litebird_sim tutorial

Marco Bortolami
Nicolò Raffuzzi

Okayama, 2023/04/05

Overview

- Introduction
- Installing the framework
- Tutorial
- Simulations
- Detectors, channels and Instruments
- Observations
- Map-making
- Synthetic sky maps
- Scanning strategy

Introduction

- The LiteBIRD Simulation Framework (litebird_sim) is a set of Python modules capable of simulating the instruments onboard the LiteBIRD spacecraft
- litebird_sim is a library: users should write scripts using the litebird_sim modules
- Constantly maintained by many members of the LiteBIRD collaboration
- GitHub repository
- Documentation

Installing the framework - regular user

1. Create a directory for your scripts, notebooks, etc.

```
mkdir -p ~/litebird && cd ~/litebird
```

2. Create a virtual environment

```
conda create -n lbs_env python=3.8
```

or with `virtualenv lbs_env`

or with `python3 -m venv lbs_env`

3. Activate the environment

```
conda activate lbs_env
```

or with `. lbs_env/bin/activate`

4. Install litebird_sim with pip

```
pip install litebird_sim
```

Installing the framework - developer

1. Create a directory for your scripts, notebooks, etc.

```
mkdir -p ~/litebird && cd ~/litebird
```

2. Create a virtual environment

```
conda create -n lbs_env python=3.8
```

or with `virtualenv lbs_env`

or with `python3 -m venv lbs_env`

3. Activate the environment

```
conda activate lbs_env
```

or with `. lbs_env/bin/activate`

4. Clone GitHub repository

```
git clone https://github.com/litebird/litebird_sim litebird_sim
```

5. Install litebird_sim

```
cd litebird_sim && pip install -e .
```

Installing the framework

In order to use MPI functions (parallelization):

1. Activate the environment (if you have not already)

`conda activate lbs_env` or `. lbs_env/bin/activate`

2. Install mpi4py

`pip install mpi4py`

if mpi4py installation fails, try:

```
$ sudo apt update
$ sudo apt-get install libopenmpi-dev
```

Tutorial - A «Hello world» example

1. Go in the directory you created for your scripts, notebooks, etc.
2. Activate the environment (if you have not already)
3. Create a file called `tut01.py`
4. Write:

```
import litebird_sim as lbs # import litebird_sim

print("Starting the program...")
sim = lbs.Simulation(base_path="./tut01") # create instance of Simulation
sim.append_to_report("Hello, world!") # add something to the report
sim.flush() # write report to disk
print("Done!")
```

5. Run the script

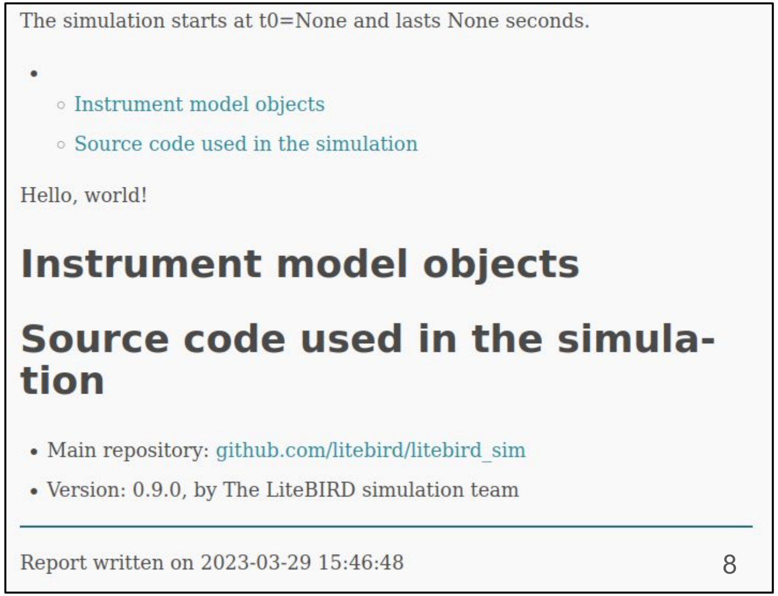
Here an empty report is created!

