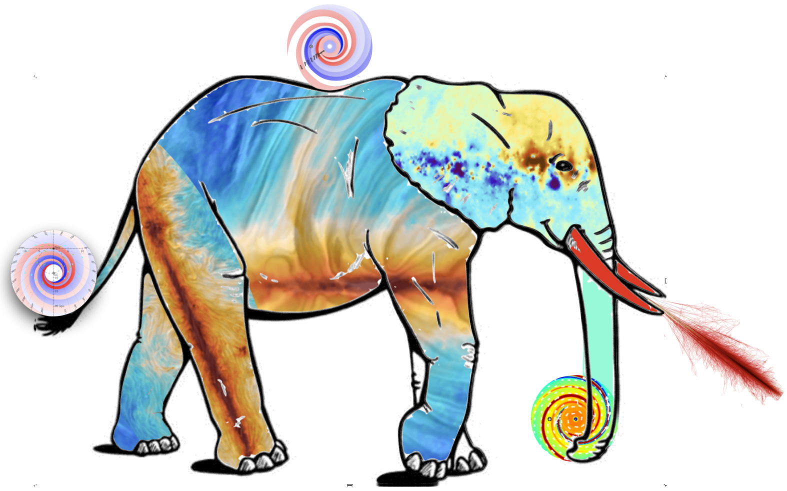
IMAGINE

Jan 08, 2021

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 docker.pkg.github.com/imagine-consortium/imagine/
↪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.3.tar.gz
tar -xvvzf v2.0.0-alpha.3.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 Factories* class. Likewise, *Observable Dictionaries* help one organising and manipulating *Observables*.

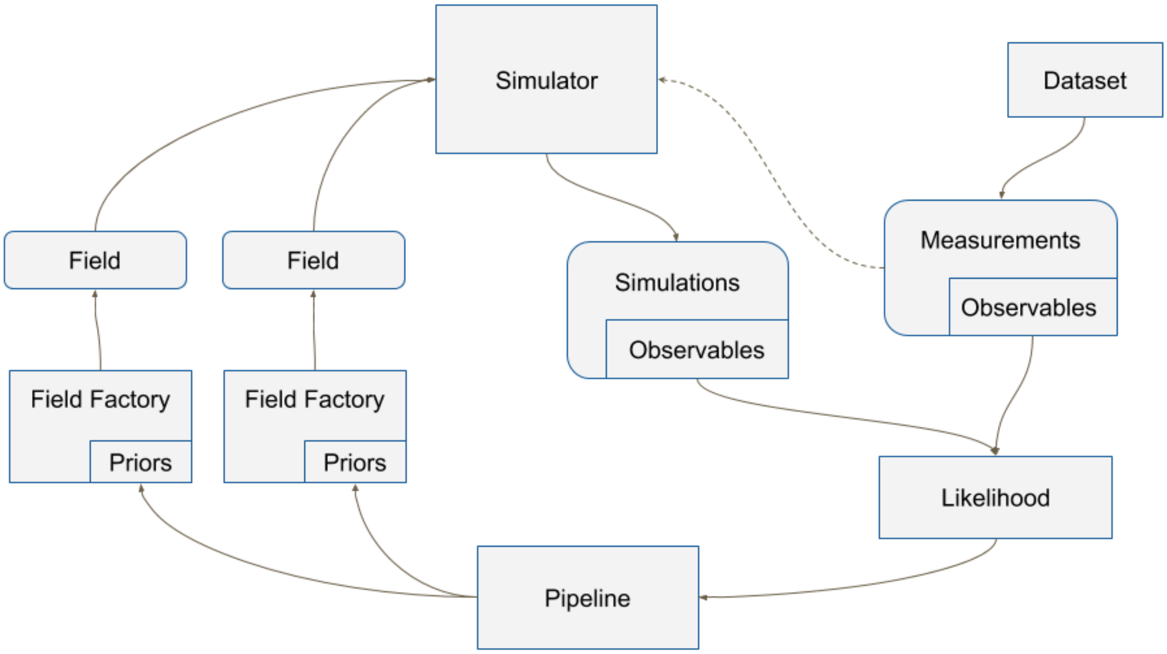


Fig. 2.1: The structure of the IMAGINE pipeline.

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, default values (in case of inactive parameters) and *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.Field` — and if 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.Field` 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 in the `PARAMETER_NAMES` field class attribute (see the templates below for examples).

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.UniformGrid` that accompanies IMAGINE. For example, one can create a grid where the cylindrical coordinates are equally spaced using:

```
from imagine.fields import UniformGrid
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')
```

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 `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.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'

```

(continues on next page)

(continued from previous page)

```
    # 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 = []
    # List of all parameters for the field
    PARAMETER_NAMES = ['Parameter_A', 'Parameter_B']

    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.ThermalElectronDensityField` can be seen bellow:

```
from imagine.fields import ThermalElectronDensityField

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

    NAME = 'exponential_disc_thermal_electrons'
    PARAMETER_NAMES = ['central_density',
                       'scale_radius',
                       'scale_height']

    def compute_field(self, seed):
```

(continues on next page)

(continued from previous page)

```
        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))
```

3.1.3 Magnetic Fields

One can add a new model for magnetic fields subclassing `imagine.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 = []
    # List of all parameters for the field
    PARAMETER_NAMES = ['Parameter_A', 'Parameter_B']

    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.fields import MagneticField

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

    NAME = 'constant_B'
    PARAMETER_NAMES = ['Bx', 'By', 'Bz']

    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']
        # Computes By
        B[:, :, :, 1] = self.parameters['By']
        # Computes Bz
        B[:, :, :, 2] = self.parameters['Bz']
        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.DummyField` as shown bellow.

```python
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).

3.2 Field Factories

Field Factories, represented in IMAGINE by a `imagine.fields.field_factory.FieldFactory` object 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 (active) 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 `Field`), one can easily construct a *FieldFactory* object following:

```
from imagine import FieldFactory
my_factory = FieldFactory(field_class=YourFieldClass,
                          grid=your_grid,
                          active_parameters=['param_1_active'],
                          default_parameters = {'param_2_inactive': value_2,
                                                'param_3_inactive': value_3},
                          priors={'param_1_active': YourPriorChoice})
```

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

Since re-using a FieldFactory associated with a given Field is not uncommon, it is sometimes convenient to create a specialized subclass for a particular field, where the typical/recommended choices of parameters and priors are saved. This can be done following the template below:

```
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(xmin=A_min, xmax=A_max),
              'Parameter_B': GaussianPrior(mu=B_std_val, sigma=B_sig)}
```

One can initialize this specialized FieldFactory subclass by supplying the grid on which the corresponding Field will be evaluated:

```
myFactory = FieldFactoryTemplate(grid=cartesian_grid)
```

The standard values defined in the subclass can be adjusted in the instance using the attributes `myFactory.active_parameters`, `myFactory.priors` and `myFactory.default_parameters` (note that the latter are dictionaries, but the attribution = is modified so that it *updates* the corresponding internal dictionaries instead of replacing them).

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

3.3 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*.

3.3.1 Repository datasets

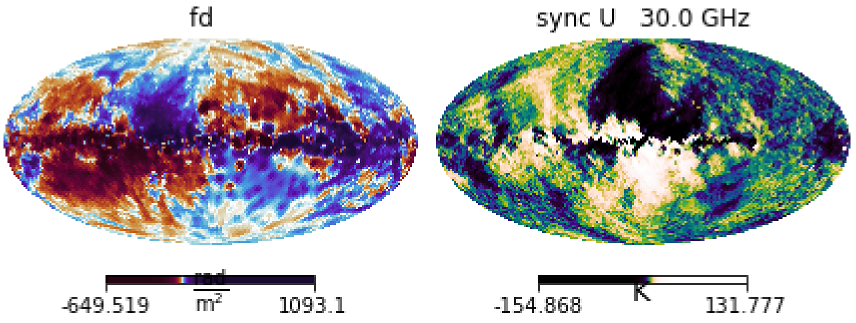
A number of ready-to-use datasets are available at the community maintained imagine-datasets repository.

To be able to use this, it is necessary to install the *imagine_datasets* extension package (note that this comes already installed if you are using the *docker* image). This can be done executing the following command:

```
conda activate imagine  # if using conda
pip install git+https://github.com/IMAGINE-Consortium/imagine-datasets.git
```

Below the usage of an imported dataset is illustrated:

```python
import imagine as img
import imagine_datasets as img_data

# Loads the datasets (will download the datasets!)
dset_fd = img_data.HEALPix.fd.Oppermann2012(Nside=32)
dset_sync = img_data.HEALPix.sync.Planck2018_Commander_U(Nside=32)

# Initialises ObservableDict objects
measurements = img.observables.Measurements(dset_fd, dset_sync)

# Shows the contents
measurements.show()
```



3.3.2 Observable names

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

Observable names

- ‘fd’ - Faraday depth
- ‘dm’ - Dispersion measure
- ‘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

3.3.3 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_col='RM', err_col='e_RM', tag=None,
                            lat_col='GLAT', lon_col='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.

3.3.4 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 and third examples, 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* (the error is assumed to correspond to the square root of the variance in each datapoint).

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 also requires the user to supply the frequency of the observation and the subtype (in this case, 'Q').

3.4 Observables and observable dictionaries

In IMAGINE, observable quantities (either measured or simulated) are represented internally by instances of the `imagine.observables.Observable` class. These are grouped in *observable dictionaries* (subclasses of `imagine.observables.ObservableDict`) which are used to exchange multiple observables between IMAGINE's components. There three main kinds of observable dictionaries: *Measurements*, *Simulations*, and *Covariances*. There is also an auxiliary observable dictionary: the *Masks*.

3.4.1 Measurements

The `imagine.observables.Measurements` object is used, as the name implies, to hold a set of actual measured physical datasets (e.g. a set of intensity maps of the sky at different wavelengths).

A `Measurements` object must be provided to initialize *Simulators* (allowing them to know which datasets need to be computed) and *Likelihoods*.

There are a number of ways data can be provided to `Measurements`. The simplest case is when the data is stored in a `Datasets` objects, one can provide them to the measurements object upon initialization:

```
measurements = img.observables.Measurements(dset1, dset2, dset3)
```

One can also append a dateset to a already initialized `Measurements` object:

```
measurements.append(dataset=dset4)
```

The final (not usually recommended) option is appending the data manually to the *ObservableDict*, which can be done in the following way:

```
measurements.append(name=key, data=data,
                    cov_data=cov, otype='HEALPix')
```

In this, `data` should contain a `ndarray` or `Quantity`, `cov_data` should contain the covariance matrix, and `otype` must indicate whether the data corresponds to: a 'HEALPix' map; 'tabular'; or a 'plain' array.

The `name` argument refers to the key that will be used to hash the elements in the dictionary. This has to have the following form:

```
key = (data_name, data_freq, Nside_or_str, tag)
```

- The first value should be the one of the *observable names*
- If data is independent from frequency, `data_freq` is set to `None`, otherwise it is the frequency in GHz.
- The third value in the key-tuple should contain the HEALPix Nside (for maps) or the string 'tab' for tabular data.
- Finally, the last value, ext can be 'I', 'Q', 'U', 'PI', 'PA', `None` or other customized tags depending on the nature of the observable.

The contents of a `Measurements` object can be accessed as a dictionary, using the keys with the above structure:

```
observable = measurements[('sync', 30, 32, 'I')]
```

Assuming that this key is present, the `observable` object returned by the above line will be an instance of the `Observable` class. The data contents and properties can be then accessed using its properties/attributes, for example:

```
data_array = observable.global_data
data_units = observable.unit
```

Where the `data_array` will be a (1, N)-array.

`Measurements`, support the `show` method (exemplified in *Repository datasets*) which displays a summary of the data in the `ObservableDict`.

3.4.2 Simulations

The `Simulations` object is the `ObservableDict` that is returned when one runs an IMAGINE *Simulator*.

In the case of a `Simulations` object, `max_realizations` keyword argument can be used to limit the number of ensemble realisations that are shown (one realisation per line).

3.4.3 Covariances

`imagine.observables.Covariances`

3.4.4 Masks

`imagine.observables.Masks`

3.5 Simulators

Simulators are reponsible for mapping a set of *Fields* onto a set of *Observables*. They are represented by a subclass of `imagine.simulators.Simulator`.

```
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
```

(continues on next page)

(continued from previous page)

```
    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

    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 allow _dummy fields_ to send specific information to
        # simulators about specific parameters
        checklist_params = self.field_checklist
        # 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,
```

(continues on next page)

(continued from previous page)

```
                                         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)

        # 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
```

3.5.1 Hammurabi simulator

3.6 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.

3.7 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 Factories*. 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 $1\mu\mathrm{G}$ and standard deviation $5\mu\mathrm{G}$. Moreover, let us assumed that the model only works within the range $[-30\mu\mathrm{G}, 30\mu\mathrm{G}]$. 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])
```

3.8 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:

```python
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
    # 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']
```

(continues on next page)

(continued from previous page)

```
        return self.results

    def get_intermediate_results(self):
        # If the sampler saves intermediate results on disk or internally,
        # these should be read here, so that a progress report can be produced
        # every `pipeline.n_evals_report` likelihood evaluations.
        # For this to work, the following should be added to the
        # `intermediate_results` dictionary (currently commented out):

        ## Sets current rejected/dead points, as a numpy array of shape (n, npar)
        #self.intermediate_results['rejected_points'] = rejected
        ## Sets likelihood value of *rejected* points
        #self.intermediate_results['logLikelihood'] = likelihood_rejected
        ## Sets the prior volume/mass associated with each rejected point
        #self.intermediate_results['lnX'] = rejected_data[:, nPar+1]
        ## Sets current live nested sampling points (optional)
        #self.intermediate_results['live_points'] = live

        pass
```

3.9 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:

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

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
     from astropy.table import Table
     import astropy.units as u
     import astropy as apy
     import corner, os
     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 $[0, 2\pi]\,\mathrm{kpc}$.

[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' : u.Quantity(fd, u.microgauss*u.cm**-3),
              '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
      uG / cm3
      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]:

```
mockDataset = img.observables.TabularDataset(data, name='test',
                                             data_col='meas',
                                             err_col='err')
```

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_col` argument specifies the key or name of the column containing the relevant measurement. Coordinates (`coords_type`) can be given in either `'cartesian'` or `'galactic'`. If not provided, the coordinates type is derived from the provided data. In this example, we provided x, y and z in `kpc` and therefore the coordinates type is assumed to be 'cartesian'. The units of the dataset are represented using astropy.units objects and can be supplied (if they are, 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 a `Measurements` object, which is subclass of `ObservableDict`. This object allows one to supply multiple datasets to the pipeline.

[4]:
```
# Create Measurements object using mockDataset
mock_data = img.observables.Measurements(mockDataset)
```

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

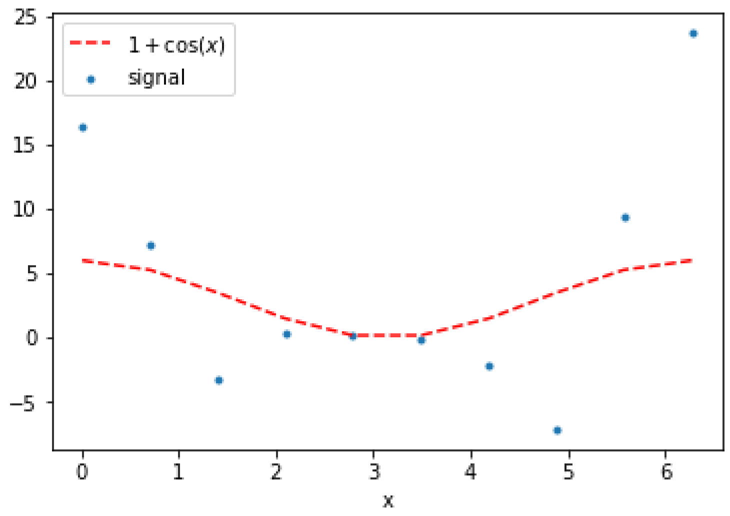
[5]:
```
keys = list(mock_data.keys())
keys
```

[5]:
```
[('test', None, 'tab', None)]
```

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.

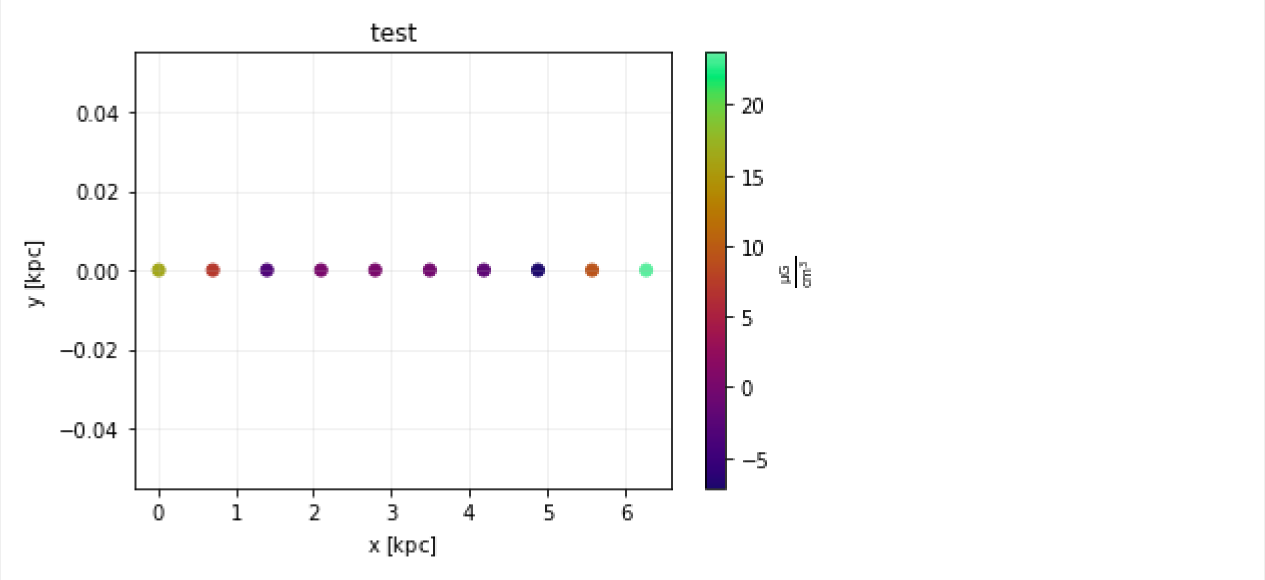
[6]:
```
plt.scatter(x, mock_data[keys[0]].global_data[0], marker='.', label='signal')
plt.plot(x,(1+np.cos(x))*a0,'r--',label='$1+\cos(x)$')
plt.xlabel('x'); plt.legend();
```



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.

IMAGINE also comes with a built-in method for showing the contents of a Measurements objects on a skymap. In the next cell this is exemplified, though not very useful for this specific example.

[7]:
```
mock_data.show(cartesian_axes='xy')
```



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.

[8]:
```
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.

[9]:
```
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:

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

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

```
[10]: 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
```

```
[10]: {'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>}
```

```
[11]: ne_factory.active_parameters
```

```
[11]: ()
```

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*).

```
[12]: 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:

```
[13]: B_factory.active_parameters = ('a0','b0')
      B_factory.priors ={'a0': img.priors.FlatPrior(xmin=-4*u.microgauss,
                                                     xmax=5*u.microgauss),
                         'b0': img.priors.FlatPrior(xmin=2*u.microgauss,
                                                     xmax=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.

```
[14]: 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 initialized with the mock Measurements defined before, which allows it to know what is the correct format for output.

[15]:
```
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.

