PRISM documentation

Ellert van der Velden

Sep 16, 2020

User Documentation

1 Introduction **3**
- 1.1 Why use PRISM? . . . 3
- 1.2 When (not) to use PRISM? . . . 4

2 Getting started **5**
- 2.1 Installation . . . 5
- 2.2 Running tests . . . 5
- 2.3 Example usage . . . 6

3 The PRISM pipeline **7**
- 3.1 MPI implementation . . . 9

4 ModelLink: A crash course **11**
- 4.1 Writing a ModelLink subclass . . . 11
- 4.2 Data identifiers (*data_idx*) . . . 15
- 4.3 Wrapping a model (*call_model*) . . . 16
  - 4.3.1 Input arguments . . . 16
  - 4.3.2 Multi-calling . . . 17
  - 4.3.3 Backing up progress . . . 18
- 4.4 Model discrepancy variance (*md_var*) . . . 21
  - 4.4.1 Theory . . . 21
  - 4.4.2 Implementation . . . 22

5 Using PRISM **23**
- 5.1 Minimal example . . . 23
- 5.2 Projections . . . 24
  - 5.2.1 Properties . . . 24
  - 5.2.2 Options . . . 27
  - 5.2.3 Crystal (GUI) . . . 30
- 5.3 Dual nature (normal/worker mode) . . . 30
- 5.4 Hybrid sampling . . . 32
  - 5.4.1 Algorithm . . . 32
  - 5.4.2 Usage . . . 33
  - 5.4.3 Application . . . 34
- 5.5 General usage rules . . . 35
- 5.6 External data files . . . 36
  - 5.6.1 PRISM parameters file . . . 36

5.6.2 Model parameters file . . . 38
5.6.3 Model data file . . . 39

6 Descriptions **41**
6.1 Terminology . . . 41
6.2 PRISM parameters . . . 45
6.3 HDF5 . . . 47

7 FAQ **51**
7.1 How do I contribute? . . . 51
7.2 How do I report a bug/problem? . . . 51
7.3 What does the term ... mean? . . . 51
7.4 Where can I get PRISM's colormaps? . . . 51
7.5 Which OSs are supported? . . . 52

8 Community guidelines **53**
8.1 License . . . 53
8.2 Citation . . . 54
8.3 Additions . . . 54

9 Pipeline **57**

10 Emulator **79**
10.1 Classes . . . 79
10.1.1 Emulator . . . 79

11 ModelLink **93**
11.1 Classes . . . 93
11.1.1 GaussianLink . . . 93
11.1.2 ModelLink . . . 93
11.1.3 PolyLink . . . 101
11.1.4 SineWaveLink . . . 101
11.2 Utilities . . . 102

12 Utilities **103**

13 Projection GUI **107**
13.1 Classes . . . 107
13.1.1 MainViewerWindow . . . 107
13.1.2 OverviewDockWidget . . . 108
13.1.3 ViewingAreaDockWidget . . . 113
13.2 Widgets . . . 113
13.2.1 GUI Base Layout Classes . . . 113
13.2.2 GUI Base Widget Classes . . . 114
13.2.3 GUI Widgets Core . . . 115
13.2.4 GUI Widget Helpers . . . 115
13.2.5 GUI Preferences . . . 118
13.3 Functions . . . 128

14 Internal **129**

15 Acknowledgements **133**

Python Module Index **135**

Index **137**

This is the documentation for the *PRISM* package, an efficient and rapid alternative to MCMC methods for optimizing and analyzing scientific models. *PRISM* was made by **Ellert van der Velden** (@1313e) as part of a Ph.D under supervision of A/Prof. Alan Duffy at Swinburne University of Technology. It is written in pure Python 3 and publicly available on GitHub.

The documentation of *PRISM* is spread out over several sections:

- *User Documentation*
- *API Reference*

CHAPTER 1

Introduction

Rapid technological advancements allow for both computational resources and observational/experimental instruments to become better, faster and more precise with every passing year. This leads to an ever-increasing amount of scientific data being available and more research questions being raised. As a result, scientific models that attempt to address these questions are becoming more abundant, and are pushing the available resources to the limit as these models incorporate more complex science and more closely resemble reality.

However, as the number of available models increases, they also tend to become more distinct, making it difficult to keep track of their individual qualities. A full analysis of every model would be required in order to recognize these qualities. It is common to employ Markov chain Monte Carlo (MCMC) methods and Bayesian statistics for performing this task. However, as these methods are meant to be used for making approximations of the posterior probability distribution function, there must be a more efficient way of analyzing them.

PRISM tries to tackle this problem by using the Bayes linear approach, the emulation technique and history matching to construct an approximation ('emulator') of any given model. The use of these techniques can be seen as special cases of Bayesian statistics, where limited model evaluations are combined with advanced regression techniques, covariances and probability calculations. *PRISM* is designed to easily facilitate and enhance existing MCMC methods by restricting plausible regions and exploring parameter space efficiently. However, *PRISM* can additionally be used as a standalone alternative to MCMC for model analysis, providing insight into the behavior of complex scientific models. With *PRISM*, the time spent on evaluating a model is minimized, providing developers with an advanced model analysis for a fraction of the time required by more traditional methods.

1.1 Why use PRISM?

- Written in pure Python 3, for versatility;
- Stores results in HDF5-files, allowing for easy user-access;
- Can be executed in serial or MPI, on any number of processes;
- Compatible with Windows, Mac OS and Unix-based machines;
- Accepts any type of model and comparison data;
- Built as a plug-and-play tool: all main classes can also be used as base classes;

- Easily linked to any model by writing a single custom `ModelLink` subclass (see *ModelLink: A crash course*);
- Capable of reducing relevant parameter space by factors over 100,000 using only a few thousand model evaluations;
- Can be used alone for analyzing models, or combined with MCMC for efficient model parameter estimations.

1.2 When (not) to use PRISM?

It may look very tempting to use *PRISM* for basically everything, but keep in mind that emulation has its limits. Below is a general (but non-exhaustive) list of scenarios where *PRISM* can become really valuable:

- In almost any situation where one wishes to perform a parameter estimation using an MCMC Bayesian analysis (by using *Hybrid sampling*). This is especially true for poorly constrained models (low number of available observational constraints);
- Whenever one wishes to visualize the correlation behavior between different model parameters;
- For quickly exploring the parameter space of a model without performing a full parameter estimation. This can be very useful when trying out different sets of observational data to study their constraining power;
- For obtaining a reasonably accurate approximation of a model in very close proximity to the most optimal parameter set.

There are however also situations where one is better off using a different technique, with a general non-exhaustive list below:

- For obtaining a reasonably accurate approximation of a model in all of parameter space. Due to the way an emulator is constructed, this could easily require millions of model evaluations and a lot of time and memory;
- When dealing with a model that has a large number of parameters/degrees-of-freedom (>50). This however still heavily depends on the type of model that is used;
- Whenever a very large number of observational constraints are available and one wishes to use all of them (unless one also has access to a large supercomputer). In this case, it is a better idea to use full Bayesian instead;
- One wishes to obtain the posterior probability distribution function (PDF) of a model.

A very general and easy way to check if one should use *PRISM*, is to ask oneself the question: *"Would I use a full Bayesian analysis for this problem, given the required time and resources?"*. If the answer is 'yes', then *PRISM* is probably a good choice, especially as it requires near-similar resources as a Bayesian analysis does (definition of parameter space; provided comparison data; and a way to evaluate the model).

CHAPTER 2

Getting started

2.1 Installation

PRISM can be easily installed by either cloning the repository and installing it manually:

```
$ git clone https://github.com/1313e/PRISM
$ cd PRISM
$ pip install .
```

or by installing it directly from PyPI with:

```
$ pip install prism
```

PRISM can now be imported as a package with `import prism`. For using *PRISM* in MPI, `mpi4py >= 3.0.0` is required (not installed automatically).

The *PRISM* package comes with two ModelLink subclasses. These ModelLink subclasses can be used to experiment with *PRISM* to see how it works. *Using PRISM* and the tutorials has several examples explaining the different functionalities of the package.