Tutorial - A «Hello world» example

The output is a folder named `tut01` containing some files:

```
$ ls ./tut01
report.html    report.md    sakura.css
```

Open the file `report.html` using your browser (e.g., `firefox tut01/report.html`)



See here and here for more info about reports

Tutorial - Interacting with the IMO

1. Real simulations use information stored in the Instrument MOdel database, a collection of information about the satellite
2. Download the IMO from this link or this link and run the following to configure the IMO (the name of the file should be `schema.json`)

```
python -m litebird_sim.install_imo
```

3. Choose «local copy» (option `3`) and specify the folder where the file `schema.json` resides (e.g. `…/litebird_imo/IMO`)
4. Provide a descriptive name, e.g. “IMO”
5. Save the changes with `s`

Now the Simulation class can read the database content!

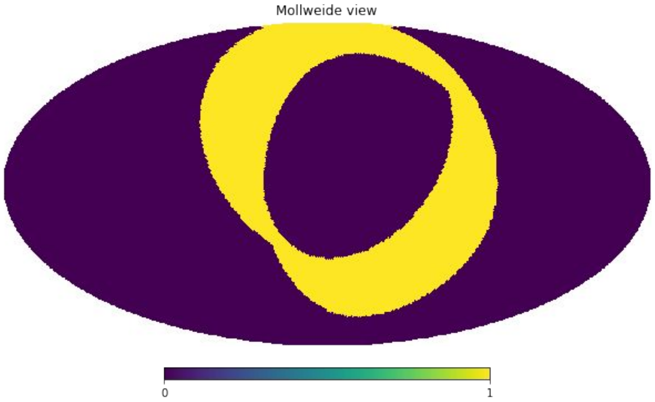
Tutorial - Interacting with the IMO

Let's build another report but including some IMO information

```
import litebird_sim as lbs
imo = lbs.Imo() # initialize IMO

sim = lbs.Simulation(base_path="./tut02")
lft_file = lbs.InstrumentInfo.from_imo( # get LFT info
    imo=imo,
    url="/releases/v1.3/satellite/LFT/instrument_info",
)
sim.append_to_report(
    "The instrument {{ name }} has {{ num }} channels.",
    name=lft_file.name,
    num=lft_file.number_of_channels,
)
sim.flush()
```

`lbs.InstrumentInfo.from_imo()` retrieves instrument info (such as LFT/MFT/HFT) and `append_to_report()` fills the report with info stored in the `DataFile` object `lft_file`

Tutorial - Creating a coverage map



Goal: compute the sky coverage of a scanning strategy over some time. Try to run the code at this link.

What the code does:

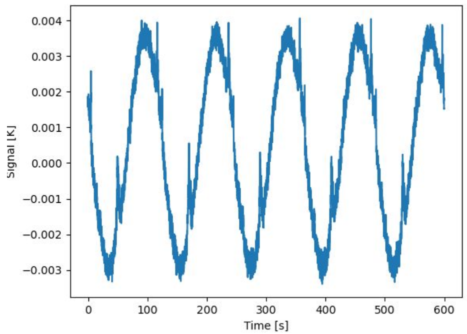
1. Define specific scanning strategy `sim.set_scanning_strategy(...)` and compute set of quaternions (orientation of the spacecraft)
2. Create instance of `InstrumentInfo` and register the quaternions using the method `sim.set_instrument()`
3. Add ideal HWP with `sim.set_hwp()`
4. Create observation `sim.create_observations()`, i.e. allocate TODs for the simulation
5. Compute pointings (direction of each detector)
6. Produce coverage map: all the visited pixels set to 1 based on the pointing information matrix

Tutorial - Creating a signal plus noise timeline

Goal: generate a 10 minutes timeline containing dipole, CMB, galactic dust, and correlated noise. Try to run the code at this link.

What the code does:

1. Create Simulation `sim = lbs.Simulation()`
2. Set scanning strategy `sim.set_scanning_strategy(...)`
3. Set instrument info with `lbs.InstrumentInfo()` and `sim.set_instrument()`
4. Add ideal HWP `sim.set_hwp()`
5. Choose detector parameters `lbs.DetectorInfo()`
6. Create CMB+fg map with `lbs.MbsParameters()`, `lbs.Mbs()` and `mbs.run all()`
7. Create observation `sim.create_observations()`
8. Compute pointings `sim.compute_pointings()`
9. Add dipole `sim.add_dipole()` and noise `sim.add_noise()`
10. Scan the map `sim.fill_tods()`
11. Write report with `sim.append_to_report()` and `sim.flush()`



Have a look at the report produced by the script!