[16]:
```
likelihood = img.likelihoods.EnsembleLikelihood(mock_data)
```

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 MultiNest 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.

[17]:
```
pipeline = img.pipelines.MultinestPipeline(simulator=simer,
                                           run_directory='../runs/tutorial_one',
                                           factory_list=factory_list,
                                           likelihood=likelihood,
                                           ensemble_size=27)
```

The `run_directory` keyword is used to setup where the state of the pipeline is saved (allowing loading the pipeline in the future). It is also where the chains generated by the sampler are saved in the sampler's native format (if the sampler supports this).

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 = {'evidence_tolerance': 0.5, 'n_live_points': 200}
```

In a standard *nested sampling approach*, a set of "live points" is initially sampled from the prior distribution. After each iteration the point with the smallest likelihood is removed (it becomes a *dead point*, and its likelihood value is stored) and a new point is sampled from the prior. As each dead point is associated to some prior volume, they can be used to estimate the evidence (see, e.g. here for details). In the `MultinestPipeline`, the number of live points is set using the `'n_live_points'` sampling controller.

The sampling parameter `'evidence_tolerance'` allows one to control the target error in the estimated evidence.

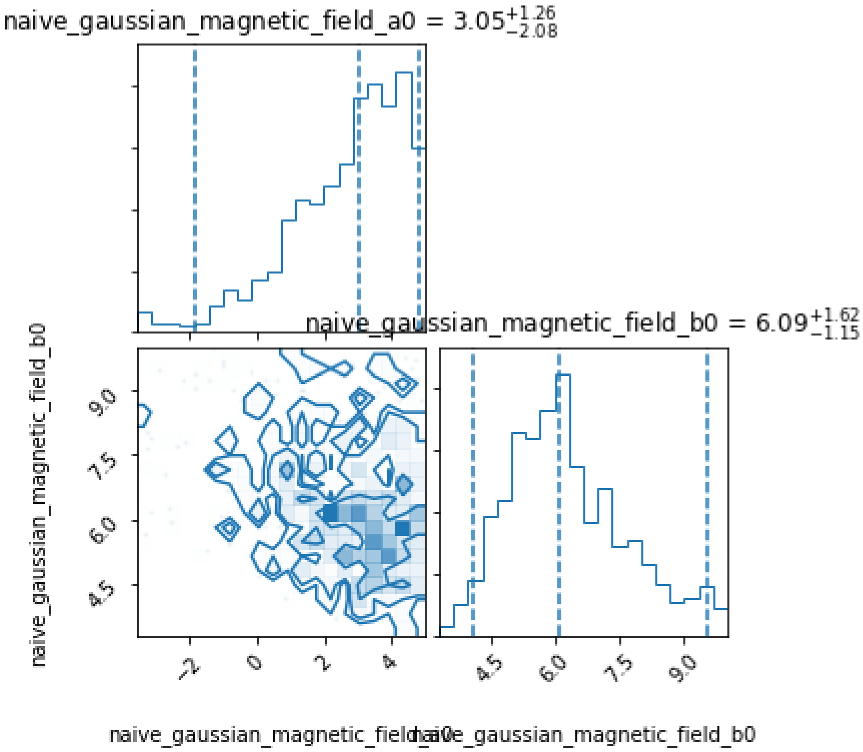
Now, we *finally* can run the pipeline!

[19]:

```
results = pipeline()
```

```
  analysing data from ../runs/tutorial_one/chains/multinest_.txt
```

Posterior report:



Evidence report:

$\log\mathcal{Z} = -33.74 \pm 0.07$

Thus, one can see that after the pipeline finishes running, a brief summary report is written to screen.

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*) $\mathcal{Z}$. 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: -33.7403
log evidence error: 0.0686
```

A dictionary containing a summary of the *constraints to the parameters* can be accessed through the property `posterior_summary`:

[21]:

```
pipeline.posterior_summary
```

[21]:

```
{'naive_gaussian_magnetic_field_a0': {'median': <Quantity 3.04653036 uG>,
  'errlo': <Quantity 2.09241063 uG>,
  'errup': <Quantity 1.26013795 uG>,
  'mean': <Quantity 2.62797986 uG>,
```

(continues on next page)

(continued from previous page)

```
  'stdev': <Quantity 1.73790351 uG>},
 'naive_gaussian_magnetic_field_b0': {'median': <Quantity 6.09011278 uG>,
  'errlo': <Quantity 1.15964546 uG>,
  'errup': <Quantity 1.61955049 uG>,
  'mean': <Quantity 6.2845587 uG>,
  'stdev': <Quantity 1.37869584 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:

[22]:

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

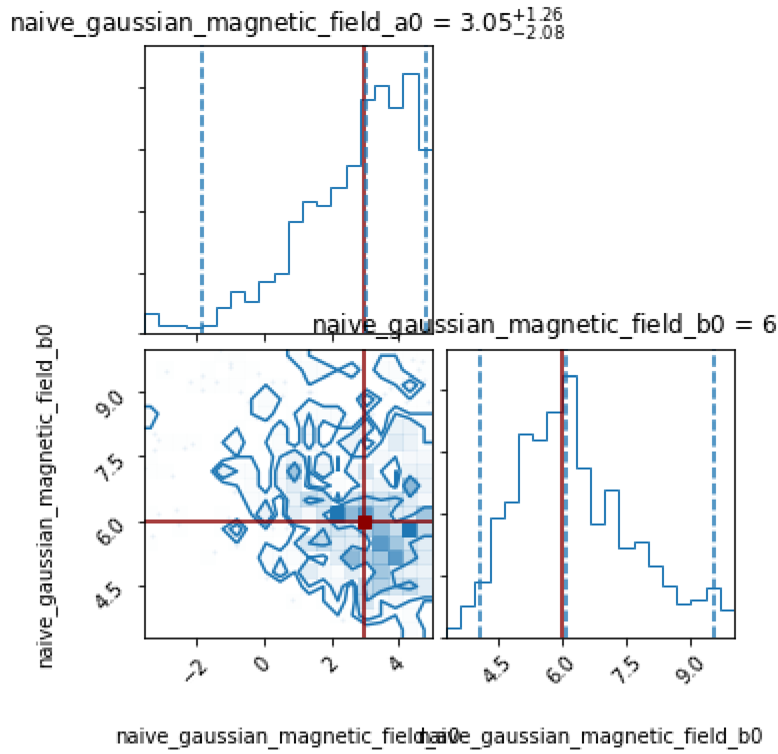
[22]:

```
<QTable length=3>
naive_gaussian_magnetic_field_a0 naive_gaussian_magnetic_field_b0
               uG                               uG
            float64                          float64
-------------------------------- --------------------------------
              -3.3186057209968567                5.630247116088867
               -3.163660407066345                6.178023815155029
                4.913007915019989                3.289425849914551
```

For convenience, the corner plot showing the posterior distribution obtained from the samples can be generated using the `corner_plot` method of the `Pipeline` object (which uses the corner library). This can show "truth" values of the parameters (in case someone is doing a test like this one).

[23]:

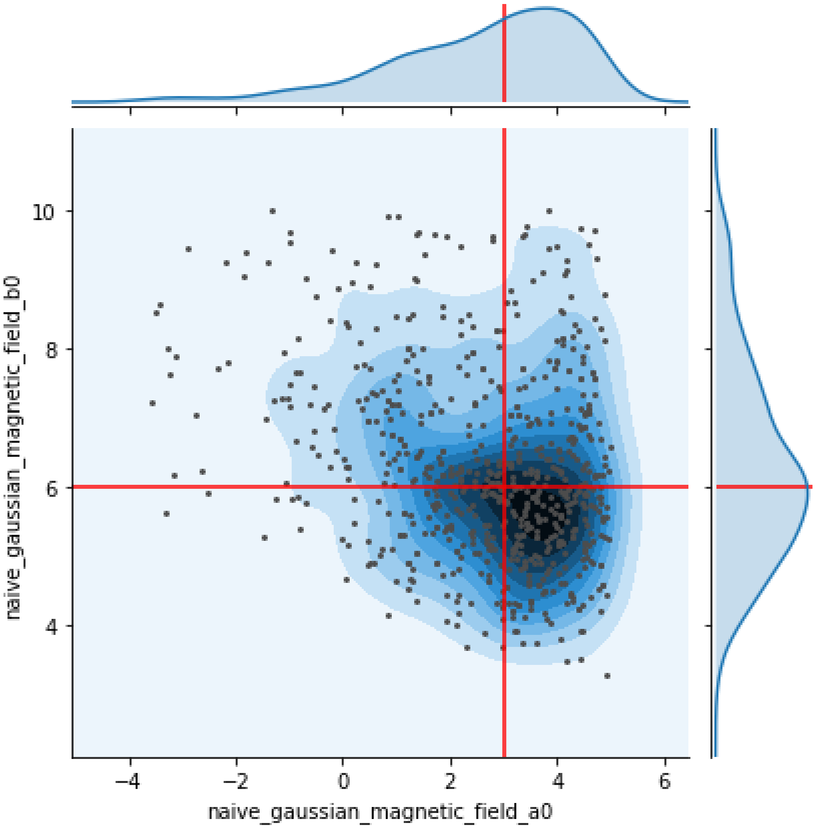
```
pipeline.corner_plot(truths_dict={'naive_gaussian_magnetic_field_a0': 3,
                                  'naive_gaussian_magnetic_field_b0': 6});
```



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

[24]:
```
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)
```

[25]:
```
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.

```
[26]: pipeline.master_seed
```

```
[26]: 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), and the random seeds which are used for generating the ensembles of stochastic fields are drawn in the beginning of the pipeline run;
- 'free' - on each execution, a new `master_seed` is drawn (using numpy.randint), moreover: at *each evaluation of the likelihood* the stochastic fields receive a new set of ensemble seeds;
- '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.

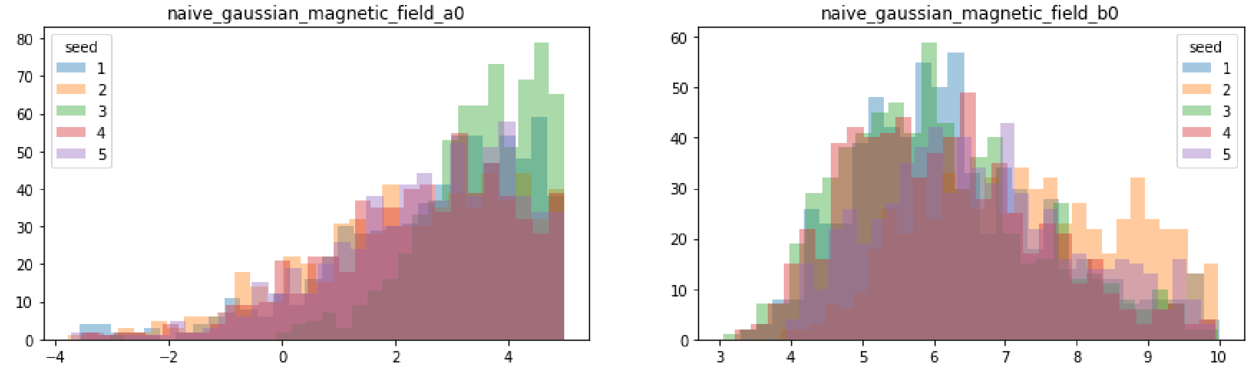
```
[27]: fig, axs = plt.subplots(1, 2, figsize=(15, 4))
repeat = 5
pipeline.show_summary_reports = False  # Avoid summary reports
pipeline.sampling_controllers['resume'] = False # Pipeline will re-run
samples_list = []
for i in range(1,repeat+1):
    print('-'*60+'\nRun {}/{}'.format(i,repeat))
    if i>1:
        # Re-runs the pipeline with a different seed
        pipeline.master_seed = i
        _ = pipeline()
    print('log Z = ', round(pipeline.log_evidence,4),
          '±', round(pipeline.log_evidence_err,4))

    samples = pipeline.samples

    for j, param in enumerate(pipeline.samples.columns):
        samp = samples[param]
        axs[j].hist(samp.value, alpha=0.4, bins=30, label=pipeline.master_seed)
        axs[j].set_title(param)
    # Stores the samples for later use
    samples_list.append(samples)

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

```
------------------------------------------------------------
Run 1/5
log Z =  -33.7403 ± 0.0686
------------------------------------------------------------
Run 2/5
  analysing data from ../runs/tutorial_one/chains/multinest_.txt
log Z =  -36.422 ± 0.0599
------------------------------------------------------------
```

(continues on next page)

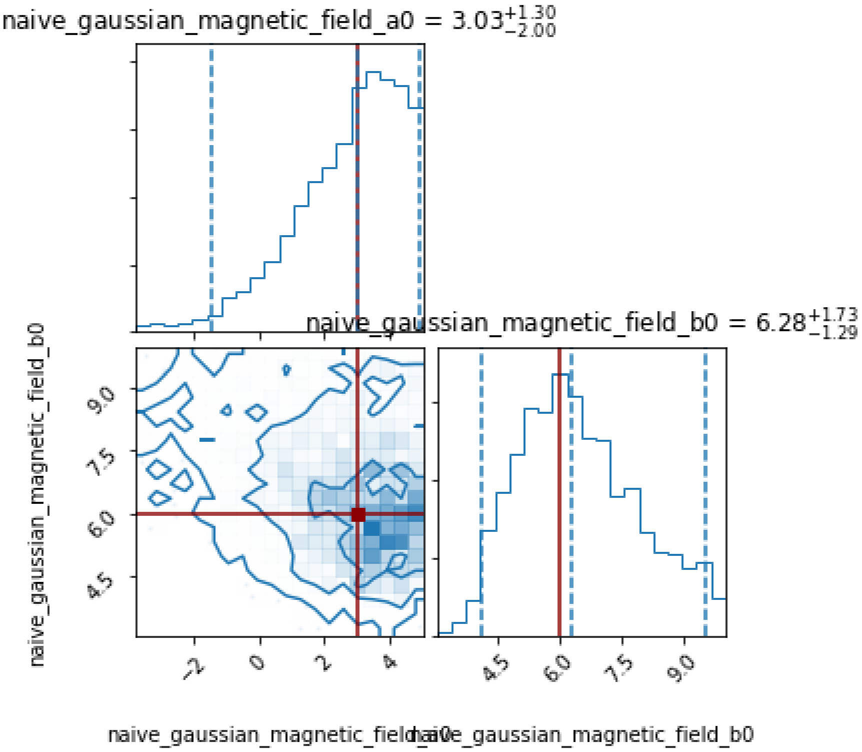
(continued from previous page)

```
Run 3/5
  analysing data from ../runs/tutorial_one/chains/multinest_.txt
log Z =  -33.593 ± 0.0788
------------------------------------------------------------
Run 4/5
  analysing data from ../runs/tutorial_one/chains/multinest_.txt
log Z =  -33.329 ± 0.0636
------------------------------------------------------------
Run 5/5
  analysing data from ../runs/tutorial_one/chains/multinest_.txt
log Z =  -34.6659 ± 0.0663
```



After being satisfied, one may want to put together all the samples. A set of rigorous tools for combining multiple pipelines/runs is *currently under development*, but a first order approximation to this can be seen below:

[28]:
```
all_samples = apy.table.vstack(samples_list)
img.tools.corner_plot(table=all_samples,
                      truths_dict={'naive_gaussian_magnetic_field_a0': 3,
                                   'naive_gaussian_magnetic_field_b0': 6});
```



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
import healpy as hp

# 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 smaller, to save memory
fd_raw = hp.pixelfunc.ud_grade(fd_raw, 4)
sigma_fd_raw = hp.pixelfunc.ud_grade(sigma_fd_raw, 4)
# 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]: from imagine.observables import FaradayDepthHEALPixDataset

class FaradayDepthOppermann2012(FaradayDepthHEALPixDataset):
    def __init__(self, Nside=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)
        # Reduces the resolution
        if Nside is not None:
            fd_raw = hp.pixelfunc.ud_grade(fd_raw, Nside)
            sigma_fd_raw = hp.pixelfunc.ud_grade(sigma_fd_raw, Nside)
        # Includes units in the data
        fd_raw *= u.rad/u.m/u.m
        sigma_fd_raw *= u.rad/u.m/u.m
        # 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(Nside=32)
```

In fact, this dataset is part of the imagine-datasets repository and can be immediately accessed using:

```
import imagine_datasets as img_data
dset = img_data.HEALPix.fd.Oppermann2012(Nside=32)
```

One of the advantages of using the datasets in the `imagine_datasets` repository is that they are cached to the hard disk and are only downloaded on the first time they are requested.

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

[5]:

```
from imagine.observables import Measurements, Covariances

# Creates an instance
mea = Measurements()

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

If the dataset contains error or covariance data, its inclusion in a `Measurements` object automatically leads to the inclusion of such dataset in an associated `Covariances` object. This can be accessed through the `cov` attribute:

[6]:

```
mea.cov
```

[6]:

```
<imagine.observables.observable_dict.Covariances at 0x7ffaaf2f1d10>
```

An alternative, rather useful, supported syntax is providing the datasets as one initializes the `Measurements` object.

[7]:

```
# Creates _and_ appends
mea = Measurements(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.

[8]:

```
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
```

[8]:

```
<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:

[9]:

```python
from imagine.observables import TabularDataset
dset_tab = TabularDataset(catalog, name='fd', tag=None,
                          units= u.rad/u.m/u.m,
                          data_col='RM', err_col='e_RM',
                          lat_col='GLAT', lon_col='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:

[10]:

```python
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_col='RM', err_col='e_RM',
                         lat_col='GLAT', lon_col='GLON')
```

[11]:

```python
dset_tab = FaradayRotationMao2010() # ta-da!
```

8.3 Measurements and Covariances

Again, we can include these observables in our `Measurements` object. This is a dictionary-like object, i.e. given a key, one can access a given item.

[12]:

```python
mea.append(dataset=dset_tab)

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

```
Measurement keys:
         ('fd', None, 32, None)
         ('fd', None, 'tab', None)
```

Associated with the `Measurements` objects there is a `Covariances` object which stores the covariance matrix associated with each entry in the `Measurements`. This is also an `ObservableDict` subclass and has, therefore, similar behaviour. If the original `Dataset` contained error information but no full covariance data, a diagonal covariance matrix is assumed. (N.B. correlations between different observables – i.e. different entries in the `Measurements` object – are still not supported.)

```
[13]: print('Covariances dictionary', mea.cov)
      print('\nCovariance keys:')
      for k in mea.cov.keys():
          print('\t',k)
```

```
Covariances dictionary <imagine.observables.observable_dict.Covariances object at␣
↪0x7ffaaf00d750>

Covariance keys:
         ('fd', None, 32, None)
         ('fd', None, 'tab', None)
```

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

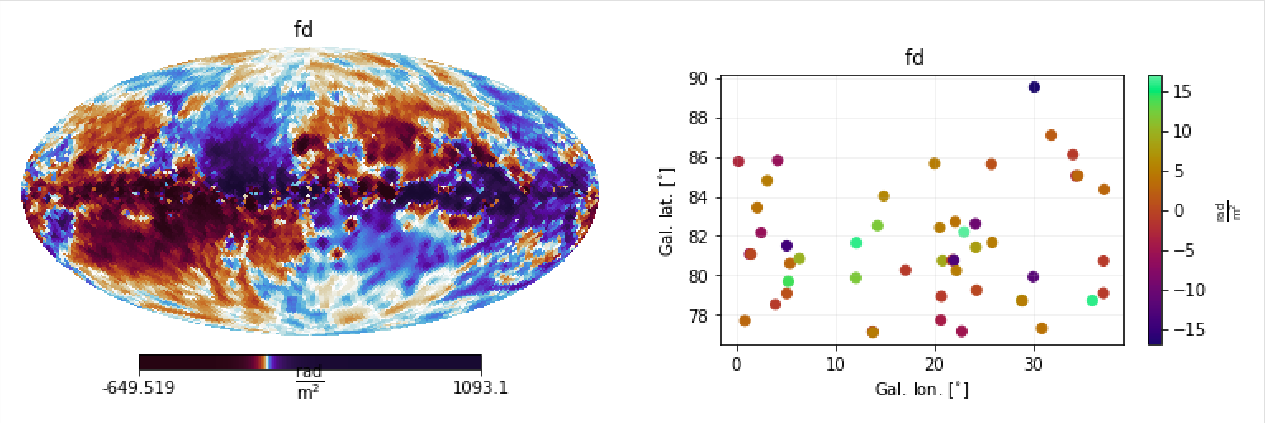
- If data is independent from frequency, data-freq is set `None`, otherwise it is the frequency in GHz.
- The third value in the key-tuple is the HEALPix Nside (for maps) or the string 'tab' for tabular data.
- Finally, the last value, `ext` can be 'I','Q','U','PI','PA', None or other customized tags depending on the nature of the observable.

