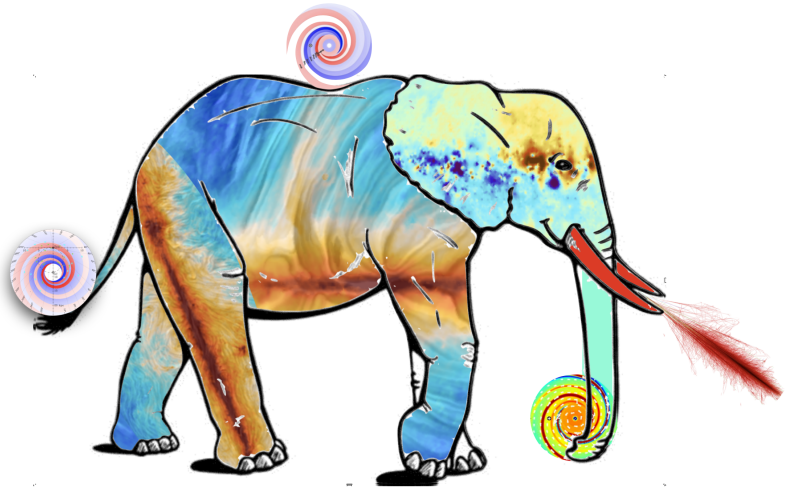
IMAGINE

Jul 31, 2020

Contents:



Welcome to the documentation of the IMAGINE software package, a publicly available Bayesian platform that allows using a variety of observational data sets to constrain models for the main ingredients of the interstellar medium of the Galaxy. IMAGINE calculates simulated data sets from the galaxy models and compares these to the observational data sets through a likelihood evaluation. It then samples this multi-dimensional likelihood space, which allows one to update prior knowledge, and thus to find the position with the *best-fit model* parameters and/or compute the *model evidence* (which enables rigorous comparison of competing models).

IMAGINE is developed and maintained by the IMAGINE consortium, a diverse group of researchers whose common interest revolves around developing an integrated understanding of the various components of the Galactic interstellar medium (with emphasis on the Galactic magnetic field and its interaction with cosmic rays). For more details on IMAGINE science case, please refer to the IMAGINE whitepaper.

CHAPTER 1

Installation and dependencies

Here you can find basic instructions for the installation of IMAGINE. There are two main installation routes:

1. one can pull and run a *Docker installation* which allows you to setup and run IMAGINE by typing only two lines. IMAGINE will run in a container, i.e. separate from your system.
2. one can *download and install* IMAGINE and all the dependencies alongside your system.

The first option is particularly useful when one is a newcomer, interested experimenting or when one is deploying IMAGINE in a cloud service or multiple machines.

The second option is better if one wants to use ones pre-installed tools and packages, or if one is interested in running on a computing cluster (running docker images in some typical cluster settings may be difficult or impossible).

Let us know if you face major difficulties.

1.1 Docker installation

This is a very convenient and fast way of deploying IMAGINE. You must first pull the image of one of IMAGINE's versions from GitHub, for example, the latest (*development*) version can be pulled using:

```
sudo docker pull docker.pkg.github.com/imagine-consortium/imagine/imagine:latest
```

If you would like to start working (or testing IMAGINE) immediately, a jupyter-lab session can be launched using:

```
sudo docker run -i -t -p 8888:8888 imagine:latest /bin/bash -c "source ~/jupyterlab.
↪bash"
```

After running this, you may copy and paste the link with a token to a browser, which will allow you to access the jupyter-lab session. From there you may, for instance, navigate to the *imagine/tutorials* directory.

1.2 Standard installation

1.2.1 Download

A copy of IMAGINE source can be downloaded from its main GitHub repository. If one does not intend to contribute to the development, one should download and unpack the latest release:

```
wget https://github.com/IMAGINE-Consortium/imagine/archive/v2.0.0-alpha.1.tar.gz
tar -xvvzf v2.0.0-alpha.1.tar.gz
```

Alternatively, if one is interested in getting involved with the development, we recommend cloning the git repository

```
git clone git@github.com:IMAGINE-Consortium/imagine.git
```

1.2.2 Setting up the environment with conda

IMAGINE depends on a number of different python packages. The easiest way of setting up your environment is using the *conda* package manager. This allows one to setup a dedicated, contained, python environment in the user area.

Conda is the package manager of the Anaconda Python distribution, which by default comes with a large number of packages frequently used in data science and scientific computing, as well as a GUI installer and other tools.

A lighter, recommended, alternative is the Miniconda distribution, which allows one to use the conda commands to install only what is actually needed.

Once one has installed (mini)conda, one can download and install the IMAGINE environment in the following way:

```
conda env create --file=imagine_conda_env.yml
conda activate imagine
python -m ipykernel install --user --name imagine --display-name "Python (imagine)"
```

The (optional) last line creates a Jupyter kernel linked to the new conda environment (which is required, for example, for executing the tutorial Jupyter notebooks).

Whenever one wants to run an IMAGINE script, one has to first activate the associated environment with the command *conda activate imagine*. To leave this environment one can simply run *conda deactivate*

1.2.3 Hammurabi X

A key dependency of IMAGINE is the Hammurabi X code, a HEALPix-based numeric simulator for Galactic polarized emission (arXiv:1907.00207).

Before proceeding with the IMAGINE installation, it is necessary to install Hammurabi X following the instructions on its project wiki. Then, one needs to install the *hampyx* python wrapper:

```
conda activate imagine # if using conda
cd PATH_TO_HAMMURABI
pip install -e .
```

1.2.4 Installing

After downloading, setting up the environment and installing Hammurabi X, IMAGINE can finally be installed through:

```
conda activate imagine # if using conda
cd IMAGINE_PATH
pip install .
```

If one does not have admistrator/root privileges/permissions, one may instead want to use

```
pip install --user .
```

Also, if you are working on further developing or modifying IMAGINE for your own needs, you may wish to use the *-e* flag, to keep links to the source directory instead of copying the files,

```
pip install -e .
```

CHAPTER 2

Design overview

Our basic objective is, given some data, to be able to constrain the parameter space of a model, and/or to compare the plausibility of different models. IMAGINE was designed to allow that different groups working on different models could be to constrain them through easy access a range of datasets and the required statistical machinery. Likewise, observers can quickly check the consequences and interpret their new data by seeing the impact on different models and toy models.

In order to be able to do this systematically and rigorously, the basic design of IMAGINE first breaks the problem into two abstractions: *Fields*, which represent models of physical fields, and *Observables*, which represent both observational and mock data.

New observational data are included in IMAGINE using the *Datasets*, which help effortlessly adjusting the format of the data to the standard specifications (and are internally easily converted into *Observables*) Also, a collection of *Datasets* contributed by the community can be found in the Consortium's dedicated Dataset repository.

The connection between a theory and reality is done by one of the *Simulators*. Each of these corresponds to a mapping from a set of model *Fields* into a mock *Observables*. The available simulators, importantly, include Hammurabi, which can compute Faraday rotation measure and diffuse synchrotron and thermal dust emission.

Each of these *IMAGINE Components* (*Fields*, *Observables*, *Datasets*, *Simulators*) are represented by a Python class in IMAGINE. Therefore, in order to extend IMAGINE with a specific new field or including a new observational dataset, one needs to create a *subclass* of one of IMAGINE's base classes. This subclass will, very often, be a wrapper around already existing code or scripts. To preserve the modularity and flexibility of IMAGINE, one should try to use (*as far as possible*) only the provided base classes.

Fig. 2.1 describes the typical workflow of IMAGINE and introduces other key base classes. Mock and measured data, in the form of *Observables*, are used to compute a likelihood through a *Likelihoods* class. This, supplemented by *Priors*, allows a *Pipeline* object to sample the parameter space and compute posterior distributions and Bayesian evidences for the models. The generation of different realisations of each Field is managed by the corresponding *Field Factory* class. Likewise, *Observable Dictionaries* help one organising and manipulating *Observables*.

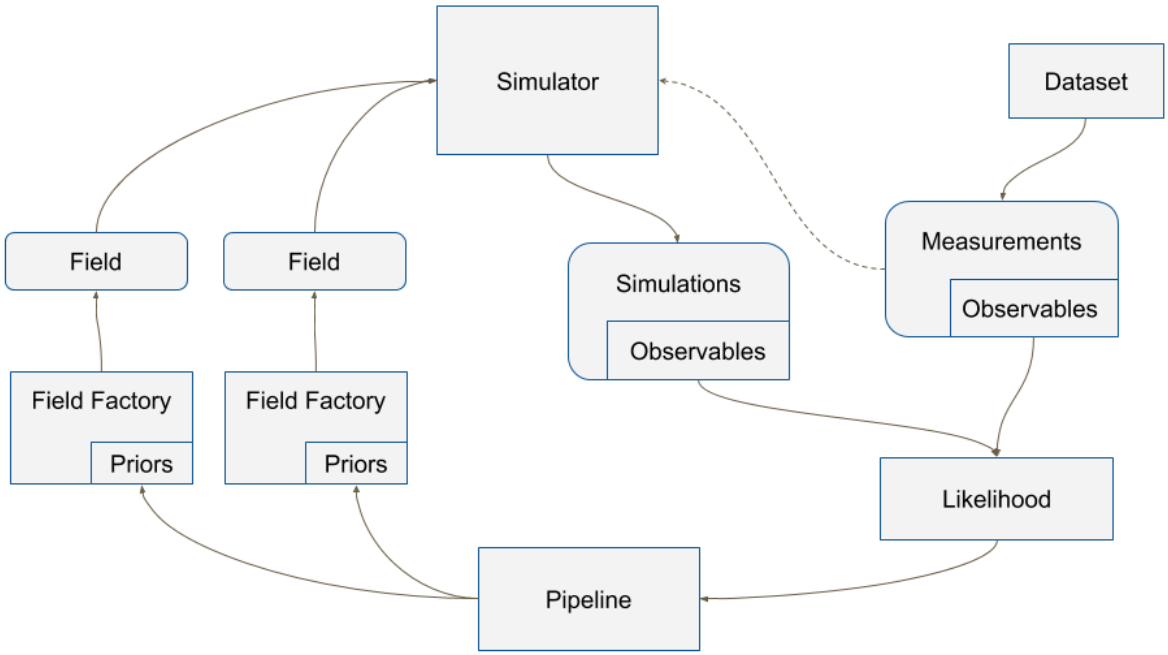


Fig. 2.1: The structure of the IMAGINE pipeline.

CHAPTER 3

IMAGINE Components

In the following sections we describe each of the basic components of IMAGINE. We also demonstrate how to write wrappers that allow the inclusion of external (pre-existing) code, and provide code templates for this.

Contents

3.1 Fields

In IMAGINE terminology, a **field** refers to any *computational model* of a spatially varying physical quantity, such as the Galactic Magnetic Field (GMF), the thermal electron distribution, or the Cosmic Ray (CR) distribution. Generally, a field object will have a set of parameters — e.g. a GMF field object may have a pitch angle, scale radius, amplitude, etc. *Field* objects are used as inputs by the *Simulators*, which *simulate* those physical models, i.e. they allow constructing *observables* based on a set models.

During the sampling, the *Pipeline* does not handle fields directly but instead relies on *Field Factory* objects. While the *field objects* will do the actual computation of the physical field, given a set of physical parameters and a coordinate grid, the *field factory objects* take care of book-keeping tasks: they hold the parameter ranges and default values, they translate the dimensionless parameters used by the sampler (always in the interval $[0, 1]$) to physical parameters, they also store the *Priors* associated with each parameter of that *field*.

To convert ones own model into a IMAGINE-compatible field, one must create a subclass of one of the base classes available the `imagine.fields.base_fields`, most likely using one of the available templates (discussed in the sections below) to write a *wrapper* to the original code. If the basic field type one is interested in is *not* available as a basic field, one can create it directly subclassing `imagine.fields.field.GeneralField` — and this could benefit the wider community, please consider submitting a pull request or openning an issue requesting the inclusion of the new field type!

It is assumed that **Field** objects can be expressed as a parametrised *mapping of a coordinate grid into a physical field*. The grid is represented by a IMAGINE *Grid* object, discussed in detail in the next section. If the field is of random or **stochastic** nature (e.g. the density field of a turbulent medium), IMAGINE will compute a *finite ensemble* of different realisations which will later be used in the inference to determine the likelihood of the actual observation, accounting for the model's expected variability.

To test a Field class, `FieldFoo`, one can instantiate the field object:

```
bar = FieldFoo(grid=example_grid, parameters={'answer': 42*u.cm}, ensemble_size=2)
```

where `example_grid` is a previously instantiated grid object. The argument `parameters` receives a dictionary of all the parameters used by `FieldFoo`, these are usually expressed as dimensional quantities (using `astropy.units`). Finally, the argument `ensemble_size`, as the name suggests allows requestion a number of different realisations of the field (for non-stochastic fields, all these will be references to the same data).

To further illustrate, assuming we are dealing with a scalar, the (spherical) radial dependence of the above defined `bar` can be easily plotted using:

```
import matplotlib.pyplot as plt
bar_data = bar.get_data(ensemble_index)
plt.plot(bar.grid.r_spherical.ravel(), bar_data.ravel())
```

The design of any field is done writing a *subclass* of one of the classes in `imagine.fields.base_fields` or `imagine.GeneralField` that overrides the method `compute_field(seed)`, using it to compute the field on each spatial point. For this, the coordinate grid on which the field should be evaluated can be accessed from `self.grid` and the parameters from `self.parameters`. The same parameters must be listed as *keys* in the `field_checklist` dictionary (the values in the dictionary are used to supply extra information to specific simulators, but can be left as *None* in the general case).

3.1.1 Grid

Field objects require (with the exception of *Dummy* fields) a coordinate grid to operate. In IMAGINE this is expressed as an instance of the `imagine.fields.grid.BaseGrid` class, which represents coordinates as a set of three 3-dimensional arrays. The grid object supports cartesian, cylindrical and spherical coordinate systems, handling any conversions between these automatically through the properties.

The convention is that 0 of the coordinates corresponds to the Galaxy (or galaxy) centre, with the z coordinate giving the distance to the midplane.

To construct a grid with uniformly-distributed coordinates one can use the *imagine.fields.grid.UniformGrid* that accompanies IMAGINE. For example, one can create a grid where the cylindrical coordinates are equally spaced using:

```
import imagine as img
cylindrical_grid = img.UniformGrid(box=[[0.25*u.kpc, 15*u.kpc],
                                        [-180*u.deg, np.pi*u.rad],
                                        [-15*u.kpc, 15*u.kpc]],
                       resolution = [9,12,9],
                       grid_type = 'cylindrical')
```

The `box` argument contains the lower and upper limits of the coordinates (respectively r, ϕ and z), `resolution` specifies the number of points for each dimension, and `grid_type` chooses this to be cylindrical coordinates.

The coordinate grid can be accessed through the properties `cylindrical_grid.x`, `cylindrical_grid.y`, `cylindrical_grid.z`, `cylindrical_grid.r_cylindrical`, `cylindrical_grid.r_spherical`, `cylindrical_grid.theta` (polar angle), and `cylindrical_grid.phi` (azimuthal angle), with conversions handled automatically. Thus, if one wants to access, for instance, the corresponding x cartesian coordinate values, this can be done simply using:

```
cylindrical_grid.x[:,:,:]
```

To create a personalised (non-uniform) grid, one needs to subclass *imagine.fields.grid.BaseGrid* and override the method *generate_coordinates*. The *imagine.fields.grid.UniformGrid* class should itself provide a good example/template of how to do this.

3.1.2 Thermal electrons

A new model for the distribution of thermal electrons can be introduced subclassing *imagine.fields.base_fields.ThermalElectronDensityField* according to the template below.

```
from imagine.fields import ThermalElectronDensityField
import numpy as np
import MY_GALAXY_MODEL # Substitute this by your own code


class ThermalElectronsDensityTemplate(ThermalElectronDensityField):
    """ Here comes the description of the electron density model """

    # Class attributes
    NAME = 'name_of_the_thermal_electrons_field'

    # Is this field stochastic or not. Only necessary if True
    STOCHASTIC_FIELD = True
    # If there are any dependencies, they should be included in this list
    DEPENDENCIES_LIST = []

    @property
    def field_checklist(self):
        # This property returns a dictionary with all the
        # available parameters as keys
        return {'Parameter_A': None, 'Parameter_B': None}
```

(continues on next page)

(continued from previous page)

```
    def compute_field(self, seed):
        # If this is an stochastic field, the integer `seed `must be
        # used to set the random seed for a single realisation.
        # Otherwise, `seed` should be ignored.

        # The coordinates can be accessed from an internal grid object
        x_coord = self.grid.x
        y_coord = self.grid.y
        z_coord = self.grid.y
        # Alternatively, one can use cylindrical or spherical coordinates
        r_cyl_coord = self.grid.r_cylindrical
        r_sph_coord = self.grid.r_spherical
        theta_coord = self.grid.theta
        phi_coord = self.grid.phi

        # One can access the parameters supplied in the following way
        param_A = self.parameters['Parameter_A']
        param_B = self.parameters['Parameter_B']

        # Now you can interface with previous code or implement here
        # your own model for the thermal electrons distribution.
        # Returns the electron number density at each grid point
        # in units of (or convertible to) cm**-3
        return MY_GALAXY_MODEL.compute_ne(param_A, param_B,
                                          r_sph_coord, theta_coord, phi_coord,
                                          # If the field is stochastic
                                          # it can use the seed
                                          # to generate a realisation
                                          seed)
```

Note that the return value of the method *compute_field()* must be of type `astropy.units.Quantity`, with shape consistent with the coordinate grid, and units of cm^{-3}.

The template assumes that one already possesses a model for distribution of thermal e^- in a module `MY_GALAXY_MODEL`. Such model needs to be able to map an arbitrary coordinate grid into densities.

Of course, one can also write ones model (if it is simple enough) into the derived subclass definition. On example of a class derived from *imagine.fields.base_fields.ThermalElectronDensityField* can be seen bellow:

```
from imagine import ThermalElectronDensityField

class ExponentialThermalElectrons(ThermalElectronDensityField):
    """Example: thermal electron density of an (double) exponential disc"""

    field_name = 'exponential_disc_thermal_electrons'

    @property
    def field_checklist(self):
        return {'central_density' : None, 'scale_radius' : None, 'scale_height' :␣
↪None}

    def compute_field(self):
        R = self.grid.r_cylindrical
        z = self.grid.z
        Re = self.parameters['scale_radius']
        he = self.parameters['scale_height']
```

(continues on next page)

(continued from previous page)

```
        n0 = self.parameters['central_density']

        return n0*np.exp(-R/Re)*np.exp(-np.abs(z/he))
```

3.1.3 Magnetic Fields

One can add a new model for magnetic fields subclassing *imagine.fields.base_fields.MagneticField* as illustrated in the template below.

```
from imagine.fields import MagneticField
import astropy.units as u
import numpy as np
# Substitute this by your own code
import MY_GMF_MODEL


class MagneticFieldTemplate(MagneticField):
    """ Here comes the description of the magnetic field model """

    # Class attributes
    NAME = 'name_of_the_magnetic_field'

    # Is this field stochastic or not. Only necessary if True
    STOCHASTIC_FIELD = True
    # If there are any dependencies, they should be included in this list
    DEPENDENCIES_LIST = []

    @property
    def field_checklist(self):
        # This property returns a dictionary with all the
        # available parameters as keys
        return {'Parameter_A': None, 'Parameter_B': None}

    def compute_field(self, seed):
        # If this is an stochastic field, the integer `seed `must be
        # used to set the random seed for a single realisation.
        # Otherwise, `seed` should be ignored.

        # The coordinates can be accessed from an internal grid object
        x_coord = self.grid.x
        y_coord = self.grid.y
        z_coord = self.grid.y
        # Alternatively, one can use cylindrical or spherical coordinates
        r_cyl_coord = self.grid.r_cylindrical
        r_sph_coord = self.grid.r_spherical
        theta_coord = self.grid.theta; phi_coord = self.grid.phi

        # One can access the parameters supplied in the following way
        param_A = self.parameters['Parameter_A']
        param_B = self.parameters['Parameter_B']

        # Now one can interface with previous code, or implement a
        # particular magnetic field
        Bx, By, Bz = MY_GMF_MODEL.compute_B(param_A, param_B,
                                            x_coord, y_coord, z_coord,
```

(continues on next page)

(continued from previous page)

```
                                            # If the field is stochastic
                                            # it can use the seed
                                            # to generate a realisation
                                            seed)

        # Creates an empty output magnetic field Quantity with
        # the correct shape and units
        MF_array = np.empty(self.data_shape) * u.microgauss
        # and saves the pre-computed components
        MF_array[:,:,:,0] = Bx
        MF_array[:,:,:,1] = By
        MF_array[:,:,:,2] = Bz

        return MF_array
```

It was assumed the existence of a hypothetical module `MY_GALAXY_MODEL` which, given a set of parameters and three 3-arrays containing coordinate values, computes the magnetic field vector at each point.

The method `compute_field()` must return an `astropy.units.Quantity`, with shape *(Nx,Ny,Nz,3)* where *Ni* is the corresponding grid resolution and the last axis corresponds to the component (with x, y and z associated with indices 0, 1 and 2, respectively). The Quantity returned by the method must correpond to a magnetic field (i.e. units must be μG, G, nT, or similar).

A simple example, comprising a constant magnetic field can be seen below:

```
from imagine import MagneticField

class ConstantMagneticField(MagneticField):
    """Example: constant magnetic field"""
    field_name = 'constantB'

    @property
    def field_checklist(self):
        return {'Bx': None, 'By': None, 'Bz': None}

    def compute_field(self):
        # Creates an empty array to store the result
        B = np.empty(self.data_shape) * self.parameters['Bx'].unit
        # For a magnetic field, the output must be of shape:
        # (Nx,Ny,Nz,Nc) where Nc is the index of the component.
        # Computes Bx
        B[:,:,:,0] = self.parameters['Bx']*np.ones(self.grid.shape)
        # Computes By
        B[:,:,:,1] = self.parameters['By']*np.ones(self.grid.shape)
        # Computes Bz
        B[:,:,:,2] = self.parameters['Bz']*np.ones(self.grid.shape)
        return B
```

3.1.4 Cosmic ray electrons

Under development

3.1.5 Dummy

There are situations when one may want to sample parameters which are not used to evaluate a field on a grid before being sent to a Simulator object. One possible use for this is representing a global property of the physical system which affects the observations (for instance, some global property of the ISM or, if modelling an external galaxy, the position of the galaxy).

Another common use of dummy fields is when a field is generated at runtime *by the simulator*. One example are the built-in fields available in Hammurabi: instead of requesting IMAGINE to produce one of these fields and hand it to Hammurabi to compute the associated synchrotron observables, one can use dummy fields to request Hammurabi to generate these fields internally for a given choice of parameters.

Using dummy fields to bypass the design of a full IMAGINE Field may simplify implementation of a Simulator wrapper and (sometimes) may be offer good performance. However, this practice of generating the actual field within the Simulator *breaks the modularity of IMAGINE*, and it becomes impossible to check the validity of the results plugging the same field on a different Simulator. Thus, use this with care!

A dummy field can be implemented by subclassing *imagine.fields.base_fields.DummyField* as shown bellow.

```
from imagine.fields import DummyField

class DummyFieldTemplate(DummyField):
    """
    Description of the dummy field
    """

    # Class attributes
    NAME = 'name_of_the_dummy_field'

    @property
    def field_checklist(self):
        return {'Parameter_A': 'parameter_A_settings',
                'Parameter_B': None}
    @property
    def simulator_controllist(self):
        return {'simulator_property_A': 'some_setting'}
```

Dummy fields are generally Simulator-specific and the properties `field_checklist` and `simulator_controllist` are convenient ways of sending extra settings information to the associated Simulator. The values in `field_checklist` allow transmitting settings associated with specific parameters, while the dictionary `simulator_controllist` can be used to tell how the presence of the the current dummy field should modify the Simulator's global settings.

For example, in the case of Hammurabi, a dummy field can be used to request one of its built-in fields, which has to be set up by modifying Hammurabi's XML parameter files. In this particular case, `field_checklist` is used to supply the position of a parameter in the XML file, while `simulator_controllist` indicates how to modify the switch in the XML file that enables this specific built-in field (see the *The Hammurabi simulator* tutorial for details).

3.1.6 Field Factory

Field Factories, represented in IMAGINE by a subclass of `imagine.fields.field_factory.GeneralFieldFactory` are an additional layer of infrastructure used by the samplers to provide the connection between the sampling of points in the likelihood space and the field object that will be given to the simulator.

A *Field Factory* object has a list of all of the field's parameters and a list of the subset of those that are to be varied in the sampler — the latter are called the **active parameters**. The Field Factory also holds the allowed value ranges for

each parameter, the default values (which are used for inactive parameters) and the prior distribution associated with each parameter.

At each step the *Pipeline* request the Field Factory for the next point in parameter space, and the Factory supplies it the form of a Field object constructed with that particular choice of parameters. This can then be handed by the Pipeline to the Simulator, which computes simulated Observables for comparison with the measured observables in the Likelihood module.

Given a Field *YourFieldClass* (which must be an instance of a class derived from `GeneralField`) the following template can be used construct a field factory:

```
from imagine.fields import FieldFactory
from imagine.priors import FlatPrior, GaussianPrior
# Substitute this by your own code
from MY_PACKAGE import MY_FIELD_CLASS
from MY_PACKAGE import A_std_val, B_std_val, A_min, A_max, B_min, B_max, B_sig

class FieldFactoryTemplate(FieldFactory):
    """Example: field factory for YourFieldClass"""

    # Class attributes
    # Field class this factory uses
    FIELD_CLASS = MY_FIELD_CLASS

    # Default values are used for inactive parameters
    DEFAULT_PARAMETERS = {'Parameter_A': A_std_val,
                          'Parameter_B': B_std_val}

    # All parameters need a range and a prior
    PRIORS = {'Parameter_A': FlatPrior(interval=[A_min, A_max]),
              'Parameter_B': GaussianPrior(mu=B_std_val, sigma=B_sig,
                                           interval=[B_min, B_max])}
```

The object *Prior A* must be an instance of *imagine.priors.prior.GeneralPrior* (see section *Priors* for details). A flat prior (i.e. a uniform, where all parameter values are equally likely) can be set using the *imagine.priors.basic_priors.FlatPrior* class.

One can initialize the Field Factory supplying the grid on which the corresponding Field will be evaluated:

```
myFactory = YourField_Factory(grid=cartesian_grid)
```

The factory object `myFactory` can now be handled to the *Pipeline*, which will generate new fields by calling the *FieldFactory()*.

3.2 Datasets

Dataset objects are helpers used for the inclusion of observational data in IMAGINE. They convert the measured data and uncertainties to a standard format which can be later handed to an *observable dictionary*. There are two main types of datasets: *Tabular datasets* and *HEALPix datasets*.

A (soon growing) number of ready-to-use datasets are available at the community maintained imagine-datasets repository. Below the usage of an imported dataset is illustrated:

```
import imagine as img
import imagine_datasets as img_datasets
```

(continues on next page)

(continued from previous page)

```
# Loads the dataset (usually involves downloading the data)
my_data = img_datasets.observable_type.AuthorYear()

# Initialises ObservableDict objects
measurement = img.Measurements()
covariances = img.Covariances()

# Loads the data on the ObservableDict's
measurement.append(dataset=my_data)
covariances.append(dataset=my_data)
```

Each observable type should has an agreed/conventional name. The presently available observable names are:

- 'fd' - Faraday depth
- 'sync' - Synchrotron emission
  - with tag 'I' - Total intensity
  - with tag 'Q' - Stokes Q
  - with tag 'U' - Stokes U
  - with tag 'PI' - polarisation intensity
  - with tag 'PA' - polarisation angle
- 'dm' - Dispersion measure

3.2.1 Tabular datasets

As the name indicates, in **tabular datasets** the observational data was originally in tabular format, i.e. a table where each row corresponds to a different *position in the sky* and columns contain (at least) the sky coordinates, the measurement and the associated error. A final requirement is that the dataset is stored in a *dictionary-like* object i.e. the columns can be selected by column name (for example, a Python dictionary, a Pandas DataFrame, or an astropy Table).

To construct a tabular dataset, one needs to initialize *imagine.observables.TabularDataset*. Below, a simple example of this, which fetches (using the package astroquery) a catalog from ViZieR and stores in in an IMAGINE tabular dataset object:

```
from astroquery.vizier import Vizier
from imagine.observables import TabularDataset

# Fetches the catalogue
catalog = Vizier.get_catalogs('J/ApJ/714/1170')[0]

# Loads it to the TabularDataset (the catalogue obj actually contains units)
RM_Mao2010 = TabularDataset(catalog, name='fd', units=catalog['RM'].unit,
                            data_column='RM', error_column='e_RM', tag=None,
                            lat_column='GLAT', lon_column='GLON')
```

From this point the object `RM_Mao2010` can be appended to a *Measurements*. We refer the reader to the the *Including new observational data* tutorial and the *TabularDataset* api documentation and for further details.

3.2.2 HEALPix datasets

HEALPix datasets will generally comprise maps of the full-sky, where HEALPix pixelation is employed. For standard observables, the datasets can be initialized by simply supplying a `Quantity` array containing the data to the corresponding class. Below some examples, employing the classes *`FaradayDepthHEALPixDataset`*, *`DispersionMeasureHEALPixDataset`* and *`SynchrotronHEALPixDataset`*, respectively:

```
from imagine.observables import FaradayDepthHEALPixDataset
from imagine.observables import DispersionMeasureHEALPixDataset
from imagine.observables import SynchrotronHEALPixDataset

my_FD_dset = FaradayDepthHEALPixDataset(data=fd_data_array,
                                        error=fd_data_array_error)
my_DM_dset = DispersionMeasureHEALPixDataset(data=fd_data_array,
                                             cov=fd_data_array_covariance)
sync_dset = SynchrotronHEALPixDataset(data=stoke_Q_data,
                                      error=stoke_Q_data_error
                                      frequency=23*u.GHz, type='Q')
```

In the first example, it was assumed that the *covariance was diagonal*, and therefore can be described by an error associated with each pixel, which is specified with the keyword argument *error*. In the second example, the covariance associated with the data is instead specified supplying a two-dimensional array using the the *cov* keyword argument. The final example requires the user to supply the frequency of the observation and the subtype (in this case, 'Q').

3.3 Measurements, Simulations and Covariances

3.4 Simulators

```
from imagine.simulators import Simulator
import numpy as np
import MY_SIMULATOR  # Substitute this by your own code

class SimulatorTemplate(Simulator):
    """
    Detailed description of the simulator
    """
    # The quantity that will be simulated (e.g. 'fd', 'sync', 'dm')
    # Any observable quantity absent in this list is ignored by the simulator
    SIMULATED_QUANTITIES = ['my_observable_quantity']
    # A list or set of what is required for the simulator to work
    REQUIRED_FIELD_TYPES = ['dummy', 'magnetic_field']
    # Fields which may be used if available
    OPTIONAL_FIELD_TYPES = ['thermal_electron_density']
    # One must specify which grid is compatible with this simulator
    ALLOWED_GRID_TYPES = ['cartesian']
    # Tells whether this simulator supports using different grids
    USE_COMMON_GRID = False

    def __init__(self, measurements, **extra_args):
        # Send the measurements to parent class
        super().__init__(measurements)
        # Any initialization task involving **extra_args can be done *here*
        pass
```

(continues on next page)

(continued from previous page)

```
    def simulate(self, key, coords_dict, realization_id, output_units):
        """
        This is the main function you need to override to create your simulator.
        The simulator will cycle through a series of Measurements and create
        mock data using this `simulate` function for each of them.

        Parameters
        ----------
        key : tuple
            Information about the observable one is trying to simulate
        coords_dict : dictionary
            If the trying to simulate data associated with discrete positions
            in the sky, this dictionary contains arrays of coordinates.
        realization_id : int
            The index associated with the present realisation being computed.
        output_units : astropy.units.Unit
            The requested output units.
        """
        # The argument key provide extra information about the specific
        # measurement one is trying to simulate
        obs_quantity, freq_Ghz, Nside, tag = key

        # If the simulator is working on tabular data, the observed
        # coordinates can be accessed from coords_dict, e.g.
        lat, lon = coords_dict['lat'], coords_dict['lon']

        # Fields can be accessed from a dictionary stored in self.fields
        B_field_values = self.fields['magnetic_field']
        # If a dummy field is being used, instead of an actual realisation,
        # the parameters can be accessed from self.fields['dummy']
        my_dummy_field_parameters = self.fields['dummy']
        # Checklists can be used to send specific information to simulators
        # about specific parameters. Usually, to keep the modularity, this is
        # only done only for dummy fields
        checklist_params = self.field_checklist['dummy']
        # Controllists in dummy fields contain a dict of simulator settings
        simulator_settings = self.controllist

        # If a USE_COMMON_GRID is set to True, the grid it can be accessed from
        # grid = self.grid

        # Otherwise, if fields are allowed to use different grids, one can
        # get the grid from the self.grids dictionary and the field type
        grid_B = self.grids['magnetic_field']

        # Finally we can _simulate_, using whichever information is needed
        # and your own MY_SIMULATOR code:
        results = MY_SIMULATOR.simulate(simulator_settings,
                                        grid_B.x, grid_B.y, grid_B.z,
                                        lat, lon, freq_Ghz, B_field_values,
                                        my_dummy_field_parameters,
                                        checklist_params)
        # The results should be in a 1-D array of size compatible with
        # your dataset. I.e. for tabular data: results.size = lat.size
        # (or any other coordinate)
        # and for HEALPix data  results.size = 12*(Nside**2)
```

(continues on next page)

(continued from previous page)

```
        # Note: Awareness of other observables
        # While this method will be called for each individual observable
        # the other observables can be accessed from self.observables
        # Thus, if your simulator is capable of computing multiple observables
        # at the same time, the results can be saved to an attribute on the first
        # call of `simulate` and accessed from this cache later.
        # To break the degeneracy between multiple realisations (which will
        # request the same key), the realisation_id can be used
        # (see Hammurabi implementation for an example)
        return results
```

3.5 Likelihoods

Likelihoods define how to quantitatively compare the simulated and measured observables. They are represented within IMAGINE by an instance of class derived from *imagine.likelihoods.Likelihood*. There are two pre-implemented subclasses within IMAGINE:

- *imagine.likelihoods.SimpleLikelihood*: this is the traditional method, which is like a χ^2 based on the covariance matrix of the measurements (i.e., noise).
- *imagine.likelihoods.EnsembleLikelihood*: combines covariance matrices from measurements with the expected galactic variance from models that include a stochastic component.

Likelihoods need to be initialized before running the pipeline, and require measurements (at the front end). In most cases, data sets will not have covariance matrices but only noise values, in which case the covariance matrix is only the diagonal.

```
from imagine.likelihoods import EnsembleLikelihood
likelihood = EnsembleLikelihood(data, covariance_matrix)
```

The optional input argument `covariance_matrix` does not have to contain covariance matrices corresponding to all entries in input data. The Likelihood automatically defines the proper way for the various cases.

If the *EnsembleLikelihood* is used, then the sampler will be run multiple times at each point in likelihood space to create an ensemble of simulated observables.

3.6 Priors

A powerful aspect of a fully Baysean analysis approach is the possibility of explicitly stating any prior expectations about the parameter values based on previous knowledge. A prior is represented by an instance of *imagine.priors.prior.GeneralPrior* or one of its subclasses.

To use a prior, one has to initialize it and include it in the associated *Field Factory*. The simplest choice is a *FlatPrior* (i.e. any parameter within the range are equally likely before the looking at the observational data), which can be initialized in the following way:

```
from imagine.priors import FlatPrior
import astropy.units as u
myFlatPrior = FlatPrior(interval=[-2,10]*u.pc)
```

where the range for this parameter was chosen to be between -2 and $10\mathrm{pc}$.

After the flat prior, a common choice is that the parameter values are characterized by a Gaussian distribution around some central value. This can be achieved using the *imagine.priors.basic_priors.GaussianPrior*

class. As an example, let us suppose one has a parameter which characterizes the strength of a component of the magnetic field, and that ones prior expectation is that this should be gaussian distributed with mean 1μG and standard deviation 5μG. Moreover, let us assumed that the model only works within the range $[-30\mu\text{G}, 30\mu\text{G}]$. A prior consistent with these requirements can be achieved using:

```
from imagine.priors import GaussianPrior
import astropy.units as u
myGaussianPrior = GaussianPrior(mu=1*u.microgauss, sigma=5*u.microgauss,
                                interval=[-30*u.microgauss,30*u.microgauss])
```

3.7 Pipeline

The final building block of an IMAGINE pipeline is the **Pipline** object. When working on a problem with IMAGINE one will always go through the following steps:

1. preparing a list of the *field factories* which define the theoretical models one wishes to constrain and specifying any *priors*;
2. preparing a *measurements* dictionary, with the observational data to be used for the inference; and
3. initializing a *likelihood* object, which defines how the likelihood function should be estimated;

once this is done, one can *supply all these* to a **Pipeline** object, which will sample the *posterior distribution* and estimate the *evidence*. This can be done in the following way:

```
from imagine.pipelines import UltranestPipeline
# Initialises the pipeline
pipeline = UltranestPipeline(simulator=my_simulator,
                             factory_list=my_factory_list,
                             likelihood=my_likelihood,
                             ensemble_size=my_ensemble_size_choice)
# Runs the pipeline
pipeline()
```

After running, the results can be accessed through the attributes of the `Pipeline` object (e.g. `pipeline.samples`, which contains the parameters values of the samples produced in the run).

But what exactly is the Pipeline? The `Pipeline` base class takes care of interfacing between all the different IMAGINE components and sets the scene so that a Monte Carlo **sampler** can explore the parameter space and compute the results (i.e. posterior and evidence).

Different samplers are implemented as sub-classes of the Pipeline There are 3 samplers included in IMAGINE standard distribution (alternatives can be found in some of the IMAGINE Consortium repositories), these are: the `MultinestPipeline`, the `UltranestPipeline` and the `DynestyPipeline`.

One can include a new *sampler* in IMAGINE by creating a sub-class of `imagine.Pipeline`. The following template illustrates this procedure:

```
from imagine.pipelines import Pipeline
import numpy as np
import MY_SAMPLER  # Substitute this by your own code

class PipelineTemplate(Pipeline):
    """
    Detailed description of sampler being adopted
    """
    # Class attributes
```

(continues on next page)

(continued from previous page)

```
    # Does this sampler support MPI? Only necessary if True
    SUPPORTS_MPI = False

    def call(self, **kwargs):
        """
        Runs the IMAGINE pipeline

        Returns
        -------
        results : dict
            A dictionary containing the sampler results
            (usually in its native format)
        """
        # Resets internal state and adjusts random seed
        self.tidy_up()

        # Initializes a sampler object
        # Here we provide a list of common options
        self.sampler = MY_SAMPLER.Sampler(
            # Active parameter names can be obtained from
            param_names=self.active_parameters,
            # The likelihood function is available in
            loglike=self._likelihood_function,
            # Some samplers need a "prior transform function"
            prior_transform=self.prior_transform,
            # Other samplers need the prior PDF, which is
            prior_pdf=self.prior_pdf,
            # Sets the directory where the sampler writes the chains
            chains_dir=self.chains_directory,
            # Sets the seed used by the sampler
            seed=self.master_seed
            )

        # Most samplers have a `run` method, which should be executed
        self.sampling_controllers.update(kwargs)
        self.results = self.sampler.run(self.sampling_controllers)

        # The samples should be converted to a numpy array and saved
        # to self._samples_array. This should have different samples
        # on different rows and each column corresponds to an active
        # parameter
        self._samples_array = self.results['samples']
        # The log of the computed evidence and its error estimate
        # should also be stored in the following way
        self._evidence = self.results['logz']
        self._evidence_err = self.results['logzerr']

        return self.results
```

3.8 Disclaimer

Nothing is written in stone and the base classes may be updated with time (so, always remember to report the code release when you make use of IMAGINE). Suggestions and improvements are welcome as GitHub issues or pull requests

CHAPTER 4

Constraining parameters

Computing *posterior distribution* given a model and a dataset.

CHAPTER 5

Model comparison

Using *Bayesian evidence* to compare models.

CHAPTER 6

Parallelisation

The IMAGINE pipeline was designed with hybrid MPI/OpenMP use on a cluster in mind: the Pipeline distributes sampling work *accross different nodes* using MPI, while Fields and Simulators are assumed to use OpenMP (or similar shared memory multiprocessing) to run in parallel *within a single multi-core node*.

CHAPTER 7

Basic elements of an IMAGINE pipeline

In this tutorial, we focus on introducing the basic building blocks of the IMAGINE package and how to use them for assembling a Bayesian analysis pipeline.

We will use mock data with only two independent free parameters. First, we will generate the mock data. Then we will assemble all elements needed for the IMAGINE pipeline, execute the pipeline and investigate its results.

The mock data are designed to "naively" mimic Faraday depth, which is affected linearly by the (Galactic) magnetic field and thermal electron density. As a function of position x, we define a constant coherent magnetic field component a_0 and a random magnetic field component which is drawn from a Gaussian distribution with standard deviation b_0. The electron density is assumed to be independently known and given by a $\cos(x)$ with arbitrary scaling. The mock data values we get are related to the Faraday depth of a background source at some arbitrary distance:

$$signal(x) = [1 + \cos(x)] \times \mathcal{G}(\mu = a_0, \sigma = b_0; seed = s)\,\mu\mathrm{G\,cm}^{-3},\ x \in [0, 2\pi]\,\mathrm{kpc}$$

where $\{a_0, b_0\}$ is the 'physical' parameter set, and s represents the seed for random variable generation.

The purpose is not to fit the exact signal, since it includes a stochastic component, but to fit the amplitude of the signal and of the variations around it. So this is fitting the strength of the coherent field a_0 and the amplitude of the random field b_0. With these mock data and its (co)variance matrix, we shall assemble the IMAGINE pipeline, execute it and examine its results.

First, import the necessary packages.

[1]:
```
import numpy as np
import logging as log
from astropy.table import Table
import astropy.units as u
import corner
import matplotlib.pyplot as plt

import imagine as img

%matplotlib inline
```

7.1 1) Preparing the mock data

In calculating the mock data values, we introduce noise as:

$$data(x) = signal(x) + noise(x)$$

For simplicity, we propose a simple gaussian noise with mean zero and a standard deviation e:

$$noise(x) = \mathcal{G}(\mu = 0, \sigma = e)$$

.

We will assume that we have 10 points in the x-direction, in the range $[0, 2\pi]$ kpc.