Simulations - Handling parameters in a simulation (naively)

You can pass these parameters to the `Simulation` constructor:

1. base_path: path where to save the results of the simulation
2. start_time: start time of the simulation
3. duration_s: simulation duration (either a float, interpreted in seconds, or a string such as `"1 hour"`, `"60 min"`, `"3600 s"`)
4. name: a string containing the name of the simulation
5. description: a string containing a (possibly long) description of what the simulation does

```
sim = lbs.Simulation(
    base_path="./tut03",
    start_time=astropy.time.Time("2020-02-01T10:30:00"),
    duration_s=3600.0,
    name="My simulation",
    description="A long description should be put here"
)
```

Simulations - Handling parameters in a simulation (responsibly)

You can achieve the same with a TOML file (e.g. `foo.toml`) containing the following lines:

```
[simulation]
base_path = "./tut03"
start_time = 2020-02-01T10:30:00
duration_s = 3600.0
name = "My simulation"
description = "A long description should be put here"
```

Then you instantiate the `Simulation` as follows:

```
sim = lbs.Simulation(parameter_file="foo.toml")
```

Simulations - Handling parameters in a simulation (responsibly)

You can achieve the same with a TOML file (e.g. `foo.toml`) containing the following lines:

```
[simulation]
base_path = "./tut03"
start_time = 2020-02-01T10:30:00
duration_s = 3600.0
name = "My simulation"
description = "A long description should be put here"
```

Then you instantiate the `Simulation` as follows:

```
sim = lbs.Simulation(parameter_file="foo.toml")
```

You can also use other sets of parameters:

```
[general]
nside = 512
imo_version = "v1.3"

[sky_model]
components = ["synchrotron", "dust", "cmb"]
```

Try this and check the folder: a copy of the toml file is there!

```
import litebird_sim as lbs

sim = lbs.Simulation(parameter_file="foo.toml")

print("NSIDE =", sim.parameters["general"]["nside"])
print("The IMO I'm going to use is",
      sim.parameters["general"]["imo_version"])

print("Here are the sky components I'm going to simulate:")
for component in sim.parameters["sky_model"]["components"]:
    print("-", component)
```

Simulations - Handling parameters in a simulation (responsibly)

You can achieve the same with a TOML file (e.g. `foo.toml`) containing the following lines:

```
[simulation]
base_path = "./tut03"
start_time = 2020-02-01T10:30:00
duration_s = 3600.0
name = "My simulation"
description = "A long description should be put here"
```

Then you instantiate the `Simulation` as follows:

```
sim = lbs.Simulation(parameter_file="foo.toml")
```

Use this for parallelisation:

```
import litebird_sim as lbs
import mpi4py
sim = lbs.Simulation(parameter_file="foo.toml",
      mpi_comm=lbs.MPI_COMM_WORLD)

$ mpiexec -n 4 python my_script.py
```

You can also use other sets of parameters:

```
[general]
nside = 512
imo_version = "v1.3"

[sky_model]
components = ["synchrotron", "dust", "cmb"]
```

Try this and check the folder: a copy of the toml file is there!

```
import litebird_sim as lbs

sim = lbs.Simulation(parameter_file="foo.toml")

print("NSIDE =", sim.parameters["general"]["nside"])
print("The IMO I'm going to use is",
      sim.parameters["general"]["imo_version"])

print("Here are the sky components I'm going to simulate:")
for component in sim.parameters["sky_model"]["components"]:
        print("-", component)
```

Detectors, channels and instruments - hard coded

Detectors

```python
import litebird_sim as lbs

detectors = [
    lbs.DetectorInfo(
        name="Dummy detector #1",
        net_ukrts=100.0,
        bandcenter_ghz=123.4,
        bandwidth_ghz=5.6,
    ),
    lbs.DetectorInfo(
        name="Dummy detector #2",
        net_ukrts=110.0,
        fwhm_arcmin=65.8,
    ),
]

# Now simulate the behavior of the two detectors
# …
```