8.3.1 Accessing and visualising data

Frequently, one may want to see what is inside a given Measurements object. For this, one can use the handy helper method `show()`, which automatically displays all the contents of your `Measurements` object.

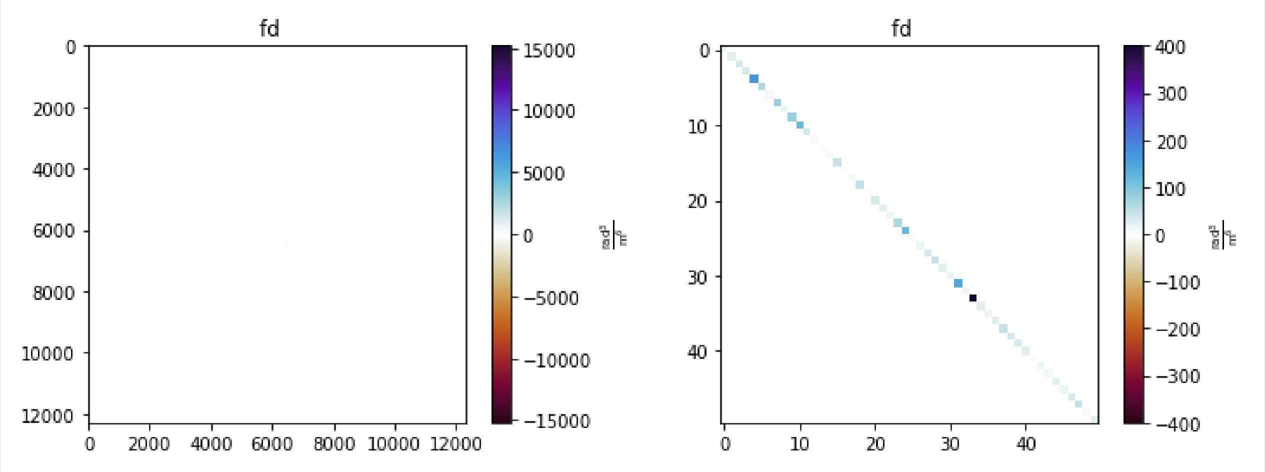
```
[14]: import matplotlib.pyplot as plt
      plt.figure(figsize=(12,3))

      mea.show()
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:209: MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax␣
↪simultaneously is deprecated since 3.3 and will become an error two minor releases␣
↪later. Please pass vmin/vmax directly to the norm when creating it.
  **kwds
```



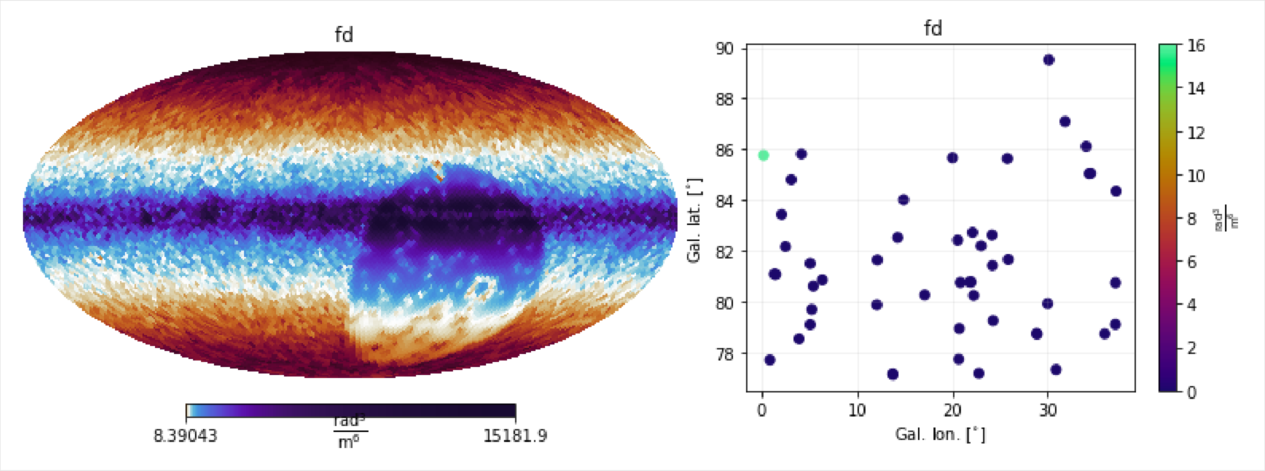
One may also be interested in visualising the associated covariance matrices, which in this case are diagonals, since the `Dataset` objects were initialized using the `error` keyword argument.

[15]:
```
plt.figure(figsize=(12,4))
mea.cov.show()
```



It is also possible to show only the variances (i.e. the diagonals of the covariance matrices). This is plotted as maps, to aid the interpretation

[16]:
```
plt.figure(figsize=(12,4))
mea.cov.show(show_variances=True)
```



Finally, to directly access the data, one needs first to find out what the keys are:

[17]:
```
list(mea.keys())
```

[17]:
```
[('fd', None, 32, None), ('fd', None, 'tab', None)]
```

and use them to get the data using the `global_data` property

[18]:
```
my_key = ('fd', None, 32, None)
extracted_obs_data = mea[my_key].global_data

print(type(extracted_obs_data))
print(extracted_obs_data.shape)
```

```
<class 'numpy.ndarray'>
(1, 12288)
```

The property `global_data` automatically gathers the data if `rc['distributed_arrays']` is set to `True`, while the attribute `data` returns the local values. If not using `'distributed_arrays'` the two options are equivalent.

8.3.2 Manually appending data

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

[19]:
```
# Creates a new dataset
dset = FaradayDepthOppermann2012(Nside=2)
# This is how HEALPix data can be included without the mediation of Datasets:
cov = np.diag(dset.var)  # Covariance matrix from the variances
mea.append(name=dset.key, data=dset.data, cov_data=cov, otype='HEALPix')
# This is what Dataset is doing:
print('The key used in the "name" arg was:', dset.key)
print('The shape of data was:', dset.data.shape)
print('The shape of the covariance matrix arg was:', cov.shape)
```

```
The key used in the "name" arg was: ('fd', None, 2, None)
The shape of data was: (1, 48)
The shape of the covariance matrix arg was: (48, 48)
```

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

[20]:
```
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('mea.otype:', mea[dset_tab.key].otype)
print('\n\nmea.cov type',type(mea.cov[dset_tab.key]))
print('mea.cov.data type:', type(mea.cov[dset_tab.key].data))
print('mea.cov.data.shape:', mea.cov[dset_tab.key].data.shape)
print('mea.cov.dtype:', mea.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
mea.otype: tabular


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

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]:
```
import imagine as img
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 = img.fields.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 = img.fields.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 = img.fields.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
```

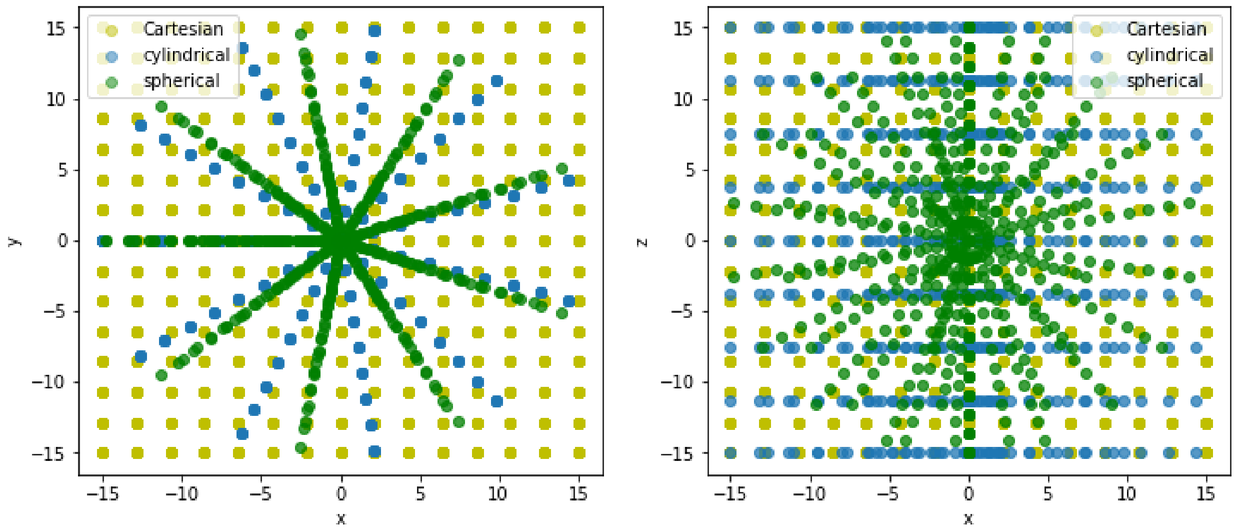
In the following figure we illustrate the effects of different choices of ‘grid_type’ while using UniformGrid.

(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'
    PARAMETER_NAMES = ['central_density', 'scale_radius', 'scale_height']

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

(continues on next page)

(continued from previous page)

```
        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 `NAME` allows one to keep track of which model we have used to generate our field.

The `PARAMETER_NAMES` attribute must contain all the required parameters for this particular kind of field.

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)
```

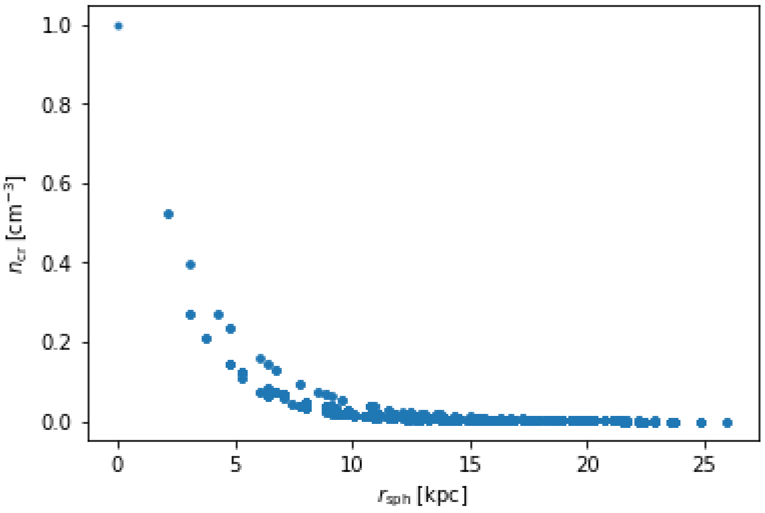
We can access the field produced by `cr_distribution` using the `get_data()` method (it invokes `compute_field` internally and does any checking required). For example:

[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'
    PARAMETER_NAMES = ['Bx', 'By', 'Bz']

    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)
```

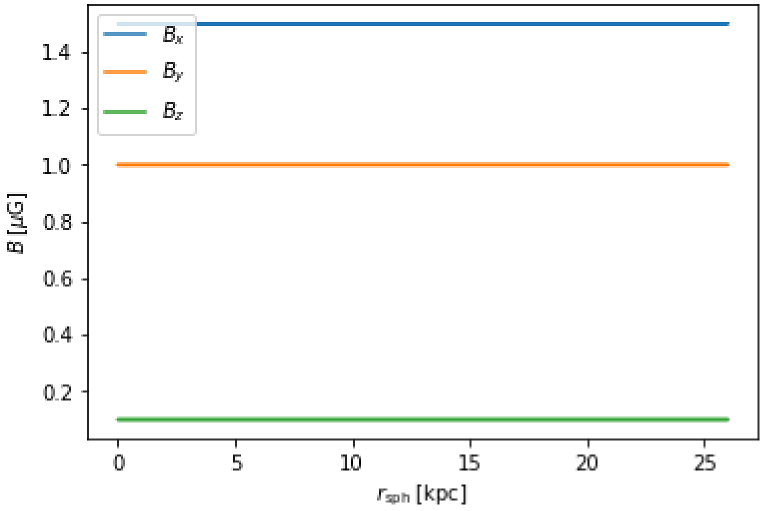
And inspect how it went

[10]:
```
r_spherical = B.grid.r_spherical.ravel()
B_data = B.get_data()
for i, name in enumerate(['x','y','z']):
    plt.plot(r_spherical, B_data[...,i].ravel(),
             label='$B_{}$'.format(name))
```

(continues on next page)

(continued from previous page)

```
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
    PARAMETER_NAMES = ['mean', 'std']

    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
```

(continues on next page)

(continued from previous page)

```
        # 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
```

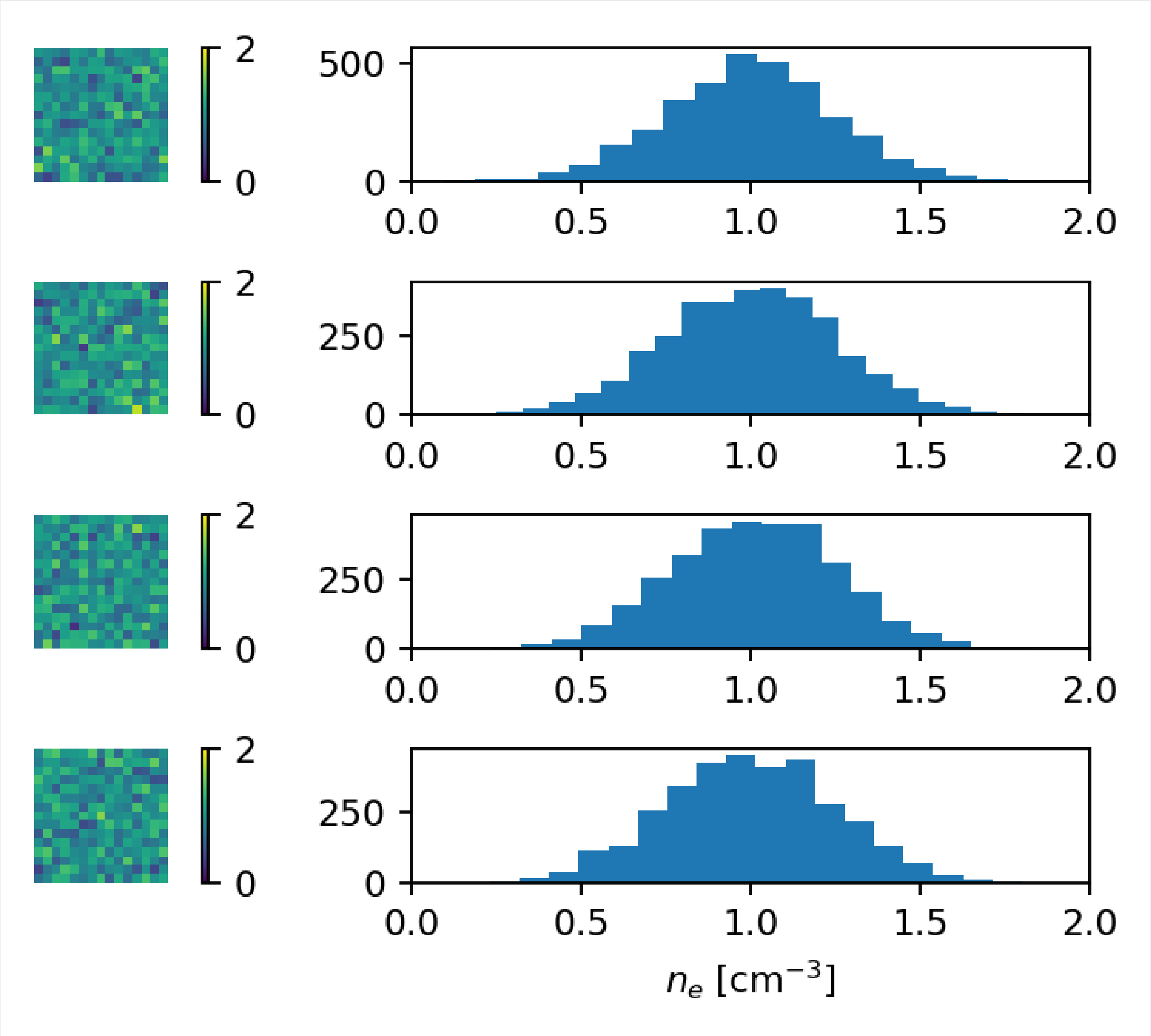
The `STOCHASTIC_FIELD` class-attribute tells whether the field is deterministic (i.e. the output depends only on the parameter values) or stochastic (i.e. the ouput is a random realisation which depends on a particular random seed). If this is absent, IMAGINE assumes the field is deterministic.

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 for dummy fields we do **not** specify `compute_field`,
`STOCHASTIC_FIELD` or `PARAMETER_NAMES`).

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

class exampleDummy(DummyField):
    NAME = 'example_dummy'
    FIELD_CHECKLIST = {'A': None, 'B': 'foo', 'C': 'bar'}
    SIMULATOR_CONTROLLIST = {'lookup_directory': '/dummy/example',
                             'choice_of_settings': ['tutorial','field']}
```

Thus, instead of a `PARAMETER_NAMES`, one needs to specify a `FIELD_CHECKLIST` which contains a dictionary with parameter names as keys. Its main use is to send to the Simulator *fixed settings* associated with a *particular parameter*. For example, the `FIELD_CHECKLIST` 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` attribute 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

[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}
```

Therefore, 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.

9.3 Field factory

Associated with each Field class we need to prepare a FieldFactory object, which, for each parameter, stores ranges, default values and priors.

This is done using the `imagine.fields.FieldFactory` class.

