouts are used (see Section 3.2). These tests also verify the correct behavior of MPI-aware modules, such as the binner and the destriper.

In addition to tests, we have adopted a defensive approach in coding LBS and implemented several `asserts` in the code. These checks ensure the consistency of the parameters at each stage of the processing.

We list here only a few checks²² that the LBS modules perform:

- Incorrect splitting of the 2D matrix of samples across the MPI processes;
- Incorrect correspondence between the components of a TOD and the components to be assembled by the map-maker;
- Wrong order in the calls to modules (for instance, the removal of baselines from a TOD is called before destripping);
- The caller requests a frequency outside the range of a detector bandpass;
- Inconsistent inputs to the module that simulates the scanning strategy.

requirements, robustness, and high performance. We implemented a dedicated Cholesky solver tailored for 3×3 matrices, rather than using existing libraries, because we store the coefficients of the symmetric matrix in a custom data type that keeps the six nontrivial matrix coefficients in a single vector. Numba unrolls most of the loops in our code, and the performance of this routine is roughly 30 % more efficient than SciPy’s `cholesky` function.

²²At the time of writing, there are 128 assertions in the code, plus 63 `raise` statements that provide detailed messages when LBS detects problems in the input.

5.3 Reproducibility

Apart from accuracy and reliability, simulation codes must ensure that their results are reproducible. There are several reasons why a software code called more than once produces different outputs:

- The input parameters might be different²³ between different runs;
- The person who ran the code might have been unaware that the source code was modified between two runs;
- The code might depend on some hidden internal state, e.g., the presence of a file that has been updated between two runs, or the seed for the random number generator being initialized using the computer's clock.

It is generally impossible to ensure that none of these conditions occur, but LBS tries to minimize the likelihood of their happening.

First, when LBS is used in a script, it always generates a report in Markdown/HTML format in the output directory. This report includes several information that are useful for archival purposes:

- The date when the simulation was run;
- The version number of LBS;
- The hash of the most recent Git commit and the output of `git diff` (see below);
- Other information.

The fact that the report is saved together with the simulation outputs (raw timelines, maps, plots, etc.) ensures that these outputs are easy to reproduce for several reasons that we detail here.

LBS encourages people implementing pipelines to provide the input parameters of their simulation scripts as parameter files in TOML format. (See Sect. Section 3.3.) These files are always copied to the output directory, and thus they can be passed as inputs to a new run of the simulation scripts.

Of course, ensuring that input files are the same is not enough, because the *code* of the simulation program must be the same. It is often the case that people modify the code on the fly once they discover a bug but do not bump the version number! LBS provides a way to check source code consistency was not changed by assuming that the source code of the script that calls LBS is kept in a Git repository. LBS automatically saves the hash of the latest Git commit in the report saved in the output directory, allowing the user to verify whether two simulations were executed using the same commit or not. Unfortunately, it is often the case that programmers test the code before saving it in a commit; for this reason, if LBS detects that there are unsaved modifications in the code, it saves the output of `git diff` in the report as well.

Finally, there is the possibility that the code has some state that is not preserved across separate runs. There is no reliable solution that can work 100% of the time to prevent hidden

²³This happens typically when the program asks the user to type input parameters using an interactive prompt. LBS does not provide any facility to input data in this way.

state²⁴ from altering the results of a computation. However, a common source of confusion is the seed used to generate pseudo-random numbers, as it is often initialized to a value derived from the system clock. LBS requires that the parameter `random_seed` accepted by the constructor of the class `Simulation` be *always* provided. The seed can be `None` or an integer number; setting it to `None` uses the internal clock and thus prevents the reproducibility of the results; however, we fell the fact that `random_seed=None` must be spelled explicitly in the source code to enable this behaviour is a good-enough “red flag”. Instead, by setting `random_seed` to an `int`, the results of separate runs of a script will produce the same sequence of pseudo-random numbers. The seed can be changed by using `sim.init_random`, i.e., the same function used at the end of the `Simulation` constructor to set up the RNG:

```
sim.init_random(
    random_seed=6789,
)
```

When running a parallel script, the `Simulation` constructor will take care of providing each process with a (different) dedicated RNG that produces uncorrelated sequences. The results will be reproducible when running the same script using the same `random_seed` and the same number of MPI processes. If the user uses the same seed but a different number of processes, the results will be different; however, LBS will include a warning in the output report.