2.2 Running tests

If one wants to run pytests on *PRISM*, all requirements_dev are required. The easiest way to run the tests is by cloning the repository, installing all requirements and then running `pytest` on it:

```
$ git clone https://github.com/1313e/PRISM
$ cd PRISM
$ pip install -r requirements_dev.txt
$ pytest
```

If *PRISM* and all requirements_dev are already installed, one can run the tests by running `pytest` in the installation directory:

```
$ cd <path_to_installation_directory>/prism
$ pytest
```

When using Anaconda, the installation directory path is probably of the form `<HOME>/anaconda3/envs/<environment_name>/lib/pythonX.X/site-packages`.

2.3 Example usage

See *Minimal example* or the tutorials for a documented explanation on this example.

```
# Imports
from prism import Pipeline
from prism.modellink import GaussianLink

# Define model data and create ModelLink object
model_data = {3: [3.0, 0.1], 5: [5.0, 0.1], 7: [3.0, 0.1]}
modellink_obj = GaussianLink(model_data=model_data)

# Create Pipeline object
pipe = Pipeline(modellink_obj)

# Construct first iteration of the emulator
pipe.construct()

# Create projections
pipe.project()
```

CHAPTER 3

The PRISM pipeline

The overall structure of *PRISM* can be seen in Fig. 3.1 and will be discussed below. The `Pipeline` object plays a key-role in the *PRISM* framework as it governs all other objects and orchestrates their communications and method calls. It also performs the process of history matching and refocusing (see the PRISM paper for the methodology used in *PRISM*). It is linked to the model by a user-written `ModelLink` object (see *ModelLink: A crash course*), allowing the `Pipeline` object to extract all necessary model information and call the model. In order to ensure flexibility and clarity, the *PRISM* framework writes all of its data to one or several HDF5-files using `h5py`, as well as `numpy`.

The analysis of a provided model and the construction of the emulator systems for every output value, starts and ends with the `Pipeline` object. When a new emulator is requested, the `Pipeline` object creates a large Latin-Hypercube design (LHD) of model evaluation samples to get the construction of the first iteration of the emulator systems started. To ensure that the maximum amount of information can be obtained from evaluating these samples, a custom Latin-Hypercube sampling code was written. This produces LHDs that attempt to satisfy both the *maximin* criterion as well as the *correlation* criterion. This code is customizable through *PRISM* and publicly available in the e13Tools Python package.

This Latin-Hypercube design is then given to the *Model Evaluator*, which through the provided `ModelLink` object evaluates every sample. Using the resulting model outputs, the *Active Parameters* for every emulator system (individual data point) can now be determined. Next, depending on the user, polynomial functions will be constructed by performing an extensive *Regression* process for every emulator system, or this can be skipped in favor of a sole Gaussian analysis (faster, but less accurate). No matter the choice, the emulator systems now have all the required information to be constructed, which is done by calculating the *Prior Expectation* and *Prior Covariance* values for all evaluated model samples ($\mathrm{E}(D_i)$ and $\mathrm{Var}(D_i)$).

Afterward, the emulator systems are fully constructed and are ready to be evaluated and analyzed. Depending on whether the user wants to prepare for the next emulator iteration or create a projection (see *Projections*), the *Emulator Evaluator* creates one or several LHDs of emulator evaluation samples, and evaluates them in all emulator systems, after which an *Implausibility Check* is carried out. The samples that survive the check can then either be used to construct the new iteration of emulator systems by sending them to the *Model Evaluator*, or they can be analyzed further by performing a *Projection*. The `Pipeline` object performs a single cycle by default (to allow for user-defined analysis algorithms), but can be easily set to continuously cycle.

In addition to the above, *PRISM* also features a high-level *Message Passing Interface* (MPI) implementation using the Python package `mpi4py`. All emulator systems in *PRISM* can be constructed independently from each other, in any order, and only require to communicate when performing the implausibility cut-off checks during history matching.

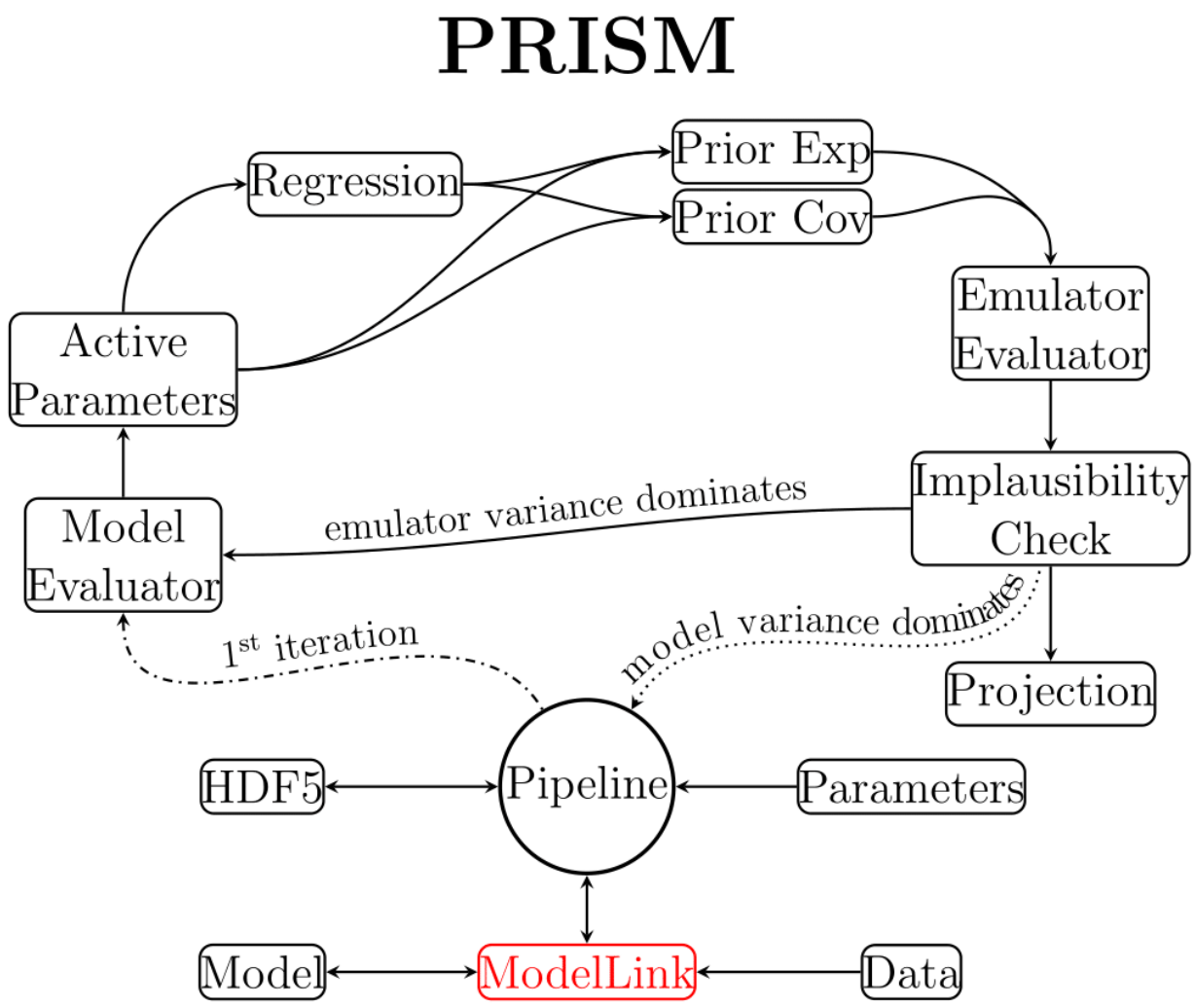


Fig. 3.1: The structure of the *PRISM* pipeline.

Additionally, since different models and/or architectures require different amounts of computational resources, *PRISM* can run on any number of MPI processes (including a single one in serial to accommodate for OpenMP codes) and the same emulator can be used on a different number of MPI processes than it was constructed on (e.g., constructing an emulator using 8 MPI processes and reloading it with 6). More details on the MPI implementation and its scaling can be found in *MPI implementation*.

In *Using PRISM* and *ModelLink: A crash course*, the various components of *PRISM* are described more extensively.