[2]:
```
a0 = 3. # true value of a in microgauss
b0 = 6. # true value of b in microgauss
e = 0.1 # std of gaussian measurement error
s = 233 # seed fixed for signal field

size = 10 # data size in measurements
x = np.linspace(0.01,2.*np.pi-0.01,size) # where the observer is looking at

np.random.seed(s) # set seed for signal field

signal = (1+np.cos(x)) * np.random.normal(loc=a0,scale=b0,size=size)

fd = signal + np.random.normal(loc=0.,scale=e,size=size)

# We load these to an astropy table for illustration/visualisation
data = Table({'meas' : fd,
              'err': np.ones_like(fd)*e,
              'x': x,
              'y': np.zeros_like(fd),
              'z': np.zeros_like(fd),
              'other': np.ones_like(fd)*42
              })
data[:4] # Shows the first 4 points in tabular form
```

[2]:
```
<Table length=4>
        meas           err            x              y       z     other
      float64       float64      float64        float64 float64 float64
------------------ ------- ------------------ ------- ------- -------
   16.4217790817552     0.1               0.01     0.0     0.0    42.0
  7.172468731201507     0.1 0.7059094785755097     0.0     0.0    42.0
-3.2254947821460433     0.1 1.4018189571510193     0.0     0.0    42.0
0.27949334758966465     0.1 2.0977284357265287     0.0     0.0    42.0
```

These data need to be converted to an IMAGINE compatible format. To do this, we first create `TabularDataset` object, which helps importing dictionary-like dataset onto IMAGINE.

[3]:
```
fd_units = u.microgauss*u.cm**-3

mockDataset = img.observables.TabularDataset(data, name='test',
                                             data_column='meas',
                                             coordinates_type='cartesian',
                                             x_column='x', y_column='y',
                                             z_column='z', error_column='err',
                                             units=fd_units)
```

These lines simply explain how to read the tabular dataset (note that the 'other' column is ignored): `name` contains the type of observable we are using (here, we use 'test', it could also be 'sync' for synchrotron observables (e.g, Stokes parameters), 'fd' for Faraday Depth, etc. The `data_column` argument specifies the key or name of the column containing the relevant measurement. Coordinates can be either `cartesian` as in this example, which requires specifying columns for x, y and z in kpc, or `galactic`, which requires setting the arguments `lat_column` and `lon_column` both in degrees. The units of the dataset are represented using astropy.units objects and must be supplied (the Simulator will later check whether these are adequate and automatically convert the data to other units if needed).

The dataset can be loaded onto `Measurements` and `Covariances` object, which are subclasses of `ObservableDict`. These objects allow one to supply multiple datasets to the pipeline.

[4]:
```
mock_data = img.observables.Measurements() # create empty Measrurements object
mock_data.append(dataset=mockDataset)

mock_cov = img.observables.Covariances() # create empty Covariance object
mock_cov.append(dataset=mockDataset)
```

The dataset object creates a standard key for each appended dataset. In our case, there is only one key.

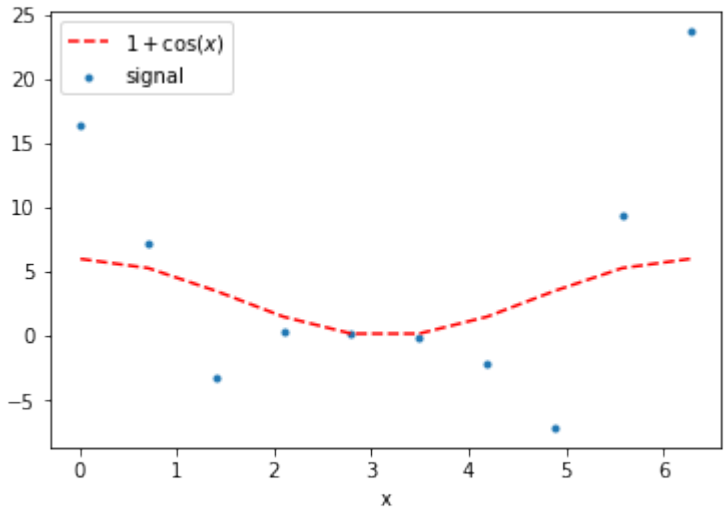
[5]:
```
keys = list(mock_data.keys())
keys
```

[5]:
```
[('test', 'nan', 'tab', 'nan')]
```

Let us plot the mock data as well as the $1 + \cos(x)$ function that is the underlying variation.

The property `Measurements.global_data` extracts arrays from the `Observable` object which is hosted inside the `ObservableDict` class.

[6]:
```
plt.scatter(x, mock_data[keys[0]].global_data[0], marker='.', label='signal')
plt.plot(x,(1+np.cos(x))*a0,'r--',label='$1+\cos(x)$')
plt.xlabel('x'); plt.legend();
```



Note that the variance in the signal is highest where the $\cos(x)$ is also strongest. This is the way we expect the Faraday depth to work, since a fluctuation in the strength of $\mathbf{B}$ has a larger effect on the RM when n_e also happens to be higher.

7.2 2) Pipeline assembly

Now that we have generated mock data, there are a few steps to set up the pipeline to estimate the input parameters. We need to specify: a grid, Field Factories, Simulators, and Likelihoods.

7.2.1 Setting the coordinate grid

Fields in IMAGINE represent models of any kind of physical field – in this particular tutorial, we will need a magnetic field and thermal electron density.

The Fields are evaluated on a grid of coordinates, represented by a `img.Grid` object. Here we exemplify how to produce a *regular cartesian* grid. To do so, we need to specify the values of the coordinates on the 6 extremities of the box (i.e. the minimum and maximum value for each coordinate), and the resolution over each dimension.

For this particular artificial example, we actually only need one dimension, so we set the resolution to 1 for y and z.

[7]:
```
one_d_grid = img.fields.UniformGrid(box=[[0,2*np.pi]*u.kpc,
                                         [0,0]*u.kpc,
                                         [0,0]*u.kpc],
                                    resolution=[30,1,1])
```

7.2.2 Preparing the Field Factory list

A particular realisation of a model for a physical field is represented within IMAGINE by a *Field* object, which, given set of parameters, evaluates the field for over the grid.

A *Field Factory* is an associated piece of infrastructure used by the Pipeline to produce new Fields. It is a Factory object that needs to be initialized and supplied to the Pipeline. This is what we will illustrate here.

[8]:
```
from imagine import fields
ne_factory = fields.CosThermalElectronDensityFactory(grid=one_d_grid)
```

The previous line instantiates `CosThermalElectronDensityFactory` with the previously defined Grid object. This Factory allows the Pipeline to produce `CosThermalElectronDensity` objects. These correspond to a toy model for electron density with the form:

$$n_e(x, y, z) = n_0[1 + \cos(ax + \alpha)][1 + \cos(by + \beta)][1 + \cos(cy + \gamma)]\,.$$

We can set and check the default parameter values in the following way:

[9]:
```
ne_factory.default_parameters= {'a': 1*u.rad/u.kpc,
                                'beta':  np.pi/2*u.rad,
                                'gamma': np.pi/2*u.rad}
ne_factory.default_parameters
```

[9]:
```
{'n0': <Quantity 1. 1 / cm3>,
 'a': <Quantity 1. rad / kpc>,
 'b': <Quantity 0. rad / kpc>,
 'c': <Quantity 0. rad / kpc>,
 'alpha': <Quantity 0. rad>,
 'beta': <Quantity 1.57079633 rad>,
 'gamma': <Quantity 1.57079633 rad>}
```

[10]:
```
ne_factory.active_parameters
```

```
[10]: ()
```

For `ne_factory`, no active parameters were set. This means that the Field will be always evaluated using the specified default parameter values.

We will now similarly define the magnetic field, using the `NaiveGaussianMagneticField` which constructs a "naive" random field (i.e. the magnitude of x, y and z components of the field are drawn from a Gaussian distribution **without** imposing *zero divergence*, thus *do not use this for serious applications*).

```
[11]: B_factory = fields.NaiveGaussianMagneticFieldFactory(grid=one_d_grid)
```

Differently from the case of `ne_factory`, in this case we would like to make the parameters active. All individual components of the field are drawn from a Gaussian distribution with mean a_0 and standard deviation b_0. To set these parameters as active we do:

```
[12]: B_factory.active_parameters = ('a0','b0')
      B_factory.priors ={'a0': img.priors.FlatPrior(interval=[-5,5]*u.microgauss),
                         'b0': img.priors.FlatPrior(interval=[0,10]*u.microgauss)}
```

In the lines above we chose uniform (flat) priors for both parameters within the above specified ranges. Any active parameter must have a Prior distribution specified.

Once the two FieldFactory objects are prepared, they put together in a list which is later supplied to the Pipeline.

```
[13]: factory_list = [ne_factory, B_factory]
```

7.2.3 Initializing the Simulator

For this tutorial, we use a customized TestSimulator which simply computes the quantity: $t(x, y, z) = B_y\, n_e$,i.e. the contribution at one specific point to the Faraday depth.

The simulator is inialized with the mock Measurements defined before, which allows it to know what is the correct format for output.

```
[14]: from imagine.simulators import TestSimulator
      simer = TestSimulator(mock_data)
```

7.2.4 Initializing the Likelihood

IMAGINE provides the `Likelihood` class with `EnsembleLikelihood` and `SimpleLikelihood` as two options. The `SimpleLikelihood` is what you expect, computing a single χ^2 from the difference of the simulated and the measured datasets. The `EnsembleLikelihood` is how IMAGINE handles a signal which itself includes a stochastic component, e.g., what we call the Galactic variance. This likelihood module makes use of a finite ensemble of simulated realizations and uses their mean and covariance to compare them to the measured dataset.

```
[15]: likelihood = img.likelihoods.EnsembleLikelihood(mock_data, mock_cov)
```

7.3 3) Running the pipeline

Now we have all the necessary components available to run our pipeline. This can be done through a `Pipeline` object, which interfaces with some algorithm to sample the likelihood space accounting for the prescribed prior distributions for the parameters.

IMAGINE comes with a range of samplers coded as different Pipeline classes, most of which are based on the nested sampling approach. In what follows we will use the UltraNest sampler as an example.

IMAGINE takes care of stochastic fields by evaluating an ensemble of random realisations for each selected point in the parameter space, and computing the associated covariance (i.e. estimating the Galactic variance). We can set this up through the `ensemble_size` argument.

Now we are ready to initialize our final pipeline.

[16]:
```
pipeline = img.pipelines.UltranestPipeline(simulator=simer,
                                           factory_list=factory_list,
                                           likelihood=likelihood,
                                           ensemble_size=150,
                                           chains_directory=None)
```

The `chains_directory` keyword may contain a path where the chains generated by the sampler are saved in the sampler's native format (if the sampler supports this). If this argument is absent or set to `None`, a temporary directory will be created in the *current working directory*, and will be automatically removed when together the Pipeline object becomes out of scope (e.g. when one runs `del pipeline` or exits the script/notebook). After initialization, we can check this choice inspecting the `chains_directory` property:

[17]:
```
pipeline.chains_directory
```

[17]:
```
'/home/lfsr/IMAGINE/imagine-pipeline/tutorials/imagine_chains_2i8qk58j'
```

The property `sampling_controllers` allows one to send sampler-specific parameters to the chosen Pipeline. Each IMAGINE Pipeline object will have slightly different sampling controllers, which can be found in the specific Pipeline's docstring.

[18]:
```
pipeline.sampling_controllers = {'max_ncalls': 500, 'min_num_live_points': 50}
```

The `UltranestPipeline`, for example, allows one to set a limit on the number of likelihood evaluations using the sampling controller `'max_ncalls'`). This is convenient for quick tests and examples (as this tutorial). Similar functionality is available in the `DynestyPipeline` under the name `'max_call'` (and also `'max_call_init'`/`'max_call_batch'` for dynamic nested sampling).

In a production run, however, one would rather set the target estimated uncertainty in the model evidence and/or target posterior uncertainty, and allow the model to converge to it. This can be done using the following sampling controllers (do check individual docs for details):

In `UltranestPipeline` this is done using `'dlogZ'` for target log-evidence error and `'dKL'` for target posterior error. In `MultinestPipeline` one can only controll the target log-evidence error, using `'evidence_tolerance'`. In `DynestyPipeline` the log-evidence error can be controlled using: `'dlogz'` (also `'dlogz_init'`), but the sampler can also try to optimize the estimate of the posterior while running in the *dynamic* mode.

Now, we *finally* can run the pipeline!

[19]:
```
results = pipeline()
```

```
[ultranest] PointStore: have 0 items
```

```
INFO:ultranest:PointStore: have 0 items
```

```
[ultranest] Sampling 50 live points from prior ...
```

```
INFO:ultranest:Sampling 50 live points from prior ...
DEBUG:ultranest:minimal_widths_sequence: [(-inf, 50.0), (inf, 50.0)]
```

```
[ultranest] Explored until L=-3e+01   [-2211.1490..-30.6137] | it/evals=144/500␣
↪eff=32.0000% N=50
```

```
INFO:ultranest:Explored until L=-3e+01
[ultranest] Likelihood function evaluations: 500
INFO:ultranest:Likelihood function evaluations: 500
[ultranest] Writing samples and results to disk ...
INFO:ultranest:Writing samples and results to disk ...
[ultranest] Writing samples and results to disk ... done
INFO:ultranest:Writing samples and results to disk ... done
DEBUG:ultranest:did a run_iter pass!
[ultranest] Reached maximum number of likelihood calls (500 > 500)...
INFO:ultranest:Reached maximum number of likelihood calls (500 > 500)...
[ultranest] done iterating.
INFO:ultranest:done iterating.
```

When one runs the pipeline it returns a results dictionary object in the native format of the chosen sampler. Alternatively, after running the pipeline object, the results can also be accessed through its attributes, which are standard interfaces (i.e. all pipelines should work in the same way).

For comparing different models, the quantity of interest is the *model evidence* or *marginal likelihood*. After a run, this can be easily accessed as follows.

[20]:
```
print('log evidence:', round(pipeline.log_evidence,4))
print('log evidence error:', round(pipeline.log_evidence_err,4))
```

```
log evidence: -32.3431
log evidence error: 0.382
```

A quick-and-dirty summary of the constraints on the parameters can be (nicely) displayed using the `posterior_report()` method.

(The "real" parameter values were $a_0 = 3\mu$G and $b_0 = 6\mu$G.)

[21]:
```
print('\nParameter constraints:')
pipeline.posterior_report()
```

```

Parameter constraints:
```

Similar information can be accessed through property `posterior_summary`, which returns a dictionary.

[22]:
```
pipeline.posterior_summary
```

[22]:
```
{'naive_gaussian_magnetic_field_a0': {'median': <Quantity 2.70238735 uG>,
  'errlo': <Quantity 2.41938506 uG>,
  'errup': <Quantity 1.31069702 uG>,
  'mean': <Quantity 2.25622256 uG>,
  'stdev': <Quantity 1.85700372 uG>},
 'naive_gaussian_magnetic_field_b0': {'median': <Quantity 5.65438118 uG>,
  'errlo': <Quantity 1.03419168 uG>,
  'errup': <Quantity 1.66290483 uG>,
  'mean': <Quantity 5.83415716 uG>,
  'stdev': <Quantity 1.30901039 uG>}}
```

In most cases, however, one would (should) prefer to work directly on the samples produced by the sampler. A table containing the parameter values of the samples generated can be accessed through:

[23]:
```
samples = pipeline.samples
samples[:3] # Displays only first 3 rows
```

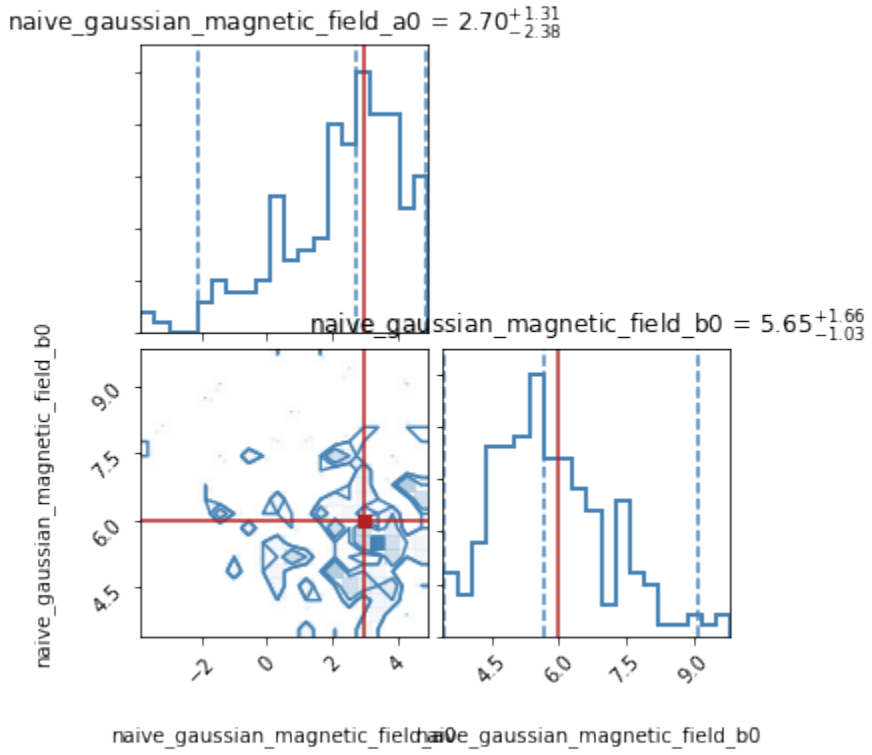
[23]:
```
<QTable length=3>
naive_gaussian_magnetic_field_a0 naive_gaussian_magnetic_field_b0
                uG                               uG
             float64                          float64
------------------------------- -------------------------------
                4.081100349132784               6.717489837154965
               0.12377231861508164               6.231348024468215
                3.830092802478381               4.537991481720585
```

The distributions of samples approximate the posterior distribution, below this is plotted with the help of the corner library, whcih also shows the best-fit values and 1σ and 2σ contours.

[24]:
```
def plot_samples_corner(samp):

    ##  See https://corner.readthedocs.io/en/latest/pages/sigmas.html about contour␣
↪levels.
    ##  "Contours are shown at 0.5, 1, 1.5, and 2 sigma" by default
    ##  according to https://pypi.org/project/corner/1.0.1/, but I want 1, 2, and 3.
    sigmas=np.array([1.,2.,3.])
    levels=1-np.exp(-0.5*sigmas*sigmas)

    # Visualize with a corner plot
    figure = corner.corner(np.vstack([samp.columns[0].value, samp.columns[1].value]).
↪T,
                           range=[0.99]*len(samp.colnames),
                           quantiles=[0.02, 0.5, 0.98],
                           labels=samp.colnames,
                           show_titles=True,
                           title_kwargs={"fontsize": 12},
                           color='steelblue',
                           truths=[a0,b0],
                           truth_color='firebrick',
                           plot_contours=True,
                           hist_kwargs={'linewidth': 2},
                           label_kwargs={'fontsize': 10},
                           levels=levels)
```

[25]:
```
plot_samples_corner(samples)
```

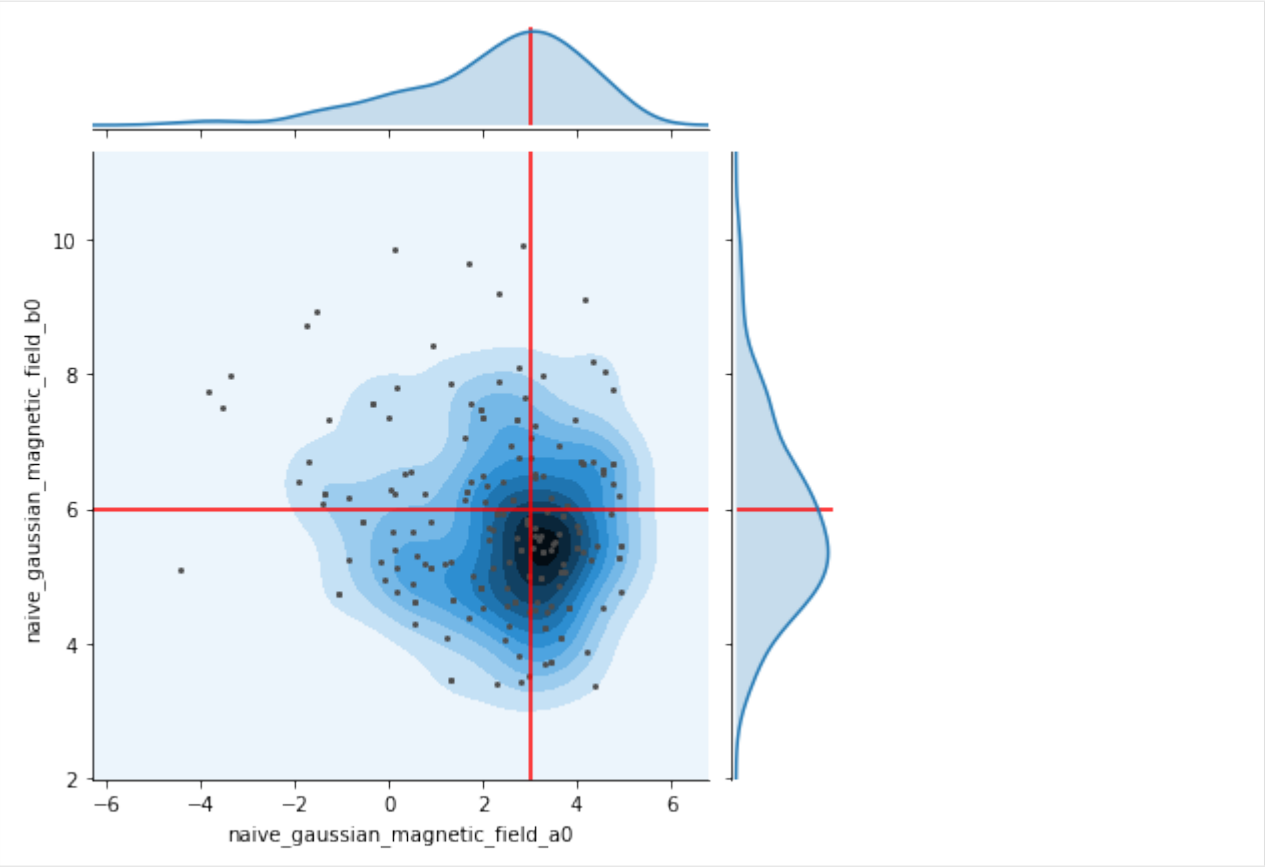
```
WARNING:root:Too few points to create valid contours
```



One can, of course, choose other plotting/analysis routines. Below, the use of seaborn is exemplified.

[26]:
```
import seaborn as sns
def plot_samples_seaborn(samp):

    def show_truth_in_jointplot(jointplot, true_x, true_y, color='r'):
        for ax in (jointplot.ax_joint, jointplot.ax_marg_x):
            ax.vlines([true_x], *ax.get_ylim(), colors=color)
        for ax in (jointplot.ax_joint, jointplot.ax_marg_y):
            ax.hlines([true_y], *ax.get_xlim(), colors=color)

    snsfig = sns.jointplot(*samp.colnames, data=samp.to_pandas(), kind='kde')
    snsfig.plot_joint(sns.scatterplot, linewidth=0, marker='.', color='0.3')
    show_truth_in_jointplot(snsfig, a0, b0)
```

[27]:
```
plot_samples_seaborn(samples)
```



7.4 Random seeds and convergence checks

The pipeline relies on random numbers in multiple ways. The Monte Carlo sampler will draw randomly chosen points in the parameter space duing its exploration (in the specific case of *nested sampling* pipelines, these are drawn from the prior distributions). Also, while evaluating the fields at each point, random realisations of the stochastic fields are generated.

It is possible to control the behaviour of the random seeding of an IMAGINE pipeline through the attribute `master_seed`. This attribute has two uses: it is passed to the sampler, ensuring that its behaviour is reproducible; and it is also used to generate a fresh list of new random seeds to each stochastic field that is evaluated.

[28]:
```
pipeline.master_seed
```

[28]:
```
1
```

By default, the master seed is fixed and set to 1, but you can alter its value before running the pipeline.

One can also change the seeding behaviour through the `random_type` attribute. There are three allowed options for this:

- 'controllable' - the `master_seed` is constant and a re-running the pipeline should lead to the exact same results (default);
- 'free' - on each execution, a new `master_seed` is drawn (using numpy.randint);
- 'fixed' - this mode is for debugging purposes. The `master_seed` is fixed, as in the 'controllable' case, however each individual stochastic field receives the exact same list of ensemble seeds every time (while in the

`controllable` these are chosen "randomly" at run-time). Such choice can be inspected using `pipeline.ensemble_seeds`.

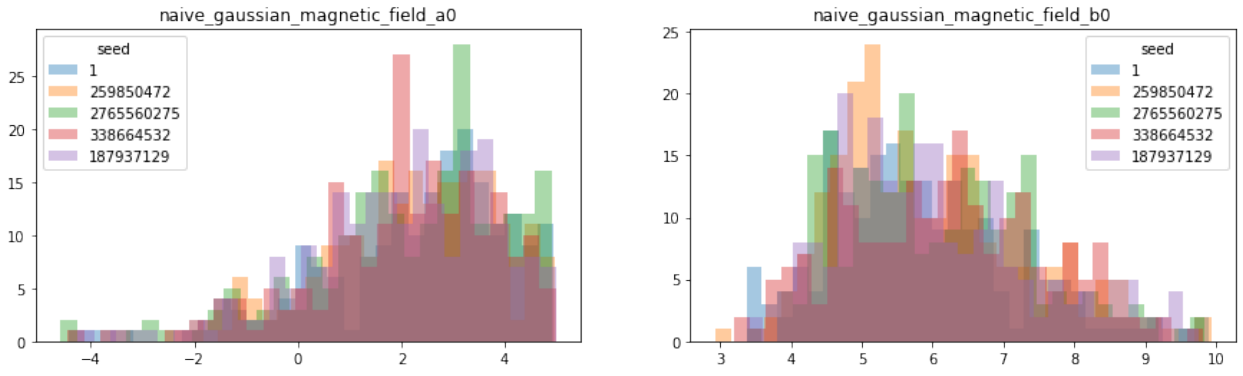
Let us now check whether different executions of the pipeline are generating consistent results. To do so, we run it five times and just overplot histograms of the outputs to see if they all look the same. There are more rigorous tests, of course, that we have done, but they take longer. The following can be done in a few minutes.

[29]:
```
# Adjust the random behaviour
pipeline.random_type = 'free'

fig, axs = plt.subplots(1, 2, figsize=(15, 4))
repeat = 5

for i in range(repeat):
    if i>0:
        print('-'*60+'\nRun {}/{}'.format(i+1,repeat))
        # Re-runs the pipeline!
        _ = pipeline()

    for j, param in enumerate(pipeline.samples.columns):
        samp = pipeline.samples[param]
        axs[j].hist(samp.value, alpha=0.4, bins=30, label=pipeline.master_seed)
        axs[j].set_title(param)

for i in range(2):
    axs[i].legend(title='seed')
```

```
------------------------------------------------------------
Run 2/5
[ultranest] PointStore: have 0 items
INFO:ultranest:PointStore: have 0 items
[ultranest] Sampling 50 live points from prior ...
INFO:ultranest:Sampling 50 live points from prior ...
DEBUG:ultranest:minimal_widths_sequence: [(-inf, 50.0), (inf, 50.0)]
[ultranest] Explored until L=-3e+01   [-31.0469..-29.8293] | it/evals=169/501 eff=37.
↪4723% N=50 0
INFO:ultranest:Explored until L=-3e+01
[ultranest] Likelihood function evaluations: 501
INFO:ultranest:Likelihood function evaluations: 501
[ultranest] Writing samples and results to disk ...
INFO:ultranest:Writing samples and results to disk ...
[ultranest] Writing samples and results to disk ... done
INFO:ultranest:Writing samples and results to disk ... done
DEBUG:ultranest:did a run_iter pass!
[ultranest] Reached maximum number of likelihood calls (501 > 500)...
INFO:ultranest:Reached maximum number of likelihood calls (501 > 500)...
[ultranest] done iterating.
INFO:ultranest:done iterating.
------------------------------------------------------------
Run 3/5
[ultranest] PointStore: have 0 items
```

```
INFO:ultranest:PointStore: have 0 items
[ultranest] Sampling 50 live points from prior ...
INFO:ultranest:Sampling 50 live points from prior ...
DEBUG:ultranest:minimal_widths_sequence: [(-inf, 50.0), (inf, 50.0)]
[ultranest] Explored until L=-3e+01   [-30.9045..-29.7086] | it/evals=168/502 eff=37.
↪1681% N=50 0
INFO:ultranest:Explored until L=-3e+01
[ultranest] Likelihood function evaluations: 502
INFO:ultranest:Likelihood function evaluations: 502
[ultranest] Writing samples and results to disk ...
INFO:ultranest:Writing samples and results to disk ...
[ultranest] Writing samples and results to disk ... done
INFO:ultranest:Writing samples and results to disk ... done
DEBUG:ultranest:did a run_iter pass!
[ultranest] Reached maximum number of likelihood calls (502 > 500)...
INFO:ultranest:Reached maximum number of likelihood calls (502 > 500)...
[ultranest] done iterating.
INFO:ultranest:done iterating.
------------------------------------------------------------
Run 4/5
[ultranest] PointStore: have 0 items
INFO:ultranest:PointStore: have 0 items
[ultranest] Sampling 50 live points from prior ...
INFO:ultranest:Sampling 50 live points from prior ...
DEBUG:ultranest:minimal_widths_sequence: [(-inf, 50.0), (inf, 50.0)]
[ultranest] Explored until L=-3e+01   [-680.4351..-30.5342] | it/evals=154/523 eff=32.
↪5581% N=50
INFO:ultranest:Explored until L=-3e+01
[ultranest] Likelihood function evaluations: 523
INFO:ultranest:Likelihood function evaluations: 523
[ultranest] Writing samples and results to disk ...
INFO:ultranest:Writing samples and results to disk ...
[ultranest] Writing samples and results to disk ... done
INFO:ultranest:Writing samples and results to disk ... done
DEBUG:ultranest:did a run_iter pass!
[ultranest] Reached maximum number of likelihood calls (523 > 500)...
INFO:ultranest:Reached maximum number of likelihood calls (523 > 500)...
[ultranest] done iterating.
INFO:ultranest:done iterating.
```

```
------------------------------------------------------------
Run 5/5
[ultranest] PointStore: have 0 items
INFO:ultranest:PointStore: have 0 items
[ultranest] Sampling 50 live points from prior ...
INFO:ultranest:Sampling 50 live points from prior ...
DEBUG:ultranest:minimal_widths_sequence: [(-inf, 50.0), (inf, 50.0)]
[ultranest] Explored until L=-3e+01   [-3169.7507..-30.2520] | it/evals=169/502␣
↪eff=37.3894% N=50
INFO:ultranest:Explored until L=-3e+01
[ultranest] Likelihood function evaluations: 502
INFO:ultranest:Likelihood function evaluations: 502
[ultranest] Writing samples and results to disk ...
INFO:ultranest:Writing samples and results to disk ...
[ultranest] Writing samples and results to disk ... done
INFO:ultranest:Writing samples and results to disk ... done
DEBUG:ultranest:did a run_iter pass!
[ultranest] Reached maximum number of likelihood calls (502 > 500)...
INFO:ultranest:Reached maximum number of likelihood calls (502 > 500)...
[ultranest] done iterating.
INFO:ultranest:done iterating.
```



7.5 Script example

A script version of this tutorial can be found in the examples directory.

CHAPTER 8

Including new observational data

In this tutorial we will see how to load observational data onto IMAGINE.

Both observational and simulated data are manipulated within IMAGINE through *observable dictionaries*. There are three types of these: `Measurements`, `Simulations` and `Covariances`, which can store multiple entries of observational, simulated and covariance (either real or mock) data, respectively. Appending data to an `ObservableDict` requires following some requirements regarding the data format, therefore we recomend the use of one of the `Dataset` classes.

8.1 HEALPix Datasets

Let us illustrate how to prepare an IMAGINE dataset with the Faraday depth map obtained by Oppermann et al. 2012 (arXiv:1111.6186).

The following snippet will download the data (a ~4MB FITS file) to RAM and open it.

[1]:
```
import requests, io
from astropy.io import fits

download = requests.get('https://wwwmpa.mpa-garching.mpg.de/ift/faraday/2012/faraday.
↪fits')
raw_dataset = fits.open(io.BytesIO(download.content))
raw_dataset.info()
```

```
Filename: <class '_io.BytesIO'>
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       7   ()
  1  TEMPERATURE    1 BinTableHDU     17   196608R x 1C   [E]
  2  signal uncertainty    1 BinTableHDU     17   196608R x 1C   [E]
  3  Faraday depth    1 BinTableHDU     17   196608R x 1C   [E]
  4  Faraday uncertainty    1 BinTableHDU     17   196608R x 1C   [E]
  5  galactic profile    1 BinTableHDU     17   196608R x 1C   [E]
  6  angular power spectrum of signal    1 BinTableHDU     12   384R x 1C   [E]
```

Now we will feed this to an IMAGINE `Dataset`. It requires converting the data into a proper numpy array of floats. To allow this notebook running on a small memory laptop, we will also reduce the size of the arrays (only taking 1 value every 256).

[2]:
```
from imagine.observables import FaradayDepthHEALPixDataset
import numpy as np
from astropy import units as u

# Adjusts the data to the right format
fd_raw = raw_dataset[3].data.astype(np.float)
sigma_fd_raw = raw_dataset[4].data.astype(np.float)
# Makes it small, to save memory
fd_raw = fd_raw[::256]
sigma_fd_raw = sigma_fd_raw[::256]
# We need to include units the data
# (this avoids later errors and inconsistencies)
fd_raw *= u.rad/u.m/u.m
sigma_fd_raw *= u.rad/u.m/u.m
# Loads into a Dataset
dset = FaradayDepthHEALPixDataset(data=fd_raw, error=sigma_fd_raw)
```

One important assumption in the previous code-block is that the covariance matrix is diagonal, i.e. that the errors in FD are *uncorrelated*. If this is not the case, instead of initializing the `FaradayDepthHEALPixDataset` with `data` and `error`, one should initialize it with `data` and `cov`, where the latter is the correct covariance matrix.

To keep things organised and useful, we *strongly recommend* to create a personalised `dataset` class and make it available to the rest of the consortium in the imagine-datasets GitHub repository. An example of such a class is the following:

[3]:
```
import requests, io
import numpy as np
from astropy.io import fits
from astropy import units as u
from imagine.observables import FaradayDepthHEALPixDataset

class FaradayDepthOppermann2012(FaradayDepthHEALPixDataset):
    def __init__(self, skip=None):
        # Fetches and reads the
        download = requests.get('https://wwwmpa.mpa-garching.mpg.de/ift/faraday/2012/
↪faraday.fits')
        raw_dataset = fits.open(io.BytesIO(download.content))
        # Adjusts the data to the right format
        fd_raw = raw_dataset[3].data.astype(np.float)
        sigma_fd_raw = raw_dataset[4].data.astype(np.float)
        # Includes units in the data
        fd_raw *= u.rad/u.m/u.m
        sigma_fd_raw *= u.rad/u.m/u.m

        # If requested, makes it small, to save memory in this example
        if skip is not None:
            fd_raw = fd_raw[::skip]
            sigma_fd_raw = sigma_fd_raw[::skip]
        # Loads into the Dataset
        super().__init__(data=fd_raw, error=sigma_fd_raw)
```

With this pre-programmed, anyone will be able to load this into the pipeline by simply doing

[4]:
```
dset = FaradayDepthOppermann2012(skip=256)
```

Once we have a dataset, we can load this into a Measurements and Covariances objects (which will be discussed a further down).

[5]:
```
from imagine.observables import Measurements, Covariances

# Creates the instances
mea = Measurements()
cov = Covariances()

# Appends the data
mea.append(dataset=dset)
cov.append(dataset=dset)
```

8.2 Tabular Datasets

So far, we looked into datasets comprising HEALPix maps. One may also want to work with *tabular* datasets. We exemplify this fetching and preparing a RM catalogue of Mao et al 2010 (arXiv:1003.4519). In the case of this particular dataset, we can import the data from VizieR using the `astroquery` library.

[6]:
```
import astroquery
from astroquery.vizier import Vizier

# Fetches the relevant catalogue from Vizier
# (see https://astroquery.readthedocs.io/en/latest/vizier/vizier.html for details)
catalog = Vizier.get_catalogs('J/ApJ/714/1170')[0]
catalog[:3] # Shows only first rows
```

[6]:
```
<Table length=3>
  RAJ2000     DEJ2000      GLON     GLAT   ...     I      S5.2  f_S5.2  NVSS
  "h:m:s"     "d:m:s"      deg      deg    ...    mJy
  bytes11     bytes11    float32  float32  ... float64  bytes3 bytes1 bytes4
----------- ----------- ------- ------- ... -------- ------ ------ ------
13 07 08.33 +24 47 00.7    0.21   85.76 ...   131.49    Yes           NVSS
13 35 48.14 +20 10 16.0    0.86   77.70 ...    71.47     No      b    NVSS
13 24 14.48 +22 13 13.1    1.33   81.08 ...   148.72    Yes           NVSS
```

The procedure for converting this to an IMAGINE data set is the following:

[7]:
```
from imagine.observables import TabularDataset
dset_tab = TabularDataset(catalog, name='fd', tag=None,
                          units= u.rad/u.m/u.m,
                          data_column='RM', error_column='e_RM',
                          lat_column='GLAT', lon_column='GLON')
```

`catalog` must be a dictionary-like object (e.g. `dict`, `astropy.Tables`, `pandas.DataFrame`) and data(/error/lat/lon)_column specify the key/column-name used to retrieve the relevant data from `catalog`. The `name` argument specifies the type of measurement that is being stored. This has to agree with the requirements of the Simulator you will use. Some standard available observable names are:

- 'fd' - Faraday depth
- 'sync' - Synchrotron emission, needs the `tag` to be interpreted
  - tag = 'I' - Total intensity

  - tag = 'Q' - Stokes Q
  - tag = 'U' - Stokes U
  - tag = 'PI' - polarisation intensity
  - tag = 'PA' - polarisation angle
- 'dm' - Dispersion measure

The units are provided as an `astropy.units.Unit` object and are converted internally to the requirements of the specific Simulator being used.

Again, the procedure can be packed and distributed to the community in a (very short!) personalised class:

[8]:
```
from astroquery.vizier import Vizier
from imagine.observables import TabularDataset

class FaradayRotationMao2010(TabularDataset):
    def __init__(self):
        # Fetches the catalogue
        catalog = Vizier.get_catalogs('J/ApJ/714/1170')[0]
        # Reads it to the TabularDataset (the catalogue obj actually contains units)
        super().__init__(catalog, name='fd', units=catalog['RM'].unit,
                         data_column='RM', error_column='e_RM',
                         lat_column='GLAT', lon_column='GLON')
```

[9]:
```
dset_tab = FaradayRotationMao2010() # ta-da!
```

8.3 Measurements and Covariances

Again, we can include these observables in our `Measurements` and `Covariances` objects. These are dictionary-like object, i.e. given a key, one can access a given item.

[10]:
```
mea.append(dataset=dset_tab)
cov.append(dataset=dset_tab)

print('Measurement keys:')
for k in mea.keys():
    print('\t',k)
print('\nCovariance keys:')
for k in cov.keys():
    print('\t',k)
```

```
Measurement keys:
         ('fd', 'nan', '8', 'nan')
         ('fd', 'nan', 'tab', 'nan')

Covariance keys:
         ('fd', 'nan', '8', 'nan')
         ('fd', 'nan', 'tab', 'nan')
```

The keys follow a strict convention: `(data-name,str(data-freq),Nside/'tab',str(ext))`

- If data is independent from frequency, data-freq is set 'nan', otherwise it is a *string* showing the frequency in GHz.
- The third value in the key-tuple is the HEALPix Nside (for maps) or the string 'tab' for tabular data.

- Finally, the last value, `ext` can be 'I','Q','U','PI','PA', 'nan' or other customized tags depending on the nature of the observable.

An alternative way to include data into an Observables dictionary is explicitly choosing the key and adjusting the data shape. One can see how this is handled by the Dataset object in the following cell

[11]:
```
# This is how HEALPix data can be included without the mediation of Datasets:
mea.append(name=dset.key, new_data=dset.data)
cov.append(name=dset.key, new_data=dset.cov)
# This is what Dataset is doing:
print('The key used in the "name" arg was:', dset.key)
print('The shape of data used in the "new" arg was:', dset.data.shape)
```

```
The key used in the "name" arg was: ('fd', 'nan', '8', 'nan')
The shape of data used in the "new" arg was: (1, 768)
```

But what exactly is stored in `mea` and `cov`? This is handled by an `Observable` object. Here we illustrate with the tabular dataset previously defined.

[12]:
```
print(type(mea[dset_tab.key]))
print('mea.data:', repr(mea[dset_tab.key].data))
print('mea.data.shape:', mea[dset_tab.key].data.shape)
print('mea.unit', repr(mea[dset_tab.key].unit))
print('mea.coords (coordinates dict -- for tabular datasets only):\n',
      mea[dset_tab.key].coords)
print('mea.dtype:', mea[dset_tab.key].dtype)
print('\n\ncov type',type(cov[dset_tab.key]))
print('cov.data type:', type(cov[dset_tab.key].data))
print('cov.data.shape:', cov[dset_tab.key].data.shape)
print('cov.dtype:', cov[dset_tab.key].dtype)
```

```
<class 'imagine.observables.observable.Observable'>
mea.data: array([[ -3.,   3.,  -6.,   0.,   4.,  -6.,   5.,  -1.,  -6.,   1., -14.,
         14.,   3.,  10.,  12.,  16.,  -3.,   5.,  12.,   6.,  -1.,   5.,
          5.,  -4.,  -1.,   9.,  -1., -13.,   4.,   5.,  -5.,  17., -10.,
          8.,   0.,   1.,   5.,   9.,   5., -12., -17.,   4.,   2.,  -1.,
         -3.,   3.,  16.,  -1.,  -1.,   3.]])
mea.data.shape: (1, 50)
mea.unit Unit("rad / m2")
mea.coords (coordinates dict -- for tabular datasets only):
 {'type': 'galactic', 'lon': <Quantity [ 0.21,  0.86,  1.33,  1.47,  2.1 ,  2.49,  3.
↪11,  3.93,  4.17,
            5.09,  5.09,  5.25,  5.42,  6.36, 12.09, 12.14, 13.76, 13.79,
           14.26, 14.9 , 17.1 , 20.04, 20.56, 20.67, 20.73, 20.85, 21.83,
           22.  , 22.13, 22.24, 22.8 , 23.03, 24.17, 24.21, 24.28, 25.78,
           25.87, 28.84, 28.9 , 30.02, 30.14, 30.9 , 31.85, 34.05, 34.37,
           34.54, 35.99, 37.12, 37.13, 37.2 ] deg>, 'lat': <Quantity [85.76, 77.7 ,␣
↪81.08, 81.07, 83.43, 82.16, 84.8 , 78.53, 85.81,
           79.09, 81.5 , 79.69, 80.61, 80.85, 79.87, 81.64, 77.15, 77.12,
           82.52, 84.01, 80.26, 85.66, 82.42, 77.73, 78.94, 80.75, 80.77,
           80.77, 82.72, 80.24, 77.17, 82.19, 82.62, 81.42, 79.25, 85.63,
           81.66, 78.74, 78.72, 79.92, 89.52, 77.32, 87.09, 86.11, 85.04,
           85.05, 78.73, 79.1 , 80.74, 84.35] deg>}
mea.dtype: measured


cov type <class 'imagine.observables.observable.Observable'>
cov.data type: <class 'numpy.ndarray'>
cov.data.shape: (50, 50)
```

(continues on next page)

(continued from previous page)

```
cov.dtype: covariance
```

The `Dataset` object may also automatically distribute the data across different nodes if one is running the code using MPI parallelisation – a strong reason for sticking to using `Datasets` instead of appending directly.