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

exp_te_factory = img.fields.FieldFactory(
    field_class=ExponentialThermalElectrons,
    grid=cartesian_grid,
    active_parameters=[],
    default_parameters = {'central_density': 1*u.cm**-3,
                          'scale_radius': 3.0*u.kpc,
                          'scale_height': 0.5*u.kpc})
```

```
[21]: Bfactory = img.fields.FieldFactory(
    field_class=ConstantMagneticField,
    grid=cartesian_grid,
    active_parameters=['Bx'],
    default_parameters={'By': 5.0*u.microgauss,
                        'Bz': 0.0*u.microgauss},
    priors={'Bx': FlatPrior(xmin=-30*u.microgauss, xmax=30*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:

```
[22]: Bfactory.default_parameters, Bfactory.parameter_ranges, Bfactory.priors
```

```
[22]: ({'By': <Quantity 5. uG>, 'Bz': <Quantity 0. uG>},
 {'Bx': <Quantity [-30.,  30.] uG>},
 {'Bx': <imagine.priors.basic_priors.FlatPrior at 0x7fb3b533da90>})
```

As a user, this is all there is to know about FieldFactories: one need to intialized them with the ones selection of active parameters, priors and default values and provide the initialized objects to the IMAGINE Pipeline class.

Internally, these instances can return Field objects (hence the name) when they are *called* internally by the Pipeline whilst running. This is exemplified below:

```
[23]: newB = Bfactory(variables={'Bx': 0.9*u.microgauss})
print('field name: {}\nparameters: {}'.format(newB.name, newB.parameters))
```

```
field name: constantB
parameters: {'By': <Quantity 5. uG>, 'Bz': <Quantity 0. uG>, 'Bx': <Quantity 0.9 uG>}
```

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 **Gaussian prior** for the parameters `By` and `Bz`, truncated in the latter case. The priors can be included creating an object `GaussianPrior` for which the mean, standard deviation and range are specified:

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

muG = u.microgauss

Bfactory = img.fields.FieldFactory(
    field_class=ConstantMagneticField,
    grid=cartesian_grid,
    active_parameters=['Bx', 'By','Bz'],
    priors={'Bx': FlatPrior(xmin=-30*muG, xmax=30*muG),
            'By': GaussianPrior(mu=5*muG, sigma=10*muG),
            'Bz': GaussianPrior(mu=0*muG, sigma=3*muG,
                                xmin=-30*muG, xmax=30*muG)})
```

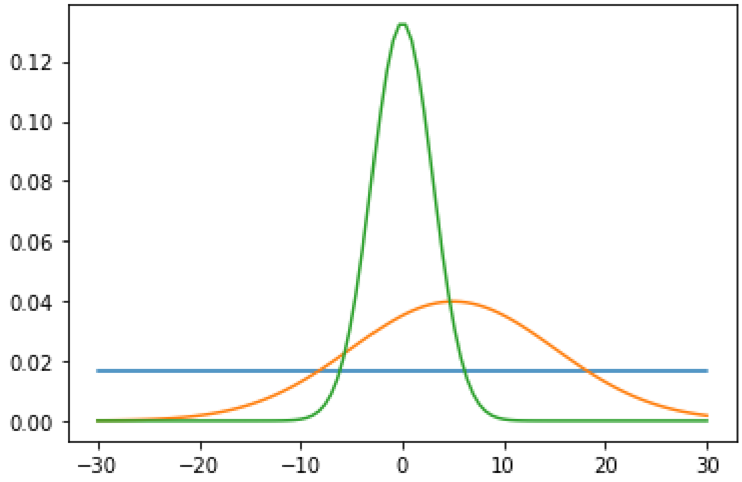
Let us now inspect an instance of updated field factory

```
[25]: Bfactory.priors, Bfactory.parameter_ranges
```

```
[25]: ({'Bx': <imagine.priors.basic_priors.FlatPrior at 0x7fb3b532cc90>,
        'By': <imagine.priors.basic_priors.GaussianPrior at 0x7fb3b533db10>,
        'Bz': <imagine.priors.basic_priors.GaussianPrior at 0x7fb3b5329f50>},
       {'Bx': <Quantity [-30.,  30.] uG>,
        'By': <Quantity [-inf,  inf] uG>,
        'Bz': <Quantity [-30.,  30.] uG>})
```

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

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

[27]:

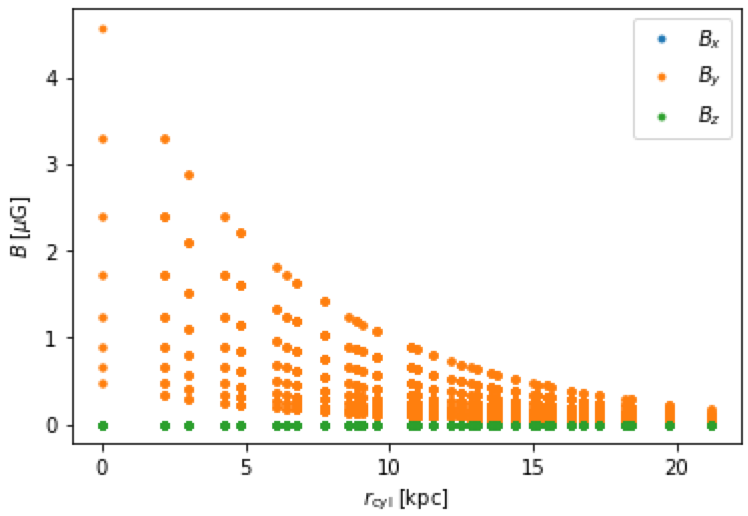
```
import astropy.constants as c
# unfortunatelly, astropy units does not perform the following automatically
gauss_conversion = u.gauss/(np.sqrt(1*u.g/u.cm**3)*u.cm/u.s)

class DependentBFieldExample(MagneticField):
    """Example: By depends on ne"""
    # Class attributes
    NAME = 'By_Beq'
    DEPENDENCIES_LIST = ['thermal_electron_density']
    PARAMETER_NAMES = ['v0']

    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.

[28]:

```
# 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.

[29]:
```
class ConstantElectrons(ThermalElectronDensityField):
    NAME = 'constant_thermal_electron_density'
    PARAMETER_NAMES = ['A']

    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"""
```

(continues on next page)

(continued from previous page)

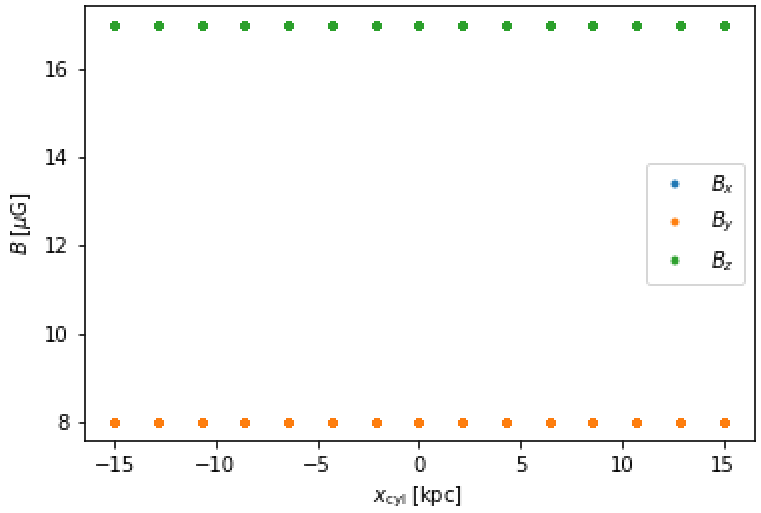
```
    NAME = 'constantBdep'
    DEPENDENCIES_LIST = [ConstantElectrons]
    PARAMETER_NAMES = []

    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.

[30]:
```
# Initializes the instances
ne_field = ConstantElectrons(cartesian_grid, parameters={'A': 64})
dependent_field = ConstantDependentB(cartesian_grid)

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

[31]:
```
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();
```


16
14
12
10
8
B [μG]
−15
−10
−5
0
5
10
15
x_{cyl} [kpc]
B_x
B_y
B_z

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]:

```
import imagine as img
fake_dset = img.observables.TabularDataset(fake_data, name='testRM', units= u.rad/u.m/
↪u.m,
                           data_col='RM', err_col='err',
                           lat_col='lat', lon_col='lon')
# Initializes Measurements
mea = img.observables.Measurements(fake_dset)
mea.keys()
```

[3]:

```
dict_keys([('testRM', None, 'tab', None)])
```

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,
                                           galaxy_longitude=gal_lon,
                                           beam_size=0.700*u.kpc,
                                           orientation='edge-on')
```

(continues on next page)

(continued from previous page)

```
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()))
```

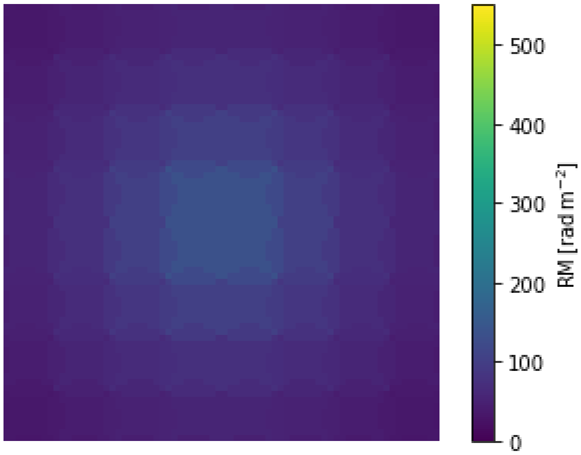
```
faceon_sim: <imagine.observables.observable_dict.Simulations object at 0x7fa2d446ff10>
faceon_sim keys: [('testRM', None, 'tab', None)]
```

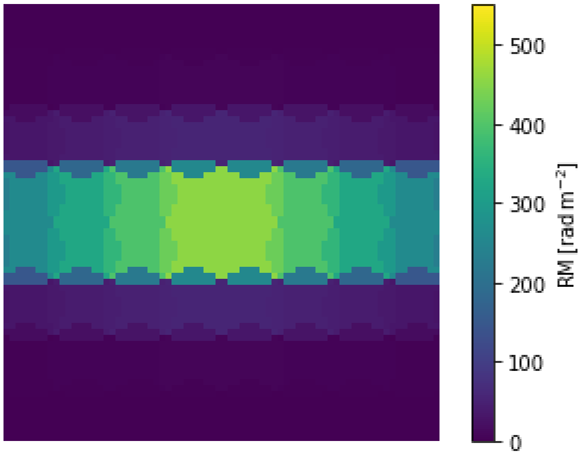
```
[7]: faceon_sim[('testRM', None, 'tab', None)].data.shape
```

```
[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]
# 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,
                                 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,
```

(continues on next page)

(continued from previous page)

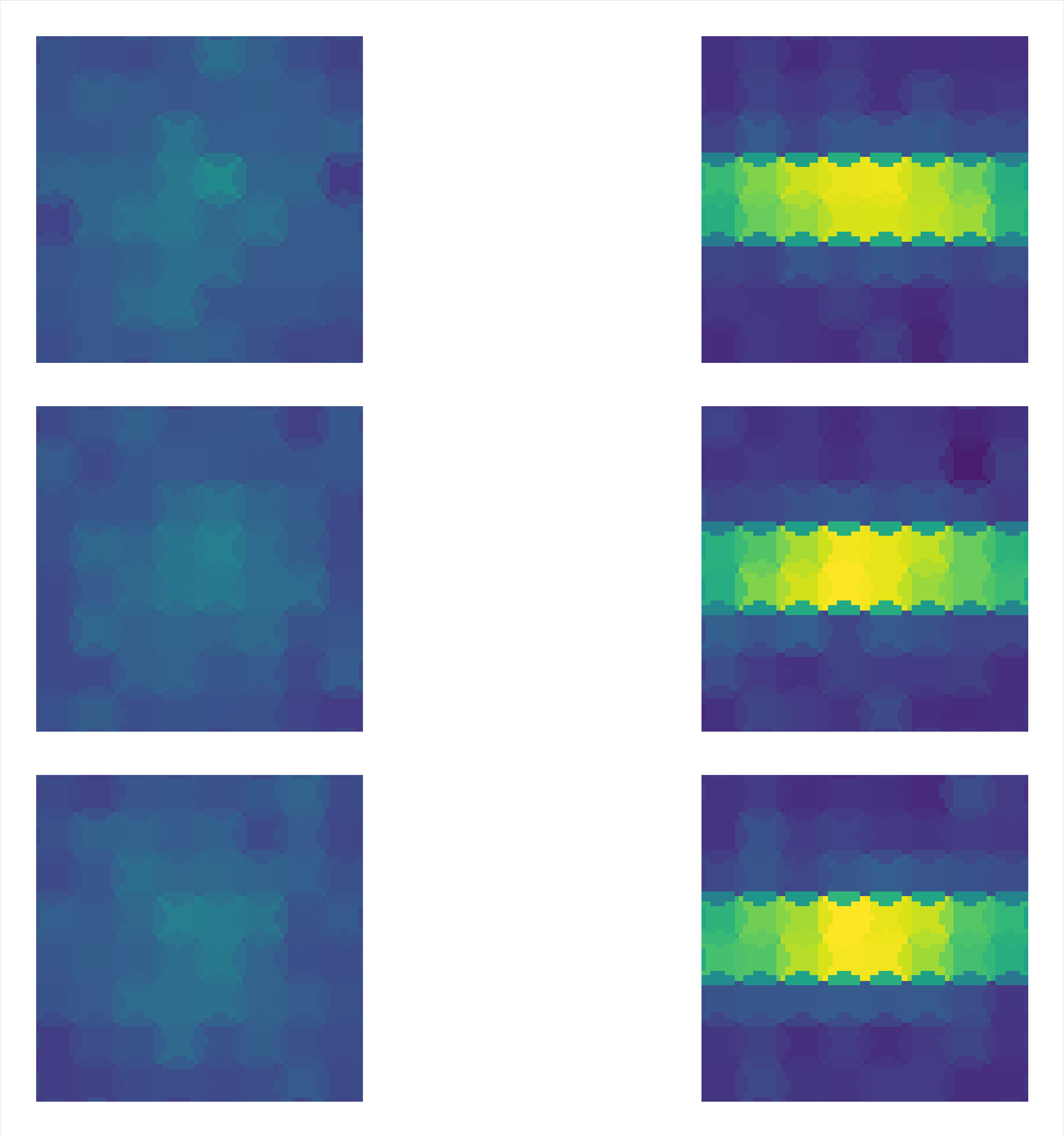
```
                                                   'std': 0.01*u.cm**-3,
                                                   'min_ne' : 0*u.cm**-3})
```

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
import cmasher as cmr
import copy

matplotlib.rcParams['figure.figsize'] = (10.0, 4.5)
```

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,
                                              frequency=23*u.GHz, typ='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(sync_dset, fd_dset, 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}
```

(continues on next page)

(continued from previous page)

```
cre_ana = CREAna(parameters=paramlist_cre, ensemble_size=ensemble_size)

##  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:

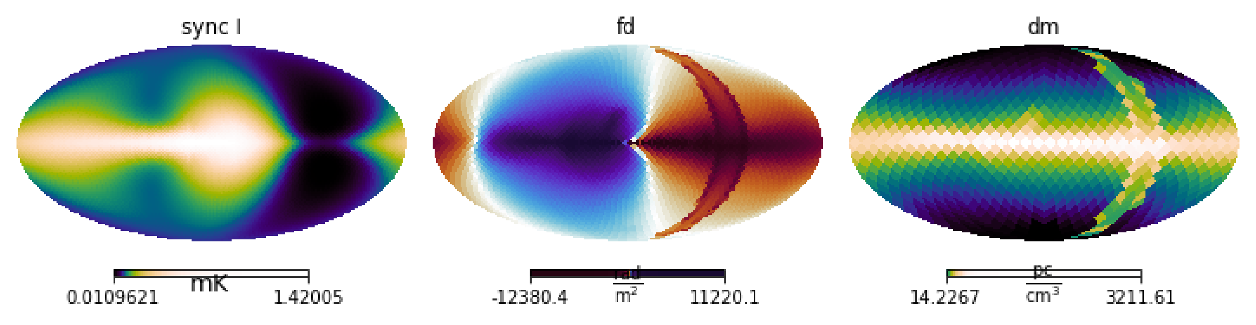
```
[7]: maps[('sync', 23., 32, 'I')].global_data
```

```
[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 (manually) extract and plot the simulated maps produced by Hammurabi:

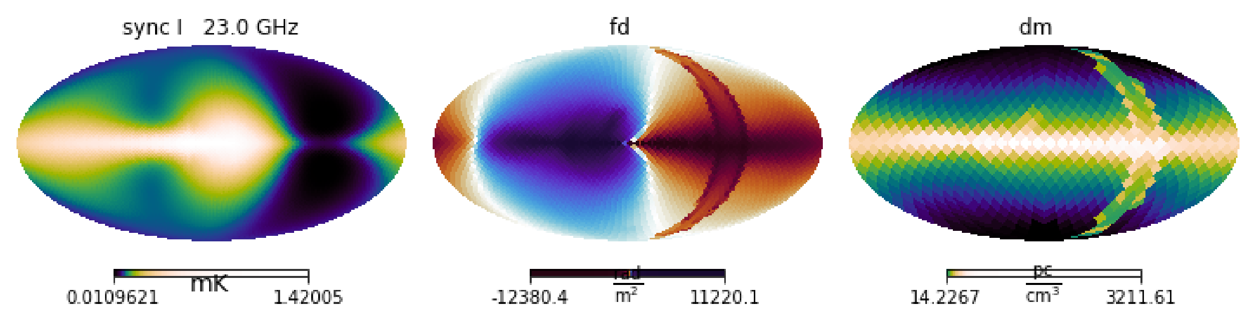
[8]:
```
from imagine.tools.visualization import _choose_cmap

for i, key in enumerate(maps):
    simulated_data = maps[key].global_data[0]
    simulated_unit = maps[key].unit
    name = key[0]
    if key[3] is not None:
        name += ' '+key[3]
    hp.mollview(simulated_data, norm='hist', cmap=_choose_cmap(name),
                unit=simulated_unit._repr_latex_(), title=name, sub=(1,3,i+1))
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:209: MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax␣
↪simultaneously is deprecated since 3.3 and will become an error two minor releases␣
↪later. Please pass vmin/vmax directly to the norm when creating it.
  **kwds
```



Alternatively, we can use a built-in method in the `Simulations` object to show its contents:

[9]:
```
maps.show(max_realizations=1)
```



The keyword argument `max_realizations` limits the number of ensemble realisations that are displayed.

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.