3.1 MPI implementation

Given that most scientific models are either already parallelized or could benefit from parallelization, we had to make sure that *PRISM* allows for both MPI and OpenMP coded models to be connected. Additionally, since individual emulator systems in an emulator iteration are independent of each other, the extra CPUs required for the model should also be usable by the emulator. For that reason, *PRISM* features a high-level MPI implementation for using MPI-coded models, while the Python package *NumPy* handles the OpenMP side. A mixture of both is also possible (using the `worker_mode` context manager).

Here, we discuss the MPI scaling tests that were performed on *PRISM*. For the tests, the same `GaussianLink` class was used as in *Minimal example*, but this time with 32 emulator systems (comparison data points) instead of 3. In *PRISM*, all emulator systems are spread out over the available number of MPI processes as much as possible while also trying to balance the number of calculations performed per MPI process. Since all emulator systems are stored in different HDF5-files, it is possible to reinitialize the `Pipeline` using the same `Emulator` class and `ModelLink` subclass on a different number of MPI processes. To make sure that the results are not influenced by the variation in evaluation rates, we constructed an emulator of the Gaussian model and used the exact same emulator in every test.

The tests were carried out using any number of MPI processes between 1 and 32, and using a single OpenMP thread each time for consistency. We generated a Latin-Hypercube design of $3 \cdot 10^6$ samples and measured the average evaluation rate of the emulator using the same Latin-Hypercube design each time. To take into account any variations in the evaluation rate caused by initializations, this test was performed 20 times. As a result, this Latin-Hypercube design was evaluated in the emulator a total of 640 times, giving an absolute total of $1.92 \cdot 10^9$ emulator evaluations.

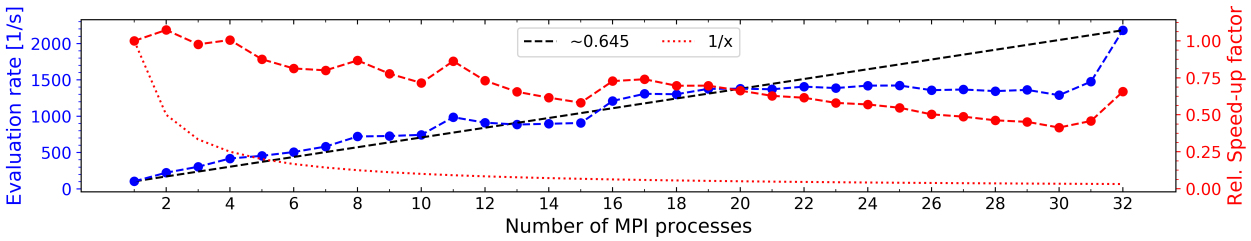


Fig. 3.2: Figure showing the MPI scaling of *PRISM* using the emulator of a simple Gaussian model with 32 emulator systems. The tests involved analyzing a Latin-Hypercube design of $3 \cdot 10^6$ samples in the emulator, determining the average evaluation rate and executing this a total of 20 times using the same sample set every time. The emulator used for this was identical in every instance. **Left axis:** The average evaluation rate of the emulator vs. the number of MPI processes it is running on. **Right axis:** The relative speed-up factor vs. the number of MPI processes, which is defined as $\frac{f(x)}{f(1)\cdot x}$ with $f(x)$ the average evaluation rate and x the number of MPI processes. **Dotted line:** The minimum acceptable relative speed-up factor, which is always $1/x$. **Dashed line:** A straight line with a slope of $\sim$0.645, connecting the lowest and highest evaluation rates. The tests were performed using the *OzSTAR computing facility* at the Swinburne University of Technology, Melbourne, Australia.

In Fig. 3.2, we show the results of the performed MPI scaling tests. On the left y-axis, the average evaluation rate vs. the number of MPI processes the test ran on is plotted, while the relative speed-up factor vs. the number of MPI processes is plotted on the right y-axis. The relative speed-up factor is defined as $f(x)/(f(1)\cdot x)$ with $f(x)$ the average

evaluation rate and x the number of MPI processes. The ideal MPI scaling would correspond to a relative speed-up factor of unity for all x.

In this figure, we can see the effect of the high-level MPI implementation. Because the emulator systems are spread out over the available MPI processes, the evaluation rate is mostly determined by the runtime of the MPI process with the highest number of systems assigned. Therefore, if the number of emulator systems (32 in this case) cannot be divided by the number of available MPI processes, the speed gain is reduced, leading to the plateaus like the one between $x = 16$ and $x = 31$. Due to the emulator systems not being the same, their individual evaluation rates are different such that a different evaluation rate has a bigger effect on the average evaluation rate of the emulator the more MPI processes there are. This is shown by the straight dashed line drawn between $f(1)$ and $f(32)$, which has a slope of ~ 0.645.

The relative speed-up factor shows the efficiency of every individual MPI process in a specific run, compared to using a single MPI process. This also shows the effect of the high-level MPI implementation, giving peaks when the maximum number of emulator systems per MPI process has decreased. The dotted line shows the minimum acceptable relative speed-up factor, which is always defined as $1/x$. On this line, the average evaluation rate $f(x)$ for any given number of MPI processes is always equal to $f(1)$.

CHAPTER 4

ModelLink: A crash course

4.1 Writing a ModelLink subclass

In *Minimal example*, a description is given of how to initialize the `Pipeline` class using a default `ModelLink` subclass. Here, the basic steps for making a custom `ModelLink` subclass are shown.

Lst. 4.1: example_link.py

```
# -*- coding: utf-8 -*-

# Future imports
from __future__ import absolute_import, division, print_function

# Package imports
import numpy as np

# PRISM imports
from prism.modellink import ModelLink


# ExampleLink class definition
class ExampleLink(ModelLink):
    # Extend class constructor
    def __init__(self, *args, **kwargs):
        # Perform any custom operations here
        pass

        # Set ModelLink flags (name, call_type, MPI_call)
        pass

        # Call superclass constructor
        super().__init__(*args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
    # Define default model parameters (optional)
    def get_default_model_parameters(self):
        par_dict = {}
        return(par_dict)

    # Define default model data (optional)
    def get_default_model_data(self):
        data_dict = {}
        return(data_dict)

    # Override call_model abstract method
    def call_model(self, emul_i, par_set, data_idx):
        # Perform operations for obtaining the model output
        # Following is provided:
        # 'emul_i': Requested iteration
        # 'par_set': Requested sample(s) dict
        # 'data_idx': Requested data point(s)
        pass

    # Override get_md_var abstract method
    def get_md_var(self, emul_i, par_set, data_idx):
        # Perform operations for obtaining the model discrepancy variance
        # Following is provided:
        # 'emul_i': Requested iteration
        # 'par_set': Requested sample dict
        # 'data_idx': Requested data point(s)
        pass
```

In the *example_link.py* file above, a minimal example of a `ModelLink` subclass is shown. It has two abstract methods that need to be overridden; `call_model()` (wrapper function for calling the model) and `get_md_var()` (calculates the model discrepancy variance). A `ModelLink` subclass cannot be initialized if either method has not been overridden. Given the importance of both methods, detailed descriptions are given in *Wrapping a model (call_model)* and *Model discrepancy variance (md_var)*, respectively.

Every `ModelLink` subclass needs to be provided with two different data sets: *model parameters* and *model data*. The model parameters define which parameters the model can take, what their names are and in what value range each parameter must be. The model data on the other hand, states where in a model realization a data value must be retrieved and compared with a provided observational value. One can think of the model data as the observational constraints used to calculate the likelihood in a Bayesian analysis. The different ways in which these two data sets can be provided are explained further in this section.

Since every model is different, with some requiring preparations in order to work properly, the `__init__()` constructor method may be extended to include any custom code to be executed when the subclass is initialized. The superclass version of the `__init__()` method must always be called, as it sets several important flags and properties, but the time at which this is done does not matter. During the initialization of the `Emulator` class, it is checked whether or not the superclass constructor of a provided `ModelLink` instance was called (to avoid this from being forgotten).

Besides executing custom code, three properties/flags can be set in `__init__()`, which have the following default values if the extended constructor does not set them:

```
self.name = self.__class__.__name__ # Set instance name to the name of the class
self.call_type = 'single'           # Request single model calls
self.MPI_call = False               # Request only controller calls
```

The first property, *name*, defines the name of the *ModelLink* instance. This name is used by the *Emulator* class during initialization to check if a constructed emulator is linked to the proper *ModelLink* instance, in order to avoid causing mismatches. If one wants to use the same *ModelLink* subclass for different models (like, using different parameter spaces), it is recommended to add an identifier for this to this name. An example of this can be found in the definition of the *GaussianLink* class, which adds the number of Gaussians in the model to its *name* property.