CHAPTER 9

Fields and Factories

Within the IMAGINE pipeline, spatially varying physical quantities are represented by Field objects. This can be a *scalar*, as the number density of thermal electrons, or a *vector*, as the Galactic magnetic field.

In order to extend or personalise adding in one's own model for an specific field, one needs to follow a small number of simple steps:

1. choose a **coordinate grid** where your model will be evaluated,
2. write a **field class**, and
3. write a **field factory** class.

The **field objects** will do the actual computation of the physical field, given a set of physical parameters and a coordinate grid. The **field factory objects** take care of book-keeping tasks: e.g. they hold the parameter ranges and default values, and translate the dimensionless parameters used by the sampler (always in the interval $[0, 1]$) to physical parameters, and hold the prior information on the parameter values.

9.1 Coordinate grid

You can create your own coordinate grid by subclassing `imagine.fields.grid.BaseGrid`. The only thing which has to actually be programmed in the new sub-class is a method overriding `generate_coordinates()`, which produces a dictionary of numpy arrays containing coordinates in *either* cartesian, cylindrical or spherical coordinates (generally assumed, in Galactic contexts, to be centred in the centre of the Milky Way).

Typically, however, it is sufficient to use a simple grid with coordinates uniformly-spaced in cartesian, spherical or cylindrical coordinates. This can be done using the `UniformGrid` class. `UniformGrid` objects are initialized with the arguments: `box`, which contains the ranges of each coordinate in `kpc` or `rad`; `resolution`, a list of integers containing the number of grid points on each dimension; and `grid_type`, which can be either 'cartesian' (default), 'cylindrical' or 'spherical'.

[1]:
```
from imagine.fields import UniformGrid
import numpy as np
import astropy.units as u
```

(continues on next page)

(continued from previous page)

```
# Fixes numpy seed to ensure this notebook leads always to the same results
np.random.seed(42)

# A cartesian grid can be constructed as follows
cartesian_grid = UniformGrid(box=[[-15*u.kpc, 15*u.kpc],
                                  [-15*u.kpc, 15*u.kpc],
                                  [-15*u.kpc, 15*u.kpc]],
                             resolution = [15,15,15])

# For cylindrical grid, the limits are specified assuming
# the order: r (cylindrical), phi, z
cylindrical_grid = UniformGrid(box=[[0.25*u.kpc, 15*u.kpc],
                                    [-180*u.deg, np.pi*u.rad],
                                    [-15*u.kpc, 15*u.kpc]],
                       resolution = [9,12,9],
                       grid_type = 'cylindrical')

# For spherical grid, the limits are specified assuming
# the order: r (spherical), theta, phi (azimuth)
spherical_grid = UniformGrid(box=[[0*u.kpc, 15*u.kpc],
                                  [0*u.rad, np.pi*u.rad],
                                  [-np.pi*u.rad,np.pi*u.rad]],
                       resolution = [12,10,10],
                       grid_type = 'spherical')
```

The grid object will produce the grid only when the a coordinate value is first accessed, through the properties 'x', 'y','z','r_cylindrical','r_spherical', 'theta' and 'phi'.

The grid object also takes care of any coordinate conversions that are needed, for example:

[2]:
```
print(spherical_grid.x[5,5,5], cartesian_grid.r_spherical[5,5,5])
```

```
6.309658489079476 kpc 7.423074889580904 kpc
```

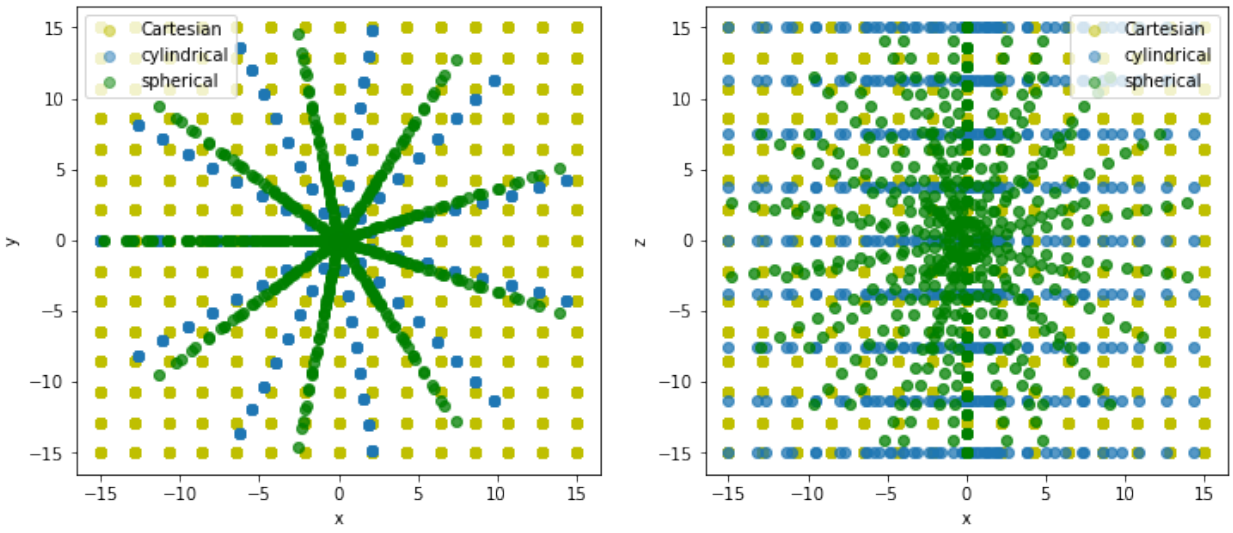
In the following figure we illustrate the effects of different choices of 'grid_type' while using UniformGrid.

(Note that, for plotting purposes, everything is converted to cartesian coordinates)

[3]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)
plt.scatter(cartesian_grid.x, cartesian_grid.y, color='y', label='Cartesian', alpha=0.
↪5)
plt.scatter(cylindrical_grid.x, cylindrical_grid.y, label='cylindrical', alpha=0.5)
plt.scatter(spherical_grid.x, spherical_grid.y, color='g', label='spherical', alpha=0.
↪5)
plt.xlabel('x'); plt.ylabel('y')
plt.legend()

plt.subplot(1,2,2)
plt.scatter(cartesian_grid.x, cartesian_grid.z, color='y', label='Cartesian', alpha=0.
↪5)
plt.scatter(cylindrical_grid.x, cylindrical_grid.z, label='cylindrical', alpha=0.5)
plt.scatter(spherical_grid.x, spherical_grid.z, label='spherical', color='g', alpha=0.
↪5)
plt.xlabel('x'); plt.ylabel('z')
plt.legend();
```

(continues on next page)

(continued from previous page)



9.2 Field objects

As we mentioned before, `Field` objects handle the calculation of any physical field.

To ensure that your new personalised field is compatible with any simulator, it needs to be a subclass of one of the pre-defined field classes. Some examples of which are:

- `MagneticField`
- `ThermalElectronDensity`
- `CosmicRayDistribution`

Let us illustrate this by defining a thermal electron number density field which decays exponentially with cylindrical radius, R,

$$n_e(R) = n_{e,0} e^{-R/R_e} e^{-|z|/h_e}$$

This has three parameters: the number density of thermal electrons at the centre, $n_{e,0}$, the scale radius, R_e, and the scale height, h_e.

[4]:
```
from imagine.fields import ThermalElectronDensityField

class ExponentialThermalElectrons(ThermalElectronDensityField):
    """Example: thermal electron density of an (double) exponential disc"""

    NAME = 'exponential_disc_thermal_electrons'

    @property
    def field_checklist(self):
        return {'central_density' : None,
                'scale_radius' : None,
                'scale_height' : None}
```

(continues on next page)

(continued from previous page)

```
    def compute_field(self, seed):
        R = self.grid.r_cylindrical
        z = self.grid.z
        Re = self.parameters['scale_radius']
        he = self.parameters['scale_height']
        n0 = self.parameters['central_density']

        return n0*np.exp(-R/Re)*np.exp(-np.abs(z/he))
```

With these few lines we have created our IMAGINE-compatible™ thermal electron density field class!

The class-attribute `field_name` allows one to keep track of which model we have used to generate our field.

The `stochastic field` class-attribute tells whether the field is deterministic (i.e. the output depends only on the parameter values) or stochastic (i.e. the ouput is a random realisation which depends on a particular random seed). In this particular case we construct a deterministic field.

The `field_checklist` property is a dictionary whose keys are required parameters for this particular kind of field. The values in the dictionary can be used for specialized checking by some simulators (but can be left as `None` in the general case).

The function `compute_field` is what actually computes the density field. Note that it can access an associated grid object, which is stored in the `grid` attribute, and a dictionary of parameters, stored in the `parameters` attribute. The `compute_field` method takes a `seed` argument, which can is only used for stochastic fields (see later).

Let us now see this at work. First, let us creat an instance of `ExponentialThermalElectrons`. Any Field instance should be initialized providing a Grid object and a dictionary of parameters.

[5]:
```
electron_distribution = ExponentialThermalElectrons(
    parameters={'central_density': 1.0*u.cm**-3,
                'scale_radius': 3.3*u.kpc,
                'scale_height': 3.3*u.kpc},
    grid=cartesian_grid)
```

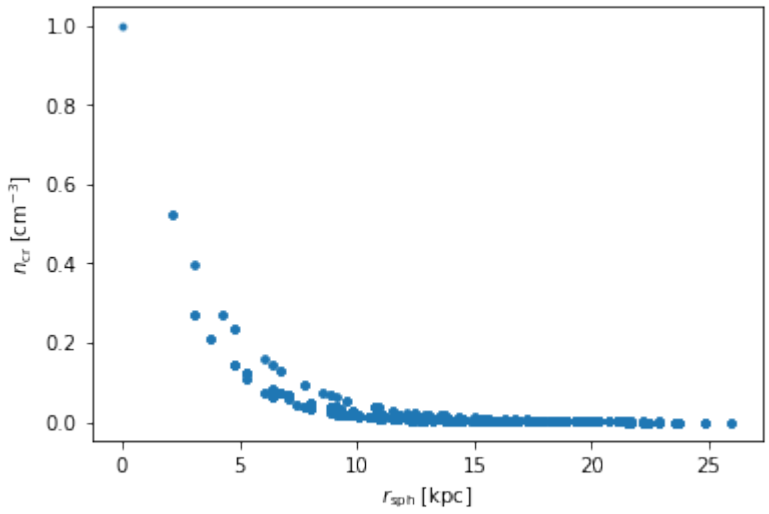
We can access the field produced by `cr_distribution` using the `get_data()` method (it invokes `compute_field` internally and does any checking required). For example:

[6]:
```
ne_data = electron_distribution.get_data()
print('data is', type(ne_data), 'of length', ne_data.shape)
print('an example slice of it is:')
ne_data[3:5,3:5,3:5]
```

```
data is <class 'astropy.units.quantity.Quantity'> of length (15, 15, 15)
an example slice of it is:
```

If we now wanted to plot the thermal electron density computed by this as a function of, say, *spherical radius*, $r_{\rm sph}$. This can be done in the following way

[7]:
```
# The spherical radius can be read from the grid object
rspherical = electron_distribution.grid.r_spherical

plt.plot(rspherical.ravel(), ne_data.ravel(), '.')
plt.xlabel(r'$r_{\rm sph}\;[\rm kpc]$'); plt.ylabel(r'$n_{\rm cr}\;[\rm cm^{-3}]$');
```



Let us do another simple field example: a constant magnetic field.

It follows the same basic template.

[8]:
```
from imagine.fields import MagneticField

class ConstantMagneticField(MagneticField):
    """Example: constant magnetic field"""
    NAME = 'constantB'

    @property
    def field_checklist(self):
        return {'Bx': None, 'By': None, 'Bz': None}

    def compute_field(self, seed):
        # Creates an empty array to store the result
        B = np.empty(self.data_shape) * self.parameters['Bx'].unit
        # For a magnetic field, the output must be of shape:
        # (Nx,Ny,Nz,Nc) where Nc is the index of the component.
        # Computes Bx
        B[:,:,:,0] = self.parameters['Bx']*np.ones(self.grid.shape)
        # Computes By
        B[:,:,:,1] = self.parameters['By']*np.ones(self.grid.shape)
        # Computes Bz
        B[:,:,:,2] = self.parameters['Bz']*np.ones(self.grid.shape)
        return B
```

The main difference from the thermal electrons case is that the shape of the final array has to accomodate all the three components of the magnetic field.

As before, we can generate a realisation of this

[9]:
```
p = {'Bx': 1.5*u.microgauss, 'By': 1e-10*u.Tesla, 'Bz': 0.1e-6*u.gauss}
B = ConstantMagneticField(parameters=p, grid=cartesian_grid)
```

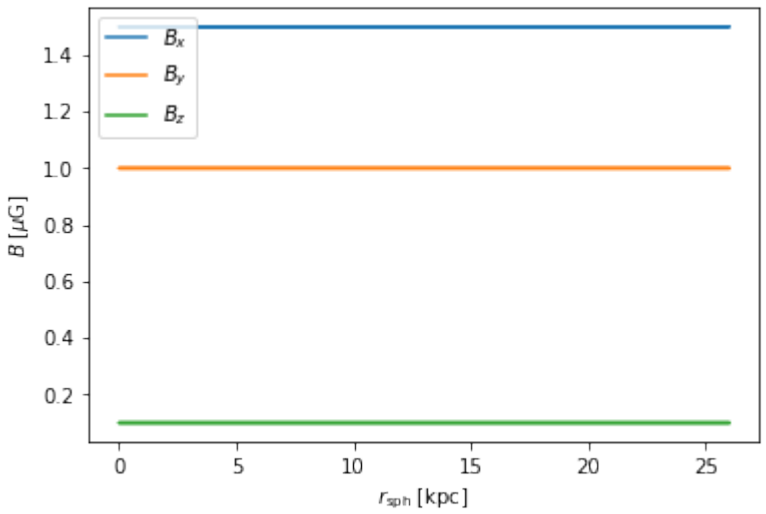
And inspect how it went

[10]:
```
r_spherical = B.grid.r_spherical.ravel()
B_data = B.get_data()
```

(continues on next page)

(continued from previous page)

```
for i, name in enumerate(['x','y','z']):
    plt.plot(r_spherical, B_data[...,i].ravel(),
             label='$B_{}$'.format(name))
plt.xlabel(r'$r_{\rm sph}\;[\rm kpc]$'); plt.ylabel(r'$B\;[\mu\rm G]$')
plt.legend();
```



More information about the field can be found inspecting the object

[11]:
```
print('Field type: ', B.type)
print('Data shape: ', B.data_shape)
print('Units: ', B.units)
print('What is each axis? Answer:', B.data_description)
```

```
Field type:  magnetic_field
Data shape:  (15, 15, 15, 3)
Units:  uG
What is each axis? Answer: ['grid_x', 'grid_y', 'grid_z', 'component (x,y,z)']
```

Let us now exemplify the construction of a stochastic field with a thermal electron density comprising random fluctuations.

[12]:
```
from imagine.fields import ThermalElectronDensityField
import scipy.stats as stats

class RandomThermalElectrons(ThermalElectronDensityField):
    """Example: Gaussian random thermal electron density

    NB this may lead to negative ne depending on the choice of
    parameters.
    """

    NAME = 'random_thermal_electrons'
    STOCHASTIC_FIELD = True

    @property
    def field_checklist(self):
        return {'mean' : None,
                'std' : None}
```

(continues on next page)

(continued from previous page)

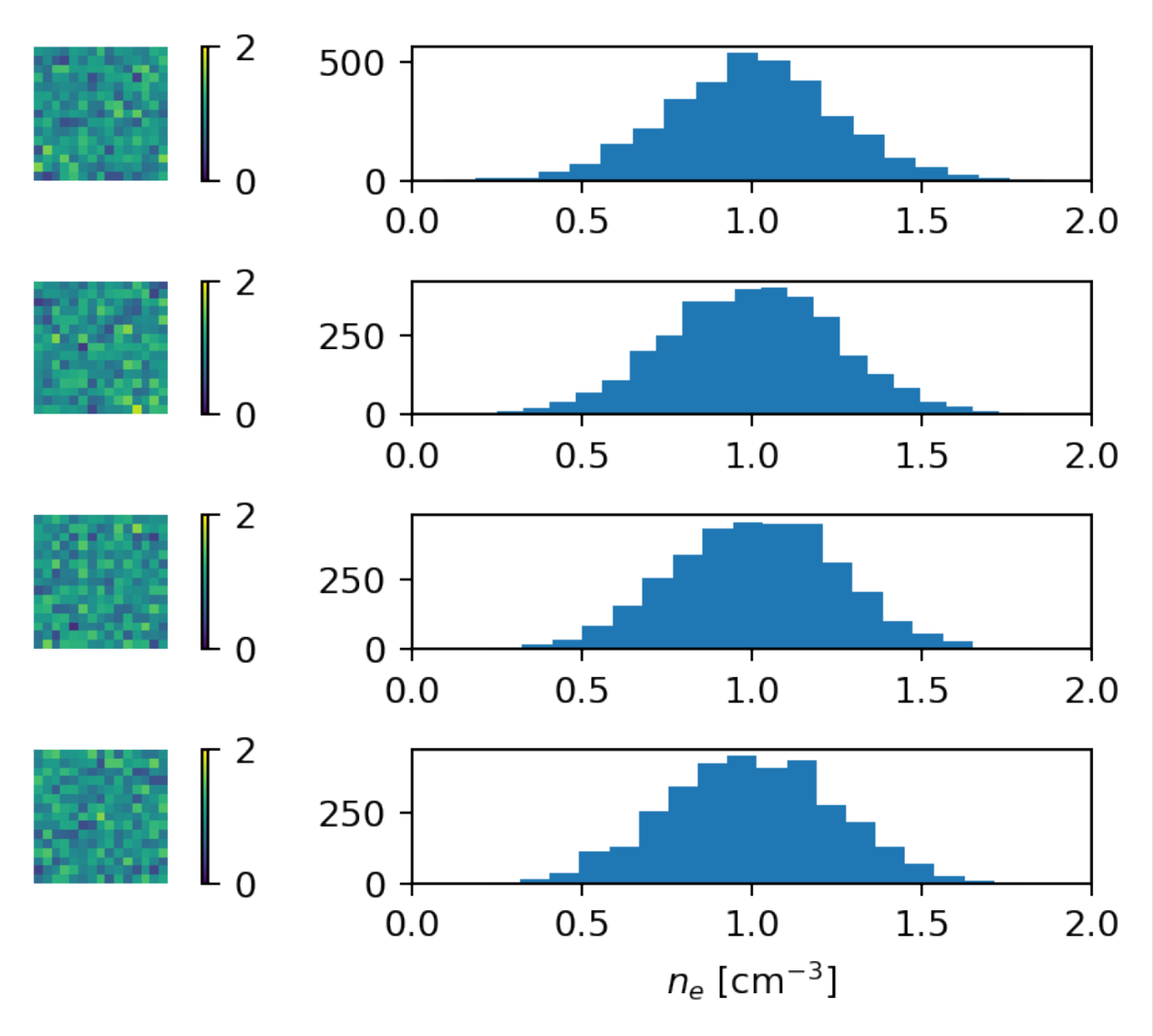
```
    def compute_field(self, seed):
        # Converts dimensional parameters into numerical values
        # in the correct units
        mu = self.parameters['mean'].to_value(self.units)
        sigma = self.parameters['std'].to_value(self.units)
        # Draws values from a normal distribution with these parameters
        # using the seed provided in the argument
        distr = stats.norm(loc=mu, scale=sigma)
        result = distr.rvs(size=self.data_shape, random_state=seed)

        return result*self.units # Restores units
```

In the example above, the field at each point of the grid is drawn from a Gaussian distribution described by the parameters 'mean' and 'std', and the `seed` argument is used to initialize the random number generator.

[13]:
```
rnd_electron_distribution = RandomThermalElectrons(
    parameters={'mean': 1.0*u.cm**-3,
                'std': 0.25*u.cm**-3},
    grid=cartesian_grid, ensemble_size=4)
```

The previous code generates an ensemble with 4 realisations of the random field. In order to inspect it, let us plot, for each realisation, a slice of the thermal electron density, and a histogram of n_e. Again, we can use the `get_data` method, but this time we provide the index of each realisation.

[14]:
```
j = 0; plt.figure(dpi=170)
for i in range(4):
    rnd_e_data = rnd_electron_distribution.get_data(i_realization=i
                                                    )
    j += 1; plt.subplot(4,2,j)

    plt.imshow(rnd_e_data[0,:,:].value, vmin=0, vmax=2)
    plt.axis('off')
    plt.colorbar()
    j += 1; plt.subplot(4,2,j)
    plt.hist(rnd_e_data.value.ravel(), bins=20)
    plt.xlim(0,2)
plt.xlabel(r'$n_e\; \left[ \rm cm^{-3}\right] $');
plt.tight_layout()
```



The previous results were generated for the *randomly chosen* random seeds:

[15]:
```
rnd_electron_distribution.ensemble_seeds
```

[15]:
```
array([1935803228,  787846414,  996406378, 1201263687])
```

Alternatively, to ensure reproducibility, one can explicitly provide the seeds instead of the ensemble size.

[16]:
```
rnd_electron_distribution = RandomThermalElectrons(
        parameters={'mean': 1.0*u.cm**-3,
                    'std': 0.25*u.cm**-3},
    grid=cartesian_grid, ensemble_seeds=[11,22,33,44])

rnd_electron_distribution.ensemble_size, rnd_electron_distribution.ensemble_seeds
```

[16]:
```
(4, [11, 22, 33, 44])
```

Before moving on, there is one specialised field type which is worth mentioning: the **dummy** field.

Dummy fields are used when one wants to send (varying) parameters *directly* to the simulator, i.e. this Field object

does not evaluate anything but the pipeline is still able to *vary its parameters*.

Why would anyone want to do this? First of all, it is worth remembering that, within the Bayesian framework, the "model" is the Field *together* with the Simulator, and the latter can also be parametrised. In other words, there can be parameters which control *how to convert* a set of models for physical fields into observables.

Another possibility is that a specific Simulator (e.g. Hammurabi) already contains built-in parametrised fields which one is willing to make use of. Dummy fields allow one to vary those parameters easily.

Below a simple example of how to define and initialize a dummy field (note that neither a `get_field` method nor `stochastic_field` property are provided for a dummy field).

[17]:
```
from imagine.fields import DummyField

class exampleDummy(DummyField):
    NAME = 'example_dummy'

    @property
    def field_checklist(self):
        return {'A': None, 'B': 'foo', 'C': 'bar'}
    @property
    def simulator_controllist(self):
        return {'lookup_directory': '/dummy/example',
                'choice_of_settings': ['tutorial','field']}
```

[18]:
```
dummy = exampleDummy(parameters={'A': 42,
                                 'B': 17*u.kpc,
                                 'C': np.pi},
                     ensemble_size=4)
```

That is it. Let us inspect the data associated with this Field:

[19]:
```
for i in range(4):
    print(dummy.get_data(i))
```

```
{'A': 42, 'B': <Quantity 17. kpc>, 'C': 3.141592653589793, 'random_seed': 423734972}
{'A': 42, 'B': <Quantity 17. kpc>, 'C': 3.141592653589793, 'random_seed': 415968276}
{'A': 42, 'B': <Quantity 17. kpc>, 'C': 3.141592653589793, 'random_seed': 670094950}
{'A': 42, 'B': <Quantity 17. kpc>, 'C': 3.141592653589793, 'random_seed': 1914837113}
```

Thus, instead of actual data arrays, the `get_data` of a dummy field returns a copy of its parameters dictionary, supplemented by a random seed which can optionally be used by the Simulator to generate stochastic fields internally.

In the context of dummy fields, the `field_checklist` property particularly important. Its main use is to send to the Simulator *fixed settings* associated with a *particular parameter*. For example, the field_checklist property is used by the Hammurabi-compatible dummy fields to inform the Simulator class where in Hammurabi XML file the parameter value should be saved.

The extra `simulator_controllist` property (which is only present in dummy fields) plays a similar role: it is used to send settings associated with a field which *are not associated with individual model parameters* to the Simulator. A typical use is the setup of global switches which enable specific builtin field in the Simulator.

9.3 Field factory

Associated with each Field class we need to prepare a FieldFactory class, which will take care (separately) of the scaling of parameter ranges, setting default values and priors over each parameter. This is can done through the following simple templates

[20]:
```
from imagine.fields import FieldFactory
from imagine.priors import FlatPrior

class ExponentialThermalElectronsFactory(FieldFactory):
    # Class attributes
    FIELD_CLASS = ExponentialThermalElectrons
    DEFAULT_PARAMETERS = {'central_density': 1*u.cm**-3,
                          'scale_radius': 3.0*u.kpc,
                          'scale_height': 0.5*u.kpc}
    PRIORS = {'central_density': FlatPrior(interval=[0,10]*u.cm**-3),
              'scale_radius': FlatPrior(interval=[1,10]*u.kpc),
              'scale_height': FlatPrior(interval=[1e-3,5]*u.kpc)}

class ConstantMagneticFieldFactory(FieldFactory):
    # Class attributes
    FIELD_CLASS = ConstantMagneticField
    DEFAULT_PARAMETERS = {'Bx': 0.0*u.microgauss,
                          'By': 5.0*u.microgauss,
                          'Bz': 0.0*u.microgauss}

    PRIORS = {'Bx': FlatPrior(interval=[-30, 30]*u.microgauss),
              'By': FlatPrior(interval=[-30, 30]*u.microgauss),
              'Bz': FlatPrior(interval=[-10, 10]*u.microgauss)}
```

We can now create instances of any of these. The priors, defaults and also parameter ranges can be accessed using the related properties:

[21]:
```
Bfactory = ConstantMagneticFieldFactory(grid=cartesian_grid)

Bfactory.default_parameters, Bfactory.parameter_ranges, Bfactory.priors
```

[21]:
```
({'Bx': <Quantity 0. uG>, 'By': <Quantity 5. uG>, 'Bz': <Quantity 0. uG>},
 {'Bx': <Quantity [-30.,  30.] uG>,
  'By': <Quantity [-30.,  30.] uG>,
  'Bz': <Quantity [-10.,  10.] uG>},
 {'Bx': <imagine.priors.basic_priors.FlatPrior at 0x7f43e85bbe10>,
  'By': <imagine.priors.basic_priors.FlatPrior at 0x7f43e85bb7d0>,
  'Bz': <imagine.priors.basic_priors.FlatPrior at 0x7f43e8097450>})
```

These instances can return Field objects through the `generate` method.

[22]:
```
newB = Bfactory()
newB.name, newB.parameters
```

[22]:
```
('constantB',
 {'Bx': <Quantity 0. uG>, 'By': <Quantity 5. uG>, 'Bz': <Quantity 0. uG>})
```

The parameters were all set to their default values. In practice, before the factory object is used, the pipeline first sets a list of *active* parameters (inactive parameters are kept with their default values)

[23]:
```
Bfactory.active_parameters = ['Bx']
```

The `generate` method can then be called with a dictionary containing *scaled dimensionless* variables as values

[24]:
```
dimensionless_scaled_variables = {'Bx': 0.7}  # This is NOT 0.7 microgauss!
newB = Bfactory(variables=dimensionless_scaled_variables)
newB.parameters
```

[24]:
```
{'Bx': <Quantity 12. uG>, 'By': <Quantity 5. uG>, 'Bz': <Quantity 0. uG>}
```

We see that the value corresponds to the 70% of the range between -30 and $30\,\mu$G

In the previous definitions, a flat (i.e. uniform) prior was assumed for all the parameters. Setting up a personalised prior is discussed in detail in the *Priors* section below. Here we demonstrate how to setup a (truncated) Gaussian prior for the parameters `Bx` and `By`. The priors can be included creating an object `GaussianPrior` for which the mean, standard deviation and range are specified:

[25]:
```
from imagine.priors import GaussianPrior

muG = u.microgauss

class ConstantMagneticFieldFactory(FieldFactory):
    # Class attributes
    FIELD_CLASS = ConstantMagneticField
    DEFAULT_PARAMETERS = {'Bx': 0.0*muG,
                          'By': 5.0*muG,
                          'Bz': 0.0*muG}
    PRIORS = {'Bx': GaussianPrior(mu=0.0*muG, sigma=10*muG,
                                  interval=[-30, 30]*muG),
              'By': GaussianPrior(mu=5.0*muG, sigma=5*muG,
                                  interval=[-30, 30]*muG),
              'Bz': FlatPrior(interval=[-10, 10]*muG)}
```

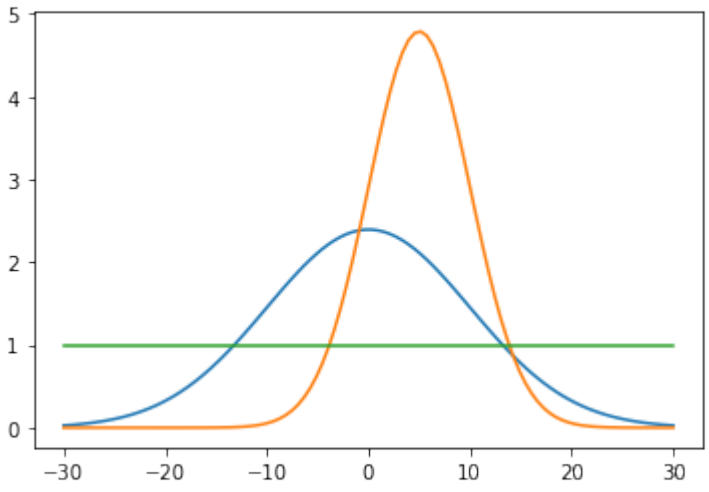
Let us now inspect an instance of updated field factory

[26]:
```
Bfactory = ConstantMagneticFieldFactory(grid=cartesian_grid)
Bfactory.priors
```

[26]:
```
{'Bx': <imagine.priors.basic_priors.GaussianPrior at 0x7f43e803ec10>,
 'By': <imagine.priors.basic_priors.GaussianPrior at 0x7f43e803ef10>,
 'Bz': <imagine.priors.basic_priors.FlatPrior at 0x7f43e80401d0>}
```

We can visualise the selected priors through auxiliary methods in the objects. E.g.

[27]:
```
b = np.linspace(-30,30,100)*muG
plt.plot(b, Bfactory.priors['Bx'].pdf_unscaled(b))
plt.plot(b, Bfactory.priors['By'].pdf_unscaled(b))
plt.plot(b, Bfactory.priors['Bz'].pdf_unscaled(b));
```



More details on how to define personalised priors can be found in the dedicated tutorial.

One final comment: the generate method can take the arguments `ensemble_seeds` or `ensemble_size` methods, propagating them to the fields it produces.

9.4 Dependencies between Fields

Sometimes, one may want to include in the inference a dependence between different fields (e.g. the cosmic ray distribution may depend on the underlying magnetic field, or the magnetic field may depend on the gas distribution). The IMAGINE *can* handle this (to a certain extent). In this section, we discuss how this works.

(NB This section is somewhat more advanced and we advice to skip it if this is your first contact with the IMAGINE software.)

9.4.1 Dependence on a field type

The most common case of dependence is when a particular model, expressed as a IMAGINE Field, depends on a 'field type', but not on another specific Field object - in other words: there is a dependence of one physical field on another physical field, and not a dependence of a particular model on another model). This is the case we are showing here.

As a concrete example, let us consider a (very artificial) model where y-component of the magnetic field strength is (for whatever reason) proportional to energy equipartition value. The Field object that represents such a field will, therefore, depend on the density distribution (which is is computed before, independently). The following snippet show how to code this.

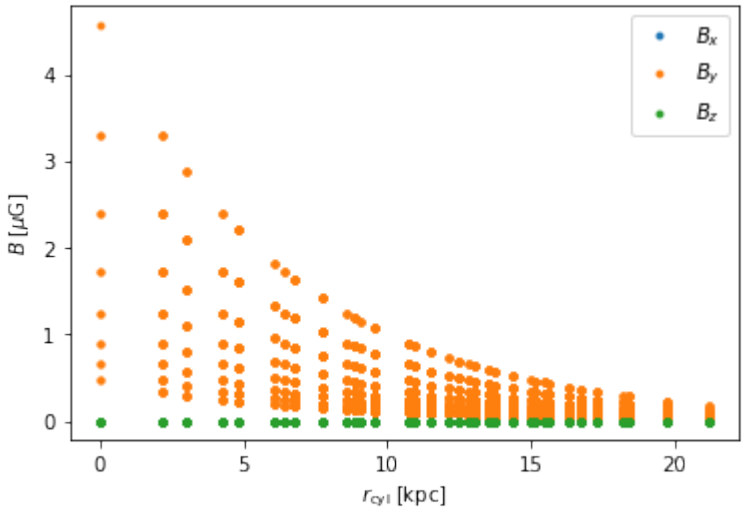
[28]:
```
import astropy.constants as c
# unfortunatelly, astropy units does not perform the following automatically
gauss_conversion = u.gauss/(np.sqrt(1*u.g/u.cm**3)*u.cm/u.s)

class DependentBFieldExample(MagneticField):
    """Example: By depends on ne"""
    # Class attributes
    NAME = 'By_Beq'
    DEPENDENCIES_LIST = ['thermal_electron_density']

    @property
    def field_checklist(self):
        return { 'v0' : None }

    def compute_field(self, seed):
        # Gets the thermal electron number density from another Field
        te_density = self.dependencies['thermal_electron_density']

        # Computes the density, assuming electrons come from H atoms
        rho = te_density * c.m_p
        Beq = np.sqrt(4*np.pi*rho)*self.parameters['v0'] * gauss_conversion

        # Sets B
        B = np.zeros(self.data_shape) * u.microgauss
        B[:,:,:,1] = Beq

        return B
```

If there is a field type string in `dependencies_list`, the pipeline will, at run-time, automatically feed a dictionary in the attribute `DependentBFieldExample.dependencies` with the request. Thus, during a run of

the Pipeline or a Simulator, the variable `te_density` will contain the *sum of all* 'thermal_electron_density' Field objects that had been supplied.

If we are willing to test the `DependentBFieldExample` above defined, we need to provide this ourselves. The following lines illustrate this.

[29]:
```
# Creates an instance
dependent_field = DependentBFieldExample(cartesian_grid,
                                         parameters={'v0': 10*u.km/u.s})

dependent_field_data = dependent_field.get_data(i_realization=0,
    dependencies={'thermal_electron_density': electron_distribution.get_data(i_
↪realization=0)})

for i, name in enumerate(['x','y','z']):
    plt.plot(dependent_field.grid.r_cylindrical.ravel(),
             dependent_field_data[...,i].ravel(), '.',
             label='$B_{}$'.format(name))
plt.xlabel(r'$r_{\rm cyl}\;[\rm kpc]$'); plt.ylabel(r'$B\;[\mu\rm G]$')
plt.legend();
```



Thus, we see that B_y decays exponentially, tracking n_e, and $B_x = B_y = 0$, as expected.

Note that, since this is a deterministic field, the `i_realization` argument may be suppressed. However, in the stochastic case, the realization index of the field and its dependencies must be aligned. (Again, this is only relevant for testing. When the Field is supplied to a Simulator, all this book-keeping is handled automatically).

9.4.2 Dependence on a field class

The second case is when a specific Field object depends on another Field object.

There are many situations where this may be needed, perhaps the most common two are:

- we have two or more Fields which share some parameters;
- we would like to write a wrapper Field classes for some pre-existing code that computes two or more physical fields at the same time.

For the latter case, the behaviour we would like to have is the following: when the first Field is invoked, the results must to be temporarily saved and later accessd by the others.

The following code illustrated the syntax to achieve this.

[30]:
```
class ConstantElectrons(ThermalElectronDensityField):
    NAME = 'constant_thermal_electron_density'

    @property
    def field_checklist(self):
        return {'A' : None}

    def compute_field(self, seed):
        #
        ne = np.ones(self.data_shape) << u.cm**-3
        ne *= self.parameters['A']
        # Suppose together with the previous calculation
        # we had computed a component of B, we can save
        # this information as an attribute
        self.Bz_should_be = 17*u.microgauss
        return  ne


class ConstantDependentB(MagneticField):
    """Example: constant magnetic field dependent on ConstantElectrons"""
    NAME = 'constantBdep'
    DEPENDENCIES_LIST = [ConstantElectrons]

    # NB field_checklist has been omitted since there are no parameters
    @property
    def field_checklist(self):
        return({})

    def compute_field(self, seed):

        # Gets the instance of the requested ConstantElectrons class
        ConstantElectrons_object = self.dependencies[ConstantElectrons]

        # Reads the common parameter A
        A = ConstantElectrons_object.parameters['A']

        # Initializes the B-field
        B = np.ones(self.data_shape) * u.microgauss
        # Sets Bx and By using A
        B[:,:,:,:2] *= np.sqrt(A)

        # Uses a Bz tha was computed earlier, elsewhere!
        B[:,:,:,2] = ConstantElectrons_object.Bz_should_be

        return B
```

If a 'class' is provided in the dependency list, the simulator will populate the `dependencies` attribute with the key-value pair: `(FieldClass: FieldObject)`, where the `FieldObject` had been evaluated earlier.

If we are willing to test, we can feed the dependencies ourselves in the following way.

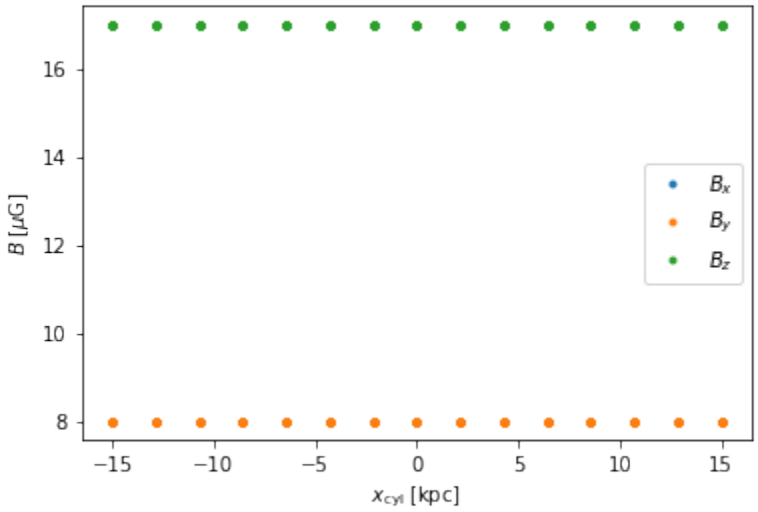
[31]:
```
# Initializes the instances
ne_field = ConstantElectrons(cartesian_grid, parameters={'A': 64})
dependent_field = ConstantDependentB(cartesian_grid)

# Evaluates the ne_field
```

(continues on next page)

(continued from previous page)

```
ne_field_data = ne_field.get_data()
# Evaluates the dependent B_field
dependent_field_data = dependent_field.get_data(
        dependencies={ConstantElectrons: ne_field})
```

[32]:
```
for i, name in enumerate(['x','y','z']):
    plt.plot(dependent_field.grid.x.ravel(),
             dependent_field_data[...,i].ravel(), '.',
             label='$B_{}$'.format(name))
plt.xlabel(r'$x_{\rm cyl}\;[\rm kpc]$'); plt.ylabel(r'$B\;[\mu\rm G]$')
plt.legend();
```



CHAPTER 10

Designing and using Simulators

Simulator objects are responsible for converting into Observables the physical quantities computed/stored by the Field objects.

Here we exemplify how to construct a Simulator for the case of computing the Faraday rotation measures on due an extended intervening galaxy with many background radio sources. For simplicity, the simulator assumes that the observed galaxy is either fully 'face-on' or 'edge-on'.

[1]:
```
import numpy as np
import astropy.units as u
from imagine.simulators import Simulator

class ExtragalacticBacklitFaradaySimulator(Simulator):
    """
    Example simulator to illustrate
    """

    # Class attributes
    SIMULATED_QUANTITIES = ['testRM']
    REQUIRED_FIELD_TYPES = ['magnetic_field', 'thermal_electron_density']
    ALLOWED_GRID_TYPES = ['cartesian', 'NonUniformCartesian']

    def __init__(self, measurements, galaxy_distance, galaxy_latitude,
                 galaxy_longitude, orientation='edge-on',
                 beam_size=2*u.kpc):
        # Send the Measurements to the parent class
        super().__init__(measurements)
        # Stores class-specific attributes
        self.galaxy_distance = galaxy_distance
        self.galaxy_lat = u.Quantity(galaxy_latitude, u.deg)
        self.galaxy_lon = u.Quantity(galaxy_longitude, u.deg)
        self.orientation = orientation
        self.beam = beam_size

    def simulate(self, key, coords_dict, realization_id, output_units):
```

(continues on next page)

(continued from previous page)