5.4 Performance

In the development of LBS, we have verified that existing codebases like TOAST2, the simulation pipelines used for Planck, and other map-makers have performance similar to pipelines developed in our implementation. We also monitored the memory usage and speed of the E2E script described in [2] and collaborated closely with its development team to optimize memory usage and reduce CPU time.

As Python is notoriously slow when implementing loops, we have utilized NumPy and Numba to accelerate execution. Numba is particularly well-suited for loops that iterate over large arrays. It can easily beat NumPy, as the broadcast operation implemented by the latter requires multiple loops. We show an example in Section A.

6 A full example

So far, the description of the LBS framework has been only theoretical. To better highlight the features of the code, in this section we implement a simple E2E pipeline step-by-step. The reader can test all the code snippets in this section, provided that LBS 0.11.0 has been installed²⁵ using `pip` and that the commands are executed in the same sequence they appear in this paper.

This example pipeline is not as polished as the official one described in the companion paper [2]. Here, our aim is to demonstrate the features of LBS and how its modules work together. The example will be split into several fragments that are meant to be read in the

²⁴In principle, using functional languages like Haskell or Clojure and purely functional data structures might prevent this kind of error. However, no language is perfectly functional and thus the problem would still be unsolved.

²⁵The reader is advised to create a virtual environment before installing the `litebird_sim` package and trying the commands listed in this work.

same order as they are presented in the paper. The interested reader can test the code on their computer, as the example was designed to be runnable on personal computers. We print the source code using colors to highlight the syntax, and if there is some output, we report it in black color under the code, like in the following example:

```
import litebird_sim as lbs
print(
    f"litebird_sim {lbs.__version__}"
)

litebird_sim 0.11.0
```

The code imports the LBS Python package, which is published on the Python Package Index under the name `litebird_sim`, and it prints the version number. In the code fragments we are going to present in the next sections, we always assume that the Python package has been imported under the name `lbs`.

6.1 Setting up the simulation

The center point of LBS is the `Simulation` class, which should be instantiated in any pipeline built using this framework. The class serves as a container for several analysis modules available to the user, and it tracks both the inputs provided by the user and the output data generated by the simulation itself. Several information about the simulation need to be shared among different modules: for instance, the launch date and the duration of the simulation impact both the code that allocates the TOD, because it needs to know how many samples to allocate in memory, and the code that simulates the presence of moving sources in the sky, because it requires to compute the ephemerides of the planets.

Here, we show how to set up a simulation that lasts one day and starts on January 1st, 2024. We provide both a name and a description, which will be included in the report that is generated automatically at the end of any simulation:

```
import astropy

sim = lbs.Simulation(
    base_path="./example",
    start_time=astropy.time.Time(
        "2024-01-01T00:00:00",
    ),
    duration_s=86_400.0,
    name="My simulation",
    description="My description",
    random_seed=12345,
    imo=lbs.Imo(
        flatfile_location=lbs.PTEP_IMO_LOCATION,
    ),
)
```

The exact meaning of each keyword is explained in the User's Manual²⁶; here we highlight a few points:

²⁶<https://litebird-sim.readthedocs.io/en/latest/>.

- Times are tracked using AstroPy²⁷.
- The parameter `duration_s`, which takes the length of the simulation, shows a feature that is used extensively in the code: each quantity associated with a measurement unit reports the unit itself as part of the name. (In this case, the time must be expressed in seconds, hence the trailing “_s”.) This makes the code easy to read and reduces the chance of making conversion errors.
- The `random_seed` parameter initializes an internal pseudo-random number generator based on PCG-64²⁸. If the program is distributed using MPI, the `Simulation` class properly creates independent pseudo-random generators for each MPI process starting from this seed.
- The `imo` parameter specifies where to look for the characteristics of the instrument. The constant `PTEP_IMO_LOCATION` refers to the data file containing a synthetic description of the instruments as provided in [1]. This is not the official IMo for *LiteBIRD*, but it has the advantage that it can be used freely, even by people who are not part of the collaboration.

The `imo` parameter is related to the way LBS tracks the provenance of the inputs for simulations, and its meaning will be explained in the next section.