[10]:
```
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.}
```

(continues on next page)

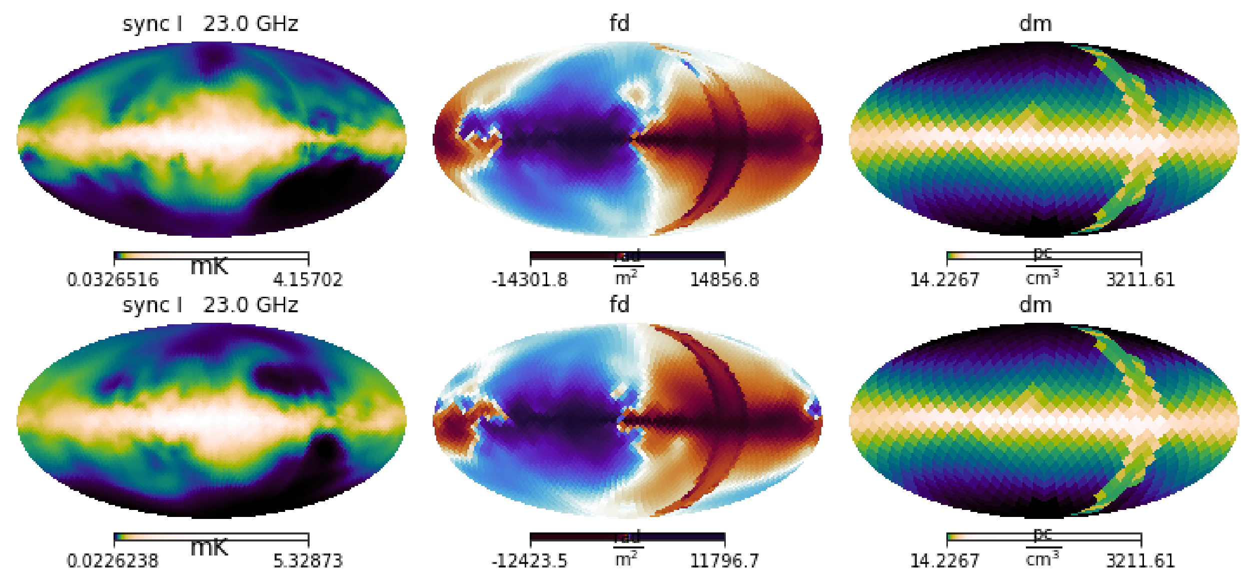
(continued from previous page)

```
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:

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

```
[12]: maps.show()
```



One can easily see the stochastic magnetic field in action by comparing the different model realisations shown in different rows.

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 = 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

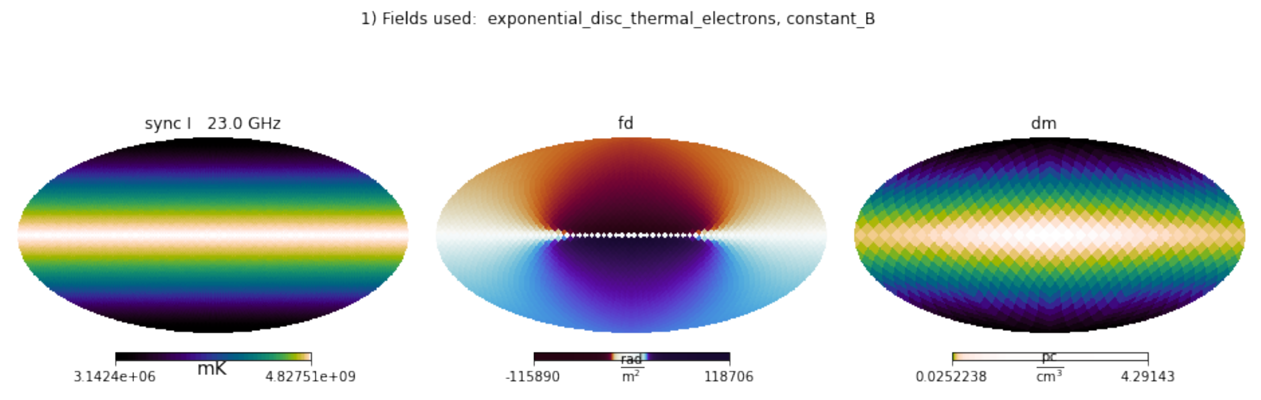
[15]:

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

We can now examine the results

[16]:

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

fig = plt.figure(figsize=(13.0, 4.0))
maps.show()
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.

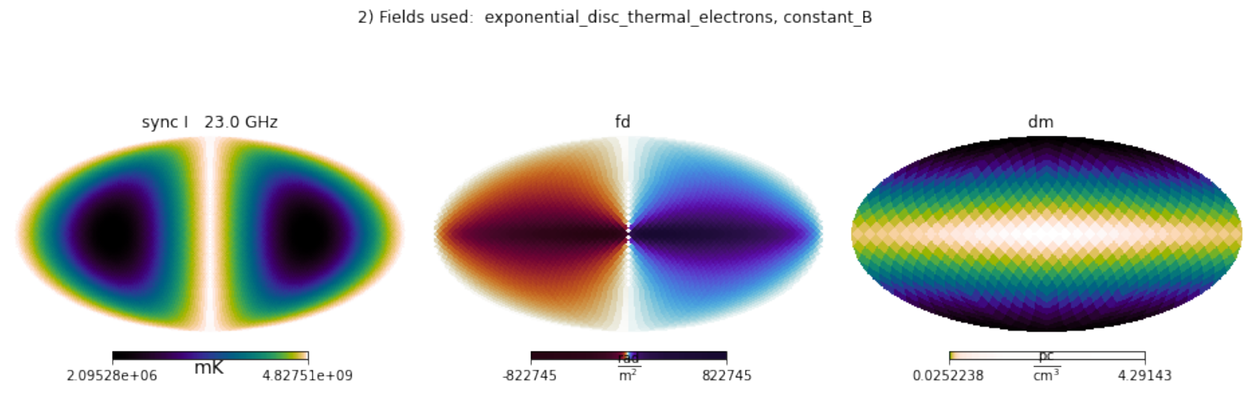
[17]:

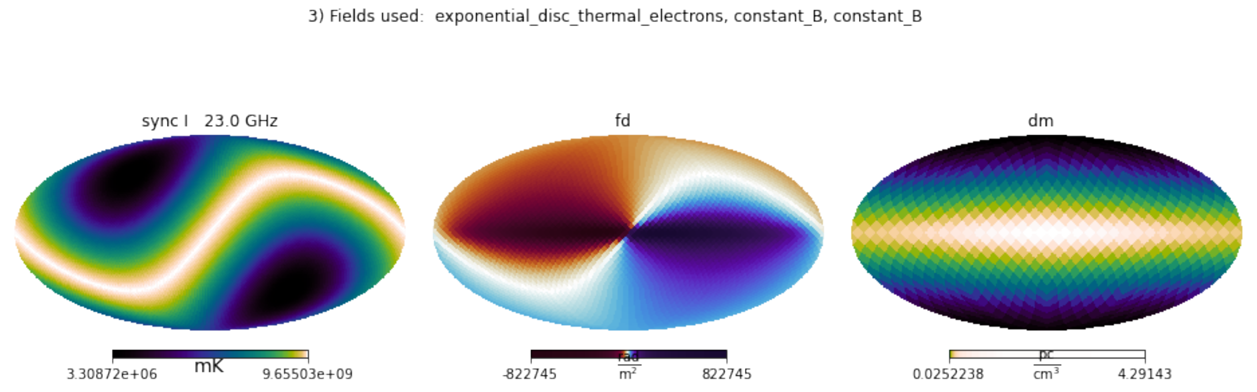
```
fields_list2 = [ne, By_only]
fields_list3 = [ne, By_only, Bz_only]

for i, fields_list in enumerate([fields_list2, fields_list3]):
```

(continues on next page)

(continued from previous page)

```
    # 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)
    maps.show()
    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.

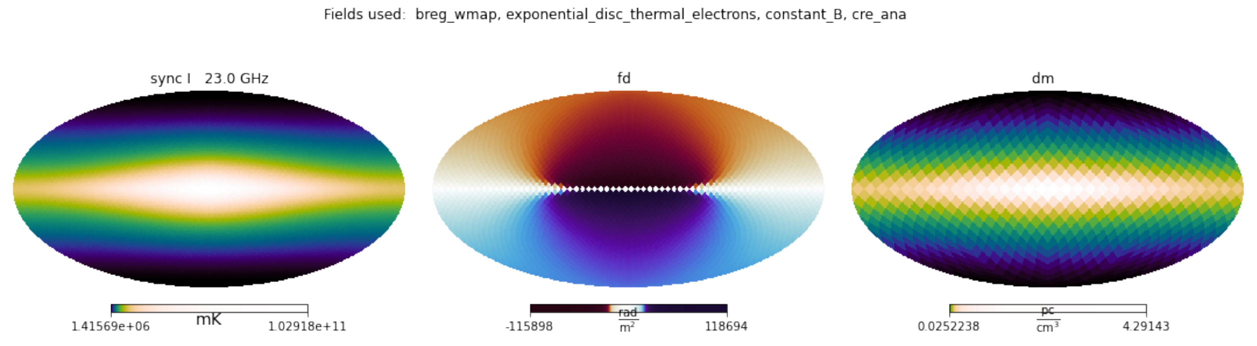
In the following example we illustrate using an IMAGINE field for the "regular magnetic field", and dummies for the "random magnetic field" and cosmic rays.

[18]:
```
# 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]
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(15.0, 4.0))
maps = simer(fields_list)
maps.show()
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 `CustomPrior` 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.

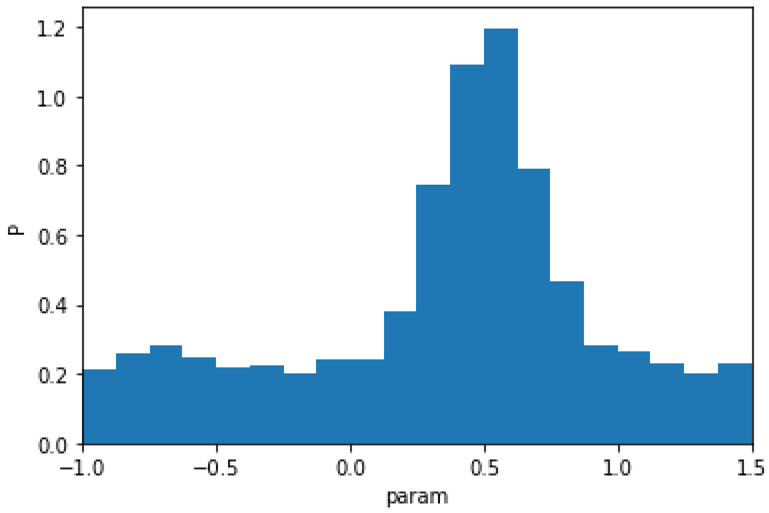
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import astropy.units as u
import imagine as img
import corner, os
import scipy.stats
```

[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
```

(continues on next page)

(continued from previous page)

```
sample *= u.microgauss
plt.hist(sample.value, bins=20, density=True)
plt.ylabel('P'); plt.xlabel('param'); plt.xlim(-1,1.5);
```



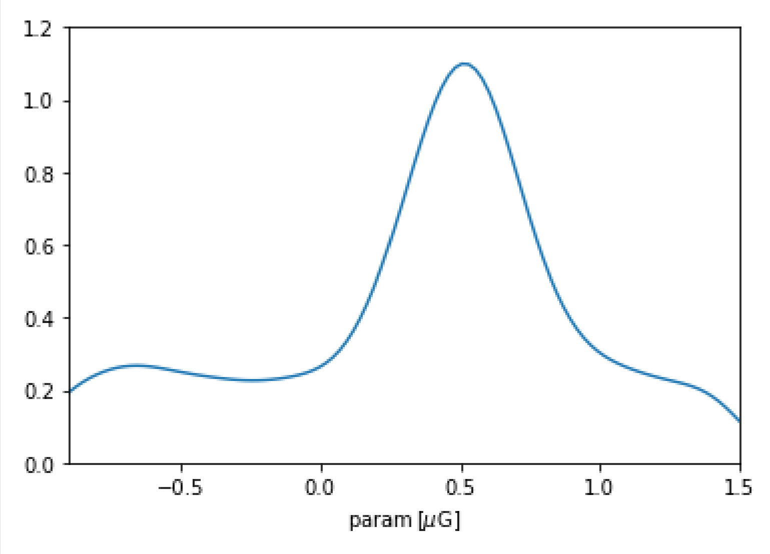
This distribution could be the result of a previous inference exercise (e.g. *a previous run of the IMAGINE pipeline* using a different set of observables).

From it, we can construct our prior using the `CustomPrior` 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]: prior_param = img.priors.CustomPrior(samples=sample,
                                          xmin=-0.9*u.microgauss,
                                          xmax=1.5*u.microgauss)
```

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

```
[4]: p = np.linspace(-0.9, 1.5, 100)*u.microgauss
     plt.plot(p, prior_param.pdf(p))
     plt.xlim(-0.9,1.5); plt.ylim(0,1.2); plt.xlabel(r'param$\,[\mu\rm G]$');
```

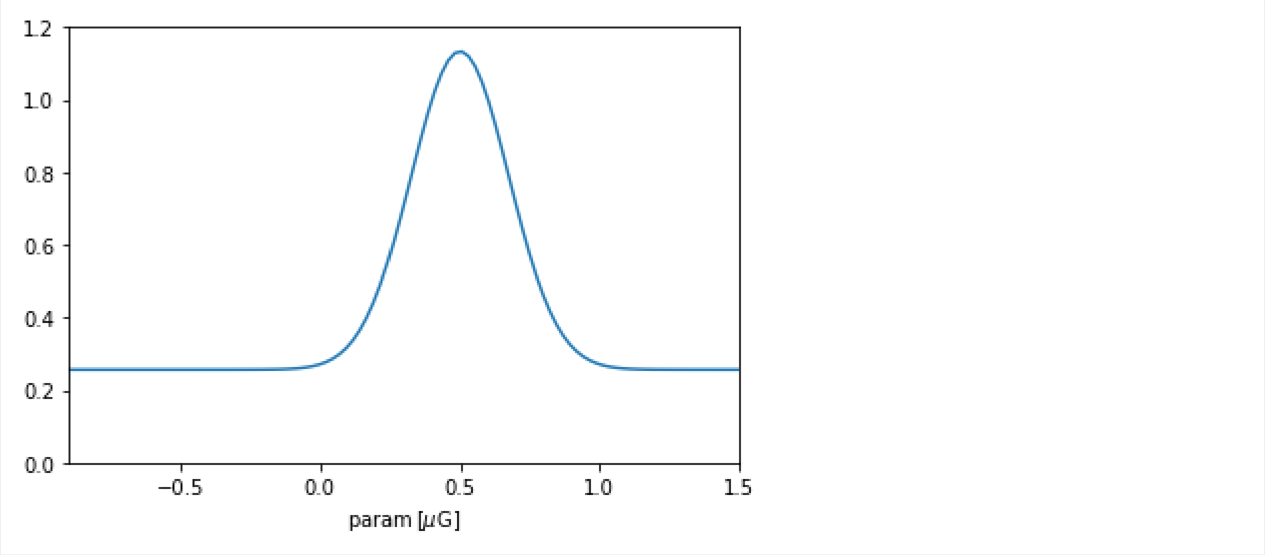


A cautionary note: the KDE used in intermediate calculation tends so smoothen the distribution and forces a slight decay close to the endpoints (reflecting the fact that a Gaussian kernel was employed). For most practical applications, this is not a big problem: one can control the degree of smoothness through the argument `bw_method` while initializing `CustomPrior`, 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

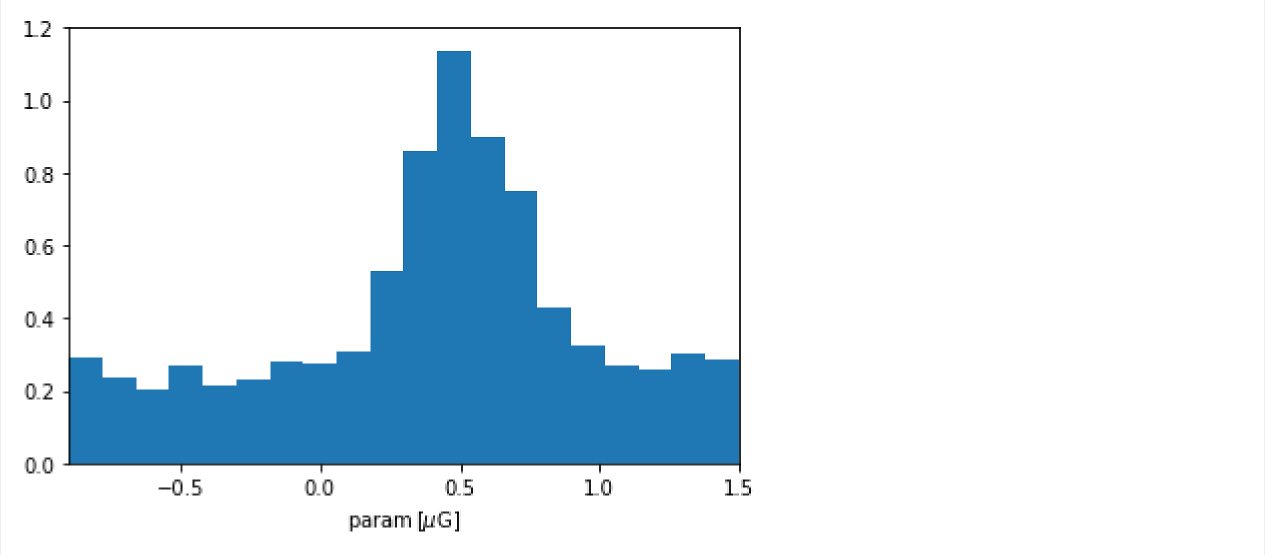
Alternatively, when one knows the analytic shape of given PDF, one can instead supply a function to `CustomPrior`. In this case, the shape of the original function is generally respected. For example:

```
[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.CustomPrior(pdf_fun=example_pdf,
                                          xmin=-0.9*u.microgauss,
                                          xmax=1.5*u.microgauss)

     plt.plot(p, prior_param.pdf(p))
     plt.xlim(-0.9,1.5); 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.

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

     plt.hist(sampled_values.value, bins=20, density=True)
     plt.xlim(-0.9,1.5); plt.ylim(0,1.2); plt.xlabel(r'param$\,[\mu\rm G]$');
```



12.1.3 Flat and Gaussian priors

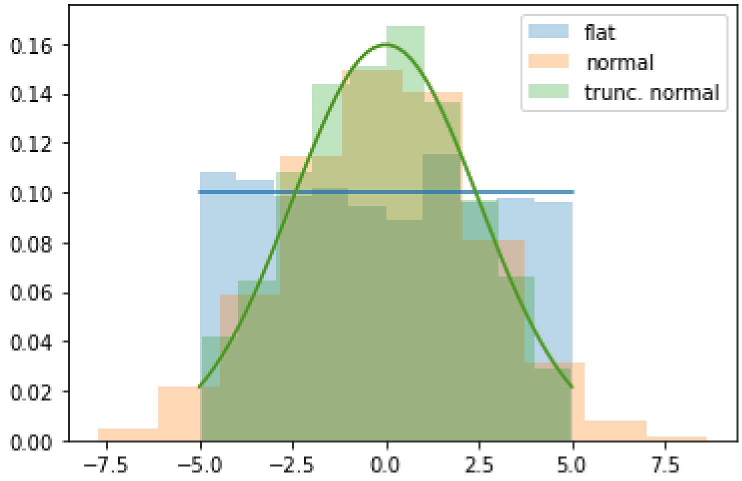
Flat and Normal distributions are common prior choices when one is starting to tackle a particular problem. These are implemented by the classes `img.priors.FlatPrior` and `img.priors.GaussianPrior`, respectively.

```
[7]: muG = u.microgauss # Convenience

     flat_prior = img.priors.FlatPrior(xmin=-5*muG, xmax=5*muG)
```

(continues on next page)

(continued from previous page)

```
gaussian_prior = img.priors.GaussianPrior(mu=0*muG, sigma=2.5*muG)
truncated_gaussian_prior = img.priors.GaussianPrior(mu=0*muG, sigma=2.5*muG,
                                                    xmin=-5*muG, xmax=5*muG)

t = np.linspace(-5,5)*muG
plt.plot(t, flat_prior.pdf(t))
plt.plot(t, gaussian_prior.pdf(t))
plt.plot(t, truncated_gaussian_prior.pdf(t))
plt.gca().set_prop_cycle(None)
# Plots the distribution of values constructed using this prior
x = np.random.random_sample(2000)
plt.hist(flat_prior(x).value,
         density=True, alpha=0.3, label='flat')
plt.hist(gaussian_prior(x).value,
         density=True, alpha=0.3, label='normal')
plt.hist(truncated_gaussian_prior(x).value,
         density=True, alpha=0.3, label='trunc. normal')
plt.legend();
```



If using a `FlatPrior`, the range of the parameter *must be specified*. In the case of a `GaussianPrior`, if one specifies the limits a renormalized truncated distribution is used.