```
        # Accesses fields and grid
        B = self.fields['magnetic_field']
        ne = self.fields['thermal_electron_density']
        grid = self.grid
        # Note: the contents of self.fields correspond the present (single)
        # realization, the realization_id variable is available if extra
        # control is needed

        if self.orientation == 'edge-on':
            integration_axis = 0
            Bpara = B[:,:,:,0] # i.e. Bpara = Bx
            depths = grid.x[:,0,0]
        elif self.orientation == 'face-on':
            integration_axis = 2
            Bpara = B[:,:,:,2] # i.e. Bpara = Bz
            depths = grid.z[0,0,:]
        else:
            raise ValueError('Orientation must be either face-on or edge-on')

        # Computes dl in parsecs
        ddepth = (np.abs(depths[1]-depths[0])).to(u.pc)

        # Convert the coordinates from angles to
        # positions on one face of the grid
        lat, lon = coords_dict['lat'], coords_dict['lon']

        # Creates the outputarray
        results = np.empty(lat.size)*u.rad/u.m**2

        # Computes RMs for the entire box
        RM_array = 0.812*u.rad/u.m**2 *((ne/(u.cm**-3)) *
                                        (Bpara/(u.microgauss)) *
                                        ddepth/u.pc).sum(axis=integration_axis)
        # NB in an *production* version this would be computed only
        #    for the relevant coordinates/sightlines instead of aroungthe
        #    the whole grid, to save memory and CPU time

        # Prepares the results
        if self.orientation=='edge-on':
            # Gets y and z for a slice of the grid
            face_y = grid.y[0,:,:]
            face_z = grid.z[0,:,:]
            # Converts the tabulated galactic coords into y and z
            y_targets = (lat-self.galaxy_lat)*self.galaxy_distance
            z_targets = (lon-self.galaxy_lon)*self.galaxy_distance
            # Adjusts and removes units
            y_targets = y_targets.to(u.kpc, u.dimensionless_angles())
            z_targets = z_targets.to(u.kpc, u.dimensionless_angles())
            # Selects the relevant values from the RM array
            # (averaging neighbouring pixes within the same "beam")
            for i, (y, z) in enumerate(zip(y_targets, z_targets)):
                mask = (face_y-y)**2+(face_z-z)**2 < (self.beam)**2
                beam = RM_array[mask]
                results[i]=np.mean(beam)
        elif self.orientation=='face-on':
            # Gets x and y for a slice of the grid
            face_x = grid.x[:,:,0]
```

(continues on next page)

(continued from previous page)

```
            face_y = grid.y[:,:,0]
            # Converts the tabulated galactic coords into x and y
            x_targets = (lat-self.galaxy_lat)*self.galaxy_distance
            y_targets = (lon-self.galaxy_lon)*self.galaxy_distance
            # Adjusts and removes units
            x_targets = x_targets.to(u.kpc, u.dimensionless_angles())
            y_targets = y_targets.to(u.kpc, u.dimensionless_angles())
            # Selects the relevant values from the RM array
            # (averaging neighbouring pixes within the same "beam"
            for i, (x, y) in enumerate(zip(x_targets, y_targets)):
                mask = (face_x-x)**2+(face_y-y)**2 < (self.beam)**2
                beam = RM_array[mask]
                results[i]=np.mean(beam)
        return results
```

Thus, when designing a Simulator, one basically overrides the `simulate()` method, substituting it by some calculation which maps the various fields to some observable quantity. The available fields can be accessed through the attribute self.fields, which is a dictionary containing the field types as keys. The details of the observable can be found through the keyword arguments: key (which is the key of Measurements dictionary), coords_dict (available for tabular datasets only) and output_units (note that the value returned does not need to be exactly in the output_units, but must be convertible to them).

To see this working, let us create some fake sky coordinates over a rectangle around a galaxy that is located at galactic coordinates $(b,\, l) = (30^{\rm o},\, 30^{\rm o})$

[2]:
```
fake_sky_position_x, fake_sky_position_y = np.meshgrid(np.linspace(-4,4,70)*u.kpc,
                                                       np.linspace(-4,4,70)*u.kpc)
gal_lat = 30*u.deg; gal_lon = 30*u.deg
fake_lat = gal_lat+np.arctan2(fake_sky_position_x,1*u.Mpc)
fake_lon =  gal_lon+np.arctan2(fake_sky_position_y,1*u.Mpc)

fake_data = {'RM': np.random.random_sample(fake_lat.size),
             'err': np.random.random_sample(fake_lat.size),
             'lat': fake_lat.ravel(),
             'lon': fake_lon.ravel()}
```

From this one can construct the dataset and append it to the `Measurements` object

[3]:
```
from imagine.observables import Covariances, Measurements, TabularDataset
fake_dset = TabularDataset(fake_data, name='testRM', units= u.rad/u.m/u.m,
                           data_column='RM', error_column='err',
                           lat_column='lat', lon_column='lon')
# Initializes Measurements and Covariances objects
mea = Measurements()
cov = Covariances()
# Appends the fake tabular data
mea.append(dataset=fake_dset)
cov.append(dataset=fake_dset)
mea.keys()
```

[3]:
```
dict_keys([('testRM', 'nan', 'tab', 'nan')])
```

The measurements object will provide enough information to setup/instantiate the simulator

[4]:
```
edgeon_RMsimulator = ExtragalacticBacklitFaradaySimulator(mea, galaxy_distance=1*u.
↪Mpc,
                                          galaxy_latitude=gal_lat,
```

(continues on next page)

(continued from previous page)

```
                                                galaxy_longitude=gal_lon,
                                                beam_size=0.700*u.kpc,
                                                orientation='edge-on')
faceon_RMsimulator = ExtragalacticBacklitFaradaySimulator(mea, galaxy_distance=1*u.
↪Mpc,
                                                galaxy_latitude=gal_lat,
                                                galaxy_longitude=gal_lon,
                                                beam_size=0.7*u.kpc,
                                                orientation='face-on')
```

To test it, we will generate a dense grid and evaluate a magnetic field and electron density on top of it

[5]:
```
from imagine.fields import ConstantMagneticField, ExponentialThermalElectrons,␣
↪UniformGrid

dense_grid = UniformGrid(box=[[-15, 15]*u.kpc,
                              [-15, 15]*u.kpc,
                              [-15, 15]*u.kpc],
                         resolution = [30,30,30])
B = ConstantMagneticField(grid=dense_grid, ensemble_size=1,
                          parameters={'Bx': 0.5*u.microgauss,
                                      'By': 0.5*u.microgauss,
                                      'Bz': 0.5*u.microgauss})
ne_disk = ExponentialThermalElectrons(grid=dense_grid, ensemble_size=1,
                                      parameters={'central_density': 0.5*u.cm**-3,
                                                  'scale_radius': 3.3*u.kpc,
                                                  'scale_height': 0.5*u.kpc})
```

Now we can call the simulator, which returns a `Simulation` object

[6]:
```
edgeon_sim = edgeon_RMsimulator([B,ne_disk])
faceon_sim = faceon_RMsimulator([B,ne_disk])
print('faceon_sim:',faceon_sim)
print('faceon_sim keys:',list(faceon_sim.keys()))
```

```
faceon_sim: <imagine.observables.observable_dict.Simulations object at 0x7efdb0e39b50>
faceon_sim keys: [('testRM', 'nan', 'tab', 'nan')]
```

[7]:
```
faceon_sim[('testRM', 'nan', 'tab', 'nan')].data.shape
```

[7]:
```
(1, 4900)
```

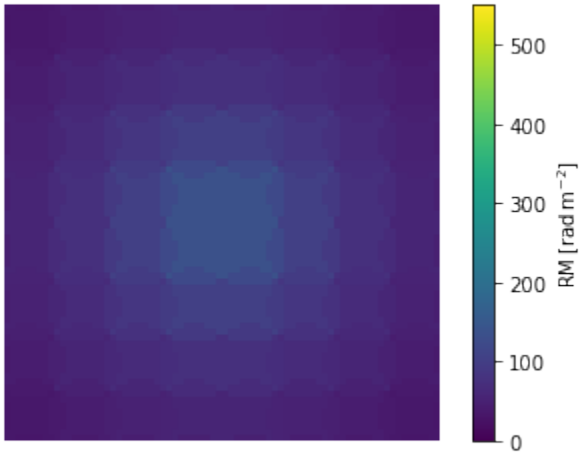
Using the fact that the original coordinates correspondended to a rectangle in the sky, we can visualize the results

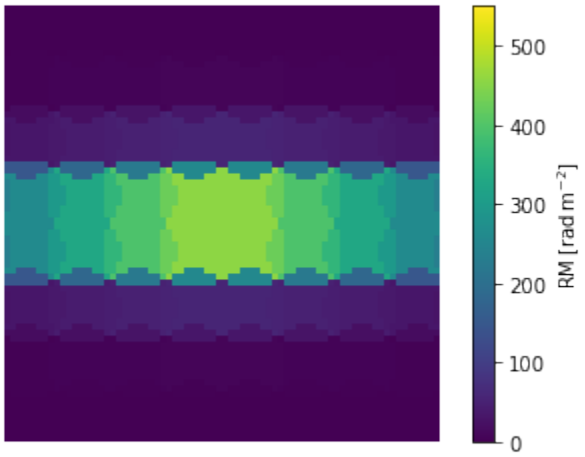
[8]:
```
import matplotlib.pyplot as plt

i = 0
key = tuple(faceon_sim.keys())[0]
d = faceon_sim[key].data[i]
# Using the fact that the coordinates correspond to a rectangle
im = d.reshape(int(np.sqrt(d.size)),int(np.sqrt(d.size)))
plt.imshow(im, vmin=0, vmax=550); plt.axis('off')
plt.colorbar(label=r'$\rm RM\,[ rad\,m^{-2}]$')

plt.figure()
key = tuple(edgeon_sim.keys())[0]
d = edgeon_sim[key].data[i]
```

(continues on next page)

(continued from previous page)

```
# Using the fact that the coordinates correspond to a rectangle
im = d.reshape(int(np.sqrt(d.size)),int(np.sqrt(d.size)))
plt.imshow(im, vmin=0, vmax=550); plt.axis('off')
plt.colorbar(label=r'$\rm RM\,[ rad\,m^{-2}]$');
```





Simulators are able to handle *multiple fields of the same type* by summing up their data. This is particularly convenient if a physical quantity is described by a both a deterministic part and random fluctuations.

We will illustrate this with another artificial example, reusing the `RandomThermalElectrons` field discussed before. We will also illustrate the usage of ensembles (note: for non-stochastic fields the ensemble size has to be kept the same to ensure consistency, but internally they will be evaluated only once).

[9]:
```
from imagine.fields import RandomThermalElectrons

dense_grid = UniformGrid(box=[[-15, 15]*u.kpc,
                              [-15, 15]*u.kpc,
                              [-15, 15]*u.kpc],
                         resolution = [30,30,30])
B = ConstantMagneticField(grid=dense_grid, ensemble_size=3,
                          parameters={'Bx': 0.5*u.microgauss,
                                      'By': 0.5*u.microgauss,
                                      'Bz': 0.5*u.microgauss})
ne_disk = ExponentialThermalElectrons(grid=dense_grid, ensemble_size=3,
```

(continues on next page)

(continued from previous page)

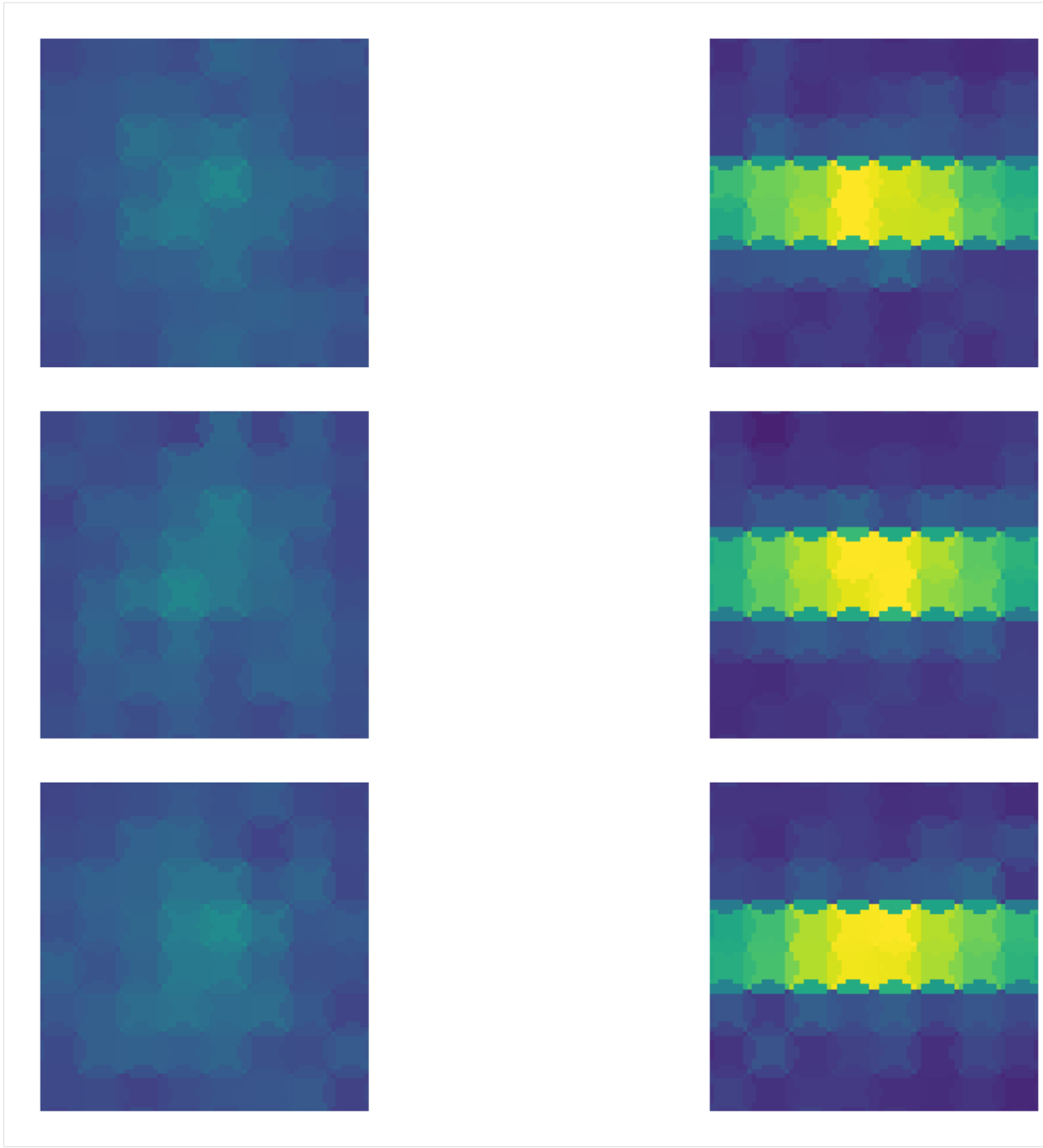
```
                                    parameters={'central_density': 0.5*u.cm**-3,
                                                'scale_radius': 3.3*u.kpc,
                                                'scale_height': 0.5*u.kpc})
ne_rnd = RandomThermalElectrons(grid=dense_grid, ensemble_size=3,
                                parameters={'mean': 0.005*u.cm**-3,
                                            'std': 0.01*u.cm**-3,
                                            'min_ne' : 0*u.cm**-3})
```

The extra field can be included in the simulator using

[10]:
```
edgeon_sim = edgeon_RMsimulator([B,ne_disk,ne_rnd])
faceon_sim = faceon_RMsimulator([B,ne_disk,ne_rnd])
```

Let us now plot, as before, the simulated observables for each realisation

[11]:
```
fig, axs = plt.subplots(3, 2, sharex=True, sharey=True, dpi=200)

for i, ax in enumerate(axs):
    key = tuple(faceon_sim.keys())[0]
    d = faceon_sim[key].data[i]
    ax[0].imshow(d.reshape(int(np.sqrt(d.size)),int(np.sqrt(d.size))),
           vmin=0, vmax=550)

    key = tuple(edgeon_sim.keys())[0]
    d = edgeon_sim[key].data[i]
    ax[1].imshow(d.reshape(int(np.sqrt(d.size)),int(np.sqrt(d.size))),
                 vmin=0, vmax=550)
    ax[1].axis('off'); ax[0].axis('off')
plt.tight_layout()
```



CHAPTER 11

The Hammurabi simulator

This tutorial shows how to use the Hammurabi simulator class the interface to hammurabiX code.

Throughout the tutorial, we will use the term 'Hammurabi' to refer to the Simulator class, and 'hammurabiX' to refer to the hammurabiX software.

[1]:
```
import matplotlib
%matplotlib inline

import numpy as np
import healpy as hp
import matplotlib.pyplot as plt

import imagine as img

import imagine.observables as img_obs
import astropy.units as u
```

11.1 Initializing

In the normal IMAGINE workflow, the simulator produces a set of mock observables (Simulators in IMAGINE jargon) which one wants to compare with a set of observational data (i.e. Measurements). Thus, the Hammurabi simulator class (as any IMAGINE simulator) has to be initialized with a Measurements object in order to know properties of the output it will need to generate.

Therefore, we begin by creating fake, empty, datasets which will help instructing Hammurabi which observational data we are interested in.

[2]:
```
from imagine.observables import Measurements

# Creates some empty fake datasets
size = 12*32**2
sync_dset = img_obs.SynchrotronHEALPixDataset(data=np.empty(size)*u.mK,
```

(continues on next page)

(continued from previous page)

```
                                                frequency=23*u.GHz, type='I')
size = 12*16**2
fd_dset = img_obs.FaradayDepthHEALPixDataset(data=np.empty(size)*u.rad/u.m**2)
size = 12*8**2
dm_dset = img_obs.DispersionMeasureHEALPixDataset(data=np.empty(size)*u.pc/u.cm**3)

# Appends them to an Observables Dictionary
fakeMeasureDict = Measurements()
fakeMeasureDict.append(dataset=sync_dset)
fakeMeasureDict.append(dataset=fd_dset)
fakeMeasureDict.append(dataset=dm_dset)
```

Now it is possible initializing the simulator. The Hammurabi simulator prints its setup after initialization, showing that we have defined three observables.

[3]:
```
from imagine.simulators import Hammurabi
simer = Hammurabi(measurements=fakeMeasureDict)
```

```
observable {}
|-->  sync {'cue': '1', 'freq': '23', 'nside': '32'}
|-->  faraday {'cue': '1', 'nside': '16'}
|-->  dm {'cue': '1', 'nside': '8'}
```

11.2 Running with dummy fields

11.2.1 Using only non-stochastic fields

In order to run an IMAGINE simulator, we need to specify a list of Field objects it will map onto observables.

The original hammurabiX code is *not only* a Simulator in IMAGINE's sense, but comes also with a large set of built-in fields. Using dummy IMAGINE fields, it is possible to instruct Hammurabi to run using one of hammurabiX's built-in fields instead of evaluating an IMAGINE Field.

A range of such dummy Fields and the associated Field Factories can be found in the subpackage imagine.fields.hamx.

Using some of these, let us initialize three dummy fields: one instructing Hammurabi to use one of hammurabiX's regular fields (`BregLSA`), one setting CR electron distribution (`CREAna`), and one setting the thermal electron distribution (`YMW16`).

[4]:
```
from imagine.fields.hamx import BregLSA, CREAna, TEregYMW16

## ensemble size
ensemble_size = 2

## Set up the BregLSA field with the parameters you want:
paramlist = {'b0': 6.0, 'psi0': 27.9, 'psi1': 1.3, 'chi0': 24.6}
breg_wmap = BregLSA(parameters=paramlist, ensemble_size=ensemble_size)

## Set up the analytic CR model CREAna
paramlist_cre = {'alpha': 3.0, 'beta': 0.0, 'theta': 0.0,
                 'r0': 5.6, 'z0': 1.2,
                 'E0': 20.5,
                 'j0': 0.03}
cre_ana = CREAna(parameters=paramlist_cre, ensemble_size=ensemble_size)
```

(continues on next page)

(continued from previous page)

```
##  The free electron model based on YMW16, ie. TEregYMW16
fereg_ymw16 = TEregYMW16(parameters={}, ensemble_size=ensemble_size)
```

Now we can run the Hammurabi to generate one set of observables

[5]:
```
maps = simer([breg_wmap, cre_ana, fereg_ymw16])
```

The `Hammurabi` class wraps around hammurabiX's own python wrapper `hampyx`. The latter can be accessed through the attribue `_ham`.

It is generally convenient not using `hampyx` directly, considering future updates in hammurabiX. Nevertheless, there situations where this is still convenient, particularly while troubleshooting.

The direct access to `hampyx` is exemplified below, where we check its initialization, after running the simulation object.

[6]:
```
simer._ham.print_par(['magneticfield', 'regular'])
simer._ham.print_par(['magneticfield', 'regular', 'wmap'])
simer._ham.print_par(['cre'])
simer._ham.print_par(['cre', 'analytic'])
simer._ham.print_par(['thermalelectron', 'regular'])
```

```
regular {'cue': '1', 'type': 'lsa'}
|-->  lsa {}
|-->  jaffe {}
|-->  unif {}
cre {'cue': '1', 'type': 'analytic'}
|-->  analytic {}
|-->  unif {}
analytic {}
|-->  alpha {'value': '3.0'}
|-->  beta {'value': '0.0'}
|-->  theta {'value': '0.0'}
|-->  r0 {'value': '5.6'}
|-->  z0 {'value': '1.2'}
|-->  E0 {'value': '20.5'}
|-->  j0 {'value': '0.03'}
regular {'cue': '1', 'type': 'ymw16'}
|-->  ymw16 {}
|-->  unif {}
```

Any `Simulator` by convention returns a `Simulations` object, which collect all required maps. We want to get them back as arrays we can visualize with healpy. The `data` attribute does this, and note that what it gets back is a **list** of two of each type of observable, since we specified `ensemble_size=2` above. But since we have not yet added a random component, they are both the same:

[7]:
```
maps[('sync','23','32','I')].global_data
```

[7]:
```
array([[0.08884023, 0.08772291, 0.08634578, ..., 0.08866684, 0.08728929,
        0.08849186],
       [0.08884023, 0.08772291, 0.08634578, ..., 0.08866684, 0.08728929,
        0.08849186]])
```

Below we exemplify how to extract and plot the simulated maps produced by Hammurabi:

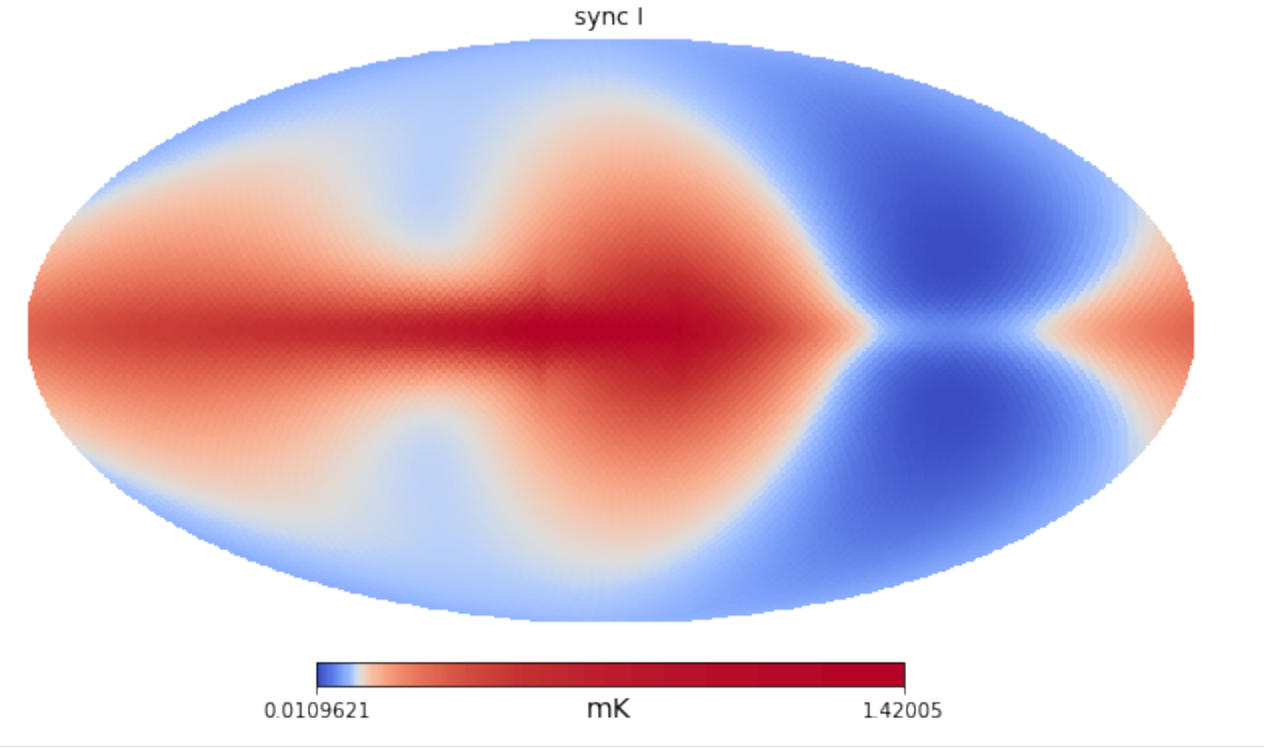
[8]:
```
for key in maps.keys():
    simulated_data = maps[key].global_data[0]
```

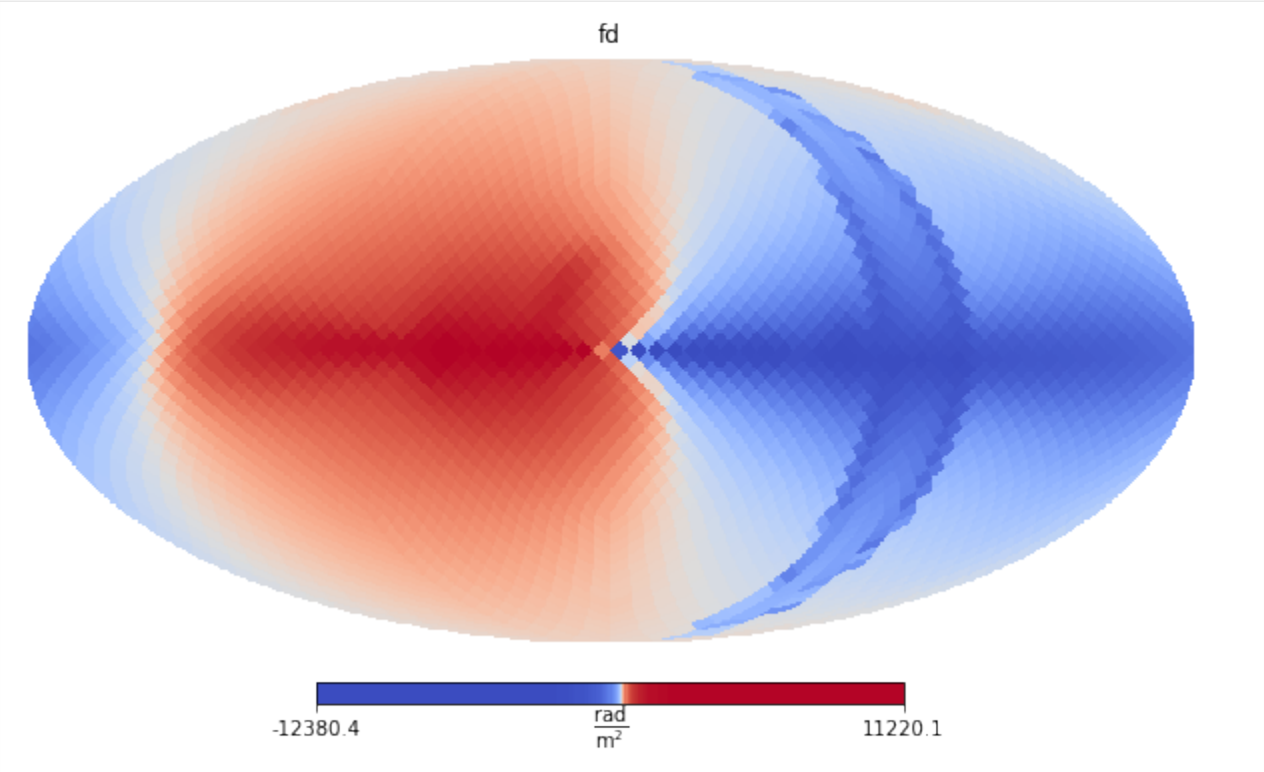
(continues on next page)

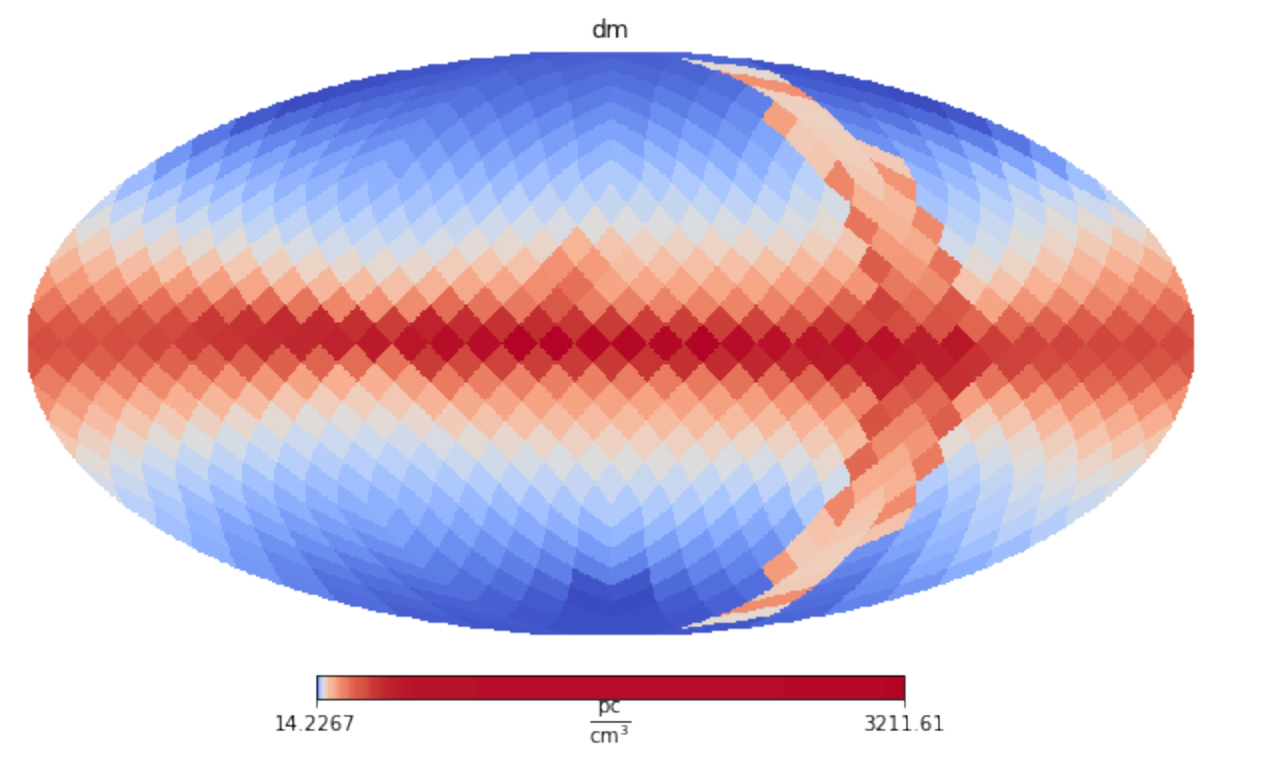
(continued from previous page)

```
    simulated_unit = maps[key].unit
    name = key[0]
    if key[3] != 'nan':
        name += ' '+key[3]
    hp.mollview(simulated_data, norm='hist', cmap='coolwarm',
                unit=simulated_unit._repr_latex_(), title=name)
```

```
/home/lfsr/.local/lib/python3.7/site-packages/healpy-1.12.8-py3.7-linux-x86_64.egg/
↪healpy/projaxes.py:989: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if matplotlib.cbook.iterable(value):
/home/lfsr/.local/lib/python3.7/site-packages/healpy-1.12.8-py3.7-linux-x86_64.egg/
↪healpy/projaxes.py:959: MatplotlibDeprecationWarning:
The iterable function was deprecated in Matplotlib 3.1 and will be removed in 3.3.␣
↪Use np.iterable instead.
  if matplotlib.cbook.iterable(value):
```




fd
-12380.4
rad
m²
11220.1


dm
14.2267
pc
cm³
3211.61

11.2.2 Using a stochastic magnetic field component

Now we add a random GMF component with the BrndES model. This model starts with a random number generator to simulate a Gaussian random field on a cartesian grid and ensures that it is divergence free. The grid is defined in hammurabiX XML parameter file.

[9]:
```
from imagine.fields.hamx import BrndES

paramlist_Brnd = {'rms': 6., 'k0': 0.5, 'a0': 1.7,
                  'k1': 0.5, 'a1': 0.0,
                  'rho': 0.5, 'r0': 8., 'z0': 1.}

brnd_es = BrndES(parameters=paramlist_Brnd, ensemble_size=ensemble_size,
                 grid_nx=100, grid_ny=100, grid_nz=40)
# The keyword arguments grid_ni modify random field grid for limiting
# the notebook's memory consumption.
```

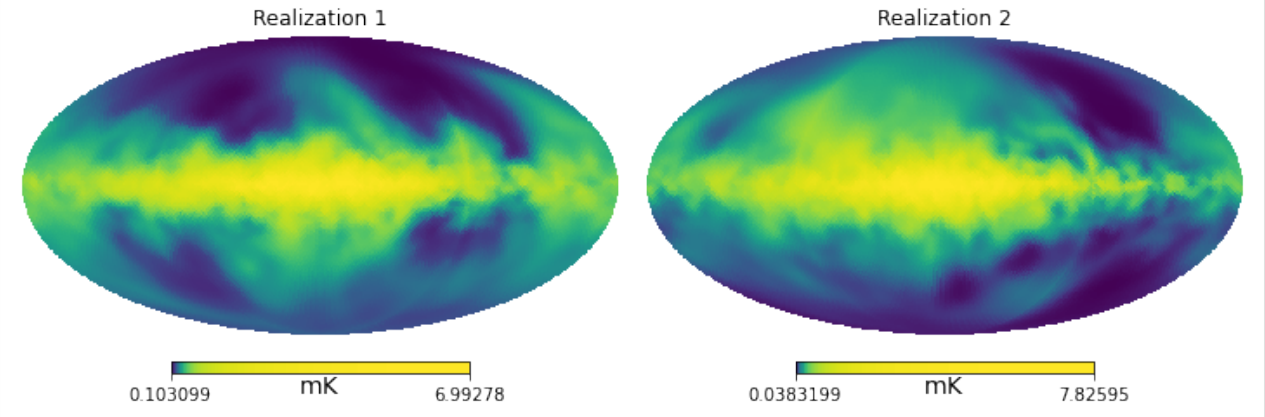
Now use the simulator to generate the maps from these field components and visualize:

[10]:
```
maps = simer([breg_wmap, brnd_es, cre_ana, fereg_ymw16])
```

[11]:
```
print(maps.keys())
```

```
dict_keys([('sync', '23', '32', 'I'), ('fd', 'nan', '16', 'nan'), ('dm', 'nan', '8',
↪'nan')])
```

[12]:
```
sync_I_data = maps[('sync','23','32','I')].global_data
sync_I_units = maps[('sync','23','32','I')].unit

matplotlib.rcParams['figure.figsize'] = (10.0, 4.0)
for i in range(1,3):
    hp.mollview(sync_I_data[i-1], norm='hist',sub=(1,2,i),
                unit=sync_I_units._repr_latex_(), title='Realization {}'.format(i))
```



11.3 Running with IMAGINE fields

11.3.1 The basics

While hammurabiX's fields are extremely useful, we want the flexibility of quickly plugging *any* IMAGINE Field to hammurabiX. Fortunately, this is actually very easy. For the sake of simplificty, let us initialize a "fresh" simulator

object.

[13]:
```
simer = Hammurabi(measurements=fakeMeasureDict)
```

```
observable {}
|-->  sync {'cue': '1', 'freq': '23', 'nside': '32'}
|-->  faraday {'cue': '1', 'nside': '16'}
|-->  dm {'cue': '1', 'nside': '8'}
```

Now, let us initialize a few simple Fields and Grid

[14]:
```
from imagine.fields import ConstantMagneticField, ExponentialThermalElectrons,␣
↪UniformGrid

# We initalize a common grid for all the tests, with 100^3 meshpoints
grid = UniformGrid([[-25,25]]*3*u.kpc,
                       resolution=[100]*3)

# Two magnetic fields: constant By and constant Bz across the box
By_only = ConstantMagneticField(grid,
                                parameters={'Bx': 0*u.microgauss,
                                            'By': 1*u.microgauss,
                                            'Bz': 0*u.microgauss})
Bz_only = ConstantMagneticField(grid,
                                parameters={'Bx': 0*u.microgauss,
                                            'By': 0*u.microgauss,
                                            'Bz': 1*u.microgauss})
# Constant electron density in the box
# ne = basic_fields.constantThermalElectrons(grid, ensemble_size=ensemble_size,
#                                             parameters={'ne':1*u.cm**-3})
ne = ExponentialThermalElectrons(grid, parameters={'central_density' : 0.01*u.cm**-3,
                                                   'scale_radius' : 3*u.kpc,
                                                   'scale_height' : 100*u.pc})
```

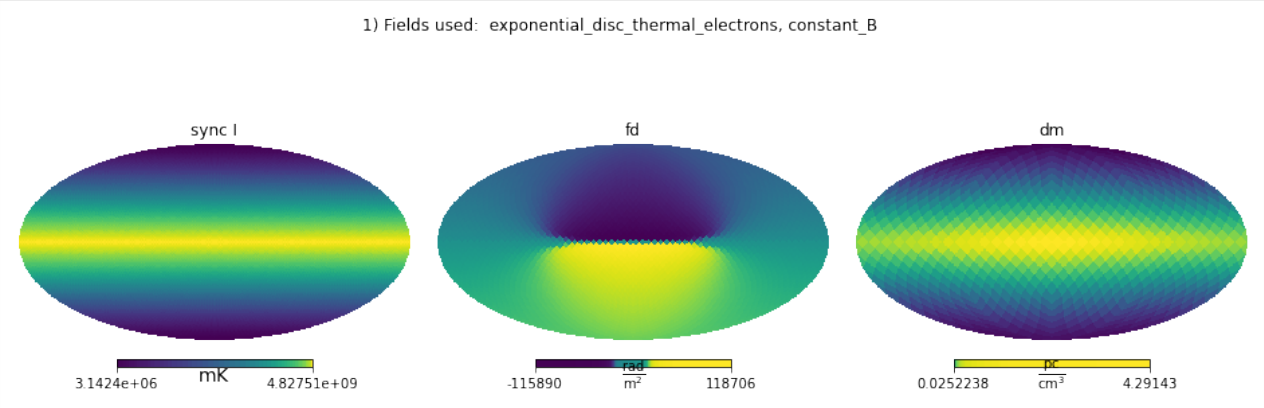
We can run hammurabi by simply provinding the fields in the fields list

[15]:
```
fields_list1 = [ne, Bz_only]
maps = simer(fields_list1)
```

We can now examine the results. First, let us type a helper to visualise.

[16]:
```
def show_maps(maps, fig):
    for i, key in enumerate(maps.keys()):
        simulated_data = maps[key].data[0]
        simulated_unit = maps[key].unit
        name = key[0]
        if key[3] != 'nan':
            name += ' '+key[3]
        hp.mollview(simulated_data, norm='hist', sub=(1,len(maps.keys()),i+1),
                    fig=fig, unit=simulated_unit._repr_latex_(), title=name)
```

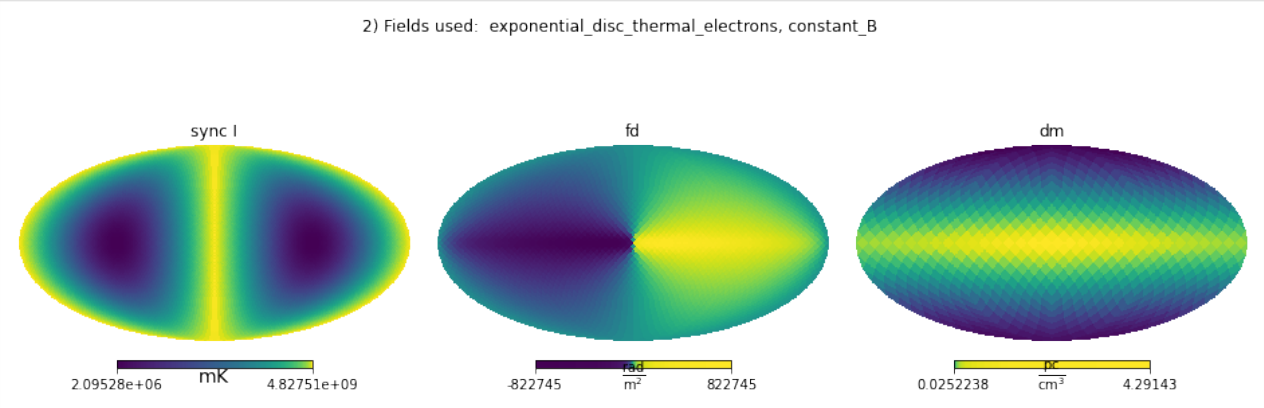
[17]:
```
# Creates a list of names
field_names = [field.name for field in fields_list1]

fig = plt.figure(figsize=(13.0, 4.0))
show_maps(maps, fig)
plt.suptitle('1) Fields used:  ' + ', '.join(field_names));
```

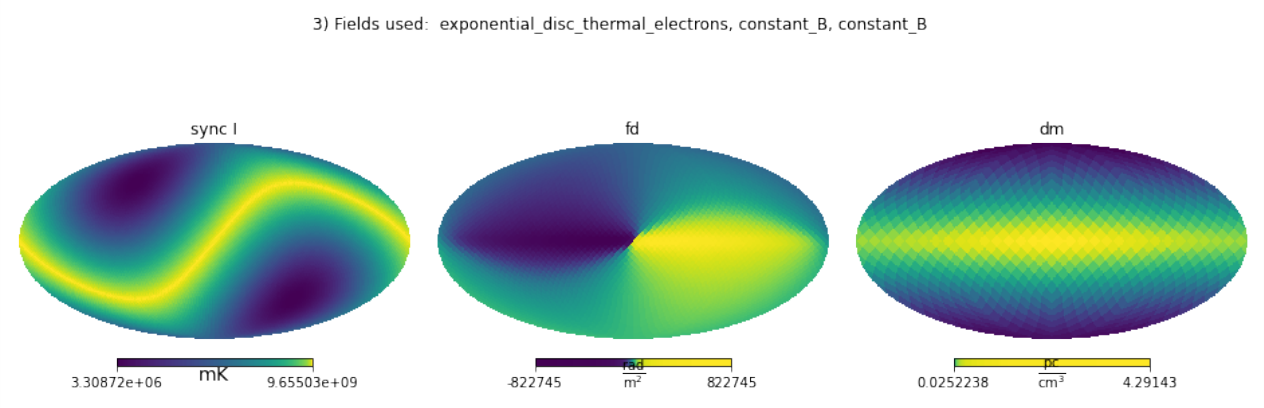


Which is what one would expect for this very artificial setup.

As usual, we can combine different fields (fields of the same type are simply summed up. The following cell illustrates this.

[18]:
```
fields_list2 = [ne, By_only]
fields_list3 = [ne, By_only, Bz_only]

for i, fields_list in enumerate([fields_list2, fields_list3]):
    # Creates a list of names
    field_names = [field.name for field in fields_list]

    fig = plt.figure(figsize=(13.0, 4.0))
    maps = simer(fields_list)
    show_maps(maps, fig)
    plt.suptitle(str(i+2)+') Fields used:  ' + ', '.join(field_names));
```





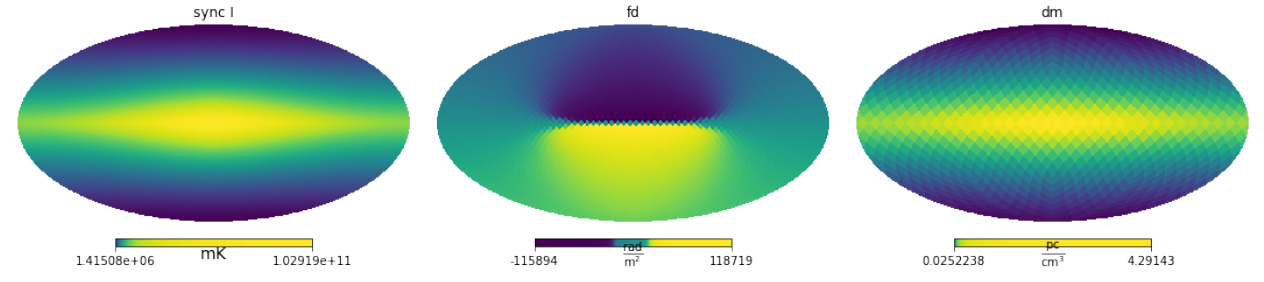
11.3.2 Mixing internal and IMAGINE fields

The dummy fields that enable hammurabiX's internal fields can be combined with normal IMAGINE Fields as long as they belong to different categories in hammurabiX's implementation. These are: * regular magnetic field * random magnetic field * regular thermal electron density * random thermal electron density * cosmic ray electrons

Thus, if you provide IMAGINE Fields of a given field type, the corresponding hammurabiX built-in field is deactivated.

In the following example we illustrate using an IMAGINE field for the "regular magnetic field", and dummies for the "random magnetic field" and cosmic rays.

[19]:
```
# Re-defines the fields, with the default ensemble size of 1
brnd_es = BrndES(parameters=paramlist_Brnd)
brnd_es.set_grid_size(nx=100, ny=100, nz=40)
cre_ana = CREAna(parameters=paramlist_cre)

fields_list = [brnd_es, ne, Bz_only, cre_ana]

field_names = [field.name for field in fields_list]
fig = plt.figure(figsize=(15.0, 4.0))
maps = simer(fields_list)
show_maps(maps, fig)
plt.suptitle('Fields used:  ' + ', '.join(field_names));
```



CHAPTER 12

Priors

A powerful aspect of a fully bayesian analysis approach is the possibility of explicitly stating any prior expectations about the parameter values based on previous knowledge.

The most typical use of the IMAGINE employs Pipeline objects based on the Nested Sampling approach (e.g. Ultranest). This requires the priors to be specified as a *prior transform function*, that is: a mapping between uniformly distributed values to the actual distribution. The IMAGINE prior classes can handle this automatically and output either the *probability density function* (PDF) or the *prior transform function*, depending on the needs of the chosen sampler.

12.1 Marginal prior distributions

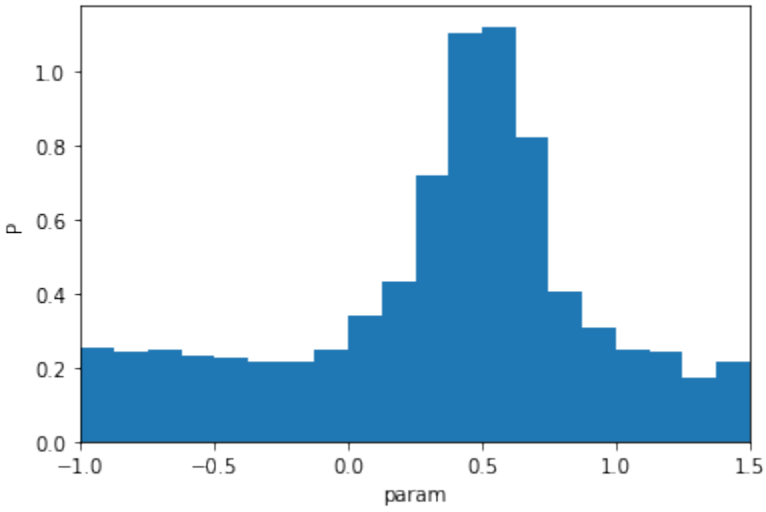
We will first approach the case where we only have access independent prior information for each parameter (i.e. there is no prior information on correlation between parameters). The `GeneralPrior` class helps constructing an IMAGINE prior from either: a know prior PDF, or a previous sampling of the parameter space.

12.1.1 Prior from a sample

To illustrate this, we will first construct a sample associated with a hypothetical parameter. To keep things simple but still illustrative, we construct this combining a uniform distribution and a normal distribution.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import astropy.units as u
import imagine as img
```