The other two properties, *call_type* and *MPI_call*, are flags that tell *PRISM* how the *call_model()* method should be used. By default, every model evaluation sample is requested individually in serial, since this would be the most expected behavior. However, this is most likely not enough for sophisticated models, as they can require some preparation (e.g., having to read in data files) or more than a single core (in MPI) to function. Therefore, *call_type* can be set to accept solely individual samples (`'single'`), solely entire sample sets (`'multi'`) or both (`'hybrid'`). In the same way, *MPI_call* can be set to `True` or `False` to identify that the model needs to be executed in serial or in MPI.

Note: If a model uses OpenMP parallelization, it is recommended to set *MPI_call* to `False` in the *ModelLink* subclass. This allows for all worker ranks to be used in OpenMP threads, while only the controller rank calls the model.

Finally, the *ModelLink* class has three methods that can be overridden for adding utility to the class (of which two are shown in *example_link.py*). The *get_default_model_parameters()* and *get_default_model_data()* methods return dictionaries containing the default model parameters and model data to use in this class instance, respectively. By overriding these methods, one can hard-code the use of specific parameters or comparison data, avoiding having to provide them when initializing the *ModelLink* subclass. Additionally, if a default parameter or data point is also provided during initialization, the provided information will override the defaults.

Example

The *GaussianLink* class has default parameters defined:

```
>>> from prism.modellink import GaussianLink
>>> model_data = {3: [3.0, 0.1]}
>>> modellink_obj = GaussianLink(model_data=model_data)
>>> modellink_obj
GaussianLink(model_parameters={'A1': [1.0, 10.0, 5.0], 'B1': [0.0, 10.0, 5.0],
                               'C1': [0.0, 5.0, 2.0]},
             model_data={3: [3.0, 0.1]})
```

Providing a custom set of parameters will override the coded defaults:

```
>>> model_parameters = {'A1': [-5, 7, 2]}
>>> modellink_obj = GaussianLink(model_parameters=model_parameters, model_data=model_
↪data)
>>> modellink_obj
GaussianLink(model_parameters={'A1': [-5.0, 7.0, 2.0], 'B1': [0.0, 10.0, 5.0],
                               'C1': [0.0, 5.0, 2.0]},
             model_data={3: [3.0, 0.1]})
```

The third method, *get_str_repr()*, is a simple function that returns a list containing the representations of all non-default input arguments the *ModelLink* subclass takes. It can be overridden to add the missing input arguments to the full representation of the class, which is automatically called whenever the representation is requested. The *GaussianLink* class overrides this method to add its *n_gaussians* input argument.

Lst. 4.2: line_link.py

```
# -*- coding: utf-8 -*-

# Future imports
from __future__ import absolute_import, division, print_function

# Package imports
import numpy as np

# PRISM imports
from prism.modellink import ModelLink


# LineLink class definition
class LineLink(ModelLink):
    # Extend class constructor
    def __init__(self, *args, **kwargs):
        # No custom operations or flags required
        pass

        # Call superclass constructor
        super().__init__(*args, **kwargs)

    # Define default model parameters (optional)
    def get_default_model_parameters(self):
        par_dict = {
            'A': [-10, 10, 3],  # Intercept in [-10, 10] with estimate of 3
            'B': [0, 5, 1.5]}   # Slope in [0, 5] with estimate of 1.5
        return(par_dict)

    # Define default model data (optional)
    def get_default_model_data(self):
        data_dict = {
            1: [4.5, 0.1],     # f(1) = 4.5 +- 0.1
            2.5: [6.8, 0.1],   # f(2.5) = 6.8 +- 0.1
            -2: [0, 0.1]}      # f(-2) = 0 +- 0.1
        return(data_dict)

    # Override call_model abstract method
    def call_model(self, emul_i, par_set, data_idx):
        # Calculate the value on a straight line for requested data_idx
        vals = par_set['A']+np.array(data_idx)*par_set['B']
        return(vals)

    # Override get_md_var abstract method
    def get_md_var(self, emul_i, par_set, data_idx):
        # Calculate the model discrepancy variance
        # For a straight line, this value can be set to a constant
        return(1e-4*np.ones_like(data_idx))
```

Using all the information above and the template given in *example_link.py*, a `ModelLink` subclass can be written for a straight line model, shown in the *line_link.py* file above. Here, all methods discussed before (besides the `get_str_repr()` method, since no additional input arguments are used) have been overridden. Given that this model is very simple, no changes have been made to the instance constructor, `__init__()`. Therefore, only single evaluation samples in serial are requested.

PRISM provides the *test_subclass()* function that allows the user to check if a *ModelLink* subclass is properly written. It returns an instance of the subclass if the test passes, or raises a specific error if not. We can use this function to initialize our newly written subclass:

```
>>> from line_link import LineLink
>>> from prism.modellink import test_subclass
>>> modellink_obj = test_subclass(LineLink)
>>> modellink_obj
LineLink(model_parameters={'A': [-10.0, 10.0, 3.0], 'B': [0.0, 5.0, 1.5]},
         model_data={2.5: [6.8, 0.1], -2: [0.0, 0.1], 1: [4.5, 0.1]})
```

Since no errors were raised, we can now use the initialized *ModelLink* subclass to initialize the *Pipeline* class:

```
>>> from prism import Pipeline
>>> pipe = Pipeline(modellink_obj)
```

4.2 Data identifiers (*data_idx*)

The comparison data points that are given to the *ModelLink* class each require a unique data point identifier, allowing *PRISM* to distinguish between them. This data identifier (called *data_idx*) can be used by the model wrapped in the *call_model()* method as a description of how to calculate/extract the data point. It can be provided as a non-mutable sequence (a Python tuple) of a combination of booleans; integers; floats; and strings, each element describing a part of the operations required. The data identifier sequence can be of any length, and the length can differ between data points.

Note: If a data identifier is given as a single element, then the identifier is saved as that single element instead of a tuple. For example, `data_idx = [(1), (2), (3, 4), ...]` would be saved as `data_idx = [1, 2, (3, 4), ...]`.

In its simplest form, the data identifier is a single value that is given to a function $f(x)$, which is a function that is defined for a given model parameter set and returns the function value belonging to the input x. This is the way the data identifier works for the three standard *ModelLink* subclasses; *SineWaveLink*; *GaussianLink*; and *PolyLink*. It is also used in the `LineLink` class described in the *line_link.py* file above.

For more sophisticated models, a single value/element is not enough to uniquely identify a data point. A simple example of this would be if the model generates a two-dimensional array of values, where one specific value needs to be returned. Then, the data identifier can be given as a tuple of two integers, like `data_idx = [(1, 1), (4, 8), ...]`. In the case that the model also generates several two-dimensional arrays which are named, an extra string could be used to identify this array first: `data_idx = [('array1', 1, 1), ('array4', 4, 8), ...]`.

An even more complex example is when a data point needs to be retrieved from a specific named data set at a certain point in a model simulation, after which an operation needs to be carried out (like, making a histogram of the results) and the resulting data point is then found at a specific value in that histogram. The histogram here might only be necessary to make for specific data sets, while different operations are required for others. *PRISM* allows for such complex data identifiers to be given, as it treats every sequence of data identifier elements as separated. Two different data identifiers working as described above can for example be written as `data_idx = [(14, 'array1', 'histogram', 7.5), (17, 'array7', 'average'), ...]`, where the first data point requires an extra (float) value for the histogram and the second does not. In order to do this, one would of course be required to make sure that the *call_model()* method can perform these operations when provided with the proper data identifier.

4.3 Wrapping a model (*call_model*)

The *call_model()* method is the most important method in the entire *PRISM* package. It provides the *Pipeline* instance with a way to call the model that is wrapped in the user-defined *ModelLink* subclass. For *PRISM*, this method is a black box: it takes a parameter/sample set, performs a series of unknown operations and returns the values corresponding to the requested data points and sample(s). Therefore, the *call_model()* method must be written with great care.