6.2 Accessing the IMo

The constant `lbs.PTEP_IMO_LOCATION` in the code fragment shown in the previous section tells LBS that we will use the reduced IMo database in this example; in this way, any reader can run the code fragments in this paper, even if they do not have access to the (restricted) full *LiteBIRD* IMo database.

Here is the code needed to access information about LFT, one of its frequency channels (40 GHz), and two of its detectors:

```
# Get a general description of the LFT instrument
# (use the specification from the PTEP 2022 paper)
lft_file = sim.imo.query("/releases/vPTEP/satellite/LFT/instrument_info")
print(
    "The instrument {name} has {num} channels.".format(
        name=lft_file.metadata["name"],
        num=lft_file.metadata["number_of_channels"],
    )
)
```

The instrument LFT has 12 channels.

This short code fragment shows the simplest way to access information in the IMo. The `sim.imo` object is an instance of the `Imo` class, which represents either a connection to a remote database or a link to a local copy of the IMo, as it is the case here. The `query` method accepts a path to a resource in the database; here, the odd-looking string

²⁷<https://www.astropy.org/>.

²⁸https://numpy.org/doc/stable/reference/random/bit_generators/pcg64.html.

`/releases/vPTEP/satellite/LFT/instrument_info` uniquely indicates the kind of information we are accessing in the database. Specifically, the part `/releases/vPTEP` refers to the fact that we want a specific *version* of the quantity: the one that was described in the so-called “PTEP paper”, i.e., [1]. (InstrumentDB can keep different versions of the same quantity, and in fact the restricted database used by the *LiteBIRD* collaboration contains a more updated version of this quantity.) The remainder of the path-like string indicates that we are looking for a quantity named `instrument_info`, which is stored within a pseudo-folder named LFT. The result is an object that provides several fields through its `metadata` component; in the code fragment, we print the name of the instrument, “LFT” (kept under the key `name`) and the number of detector channels, 12 (kept under the key `number_of_channels`).

Importing quantities from the `metadata` field requires knowing their names, such as `name` or `number_of_channels` in the example above, which can make programming with LBS more challenging. For this reason, LBS provided a number of data classes like `InstrumentInfo`, `FreqChannelInfo`, `DetectorInfo`, etc., that allow writing more readable code. For instance, the code above can be rewritten using the class `InstrumentInfo` in the following way:

```
# Ask the InstrumentInfo class to decode the "metadata"
# of the quantity for us
lft_instrument = lbs.InstrumentInfo.from_imo(
    imo=sim.imo, url="/releases/vPTEP/satellite/LFT/instrument_info"
)

# Accessing the fields of "lft_instrument" is done through
# object fields, not Python dictionaries
print(
    "The instrument {name} has {num} channels (again)".format(
        name=lft_instrument.name,
        num=lft_instrument.number_of_channels,
    )
)
```

The instrument LFT has 12 channels (again).

We have found that using these classes instead of directly accessing keys in the `metadata` dictionary has significantly improved our productivity, as modern IDEs and editors can suggest completions for field names. For example, when typing `name=lft_instrument.[...]`, editors can trigger an auto-completion widget once the dot `.` has been typed and prompt for the available choices: `name`, `number_of_channels`, etc.

Every time we access the `IMo`, the `Simulation` object keeps track of our requests and stores them in a list that is saved alongside the output of the simulation. For instance, after the requests in the two code fragments above, the `sim` object contains the information that (1) we requested the `instrument_info` quantity for LFT, and (2) we used the value of the quantity associated with the version named `vPTEP`, i.e., the description of the object that was presented in [1]. This type of information is advantageous when archiving simulation outputs, as it clarifies what the inputs were and whether the results remain updated with the instrument’s current design.

We are now going to fetch information about two LFT detectors that will be used in the simulation we are developing. As these detectors belong to LFT, we must first inform

LBS that this is the instrument we will be simulating. In this way, LBS will know the proper orientation of the focal plane, as the orientations of the focal planes of LFT and MHFT are separated by 180° around the spin axis of the spacecraft. Setting the instrument is a matter of calling the method `Simulation.set_instrument`:

```
sim.set_instrument(lft_instrument)
```