Three ways to include information:

1. Fill the information manually (*Missing information is usually initialized with zero or a sensible default*)
2. Read info from the IMO (next slides)
3. Read info from a file (next slides)

Detectors, channels and instruments - hard coded

Channels

Detectors are grouped according to their central frequency in *frequency channels*. The `FreqChannelInfo` class can produce a mock DetectorInfo object by taking out the «average» of the frequency channel:

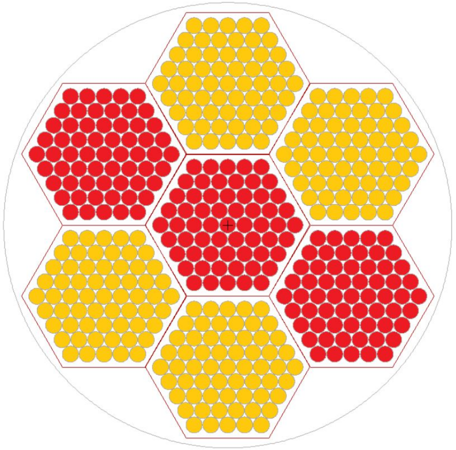
```
import litebird_sim as lbs

chinfo = lbs.FreqChannelInfo(
    bandcenter_ghz=40.0,
    net_channel_ukrts=40.0, # Taken from Sugai et al. 2020 (JLTP)
    number_of_detectors=64, # 32 pairs
)

# Return a "mock" detector that is representative of the
# frequency channel
mock_det = chinfo.get_boresight_detector(name="mydet")

# Look, ma, a detector!
assert isinstance(mock_det, lbs.DetectorInfo)
assert mock_det.name == "mydet"

print("The NET of one detector at 40 GHz is {0:.1f} μK·√s".format(mock_det.net_ukrts))
```

```
The NET of one detector at 40 GHz is 320.0 μK·√s
```

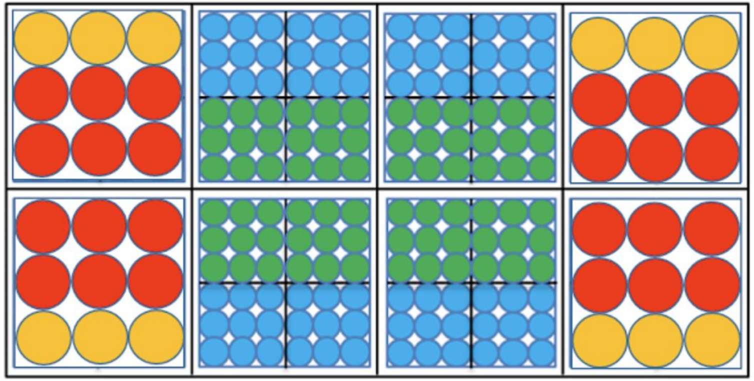
Detectors, channels and instruments - from IMO

IMO wiki here

```python
import litebird_sim as lbs

# You must have configured the IMO before using this!
imo = lbs.Imo()

det = lbs.DetectorInfo.from_imo(
    imo=imo,
    url="/releases/v1.3/satellite/LFT/L1-040/"
        "000_000_003_QA_040_T/detector_info",
)

freqch = lbs.FreqChannelInfo.from_imo(
    imo=imo,
    url="/releases/v1.3/satellite/LFT/L1-040/channel_info",
)
```