4.3.1 Input arguments

Depending on the values of the *multi_call* and *MPI_call* flags (where the first is set by the *call_type* flag), the *Pipeline* instance will use the *call_model()* method differently. As explained in *Writing a ModelLink subclass*, every model evaluation sample is requested individually in serial by default, which corresponds to *multi_call* is `False` and *MPI_call* is `False`. When single-calling a model, *PRISM* expects an array-like container back with shape `(n_data,)`, where the order of the elements is the same as the order of the requested `data_idx`. If we assume that we have an instance of the `LineLink` class (introduced in *line_link.py*) called `modellink_obj` and want to evaluate the model three times for all data points, then the model would be called as (solely by the controller rank):

```
# Import SortedDict class
from sortedcontainers import SortedDict as sdict

# Get emul_i, sam_set and data_idx
emul_i = 1
sam_set = np.random.rand(3, modellink_obj.n_par)
data_idx = modellink_obj.data_idx

# Evaluate model
mod_set = np.zeros([sam_set.shape[0], len(data_idx)])
for i, par_set in enumerate(sam_set):
    par_dict = sdict(zip(modellink_obj.par_name, par_set))
    mod_set[i] = modellink_obj.call_model(emul_i=emul_i,
                                          par_set=par_dict,
                                          data_idx=data_idx)
```

Here, we looped through the entire sample set one-by-one, converted every individual sample to a (sorted) dict and called the model with it. The emulator iteration is given as a normal integer and the data identifiers `data_idx` is provided as a list of individual data identifiers (which are either single elements or tuples of elements, as described in *Data identifiers (data_idx)*). The requested data identifiers are not necessarily the same as those given in *data_idx*. An individual sample provided in this way will be of the form:

```
par_dict = {'par_1_name': par_1_val,
            'par_2_name': par_2_val,
            ...,
            'par_n_name': par_n_val}
```

An example of this would be `par_dict = {'A': 1.0, 'B': 2.0}` for the `LineLink` class. This works very well for models that do not require any preparation before they can start evaluating and requires a minimal amount of effort to implement. However, if the sample set is very large, then evaluating the model in this fashion can be inefficient due to many memory look-ups.

Therefore, the *GaussianLink* class accepts both single and multi-calls. When multi-calling a model, *PRISM* expects an array-like container back with shape `(n_sam, n_data)`, where the order of the columns is the same as the order of the requested `data_idx`. So, if we use the same example again, but this time have an instance of the

GaussianLink class with *multi_call* is `True`, then the model would be called as (again solely by the controller rank):

```
# Get emul_i, sam_set and data_idx
emul_i = 1
sam_set = np.random.rand(3, modellink_obj.n_par)
data_idx = modellink_obj.data_idx

# Evaluate model
sam_dict = sdict(zip(modellink_obj.par_name, sam_set.T))
mod_set = modellink_obj.call_model(emul_i=emul_i,
                                   par_set=sam_dict,
                                   data_idx=data_idx)
```

This call is roughly the same as before, but this time the entire sample set is provided as a (sorted) dict instead of individual samples. The lay-out of this sample dict is of the form:

```
sam_dict = {'par_1_name': [par_1_val_1, par_1_val_2, ..., par_1_val_m],
            'par_2_name': [par_2_val_1, par_2_val_2, ..., par_2_val_m],
            ...,
            'par_n_name': [par_n_val_1, par_n_val_2, ..., par_n_val_m]}
```

Again, in the case of the *GaussianLink* class, this sample dict could look like `sam_dict = {'A1': [1.0, 5.5, 10.0], 'B1': [0.0, 5.0, 10.0], 'C1': [0.0, 2.5, 5.0]}`. This can be used when the model requires some kind of preparation before being able to perform evaluations, or when it is simply more efficient to provide all requested samples at once (like for the *GaussianLink* class).

Note: If a model uses OpenMP parallelization, it is recommended to set *MPI_call* to `False` in the *ModelLink* subclass. This allows for all worker ranks to be used in OpenMP threads, while only the controller rank calls the model.

Note: If one wishes to transform the received `sam_dict` back into a normal NumPy array of shape `(n_sam, n_par)`, this can be done quite easily by executing `sam_set = np.array(par_set.values()).T`, where `par_set` is the `sam_dict` provided to the *call_model()* method. Keep in mind that doing so means that the columns are sorted on the names of the model parameters. If one instead wishes to transform it into a generator, use `sam_set=map(lambda *args: args, *par_set.values())`.

New in version 1.1.2: It is also possible to make *call_model()* return a dict instead, where it has the identifiers in the requested `data_idx` as its keys and scalars (single-call) or 1D array-likes of shape `(n_sam)` (multi-call) as its values. *PRISM* will automatically convert the dict back to the array-like container format that is normally expected.

When the *MPI_call* flag is set to `True`, the calls to the *call_model()* method are almost the same as described above. The only difference is that all ranks call the method (each providing the same `emul_i`, `par_dict`/`sam_dict` and `data_idx`) instead of just the controller rank.

4.3.2 Multi-calling

When the *multi_call* flag is set to `False`, the *call_model()* method is most likely nothing more than a simple function. But, when *multi_call* is set to `True`, *call_model()* can be a lot more complex. An example of this would be if we tried to make an emulator of an emulator (which is possible, but completely pointless). In this case, it would be necessary for the "model" (as we are going to call the emulated emulator from now on) to be loaded into memory first before it can be evaluated. Although loading an emulator into memory usually does not take that long,

we do not want to do this for every single "model" evaluation. Besides, evaluating an emulator is much quicker when all samples are evaluated at once (due to the way the *_evaluate_sam_set()* method is written).

So, therefore, it is necessary to use *multi_call* is `True` for this "model". If we assume that we have already made an emulator of the `LineLink` class, then, the *call_model()* method could be written as:

```
def call_model(self, emul_i, par_set, data_idx):
    # Initialize Pipeline object as a model
    modellink_obj = LineLink()
    pipe_model = Pipeline(modellink_obj, working_dir='linelink_0')

    # Call pipe_model
    mod_set = pipe_model.evaluate(par_set, emul_i)['adj_exp_val']

    # Make sure only the requested data points are kept
    req_idx = [pipe_model.emulator._data_idx[emul_i].index(idx) for idx in data_idx]
    mod_set = mod_set[:, req_idx]

    # Return mod_set
    return(mod_set)
```

Here, we only initialize the "model" once per model call, and then evaluate all samples in it by using the *evaluate()* method (which can take sample dicts as a valid input argument). This returns a dict of the evaluation results, where we are only interested in the adjusted expectation values. Note that making an emulator of an emulator is pointless, but used here as an example.

Note: Due to the way *PRISM* is written, it is technically speaking not necessary to reinitialize the *Pipeline* class every time that *call_model()* is called. It is possible to initialize it when the corresponding *ModelLink* subclass is initialized and keep it in memory. The code above would however be necessary if the "model" works in the same way as *PRISM*'s *worker_mode*, where all worker ranks are listening for calls until the "model" is finalized. This finalization would be required in order to give *PRISM* control back over all ranks.

4.3.3 Backing up progress

New in version 1.1.1.

Warning: This feature is still experimental and it may see significant changes or be (re)moved in the future.

In *PRISM*, an emulator system is constructed by calculating all required components individually. This means that the construction process of an emulator iteration can easily be interrupted and restored at a later time, only losing the progress that was made in the current step (e.g., interrupting construction during the calculation of the covariance matrix will lose progress made there, but not the already previously finished steps). This system was implemented to accommodate for *PRISM* running on clusters, where the construction is more prone to interruptions due to, for example, jobs timing out, and to allow for *PRISM* to be loaded unto any number of MPI processes.

However, the biggest step in the construction of all emulator systems, is the evaluation of the model. Since the evaluation of the model is carried out by the *call_model()* method, *PRISM* has no control over what is happening until this method gives control back to the *Pipeline* instance (by returning the requested data points). Therefore, automated backups of already calculated data points cannot be performed by *PRISM* itself, running the risk that many CPU hours are wasted if a job on a cluster takes longer than initially expected and times out. While this could be avoided if the user writes its own backup system, this would require more work from the user, which clashes with *PRISM*'s ease-of-use policy.