12.1.4 Prior from scipy.stats distribution

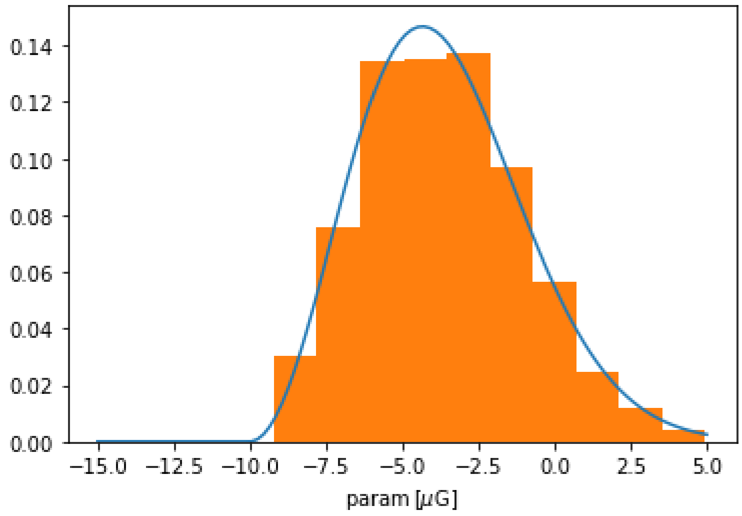
We now demonstrate a helper class which allows to easily construct priors from any of the scipy.stats distributions. Lets say that we would like to impose a chi prior distribution for a given parameter, we can achive this by using the `ScipyPrior` helper class.

```
[8]: chiPrior = img.priors.ScipyPrior(scipy.stats.chi, 3, loc=-10*muG,
                                      scale=4*muG, xmin=-15*muG, xmax=5*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.

[9]:
```
# Plots the PDF associated with this prior
t = np.linspace(*chiPrior.range,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).value, density=True);
plt.xlabel(r'param$\,[\mu\rm G]$');
```

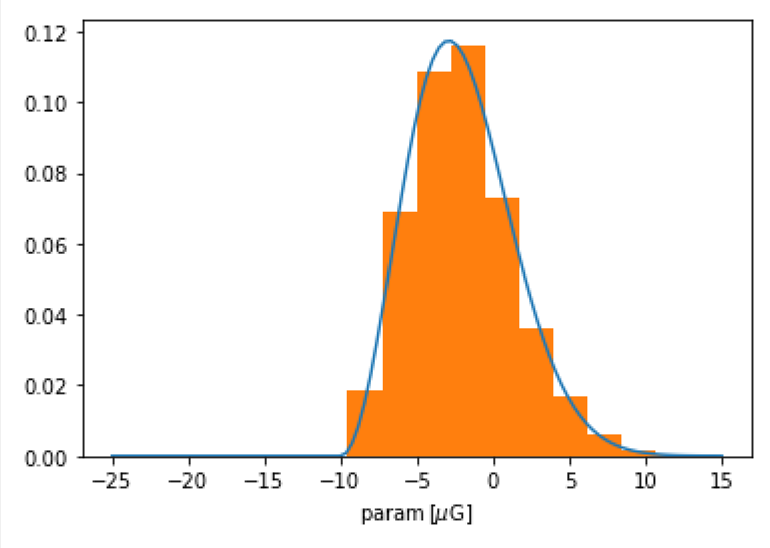


One might have noticed that the distribution above was truncated at the specified interval. As shown in the `GaussianPrior` case (above), IMAGINE also supports unbounded parameter ranges, this can be achieved by refraining from specifying the arguments `xmin` and `xmax` (or by setting them to `None`).

[10]:
```
chi_prior = img.priors.ScipyPrior(scipy.stats.chi, 3, loc=-10*muG, unit=muG,
                                  scale=5*muG)
print('The range is now:', chiPrior.range)
```

```
The range is now: [-15.   5.] uG
```

[11]:
```
# Plots the PDF associated with this prior
t = np.linspace(-25,15,100)*muG
plt.plot(t, chi_prior.pdf(t))
# Plots the distribution of values constructed using this prior
x = np.random.random_sample(2000)
plt.hist(chi_prior(x).value, density=True);
plt.xlabel(r'param$\,[\mu\rm G]$');
```



12.2 Correlated priors

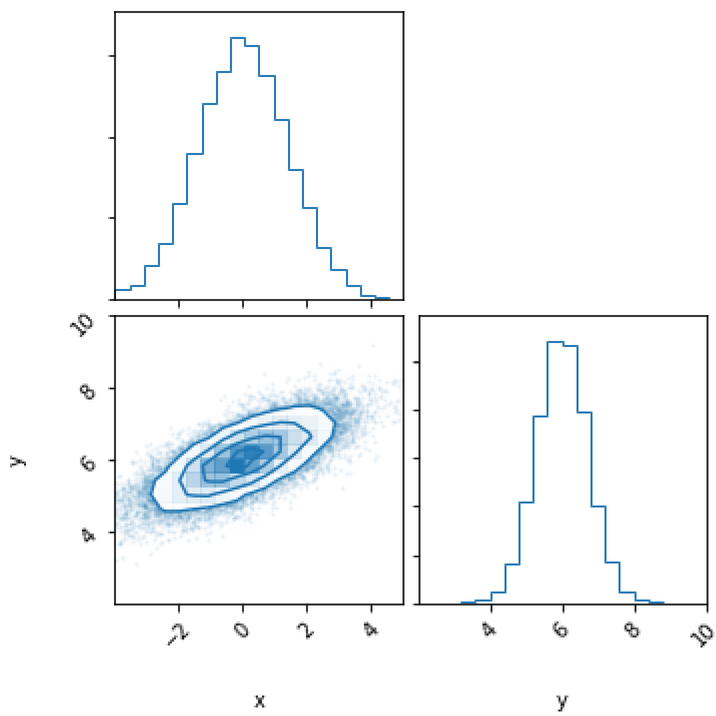
It is also possible to set IMAGINE priors where the parameters are correlated. As previously mentioned, most common case where this is needed is when one starts from the results (samples) of a previous inference and wants now to include a different observable, which is where we begin.

12.2.1 Correlated priors from samples

Given two or more samples supplied to `CustomPrior`, IMAGINE is able to estimate the correlation and use this information while running the Pipeline. To demonstrate this, we begin by artificially constructing samples of correlated priors using scipy's mutivariate normal distribution.

[12]:
```
# Sets up the distribution object
distr = scipy.stats.multivariate_normal(mean=[0, 6],
                                        cov=[[2.0, 0.7],
                                             [0.7, 0.5]])
# Computes the samples and plots them
samples = distr.rvs(25000)
corner.corner(samples, range=[[-4,5],[2,10]], color='tab:blue', labels=['x','y']);
print('Pearson-r correlation: {0:.1f}'.format(scipy.stats.pearsonr(samples[:,0],␣
↪samples[:,1])[0]))
```

```
Pearson-r correlation: 0.7
```



Now, we initialize two `CustomPrior` instances with each of the correlated samples

```
[13]: prior_a = img.priors.CustomPrior(samples=samples[:,0]*muG)
      prior_b = img.priors.CustomPrior(samples=samples[:,1]*muG)
```

For definiteness, let us prepare an imagine pipeline with a similar setup tutorial_one ("Basic pipeline…").

```
[14]: mockData = img.observables.TabularDataset({'test': [1,], 'err': [0.1]},
                                                name='test', err_col='err')
      meas = img.observables.Measurements()
      meas.append(mockData)
      cov = img.observables.Covariances()
      cov.append(mockData)

      grid = img.fields.UniformGrid(box=[[0,2*np.pi]*u.kpc,
                                         [0,0]*u.kpc,
                                         [0,0]*u.kpc],
                                    resolution=[2,1,1])

      likelihood = img.likelihoods.EnsembleLikelihood(meas, cov)
      simer = img.simulators.TestSimulator(meas)

      ne_factory = img.fields.CosThermalElectronDensityFactory(grid=grid)
      B_factory = img.fields.NaiveGaussianMagneticFieldFactory(grid=grid)

      B_factory.active_parameters = ('a0','b0')
      B_factory.priors = {'a0': prior_a, 'b0': prior_b}
```

In the last line, we associated the priors generated from the correlated samples to the parameters `a0` and `b0`. There is one more step to actually account for the correlations in the samples: one need to provide the `Pipeline` with a prior correlations dictionary, explicitly saying that pairs of parameters may display correlations (it will look at the original samples and try to estimate the covariance from there):

[15]:

```
prior_corr_dict = { ('a0', 'b0'): True}

run_dir = os.path.join(img.rc['temp_dir'], 'tutorial_priors')
pipeline = img.pipelines.UltranestPipeline(run_directory=run_dir,
                                           simulator=simer,
                                           factory_list=[ne_factory, B_factory],
                                           likelihood=likelihood,
                                           prior_correlations=prior_corr_dict)
```

If we now run `pipeline`, the sampler will automatically draw the points from a correlated prior.

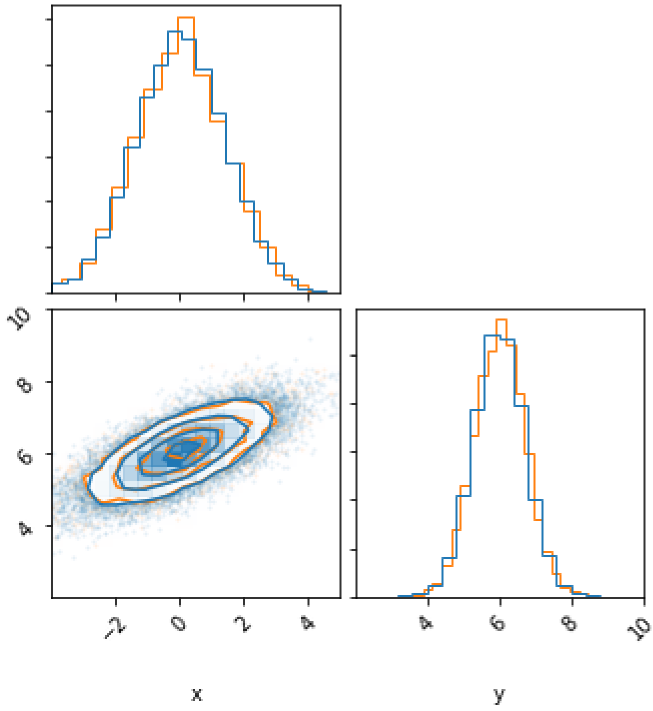
Internally, IMAGINE is providing the chosen sampler with the method `prior_transform` which takes a vector of numbers in the [0, 1] interval (the “unit cube”) and returns the parameter values of active parameters.

Let us use `prior_transform` to check how (and if) this working.

[16]:

```
n = 5000
a, b = np.random.sample(n), np.random.sample(n)
X = pipeline.prior_transform(np.array([a,b]))
fig =corner.corner(X.T, color='tab:orange',hist_kwargs={'density':True})
corner.corner(samples, range=[[-4,5],[2,10]], color='tab:blue',
              labels=['x','y'], hist_kwargs={'density':True}, fig=fig)
print('Pearson-r correlation: {0:.1f}'.format(scipy.stats.pearsonr(X[0], X[1])[0]))
```

```
Pearson-r correlation: 0.7
```



Where the dark blue curve is the new distribution.

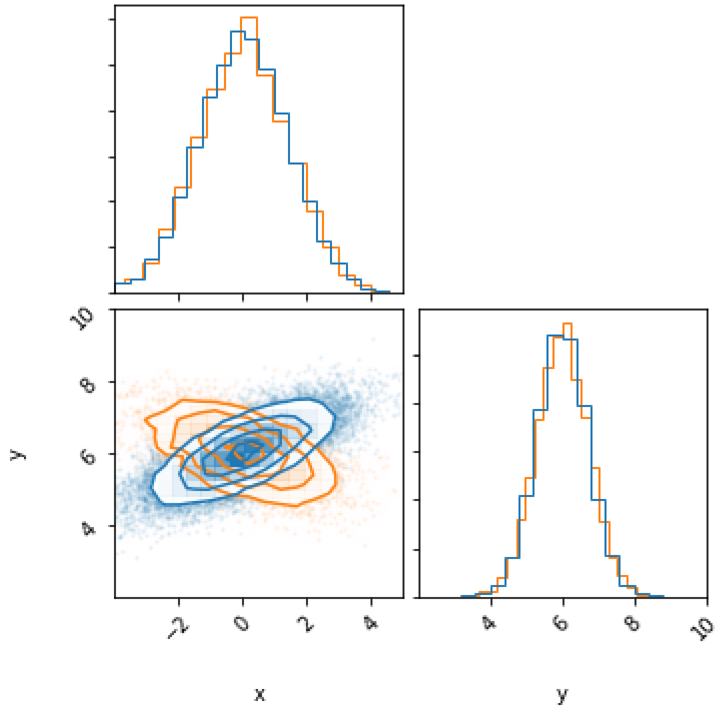
12.2.2 Setting prior correlations manually

In the situations where one is *not* constructing the priors from samples, it is still possible to set up prior correlations between parameters. To do this, we simply specify the correlation coefficient (instead of `True`) in the dictionary supplied to the pipeline with the `prior_correlations` keyword. For example,

```
[17]: prior_corr_dict = { ('a0', 'b0'): -0.5}

      pipeline_new = img.pipelines.UltranestPipeline(run_directory=run_dir,
                                                     simulator=simer,
                                                     factory_list=[ne_factory, B_factory],
                                                     likelihood=likelihood,
                                                     prior_correlations=prior_corr_dict)
```

```
[18]: n = 5000
      a, b = np.random.sample(n), np.random.sample(n)
      X = pipeline_new.prior_transform(np.array([a,b]))
      fig =corner.corner(X.T, color='tab:orange', scale_hist=True,hist_kwargs={'density':
       ↪True})

      corner.corner(samples, range=[[-4,5],[2,10]], color='tab:blue',
                    labels=['x','y'], fig=fig,hist_kwargs={'density':True})
      print('Pearson-r correlation: {0:.1f}'.format(scipy.stats.pearsonr(X[0], X[1])[0]))
```

```
Pearson-r correlation: -0.5
```



This example illustrates how the same marginal distributions kept, but with different correlations.

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]:

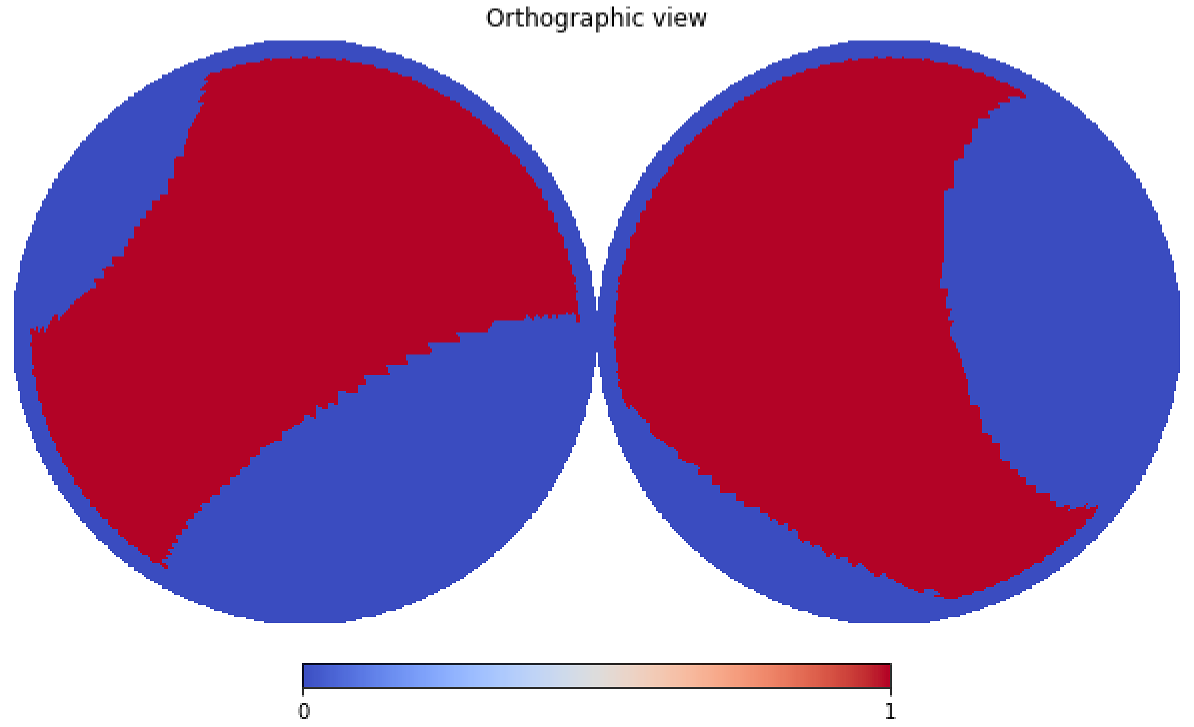
```
mask_nside = 32

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
```

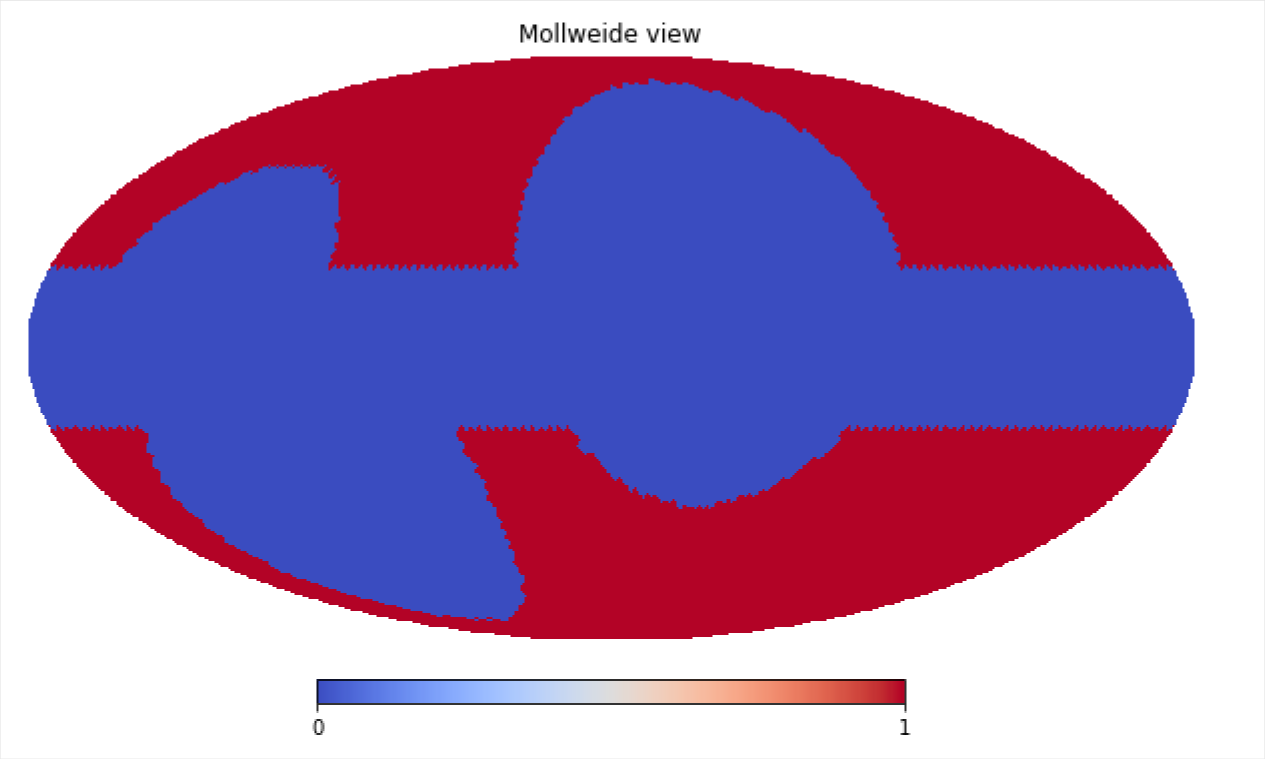
(continues on next page)

(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')
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:209: MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax␣
↪simultaneously is deprecated since 3.3 and will become an error two minor releases␣
↪later. Please pass vmin/vmax directly to the norm when creating it.
  **kwds
```