[2]:
```
sample = np.concatenate([np.random.random_sample(2000),
                         np.random.normal(loc=0.6, scale=0.07, size=1500) ])
sample = sample*2.5-1
sample *= u.microgauss
plt.hist(sample.value, bins=20, density=True)
plt.ylabel('P'); plt.xlabel('param'); plt.xlim(-1,1.5);
```



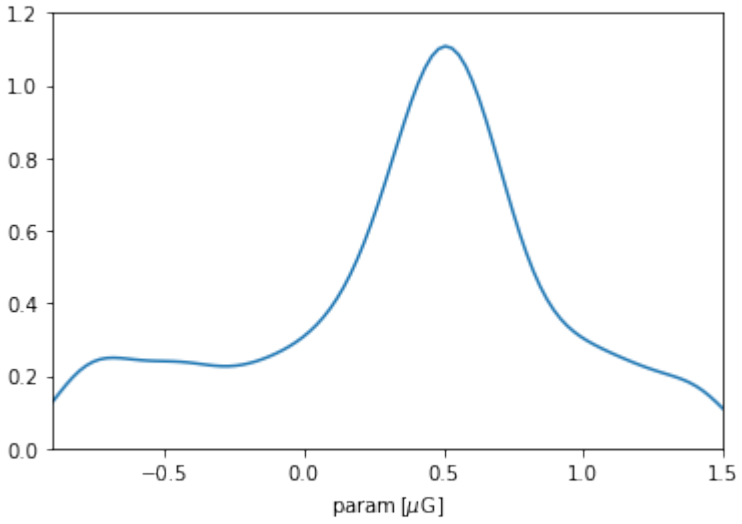
This distribution could be the result of a previous inference exercise (i.e. a previously computed marginalised posterior distribution).

From it, we can construct our prior using the `GeneralPrior` class. Lets say that, for some reason, we are only interested in the interval $[-0.9, 1.5]$ (say, for example, $p = -1$ is unphysical), this can be accounted for with the argument `interval`.

[3]:
```
interval = (-0.9,1.5)*u.microgauss

prior_param = img.priors.GeneralPrior(samples=sample, interval=interval)

del sample # to save memory
```

At this point we can inspect the PDF to see what we have.

[4]:
```
p = np.linspace(*interval, 100)
plt.plot(p, prior_param.pdf_unscaled(p))
plt.xlim(*interval.value); plt.ylim(0,1.2); plt.xlabel(r'param$\,[\mu\rm G]$');
```

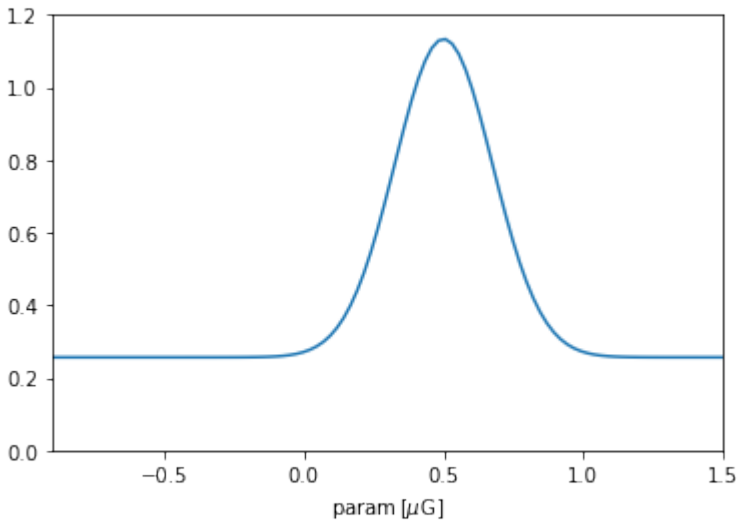


A cautionary note: the KDE used in intermediate calculation tends so smoothen the distribution and forces a slight decay close to the endpoints (reflecting the fact that a Gaussian kernel was employed). For most practical applications, this is not a big problem: one can control the degree of smoothness through the argument `bw_method` while initializing `GeneralPrior`, and the range close to endpoints are typically uninteresting. But it is recommended to always check the PDF of a prior generated from a set of samples.

12.1.2 Prior from a known PDF

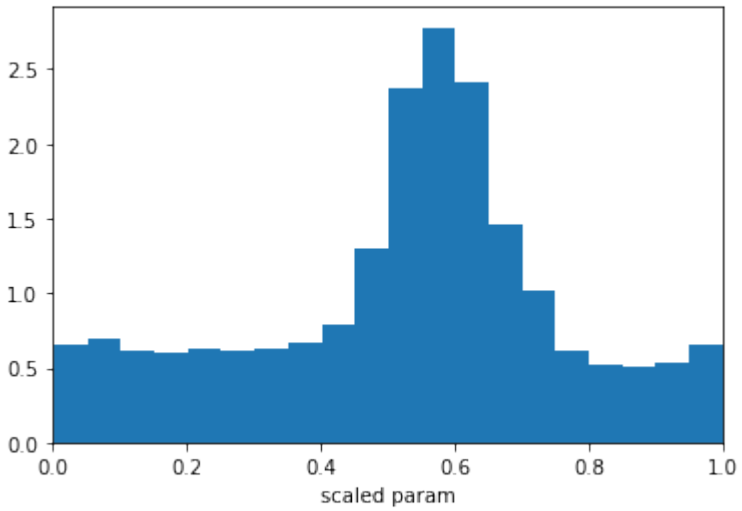
Alternatively, when one knows the analytic shape of given PDF, one can instead supply a function to `GeneralPrior`. In this case, the shape of the original function is generally respected. For example:

[5]:
```
def example_pdf(y):
    x = y.to(u.microgauss).value # Handles units
    uniform_part = 1
    sigma = 0.175; mu = 0.5
    gaussian_part = 1.5*( 1/(sigma * np.sqrt(2 * np.pi))
                         * np.exp( - (x - mu)**2 / (2 * sigma**2) ))
    return uniform_part + gaussian_part

prior_param = img.priors.GeneralPrior(pdf_fun=example_pdf, interval=interval)

plt.plot(p, prior_param.pdf_unscaled(p))
plt.xlim(*interval.value); plt.ylim(0,1.2); plt.xlabel(r'param$\,[\mu\rm G]$');
```



Once the prior object was constructed, the IMAGINE Pipeline object uses it as the mapping above described to sample new paramters. Let us illustrate this concretely and check whether the prior is working (note that the Prior will operate on scaled parameters, i.e. with values in the interval $[0, 1]$).

[6]:
```
uniform_sample = np.random.random_sample(2000)
sampled_values = prior_param(uniform_sample)

plt.hist(sampled_values, bins=20, density=True)
plt.xlabel('scaled param'); plt.xlim(0,1);
```



12.1.3 Prior from scipy.stats distribution

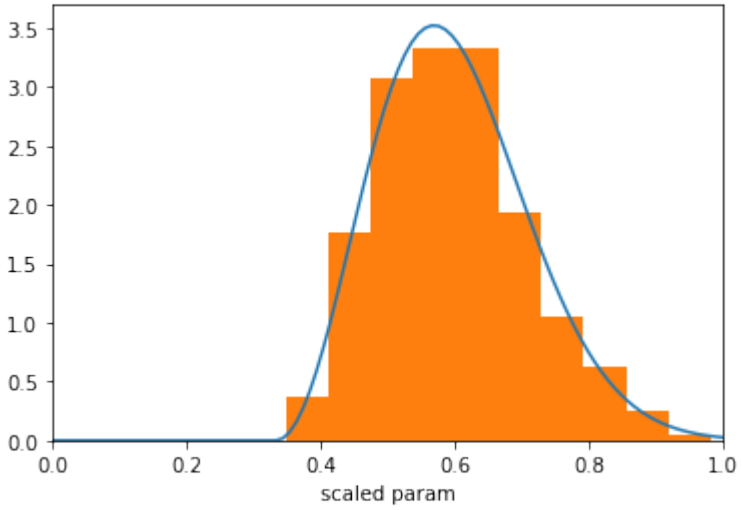
We now demonstrate a helper class which allows to easily construct priors from one scipy.stats distributions. Lets say we wanted impose a chi prior distribution for a given parameter, we can achive this using the `scipyPrior` class.

[7]:
```
import scipy.stats
muG = u.microgauss
chiPrior = img.priors.ScipyPrior(scipy.stats.chi, 3, loc=-10*muG,
                                 scale=5*muG, interval=[-20,10]*muG)
```

The first argument of `img.priors.scipyPrior` is an instance of `scipy.stats.rv_continuous`, this is followed by any args required by the scipy distribution (in this specific case, `3` is the number of degrees of freedom in the chi-distribution). The keyword arguments loc and scale have the same meaning as in the `scipy.stats` case, and interval tells maximum and minimum parameter values that will be considered.

Let us check that this works, ploting the PDF and an histogram of parameter values sampled from the prior.

[8]:
```
# Plots the PDF associated with this prior
t = np.linspace(0,1,100)
plt.plot(t, chiPrior.pdf(t))
# Plots the distribution of values constructed using this prior
x = np.random.random_sample(2000)
plt.hist(chiPrior(x), density=True);
plt.xlabel('scaled param'); plt.xlim(0,1);
```



12.2 Joint prior distributions

In the more general (and perhaps interesting) case, the priors are not expressed for individual parameters (i.e. marginalized), but instead as joint distributions. This feature is still *under development*.

CHAPTER 13

Masking HEALPix datasets

For users who do not want to simulate and fit a full sky map (e.g., to remove confusing regions) or who need patches of a HEALPix map at high resolution, IMAGINE has a Masks class derived from **ObservableDict**. It also applies the masks correctly not only to the simulation but also the measured data sets and the corresponding observational covariances.

[1]:
```
import numpy as np
import healpy as hp
import imagine as img
import astropy.units as u
import imagine.observables as img_obs
from imagine.fields.hamx import BregLSA, TEregYMW16, CREAna
```

13.1 Creating a Mask dictionary

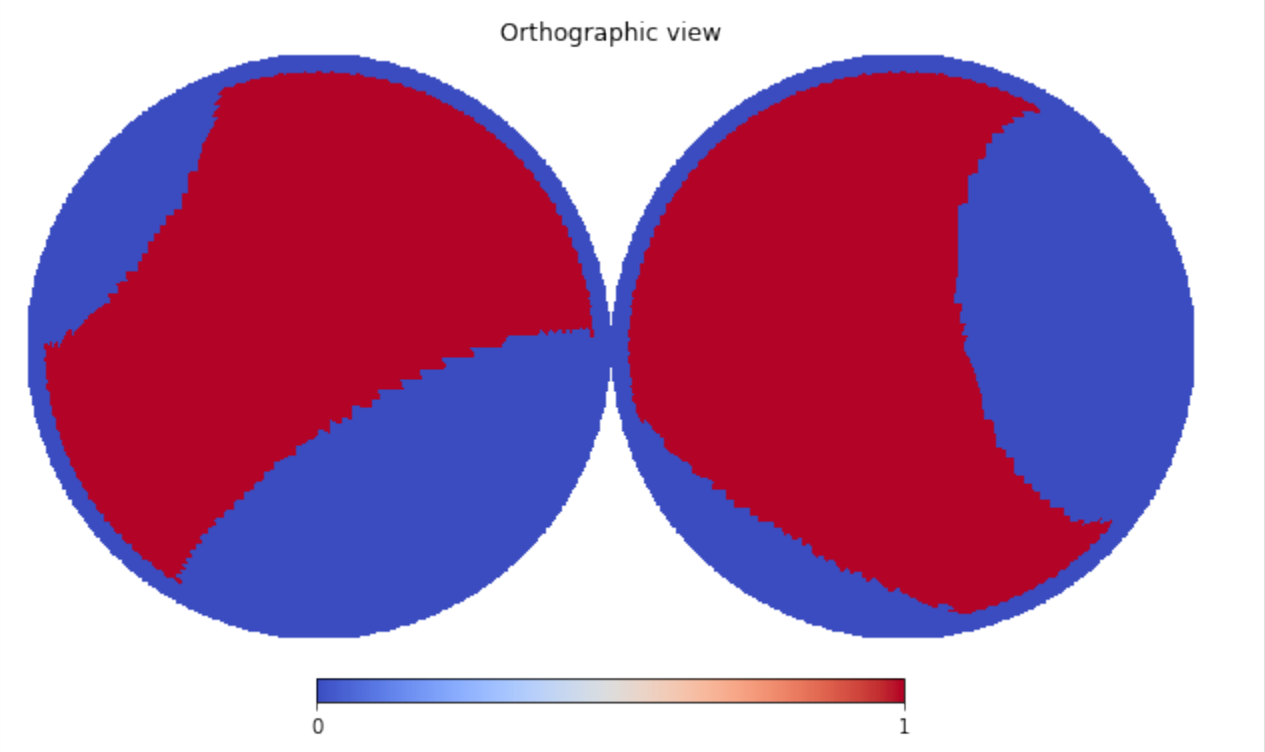
First of all, make an example, let's mask out low latitude $|l| < 20°$ pixels and those inside four local loops

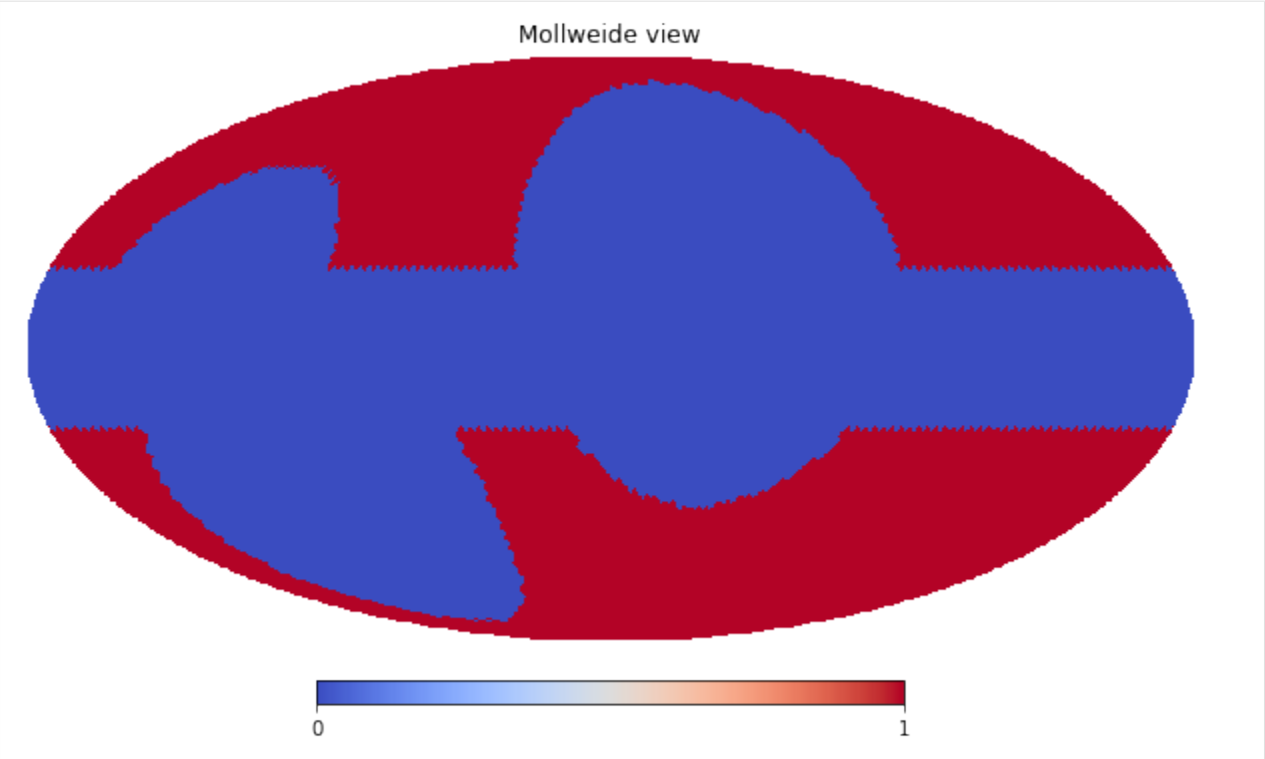
[2]:
```
mask_nside = 32

def mask_map_val(_nside,_ipix):
    """Mask loops and latitude"""
    l,b = hp.pix2ang(_nside,_ipix,lonlat=True)
    R = np.pi/180.
    cue = 1
    L = [329,100,124,315]
    B = [17.5,-32.5,15.5,48.5]
    D = [116,91,65,39.5]
    #LOOP I
    if( np.arccos(np.sin(b*R)*np.sin(B[0]*R)+np.cos(b*R)*np.cos(B[0]*R)*np.cos(l*R-
↪L[0]*R))<0.5*D[0]*R ):
        cue = 0
    #LOOP II
```

(continues on next page)

(continued from previous page)

```
    if( np.arccos(np.sin(b*R)*np.sin(B[1]*R)+np.cos(b*R)*np.cos(B[1]*R)*np.cos(l*R-
↪L[1]*R))<0.5*D[1]*R ):
        cue = 0
    #LOOP III
    if( np.arccos(np.sin(b*R)*np.sin(B[2]*R)+np.cos(b*R)*np.cos(B[2]*R)*np.cos(l*R-
↪L[2]*R))<0.5*D[2]*R ):
        cue = 0
    #LOOP IV
    if( np.arccos(np.sin(b*R)*np.sin(B[3]*R)+np.cos(b*R)*np.cos(B[3]*R)*np.cos(l*R-
↪L[3]*R))<0.5*D[3]*R ):
        cue = 0
    #STRIPE
    if(abs(b)<20.):
        cue = 0
    return cue

mask_map = np.zeros(hp.nside2npix(mask_nside))

for i in range(len(mask_map)):
    mask_map[i] = mask_map_val(mask_nside, i)

# Presents the generated mask map
hp.orthview(mask_map, cmap='coolwarm', rot=(0,90))
hp.mollview(mask_map, cmap='coolwarm')
```





The procedure to include the above created mask in a *Masks* dictionary is the same as the *Measurements* (at the moment, the is no helper equivalent to the *Dataset*, but this should not be an issue).

[3]:
```
masks = img_obs.Masks()
masks.append(('sync', '23', '32', 'I'), np.vstack([mask_map]))
```

13.2 Using the Masks

The while assembling the pipeline, the masks should be supplied while initializing the *Likelihood* object, as an extra argument, for example:

```
likelihood = img.EnsembleLikelihood(my_measurements, my_covariances,
                                    mask_dict=masks)
```

This allows the Pipeline apply the masks, during the IMAGINE run, on both: observed and simulated data.

13.3 Applying Masks directly

To understand or check what is going on internally when we provide a mask to a *Likelihood* object, we can apply it ourselves to a given Observable.

To illustrate this, let us first generate a mock synchrotron map using Hammurabi (using the usual trick).

[4]:
```
from imagine.simulators import Hammurabi

# Creates empty datasets
```

(continues on next page)

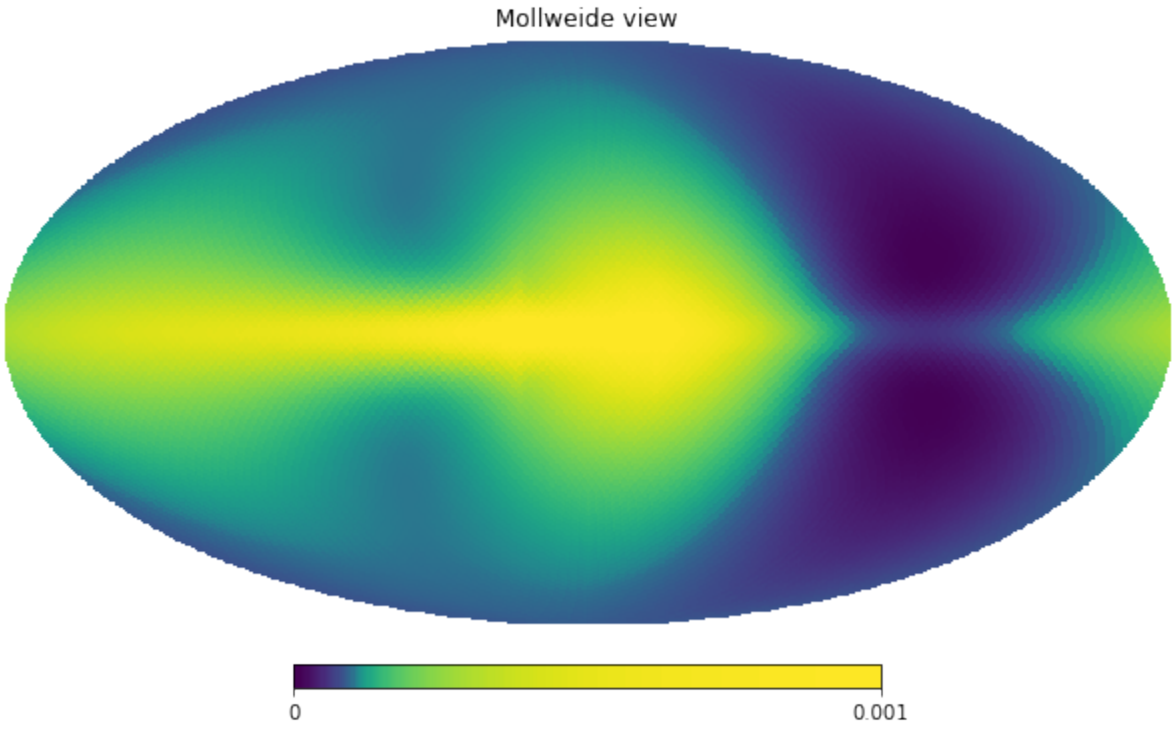
(continued from previous page)

```
sync_dset = img_obs.SynchrotronHEALPixDataset(data=np.empty(12*32**2)*u.K,
                                              frequency=23, type='I')
# Appends them to an Observables Dictionary
fakeMeasureDict = img_obs.Measurements()
fakeMeasureDict.append(dataset=sync_dset)
# Initializes the Simulator with the fake Measurements
simulator = Hammurabi(measurements=fakeMeasureDict)
# Initializes Fields
breg_wmap = BregLSA(parameters={'b0': 6.0, 'psi0': 27.9,
                                'psi1': 1.3, 'chi0': 24.6})
cre_ana = CREAna(parameters={'alpha': 3.0, 'beta': 0.0,
                             'theta': 0.0,'r0': 5.6, 'z0': 1.2,
                             'E0': 20.5, 'j0': 0.03})
fereg_ymw16 = TEregYMW16(parameters={})

# Produces the mock dataset
maps = simulator([breg_wmap, cre_ana, fereg_ymw16])
```

```
observable {}
|-->  sync {'cue': '1', 'freq': '23', 'nside': '32'}
```

We can now inspect how this data looks before the masking takes place

[5]:
```
unmasked = maps[('sync','23','32','I')].data[0]
hp.mollview(unmasked, norm='hist', min=0, max=1.0e-3)
```



The mask can be applied to the `maps` observable dictionary through the method `apply_mask`

[6]:
```
maps.apply_mask(masks)

raw_map = maps[('sync','23','4941','I')].data[0]
```

Applying a mask, however, changes the size of the data array

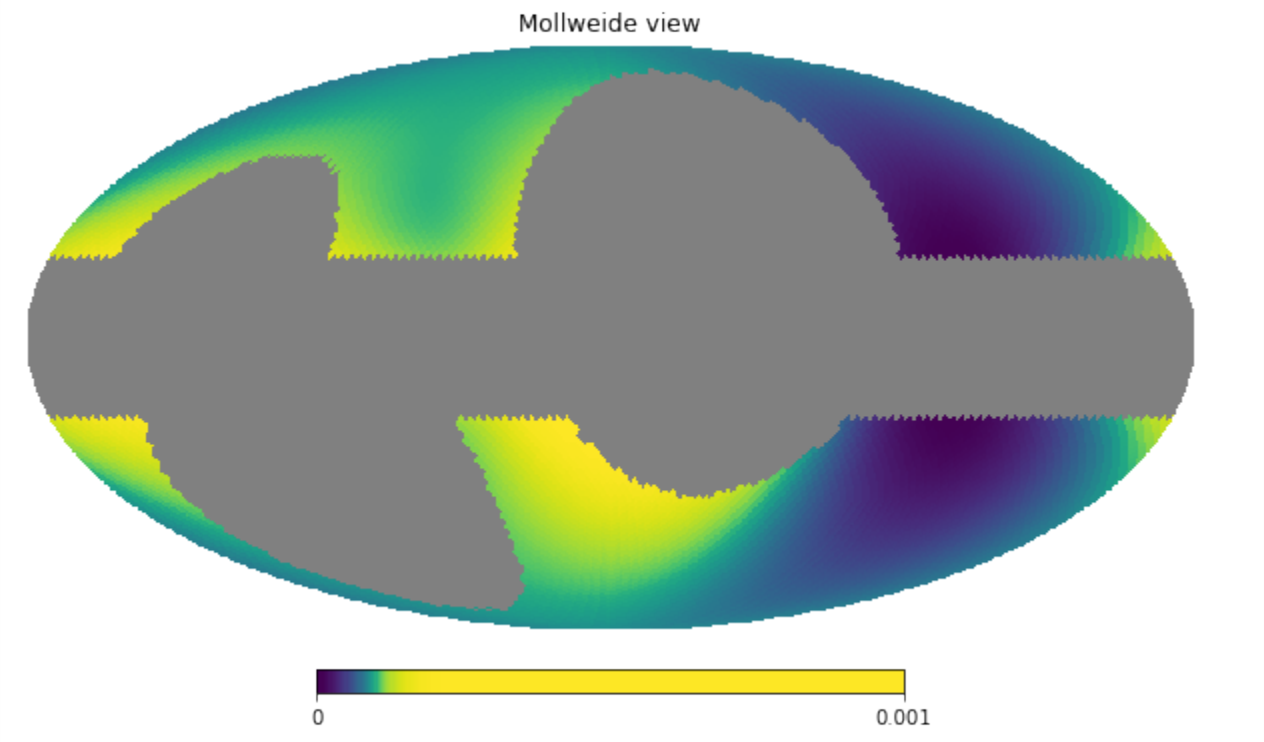
[7]:
```
print('Masked map size:', raw_map.size)
print('Orignal map size', unmasked.size)
```

```
Masked map size: 4941
Orignal map size 12288
```

This is expected: the whole point of masking is not using parts of the data which are unreliable or irrelevant for a particular purpose.

However, if, to check whether things are working correctly, we wish to *look* at masked image, we need to reconstruct it. This means creating a new image including the pixels which we previously have thrown away, as exemplified below:

[8]:
```
# Creates an empty array for the results
masked = np.empty(hp.nside2npix(mask_nside))
# Saves each pixel `raw_map` in `masked`, adding "unseen" tags for
# pixels in the mask
idx = 0
for i in range(len(mask_map)):
    if mask_map[i] == 0:
        masked[i] = hp.UNSEEN
    else:
        masked[i] = raw_map[idx]
        idx += 1
# Shows the image
hp.mollview(masked, norm='hist', min=0, max=1.0e-3)
```



13.4 Using mask in a Hammurabi X simulation

The previous procedure prevents that the masked pixels from influencing the inference, as they are not considered in the likelihood calculations. However, these pixels are still (needlessly) being computed by the Simulator.

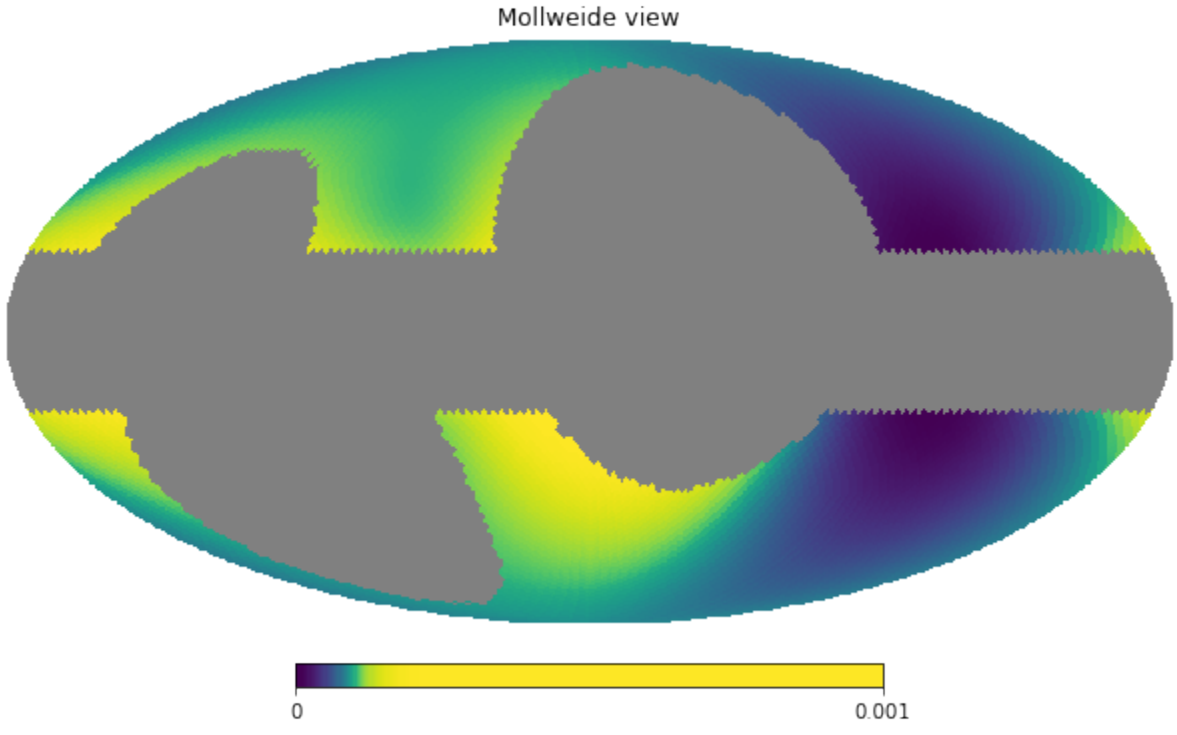
It is possible request Hammurabi to use the prepared mask, saving thus significant computing time!

But note that there is *only a single mask input entry* for hammurabi X, which means all outputs will be masked by the same mask. And the input mask will be treated at pivot resolution, output maps may be in various resolutions by adjusting the input mask into corresponding resolution. It is not trivial in changing mask resolution, for more details please check the hammurabi X wiki.

In the following we exemplify how to instruct Hammurabi to use the mask. This involves directly manipulating hammurabiX parameters through the hampyx object (an alternative, standardized, IMAGINE interface for this is under development).

[9]:
```
from os import path
# Uses IMAGINE standard temporary directory (which is erased after the end of the␣
↪session)
mask_filepath = path.join(img.rc['temp_dir'], 'mask_tutorial04.bin')
# Dumps the mask to disk using hammurabi X required format: i.e. float64, binary file
mask_map.tofile(mask_filepath)
simulator._ham.mod_par(['mask'],{'cue':'1','filename': mask_filepath, 'nside':
↪str(mask_nside)})
```

[10]:
```
# Re-runs the simulator which is using the mask internally
maps = simulator([breg_wmap, cre_ana, fereg_ymw16])
# Shows the new map
unmasked = maps[('sync','23','32','I')].data[0]
hp.mollview(masked, norm='hist', min=0, max=1.0e-3)
```



CHAPTER 14

imagine package

14.1 Subpackages

14.1.1 imagine.fields package

Subpackages

imagine.fields.hamx package

Submodules

imagine.fields.hamx.breg_lsa module

class imagine.fields.hamx.breg_lsa.**BregLSA**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin regular magnetic field WMAP-3yr LSA.

NAME = 'breg_lsa'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.breg_lsa.**BregLSAFactory**(**, grid=None, box-size=None, resolution=None, active_parameters=(), field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *BregLSA* (see its docs for details).

FIELD_CLASS
alias of *BregLSA*

DEFAULT_PARAMETERS = {'b0': 6.0, 'chi0': 25.0, 'psi0': 27.0, 'psi1': 0.9}

PRIORS = {'b0': <imagine.priors.basic_priors.FlatPrior object>, 'chi0': <imagine.prior

imagine.fields.hamx.brnd_es module

class imagine.fields.hamx.brnd_es.**BrndES**(**args*, *grid_nx=None*, *grid_ny=None*, *grid_nz=None*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin random magnetic field ES random GMF

set_grid_size(*nx=None*, *ny=None*, *nz=None*)
Changes the size of the grid used for the evaluation of the random field

NAME = 'breg_wmap'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.brnd_es.**BrndESFactory**(**args*, *grid_nx=None*, *grid_ny=None*, *grid_nz=None*, ***kwargs*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *BrndES* (see its docs for details).

FIELD_CLASS
alias of *BrndES*

DEFAULT_PARAMETERS = {'a0': 1.7, 'a1': 0, 'k0': 10, 'k1': 0.1, 'r0': 8, 'rho': 0.5, 'r

PRIORS = {'a0': <imagine.priors.basic_priors.FlatPrior object>, 'a1': <imagine.priors.

imagine.fields.hamx.cre_analytic module

class imagine.fields.hamx.cre_analytic.**CREAna**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin analytic cosmic ray electron distribution

NAME = 'cre_ana'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.cre_analytic.**CREAnaFactory**(***, *grid=None*, *box-size=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *CREAna* (see its docs for details).

FIELD_CLASS
alias of *CREAna*

DEFAULT_PARAMETERS = {'E0': 20.6, 'alpha': 3, 'beta': 0, 'j0': 0.0217, 'r0': 5, 'theta

PRIORS = {'E0': <imagine.priors.basic_priors.FlatPrior object>, 'alpha': <imagine.prio

imagine.fields.hamx.tereg_ymw16 module

class imagine.fields.hamx.tereg_ymw16.**TEregYMW16**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's thermal electron density model YMW16

NAME = 'tereg_ymw16'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.tereg_ymw16.**TEregYMW16Factory**(**, grid=None, boxsize=None, resolution=None, active_parameters=(), field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *TEregYMW16* (see its docs for details).

FIELD_CLASS
alias of *TEregYMW16*

DEFAULT_PARAMETERS = {}

PRIORS = {}

Module contents

class imagine.fields.hamx.**BregLSA**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin regular magnetic field WMAP-3yr LSA.

NAME = 'breg_lsa'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.**BregLSAFactory**(**, grid=None, boxsize=None, resolution=None, active_parameters=(), field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *BregLSA* (see its docs for details).

FIELD_CLASS
alias of *BregLSA*

DEFAULT_PARAMETERS = {'b0': 6.0, 'chi0': 25.0, 'psi0': 27.0, 'psi1': 0.9}

PRIORS = {'b0': <imagine.priors.basic_priors.FlatPrior object>, 'chi0': <imagine.prior

class imagine.fields.hamx.**BrndES**(**args*, *grid_nx=None*, *grid_ny=None*, *grid_nz=None*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin random magnetic field ES random GMF

set_grid_size(*nx=None*, *ny=None*, *nz=None*)
Changes the size of the grid used for the evaluation of the random field

NAME = 'breg_wmap'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.**BrndESFactory**(**args*, *grid_nx=None*, *grid_ny=None*, *grid_nz=None*, ***kwargs*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *BrndES* (see its docs for details).

FIELD_CLASS
alias of *BrndES*

DEFAULT_PARAMETERS = {'a0': 1.7, 'a1': 0, 'k0': 10, 'k1': 0.1, 'r0': 8, 'rho': 0.5, 'r

PRIORS = {'a0': <imagine.priors.basic_priors.FlatPrior object>, 'a1': <imagine.priors.

class imagine.fields.hamx.**CREAna**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's builtin analytic cosmic ray electron distribution

NAME = 'cre_ana'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.**CREAnaFactory**(***, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *CREAna* (see its docs for details).

FIELD_CLASS
alias of *CREAna*

DEFAULT_PARAMETERS = {'E0': 20.6, 'alpha': 3, 'beta': 0, 'j0': 0.0217, 'r0': 5, 'theta

PRIORS = {'E0': <imagine.priors.basic_priors.FlatPrior object>, 'alpha': <imagine.prio

class imagine.fields.hamx.**TEregYMW16**(**args*, ***kwargs*)
Bases: *imagine.fields.base_fields.DummyField*

This dummy field instructs the *Hammurabi* simulator class to use the HammurabiX's thermal electron density model YMW16

NAME = 'tereg_ymw16'

field_checklist
Hammurabi XML locations of physical parameters

simulator_controllist
Hammurabi XML locations of logical parameters

class imagine.fields.hamx.**TEregYMW16Factory**(***, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)
Bases: *imagine.fields.field_factory.FieldFactory*

Field factory that produces the dummy field *TEregYMW16* (see its docs for details).