Therefore, the *ModelLink* class implements its own (experimental) backup system based on the hickle package, given by the *_make_backup()* and *_read_backup()* methods. This backup system is best used for models that are multi-called (*multi_call* set to `True`), as made backups will replace previous ones (of the same type). The *_make_backup()* method is meant to be used from within the *call_model()* method and will not work if called anywhere else. Attempting to call it incorrectly (e.g., not from within *call_model()* or with incorrect arguments), will raise a *RequestWarning* and simply return without doing anything, rather than raising a *RequestError*. This is to make sure that using it incorrectly does not disrupt the *call_model()* call, as that has the exact opposite effect of what the backup system tries to achieve.

The *_make_backup()* method takes two arguments, `*args` and `**kwargs`, of which at least one is required. Calling it from within the *call_model()* method will produce an HDF5-file containing the `emul_i`, `par_set` and `data_idx` argument values that were used to call *call_model()* with, and the supplied `*args` and `**kwargs`. The name of the HDF5-file contains the values of `emul_i` and *name*, and will be saved in the current working directory (NOT the emulator working directory, as the *ModelLink* instance has no access to its path). The backup can be read in by passing the value of `emul_i` to the *_read_backup()* method of the corresponding *ModelLink* instance, which will return a dict containing the values of the five arguments that were saved to the file.

Backups can be made at any point during the execution of *call_model()*, and basically all types of objects are compatible and can be viewed freely in the HDF5-file. It is possible that instances of certain custom classes may not be supported by the hickle package, in which case they will be pickled and saved as a string, causing them to not be able to be viewed freely (but they can still be backed up). Depending on the size of the data provided, it can sometimes take a little while before a backup is made. Therefore, it is probably best to trigger making backups at specified progress points in *call_model()*.

To illustrate how this backup system can be used, assume that we have written a *ModelLink* subclass, which requires some preparation before it can start evaluating the wrapped model. Here, we will assume that this preparation is provided by a function called `prepare_model()`, which returns an instance of some class that can be used to evaluate the model after the preparation is completed. Then, we could incorporate the backup system by writing a *call_model()* method like this:

```
def call_model(self, emul_i, par_set, data_idx):
    # Prepare the model for evaluation
    model = prepare_model()

    # Controller performs evaluations
    if model.is_controller:
        # Initialize empty array of results
        mod_set = np.zeros([len(par_set['par1']), len(data_idx)])

        # Unpack par_set into a NumPy array
        sam_set = np.array(par_set.values()).T

        # Call model for every individual sample in sam_set
        for i, sam in enumerate(sam_set):
            mod_set[i] = model.evaluate(sam, data_idx)

            # Make a backup every 500 evaluations
            if not((i+1) % 500):
                self._make_backup(mod_set=mod_set[:i])

    # Finalize the model
    model.finalize()

    # Return the results on the controller
    if model.is_controller:
        return(mod_set)
```

The code above shows an example of a model that needs to be initialized before it can be multi-called in MPI, and

needs to be finalized afterward. Since such a model is probably quite complex, it may be a good idea to make a backup every once in a while. Therefore, whenever 500 evaluations have been done, a backup is made of all results gained up to that point. This means that whenever the model evaluation process is interrupted, a maximum of the last 500 evaluations is lost. The evaluations that are not lost can be loaded back in by using the *_read_backup()* method, and potentially (after a bit of formatting) be passed to the `ext_real_set` input argument of the *construct()* method when attempting to construct the emulator iteration again.

Note that if `model.evaluate()` was implemented such that it takes the entire sample set at once rather than one at a time, calling *_make_backup()* in `model.evaluate()` works perfectly fine, as long as `model.evaluate()` is always called by *call_model()* or any other function for which this is true. Put a little bit more simple: *_make_backup()* must be called either directly or indirectly by *call_model()*, as shown in the following example.

Example

```
def call_model(self, emul_i, par_set, data_idx):
    # Call a function A and return its output
    # This function does not require emul_i, so do not provide it
    return(A(self, par_set, data_idx))

def A(modellink_obj, par_set, data_idx):
    # Prepare model
    model = prepare_model()

    # Prepare par_set for evaluation
    sam_set = np.array(par_set.values()).T

    # Call a function B
    mod_set = B(modellink_obj, model, sam_set, data_idx)

    # Finalize the model
    model.finalize()

    # Return the results
    return(mod_set)

def B(modellink_obj, model_obj, sam_set, data_idx):
    # Prepare mod_set
    mod_set = np.zeros([np.shape(sam_set)[0], len(data_idx)])

    # Call model for every individual sample in sam_set
    for i, sam in enumerate(sam_set):
        mod_set[i] = model_obj.evaluate(sam, data_idx)

        # Make a backup every 500 evaluations
        if not((i+1) % 500):
            modellink_obj._make_backup(mod_set=mod_set[:i+1])

    # Return mod_set
    return(mod_set)
```

4.4 Model discrepancy variance (*md_var*)

Of the three different variances that are used for calculating the implausibility values of a parameter set, the *model discrepancy variance* is by far the most important. The model discrepancy variance describes all uncertainty about the correctness of the model output that is caused by the model itself. This includes the accuracy of the code implementation, completeness of the inclusion of the involved physics, made assumptions and the accuracy of the output itself, amongst others. It therefore acts as a measure of the quality of the model that is being emulated by *PRISM*, and as with `call_model()`, must be handled with great care.

4.4.1 Theory

When *PRISM* constructs an emulator, it attempts to make a perfect approximation of the model that covers the absolute plausible regions of parameter space. This perfect approximation would be reached if the adjusted emulator variance (*adj_var*) is zero for all samples. In this case, the emulator has the same variance associated with it as the model, which is given by the model discrepancy variance. Therefore, if the model discrepancy variance is determined incorrectly, the emulator itself will be incorrect as well.

The reason for this is as follows. The implausibility value of a parameter set states how many standard deviations the emulator system expects the model realization corresponding to this parameter set, to be away from explaining the model comparison data. When the total variance increases, the implausibility value decreases (since less standard deviations fit in the total difference). For an emulator system that is still very inaccurate (e.g., first iteration), the adjusted emulator variance dominates over the other two variances. However, later on, the adjusted emulator variance becomes less and less dominant, causing the other two variances to start playing a role. In most cases, it is safe to assume that the model discrepancy variance is higher than the observational variance, since a model would be fitting noise if this was not the case. Therefore, there is going to be a moment when the model discrepancy variance starts being close to the adjusted emulator variance.

When this happens, the plausible region of parameter space starts being determined by the model discrepancy variance. If the model discrepancy variance is generally higher than it should be, then this will often result into the emulator system not converging as far as it could have, since parts of parameter space are still marked as plausible. The opposite however (the model discrepancy variance generally being lower than it should be) can mark parts of parameter space as implausible while they are not. This means that these parts are removed from the emulator.

From the above, it becomes clear that overestimating the model discrepancy variance is much less costly than underestimating its value. It is therefore important that this variance is properly described at all times. However, since the description of the model discrepancy variance can take a large amount of time, *PRISM* uses its own default description in case none was provided, which is defined as $\mathrm{Var}(\epsilon_{\mathrm{md},i}) = (z_i/6)^2$, where $\mathrm{Var}(\epsilon_{\mathrm{md},i})$ is the model discrepancy variance of a specified model comparison data point i and z_i is the corresponding data value. If one assumes that a model output within half of the data is considered to be acceptable, with acceptable being defined as the 3σ-interval, then the model discrepancy variance is obtained as:

$$\begin{aligned}
[z_i - 3\sigma, z_i + 3\sigma] &= \left[\frac{1}{2}z_i, \frac{3}{2}z_i\right], \\
6\sigma &= z_i, \\
\sigma &= \frac{z_i}{6}, \\
\mathrm{Var}(\epsilon_{\mathrm{md},i}) = \sigma^2 &= \left(\frac{z_i}{6}\right)^2.
\end{aligned}$$

This description of the model discrepancy variance usually works well for simple models, and acts as a starting point within *PRISM*. When models become bigger and more complex, it is likely that such a description is not enough. Given that the model discrepancy variance is unique to every model and might even be different for every model output, *PRISM* cannot possibly cover all scenarios. It is therefore advised that the model discrepancy variance is provided externally by the user.