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(name=('sync', 23.0, 32, 'I'), data=np.vstack([mask_map]))
```

13.2 Applying Masks directly

Typically, after setting the masks, we supply them to the Likelihood and/or Simulator classes, which allow them (see section below) *to be used* in the pipeline run.

Nevertheless, to understand or check what is going on internally, 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
sync_dset = img_obs.SynchrotronHEALPixDataset(data=np.empty(12*32**2)*u.K,
                                              frequency=23, typ='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,
```

(continues on next page)

(continued from previous page)

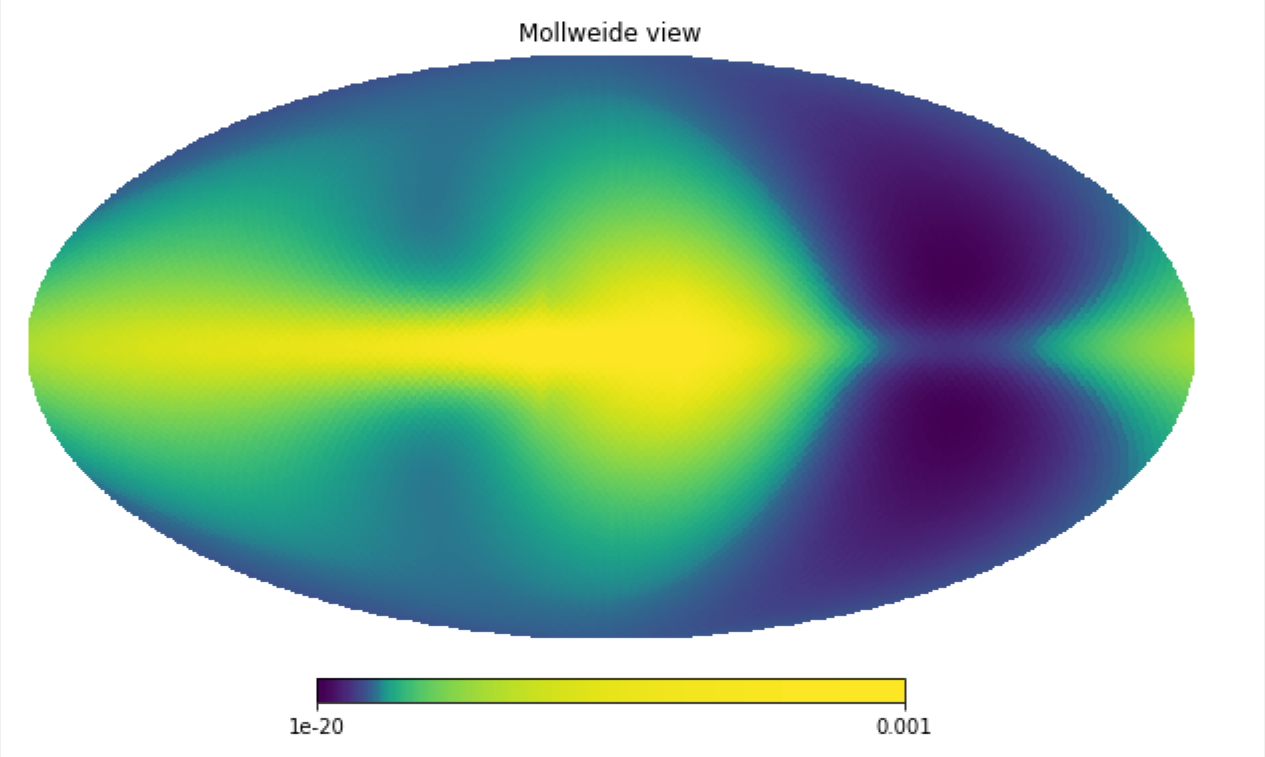
```
                              'theta': 0.0,'r0': 5.6, 'z0': 1.2,
                              'E0': 20.5, 'j0': 0.03})
fereg_ymw16 = TEregYMW16(parameters={})

# 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_map = maps[('sync', 23.0, 32, 'I')].data[0]
hp.mollview(unmasked_map, norm='hist', min=1e-20, max=1.0e-3)
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:907: MatplotlibDeprecationWarning: You are modifying the state of a globally␣
↪registered colormap. In future versions, you will not be able to modify a␣
↪registered colormap in-place. To remove this warning, you can make a copy of the␣
↪colormap first. cmap = copy.copy(mpl.cm.get_cmap("viridis"))
  newcm.set_over(newcm(1.0))
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:908: MatplotlibDeprecationWarning: You are modifying the state of a globally␣
↪registered colormap. In future versions, you will not be able to modify a␣
↪registered colormap in-place. To remove this warning, you can make a copy of the␣
↪colormap first. cmap = copy.copy(mpl.cm.get_cmap("viridis"))
  newcm.set_under(bgcolor)
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:909: MatplotlibDeprecationWarning: You are modifying the state of a globally␣
↪registered colormap. In future versions, you will not be able to modify a␣
↪registered colormap in-place. To remove this warning, you can make a copy of the␣
↪colormap first. cmap = copy.copy(mpl.cm.get_cmap("viridis"))
  newcm.set_bad(badcolor)
```



The `masks` object acts as a function that takes an observable dictionary (i.e. `Measurements`, `Covariances` or `Simulations`) and returns a new `ObservablesDict` with relevant maps already masked.

```
[6]: # Creates the masked map(s)
new_maps = masks(maps)
# Stores it, for conveniency
masked_map = new_maps[('sync', 23.0, 4941,'I')].data[0]
```

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

```
[7]: print('Masked map size:', masked_map.size)
print('Orignal map size', unmasked_map.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.

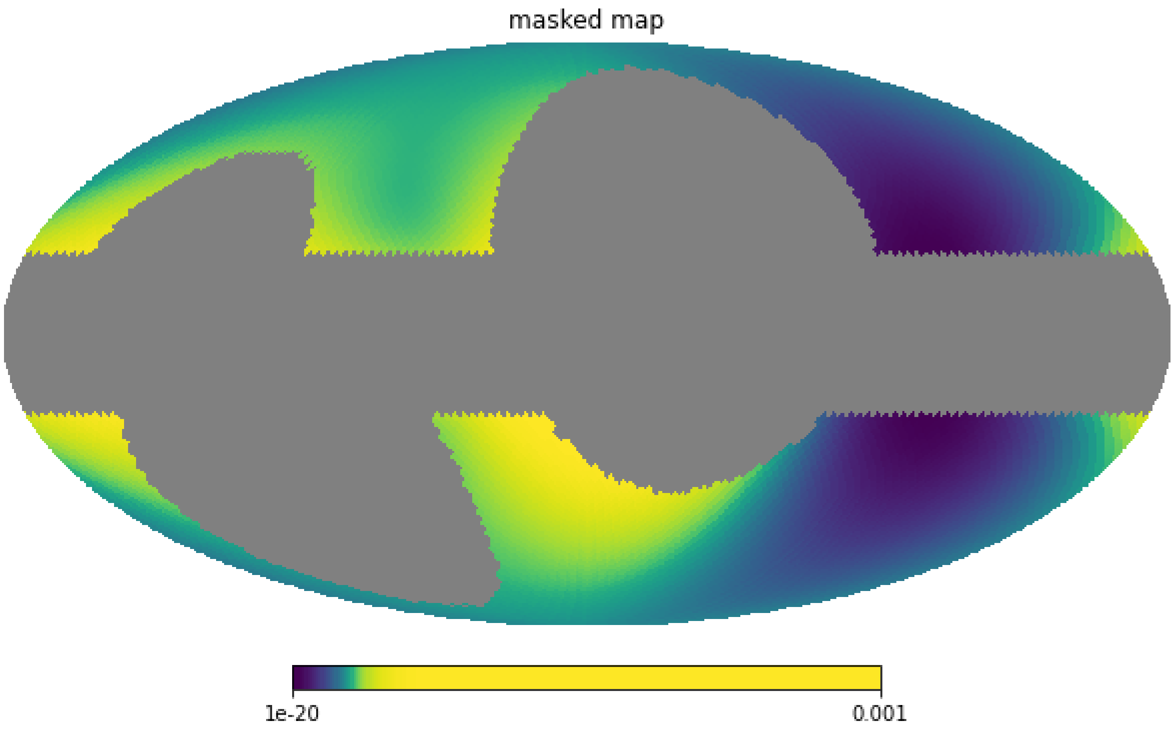
However, if, to check whether things are working correctly, we wish to *look* at masked image, we need to reconstruct it. This means creating a new image including the pixels which we previously have thrown away, as exemplified below:

```
[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
```

(continues on next page)

(continued from previous page)

```
    else:
        masked[i] = masked_map[idx]
        idx += 1
# Shows the image
hp.mollview(masked, norm='hist', min=1e-20, max=1.0e-3, title='masked map')
```



13.3 Using the Masks

The masks affect (separately) two aspects of the calculation: they can influence the Likelihood object, making it ignoring the masked parts of the maps in the likelihood calculation; and they can change the behaviour of the Simulator object, allowing it to ignore the masked pixels while computing the simulated maps.

13.3.1 Masks in likelihood calculation

To use the masks in the likelihood calculation, they should be supplied while initializing the *Likelihood* object, as an extra argument, for example:

```
[9]: likelihood = img.likelihoods.SimpleLikelihood(fakeMeasureDict, mask_dict=masks)
```

When a mask is supplied, the likelihood object stores internally only the masked version of the `Measurements`, which can be checked in the following way:

```
[10]: likelihood.measurement_dict.keys()
```

```
[10]: dict_keys([('sync', 23, 4941, 'I')])
```

13.3.2 Masks with the Hammurabi simulator

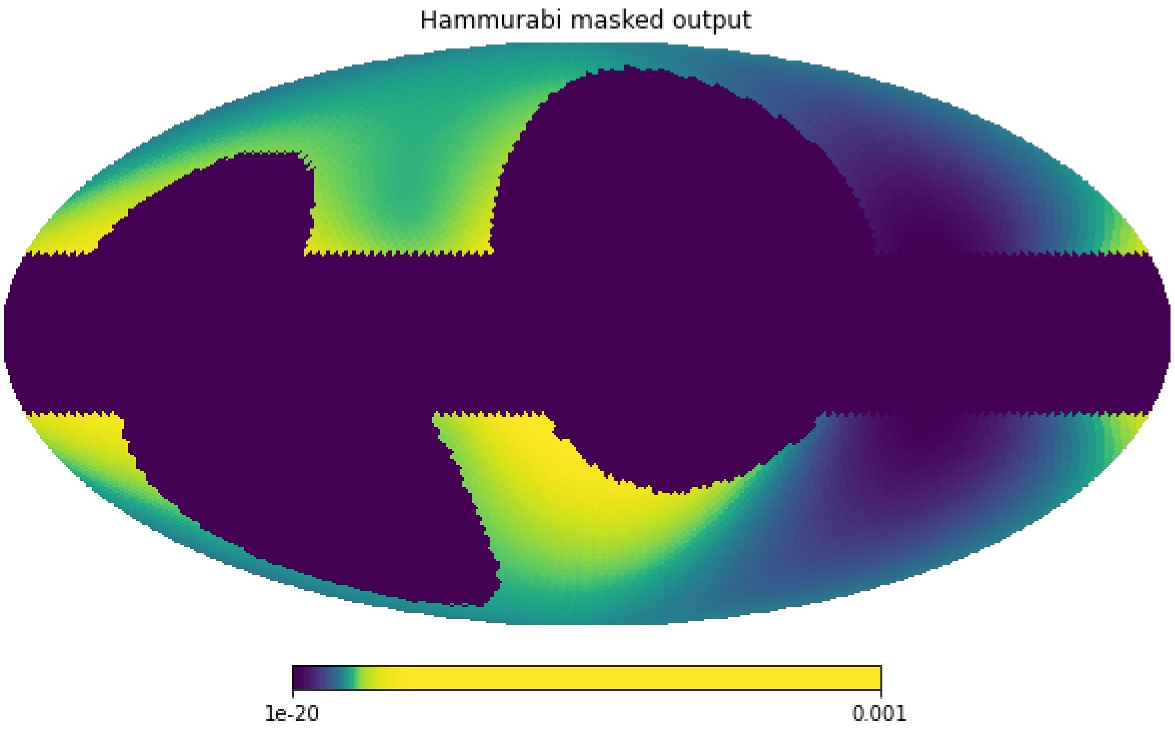
To set-up Hammurabi to use masks, it is sufficient to initialize the simulator providing the masks using the `masks` keyword argument. However, there a subtlety: while Hammurabi X does support masks, there is *only a single mask input entry*, which means all outputs will be masked by the same mask.

Thus, the masks provided to the IMAGINE Hammurabi simulator must cover all Hammurabi-compatible observables and (currently) they must be *identical* (including having the *same resolution*, in the future, however, IMAGINE will support compatible masks of different resolutions to be applied).

[11]:
```
# Initializing Hammurabi including the masks
simulator = Hammurabi(measurements=fakeMeasureDict, masks=masks)
```

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

[12]:
```
# Re-runs the simulator which is using the mask internally
maps = simulator([breg_wmap, cre_ana, fereg_ymw16])
# Shows the new map
hammurabi_masked = maps[('sync', 23.0, 32,'I')].data[0]
hp.mollview(hammurabi_masked, norm='hist', min=1e-20, max=1.0e-3,
            title='Hammurabi masked output')
```



We see that, when the masks are used by Hammurabi, the simulator does not do any work on the masked pixels and they receive `0` in the output produced.

CHAPTER 14

Example pipeline

In this tutorial, we make a slightly more realistic use of IMAGINE: to constrain a few parameters of (part of) the WMAP GMF model. Make sure you have already read (at least) the Basic elements of an IMAGINE pipeline (aka tutorial_one) before proceeding.

We will use Hammurabi as our Simulator and on Hammurabi's built-in models as Fields. We will construct mock data using Hammurabi itself. We will also show how to:

- configure IMAGINE logging,
- test the setup,
- monitor to progress, and
- save or load IMAGINE runs.

First, let us import the required packages/modules.

[1]:

```
# Builtin
import os
# External packages
import numpy as np
import healpy as hp
import astropy.units as u
import corner
import matplotlib.pyplot as plt
import cmasher as cmr
# IMAGINE
import imagine as img
import imagine.observables as img_obs
## "WMAP" field factories
from imagine.fields.hamx import BregLSA, BregLSAFactory
from imagine.fields.hamx import TEregYMW16, TEregYMW16Factory
from imagine.fields.hamx import CREAna, CREAnaFactory
```

14.1 Logging

IMAGINE comes with logging features using Python’s native logging package. To enable them, one can simply set where one wants the log file to be saved and what is the level of the logging.

[2]:

```
import logging

logging.basicConfig(filename='tutorial_wmap.log', level=logging.INFO)
```

Under `logging.INFO` level, IMAGINE will report major steps and log the likelihood evaluations. If one wants to trace a specific problem, one can use the `logging.DEBUG` level, which reports when most of the functions or methods are accessed.

14.2 Preparing the mock data

Let’s make a very low resolution map of synchrotron total I and Faraday depth from the WMAP model, but without including a random compnent (this way, we can limit the ensemble size to 1, speeding up the inference):

[3]:

```
## Sets the resolution
nside=2
size = 12*nside**2

# Generates the fake datasets
sync_dset = img_obs.SynchrotronHEALPixDataset(data=np.empty(size)*u.K,
                                              frequency=23, typ='I')
fd_dset = img_obs.FaradayDepthHEALPixDataset(data=np.empty(size)*u.rad/u.m**2)

# Appends them to an Observables Dictionary
trigger = img_obs.Measurements(sync_dset, fd_dset)

# Prepares the Hammurabi simmulator for the mock generation
mock_generator = img.simulators.Hammurabi(measurements=trigger)
```

```
observable {}
|-->  sync {'cue': '1', 'freq': '23', 'nside': '2'}
|-->  faraday {'cue': '1', 'nside': '2'}
```

We will feed the `mock_generator` simulator with selected Dummy fields.

[4]:

```
# BregLSA field
breg_lsa = BregLSA(parameters={'b0':3, 'psi0': 27.0, 'psi1': 0.9, 'chi0': 25.0})

# CREAna field
cre_ana = CREAna(parameters={'alpha': 3.0, 'beta': 0.0, 'theta': 0.0,
                             'r0': 5.0, 'z0': 1.0,
                             'E0': 20.6, 'j0': 0.0217})

# TEregYMW16 field
tereg_ymw16 = TEregYMW16(parameters={})
```

[5]:

```
## Generate mock data (run hammurabi)
outputs = mock_generator([breg_lsa, cre_ana, tereg_ymw16])
```

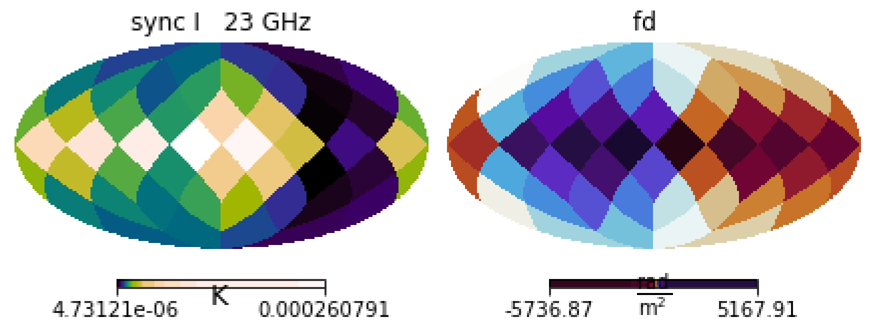
To make a realistic mock, we add to these outputs, which where constructed from a model with known parameter, some noise, which assumed to be proportional to the average synchrotron intensity.

[6]:
```
## Collect the outputs
mockedI = outputs[('sync', 23.0, nside, 'I')].global_data[0]
mockedRM = outputs[('fd', None, nside, None)].global_data[0]
dm=np.mean(mockedI)
dv=np.std(mockedI)

## Add some noise that's just proportional to the average sync I by the factor err
err=0.01
dataI = (mockedI + np.random.normal(loc=0, scale=err*dm, size=size)) << u.K
errorI = (err*dm) << u.K
sync_dset = img_obs.SynchrotronHEALPixDataset(data=dataI, error=errorI,
                                              frequency=23, typ='I')