FIELD_CLASS
alias of *TEregYMW16*

DEFAULT_PARAMETERS = {}

PRIORS = {}

Submodules

imagine.fields.base_fields module

This module contains basic base classes that can be used to include new fields in IMAGINE. The classes found here here correspond to the physical fields most commonly found by members of the IMAGINE community and may be improved in the future.

A brief summary of the module:

- *MagneticField* — for models of the galactic/Galactic Magnetic Field, $\mathbf{B}(\mathbf{r})$
- *ThermalElectronDensityField* — for models of the density of thermal electrons, $n_e(\mathbf{r})$
- CosmicRayElectronDensityField— for models of the density/flux of cosmic ray electrons, $n_{\rm cr}(\mathbf{r})$
- *DummyField* — allows passing parameters to a *Simulator* without having to evaluate anything on a Grid

See also *IMAGINE Components* section of the docs.

class imagine.fields.base_fields.**MagneticField**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: *imagine.fields.field.Field*

Base class for the inclusion of new models for magnetic fields. It should be subclassed following the template provided.

For more details, check the *Magnetic Fields* Section of the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of *imagine.fields.grid.BaseGrid* containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}

- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** – Random seed(s) for generating random field realisations

TYPE = 'magnetic_field'

UNITS = Unit("uG")

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

class imagine.fields.base_fields.**ThermalElectronDensityField**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.field.Field*

Base class for the inclusion of models for spatial distribution of thermal electrons. It should be subclassed following the template provided.

For more details, check the *Thermal electrons* Section of the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of *imagine.fields.grid.BaseGrid* containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}
- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** – Random seed(s) for generating random field realisations

TYPE = 'thermal_electron_density'

UNITS = Unit("1 / cm3")

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

class imagine.fields.base_fields.**DummyField**(**args*, ***kwargs*)

Bases: *imagine.fields.field.Field*

Base class for a dummy Field used for sending parameters and settings to specific Simulators rather than computing and storing a physical field.

compute_field(**args*, ***kwargs*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use *get_data()* instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

get_data(*i_realization=0*, *dependencies={}*)
Mock evaluation of the dummy field defined by this class.

Parameters

- **i_realization** (*int*) – Index of the current realization
- **dependencies** (*dict*) – If the `dependencies_list` is non-empty, a dictionary containing the requested dependencies must be provided.

Returns **parameters** – Dictionary of containing a copy of the Field parameters including an extra entry with the random seed that should be used with the present realization (under the key: 'random_seed')

Return type dict

TYPE = 'dummy'

UNITS = None

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

field_checklist
Dictionary with all parameter names as keys

simulator_controllist
Dictionary containing fixed Simulator settings

imagine.fields.basic_fields module

class imagine.fields.basic_fields.**ConstantMagneticField**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.MagneticField*

Constant magnetic field

The field parameters are: 'Bx', 'By', 'Bz', which correspond to the fixed components B_x, B_x and B_z.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'constant_B'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.basic_fields.**ConstantThermalElectrons**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Constant magnetic field

The field parameters are: ‘ne’, the number density of thermal electrons

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'constant_TE'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.basic_fields.**ExponentialThermalElectrons**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Thermal electron distribution in a double exponential disc characterized by a scale-height and a scale-radius, i.e.

..math:

```
n_e(R) = n_0 e^{-R/R_e} e^{-|z|/h_e}
```

where R is the cylindrical radius and z is the vertical coordinate.

The field parameters are: the ‘central_density’, *n_0*; ‘scale_radius‘, R_e; and ‘scale_height’, h_e.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'exponential_disc_thermal_electrons'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.basic_fields.**RandomThermalElectrons**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Thermal electron densities drawn from a Gaussian distribution

NB This may lead to negative densities depending on the choice of parameters. This may be controlled with the ‘min_ne’ parameter which sets a minimum value for the density field (i.e. any value smaller than the minimum density is set to min_ne).

The field parameters are: ‘mean’, the mean of the distribution; ‘std’, the standard deviation of the distribution; and ‘min_ne’, the aforementioned minimum density. To disable the minimum density requirement, it may be set to NaN.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'random_thermal_electrons'

STOCHASTIC_FIELD = True

field_checklist
Dictionary with all parameter names as keys

imagine.fields.field module

class imagine.fields.field.**Field**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: *imagine.tools.class_tools.BaseClass*

This is the base class which can be used to include a completely new field in the IMAGINE pipeline. Base classes for specific physical quantites (e.g. magnetic fields) are already available in the module *imagine.fields.basic_fields*. Thus, before subclassing *GeneralField*, check whether a more specialized subclass is not available.

For more details check the *Fields* section in the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of *imagine.fields.grid.BaseGrid* containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}
- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** (*list*) – Random seeds for generating random field realisations

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use *get_data()* instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

get_data(*i_realization=0*, *dependencies={}*)
Evaluates the physical field defined by this class.

Parameters

- **i_realization** (*int*) – If the field is stochastic, this indexes the realization generated. Default value: 0 (i.e. the first realization).
- **dependencies** (*dict*) – If the *dependencies_list* is non-empty, a dictionary containing the requested dependencies must be provided.

Returns **field** – Array of shape *data_shape* whose contents are described by *data_description* in units `field_units`.

Return type astropy.units.quantity.Quantity

REQ_ATTRS = ['TYPE', 'NAME', 'UNITS']

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

dependencies_list
Dependencies on other fields

ensemble_seeds

field_checklist
Dictionary with all parameter names as keys

name
Name of the field

parameters
Dictionary containing parameters used for this field.

stochastic_field
True if the field is stochastic or *False* if the field is deterministic (i.e. the output depends only on the parameter values and not on the seed value). Default value is *False* if subclass does not override `STOCHASTIC_FIELD`.

type
Type of the field

units
Physical units of the field

imagine.fields.field_factory module

class imagine.fields.field_factory.**FieldFactory**(*, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)

Bases: *imagine.tools.class_tools.BaseClass*

FieldFactory is designed for generating ensemble of field configuration DIRECTLY and/or handle of field to be conducted by simulators

Through calling the factory, the factory object takes a given set of variable values (can be at any chain point in bayesian analysis) and translates it into physical parameter values, returning a field object with current parameter set.

Example

To include a new Field_Factory, one needs to create a derived class with customized initialization. Below we show an example which is compatible with the `xConstantField` showed in the *Fields* section of the documentation:

```
@icy
class xConstantField_Factory(GeneralFieldFactory):
    def __init__(self, grid=None, boxsize=None, resolution=None):
        super().__init__(grid, boxsize, resolution)
        self.field_class = xConstantField
```

(continues on next page)

(continued from previous page)

```
        self.default_parameters = {'constantA': 5.0}
        self.parameter_ranges = {'constantA': [-10., 10.]}
```

Parameters

- **boxsize** (*list/tuple of floats*) – The physical size of simulation box (i.e. edges of the box).
- **resolution** (*list/tuple of ints*) – The discretization size in corresponding dimension
- **grid** (*imagine.fields.BaseGrid or None*) – If present, the supplied instance of *imagine.fields.BaseGrid* is used and the arguments *boxsize* and *resolution* are ignored
- **field_kwargs** (*dict*) – Any extra keyword arguments that should be used in the field instantiation

__call__(*, *variables={}*, *ensemble_size=None*, *ensemble_seeds=None*)

Takes an active variable dictionary, an ensemble size and a random seed value, translates the active variables to parameter values (updating the default parameter dictionary accordingly) and send this to an instance of the field class.

Parameters

- **variables** (*dict*) – Dictionary of variables with name and value
- **ensemble_size** (*int*) – Number of instances in a field ensemble
- **ensemble_seeds** – seeds for generating random numbers in realising instances in field ensemble if ensemble_seeds is None, field_class initialization will take all seed as 0

Returns **result_field** – a Field object

Return type *imagine.fields.field.Field*

REQ_ATTRS = ['FIELD_CLASS', 'DEFAULT_PARAMETERS', 'PRIORS']

active_parameters

Tuple of parameter names which can vary, not necessary to cover all default parameters

default_parameters

Dictionary storing parameter name as entry, default parameter value as content

default_variables

A dictionary containing default parameter values converted into default normalized variables (i.e with values scaled to be in the range [0,1]).

field_class

Python class whose instances are produced by the present factory

field_name

Name of the physical field

field_type

Type of physical field.

field_units

Units of physical field.

grid

Instance of *imagine.fields.BaseGrid* containing a 3D grid where the field is/was evaluated

name

parameter_ranges
Dictionary storing varying range of all default parameters in the form {'parameter-name': (min, max)}

priors
A dictionary containing the priors associated with each parameter. Each prior is represented by an instance of *imagine.priors.prior.GeneralPrior*.

To set new priors one can update the priors dictionary using attribution (any missing values will be set to *imagine.priors.basic_priors.FlatPrior*).

resolution
How many bins on each direction of simulation box

imagine.fields.grid module

Contains the definition of the BaseGrid class and an example of its application: a basic uniform grid.

This was strongly based on GalMag's Grid class, initially developed by Theo Steininger

class imagine.fields.grid.**BaseGrid**(*box*, *resolution*)
Bases: *imagine.tools.class_tools.BaseClass*

Defines a 3D grid object for a given choice of box dimensions and resolution.

This is a base class. To create your own grid, you need to subclass *BaseGrid* and override the method *generate_coordinates()*.

Calling the attributes does the conversion between different coordinate systems automatically (spherical, cylindrical and cartesian coordinates centred at the galaxy centre).

Parameters

- **box** (*3x2-array_like*) – Box limits
- **resolution** (*3-array_like*) – containing the resolution along each axis.

box
Box limits

Type 3x2-array_like

resolution
Containing the resolution along each axis (the *shape* of the grid).

Type 3-array_like

generate_coordinates()
Placeholder for method which uses the information in the attributes *box* and *resolution* to return a dictionary containing the values of (either) the coordinates ('x','y','z') or ('r_cylindrical', 'phi','z'), ('r_spherical','theta', 'phi')

This method is *automatically* called the first time any coordinate is read.

coordinates
A dictionary contaning all the coordinates

cos_phi
$\cos(\phi)$

cos_theta
$\cos(\theta)$

phi
Azimuthal coordinate, ϕ

r_cylindrical
Cylindrical radial coordinate, s

r_spherical
Spherical radial coordinate, r

shape
The same as *resolution*

sin_phi
$\sin(\phi)$

sin_theta
$\sin(\theta)$

theta
Polar coordinate, θ

x
Horizontal coordinate, x

y
Horizontal coordinate, y

z
Vertical coordinate, z

class imagine.fields.grid.**UniformGrid**(*box*, *resolution*, *grid_type='cartesian'*)
Bases: *imagine.fields.grid.BaseGrid*

Defines a 3D grid object for a given choice of box dimensions and resolution. The grid is uniform in the selected coordinate system (which is chosen through the parameter *grid_type*.

Example

```
>>> import magnetizer.grid as grid
>>> import astropy.units as u
>>> xlims = [0,4]*u.kpc; ylims = [1,2]*u.kpc; zlims = [1,1]*u.kpc
>>> g = grid.UniformGrid([xlims, ylims, zlims], [5,2,1])
>>> g.x
array([[[0.],[0.]], [[1.],[1.]], [[2.],[2.]], [[3.],[3.]], [[4.],[4.]]])
>>> g.y
array([[[1.], [2.]], [[1.], [2.]], [[1.], [2.]], [[1.], [2.]], [[1.], [2.]]])
```

Calling the attributes does the conversion between different coordinate systems automatically.

Parameters

- **box** (*3x2-array_like*) – Box limits. Each row corresponds to a different coordinate and should contain units. For 'cartesian' grid_type, the rows should contain (in order)'x','y' and 'z'. For 'cylindrical' they should have 'r_cylindrical', 'phi' and 'z'. for 'spherical', 'r_spherical','theta' and 'phi'.
- **resolution** (*3-array_like*) – containing the resolution along each axis.
- **grid_type** (*str, optional*) – Choice between 'cartesian', 'spherical' and 'cylindrical' *uniform* coordinate grids. Default: 'cartesian'

generate_coordinates()
This method is *automatically* called internally the first time any coordinate is requested.

Generates a uniform grid based on the attributes *box*, *resolution* and *grid_type* and returns it in a dictionary.

Returns **coordinates_dict** – Dictionary containing the keys ('x','y','z') if *grid_type* is 'cartesian', ('r_cylindrical', 'phi','z') if *grid_type* is spherical, and ('r_spherical','theta', 'phi') if *grid_type* is 'cylindrical'.

Return type dict

imagine.fields.test_field module

class imagine.fields.test_field.**CosThermalElectronDensity**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Toy model for naively oscilating thermal electron distribution following:

$$n_e(x, y, z) = n_0[1 + \cos(ax + \alpha)][1 + \cos(by + \beta)][1 + \cos(cy + \gamma)]$$

The field parameters are: 'n0', which corresponds to n_0; and 'a', 'b', 'c', 'alpha', 'beta', 'gamma', which are a, b, c, α, β, γ, respectively.

compute_field(*seed*)

This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'cos_therm_electrons'

field_checklist

Dictionary with all parameter names as keys

class imagine.fields.test_field.**CosThermalElectronDensityFactory**(*, *grid=None*, *box-size=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)

Bases: *imagine.fields.field_factory.FieldFactory*

Field factory associated with the `CosThermalElectronDensityw` class

FIELD_CLASS

alias of *CosThermalElectronDensity*

DEFAULT_PARAMETERS = {'a': <Quantity 0. rad / kpc>, 'alpha': <Quantity 0. rad>, 'b': <

PRIORS = {'a': <imagine.priors.basic_priors.FlatPrior object>, 'alpha': <imagine.prior

d = <imagine.priors.basic_priors.FlatPrior object>

k = <imagine.priors.basic_priors.FlatPrior object>

class imagine.fields.test_field.**NaiveGaussianMagneticField**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.MagneticField*

Toy model for naive Gaussian random field for testing.

The values of each of the magnetic field components are individually drawn from a Gaussian distribution with mean 'a0' and standard deviation 'b0'.

Warning: divergence may be non-zero!

compute_field(*seed*)

This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'naive_gaussian_magnetic_field'

STOCHASTIC_FIELD = True

field_checklist

Dictionary with all parameter names as keys

class imagine.fields.test_field.**NaiveGaussianMagneticFieldFactory**(*, *grid=None*, *box-size=None*, *resolu-tion=None*, *ac-tive_parameters=()*, *field_kwargs={}*)

Bases: *imagine.fields.field_factory.FieldFactory*

Field factory associated with the *NaiveGaussianMagneticField* class

FIELD_CLASS

alias of *NaiveGaussianMagneticField*

DEFAULT_PARAMETERS = {'a0': <Quantity 1. uG>, 'b0': <Quantity 0.1 uG>}

PRIORS = {'a0': <imagine.priors.basic_priors.FlatPrior object>, 'b0': <imagine.priors.

Module contents

class imagine.fields.**MagneticField**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.field.Field*

Base class for the inclusion of new models for magnetic fields. It should be subclassed following the template provided.

For more details, check the *Magnetic Fields* Section of the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of `imagine.fields.grid.BaseGrid` containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}
- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** – Random seed(s) for generating random field realisations

TYPE = 'magnetic_field'

UNITS = Unit("uG")

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

class imagine.fields.**ThermalElectronDensityField**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: `imagine.fields.field.Field`

Base class for the inclusion of models for spatial distribution of thermal electrons. It should be subclassed following the template provided.

For more details, check the *Thermal electrons* Section of the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of `imagine.fields.grid.BaseGrid` containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}
- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** – Random seed(s) for generating random field realisations

TYPE = 'thermal_electron_density'

UNITS = Unit("1 / cm3")

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

class imagine.fields.**DummyField**(**args*, ***kwargs*)
Bases: `imagine.fields.field.Field`

Base class for a dummy Field used for sending parameters and settings to specific Simulators rather than computing and storing a physical field.

compute_field(**args*, ***kwargs*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

get_data(*i_realization=0*, *dependencies={}*)
Mock evaluation of the dummy field defined by this class.

Parameters

- **i_realization** (*int*) – Index of the current realization
- **dependencies** (*dict*) – If the `dependencies_list` is non-empty, a dictionary containing the requested dependencies must be provided.

Returns **parameters** – Dictionary of containing a copy of the Field parameters including an extra entry with the random seed that should be used with the present realization (under the key: 'random_seed')

Return type dict

TYPE = 'dummy'

UNITS = None

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

field_checklist
Dictionary with all parameter names as keys

simulator_controllist
Dictionary containing fixed Simulator settings

class imagine.fields.**ConstantMagneticField**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.MagneticField*

Constant magnetic field

The field parameters are: 'Bx', 'By', 'Bz', which correspond to the fixed components B_x, B_x and B_z.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'constant_B'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.**ConstantThermalElectrons**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Constant magnetic field

The field parameters are: 'ne', the number density of thermal electrons

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'constant_TE'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.**ExponentialThermalElectrons**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Thermal electron distribution in a double exponential disc characterized by a scale-height and a scale-radius, i.e.

..math:

```
n_e(R) = n_0 e^{-R/R_e} e^{-|z|/h_e}
```

where R is the cylindrical radius and z is the vertical coordinate.

The field parameters are: the 'central_density', *n_0*; 'scale_radius', R_e; and 'scale_height', h_e.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'exponential_disc_thermal_electrons'

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.**RandomThermalElectrons**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Thermal electron densities drawn from a Gaussian distribution

NB This may lead to negative densities depending on the choice of parameters. This may be controlled with the 'min_ne' parameter which sets a minimum value for the density field (i.e. any value smaller than the minimum density is set to min_ne).

The field parameters are: 'mean', the mean of the distribution; 'std', the standard deviation of the distribution; and 'min_ne', the aforementioned minimum density. To disable the minimum density requirement, it may be set to NaN.

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'random_thermal_electrons'

STOCHASTIC_FIELD = True

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.**Field**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)
Bases: *imagine.tools.class_tools.BaseClass*

This is the base class which can be used to include a completely new field in the IMAGINE pipeline. Base classes for specific physical quantites (e.g. magnetic fields) are already available in the module *imagine.fields.basic_fields*. Thus, before subclassing *GeneralField*, check whether a more specialized subclass is not available.

For more details check the *Fields* section in the documentation.

Parameters

- **grid** (*imagine.fields.grid.BaseGrid*) – Instance of *imagine.fields.grid.BaseGrid* containing a 3D grid where the field is evaluated
- **parameters** (*dict*) – Dictionary of full parameter set {name: value}
- **ensemble_size** (*int*) – Number of realisations in field ensemble
- **ensemble_seeds** (*list*) – Random seeds for generating random field realisations

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use *get_data()* instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

get_data(*i_realization=0*, *dependencies={}*)
Evaluates the physical field defined by this class.

Parameters

- **i_realization** (*int*) – If the field is stochastic, this indexes the realization generated. Default value: 0 (i.e. the first realization).
- **dependencies** (*dict*) – If the *dependencies_list* is non-empty, a dictionary containing the requested dependencies must be provided.

Returns **field** – Array of shape *data_shape* whose contents are described by *data_description* in units `field_units`.

Return type astropy.units.quantity.Quantity

REQ_ATTRS = ['TYPE', 'NAME', 'UNITS']

data_description
Summary of what is in each axis of the data array

data_shape
Shape of the field data array

dependencies_list
Dependencies on other fields

ensemble_seeds

field_checklist
Dictionary with all parameter names as keys

name
Name of the field

parameters
Dictionary containing parameters used for this field.

stochastic_field
True if the field is stochastic or *False* if the field is deterministic (i.e. the output depends only on the parameter values and not on the seed value). Default value is *False* if subclass does not override `STOCHASTIC_FIELD`.

type
Type of the field

units
Physical units of the field

class imagine.fields.**FieldFactory**(*, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)

Bases: *imagine.tools.class_tools.BaseClass*

FieldFactory is designed for generating ensemble of field configuration DIRECTLY and/or handle of field to be conducted by simulators

Through calling the factory, the factory object takes a given set of variable values (can be at any chain point in bayesian analysis) and translates it into physical parameter values, returning a field object with current parameter set.

Example

To include a new Field_Factory, one needs to create a derived class with customized initialization. Below we show an example which is compatible with the `xConstantField` showed in the *Fields* section of the documentation:

```
@icy
class xConstantField_Factory(GeneralFieldFactory):
    def __init__(self, grid=None, boxsize=None, resolution=None):
        super().__init__(grid, boxsize, resolution)
        self.field_class = xConstantField
        self.default_parameters = {'constantA': 5.0}
        self.parameter_ranges = {'constantA': [-10., 10.]}
```

Parameters

- **boxsize** (*list/tuple of floats*) – The physical size of simulation box (i.e. edges of the box).
- **resolution** (*list/tuple of ints*) – The discretization size in corresponding dimension
- **grid** (*imagine.fields.BaseGrid or None*) – If present, the supplied instance of *imagine.fields.BaseGrid* is used and the arguments *boxsize* and *resolution* are ignored
- **field_kwargs** (*dict*) – Any extra keyword arguments that should be used in the field instantiation

__call__(*, *variables={}*, *ensemble_size=None*, *ensemble_seeds=None*)
Takes an active variable dictionary, an ensemble size and a random seed value, translates the active variables to parameter values (updating the default parameter dictionary accordingly) and send this to an instance of the field class.

Parameters

- **variables** (*dict*) – Dictionary of variables with name and value
- **ensemble_size** (*int*) – Number of instances in a field ensemble
- **ensemble_seeds** – seeds for generating random numbers in realising instances in field ensemble if ensemble_seeds is None, field_class initialization will take all seed as 0

Returns **result_field** – a Field object

Return type *imagine.fields.field.Field*

REQ_ATTRS = ['FIELD_CLASS', 'DEFAULT_PARAMETERS', 'PRIORS']

active_parameters
Tuple of parameter names which can vary, not necessary to cover all default parameters

default_parameters
Dictionary storing parameter name as entry, default parameter value as content

default_variables
A dictionary containing default parameter values converted into default normalized variables (i.e with values scaled to be in the range [0,1]).

field_class
Python class whose instances are produced by the present factory

field_name
Name of the physical field

field_type
Type of physical field.

field_units
Units of physical field.

grid
Instance of *imagine.fields.BaseGrid* containing a 3D grid where the field is/was evaluated

name

parameter_ranges
Dictionary storing varying range of all default parameters in the form {'parameter-name': (min, max)}

priors
A dictionary containing the priors associated with each parameter. Each prior is represented by an instance of *imagine.priors.prior.GeneralPrior*.

To set new priors one can update the priors dictionary using attribution (any missing values will be set to *imagine.priors.basic_priors.FlatPrior*).

resolution
How many bins on each direction of simulation box

class imagine.fields.**BaseGrid**(*box*, *resolution*)
Bases: *imagine.tools.class_tools.BaseClass*

Defines a 3D grid object for a given choice of box dimensions and resolution.

This is a base class. To create your own grid, you need to subclass *BaseGrid* and override the method *generate_coordinates()*.

Calling the attributes does the conversion between different coordinate systems automatically (spherical, cylindrical and cartesian coordinates centred at the galaxy centre).

Parameters

- **box** (*3x2-array_like*) – Box limits
- **resolution** (*3-array_like*) – containing the resolution along each axis.

box
Box limits

Type 3x2-array_like

resolution
Containing the resolution along each axis (the *shape* of the grid).

Type 3-array_like

generate_coordinates()
Placeholder for method which uses the information in the attributes *box* and *resolution* to return a dictionary containing the values of (either) the coordinates ('x','y','z') or ('r_cylindrical', 'phi','z'), ('r_spherical','theta', 'phi')

This method is *automatically* called the first time any coordinate is read.

coordinates
A dictionary contaning all the coordinates

cos_phi
$\cos(\phi)$

cos_theta
$\cos(\theta)$

phi
Azimuthal coordinate, ϕ

r_cylindrical
Cylindrical radial coordinate, s

r_spherical
Spherical radial coordinate, r

shape
The same as *resolution*

sin_phi
$\sin(\phi)$

sin_theta
$\sin(\theta)$

theta
Polar coordinate, θ

x
Horizontal coordinate, x

y
Horizontal coordinate, y

z
Vertical coordinate, z

class imagine.fields.**UniformGrid**(*box*, *resolution*, *grid_type='cartesian'*)
Bases: *imagine.fields.grid.BaseGrid*

Defines a 3D grid object for a given choice of box dimensions and resolution. The grid is uniform in the selected coordinate system (which is chosen through the parameter *grid_type*.

Example

```
>>> import magnetizer.grid as grid
>>> import astropy.units as u
>>> xlims = [0,4]*u.kpc; ylims = [1,2]*u.kpc; zlims = [1,1]*u.kpc
>>> g = grid.UniformGrid([xlims, ylims, zlims], [5,2,1])
>>> g.x
array([[[0.],[0.]], [[1.],[1.]], [[2.],[2.]], [[3.],[3.]], [[4.],[4.]]])
>>> g.y
array([[[1.], [2.]], [[1.], [2.]], [[1.], [2.]], [[1.], [2.]], [[1.], [2.]]])
```

Calling the attributes does the conversion between different coordinate systems automatically.

Parameters

- **box** (*3x2-array_like*) – Box limits. Each row corresponds to a different coordinate and should contain units. For 'cartesian' grid_type, the rows should contain (in order)'x','y' and 'z'. For 'cylindrical' they should have 'r_cylindrical', 'phi' and 'z'. for 'spherical', 'r_spherical','theta' and 'phi'.
- **resolution** (*3-array_like*) – containing the resolution along each axis.
- **grid_type** (*str, optional*) – Choice between 'cartesian', 'spherical' and 'cylindrical' *uniform* coordinate grids. Default: 'cartesian'

generate_coordinates()

This method is *automatically* called internally the first time any coordinate is requested.

Generates a uniform grid based on the attributes *box*, *resolution* and *grid_type* and returns it in a dictionary.

Returns **coordinates_dict** – Dictionary containing the keys ('x','y','z') if *grid_type* is 'cartesian', ('r_cylindrical', 'phi','z') if *grid_type* is spherical, and ('r_spherical','theta', 'phi') if *grid_type* is 'cylindrical'.

Return type dict

class imagine.fields.**CosThermalElectronDensity**(*grid*, ***, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.ThermalElectronDensityField*

Toy model for naively oscilating thermal electron distribution following:

$$n_e(x, y, z) = n_0[1 + \cos(ax + \alpha)][1 + \cos(by + \beta)][1 + \cos(cy + \gamma)]$$

The field parameters are: 'n0', which corresponds to n_0; and 'a', 'b', 'c', 'alpha', 'beta', 'gamma', which are a, b, c, α, β, γ, respectively.

compute_field(*seed*)

This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'cos_therm_electrons'

field_checklist

Dictionary with all parameter names as keys

class imagine.fields.**CosThermalElectronDensityFactory**(*, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)

Bases: *imagine.fields.field_factory.FieldFactory*

Field factory associated with the `CosThermalElectronDensityw` class

FIELD_CLASS
alias of *CosThermalElectronDensity*

DEFAULT_PARAMETERS = {'a': <Quantity 0. rad / kpc>, 'alpha': <Quantity 0. rad>, 'b': <

PRIORS = {'a': <imagine.priors.basic_priors.FlatPrior object>, 'alpha': <imagine.prior

d = <imagine.priors.basic_priors.FlatPrior object>

k = <imagine.priors.basic_priors.FlatPrior object>

class imagine.fields.**NaiveGaussianMagneticField**(*grid*, *, *parameters={}*, *ensemble_size=None*, *ensemble_seeds=None*, *dependencies={}*)

Bases: *imagine.fields.base_fields.MagneticField*

Toy model for naive Gaussian random field for testing.

The values of each of the magnetic field components are individually drawn from a Gaussian distribution with mean 'a0' and standard deviation 'b0'.

Warning: divergence may be non-zero!

compute_field(*seed*)
This should be overridden with a derived class. It must return an array with dimensions compatible with the associated *field_type*. See *documentation*.

Should not be used directly (use `get_data()` instead).

Parameters **seed** (*int*) – If the field is stochastic, this argument allows setting the random number generator seed accordingly.

NAME = 'naive_gaussian_magnetic_field'

STOCHASTIC_FIELD = True

field_checklist
Dictionary with all parameter names as keys

class imagine.fields.**NaiveGaussianMagneticFieldFactory**(*, *grid=None*, *boxsize=None*, *resolution=None*, *active_parameters=()*, *field_kwargs={}*)

Bases: *imagine.fields.field_factory.FieldFactory*

Field factory associated with the *NaiveGaussianMagneticField* class

FIELD_CLASS
alias of *NaiveGaussianMagneticField*

DEFAULT_PARAMETERS = {'a0': <Quantity 1. uG>, 'b0': <Quantity 0.1 uG>}

PRIORS = {'a0': <imagine.priors.basic_priors.FlatPrior object>, 'b0': <imagine.priors.

14.1.2 imagine.likelihoods package

Submodules

imagine.likelihoods.ensemble_likelihood module

ensemble likelihood, described in IMAGINE techincal report in principle it combines covariance matrices from both observations and simulations

class imagine.likelihoods.ensemble_likelihood.**EnsembleLikelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.likelihoods.likelihood.Likelihood*

EnsembleLikelihood class initialization function

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

call(*simulations_dict*)

EnsembleLikelihood class call function

Parameters **simulations_dict** (*imagine.observables.observable_dict.Simulations*) – Simulations object

Returns **likelicache** – log-likelihood value (copied to all nodes)

Return type float

imagine.likelihoods.likelihood module

Likelihood class defines likelihood posterior function to be used in Bayesian analysis

member fuctions:

__init__

requires Measurements object Covariances object (optional) Masks object (optional)

call

running LOG-likelihood calculation requires ObservableDict object

class imagine.likelihoods.likelihood.**Likelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.tools.class_tools.BaseClass*

Base class that defines likelihood posterior function to be used in Bayesian analysis

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

__call__(**args*, ***kwargs*)
Call self as a function.

call(*observable_dict*)

Parameters

- **observable_dict** (*imagine.observables.observable_dict*)
- **variables**

covariance_dict

mask_dict

measurement_dict

imagine.likelihoods.simple_likelihood module

class imagine.likelihoods.simple_likelihood.**SimpleLikelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.likelihoods.likelihood.Likelihood*

A simple Likelihood class

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

call(*observable_dict*)
SimpleLikelihood object call function

Parameters **observable_dict** (*imagine.observables.observable_dict.Simulations*) – Simulations object

Returns **likelicache** – log-likelihood value (copied to all nodes)

Return type float

Module contents

class imagine.likelihoods.**EnsembleLikelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.likelihoods.likelihood.Likelihood*

EnsembleLikelihood class initialization function

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

call(*simulations_dict*)
EnsembleLikelihood class call function

Parameters **simulations_dict** (*imagine.observables.observable_dict.Simulations*) – Simulations object

Returns **likelicache** – log-likelihood value (copied to all nodes)

Return type float

class imagine.likelihoods.**Likelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.tools.class_tools.BaseClass*

Base class that defines likelihood posterior function to be used in Bayesian analysis

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

__call__(**args*, ***kwargs*)
Call self as a function.

call(*observable_dict*)

Parameters

- **observable_dict** (*imagine.observables.observable_dict*)
- **variables**

covariance_dict

mask_dict

measurement_dict

class imagine.likelihoods.**SimpleLikelihood**(*measurement_dict*, *covariance_dict=None*, *mask_dict=None*)

Bases: *imagine.likelihoods.likelihood.Likelihood*

A simple Likelihood class

Parameters

- **measurement_dict** (*imagine.observables.observable_dict.Measurements*) – Measurements
- **covariance_dict** (*imagine.observables.observable_dict.Covariances*) – Covariances
- **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks

call(*observable_dict*)
SimpleLikelihood object call function

Parameters **observable_dict** (*imagine.observables.observable_dict.Simulations*) – Simulations object

Returns **likelicache** – log-likelihood value (copied to all nodes)

Return type float

14.1.3 imagine.observables package

Submodules

imagine.observables.dataset module

Datasets are auxiliary classes used to facilitate the reading and inclusion of observational data in the IMAGINE pipeline

class imagine.observables.dataset.**Dataset**

Bases: *imagine.tools.class_tools.BaseClass*

Base class for writing helpers to convert arbitrary observational datasets into IMAGINE's standardized format

REQ_ATTRS = ['NAME']

cov

data

Array in the shape (1, N)

frequency

key

Key used in the Observables_dictionary

name

class imagine.observables.dataset.**TabularDataset**(*data*, *name*, *data_column=None*, *units=None*, *coordinates_type='galactic'*, *lon_column='lon'*, *lat_column='lat'*, *x_column='x'*, *y_column='y'*, *z_column='z'*, *error_column=None*, *frequency='nan'*, *tag='nan'*)

Bases: *imagine.observables.dataset.Dataset*

Base class for tabular datasets, where the data is input in either in a Python dictionary-like object (`dict`, `astropy.table.Table`, `pandas.DataFrame`, etc).

Parameters

- **data** (*dict-like*) – Can be a `dict`, `astropy.table.Table`, `pandas.DataFrame`, or similar object containing the data.
- **name** (*str*) – Standard name of this type of observable. E.g. 'fd', 'sync', 'dm'.
- **data_column** (*str*) – Key used to access the relevant dataset from the provided data (i.e. data[data_column]).
- **units** (*astropy.units.Unit or str*) – Units used for the data.
- **coordinates_type** (*str*) – Type of coordinates used. Can be 'galactic' or 'cartesian'.
- **lon_column** (*str*) – Key used to access the Galactic longitudes (in deg) from *data*.
- **lat_column** (*str*) – Key used to access the Galactic latitudes (in deg) from *data*.
- **lat_column** (*str*) – Key used to access the Galactic latitudes (in deg) from *data*.
- **x_column, y_column, z_column** (*str*) – Keys used to access the coordinates (in kpc) from *data*.
- **frequency** (*astropy.units.Quantity*) – Frequency of the measurement (if relevant)
- **tag** (*str*) –

  Extra information associated with the observable.

  – 'I' - total intensity (in unit K-cmb)

- 'Q' - Stokes Q (in unit K-cmb, IAU convention)
- 'U' - Stokes U (in unit K-cmb, IAU convention)
- 'PI' - polarisation intensity (in unit K-cmb)
- 'PA' - polarisation angle (in unit rad, IAU convention)

class imagine.observables.dataset.**HEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.Dataset*

Base class for HEALPix datasets, which are input as a simple 1D-array without explicit coordinate information

class imagine.observables.dataset.**FaradayDepthHEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a Faraday depth map into an IMAGINE-compatible dataset

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'fd'

class imagine.observables.dataset.**SynchrotronHEALPixDataset**(*data*, *frequency*, *type*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a synchrotron emission map into an IMAGINE-compatible dataset. This can be Stokes parameters, total and polarised intensity, and polarisation angle.

The parameter *type* and the units of the map in *data* must follow:

- 'I' - total intensity (in unit K-cmb)
- 'Q' - Stokes Q (in unit K-cmb, IAU convention)
- 'U' - Stokes U (in unit K-cmb, IAU convention)
- 'PI' - polarisation intensity (in unit K-cmb)
- 'PA' - polarisation angle (in unit rad, IAU convention)

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **frequency** (*astropy.units.Quantity*) – Frequency of the measurement (if relevant)
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches data.size
- **type** (*str*) – The type of map being supplied in *data*.

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'sync'

class imagine.observables.dataset.**DispersionMeasureHEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a dispersion measure map into an IMAGINE-compatible dataset

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'dm'

imagine.observables.observable module

In the Observable class we define three data types, i.e., - 'measured' - 'simulated' - 'covariance' where 'measured' indicates the hosted data is from measurements, which has a single realization, 'simulated' indicates the hosted data is from simulations, which has multiple realizations, 'covariance' indicates the hosted data is a covariance matrix, which has a single realization but by default should not be stored/read/written by a single computing node.

'measred' data puts its identical copies on all computing nodes, which means each node has a full storage of 'measured' data.

'simulated' data puts different realizations on different nodes, which means each node has part of the full realizations, but at least a full version of one single realization.

'covariance' data distributes itself into all computing nodes, which means to have a full set of 'covariance' data, we have to collect pieces from all the computing nodes.

class imagine.observables.observable.**Observable**(*data=None*, *dtype=None*, *coords=None*)

Bases: object

Observable class is designed for storing/manipulating distributed information. For the testing suits, please turn to "imagine/tests/observable_tests.py".

Parameters

- **data** (*numpy.ndarray*) – distributed/copied data
- **dtype** (*str*) – Data type, must be either: 'measured', 'simulated' or 'covariance'

append(*new_data*)
appending new data happens only to SIMULATED dtype the new data to be appended should also be distributed which makes the appending operation naturally in parallel

rewrite flag will be switched off once rewriten has been performed

data
Data stored in the LOCAL processor (*numpy.ndarray*, read-only).

dtype
'measured', 'simulated' or 'covariance'

Type Data type, can be either

ensemble_mean

global_data
Data gathered from ALL processors (*numpy.ndarray*, read-only). Note that only master node hosts the global data, while slave nodes hosts None.

rw_flag
Rewriting flag, if true, append method will perform rewriting

shape
Shape of the GLOBAL array, i.e. considering all processors (*numpy.ndarray*, read-only).

size
Local data size (*int*, read-only) this size means the dimension of input data not the sample size of realizations

imagine.observables.observable_dict module

For convenience we define dictionary of Observable objects as ObservableDict from which one can define define the classes Measurements, Covariances, Simulations and Masks, which can be used to store:

- measured data sets
- measured/simulated covariances
- simulated ensemble sets
- mask "maps" (but actually mask lists)

Conventions for observables entries

- **Faraday depth:** *('fd','nan',str(size/Nside),'nan')*
- **Dispersion measure**: *('dm','nan',str(pix),'nan')*
- **Synchrotron emission:** ***('sync',str(freq),str(pix),X)***

  where X stands for:

  - 'I' - total intensity (in unit K-cmb)
  - 'Q' - Stokes Q (in unit K-cmb, IAU convention)
  - 'U' - Stokes U (in unit K-cmb, IAU convention)
  - 'PI' - polarisation intensity (in unit K-cmb)
  - 'PA' - polarisation angle (in unit rad, IAU convention)

Remarks:

- ***str(freq)*, polarisation-related-flag are redundant for Faraday depth** and dispersion measure so we put 'nan' instead
- ***str(pix/nside)* stores either Healpix Nside, or just number of** pixels/points we do this for flexibility, in case users have non-HEALPix based in/output

Remarks on the observable tags str(freq), polarisation-related-flag are redundant for Faraday depth and dispersion measure so we put 'nan' instead

str(pix/nside) stores either Healpix Nisde, or just number of pixels/points we do this for flexibility, in case users have non-HEALPix-map-like in/output

Masking convention masked erea associated with pixel value 0, unmasked area with pixel value 1

Masking method mask only applies to observables/covariances observable after masking will be re-recorded as plain data type

---

Note: Distribution with MPI

- all data are either distributed or copied, where "copied" means each node stores the identical copy which is convenient for hosting measured data and mask map
- Covariances has Field object with global shape "around" (data_size//mpisize, data_size) "around" means to distribute matrix correctly as described in "imagine/tools/mpi_helper.py"

---

class imagine.observables.observable_dict.**ObservableDict**
Bases: *imagine.tools.class_tools.BaseClass*

Base class from which *imagine.observables.observable_dict.Measurements*, *imagine.observables.observable_dict.Covariances*, *imagine.observables.observable_dict.Simulations* and *imagine.observables.observable_dict.Masks* are derived.

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)
Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside/size),str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

keys()

archive

class imagine.observables.observable_dict.**Masks**
Bases: *imagine.observables.observable_dict.ObservableDict*

Stores HEALPix mask maps

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)
Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside/"tab"),str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(**args*, ***kwargs*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.observable_dict.**Measurements**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores observational data sets

See *imagine.observables.observable_dict* module documentation for further details.

append(*name=None*, *new_data=None*, *plain=False*, *dataset=None*)

Adds/updates name and data

Parameters

- **dataset** (*imagine.observables.dataset.Dataset*) – The IMAGINE dataset already adjusts the format of the data and sets the adequate key. If *dataset* is present, all other arguments will be ignored.
- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** (*numpy.ndarray or imagine.observables.observable.Observable*) – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured/tabular data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict=None*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.observable_dict.**Simulations**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores simulated ensemble sets

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)

Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.observable_dict.**Covariances**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores observational covariances

See *imagine.observables.observable_dict* module documentation for further details.

append(*name=None*, *new_data=None*, *plain=False*, *dataset=None*)

Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **data** – distributed/copied ndarray/Observable
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

Module contents

class imagine.observables.**Dataset**

Bases: *imagine.tools.class_tools.BaseClass*

Base class for writing helpers to convert arbitrary observational datasets into IMAGINE's standardized format

REQ_ATTRS = ['NAME']

cov

data

Array in the shape (1, N)

frequency

key

Key used in the Observables_dictionary

name

class imagine.observables.**TabularDataset**(*data*, *name*, *data_column=None*, *units=None*, *coordinates_type='galactic'*, *lon_column='lon'*, *lat_column='lat'*, *x_column='x'*, *y_column='y'*, *z_column='z'*, *error_column=None*, *frequency='nan'*, *tag='nan'*)

Bases: *imagine.observables.dataset.Dataset*

Base class for tabular datasets, where the data is input in either in a Python dictionary-like object (`dict`, `astropy.table.Table`, `pandas.DataFrame`, etc).

Parameters

- **data** (*dict-like*) – Can be a `dict`, `astropy.table.Table`, `pandas.DataFrame`, or similar object containing the data.
- **name** (*str*) – Standard name of this type of observable. E.g. 'fd', 'sync', 'dm'.

- **data_column** (*str*) – Key used to access the relevant dataset from the provided data (i.e. data[data_column]).
- **units** (*astropy.units.Unit or str*) – Units used for the data.
- **coordinates_type** (*str*) – Type of coordinates used. Can be 'galactic' or 'cartesian'.
- **lon_column** (*str*) – Key used to access the Galactic longitudes (in deg) from *data*.
- **lat_column** (*str*) – Key used to access the Galactic latitudes (in deg) from *data*.
- **lat_column** (*str*) – Key used to access the Galactic latitudes (in deg) from *data*.
- **x_column, y_column, z_column** (*str*) – Keys used to access the coordinates (in kpc) from *data*.
- **frequency** (*astropy.units.Quantity*) – Frequency of the measurement (if relevant)
- **tag** (*str*) –

  Extra information associated with the observable.