4.4.2 Implementation

The model discrepancy variance is given by the *get_md_var()* method. This method is, like *call_model()*, an abstract method and must be overridden by the *ModelLink* subclass before it can be initialized. The *get_md_var()* method is called every time the implausibility value of an emulator evaluation sample is determined. Unlike the *call_model()* method, the *get_md_var()* method is called by individual emulator systems, as they determine implausibility values individually.

For this reason, the *get_md_var()* method is provided with the emulator iteration `emul_i`, a single parameter set `par_set` and the data identifiers requested by the emulator system `data_idx`. The *call_type* and *MPI_call* flags have no influence on the way the *get_md_var()* method is used, as it is always called in serial for a single parameter set. When it is called, *PRISM* expects an array-like container back with shape `(n_data)` (if 1σ-interval is centered) or shape `(n_data, 2)` (if 1σ-interval is given by upper and lower errors), where the order of the elements is the same as the order of the requested `data_idx`. The default model discrepancy variance description given above is used if the *get_md_var()* method raises a `NotImplementedError`, but this is discouraged.

Warning: Because the *get_md_var()* method is always called for single parameter sets, it is important that it can be called without requiring any preparation of data or models.

New in version 1.1.2: It is also possible to make *get_md_var()* return a dict instead, where it has the identifiers in the requested `data_idx` as its keys and scalars (centered) or 1D array-likes of shape `(2)` (non-centered) as its values. *PRISM* will automatically convert the dict back to the array-like container format that is normally expected.

CHAPTER 5

Using PRISM

Here, various different aspects of how the *PRISM* package can be used are described.

5.1 Minimal example

A minimal example on how to initialize and use the *PRISM* pipeline is shown here. First, one has to import the *Pipeline* class and a *ModelLink* subclass:

```
>>> from prism import Pipeline
>>> from prism.modellink import GaussianLink
```

Normally, one would import a custom-made *ModelLink* subclass, but for this example one of the two *ModelLink* subclasses that come with the *PRISM* package is used (see *Writing a ModelLink subclass* for the basic structure of writing a custom *ModelLink* subclass).

Next, the *ModelLink* should be initialized, which is the *GaussianLink* class in this case. In addition to user-defined arguments, every *ModelLink* subclass takes two optional arguments, *model_parameters* and *model_data*. The use of either one will add the provided parameters/data to the default parameters/data defined in the class. Since the *GaussianLink* class does not have default data defined, it is required to supply it with some data during initialization (using an array, dict or external file):

```
>>> # f(3) = 3.0 +- 0.1, f(5) = 5.0 +- 0.1, f(7) = 3.0 +- 0.1
>>> model_data = {3: [3.0, 0.1], 5: [5.0, 0.1], 7: [3.0, 0.1]}
>>> modellink_obj = GaussianLink(model_data=model_data)
```

Here, the *GaussianLink* class was initialized by giving it three custom data points and using its default parameters. One can check this by looking at the representation of this *GaussianLink* object:

```
>>> modellink_obj
GaussianLink(model_parameters={'A1': [1.0, 10.0, 5.0], 'B1': [0.0, 10.0, 5.0],
                               'C1': [0.0, 5.0, 2.0]},
             model_data={7: [3.0, 0.1], 5: [5.0, 0.1], 3: [3.0, 0.1]})
```

The *Pipeline* class takes several optional arguments, which are mostly paths and the type of *Emulator* class that must be used. It also takes one mandatory argument, which is an instance of the *ModelLink* subclass to use. Since it has already been initialized above, the *Pipeline* class can be initialized:

```
>>> pipe = pipeline(modellink_obj)
>>> pipe
Pipeline(GaussianLink(model_parameters={'A1': [1.0, 10.0, 5.0], 'B1': [0.0, 10.0, 5.
↪0],
                                        'C1': [0.0, 5.0, 2.0]},
                      model_data={7: [3.0, 0.1], 5: [5.0, 0.1], 3: [3.0, 0.1]}),
         working_dir='prism_0')
```

Since no working directory was provided to the *Pipeline* class and none already existed, it automatically created one (`prism_0`).

PRISM is now completely ready to start emulating the model. The *Pipeline* allows for all steps in a full cycle (see *PRISM pipeline*) to be executed automatically:

```
>>> pipe.run()
```

which is equivalent to:

```
>>> pipe.construct(analyze=False)
>>> pipe.analyze()
>>> pipe.project()
```

This will construct the next iteration (first in this case) of the emulator, analyze it to check if it contains plausible regions and make projections of all active parameters. The current state of the *Pipeline* object can be viewed by calling the *details()* method (called automatically after most user-methods), which gives an overview of many properties that the *Pipeline* object currently has.

This is all that is required to construct an emulator of the model of choice. All user-methods, with one exception (*evaluate()*), solely take optional arguments and perform the operations that make the most sense given the current state of the *Pipeline* object if no arguments are given. These arguments allow for one to modify the performed operations, like reconstructing/reanalyzing previous iterations, projecting specific parameters, evaluating the emulator and more.

5.2 Projections

After having made an emulator of a given model, *PRISM* can show the user the knowledge it has about the behavior of this model by making *projections* of the active parameters in a specific emulator iteration. These projections are created by the *project()* method, which has many different properties and options. For showing them below, the same emulator as the one in *Minimal example* is used.

5.2.1 Properties

Projections (and their figures) are made by analyzing a large set of evaluations samples. For 3D projections, this set is made up of a grid of *proj_res* x *proj_res* samples for the plotted (active) parameters, where the values for the remaining parameters in every individual grid point are given by an LHD of *proj_depth* samples. This gives the total number of analyzed samples as *proj_res* x *proj_res* x *proj_depth*.

Every sample in the sample set is then analyzed in the emulator, saving whether or not this sample is plausible and what the implausibility value at the first cut-off is (the first value in *impl_cut*). This yields *proj_depth* results per grid point, which can be used to determine the fraction of samples that is plausible and the minimum implausibility

value at the first cut-off in this point. Doing this for the entire grid and interpolating them, creates a map of results that is independent of the values of the non-plotted parameters. For 2D projections, it works the same way, except that only a single active parameter is plotted.

Note: When using a 2D model, the *projection depth* used to make a 2D projection will be `proj_depth`, which is to be expected. However, when using an nD model, the projection depth of a 2D projection is equal to `proj_res` x `proj_depth`. This is to make sure that for an nD model, the density of samples in a 2D projection is the same as in a 3D projection.

The `project()` method solely takes optional arguments. Calling it without any arguments will produce six projection figures: three 2D projections and three 3D projections. One of each type is shown below.