MFT

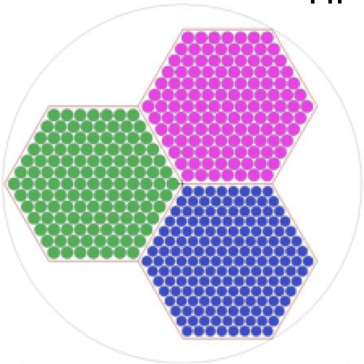


LFT

Reading info from the IMO is similar to the precedent cases

`DetectorInfo.from_imo()`
`FreqChannelInfo.from_imo()`

HFT



Detectors, channels and instruments - from file

"det_list1.toml"

```
[[detectors]] # First element: take the parameters for the detector from a mock IMO using UUID
detector_info_obj = "/data_files/78fe75f1-a011-44b6-86dd-445dc9634416"

[[detectors]] # Append a few more elements
# Take the parameters for the set of detectors from the channel information in the mock IMO with UUID
channel_info_obj = "/data_files/ff087ba3-d973-4dc3-b72b-b68abb979a90"
# Generate two detectors (default is to generate all the detectors in the frequency channel)
num_of_detectors_from_channel = 2

[[detectors]] # Append one more element
channel_info_obj = "/data_files/ff087ba3-d973-4dc3-b72b-b68abb979a90"
# Generate *one* mock detector associated with a specified frequency channel, aligned with the boresight
of the focal plane, and using the «average» detector parameters for this channel
use_only_one_boresight_detector = true
detector_name = "foo_boresight"

[[detectors]] # Add one last element, setting up every parameter manually
name = "planck30GHz"
channel = "30 GHz"
fwhm_arcmin = 33.10
fknee_mhz = 113.9
bandwidth_ghz = 9.89
bandcenter_ghz = 28.4
sampling_rate_hz = 32.5

[[detectors]] # Take the parameters from the IMO but fix one parameter by hand
detector_info_obj = "/data_files/78fe75f1-a011-44b6-86dd-445dc9634416"
sampling_rate_hz = 1.0   # Fix this parameter
```

There is a mock imo in `litebird_sim/test/mock_imo`

Detectors, channels and instruments - from file

"det_list1.toml"

Let’s load the mock IMO

```python
from pathlib import Path
import litebird_sim as lbs

# Load a mock IMO that actually defines the UUID listed above
imo = lbs.Imo(
    flatfile_location=Path(".").parent / ".." / "test" / "mock_imo",
)

# Tell Simulation to load the TOML file shown above
sim = lbs.Simulation(imo=imo, parameter_file="det_list1.toml")

det_list = lbs.detector_list_from_parameters(
    imo=sim.imo,
    definition_list=sim.parameters["detectors"],
)

for idx, det in enumerate(det_list):
    print(f"{idx + 1}. {det.name}: band center at {det.bandcenter_ghz} GHz")
```

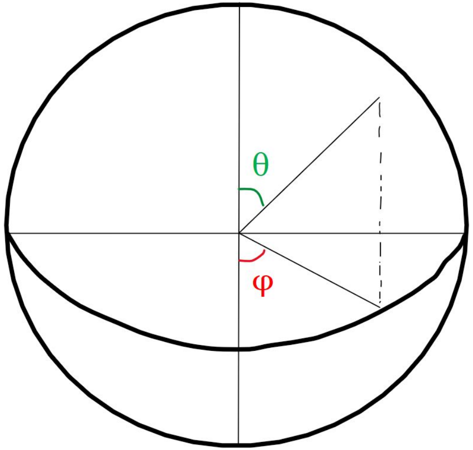
Observations - serial

Observation: data acquired by the telescope during a scanning period

```
sim.create_observations(
    detectors=[det1, det2, det3],
    tods=[lbs.TodDescription(name="tod",  description="TOD",            dtype=np.float64),
          lbs.TodDescription(name="noise",description="1/f+white noise",dtype=np.float32)]
)

for cur_obs in sim.observations:
    print("Shape of 'tod': ", cur_obs.tod.shape)
    print("Shape of 'noise': ", cur_obs.noise.shape)
```

Important fields:

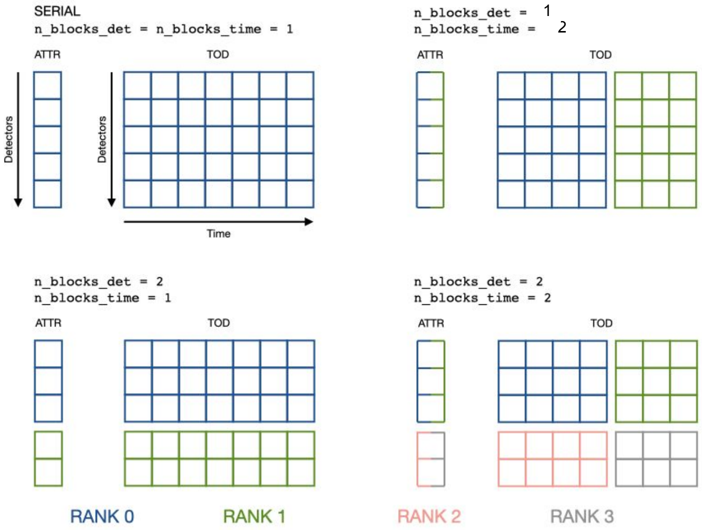
- `Observation.tod` with shape (N_{det}, N_{samp})
- `Observation.pointings` with shape $(N_{det}, N_{samp}, 2)$ for colatitude (θ) and longitude (φ)
- `Observation.pointing_coords` for **`CoordinateSystem`**
- `Observation.psi` with shape (N_{det}, N_{samp}) for polarization angles (ψ)



Observations - parallel

You can distribute the computation among different MPI ranks!
Detector attributes are divided accordingly

```python
import litebird_sim as lbs
from mpi4py import MPI

comm = MPI.COMM_WORLD

[...]

(obs_multitod,) = sim.create_observations(detectors=[det1, det2, det3],
                                          n_blocks_det=2, # Split detector axis in 2
                                          n_blocks_time=3) # Split time axis in 3
```




```python
descr = sim.describe_mpi_distribution()

print(descr)
```

```
# MPI rank #1

## Observation #0
- Start time: 0.0
- Duration: 21600.0 s
- 1 detector(s) (0A)
- TOD(s): tod
- TOD shape: 1×216000
- Types of the TODs: float64

## Observation #1
- Start time: 43200.0
- Duration: 21600.0 s
- 1 detector(s) (0A)
- TOD(s): tod
- TOD shape: 1×216000
- Types of the TODs: float64

# MPI rank #2

## Observation #0
- Start time: 21600.0
- Duration: 21600.0 s
- 1 detector(s) (0A)
- TOD(s): tod
- TOD shape: 1×216000
- Types of the TODs: float64

## Observation #1
```

Observations - read and write

You can save and read an Observation through an HDF5 file containing:

- Time format
- TODs
- pointings, psi, pixidx
- Flags

Save with `write_observations()` or `write_list_of_observations()`

Read with `read_observations()` or `read_list_of_observations()`

```
#save tods (parallel case)
tod_filename_dict = [{ "myvalue": "LB_LFT_50_obs_rank"+str(rank).zfill(4) }]

lbs.io.write_list_of_observations(obs=obs,        #Observation
                                  path=obs_path, #where to store the file
                                  file_name_mask="{myvalue}.hdf5",
                                  custom_placeholders=tod_filename_dict,
                                  collective_mpi_call=True,
                                  tod_fields=[“tod”,”tod_noise”])
```

Map-making

The framework provides the following 3 solutions:

1. binner, assuming only uncorrelated noise

```
map, cov = lbs.make_bin_map(obs, nside=128, do_covariance=True)
```

2. destriper, able to remove correlated noise (work in progress) →TOAST

```
params = lbs.DestriperParameters(
    nside=16,
    return_hit_map=True,
    return_binned_map=True,
    return_destriped_map=True,
)
```

→

```
result = lbs.destripe(sim, params)
```

Use `np.float64` for this case /!\

3. save data in Madam-friendly format with `save_simulation_for_madam()` and run Madam (`$ madam madam.par`). See here for memory-saving tricks when producing several maps

Synthetic sky maps - tools necessary to produce synthetic sky maps

The LiteBIRD Simulation Framework provides the tools necessary to produce synthetic maps of the sky!

```
import litebird_sim as lbs
import matplotlib.pyplot as plt

sim = lbs.Simulation(base_path="./tut06")
params = lbs.MbsParameters(
    make_cmb=True,
    fg_models=["pysm_synch_0", "pysm_freefree_1"],
)
mbs = lbs.Mbs(
    simulation=sim,
    parameters=params,
    channel_list=[lbs.FreqChannelInfo.from_imo(
          sim.imo,
          "/releases/v1.3/satellite/LFT/L1-040/channel_info")]
)
(healpix_maps, file_paths) = mbs.run_all()

import healpy
healpy.mollview(healpix_maps["L1-040"][0])


sim.append_to_report("""
## Map
Here is the map:
![](map.png)
""",
      figures=[(plt.gcf(), "map.png")],
)
sim.flush()
```