The next step is to load information about the detectors. This task is similar to what we have done above for the instrument, but this time we fill a `DetectorInfo` object instead of `InstrumentInfo`, and the pathlike string referring to the detector obviously changes. For the sake of simplicity and to keep the amount of calculations reasonable, we only load two detectors:

```
# Get information about two 40 GHz detectors
det1 = lbs.DetectorInfo.from_imo(
    imo=sim.imo,
    url="/releases/vPTEP/satellite/LFT/"
        "L1-040/000_000_003_QA_040_T/detector_info",
)
print(
    "The NET of detector {det1_name} is {det1_net}  $\mu\text{K}\cdot\sqrt{s}$ ".format(
        det1_name=det1.name,
        det1_net=det1.net_ukrts,
    )
)

det2 = lbs.DetectorInfo.from_imo(
    imo=sim.imo,
    url="/releases/vPTEP/satellite/LFT/"
        "LFT/L1-040/000_000_003_QA_040_B/detector_info",
)

```

The NET of detector 000_000_003_QA_040_T is 114.63 $\mu\text{K}\cdot\sqrt{s}$

In the examples shown so far, we have hard-coded the paths to detectors and objects in the Python code. However, in real-world applications, it is advisable to separate the code from the specification of the input data. LBS permits specifying the detectors to simulate using external parameter files saved in TOML²⁹. The following TOML file contains the same information that we provided in the code fragments presented so far:

```
[simulation]
base_path = "./example"
start_time = "2030-01-01T00:00:00"
duration_s = 86400.0
name = "My simulation"
description = "My description"
random_seed = 12345
```

²⁹<https://toml.io/>.

```
[[detectors]]
detector_info_obj = "/releases/vPTEP/satellite/LFT/↵
    L1-040/000_000_003_QA_040_T/detector_info"
```

```
[[detectors]]
detector_info_obj = "/releases/vPTEP/satellite/LFT/↵
    L1-040/000_000_003_QA_040_B/detector_info"
```

This file can be passed to the constructor of the `Simulation` class instead of the parameters we used. This approach has a few advantages:

- All the parameters are kept in one file, and thus modifying the inputs is easier;
- LBS copies the TOML file in the folder where the output of the simulation is saved, which can thus be run again, passing the same TOML file.

So far, we have loaded the descriptions of two detectors into memory; the next step is to instantiate the memory to store the samples measured by the simulated instruments. These are kept in “observations”, which are the data structures that hold the timelines.

6.3 Instantiating observations

LBS creates a set of `Observation` objects, which are basically wrappers around the 2D matrices containing the scientific samples see Sect. 3.2, and spreads them among the processes according to a scheme that can be tuned by the caller. Any `Observation` object can handle several 2D matrices at once; each of them is represented by a `TodDescription` object, where the acronym TOD stands for Time-Ordered Data, and it is common jargon in the CMB world for denoting these 2D matrices.

The way we allocate `Observation` objects is through the method `create_observations` of the `Simulation` class. In the following example, we allocate two 2D matrices per each `Observation` object, where the first will contain the actual signal (the TOD), and the second will contain noise:

```
import numpy as np
sim.create_observations(
    # We are going to simulate the two detectors
    # we fetched before
    detectors=[det1, det2],
    # For the sake of simplicity, here we just
    # keep track of the sky signal and the noise;
    # more realistic simulations would split "tod"
    # into its components (CMB, dust, ...)
    tods=[
        lbs.TodDescription(
            name="tod",
            description="TOD",
            dtype=np.float64,
        ),
```

```

        lbs.TodDescription(
            name="noise",
            description="1/f+white noise",
            dtype=np.float32
        ),
    ],
)

```

(We use different datatypes for `tod` (64-bit floating point) and `noise` (32-bit floating point) for demonstration purpose.)

As we said above, if we ran our example using MPI, we could take advantage of the many processes by passing further arguments to `sim.create_observation`; these arguments specify how the timelines should be split among the MPI processes. For the sake of simplicity, we assume that this example is ran serially using just one computer. However, more realistic codes can take advantage of several data split layouts, where the global 2D matrix is either split along rows (different MPI processes handle different detectors), along columns (different MPI processes simulate different time chunks), or both.