## Just 0.01*50 rad/m^2 of error for noise.
dataRM = (mockedRM + np.random.normal(loc=0.,scale=err*50.,size=12*nside**2))*u.rad/u.
↪m/u.m
errorRM = (err*50.) << u.rad/u.m**2
fd_dset = img_obs.FaradayDepthHEALPixDataset(data=dataRM, error=errorRM)
```

We are ready to include the above data in `Measurements` and objects

[7]:
```
mock_data = img_obs.Measurements(sync_dset, fd_dset)

mock_data.show()
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:209: MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax␣
↪simultaneously is deprecated since 3.3 and will become an error two minor releases␣
↪later. Please pass vmin/vmax directly to the norm when creating it.
  **kwds
```



14.3 Assembling the pipeline

After preparing our mock data, we can proceed with the set up of the IMAGINE pipeline. First, we initialize the `Likelihood`, using the mock observational data

[8]:
```
## Use an ensemble to estimate the galactic variance
likelihood = img.likelihoods.EnsembleLikelihood(mock_data)
```

Then, we prepare the `FieldFactory` list:

[9]:
```
## WMAP B-field, vary only b0 and psi0
breg_factory = BregLSAFactory()
```

(continues on next page)

(continued from previous page)

```
breg_factory.active_parameters = ('b0', 'psi0')
breg_factory.priors = {'b0':  img.priors.FlatPrior(xmin=0., xmax=10.),
                       'psi0': img.priors.FlatPrior(xmin=0., xmax=50.)}
## Fixed CR model
cre_factory = CREAnaFactory()
## Fixed FE model
fereg_factory = TEregYMW16Factory()

# Final Field factory list
factory_list = [breg_factory, cre_factory, fereg_factory]
```

We initialize the `Simulator`, in this case: `Hammurabi` .

[10]:
```
simulator = img.simulators.Hammurabi(measurements=mock_data)
```

```
observable {}
|-->  sync {'cue': '1', 'freq': '23', 'nside': '2'}
|-->  faraday {'cue': '1', 'nside': '2'}
```

Finally, we initialize and setup the `Pipeline` itself, using the `Multinest` sampler.

[11]:
```
# Assembles the pipeline using MultiNest as sampler
pipeline = img.pipelines.MultinestPipeline(run_directory='../runs/tutorial_example/',
                                           simulator=simulator,
                                           show_progress_reports=True,
                                           factory_list=factory_list,
                                           likelihood=likelihood,
                                           ensemble_size=1, n_evals_report=15)
pipeline.sampling_controllers = {'n_live_points': 500}
```

We set a run directory, `'runs/tutorial_example'` for storing this state of the run and MultiNest's chains. This is *strongly recommended*, as it makes it easier to resume a crashed or interrupted IMAGINE run.

Since there are no stochastic fields in this model, we chose an ensemble size of 1.

14.4 Checking the setup

Before running a heavy job, it is a good idea to be able to roughly estimate how long it will take (so that one can e.g. decide whether one will read emails, prepare some coffee, or take a week of holidays while waiting for the results).

One thing that can be done is checking how long the code takes to do an individual likelihood function evaluation (note that each of these include the whole ensemble). This can be done using the `test()` method, which allows one to test the `Pipeline` object evaluating a few times the final likelihood function (which is handed to the sampler in an actual run) and timing it.

[12]:
```
pipeline.test(n_points=4);
```

```
Sampling centres of the parameter ranges.
        Evaluating point: [5.0, 25.0]
        Log-likelihood -63230233.62965755
        Total execution time:  3.699134146794677 s

Randomly sampling from prior.
        Evaluating point: [3.02332573 7.33779454]
        Log-likelihood -2440296.9511515824
```

(continues on next page)

(continued from previous page)

```
        Total execution time:  3.7439119834452868 s

Randomly sampling from prior.
        Evaluating point: [0.92338595 9.31301057]
        Log-likelihood -56235239.046620436
        Total execution time:  3.723511670716107 s

Randomly sampling from prior.
        Evaluating point: [ 3.45560727 19.83837371]
        Log-likelihood -5868446.043333106
        Total execution time:  3.6879931911826134 s

Average execution time: 3.713637748034671 s
```

This is not a small amount of time. Note that thousands of evaluations will be needed to be able to estimate the evidence (and/or posterior distributions). Fortunately, this tutorial comes with the results from an interrupted run, reducing the amount of waiting in case someone is interested in seeing the pipeline in action.

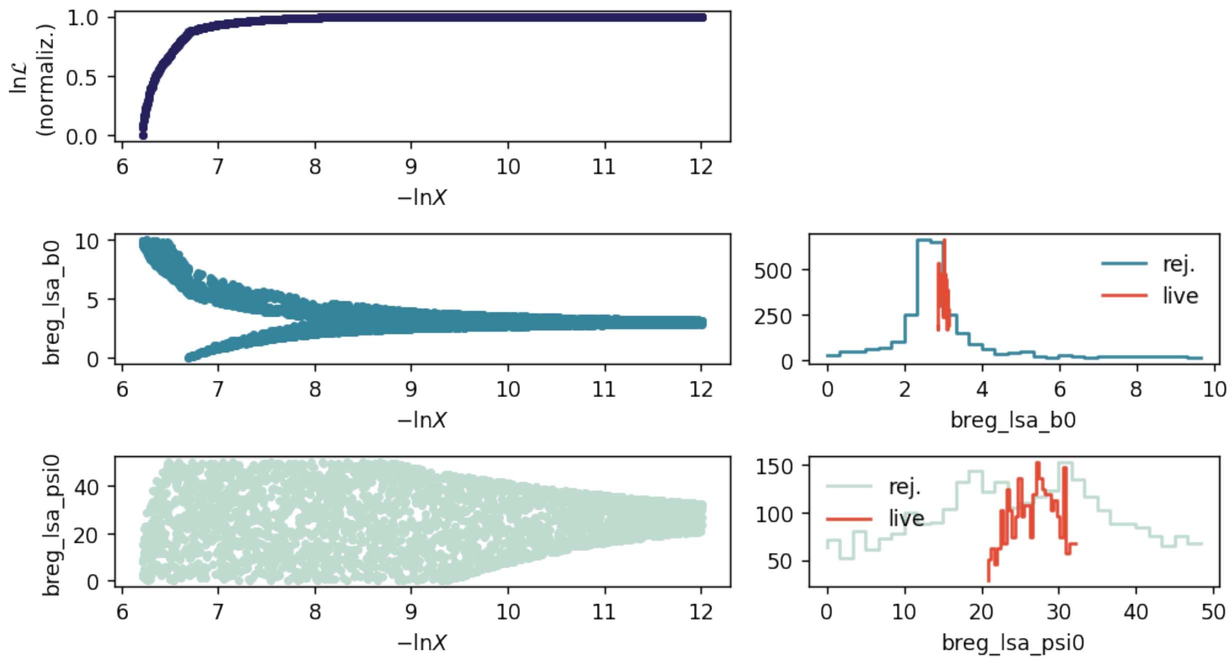
14.5 Running on a jupyter notebook

While running the pipeline in a jupyter notebook, a simple progress report is generated every `pipeline.n_evals_report` evaluations of the likelihood. In the nested sampling case, this shows the parameter choices for rejected ("dead") points as a function of log "prior volume", $\ln X$, and the distributions of both rejected points and "live" points (the latter in red).

One may allow the pipeline to run (for many hours) to completion or simply skip the next cell. The last section of this tutorial *loads* a completed version this very same pipeline from disk, if one is curious about how the results look like.

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

Progress report: number of likelihood evaluations 4500



14.6 Monitoring progress when running as a script

When IMAGINE is launched using a script (e.g. when running on a cluster) it is still possible to monitor the progress by inspecting the file `progress_report.pdf` in the selected run directory.

14.7 Saving and loading

The `pipeline` is automatically saved to the specified `run_directory` immediately before and immediately after running the sampler. It can also be saved, at any time, using the `pipeline.save()`.

Being able to save and load IMAGINE Pipelines is particularly convenient if one has to alternate between a workstation and an HPC cluster: one can start preparing the setup on the workstation, using a jupyter notebook; then submit a script to the cluster which runs the previously saved pipeline; then reopen the pipeline on a jupyter notebook to inspect the results.

To illustrate these tools, let us load a completed version of the above example:

```
[14]: previous_pipeline = img.load_pipeline('../runs/tutorial_example_completed/')
```

This object should contain (in its attributes) all the information one can need to run, manipulate or analise an IMAGINE pipeline. For example, one can inspect which field factories were used in this run

```
[15]: print('Field factories used: ', end='')
for field_factory in previous_pipeline.factory_list:
    print(field_factory.name, end=', ')
```

```
Field factories used: breg_lsa, cre_ana, tereg_ymw16,
```

Or, perhaps one may be interested in finding which simulator was used

```
[16]: previous_pipeline.simulator
```

```
[16]: <imagine.simulators.hammurabi.Hammurabi at 0x7f77e89d1b50>
```

The previously selected observational data can be accessed inspecting the Likelihood object which was supplied to the Pipeline:

```
[17]: print('The Measurements object keys are:')
print(list(previous_pipeline.likelihood.measurement_dict.keys()))
```

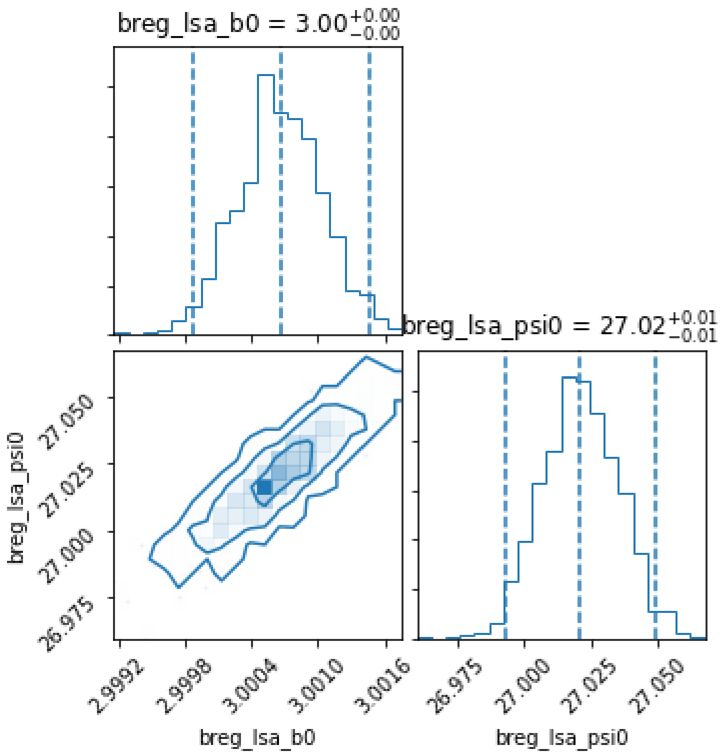
```
The Measurements object keys are:
[('sync', 23, 2, 'I'), ('fd', None, 2, None)]
```

And one can, of course, also *run the pipeline*

```
[18]: previous_pipeline(save_pipeline_state=False);
```

```
  analysing data from ../runs/tutorial_example_completed/chains/multinest_.txt
```

Posterior report:



Evidence report:

$\log \mathcal{Z} = 609.2 \pm 0.2$

14.8 Best-fit model

Frequently, it is useful (or needed) to inspect (or re-use) the best-fit choice of parameters found. For covenience, this can be done using the property `pipeline.median_model`, which generates a list of Fields corresponding to the median values of all the paramters varied in the inference.

[19]:

```python
# Reads the list of Fields corresponding the best-fit model
best_fit_fields_list = previous_pipeline.median_model

# Prints Field name and parameter choices
# (NB this combines default and active parameters)
for field in best_fit_fields_list:
    print('\n',field.name)
    for p, v in field.parameters.items():
        print('\t',p,'\t',v)
```

```
 breg_lsa
         b0      3.0006600127943743
         psi0    27.020995955659178
         psi1    0.9
         chi0    25.0

 cre_ana
         alpha   3.0
```

(continues on next page)

(continued from previous page)

```
        beta     0.0
        theta    0.0
        r0       5.0
        z0       1.0
        E0       20.6
        j0       0.0217

tereg_ymw16
```

This list of Fields can, then, be directly used as argument by any compatible Simulator object.

Often, however, one is interested in inspecting the Simulation associated with the best-fit model under the exact same conditions (including random seeds for stochastic components) used in the original pipeline run. For this, we can use the `median_simulation` Pipeline property.

[20]:

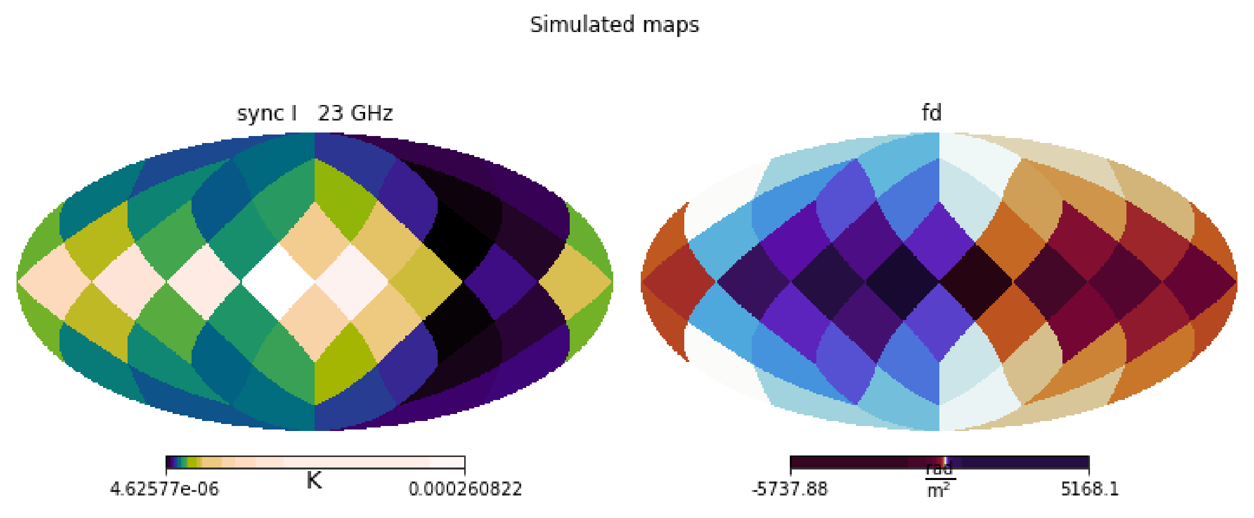
```
simulated_maps = previous_pipeline.median_simulation
```

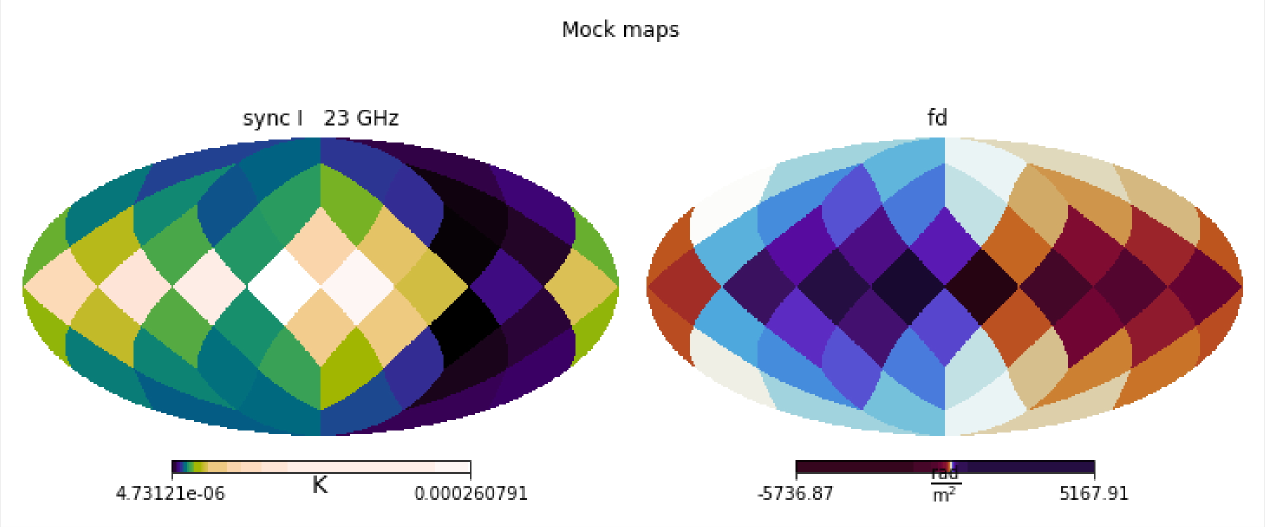
This allows us to *visually inspect* the outcomes of the run, comparing them to the original observational data (or mock data, in the case of this tutorial), and spot any larger issues.

[21]:

```
plt.figure(figsize=(10,4))
simulated_maps.show()
plt.suptitle('Simulated maps')
plt.figure(figsize=(10,4))
plt.suptitle('Mock maps')
mock_data.show()
```

```
/home/lrodrigues/miniconda3/envs/imagine/lib/python3.7/site-packages/healpy/projaxes.
↪py:209: MatplotlibDeprecationWarning: Passing parameters norm and vmin/vmax␣
↪simultaneously is deprecated since 3.3 and will become an error two minor releases␣
↪later. Please pass vmin/vmax directly to the norm when creating it.
  **kwds
```




Mock maps
sync I   23 GHz
4.73121e-06
K
0.000260791
fd
-5736.87
$\frac{\mathrm{rad}}{\mathrm{m}^2}$
5167.91

CHAPTER 15

imagine package

15.1 Subpackages

15.1.1 imagine.fields package

Subpackages

imagine.fields.hamx package

Submodules

imagine.fields.hamx.breg_lsa module

imagine.fields.hamx.brnd_es module

imagine.fields.hamx.cre_analytic module

imagine.fields.hamx.tereg_ymw16 module

Module contents

Submodules

imagine.fields.base_fields module

imagine.fields.basic_fields module

imagine.fields.field module

imagine.fields.field_factory module

imagine.fields.grid module

imagine.fields.test_field module

Module contents

15.1.2 imagine.likelihoods package

Submodules

imagine.likelihoods.ensemble_likelihood module

imagine.likelihoods.likelihood module

imagine.likelihoods.simple_likelihood module

Module contents

15.1.3 imagine.observables package

Submodules

imagine.observables.dataset module

imagine.observables.observable module

imagine.observables.observable_dict module

Module contents

15.1.4 imagine.pipelines package

Submodules

imagine.pipelines.dynesty_pipeline module

imagine.pipelines.multinest_pipeline module

imagine.pipelines.pipeline module

imagine.pipelines.ultranest_pipeline module

Module contents

15.1.5 imagine.priors package

Submodules

imagine.priors.basic_priors module

imagine.priors.prior module

Module contents

15.1.6 imagine.simulators package

Submodules

imagine.simulators.hammurabi module

imagine.simulators.simulator module

imagine.simulators.test_simulator module

Module contents

15.1.7 imagine.tools package

Submodules

imagine.tools.carrier_mapper module

imagine.tools.class_tools module

imagine.tools.config module

imagine.tools.covariance_estimator module

imagine.tools.io module

imagine.tools.masker module

imagine.tools.misc module

imagine.tools.mpi_helper module

imagine.tools.parallel_ops module

imagine.tools.random_seed module

imagine.tools.timer module

imagine.tools.visualization module

Module contents

15.2 Module contents

CHAPTER 16

Indices and tables

- genindex
- modindex
- search