  - 'I' - total intensity (in unit K-cmb)
  - 'Q' - Stokes Q (in unit K-cmb, IAU convention)
  - 'U' - Stokes U (in unit K-cmb, IAU convention)
  - 'PI' - polarisation intensity (in unit K-cmb)
  - 'PA' - polarisation angle (in unit rad, IAU convention)

class imagine.observables.**HEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.Dataset*

Base class for HEALPix datasets, which are input as a simple 1D-array without explicit coordinate information

class imagine.observables.**FaradayDepthHEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a Faraday depth map into an IMAGINE-compatible dataset

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'fd'

class imagine.observables.**SynchrotronHEALPixDataset**(*data*, *frequency*, *type*, *error=None*, *cov=None*, *Nside=None*)

Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a synchrotron emission map into an IMAGINE-compatible dataset. This can be Stokes parameters, total and polarised intensity, and polarisation angle.

The parameter *type* and the units of the map in *data* must follow:

- 'I' - total intensity (in unit K-cmb)
- 'Q' - Stokes Q (in unit K-cmb, IAU convention)
- 'U' - Stokes U (in unit K-cmb, IAU convention)
- 'PI' - polarisation intensity (in unit K-cmb)
- 'PA' - polarisation angle (in unit rad, IAU convention)

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **frequency** (*astropy.units.Quantity*) – Frequency of the measurement (if relevant)
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches data.size
- **type** (*str*) – The type of map being supplied in *data*.

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'sync'

class imagine.observables.**DispersionMeasureHEALPixDataset**(*data*, *error=None*, *cov=None*, *Nside=None*)
Bases: *imagine.observables.dataset.HEALPixDataset*

Stores a dispersion measure map into an IMAGINE-compatible dataset

Parameters

- **data** (*numpy.ndarray*) – 1D-array containing the HEALPix map
- **Nside** (*int, optional*) – For extra internal consistency checking. If *Nside* is present, it will be checked whether $12 \times N_{side}^2$ matches

data
Data in ObservablesDict-compatible shape

key
Standard key associated with this observable

NAME = 'dm'

class imagine.observables.**Observable**(*data=None*, *dtype=None*, *coords=None*)
Bases: object

Observable class is designed for storing/manipulating distributed information. For the testing suits, please turn to "imagine/tests/observable_tests.py".

Parameters

- **data** (*numpy.ndarray*) – distributed/copied data
- **dtype** (*str*) – Data type, must be either: 'measured', 'simulated' or 'covariance'

append(*new_data*)
appending new data happens only to SIMULATED dtype the new data to be appended should also be distributed which makes the appending operation naturally in parallel

rewrite flag will be switched off once rewriten has been performed

data
Data stored in the LOCAL processor (*numpy.ndarray*, read-only).

dtype
'measured', 'simulated' or 'covariance'

Type Data type, can be either

ensemble_mean

global_data
Data gathered from ALL processors (*numpy.ndarray*, read-only). Note that only master node hosts the global data, while slave nodes hosts None.

rw_flag
Rewriting flag, if true, append method will perform rewriting

shape
Shape of the GLOBAL array, i.e. considering all processors (*numpy.ndarray*, read-only).

size
Local data size (*int*, read-only) this size means the dimension of input data not the sample size of realizations

class imagine.observables.**ObservableDict**
Bases: *imagine.tools.class_tools.BaseClass*

Base class from which *imagine.observables.observable_dict.Measurements*, *imagine.observables.observable_dict.Covariances*, *imagine.observables.observable_dict.Simulations* and *imagine.observables.observable_dict.Masks* are derived.

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)
Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside/size),str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

keys()

archive

class imagine.observables.**Masks**
Bases: *imagine.observables.observable_dict.ObservableDict*

Stores HEALPix mask maps

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)
Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside/"tab"),str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(**args*, ***kwargs*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.**Measurements**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores observational data sets

See *imagine.observables.observable_dict* module documentation for further details.

append(*name=None*, *new_data=None*, *plain=False*, *dataset=None*)

Adds/updates name and data

Parameters

- **dataset** (*imagine.observables.dataset.Dataset*) – The IMAGINE dataset already adjusts the format of the data and sets the adequate key. If *dataset* is present, all other arguments will be ignored.
- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** (*numpy.ndarray or imagine.observables.observable.Observable*) – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured/tabular data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict=None*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.**Simulations**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores simulated ensemble sets

See *imagine.observables.observable_dict* module documentation for further details.

append(*name*, *new_data*, *plain=False*)

Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **new_data** – distributed/copied `numpy.ndarray` or *Observable*
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

class imagine.observables.**Covariances**

Bases: *imagine.observables.observable_dict.ObservableDict*

Stores observational covariances

See *imagine.observables.observable_dict* module documentation for further details.

append(*name=None*, *new_data=None*, *plain=False*, *dataset=None*)

Adds/updates name and data

Parameters

- **name** (*str tuple*) – Should follow the convention: `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`. If data is independent from frequency, set 'nan'. *ext* can be 'I','Q','U','PI','PA', 'nan' or other customized tags.
- **data** – distributed/copied ndarray/Observable
- **plain** (*bool*) – If True, means unstructured data. If False (default case), means HEALPix-like sky map.

apply_mask(*mask_dict*)

Parameters **mask_dict** (*imagine.observables.observable_dict.Masks*) – Masks object

14.1.4 imagine.pipelines package

Submodules

imagine.pipelines.dynesty_pipeline module

class imagine.pipelines.dynesty_pipeline.**DynestyPipeline**(***, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: *imagine.pipelines.pipeline.Pipeline*

Bayesian analysis pipeline with dynesty

This pipeline may use `DynamicNestedSampler` if the sampling parameter 'dynamic' is set to *True* or `NestedSampler` if 'dynamic' is False (default).

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **dynamic** (*bool*) – If *True*, use `dynesty.DynamicNestedSampler` otherwise uses `dynesty.NestedSampler`
- **dlogz** (*float*) – Iteration will stop, in the *dynamic==False* case, when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is *ln(z + z_est) - ln(z) < dlogz*, where *z* is the current evidence from all saved samples and *z_est* is the estimated contribution from the remaining volume. If *add_live* is *True*, the default is *1e-3 * (nlive - 1) + 0.01*. Otherwise, the default is *0.01*.

- **dlogz_init** (*float*) – The baseline run will stop, in the *dynamic==True* case, when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is *ln(z + z_est) - ln(z) < dlogz*, where *z* is the current evidence from all saved samples and *z_est* is the estimated contribution from the remaining volume. If *add_live* is *True*, the default is *1e-3 * (nlive - 1) + 0.01*. Otherwise, the default is *0.01*.
- **nlive** (*int*) – If *dynamic* is *False*, this sets the number of live points used. Default is 400.
- **nlive_init** (*int*) – If *dynamic* is *True*, this sets the number of live points used during the initial ("baseline") nested sampling run. Default is 400.
- **nlive_batch** (*int*) – If *dynamic* is *True*, this sets the number of live points used when adding additional samples from a nested sampling run within each batch. Default is *400*.
- **logl_max** (*float*) – Iteration will stop when the sampled ln(likelihood) exceeds the threshold set by *logl_max*. Default is no bound (*np.inf*).
- **logl_max_init** (*float*) – The baseline run will stop, in the *dynamic==True* case, when the sampled ln(likelihood) exceeds this threshold. Default is no bound (*np.inf*).
- **maxiter** (*int*) – Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is (no limit).
- **maxiter_init** (*int*) – If *dynamic* is *True*, this sets the maximum number of iterations for the initial baseline nested sampling run. Iteration may stop earlier if the termination condition is reached. Default is sys.maxsize (no limit).
- **maxiter_batch** (*int*) – If *dynamic* is *True*, this sets the maximum number of iterations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxcall** (*int*) – Maximum number of likelihood evaluations (without considering the initial points, i.e. maxcall_effective = maxcall + nlive). Iteration may stop earlier if termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxcall_init** (*int*) – If *dynamic* is *True*, maximum number of likelihood evaluations in the baseline run.
- **maxcall_batch** (*int*) – If *dynamic* is *True*, maximum number of likelihood evaluations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxbatch** (*int*) – If *dynamic* is *True*, maximum number of batches allowed. Default is *sys.maxsize* (no limit).
- **use_stop** (*bool, optional*) – Whether to evaluate our stopping function after each batch. Disabling this can improve performance if other stopping criteria such as `maxcall` are already specified. Default is *True*.
- **n_effective** (*int*) – Minimum number of effective posterior samples. If the estimated "effective sample size" (ESS) exceeds this number, sampling will terminate. Default is no ESS (*np.inf*).
- **n_effective_init** (*int*) – Minimum number of effective posterior samples during the baseline run. If the estimated "effective sample size" (ESS) exceeds this number, sampling will terminate. Default is no ESS (*np.inf*).
- **add_live** (*bool*) – Whether or not to add the remaining set of live points to the list of samples at the end of each run. Default is *True*.

- **print_progress** (*bool*) – Whether or not to output a simple summary of the current run that updates with each iteration. Default is *True*.
- **print_func** (*function*) – A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.
- **save_bounds** (*bool*) –

  Whether or not to save past bounding distributions used to bound the live points internally. Default is *True*.
- **bound** ({*'none'*, *'single'*, *'multi'*, *'balls'*, *'cubes'*}) – Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points. Choices are no bound (*'none'*), a single bounding ellipsoid (*'single'*), multiple bounding ellipsoids (*'multi'*), balls centered on each live point (*'balls'*), and cubes centered on each live point (*'cubes'*). Default is *'multi'*.
- **sample** ({*'auto'*, *'unif'*, *'rwalk'*, *'rstagger'*,) – *'slice'*, *'rslice'*} Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds. Unique methods available are: uniform sampling within the bounds(*'unif'*), random walks with fixed proposals (*'rwalk'*), random walks with variable ("staggering") proposals (*'rstagger'*), multivariate slice sampling along preferred orientations (*'slice'*), and "random" slice sampling along all orientations (*'rslice'*). *'auto'* selects the sampling method based on the dimensionality of the problem (from *ndim*). When *ndim < 10*, this defaults to *'unif'*. When *10 <= ndim <= 20*, this defaults to *'rwalk'*. When *ndim > 20*, this defaults to *'slice'*. *'rstagger'* and *'rslice'* are provided as alternatives for *'rwalk'* and *'slice'*, respectively. Default is *'auto'*. Note that Dynesty's 'hslice' option is not supported within IMAGINE.
- **update_interval** (*int or float*) – If an integer is passed, only update the proposal distribution every *update_interval*-th likelihood call. If a float is passed, update the proposal after every *round(update_interval * nlive)*-th likelihood call. Larger update intervals larger can be more efficient when the likelihood function is quick to evaluate. Default behavior is to target a roughly constant change in prior volume, with *1.5* for *'unif'*, *0.15 * walks* for *'rwalk'* and *'rstagger'*, *0.9 * ndim * slices* for *'slice'*, *2.0 * slices* for *'rslice'*, and *25.0 * slices* for *'hslice'*.
- **enlarge** (*float*) – Enlarge the volumes of the specified bounding object(s) by this fraction. The preferred method is to determine this organically using bootstrapping. If *bootstrap > 0*, this defaults to *1.0*. If *bootstrap = 0*, this instead defaults to *1.25*.
- **bootstrap** (*int*) – Compute this many bootstrapped realizations of the bounding objects. Use the maximum distance found to the set of points left out during each iteration to enlarge the resulting volumes. Can lead to unstable bounding ellipsoids. Default is *0* (no bootstrap).
- **vol_dec** (*float*) – For the *'multi'* bounding option, the required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is *0.5*.
- **vol_check** (*float*) – For the *'multi'* bounding option, the factor used when checking if the volume of the original bounding ellipsoid is large enough to warrant *> 2* splits via *ell.vol > vol_check * nlive * pointvol*. Default is *2.0*.
- **walks** (*int*) – For the *'rwalk'* sampling option, the minimum number of steps (minimum 2) before proposing a new live point. Default is *25*.
- **facc** (*float*) – The target acceptance fraction for the *'rwalk'* sampling option. Default is *0.5*. Bounded to be between *[1. / walks, 1.]*.
- **slices** (*int*) – For the *'slice'* and *'rslice'* sampling options, the number of times to execute a "slice update" before proposing a new live point. Default is *5*. Note that *'slice'* cycles through **all dimensions** when executing a "slice update".

Note: Instances of this class are callable. Look at the *DynestyPipeline.call()* for details.

call(***kwargs*)

Runs the IMAGINE pipeline using the Dynesty sampler

Returns **results** – Dynesty sampling results

Return type dict

imagine.pipelines.multinest_pipeline module

class imagine.pipelines.multinest_pipeline.**MultinestPipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: *imagine.pipelines.pipeline.Pipeline*

Bayesian analysis pipeline with pyMultinest

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **resume** (*bool*) – If False the Pipeline the sampling starts from the beginning, overwriting any previous work in the *chains_directory*. Otherwise, tries to resume a previous run.
- **n_live_points** (*int*) – Number of live points to be used.
- **evidence_tolerance** (*float*) – A value of 0.5 should give good enough accuracy.
- **max_iter** (*int*) – Maximum number of iterations. *0* (default) is unlimited (i.e. only stops after convergence).
- **log_zero** (*float*) – Points with loglike < logZero will be ignored by MultiNest
- **importance_nested_sampling** (*bool*) – If *True* (default), Multinest will use Importance Nested Sampling (see arXiv:1306.2144)
- **sampling_efficiency** (*float*) – Efficieny of the sampling. *0.8* (default) and *0.3* are recommended values for parameter estimation & evidence evaluation respectively.
- **multimodal** (*bool*) – If *True*, MultiNest will attempt to separate out the modes using a clustering algorithm.
- **mode_tolerance** (*float*) – MultiNest can find multiple modes and specify which samples belong to which mode. It might be desirable to have separate samples and mode statistics for modes with local log-evidence value greater than a particular value in which case *mode_tolerance* should be set to that value. If there isn't any particularly interesting *mode_tolerance* value, then it should be set to a very negative number (e.g. -1e90, default).
- **null_log_evidence** (*float*) – If *multimodal* is *True*, MultiNest can find multiple modes and also specify which samples belong to which mode. It might be desirable to have separate samples and mode statistics for modes with local log-evidence value greater than a particular value in which case nullZ should be set to that value. If there isn't any particulrly interesting nullZ value, then nullZ should be set to a very large negative number (e.g. -1.d90).

- **n_clustering_params** (*int*) – Mode separation is done through a clustering algorithm. Mode separation can be done on all the parameters (in which case nCdims should be set to ndims) & it can also be done on a subset of parameters (in which case nCdims < ndims) which might be advantageous as clustering is less accurate as the dimensionality increases. If nCdims < ndims then mode separation is done on the first nCdims parameters.
- **max_modes** (*int*) – Maximum number of modes (if *multimodal* is *True*).

Note: Instances of this class are callable. Look at the *MultinestPipeline.call()* for details.

---

call(***kwargs*)
Runs the IMAGINE pipeline using the MultiNest sampler

Returns **results** – pyMultinest sampling results in a dictionary containing the keys: logZ (the log-evidence), logZerror (the error in log-evidence) and samples (equal weighted posterior)

Return type dict

SUPPORTS_MPI = True

imagine.pipelines.pipeline module

class imagine.pipelines.pipeline.**Pipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)
Bases: *imagine.tools.class_tools.BaseClass*

Base class used for for initialing Bayesian analysis pipeline

dynesty_parameter_dict
extra parameters for controlling Dynesty i.e., 'nlive', 'bound', 'sample'

Type dict

sample_callback
not implemented yet

Type bool

likelihood_rescaler
Rescale log-likelihood value

Type double

random_type
If set to 'fixed', the exact same set of ensemble seeds will be used for the evaluation of all fields, generated using the *master_seed*. If set to 'controllable', each individual field will get their own set of ensemble fields, but multiple runs will lead to the same results, as they are based on the same *master_seed*. If set to 'free', every time the pipeline is run, the *master_seed* is reset to a different value, and the ensemble seeds for each individual field are drawn based on this.

Type str

master_seed
Master seed used by the random number generators

Type int

Parameters

- **simulator** (*imagine.simulators.simulator.Simulator*) – Simulator object

- **factory_list** (*list*) – List or tuple of field factory objects
- **likelihood** (*imagine.likelihoods.likelihood.Likelihood*) – Likelihood object
- **prior** (*imagine.priors.prior.Prior*) – Prior object
- **ensemble_size** (*int*) – Number of observable realizations PER COMPUTING NODE to be generated in simulator
- **chains_directory** (*str*) – Path of the directory where the chains should be saved

__call__(**args*, ***kwargs*)
Call self as a function.

call(***kwargs*)

posterior_report(*sdigits=2*)
Displays the best fit values and 1-sigma errors for each active parameter.

If running on a jupyter-notebook, a nice LaTeX display is used.

Parameters **sdigits** (*int*) – The number of significant digits to be used

prior_pdf(*cube*)
Probability distribution associated with the all parameters being used by the multiple Field Factories

Parameters **cube** (*array*) – Each row of the array corresponds to a different parameter in the sampling.

Returns The modified cube

Return type cube_rtn

prior_transform(*cube*)
Prior transform cube (i.e. MultiNest style prior).

Takes a cube containing a uniform sampling of values and maps then onto a distribution compatible with the priors specified in the Field Factories.

Parameters **cube** (*array*) – Each row of the array corresponds to a different parameter in the sampling. Warning: the function will modify *cube* inplace.

Returns The modified cube

Return type cube

tidy_up()
Resets internal state before a new run

active_parameters
List of all the active parameters

active_ranges
Ranges of all active parameters

chains_directory
Directory where the chains are stored (NB details of what is stored are sampler-dependent)

distribute_ensemble
If True, whenever the sampler requires a likelihood evaluation, the ensemble of stochastic fields realizations is distributed among all the nodes.

Otherwise, each likelihood evaluations will go through the whole ensemble size on a single node. See *Parallelisation* for details.

ensemble_size

factory_list
List of the `Field Factories` currently being used.

Updating the factory list automatically extracts active_parameters, parameter ranges and priors from each field factory.

likelihood
The *Likelihood* object used by the pipeline

log_evidence
Natural logarithm of the *marginal likelihood* or *Bayesian model evidence*, $\ln \mathcal{Z}$, where

$$\mathcal{Z} = P(d|m) = \int_{\Omega_\theta} P(d|\theta, m) P(\theta|m) \mathrm{d}\theta.$$

Note: Available only after the pipeline is run.

log_evidence_err
Error estimate in the natural logarithm of the *Bayesian model evidence*. Available once the pipeline is run.

Note: Available only after the pipeline is run.

posterior_summary
A dictionary containing a summary of posterior statistics for each of the active parameters. These are: 'median', 'errlo' (15.87th percentile), 'errup' (84.13th percentile), 'mean' and 'stdev'.

priors
Dictionary containing priors for all active parameters

sampler_supports_mpi

samples
An `astropy.table.QTable` object containing parameter values of the samples produced in the run.

samples_scaled
An `astropy.table.QTable` object containing parameter values of the samples produced in the run, scaled to the interval [0,1].

sampling_controllers
Settings used by the sampler (e.g. *'dlogz'*). See the documentation of each specific pipeline subclass for details.

After the pipeline runs, this property is updated to reflect the actual final choice of sampling controllers (including default values).

simulator
The *Simulator* object used by the pipeline

imagine.pipelines.ultranest_pipeline module

class imagine.pipelines.ultranest_pipeline.**UltranestPipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: *imagine.pipelines.pipeline.Pipeline*

Bayesian analysis pipeline with UltraNest

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **resume** (*bool*) – If False the Pipeline the sampling starts from the beginning, erasing any previous work in the *chains_directory*. Otherwise, tries to resume a previous run.
- **dlogz** (*float*) – Target evidence uncertainty. This is the std between bootstrapped logz integrators.
- **dKL** (*float*) – Target posterior uncertainty. This is the Kullback-Leibler divergence in nat between bootstrapped integrators.
- **frac_remain** (*float*) – Integrate until this fraction of the integral is left in the remainder. Set to a low number (1e-2 ... 1e-5) to make sure peaks are discovered. Set to a higher number (0.5) if you know the posterior is simple.
- **Lepsilon** (*float*) – Terminate when live point likelihoods are all the same, within Lepsilon tolerance. Increase this when your likelihood function is inaccurate, to avoid unnecessary search.
- **min_ess** (*int*) – Target number of effective posterior samples.
- **max_iters** (*int*) – maximum number of integration iterations.
- **max_ncalls** (*int*) – stop after this many likelihood evaluations.
- **max_num_improvement_loops** (*int*) – run() tries to assess iteratively where more samples are needed. This number limits the number of improvement loops.
- **min_num_live_points** (*int*) – minimum number of live points throughout the run
- **cluster_num_live_points** (*int*) – require at least this many live points per detected cluster
- **num_test_samples** (*int*) – test transform and likelihood with this number of random points for errors first. Useful to catch bugs.
- **draw_multiple** (*bool*) – draw more points if efficiency goes down. If set to False, few points are sampled at once.
- **num_bootstraps** (*int*) – number of logZ estimators and MLFriends region bootstrap rounds.
- **update_interval_iter_fraction** (*float*) – Update region after (update_interval_iter_fraction*nlive) iterations.

Note: Instances of this class are callable. Look at the `UltranestPipeline.call()` for details.

`call` (***kwargs*)

Runs the IMAGINE pipeline using the UltraNest `ReactiveNestedSampler`.

Any keyword argument provided is used to update the *sampling_controllers*.

Returns **results** – UltraNest sampling results in a dictionary containing the keys: logZ (the log-evidence), logZerror (the error in log-evidence) and samples (equal weighted posterior)

Return type dict

Notes

See base class for other attributes/properties and methods

SUPPORTS_MPI = True

Module contents

class imagine.pipelines.**DynestyPipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: *imagine.pipelines.pipeline.Pipeline*

Bayesian analysis pipeline with dynesty

This pipeline may use `DynamicNestedSampler` if the sampling parameter 'dynamic' is set to *True* or `NestedSampler` if 'dynamic' is False (default).

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **dynamic** (*bool*) – If *True*, use `dynesty.DynamicNestedSampler` otherwise uses `dynesty.NestedSampler`
- **dlogz** (*float*) – Iteration will stop, in the *dynamic==False* case, when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is *ln(z + z_est) - ln(z) < dlogz*, where *z* is the current evidence from all saved samples and *z_est* is the estimated contribution from the remaining volume. If *add_live* is *True*, the default is *1e-3 * (nlive - 1) + 0.01*. Otherwise, the default is *0.01*.
- **dlogz_init** (*float*) – The baseline run will stop, in the *dynamic==True* case, when the estimated contribution of the remaining prior volume to the total evidence falls below this threshold. Explicitly, the stopping criterion is *ln(z + z_est) - ln(z) < dlogz*, where *z* is the current evidence from all saved samples and *z_est* is the estimated contribution from the remaining volume. If *add_live* is *True*, the default is *1e-3 * (nlive - 1) + 0.01*. Otherwise, the default is *0.01*.
- **nlive** (*int*) – If *dynamic* is *False*, this sets the number of live points used. Default is 400.
- **nlive_init** (*int*) – If *dynamic* is *True*, this sets the number of live points used during the initial ("baseline") nested sampling run. Default is 400.
- **nlive_batch** (*int*) – If *dynamic* is *True*, this sets the number of live points used when adding additional samples from a nested sampling run within each batch. Default is *400*.
- **logl_max** (*float*) – Iteration will stop when the sampled ln(likelihood) exceeds the threshold set by *logl_max*. Default is no bound (*np.inf*).
- **logl_max_init** (*float*) – The baseline run will stop, in the *dynamic==True* case, when the sampled ln(likelihood) exceeds this threshold. Default is no bound (*np.inf*).
- **maxiter** (*int*) – Maximum number of iterations. Iteration may stop earlier if the termination condition is reached. Default is (no limit).
- **maxiter_init** (*int*) – If *dynamic* is *True*, this sets the maximum number of iterations for the initial baseline nested sampling run. Iteration may stop earlier if the termination condition is reached. Default is sys.maxsize (no limit).

- **maxiter_batch** (*int*) – If *dynamic* is *True*, this sets the maximum number of iterations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxcall** (*int*) – Maximum number of likelihood evaluations (without considering the initial points, i.e. maxcall_effective = maxcall + nlive). Iteration may stop earlier if termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxcall_init** (*int*) – If *dynamic* is *True*, maximum number of likelihood evaluations in the baseline run.
- **maxcall_batch** (*int*) – If *dynamic* is *True*, maximum number of likelihood evaluations for the nested sampling run within each batch. Iteration may stop earlier if the termination condition is reached. Default is *sys.maxsize* (no limit).
- **maxbatch** (*int*) – If *dynamic* is *True*, maximum number of batches allowed. Default is *sys.maxsize* (no limit).
- **use_stop** (*bool, optional*) – Whether to evaluate our stopping function after each batch. Disabling this can improve performance if other stopping criteria such as `maxcall` are already specified. Default is *True*.
- **n_effective** (*int*) – Minimum number of effective posterior samples. If the estimated "effective sample size" (ESS) exceeds this number, sampling will terminate. Default is no ESS (*np.inf*).
- **n_effective_init** (*int*) – Minimum number of effective posterior samples during the baseline run. If the estimated "effective sample size" (ESS) exceeds this number, sampling will terminate. Default is no ESS (*np.inf*).
- **add_live** (*bool*) – Whether or not to add the remaining set of live points to the list of samples at the end of each run. Default is *True*.
- **print_progress** (*bool*) – Whether or not to output a simple summary of the current run that updates with each iteration. Default is *True*.
- **print_func** (*function*) – A function that prints out the current state of the sampler. If not provided, the default `results.print_fn()` is used.
- **save_bounds** (*bool*) –

  Whether or not to save past bounding distributions used to bound the live points internally. Default is *True*.
- **bound** ({*'none'*, *'single'*, *'multi'*, *'balls'*, *'cubes'*}) – Method used to approximately bound the prior using the current set of live points. Conditions the sampling methods used to propose new live points. Choices are no bound (*'none'*), a single bounding ellipsoid (*'single'*), multiple bounding ellipsoids (*'multi'*), balls centered on each live point (*'balls'*), and cubes centered on each live point (*'cubes'*). Default is *'multi'*.
- **sample** ({*'auto'*, *'unif'*, *'rwalk'*, *'rstagger'*,) – *'slice'*, *'rslice'*} Method used to sample uniformly within the likelihood constraint, conditioned on the provided bounds. Unique methods available are: uniform sampling within the bounds(*'unif'*), random walks with fixed proposals (*'rwalk'*), random walks with variable ("staggering") proposals (*'rstagger'*), multivariate slice sampling along preferred orientations (*'slice'*), and "random" slice sampling along all orientations (*'rslice'*). *'auto'* selects the sampling method based on the dimensionality of the problem (from *ndim*). When *ndim* < *10*, this defaults to *'unif'*. When *10* <= *ndim* <= *20*, this defaults to *'rwalk'*. When *ndim* > *20*, this defaults to *'slice'*. *'rstagger'* and *'rslice'* are provided as alternatives for *'rwalk'* and *'slice'*, respectively. Default is *'auto'*. Note that Dynesty's 'hslice' option is not supported within IMAGINE.

- **update_interval** (*int or float*) – If an integer is passed, only update the proposal distribution every *update_interval*-th likelihood call. If a float is passed, update the proposal after every *round(update_interval * nlive)*-th likelihood call. Larger update intervals larger can be more efficient when the likelihood function is quick to evaluate. Default behavior is to target a roughly constant change in prior volume, with *1.5* for *'unif'*, *0.15 * walks* for *'rwalk'* and *'rstagger'*, *0.9 * ndim * slices* for *'slice'*, *2.0 * slices* for *'rslice'*, and *25.0 * slices* for *'hslice'*.
- **enlarge** (*float*) – Enlarge the volumes of the specified bounding object(s) by this fraction. The preferred method is to determine this organically using bootstrapping. If *bootstrap > 0*, this defaults to *1.0*. If *bootstrap = 0*, this instead defaults to *1.25*.
- **bootstrap** (*int*) – Compute this many bootstrapped realizations of the bounding objects. Use the maximum distance found to the set of points left out during each iteration to enlarge the resulting volumes. Can lead to unstable bounding ellipsoids. Default is *0* (no bootstrap).
- **vol_dec** (*float*) – For the *'multi'* bounding option, the required fractional reduction in volume after splitting an ellipsoid in order to to accept the split. Default is *0.5*.
- **vol_check** (*float*) – For the *'multi'* bounding option, the factor used when checking if the volume of the original bounding ellipsoid is large enough to warrant *> 2* splits via *ell.vol > vol_check * nlive * pointvol*. Default is *2.0*.
- **walks** (*int*) – For the *'rwalk'* sampling option, the minimum number of steps (minimum 2) before proposing a new live point. Default is *25*.
- **facc** (*float*) – The target acceptance fraction for the *'rwalk'* sampling option. Default is *0.5*. Bounded to be between *[1. / walks, 1.]*.
- **slices** (*int*) – For the *'slice'* and *'rslice'* sampling options, the number of times to execute a "slice update" before proposing a new live point. Default is *5*. Note that *'slice'* cycles through **all dimensions** when executing a "slice update".

Note: Instances of this class are callable. Look at the `DynestyPipeline.call()` for details.

---

call(***kwargs*)

Runs the IMAGINE pipeline using the Dynesty sampler

Returns **results** – Dynesty sampling results

Return type dict

class imagine.pipelines.**MultinestPipeline**(***, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: `imagine.pipelines.pipeline.Pipeline`

Bayesian analysis pipeline with pyMultinest

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **resume** (*bool*) – If False the Pipeline the sampling starts from the beginning, overwriting any previous work in the *chains_directory*. Otherwise, tries to resume a previous run.
- **n_live_points** (*int*) – Number of live points to be used.
- **evidence_tolerance** (*float*) – A value of 0.5 should give good enough accuracy.

- **max_iter** (*int*) – Maximum number of iterations. *0* (default) is unlimited (i.e. only stops after convergence).
- **log_zero** (*float*) – Points with loglike < logZero will be ignored by MultiNest
- **importance_nested_sampling** (*bool*) – If *True* (default), Multinest will use Importance Nested Sampling (see arXiv:1306.2144)
- **sampling_efficiency** (*float*) – Efficieny of the sampling. *0.8* (default) and *0.3* are recommended values for parameter estimation & evidence evaluation respectively.
- **multimodal** (*bool*) – If *True*, MultiNest will attempt to separate out the modes using a clustering algorithm.
- **mode_tolerance** (*float*) – MultiNest can find multiple modes and specify which samples belong to which mode. It might be desirable to have separate samples and mode statistics for modes with local log-evidence value greater than a particular value in which case *mode_tolerance* should be set to that value. If there isn't any particularly interesting *mode_tolerance* value, then it should be set to a very negative number (e.g. -1e90, default).
- **null_log_evidence** (*float*) – If *multimodal* is *True*, MultiNest can find multiple modes and also specify which samples belong to which mode. It might be desirable to have separate samples and mode statistics for modes with local log-evidence value greater than a particular value in which case nullZ should be set to that value. If there isn't any particulrly interesting nullZ value, then nullZ should be set to a very large negative number (e.g. -1.d90).
- **n_clustering_params** (*int*) – Mode separation is done through a clustering algorithm. Mode separation can be done on all the parameters (in which case nCdims should be set to ndims) & it can also be done on a subset of parameters (in which case nCdims < ndims) which might be advantageous as clustering is less accurate as the dimensionality increases. If nCdims < ndims then mode separation is done on the first nCdims parameters.
- **max_modes** (*int*) – Maximum number of modes (if *multimodal* is *True*).

Note: Instances of this class are callable. Look at the `MultinestPipeline.call()` for details.

call(***kwargs*)

Runs the IMAGINE pipeline using the MultiNest sampler

Returns **results** – pyMultinest sampling results in a dictionary containing the keys: logZ (the log-evidence), logZerror (the error in log-evidence) and samples (equal weighted posterior)

Return type dict

SUPPORTS_MPI = True

class imagine.pipelines.**Pipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: `imagine.tools.class_tools.BaseClass`

Base class used for for initialing Bayesian analysis pipeline

dynesty_parameter_dict

extra parameters for controlling Dynesty i.e., 'nlive', 'bound', 'sample'

Type dict

sample_callback

not implemented yet

Type bool

likelihood_rescaler
Rescale log-likelihood value

Type double

random_type
If set to 'fixed', the exact same set of ensemble seeds will be used for the evaluation of all fields, generated using the *master_seed*. If set to 'controllable', each individual field will get their own set of ensemble fields, but multiple runs will lead to the same results, as they are based on the same *master_seed*. If set to 'free', every time the pipeline is run, the *master_seed* is reset to a different value, and the ensemble seeds for each individual field are drawn based on this.

Type str

master_seed
Master seed used by the random number generators

Type int

Parameters

- **simulator** (*imagine.simulators.simulator.Simulator*) – Simulator object
- **factory_list** (*list*) – List or tuple of field factory objects
- **likelihood** (*imagine.likelihoods.likelihood.Likelihood*) – Likelihood object
- **prior** (*imagine.priors.prior.Prior*) – Prior object
- **ensemble_size** (*int*) – Number of observable realizations PER COMPUTING NODE to be generated in simulator
- **chains_directory** (*str*) – Path of the directory where the chains should be saved

__call__(**args*, ***kwargs*)
Call self as a function.

call(***kwargs*)

posterior_report(*sdigits=2*)
Displays the best fit values and 1-sigma errors for each active parameter.

If running on a jupyter-notebook, a nice LaTeX display is used.

Parameters **sdigits** (*int*) – The number of significant digits to be used

prior_pdf(*cube*)
Probability distribution associated with the all parameters being used by the multiple Field Factories

Parameters **cube** (*array*) – Each row of the array corresponds to a different parameter in the sampling.

Returns The modified cube

Return type cube_rtn

prior_transform(*cube*)
Prior transform cube (i.e. MultiNest style prior).

Takes a cube containing a uniform sampling of values and maps then onto a distribution compatible with the priors specified in the Field Factories.

Parameters **cube** (*array*) – Each row of the array corresponds to a different parameter in the sampling. Warning: the function will modify *cube* inplace.

Returns The modified cube

Return type cube

tidy_up()
Resets internal state before a new run

active_parameters
List of all the active parameters

active_ranges
Ranges of all active parameters

chains_directory
Directory where the chains are stored (NB details of what is stored are sampler-dependent)

distribute_ensemble
If True, whenever the sampler requires a likelihood evaluation, the ensemble of stochastic fields realizations is distributed among all the nodes.

Otherwise, each likelihood evaluations will go through the whole ensemble size on a single node. See *Parallelisation* for details.

ensemble_size

factory_list
List of the `Field Factories` currently being used.

Updating the factory list automatically extracts active_parameters, parameter ranges and priors from each field factory.

likelihood
The *Likelihood* object used by the pipeline

log_evidence
Natural logarithm of the *marginal likelihood* or *Bayesian model evidence*, $\ln \mathcal{Z}$, where

$$\mathcal{Z} = P(d|m) = \int_{\Omega_\theta} P(d|\theta, m) P(\theta|m) \mathrm{d}\theta.$$

Note: Available only after the pipeline is run.

log_evidence_err
Error estimate in the natural logarithm of the *Bayesian model evidence*. Available once the pipeline is run.

Note: Available only after the pipeline is run.

posterior_summary
A dictionary containing a summary of posterior statistics for each of the active parameters. These are: 'median', 'errlo' (15.87th percentile), 'errup' (84.13th percentile), 'mean' and 'stdev'.

priors
Dictionary containing priors for all active parameters

sampler_supports_mpi

samples
An `astropy.table.QTable` object containing parameter values of the samples produced in the run.

samples_scaled
An `astropy.table.QTable` object containing parameter values of the samples produced in the run, scaled to the interval [0,1].

sampling_controllers
Settings used by the sampler (e.g. *'dlogz'*). See the documentation of each specific pipeline subclass for details.

After the pipeline runs, this property is updated to reflect the actual final choice of sampling controllers (including default values).

simulator
The *Simulator* object used by the pipeline

class imagine.pipelines.**UltranestPipeline**(*, *simulator*, *factory_list*, *likelihood*, *ensemble_size=1*, *chains_directory=None*)

Bases: *imagine.pipelines.pipeline.Pipeline*

Bayesian analysis pipeline with UltraNest

See base class for initialization details.

The sampler behaviour is controlled using the *sampling_controllers* property. A description of these can be found below.

Other Parameters

- **resume** (*bool*) – If False the Pipeline the sampling starts from the beginning, erasing any previous work in the *chains_directory*. Otherwise, tries to resume a previous run.
- **dlogz** (*float*) – Target evidence uncertainty. This is the std between bootstrapped logz integrators.
- **dKL** (*float*) – Target posterior uncertainty. This is the Kullback-Leibler divergence in nat between bootstrapped integrators.
- **frac_remain** (*float*) – Integrate until this fraction of the integral is left in the remainder. Set to a low number (1e-2 … 1e-5) to make sure peaks are discovered. Set to a higher number (0.5) if you know the posterior is simple.
- **Lepsilon** (*float*) – Terminate when live point likelihoods are all the same, within Lepsilon tolerance. Increase this when your likelihood function is inaccurate, to avoid unnecessary search.
- **min_ess** (*int*) – Target number of effective posterior samples.
- **max_iters** (*int*) – maximum number of integration iterations.
- **max_ncalls** (*int*) – stop after this many likelihood evaluations.
- **max_num_improvement_loops** (*int*) – run() tries to assess iteratively where more samples are needed. This number limits the number of improvement loops.
- **min_num_live_points** (*int*) – minimum number of live points throughout the run
- **cluster_num_live_points** (*int*) – require at least this many live points per detected cluster
- **num_test_samples** (*int*) – test transform and likelihood with this number of random points for errors first. Useful to catch bugs.
- **draw_multiple** (*bool*) – draw more points if efficiency goes down. If set to False, few points are sampled at once.
- **num_bootstraps** (*int*) – number of logZ estimators and MLFriends region bootstrap rounds.

- **update_interval_iter_fraction** (*float*) – Update region after (update_interval_iter_fraction*nlive) iterations.

Note: Instances of this class are callable. Look at the *UltranestPipeline.call()* for details.

call(***kwargs*)

Runs the IMAGINE pipeline using the UltraNest ReactiveNestedSampler.

Any keyword argument provided is used to update the *sampling_controllers*.

Returns **results** – UltraNest sampling results in a dictionary containing the keys: logZ (the log-evidence), logZerror (the error in log-evidence) and samples (equal weighted posterior)

Return type dict

Notes

See base class for other attributes/properties and methods

SUPPORTS_MPI = True

14.1.5 imagine.priors package

Submodules

imagine.priors.basic_priors module

class imagine.priors.basic_priors.**FlatPrior**(*interval=[0, 1]*)

Bases: *imagine.priors.prior.GeneralPrior*

Prior distribution stating that any parameter values within the valid interval have the same prior probability.

No initialization is required.

__call__(*cube*)

Return variable value as it is

Parameters **cube** (*list*) – List of variable values in range [0,1]

Returns

Return type List of variable values in range [0,1]

class imagine.priors.basic_priors.**GaussianPrior**(*mu=0.0, sigma=1.0, interval=[-1.0, 1.0]*)

Bases: *imagine.priors.prior.ScipyPrior*

Truncated normal prior distribution

Parameters

- **mu** (*float*) – The position of the mode (mean, if the truncation is symmetric) of the Gaussian
- **sigma** (*float*) – Width of the distribution (standard deviation, if there was no tuncation)
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered.

imagine.priors.prior module

class imagine.priors.prior.**GeneralPrior**(*samples=None*, *pdf_fun=None*, *pdf_x=None*, *pdf_y=None*, *interval=None*, *bw_method=None*, *pdf_npoints=1500*, *inv_cdf_npoints=1500*)

Bases: `object`

Allows constructing a prior from a pre-existing sampling of the parameter space or a known probability density function (PDF).

Like in MultiNest, priors here are represented by a mapping of uniformly distributed scaled parameter values into a distribution with the desired properties (i.e. following the expected PDF). Such mapping is equivalent to the inverse of the cumulative distribution function (CDF) associated with the prior distribution.

Notes

Here we make a summary of the algorithm that allows finding the inverse of the CDF (and therefore, a IMAGINE prior) from a set of samples or PDF. (1) If set of samples is provided, computes Kernel Density Estimate (KDE) representation of it using Gaussian kernels. (2) The KDE is evaluated on *pdf_npoints* and thi si used to construct a interpolated cubic spline, which can be inspected through the method `pdf()`. (4) From PDF spline, the CDF is computed, which can be accessed using `cdf()`. (5) The CDF is evaluated on *inv_cdf_npoints*, and the inverse of the CDF is, again, constructed as a interpolated cubic spline. The spline object is available at `inv_cdf()`.

Parameters

- **samples** (*array_like*) – Array containing a sample of the prior distribution one wants to use. Note: this will use `scipy.stats.gaussian_kde` to compute the probability density function (PDF) through kernel density estimate using Gaussian kernels.
- **pdf_fun** (*function*) – A Python function containing the PDF associated with this prior. If an no *interval* is provided, the domain of PDF(x) will be assumed to be within the interval [0,1].
- **pdf_x, pdf_y** (*array_like*) – The PDF can be provided as two arrays of points following (pdf_x, pdf_y) = (x, PDF(x)). In the absence of an *interval*, the interval [min(pdf_x), max(pdf_x)] will be used.
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered.
- **bw_method** (*scalar or str*) – Used by `scipy.stats.gaussian_kde` to select the bandwidth employed to estimate the PDF from provided samples. Can be a number, if using fixed bandwidth, or strings ‘scott’ or ‘silverman’ if these rules are to be selected.
- **pdf_npoints** (*int*) – Number of points used to evaluate pdf_fun or the KDE constructed from the samples.
- **inv_cdf_npoints** (*int*) – Number of points used to evaluate the CDF for the calculation of its inverse.

__call__(*x*)
The “prior mapping”, i.e. returns the value of the inverse of the CDF at point(s) *x*.

pdf_unscaled(*x*)
Probability density function (PDF) associated with this prior.

cdf
Cumulative distribution function (CDF) associated with this prior.

inv_cdf
Inverse of the CDF associated with this prior, expressed as a scipy.interpolate.CubicSpline object.

pdf
Probability density function (PDF) associated with this prior.

class imagine.priors.prior.**ScipyPrior**(*distr, *args, loc=0.0, scale=1.0, interval=[-1.0, 1.0], **kwargs*)
Bases: *imagine.priors.prior.GeneralPrior*

Constructs a prior from a continuous distribution defined in scipy.stats.