6.4 Simulation of input maps

Here we use MBS to produce synthetic sky maps:

```

params = lbs.MbsParameters(
    # Resolution of the maps to create
    nside=32,
    # Include the CMB
    make_cmb=True,
    # Include the foregrounds
    make_fg=True,
    # List of foregrounds: synchrotron (model #0) and free-free (model #1)
    fg_models=["pysm_synch_0", "pysm_freefree_1"],
)

# We will simulate the sky as observed by the 40 GHz channels
channel = lbs.FreqChannelInfo.from_imo(
    imo=sim.imo,
    url="/releases/vPTEP/satellite/LFT/L1-040/channel_info",
)

mbs = lbs.Mbs(
    simulation=sim,
    parameters=params,
    channel_list=[channel],
)
input_maps = mbs.run_all()[0]

```

The input maps are stored using the Healpix pixelization scheme [10].

In the next section, we will describe how LBS simulates the scanning strategy and produces a simulation of the signal measurement from the synthetic sky stored in `input_maps`.

6.5 Scanning strategy

The scanning strategy is encoded in the IMo via a set of angles and angular speeds, and thus it can be quickly retrieved using the IMo API. The following code loads the nominal scanning strategy described in [1], computes the quaternions, and produces the pointing information, which is stored in the same `Observation` objects that were allocated by the call to `sim.create_observations` (see above).

```
sim.set_scanning_strategy(  
    lbs.SpiningScanningStrategy.from_imo(  
        imo=sim.imo, url="/releases/vPTEP/satellite/scanning_parameters"  
    )  
)  
  
sim.compute_pointings()
```

Each pointing is encoded as a set of three angles: the colatitude and longitude in Ecliptic coordinates (in radians), and the orientation angle of the detector projected in the same coordinate system; the latter is used to determine the amount of polarized light coming from the sky that is measured by the detector. It is possible to compute other parameters related to the spacecraft’s motion, but we will discuss them in Section 6.8, where we will describe how LBS simulates the CMB dipole signal.

6.6 An HWP

We include an ideal HWP using `IdealHWP`, which is a descendant of the class `HWP`:

```
sim.set_hwp(  
    # The number is arbitrary  
    lbs.IdealHWP(ang_speed_radpsec=1.2345),  
)
```

6.7 Map scanning

Once realistic sky maps have been produced (Section 4.1.1) and a scanning strategy is set (Section 6.5), it is possible to simulate the actual observation of the signal coming from the celestial sphere and detected by the bolometers once it has been focused by the optical system. At the moment, LBS is not able to simulate a convolution of the beam pattern of the optical system with the sky map, so the code assumes that the optical system is represented by perfect pencil beams³⁰ with a Full Width Half Maximum close to zero and no sidelobes; the axis of the pencil beam is aligned with the main axis of each beam.

Under the assumption of a pencil beam, scanning a map is just a matter of projecting the boresight direction of each detector on the celestial map simulated by PySM (Section 4.1.1) and storing the value of the corresponding pixel on the map in each sample in the timeline:

```
lbs.scan_map_in_observations(  
    sim.observations[0],
```

³⁰In LBS 0.15.0, we used a module in the Ducc library, called `totalconvolve`, that implements a high-speed algorithm for convolutions of functions over the sphere. With LBS 0.15.0, we can properly simulate the optical response of full 4π beam patterns.

```

    input_maps,
    input_map_in_galactic=False,
    component="tod", # Save the measurements in the "tod" 2D matrix
    interpolation="", # Do not interpolate pixels
)

```

The method `scan_map_in_observation` can optionally perform a linear interpolation on the value of the pixels on the map through the function `pixelfunc.get_interp_val`, implemented by Healpy.

6.8 CMB dipole

To calculate the dipole signal, LBS must simulate an additional piece of information: the spacecraft's orbit, position, and velocity. The implementation of this calculation takes into account both the revolution of the Earth around the Sun and the orbit of the spacecraft around the Second Lagrangean point of the Sun-Earth system; the latter can optionally simulate a Lissajous orbit, as it was the case for the WMAP and Planck spacecraft, but in this example we will not turn on this option.

The code that implements the calculation of the velocity of the spacecraft is the following:

```

orbit = lbs.SpacecraftOrbit(
    sim.observations[0].start_time,
)
pos_vel = lbs.spacecraft_pos_and_vel(
    orbit,
    sim.observations,
    delta_time_s=60.0,
)

```

The variable `orbit` contains all the details about the motion of the Earth with respect to the Sun and about the orbit of the spacecraft, while `pos_vel` is a matrix that contains the position and velocity of the spacecraft computed every minute (the parameter `delta_time_s`) for the whole period covered by the observations in `sim.observations`.