Available emission models

PySM3 library to generate these sky maps

In the dictionary containing the maps, `Mbs` returns also two useful variables:

- The *coordinates* of the generated maps, in the key `Coordinates`
- The *used parameters* for the synthetic map generation, in the key `Mbs_parameters`

Try to click the UUID link in the Data Files section of the report

Scanning strategy

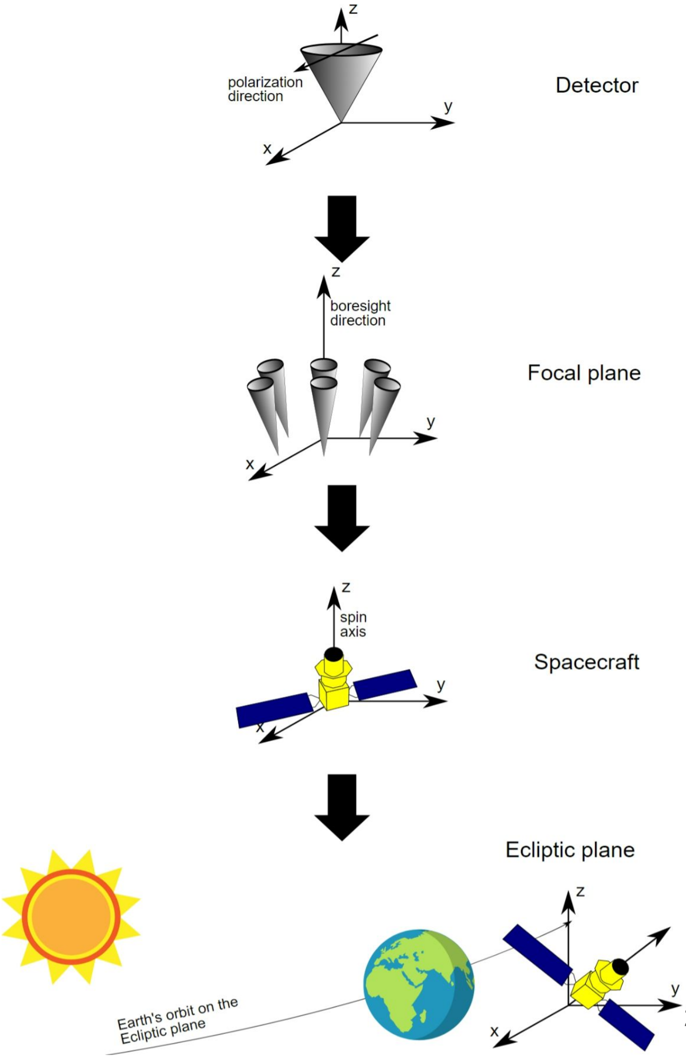
To compute where a detector is looking at (*pointing*), there is a number of transformations that need to be carried out!

Pointings for a detector are computed with 3 kinds the quaternions:

1. From the detector (main beam of radiation pattern along z axis) to the focal plane. Included in the IMO, initialized with `DetectorInfo.from_imo()`

2. From the focal plane (z axis aligned with boresight) to the spacecraft (z axis aligned with spin axis); field `bore2spin_quat` of the class `InstrumentInfo`, initialized with `Simulation.set_scanning_strategy()`

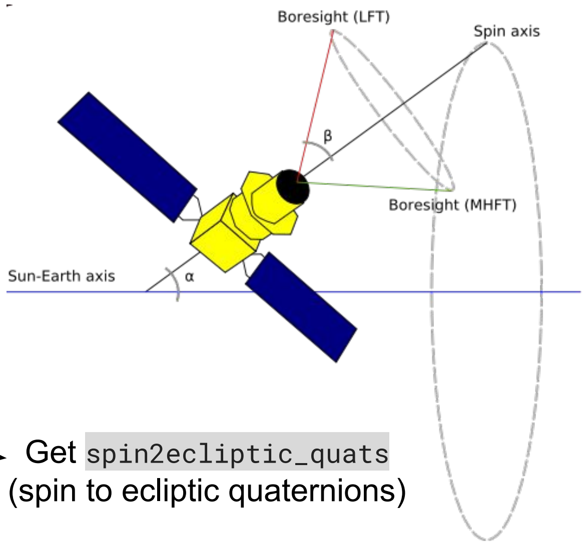
3. From spacecraft to ecliptic reference frame (z axis aligned with Ecliptic North Pole); can be done assuming circular motion or with ephemeridis tables



Another interesting reference here for quaternions!