Parameters

- **distr** (*scipy.stats.rv_continuous*) – A distribution function expressed as an instance of scipy.stats.rv_continuous.
- ***args** – Any positional arguments required by the function selected in distr (e.g for scipy.stats.chi2, one needs to supply the number of degrees of freedom, df)
- **loc** (*float*) – Same meaning as in scipy.stats.rv_continuous: sets the centre of the distribution (generally, the mean or mode).
- **scale** (*float*) – Same meaning as in scipy.stats.rv_continuous: sets the width of the distribution (e.g. the standard deviation in the normal case).
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered (will be used to rescale the interval).

Module contents

class imagine.priors.**FlatPrior**(*interval=[0, 1]*)
Bases: *imagine.priors.prior.GeneralPrior*

Prior distribution stating that any parameter values within the valid interval have the same prior probability.

No initialization is required.

__call__(*cube*)
Return variable value as it is

Parameters **cube** (*list*) – List of variable values in range [0,1]

Returns

Return type List of variable values in range [0,1]

class imagine.priors.**GaussianPrior**(*mu=0.0, sigma=1.0, interval=[-1.0, 1.0]*)
Bases: *imagine.priors.prior.ScipyPrior*

Truncated normal prior distribution

Parameters

- **mu** (*float*) – The position of the mode (mean, if the truncation is symmetric) of the Gaussian
- **sigma** (*float*) – Width of the distribution (standard deviation, if there was no tuncation)
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered.

class imagine.priors.**GeneralPrior**(*samples=None*, *pdf_fun=None*, *pdf_x=None*, *pdf_y=None*, *interval=None*, *bw_method=None*, *pdf_npoints=1500*, *inv_cdf_npoints=1500*)

Bases: object

Allows constructing a prior from a pre-existing sampling of the parameter space or a known probability density function (PDF).

Like in MultiNest, priors here are represented by a mapping of uniformly distributed scaled parameter values into a distribution with the desired properties (i.e. following the expected PDF). Such mapping is equivalent to the inverse of the cumulative distribution function (CDF) associated with the prior distribution.

Notes

Here we make a summary of the algorithm that allows finding the inverse of the CDF (and therefore, a IMAGINE prior) from a set of samples or PDF. (1) If set of samples is provided, computes Kernel Density Estimate (KDE) representation of it using Gaussian kernels. (2) The KDE is evaluated on *pdf_npoints* and thi si used to construct a interpolated cubic spline, which can be inspected through the method *pdf()*. (4) From PDF spline, the CDF is computed, which can be accessed using *cdf()*. (5) The CDF is evaluated on *inv_cdf_npoints*, and the inverse of the CDF is, again, constructed as a interpolated cubic spline. The spline object is available at *inv_cdf()*.

Parameters

- **samples** (*array_like*) – Array containing a sample of the prior distribution one wants to use. Note: this will use scipy.stats.gaussian_kde to compute the probability density function (PDF) through kernel density estimate using Gaussian kernels.
- **pdf_fun** (*function*) – A Python function containing the PDF associated with this prior. If an no *interval* is provided, the domain of PDF(x) will be assumed to be within the interval [0,1].
- **pdf_x, pdf_y** (*array_like*) – The PDF can be provided as two arrays of points following (pdf_x, pdf_y) = (x, PDF(x)). In the absence of an *interval*, the interval [min(pdf_x), max(pdf_x)] will be used.
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered.
- **bw_method** (*scalar or str*) – Used by scipy.stats.gaussian_kde to select the bandwidth employed to estimate the PDF from provided samples. Can be a number, if using fixed bandwidth, or strings 'scott' or 'silverman' if these rules are to be selected.
- **pdf_npoints** (*int*) – Number of points used to evaluate pdf_fun or the KDE constructed from the samples.
- **inv_cdf_npoints** (*int*) – Number of points used to evaluate the CDF for the calculation of its inverse.

__call__(*x*)
The "prior mapping", i.e. returns the value of the inverse of the CDF at point(s) *x*.

pdf_unscaled(*x*)
Probability density function (PDF) associated with this prior.

cdf
Cumulative distribution function (CDF) associated with this prior.

inv_cdf
Inverse of the CDF associated with this prior, expressed as a scipy.interpolate.CubicSpline object.

pdf
Probability density function (PDF) associated with this prior.

class imagine.priors.**ScipyPrior**(*distr, *args, loc=0.0, scale=1.0, interval=[-1.0, 1.0], **kwargs*)

Bases: *imagine.priors.prior.GeneralPrior*

Constructs a prior from a continuous distribution defined in scipy.stats.

Parameters

- **distr** (*scipy.stats.rv_continuous*) – A distribution function expressed as an instance of `scipy.stats.rv_continuous`.
- ***args** – Any positional arguments required by the function selected in `distr` (e.g for scipy.stats.chi2, one needs to supply the number of degrees of freedom, `df`)
- **loc** (*float*) – Same meaning as in `scipy.stats.rv_continuous`: sets the centre of the distribution (generally, the mean or mode).
- **scale** (*float*) – Same meaning as in `scipy.stats.rv_continuous`: sets the width of the distribution (e.g. the standard deviation in the normal case).
- **interval** (*tuple or list*) – A pair of points representing, respectively, the minimum and maximum parameter values to be considered (will be used to rescale the interval).

14.1.6 imagine.simulators package

Submodules

imagine.simulators.hammurabi module

class imagine.simulators.hammurabi.**Hammurabi**(*measurements*, *xml_path=None*, *exe_path=None*)

Bases: *imagine.simulators.simulator.Simulator*

This is an interface to hammurabi X Python wrapper.

Upon initialization, a Hampyx object is initialized and its XML tree should be modified according to measurements without changing its base file.

If a *xml_path* is provided, it will be used as the base XML tree, otherwise, hammurabiX's default 'params_template.xml' will be used.

`Dummy` fields can be used to change the parameters of hammurabiX. Other fields are temporarily saved to disk (using `imagine.rc` 'temp_dir' directory) and loaded into hammurabi.

Parameters

- **measurements** (*Measurements object*) – IMAGINE defined dictionary of measured data.
- **exe_path** (*string*) – Absolute hammurabi executable path.
- **xml_path** (*string*) – Absolute hammurabi xml parameter file path.

initialize_ham_xml()
Modify hammurabi XML tree according to the requested measurements.

simulate(*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).
- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

ALLOWED_GRID_TYPES = ['cartesian']

OPTIONAL_FIELD_TYPES = ['dummy', 'magnetic_field', 'thermal_electron_density', 'cosmic

REQUIRED_FIELD_TYPES = []

SIMULATED_QUANTITIES = ['fd', 'dm', 'sync']

imagine.simulators.simulator module

class imagine.simulators.simulator.**Simulator**(*measurements*)

Bases: *imagine.tools.class_tools.BaseClass*

Simulator base class

New Simulators must be introduced by sub-classing the present class. Overriding the method *simulate()* to convert a list of fields into simulated observables. For more details see *Simulators* section of the documentation.

Parameters **measurements** (*imagine.Measurements*) – An observables dictionary containing the set of measurements that will be used to prepare the mock observables

grid

Grid object where the fields were evaluated (NB if a common grid is not being used, this is set to None

Type imagine.Basegrid

grids

Grid objects for each individual field None if common grid is being used)

Type imagine.Basegrid

fields

Dictionary containing field types as keys and the sum of evaluated fields as values

Type dict

observables

List of Observable keys

Type list

output_units

Output units used in the simulator

Type astropy.units.Unit

__call__(*field_list*)

Runs the simulator over a Fields list

Parameters **field_list** (*list*) – List of imagine.Field object which must include all the *required_field_types*

Returns **sims** – A Simulations object containing all the specified mock data

Return type imagine.Simulations

prepare_fields (*field_list*, *i*)
Registers the available fields checking whether all requirements are satisfied. all data is saved on a dictionary, simulator.fields, where field_types are keys.

The *fields* dictionary is reconstructed for *each realisation* of the ensemble. It relies on caching within the Field objects to avoid computing the same quantities multiple times.

If there is more than one field of the same type, they are summed together.

Parameters

- **field_list** (*list*) – List containing Field objects
- **i** (*int*) – Index of the realisation of the fields that is being registred

register_ensemble_size (*field_list*)
Checks whether fields have consistent ensemble size and stores this information

register_observables (*measurements*)
Called during initalization to store the relevant information in the measurements dictionary

Parameters **measurements** (*imagine.Measurements*) – An observables dictionary containing the set of measurements that will be used to prepare the mock observables

simulate (*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq), str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).
- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

REQ_ATTRS = ['SIMULATED_QUANTITIES', 'REQUIRED_FIELD_TYPES', 'ALLOWED_GRID_TYPES']

allowed_grid_types
Must be overriden with a list or set of allowed grid types that work with this Simulator. Example: [‘cartesian’]

ensemble_size

optional_field_types
Can be overriden with a list or set of field types that Simulator can use if available. Example: [‘magnetic_field’, ‘cosmic_ray_electron_density’]

required_field_types
Must be overriden with a list or set of required field types that the Simulator needs. Example: [‘magnetic_field’, ‘cosmic_ray_electron_density’]

simulated_quantities
Must be overriden with a list or set of simulated quantities this Simulator produces. Example: [‘fd’, ‘sync’]

use_common_grid
Must be overriden with a list or set of allowed grid types that work with this Simulator. Example: [‘cartesian’]

imagine.simulators.test_simulator module

For testing purposes only

class imagine.simulators.test_simulator.**TestSimulator**(*measurements*, *LoS_axis='y'*)
Bases: *imagine.simulators.simulator.Simulator*

Example simulator for illustration and testing

Computes a Faraday-depth-like property at a given point without performing the integration, i.e. computes:

$$\mathrm{t(x,y,z)} = \mathrm{B}_y \, n_e$$

simulate(*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).
- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

ALLOWED_GRID_TYPES = ['cartesian']

REQUIRED_FIELD_TYPES = ['magnetic_field', 'thermal_electron_density']

SIMULATED_QUANTITIES = ['test']

Module contents

class imagine.simulators.**Hammurabi**(*measurements*, *xml_path=None*, *exe_path=None*)
Bases: *imagine.simulators.simulator.Simulator*

This is an interface to hammurabi X Python wrapper.

Upon initialization, a Hampyx object is initialized and its XML tree should be modified according to measurements without changing its base file.

If a *xml_path* is provided, it will be used as the base XML tree, otherwise, hammurabiX's default 'params_template.xml' will be used.

`Dummy` fields can be used to change the parameters of hammurabiX. Other fields are temporarily saved to disk (using `imagine.rc` 'temp_dir' directory) and loaded into hammurabi.

Parameters

- **measurements** (*Measurements object*) – IMAGINE defined dictionary of measured data.
- **exe_path** (*string*) – Absolute hammurabi executable path.
- **xml_path** (*string*) – Absolute hammurabi xml parameter file path.

initialize_ham_xml()
Modify hammurabi XML tree according to the requested measurements.

simulate(*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).
- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

ALLOWED_GRID_TYPES = ['cartesian']

OPTIONAL_FIELD_TYPES = ['dummy', 'magnetic_field', 'thermal_electron_density', 'cosmic

REQUIRED_FIELD_TYPES = []

SIMULATED_QUANTITIES = ['fd', 'dm', 'sync']

class imagine.simulators.**Simulator**(*measurements*)
Bases: *imagine.tools.class_tools.BaseClass*

Simulator base class

New Simulators must be introduced by sub-classing the present class. Overriding the method *simulate()* to convert a list of fields into simulated observables. For more details see *Simulators* section of the documentation.

Parameters **measurements** (*imagine.Measurements*) – An observables dictionary containing the set of measurements that will be used to prepare the mock observables

grid
Grid object where the fields were evaluated (NB if a common grid is not being used, this is set to None

Type imagine.Basegrid

grids
Grid objects for each individual field None if common grid is being used)

Type imagine.Basegrid

fields
Dictionary containing field types as keys and the sum of evaluated fields as values

Type dict

observables
List of Observable keys

Type list

output_units
Output units used in the simulator

Type astropy.units.Unit

__call__ (*field_list*)
Runs the simulator over a Fields list

Parameters **field_list** (*list*) – List of imagine.Field object which must include all the *required_field_types*

Returns **sims** – A Simulations object containing all the specified mock data

Return type imagine.Simulations

prepare_fields (*field_list*, *i*)
Registers the available fields checking whether all requirements are satisfied. all data is saved on a dictionary, simulator.fields, where field_types are keys.

The *fields* dictionary is reconstructed for *each realisation* of the ensemble. It relies on caching within the Field objects to avoid computing the same quantities multiple times.

If there is more than one field of the same type, they are summed together.

Parameters

- **field_list** (*list*) – List containing Field objects
- **i** (*int*) – Index of the realisation of the fields that is being registred

register_ensemble_size (*field_list*)
Checks whether fields have consistent ensemble size and stores this information

register_observables (*measurements*)
Called during initalization to store the relevant information in the measurements dictionary

Parameters **measurements** (*imagine.Measurements*) – An observables dictionary containing the set of measurements that will be used to prepare the mock observables

simulate (*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).

- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

REQ_ATTRS = ['SIMULATED_QUANTITIES', 'REQUIRED_FIELD_TYPES', 'ALLOWED_GRID_TYPES']

allowed_grid_types
Must be overriden with a list or set of allowed grid types that work with this Simulator. Example: ['cartesian']

ensemble_size

optional_field_types
Can be overriden with a list or set of field types that Simulator can use if available. Example: ['magnetic_field', 'cosmic_ray_electron_density']

required_field_types
Must be overriden with a list or set of required field types that the Simulator needs. Example: ['magnetic_field', 'cosmic_ray_electron_density']

simulated_quantities
Must be overriden with a list or set of simulated quantities this Simulator produces. Example: ['fd', 'sync']

use_common_grid
Must be overriden with a list or set of allowed grid types that work with this Simulator. Example: ['cartesian']

class imagine.simulators.**TestSimulator**(*measurements*, *LoS_axis='y'*)
Bases: *imagine.simulators.simulator.Simulator*

Example simulator for illustration and testing

Computes a Faraday-depth-like property at a given point without performing the integration, i.e. computes:

$$t(x,y,z) = B_y\, n_e$$

simulate(*key*, *coords_dict*, *realization_id*, *output_units*)
Must be overriden with a function that returns the observable described by *key* using the fields in self.fields, in units *output_units*.

Parameters

- **key** (*tuple*) – Observable key in the standard form `(data-name,str(data-freq),str(data-Nside)/"tab",str(ext))`
- **coords_dict** (*dictionary*) – Dictionary containing coordinates associated with the observable (or None for HEALPix datasets).
- **Nside** (*int*) – HEALPix Nside parameter for HEALPix datasets (or None for tabular datasets).
- **output_units** (*astropy.units.Unit*) – The physical units that should be used for this mock observable

Returns 1D *pure* numpy array of length compatible with Nside or coords_dict containing the mock observable in the output_units.

Return type numpy.ndarray

ALLOWED_GRID_TYPES = ['cartesian']

REQUIRED_FIELD_TYPES = ['magnetic_field', 'thermal_electron_density']

SIMULATED_QUANTITIES = ['test']

14.1.7 imagine.tools package

Submodules

imagine.tools.carrier_mapper module

The mapper module is designed for implementing distribution mapping functions.

imagine.tools.carrier_mapper.**exp_mapper**(*x*, *a=0*, *b=1*)
Maps x from [0, 1] into the interval [exp(a), exp(b)].

Parameters

- **x** (*float*) – The variable to be mapped.
- **a** (*float*) – The lower parameter value limit.
- **b** (*float*) – The upper parameter value limit.

Returns The mapped parameter value.

Return type numpy.float64

imagine.tools.carrier_mapper.**unity_mapper**(*x*, *a=0.0*, *b=1.0*)
Maps x from [0, 1] into the interval [a, b].

Parameters

- **x** (*float*) – The variable to be mapped.
- **a** (*float*) – The lower parameter value limit.
- **b** (*float*) – The upper parameter value limit.

Returns The mapped parameter value.

Return type numpy.float64

imagine.tools.class_tools module

class imagine.tools.class_tools.**BaseClass**
Bases: object

REQ_ATTRS = []

imagine.tools.class_tools.**req_attr**(*meth*)

imagine.tools.config module

IMAGINE global configuration

The default behaviour of some aspects of IMAGINE can be set using global *rc* configuration variables.

These can be accessed and modified using the `imagine.rc` dictionary or setting the corresponding environment variables (named 'IMAGINE_'+RC_VAR_NAME).

For example to set the default path for the hamx executable, one can either do:

```
import imagine
imagine.rc.hammurabi_hamx_path = 'my_desired_path'
```

or, alternatively, set this as an environment variable before the exectution of the script:

```
export IMAGINE_HAMMURABI_HAMX_PATH='my_desired_path'
```

The following list describes all the available global settings variables.

IMAGINE rc variables

temp_dir Default temporary directory used by IMAGINE. If not set, a temporary directory will be created at /tmp/ with a safe name.

distributed_arrays If *True*, arrays containing covariances are distributed among different MPI processes (and so are the corresponding array operations).

pipeline_default_seed The default value for the master seed used by a Pipeline object (see *Pipeline.master_seed*).

pipeline_distribute_ensemble The default value of (see *Pipeline.distribute_ensemble*).

hammurabi_hamx_path Default location of the Hammurabi X executable file, *hamx*.

imagine.tools.covariance_estimator module

This module contains estimation algorithms for the covariance matrix based on a finite number of samples.

For the testing suits, please turn to "imagine/tests/tools_tests.py".

`imagine.tools.covariance_estimator.`**`empirical_cov`**(*data*)

Empirical covariance estimator

Given some data matrix, D, where rows are different samples and columns different properties, the covariance can be estimated from

$$U_{ij} = D_{ij} - \overline{D}_j\,, \text{ with } \overline{D}_j = \frac{1}{N}\sum_{i=1}^{N} D_{ij}$$

$$\text{cov} = \frac{1}{N} U^T U$$

Notes

While conceptually simple, this is usually not the best option.

Parameters **data** (*numpy.ndarray*) – Ensemble of observables, in global shape (ensemble size, data size).

Returns **cov** – Distributed (not copied) covariance matrix in global shape (data size, data size), each node takes part of the rows.

Return type numpy.ndarray

imagine.tools.covariance_estimator.**oas_cov**(*data*)

Estimate covariance with the Oracle Approximating Shrinkage algorithm.

Given some $n \times m$ data matrix, D, where rows are different samples and columns different properties, the covariance can be estimated in the following way.

$$U_{ij} = D_{ij} - \overline{D}_j \,, \text{ with } \overline{D}_j = \frac{1}{n} \sum_{i=1}^{n} D_{ij}$$

Let

$$S = \frac{1}{n} U^T U \,, \; T = \mathrm{tr}(S) \quad \text{and} \quad V = \mathrm{tr}(S^2)$$

$$\tilde{\rho} = \min \left[1, \frac{(1 - 2/m)V + T^2}{(n + 1 - 2/m)(V - T^2/m)} \right]$$

The covariance is given by

$$\mathrm{cov}_{\mathrm{OAS}} = (1 - \rho) S + \frac{1}{N} \rho T I_m$$

Parameters **data** (*numpy.ndarray*) – Distributed data in global shape (ensemble_size, data_size).

Returns **cov** – Covariance matrix in global shape (data_size, data_size).

Return type numpy.ndarray

imagine.tools.covariance_estimator.**oas_mcov**(*data*)

Estimate covariance with the Oracle Approximating Shrinkage algorithm.

See *imagine.tools.covariance_estimator.oas_cov* for details. This function aditionally returns the computed ensemble mean.

Parameters **data** (*numpy.ndarray*) – Distributed data in global shape (ensemble_size, data_size).

Returns

- **mean** (*numpy.ndarray*) – Copied ensemble mean (on all nodes).
- **cov** (*numpy.ndarray*) – Distributed covariance matrix in shape (data_size, data_size).

imagine.tools.io_handler module

The io_handler class is designed for IMAGINE I/O with HDF5+MPI, but parallel HDF5 is not required.

There are two types of data reading, corresponding to the data types defined in the Observable class.

1. for reading 'measured' data (including mask maps), each node reads the full data. 'read_copy' is designed for this case.

2. for reading 'covariance' data, each node reads a certain rows. 'read_dist' is designed for this case.

We do not require writing in parallel since the output workload is not heavy. And there are also two types of data writing out, corresponds to the reading, i.e., 'write_copy' and 'write_dist'.

For the testing suits, please turn to "imagine/tests/tools_tests.py".

class imagine.tools.io_handler.**IOHandler**(*wk_dir=None*)
Bases: object

Handles the I/O.

Parameters **wkdir** (*string*) – The absolute path of the working directory.

read_copy(*file*, *key*)
Reads from a HDF5 file identically to all nodes, by doing so, each node contains an identical copy of the data stored in the file.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

Returns The output must be in (1,n) shape on each node.

Return type Copied numpy.ndarray.

read_dist(*file*, *key*)
Reads from a HDF5 file and returns a distributed data-set. Note that the binary file data should contain enough rows to be distributed on the available computing nodes, otherwise the mpi_arrange function will raise an error.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

Returns The output must be in either at least (1,n), or (m,n) shape on each node.

Return type Distributed numpy.ndarray.

write_copy(*data*, *file*, *key*)
Writes a copied data-set into a HDF5 file. In practice, it writes out the data stored in the master node, by defaut taking all nodes have the same copies.

Parameters

- **data** (*numpy.ndarray*) – Distributed/copied data.
- **file** (*str*) – Strong for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

write_dist(*data*, *file*, *key*)
Writes a distributed data-set into a HDF5 file. If the given filename does not exist then creates one the data shape must be either in (m,n) on each node, each node will pass its content to the master node who is in charge of sequential writing.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

file_path
Absolute path of the HDF5 binary file.

wk_dir
String containing the absolute path of the working directory.

imagine.tools.masker module

This module defines methods related to masking out distributed data and/or the associated covariance matrix. For the testing suits, please turn to "imagine/tests/tools_tests.py".

Implemented with numpy.ndarray raw data.

imagine.tools.masker.**mask_cov**(*cov*, *mask*)
Applies mask to the observable covariance.

Parameters

- **cov** (*distributed numpy.ndarray*) – Covariance matrix of observalbes in global shape (data size, data size) each node contains part of the global rows.
- **mask** (*numpy.ndarray*) – Copied mask map in shape (1, data size).

Returns Masked covariance matrix of shape (masked data size, masked data size).

Return type numpy.ndarray

imagine.tools.masker.**mask_obs**(*obs*, *mask*)
Applies a mask to an observable.

Parameters

- **data** (*distributed numpy.ndarray*) – Ensemble of observables, in global shape (ensemble size, data size) each node contains part of the global rows.
- **mask** (*numpy.ndarray*) – Copied mask map in shape (1, data size) on each node.

Returns Masked observable of shape (ensemble size, masked data size).

Return type numpy.ndarray

imagine.tools.misc module

imagine.tools.misc.**adjust_error_intervals**(*value*, *errlo*, *errup*, *sdigits=2*)
Takes the value of a quantity *value* with associated errors *errlo* and *errup*; and prepares them to be reported as $v^{+err\,up}_{-err\,down}$.

Parameters

- **value** (*int or float*) – Value of quantity.
- **errlo, errup** (*int or float*) – Associated lower and upper errors of *value*.

Returns

- **value** (*float*) – Rounded value
- **errlo, errup** (*float*) – Assimetric error values

imagine.tools.misc.**is_notebook**()
Finds out whether python is running in a Jupyter notebook or as a shell.

imagine.tools.mpi_helper module

This MPI helper module is designed for parallel computing and data handling.

For the testing suits, please turn to "imagine/tests/tools_tests.py".

imagine.tools.mpi_helper.**mpi_arrange**(*size*)
With known global size, number of mpi nodes, and current rank, returns the begin and end index for distributing the global size.

Parameters **size** (*integer (positive)*) – The total size of target to be distributed. It can be a row size or a column size.

Returns **result** – The begin and end index [begin,end] for slicing the target.

Return type numpy.uint

imagine.tools.mpi_helper.**mpi_shape**(*data*)
Returns the global number of rows and columns of given distributed data.

Parameters **data** (*numpy.ndarray*) – The distributed data.

Returns **result** – Glboal row and column number.

Return type numpy.uint

imagine.tools.mpi_helper.**mpi_prosecutor**(*data*)
Check if the data is distributed in the correct way covariance matrix is distributed exactly the same manner as multi-realization data if not, an error will be raised.

Parameters **data** (*numpy.ndarray*) – The distributed data to be examined.

imagine.tools.mpi_helper.**mpi_mean**(*data*)
calculate the mean of distributed array prefers averaging along column direction but if given (1,n) data shape the average is done along row direction the result note that the numerical values will be converted into double

Parameters **data** (*numpy.ndarray*) – Distributed data.

Returns **result** – Copied data mean, which means the mean is copied to all nodes.

Return type numpy.ndarray

imagine.tools.mpi_helper.**mpi_trans**(*data*)
Transpose distributed data, note that the numerical values will be converted into double.

Parameters **data** (*numpy.ndarray*) – Distributed data.

Returns **result** – Transposed data in distribution.

Return type numpy.ndarray

imagine.tools.mpi_helper.**mpi_mult**(*left*, *right*)
Calculate matrix multiplication of two distributed data, the result is data1*data2 in multi-node distribution note that the numerical values will be converted into double. We send the distributed right rows into other nodes (aka cannon method).

Parameters

- **left** (*numpy.ndarray*) – Distributed left side data.
- **right** (*numpy.ndarray*) – Distributed right side data.

Returns **result** – Distributed multiplication result.

Return type numpy.ndarray

imagine.tools.mpi_helper.**mpi_trace**(*data*)
Computes the trace of the given distributed data.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns **result** – Copied trace of given data.

Return type numpy.float64

imagine.tools.mpi_helper.**mpi_eye**(*size*)
Produces an eye matrix according of shape (size,size) distributed over the various running MPI processes

Parameters **size** (*integer*) – Distributed matrix size.

Returns **result** – Distributed eye matrix.

Return type numpy.ndarray, double data type

imagine.tools.mpi_helper.**mpi_distribute_matrix**(*full_matrix*)

Parameters **size** (*integer*) – Distributed matrix size.

Returns **result** – Distributed eye matrix.

Return type numpy.ndarray, double data type

imagine.tools.mpi_helper.**mpi_lu_solve**(*operator*, *source*)
Simple LU Gauss method WITHOUT pivot permutation.

Parameters

- **operator** (*distributed numpy.ndarray*) – Matrix representation of the left-hand-side operator.
- **source** (*copied numpy.ndarray*) – Vector representation of the right-hand-side source.

Returns **result** – Copied solution to the linear algebra problem.

Return type numpy.ndarray, double data type

imagine.tools.mpi_helper.**mpi_slogdet**(*data*)
Computes log determinant according to simple LU Gauss method WITHOUT pivot permutation.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns

- **sign** (*numpy.ndarray*) – Single element numpy array containing the sign of the determinant (copied to all nodes).
- **logdet** (*numpy.ndarray*) – Single element numpy array containing the log of the determinant (copied to all nodes).

imagine.tools.mpi_helper.**mpi_global**(*data*)
Gathers data spread accross different processes.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns **global array** – The root process returns the gathered data, other processes return *None*.

Return type numpy.ndarray

imagine.tools.mpi_helper.**mpi_local**(*data*)
Distributes data over available processes

Parameters **data** (*numpy.ndarray*) – Array of data to be distributed over available processes.

Returns **local array** – Return the distributed array on all preocesses.

Return type numpy.ndarray

imagine.tools.parallel_ops module

Interface module which allows automatically switching between the routines in the *imagine.tools.mpi_helper* module and their:py:mod:*numpy* or pure Python equivalents, depending on the contents of `imagine.rc['distributed_arrays']`

`imagine.tools.parallel_ops.`**`pshape`**(*data*)
imagine.tools.mpi_helper.mpi_shape() or `numpy.ndarray.shape()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`prosecutor`**(*data*)
imagine.tools.mpi_helper.mpi_prosecutor() or *nothing* depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`pmean`**(*data*)
imagine.tools.mpi_helper.mpi_mean() or `numpy.mean()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`ptrans`**(*data*)
imagine.tools.mpi_helper.mpi_mean() or `numpy.ndarray.T()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`pmult`**(*left*, *right*)
imagine.tools.mpi_helper.mpi_mult() or `numpy.matmul()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`ptrace`**(*data*)
imagine.tools.mpi_helper.mpi_trace() or `numpy.trace()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`peye`**(*size*)
imagine.tools.mpi_helper.mpi_eye() or `numpy.eye()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`distribute_matrix`**(*full_matrix*)
imagine.tools.mpi_helper.mpi_distribute_matrix() or *nothing* depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`plu_solve`**(*operator*, *source*)
imagine.tools.mpi_helper.mpi_lu_solve() or `numpy.linalg.solve()` depending on `imagine.rc['distributed_arrays']`.

Notes

In the non-distributed case, the source is transposed before the calculation

`imagine.tools.parallel_ops.`**`pslogdet`**(*data*)
imagine.tools.mpi_helper.mpi_slogdet() or `numpy.linalg.slogdet()` depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`pglobal`**(*data*)
imagine.tools.mpi_helper.mpi_global() or *nothing* depending on `imagine.rc['distributed_arrays']`.

`imagine.tools.parallel_ops.`**`plocal`**(*data*)
imagine.tools.mpi_helper.mpi_local() or *nothing* depending on `imagine.rc['distributed_arrays']`.

imagine.tools.random_seed module

This module provides a time-thread dependent seed value.

For the testing suites, please turn to "imagine/tests/tools_tests.py".

imagine.tools.random_seed.**ensemble_seed_generator**(*size*)
Generates fixed random seed values for each realization in ensemble.

Parameters **size** (*int*) – Number of realizations in ensemble.

Returns **seeds** – An array of random seeds.

Return type numpy.ndarray

imagine.tools.random_seed.**seed_generator**(*trigger*)
Sets trigger as 0 will generate time-thread dependent method otherwise returns the trigger as seed.

Parameters **trigger** (*int*) – Non-negative pre-fixed seed.

Returns **seed** – A random seed value.

Return type int

imagine.tools.timer module

Timer class is designed for time recording.

class imagine.tools.timer.**Timer**
Bases: object

Class designed for time recording.

Simply provide an event name to the *tick* method to start recording. The *tock* method stops the recording and the *record* property allow one to access the recorded time.

tick(*event*)
Starts timing with a given event name.

Parameters **event** (*str*) – Event name (will be key of the record attribute).

tock(*event*)
Stops timing of the given event.

Parameters **event** (*str*) – Event name (will be key of the record attribute).

record
Dictionary of recorded times using event name as keys.

Module contents

class imagine.tools.**BaseClass**
Bases: object

REQ_ATTRS = []

class imagine.tools.**IOHandler**(*wk_dir=None*)
Bases: object

Handles the I/O.

Parameters **wkdir** (*string*) – The absolute path of the working directory.

read_copy (*file*, *key*)
Reads from a HDF5 file identically to all nodes, by doing so, each node contains an identical copy of the data stored in the file.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

Returns The output must be in (1,n) shape on each node.

Return type Copied numpy.ndarray.

read_dist (*file*, *key*)
Reads from a HDF5 file and returns a distributed data-set. Note that the binary file data should contain enough rows to be distributed on the available computing nodes, otherwise the mpi_arrange function will raise an error.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

Returns The output must be in either at least (1,n), or (m,n) shape on each node.

Return type Distributed numpy.ndarray.

write_copy (*data*, *file*, *key*)
Writes a copied data-set into a HDF5 file. In practice, it writes out the data stored in the master node, by defaut taking all nodes have the same copies.

Parameters

- **data** (*numpy.ndarray*) – Distributed/copied data.
- **file** (*str*) – Strong for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

write_dist (*data*, *file*, *key*)
Writes a distributed data-set into a HDF5 file. If the given filename does not exist then creates one the data shape must be either in (m,n) on each node, each node will pass its content to the master node who is in charge of sequential writing.

Parameters

- **data** (*numpy.ndarray*) – Distributed data.
- **file** (*str*) – String for filename.
- **key** (*str*) – String for HDF5 group and dataset names, e.g., 'group name/dataset name'.

file_path
Absolute path of the HDF5 binary file.

wk_dir
String containing the absolute path of the working directory.

class imagine.tools.**Timer**
Bases: object

Class designed for time recording.

Simply provide an event name to the *tick* method to start recording. The *tock* method stops the recording and the *record* property allow one to access the recorded time.

tick(*event*)
Starts timing with a given event name.

Parameters **event** (*str*) – Event name (will be key of the record attribute).

tock(*event*)
Stops timing of the given event.

Parameters **event** (*str*) – Event name (will be key of the record attribute).

record
Dictionary of recorded times using event name as keys.

imagine.tools.**exp_mapper**(*x*, *a=0*, *b=1*)
Maps x from [0, 1] into the interval [exp(a), exp(b)].

Parameters

- **x** (*float*) – The variable to be mapped.
- **a** (*float*) – The lower parameter value limit.
- **b** (*float*) – The upper parameter value limit.

Returns The mapped parameter value.

Return type numpy.float64

imagine.tools.**unity_mapper**(*x*, *a=0.0*, *b=1.0*)
Maps x from [0, 1] into the interval [a, b].

Parameters

- **x** (*float*) – The variable to be mapped.
- **a** (*float*) – The lower parameter value limit.
- **b** (*float*) – The upper parameter value limit.

Returns The mapped parameter value.

Return type numpy.float64

imagine.tools.**req_attr**(*meth*)

imagine.tools.**empirical_cov**(*data*)
Empirical covariance estimator

Given some data matrix, D, where rows are different samples and columns different properties, the covariance can be estimated from

$$U_{ij} = D_{ij} - \overline{D}_j\,, \text{ with } \overline{D}_j = \frac{1}{N}\sum_{i=1}^{N} D_{ij}$$

$$\text{cov} = \frac{1}{N} U^T U$$

Notes

While conceptually simple, this is usually not the best option.

Parameters **data** (*numpy.ndarray*) – Ensemble of observables, in global shape (ensemble size, data size).

Returns **cov** – Distributed (not copied) covariance matrix in global shape (data size, data size), each node takes part of the rows.

Return type numpy.ndarray

imagine.tools.**oas_cov**(*data*)

Estimate covariance with the Oracle Approximating Shrinkage algorithm.

Given some $n \times m$ data matrix, D, where rows are different samples and columns different properties, the covariance can be estimated in the following way.

$$U_{ij} = D_{ij} - \overline{D}_j \,, \text{ with } \overline{D}_j = \frac{1}{n} \sum_{i=1}^{n} D_{ij}$$

Let

$$S = \frac{1}{n} U^T U \,, \; T = \text{tr}(S) \quad \text{and} \quad V = \text{tr}(S^2)$$

$$\tilde{\rho} = \min \left[1, \frac{(1 - 2/m)V + T^2}{(n + 1 - 2/m)(V - T^2/m)} \right]$$

The covariance is given by

$$\text{cov}_{\text{OAS}} = (1 - \rho) S + \frac{1}{N} \rho T I_m$$

Parameters **data** (*numpy.ndarray*) – Distributed data in global shape (ensemble_size, data_size).

Returns **cov** – Covariance matrix in global shape (data_size, data_size).

Return type numpy.ndarray

imagine.tools.**oas_mcov**(*data*)

Estimate covariance with the Oracle Approximating Shrinkage algorithm.

See *imagine.tools.covariance_estimator.oas_cov* for details. This function aditionally returns the computed ensemble mean.

Parameters **data** (*numpy.ndarray*) – Distributed data in global shape (ensemble_size, data_size).

Returns

- **mean** (*numpy.ndarray*) – Copied ensemble mean (on all nodes).
- **cov** (*numpy.ndarray*) – Distributed covariance matrix in shape (data_size, data_size).

imagine.tools.**mask_cov**(*cov*, *mask*)

Applies mask to the observable covariance.

Parameters

- **cov** (*distributed numpy.ndarray*) – Covariance matrix of observalbes in global shape (data size, data size) each node contains part of the global rows.
- **mask** (*numpy.ndarray*) – Copied mask map in shape (1, data size).

Returns Masked covariance matrix of shape (masked data size, masked data size).

Return type numpy.ndarray

imagine.tools.**mask_obs**(*obs*, *mask*)
Applies a mask to an observable.

Parameters

- **data** (*distributed numpy.ndarray*) – Ensemble of observables, in global shape (ensemble size, data size) each node contains part of the global rows.
- **mask** (*numpy.ndarray*) – Copied mask map in shape (1, data size) on each node.

Returns Masked observable of shape (ensemble size, masked data size).

Return type numpy.ndarray

imagine.tools.**mpi_arrange**(*size*)
With known global size, number of mpi nodes, and current rank, returns the begin and end index for distributing the global size.

Parameters **size** (*integer (positive)*) – The total size of target to be distributed. It can be a row size or a column size.

Returns **result** – The begin and end index [begin,end] for slicing the target.

Return type numpy.uint

imagine.tools.**mpi_shape**(*data*)
Returns the global number of rows and columns of given distributed data.

Parameters **data** (*numpy.ndarray*) – The distributed data.

Returns **result** – Glboal row and column number.

Return type numpy.uint

imagine.tools.**mpi_prosecutor**(*data*)
Check if the data is distributed in the correct way covariance matrix is distributed exactly the same manner as multi-realization data if not, an error will be raised.

Parameters **data** (*numpy.ndarray*) – The distributed data to be examined.

imagine.tools.**mpi_mean**(*data*)
calculate the mean of distributed array prefers averaging along column direction but if given (1,n) data shape the average is done along row direction the result note that the numerical values will be converted into double

Parameters **data** (*numpy.ndarray*) – Distributed data.

Returns **result** – Copied data mean, which means the mean is copied to all nodes.

Return type numpy.ndarray

imagine.tools.**mpi_trans**(*data*)
Transpose distributed data, note that the numerical values will be converted into double.

Parameters **data** (*numpy.ndarray*) – Distributed data.

Returns **result** – Transposed data in distribution.

Return type numpy.ndarray

imagine.tools.**mpi_mult**(*left*, *right*)
Calculate matrix multiplication of two distributed data, the result is data1*data2 in multi-node distribution note that the numerical values will be converted into double. We send the distributed right rows into other nodes (aka cannon method).

Parameters

- **left** (*numpy.ndarray*) – Distributed left side data.

- **right** (*numpy.ndarray*) – Distributed right side data.

Returns **result** – Distributed multiplication result.

Return type numpy.ndarray

imagine.tools.**mpi_trace**(*data*)
Computes the trace of the given distributed data.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns **result** – Copied trace of given data.

Return type numpy.float64

imagine.tools.**mpi_eye**(*size*)
Produces an eye matrix according of shape (size,size) distributed over the various running MPI processes

Parameters **size** (*integer*) – Distributed matrix size.

Returns **result** – Distributed eye matrix.

Return type numpy.ndarray, double data type

imagine.tools.**mpi_distribute_matrix**(*full_matrix*)

Parameters **size** (*integer*) – Distributed matrix size.

Returns **result** – Distributed eye matrix.

Return type numpy.ndarray, double data type

imagine.tools.**mpi_lu_solve**(*operator*, *source*)
Simple LU Gauss method WITHOUT pivot permutation.

Parameters

- **operator** (*distributed numpy.ndarray*) – Matrix representation of the left-hand-side operator.
- **source** (*copied numpy.ndarray*) – Vector representation of the right-hand-side source.

Returns **result** – Copied solution to the linear algebra problem.

Return type numpy.ndarray, double data type

imagine.tools.**mpi_slogdet**(*data*)
Computes log determinant according to simple LU Gauss method WITHOUT pivot permutation.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns

- **sign** (*numpy.ndarray*) – Single element numpy array containing the sign of the determinant (copied to all nodes).
- **logdet** (*numpy.ndarray*) – Single element numpy array containing the log of the determinant (copied to all nodes).

imagine.tools.**mpi_global**(*data*)
Gathers data spread accross different processes.

Parameters **data** (*numpy.ndarray*) – Array of data distributed over different processes.

Returns **global array** – The root process returns the gathered data, other processes return *None*.

Return type numpy.ndarray

imagine.tools.**mpi_local**(*data*)
Distributes data over available processes

Parameters **data** (*numpy.ndarray*) – Array of data to be distributed over available processes.

Returns **local array** – Return the distributed array on all preocesses.

Return type numpy.ndarray

imagine.tools.**pshape**(*data*)
imagine.tools.mpi_helper.mpi_shape() or `numpy.ndarray.shape()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**prosecutor**(*data*)
imagine.tools.mpi_helper.mpi_prosecutor() or *nothing* depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**pmean**(*data*)
imagine.tools.mpi_helper.mpi_mean() or `numpy.mean()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**ptrans**(*data*)
imagine.tools.mpi_helper.mpi_mean() or `numpy.ndarray.T()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**pmult**(*left*, *right*)
imagine.tools.mpi_helper.mpi_mult() or `numpy.matmul()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**ptrace**(*data*)
imagine.tools.mpi_helper.mpi_trace() or `numpy.trace()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**peye**(*size*)
imagine.tools.mpi_helper.mpi_eye() or `numpy.eye()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**distribute_matrix**(*full_matrix*)
imagine.tools.mpi_helper.mpi_distribute_matrix() or *nothing* depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**plu_solve**(*operator*, *source*)
imagine.tools.mpi_helper.mpi_lu_solve() or `numpy.linalg.solve()` depending on `imagine.rc['distributed_arrays']`.

Notes

In the non-distributed case, the source is transposed before the calculation

imagine.tools.**pslogdet**(*data*)
imagine.tools.mpi_helper.mpi_slogdet() or `numpy.linalg.slogdet()` depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**pglobal**(*data*)
imagine.tools.mpi_helper.mpi_global() or *nothing* depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**plocal**(*data*)
imagine.tools.mpi_helper.mpi_local() or *nothing* depending on `imagine.rc['distributed_arrays']`.

imagine.tools.**ensemble_seed_generator**(*size*)
Generates fixed random seed values for each realization in ensemble.

Parameters **size** (*int*) – Number of realizations in ensemble.

Returns **seeds** – An array of random seeds.

Return type numpy.ndarray

imagine.tools.**seed_generator**(*trigger*)

Sets trigger as 0 will generate time-thread dependent method otherwise returns the trigger as seed.

Parameters **trigger** (*int*) – Non-negative pre-fixed seed.

Returns **seed** – A random seed value.

Return type int

14.2 Module contents

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

i

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

W

X

Y

Z