Once the variable `pos_vel` is ready, we can inject the dipole signal into the timelines with a few lines of code:

```

dipole_type = lbs.DipoleType.TOTAL_FROM_LIN_T
lbs.add_dipole_to_observations(
    sim.observations,
    pos_vel,
    dipole_type=dipole_type,
    component="tod",
)

```

The dipole signal is *added* to the timeline named `tod`, so the result of the scanning of the synthetic maps described in Section 6.7 that was added in `tod` too is not overwritten. We use the model `TOTAL_FROM_LIN_T` that is described in Section 4.1.4.

6.9 Noise generation

For the sake of simplicity, we will include white noise in our example, and we will save it into the matrix called `noise`:

```
lbs.noise.add_noise_to_observations(  
    sim.observations,  
    noise_type="white",  
    component="noise",  
    random=sim.random,  
)
```

We did not specify the value of the noise parameters σ^2 , f_k , and α used in (4.1), as these are available to the module through the `DetectorInfo` class, which was already provided to the `Observation` objects instantiated by `sim.create_observations`; see Section 6.3.

6.10 Map-making

In our example, we will use the internal destriper provided by LBS:

```
result = lbs.make_destriped_map(  
    nside=32,  
    obs=sim.observations,  
    params=lbs.DestriperParameters(),  
    components=["tod", "noise"],  
)
```

```
[2025-04-11 20:30:09,554 INFO MPI#0000] Destriper CG iteration 1/100, ←  
    stopping factor: 7.028e-08  
[2025-04-11 20:30:09,714 INFO MPI#0000] Destriper CG iteration 2/100, ←  
    stopping factor: 1.870e-08
```

The variable `result` is an instance of the class `DestriperResults`, and it includes the following quantities:

- The estimates of all the baselines of the $1/f$ noise component;
- The destriped I/Q/U maps;
- The hit map, where each pixel contains an integer number corresponding to the number of samples in the TOD that have been projected onto that pixel;
- The binned I/Q/U maps, computed under the assumption that all the $1/f$ baselines were zero (this is mostly useful to see the effect of the destriper);
- Details about the convergence of the conjugated gradient algorithm and the overall time spent to produce the maps.

6.11 TOD and map saving

The last part of our example saves the data produced by the simulation to disk. We have adopted the HDF5 format to save the timelines to disk, and the API to save the data is straightforward to call:

```
# Save the timelines to HDF5 files
lbs.io.write_observations(sim)
```

The HDF5 files contain both the timelines of samples and the complete pointing information, as well as other details about how the computation was split among MPI processes and other ancillary information that are useful for archival purposes.

As the maps produced by the destriper (Section 4.2) have been produced using the Healpix pixelization scheme, LBS employs the FITS format to save them; this ensures maximum compatibility with other codes and experiments:

```
sim.write_healpix_map("hit_map.fits", result.hit_map)
sim.write_healpix_map("binned_map.fits", result.binned_map)
sim.write_healpix_map("destriped_map.fits", result.destriped_map)

PosixPath("example/destriped_map.fits")
```

The LBS repository at https://github.com/litebird/litebird_sim contains a folder named `notebooks`, which includes several Jupyter notebooks that showcase various characteristics of the framework.

7 Conclusions

In this paper, we have presented the *LiteBIRD* Simulation Framework (LBS), a Python library used for simulating the operations of the three instruments (LFT, MFT, and HFT) onboard the *LiteBIRD* spacecraft.

LBS has been developed to provide a user-friendly yet robust framework. It implements several features to enhance the accuracy, reliability, and reproducibility of the results. These include the integration of fast Python libraries like NumPy and Numba, automated report generation, source code tracking, and versioned access to the instrument model.

LBS has successfully been used to conduct E2E simulations of the nominal data acquisition, with detailed results outlined in the companion paper [2].

A Using Numba to optimize intensive computations

In Section 3, we explained that LBS utilizes NumPy for most of the code, but employs Numba for the most CPU-intensive tasks. We have chosen to use Numba on a case-by-case basis, depending on whether the E2E scripts required excessive memory or CPU time to run. Numba provided significant performance boosts in modules like the scanning strategy simulator and the generation of dipole timelines.