Scanning strategy - a practical example

```python
import litebird_sim as lbs
import astropy.units as u
import numpy as np

sim = lbs.Simulation(
    start_time=0,
    duration_s=60.0,
    description="Simple simulation",
)

# We now simulate the motion of the spacecraft over one minute
sim.set_scanning_strategy(
    scanning_strategy=lbs.SpinningScanningStrategy(
        spin_sun_angle_rad=np.deg2rad(30), # CORE-specific parameter
        spin_rate_hz=0.5 / 60.0,
        # We use astropy to convert the period (4 days) in seconds
        precession_rate_hz=1.0 / (4 * u.day).to("s").value,
    )
)

# Here we specify the β angle of the focal plane of the instrument
sim.set_instrument(
    lbs.InstrumentInfo(
        name="core",
        spin_boresight_angle_rad=np.deg2rad(65),
    ),
)

# The motion of the spacecraft is now encoded in a set of quaternions,
# in the field `sim.spin2ecliptic_quats`. We use it to produce the
# pointing information for a fake boresight detector `det`, belonging to `core`
det = lbs.DetectorInfo(name="foo", sampling_rate_hz=10)

# By default, `create_observations` creates just *one* observation
obs, = sim.create_observations(detectors=[det])

# Compute the pointings at the same sampling frequency as the TOD
sim.compute_pointings()

pointings = obs.pointings

print("Shape:", pointings.shape)
print("Pointings:")
print(np.array_str(pointings, precision=3))
```



Get `spin2ecliptic_quats` (spin to ecliptic quaternions)

Set `spin_boresight_angle_rad` (boresight to spin angle)

Get θ,φ,ψ

A real example of litebird_sim usage

- litebird_sim has been used to produce the official Post-PTEP end-to-end simulations!
- Here you can find some details:

[LiteBIRD post-PTEP e2e simulations]

Summary

- Introduction: litebird_sim as a library to develop your own simulation codes
- Installing the framework: regular user or developer?
- Tutorial: create first report and `flush()` it
- Simulations: parameters handling (manual vs `.toml` file)
- Detectors, channels and Instruments: set detectors, channels, instrument properties by either manually inserting them, or reading from the IMO or from a file
- Observations: data acquired by the telescope during a scanning period
- Map-making: 3 possibilities (binner, destriper TOAST, MADAM)
- Synthetic sky maps: produce your own maps with different sky components (*PySM3*)
- Scanning strategy: set of transformations from the detector direction of observation to the spacecraft orientation in the ecliptic plane

ありがとうございます

Thank you

Grazie

Additional material

- Bandpasses
- Dipole anisotropy
- The Instrument Model Database (IMO)
- Time Ordered Simulations
- Creating reports with litebird_sim
- Using MPI
- External modules
- Integrating existing codes
- Bibliography

Bandpasses

Through the class BandPassInfo, one either can load bandapsses from the IMO or generate them by scratch, e.g. example below

```
import litebird_sim as lbs
import matplotlib.pylab as plt

# Generate the «model» (i.e., ideal band, with no wiggles)
band = lbs.BandPassInfo(
    bandcenter_ghz=43.0,
    bandwidth_ghz=10.0,
    bandtype="top-hat-cosine",
    normalize=True,
    nsamples_inband=100,
)
plt.plot(band.freqs_ghz, band.weights, label="Ideal band")

# Now generate a more realistic band with random wiggles
new_weights = band.bandpass_resampling()
plt.plot(band.freqs_ghz, new_weights,label="Random realization",
)

plt.xlabel("Frequency [GHz]")
plt.ylabel("Weight")
plt.legend()
```

Essential parameters:

- central frequency
- bandwidth
- random component (to make the band more realistic)