To illustrate how Numba can accelerate numerical codes that operate on long vectors, we consider a simple example. We have three arrays a , b , and c , each containing 10^6 elements, and we want to compute the value $2a_i + \sin b_i / 3 \cos c_i$. The following code implements both calculations, which lead to equally correct results, as the printed numbers are the same:


```

import numpy as np
from numba import njit, prange

def numpy_calculation(a, b, c, result):
    result[:] = 2 * a + np.sin(b) / (3 * np.cos(c))

@njit(parallel=True)
def numba_calculation(a, b, c, result):
    for i in prange(len(result)):
        result[i] = 2 * a[i] + np.sin(b[i]) / (3 * np.cos(c[i]))

N = 1_000_000
a = np.random.rand(N)
b = np.random.rand(N)
c = np.random.rand(N)
result = np.empty(N)

numpy_calculation(a, b, c, result)
print("First items calculated by NumPy: ", result[0:3])
numba_calculation(a, b, c, result)
print("First items calculated by Numba: ", result[0:3])

First items calculated by NumPy: [1.19619723 1.30527138 0.2299876 ]
First items calculated by Numba: [1.19619723 1.30527138 0.2299876 ]

```

However, if we measure the time spent in the two functions `numpy_calculation()` and `numba_calculation()`, the latter is significantly faster:

```

from timeit import timeit

print("""NumPy: {:.4f} s
Numba: {:.4f} s
""".format(
    timeit(lambda: numpy_calculation(a, b, c, result), number=5),
    timeit(lambda: numba_calculation(a, b, c, result), number=5)))

NumPy: 0.0712 s
Numba: 0.0101 s

```

This result shows that Numba is roughly 6 times faster than NumPy. A critical detail that led to this result is that before calling `timeit` we called once `numba_calculation`: this triggered the Numba compiler, which compiled a machine-code version of the routine. This time is spent only once but it can affect benchmarks: if we avoided using `print()` in the example before, the result of `timeit` on `numba_calculation()` would have included the compilation time, and thus the difference would have been smaller. To fully exploit Numba's advantages, we only used it for code that is called repeatedly or where the amount of time

spent in processing is significantly larger than the time needed for Numba to compile the function. Examples include pointing generation and estimation of the dipole signal on all samples of a TOD, among others.

Acknowledgments

This work is supported in Japan by ISAS/JAXA for Pre-Phase A2 studies, by the acceleration program of JAXA research and development directorate, by the World Premier International Research Center Initiative (WPI) of MEXT, by the JSPS Core-to-Core Program of A. Advanced Research Networks, and by JSPS KAKENHI Grant Numbers JP15H05891, JP17H01115, and JP17H01125. The Canadian contribution is supported by the Canadian Space Agency. The French *LiteBIRD* phase A contribution is supported by the Centre National d'Etudes Spatiale (CNES), by the Centre National de la Recherche Scientifique (CNRS), and by the Commissariat à l'Energie Atomique (CEA). The German participation in *LiteBIRD* is supported in part by the Excellence Cluster ORIGINS, which is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (Grant No. EXC-2094 - 390783311). The Italian *LiteBIRD* phase A contribution is supported by the Italian Space Agency (ASI Grants No. 2020-9-HH.0 and 2016-24-H.1-2018), the National Institute for Nuclear Physics (INFN) and the National Institute for Astrophysics (INAF). Norwegian participation in *LiteBIRD* is supported by the Research Council of Norway (Grant No. 263011 and 351037) and has received funding from the European Research Council (ERC) under the Horizon 2020 Research and Innovation Programme (Grant agreement No. 772253, 819478, and 101141621). The Spanish *LiteBIRD* phase A contribution is supported by MCIN/AEI/10.13039/501100011033, project refs. PID2019-110610RB-C21, PID2020-120514GB-I00, PID2022-139223OB-C21, PID2023-150398NB-I00 (funded also by European Union NextGenerationEU/PRTR), and by MCIN/CDTI ICTP20210008 (funded also by EU FEDER funds). Funds that support contributions from Sweden come from the Swedish National Space Agency (SNSA/Rymdstyrelsen) and the Swedish Research Council (Reg. no. 2019-03959). The UK *LiteBIRD* contribution is supported by the UK Space Agency under grant reference ST/Y006003/1 - "LiteBIRD UK: A major UK contribution to the LiteBIRD mission - Phase1 (March 25)." The US contribution is supported by NASA grant no. 80NSSC18K0132.

We acknowledge the use of CINECA HPC resources from the *LiteBIRD* INFN project and the computing facilities provided by NERSC.

We acknowledge the use of Google Gemini 2.0 and Grammarly to generate suggestions for improving writing style and proofreading. No content generated by AI technologies has been presented as our own work.

Bibliography

- [1] LiteBIRD Collaboration, *Probing cosmic inflation with the LiteBIRD cosmic microwave background polarization survey*, *Progress of Theoretical and Experimental Physics* **2023** (2023) [042F01](#) [[2202.02773](#)].
- [2] Puglisi, Giuseppe and The LiteBIRD Simulation Production Team, *First release from an end-to-end litebird simulation pipeline*, *JCAP* (2025) submitted.
- [3] R. Aurvik, K.J. Andersen, R. Banerji, M. Bersanelli, S. Bertocco, M. Brilenkov et al., *On the computational feasibility of bayesian end-to-end analysis of litebird simulations within cosmoglobe*, *JCAP* (2025) submitted.

- [4] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau et al., *Array programming with NumPy*, *Nature* **585** (2020) 357.
- [5] S.K. Lam, A. Pitrou and S. Seibert, *Numba: A llvm-based python jit compiler*, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pp. 1–6, 2015.
- [6] Astropy Collaboration, T.P. Robitaille, E.J. Tollerud, P. Greenfield, M. Droettboom, E. Bray et al., *Astropy: A community Python package for astronomy*, *Astronomy & Astrophysics* **558** (2013) A33 [[1307.6212](#)].
- [7] Astropy Collaboration, A.M. Price-Whelan, B.M. Sipőcz, H.M. Günther, P.L. Lim, S.M. Crawford et al., *The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package*, *The Astronomical Journal* **156** (2018) 123 [[1801.02634](#)].
- [8] Astropy Collaboration, A.M. Price-Whelan, P.L. Lim, N. Earl, N. Starkman, L. Bradley et al., *The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package*, *The Astrophysical Journal* **935** (2022) 167 [[2206.14220](#)].
- [9] A. Zonca, L. Singer, D. Lenz, M. Reinecke, C. Rosset, E. Hivon et al., *healpy: equal area pixelization and spherical harmonics transforms for data on the sphere in python*, *Journal of Open Source Software* **4** (2019) 1298.
- [10] K.M. Górski, E. Hivon, A.J. Banday, B.D. Wandelt, F.K. Hansen, M. Reinecke et al., *HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere*, *The Astrophysical Journal* **622** (2005) 759 [[arXiv:astro-ph/0409513](#)].
- [11] R.D. Hipp, *SQLite*, 2020.
- [12] B. Thorne, J. Dunkley, D. Alonso and S. Næss, *The Python Sky Model: software for simulating the Galactic microwave sky*, *Monthly Notices of the Royal Astronomical Society* **469** (2017) 2821–2833.
- [13] A. Zonca, B. Thorne, N. Krachmalnicoff and J. Borrill, *The Python Sky Model 3 software*, *Journal of Open Source Software* **6** (2021) 3783.
- [14] T. Kisner, R. Keskitalo, A. Zonca, J.R. Madsen, J. Savarit, M. Tomasi et al., *Toast*, Oct., 2021. 10.5281/zenodo.5559597.
- [15] E. Keihänen, H. Kurki-Suonio and T. Poutanen, *MADAM- a map-making method for CMB experiments*, *Monthly Notices of the Royal Astronomical Society* **360** (2005) 390 [[astro-ph/0412517](#)].
- [16] T. Duff, J. Burgess, P. Christensen, C. Hery, A. Kensler, M. Liani et al., *Building an orthonormal basis, revisited*, *Journal of Computer Graphics Techniques (JCGT)* **6** (2017) 1.
- [17] J.R. Frisvad, *Building an orthonormal basis from a 3d unit vector without normalization*, *Journal of Graphics Tools* **16** (2012) 151 [[https://doi.org/10.1080/2165347X.2012.689606](#)].
- [18] Kurki-Suonio, H., Keihänen, E., Keskitalo, R., Poutanen, T., Sirviö, A.-S., Maino, D. et al., *Destriping cmb temperature and polarization maps*, *Astronomy & Astrophysics* **506** (2009) 1511.