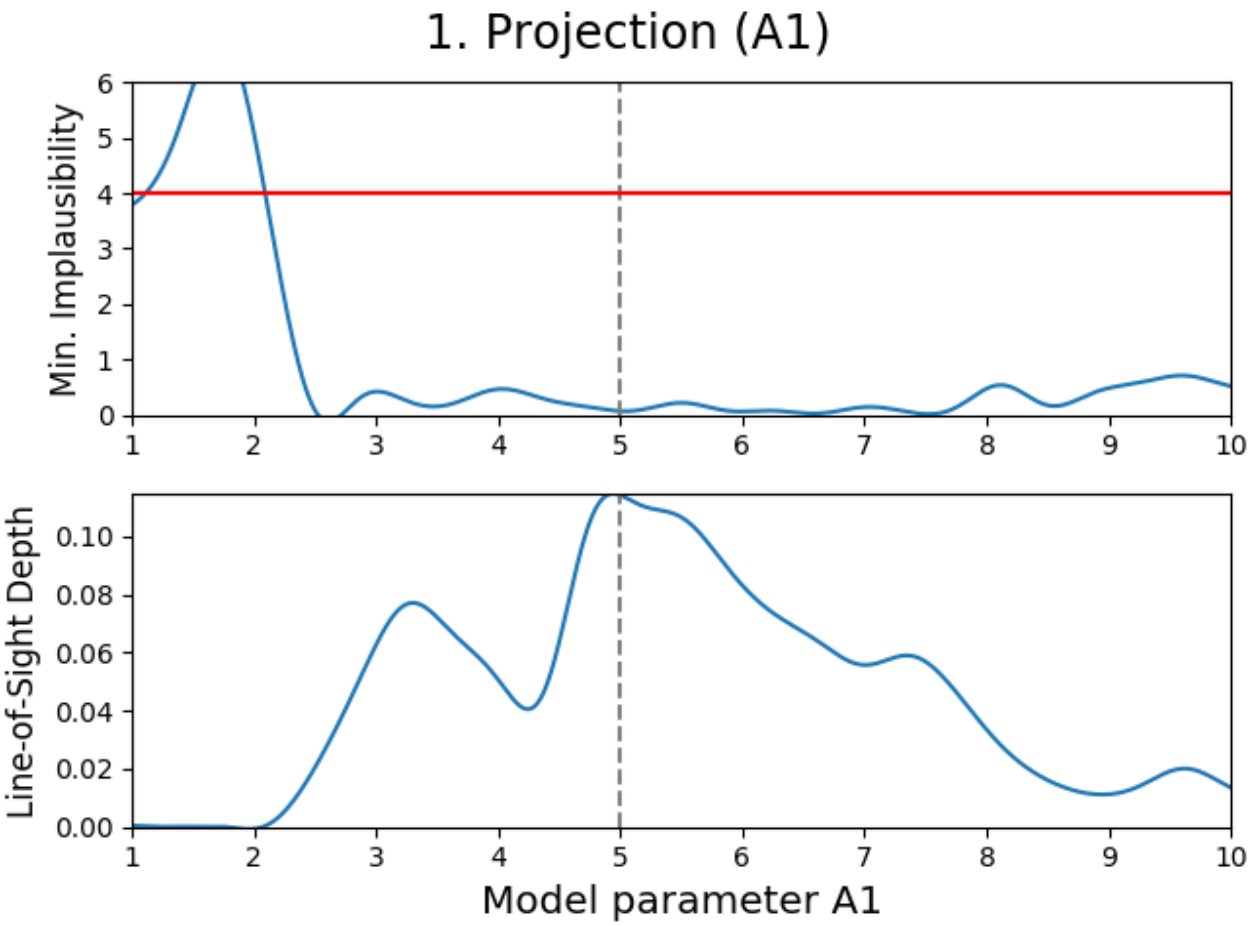


Fig. 5.1: 2D projection figure of model parameter A_1. The vertical dashed line shows the parameter estimate of A_1, whereas the horizontal red line shows the first implausibility cut-off value.

A projection figure is made up of two subplots. The upper subplot shows a map of minimum implausibility values that can be reached for any given value (combination) of the plotted parameter(s). The lower subplot gives a map of the fraction of samples that is plausible in a specified point on the grid (called "line-of-sight depth" due to the way it is calculated). Another way of describing this map is that it gives the probability that a parameter set with given plotted value(s) is plausible.

Both projection types have a different purpose. A 3D projection gives insight into what the dependencies (or correlations) are between the two plotted parameters, by showing where the best (top) and most (bottom) plausible samples

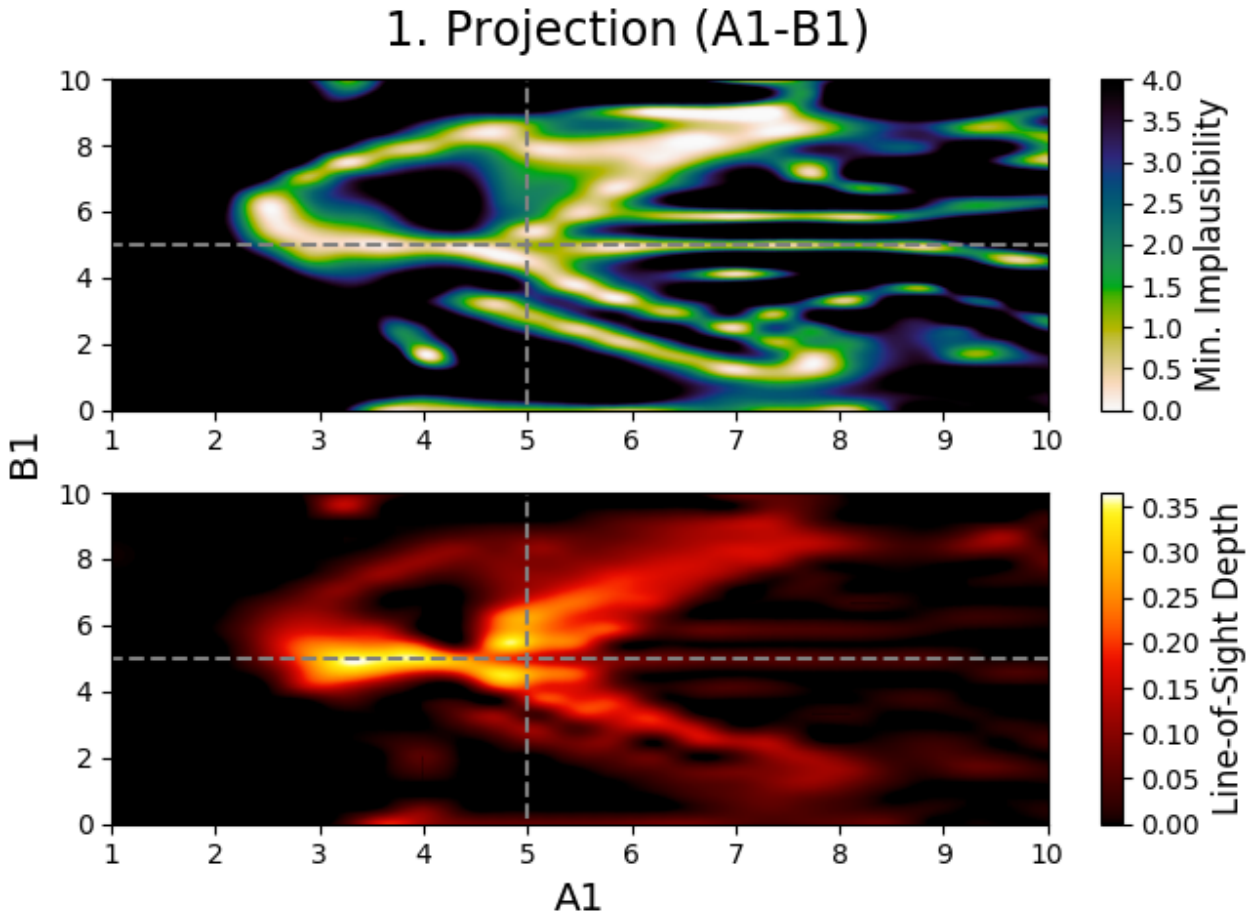


Fig. 5.2: 3D projection figure of model parameters A_1 and B_1. The dashed lines show the estimates of both parameters.

can be found. On the other hand, a 2D projection is quite similar in meaning to a *maximum likelihood* optimization performed by MCMC methods, with the difference being that the projection is based on expectations rather than real model output. A combination of both subplots allows for many model properties to be derived, especially when they do not agree with each other.

5.2.2 Options

The `project()` method takes two (optional) arguments, *emul_i* and *proj_par*. The first controls which emulator iteration should be used, while the latter provides the model parameters of which projections need to be made. Since it only makes sense to make projections of active parameters, all passive parameters are filtered out of *proj_par*. The remaining parameters are then used to determine which projections are required (which also depends on the requested projection types). For example, if one wishes to only obtain projections of the A_1 and B_1 parameters (which are both active) in iteration 1, then this can be done with:

```
>>> pipe.project(1, ('A1', 'B1'))
```

This would generate the figures shown above, as well as the 2D projection figure of B_1. By default, the last constructed emulator iteration and all model parameters are requested.

The remaining input arguments can only be given as keyword arguments, since they control many different aspects of the `project()` method. The *proj_type* argument controls which projection types to make. For 2D models, this is always *'2D'* and cannot be modified. However, for nD models, this can be set to *'2D'* (only 2D projections), *'3D'* (only 3D projections) or *'both'* (both 2D and 3D projections). By default, it is set to *'both'*.

The *figure* argument is a bool, that determines whether or not the projection figures should be created after calculating the projection data. If `True`, the projection figures will be created and saved, which is done by default. If `False`, the data that is contained within the projection figures will be calculated and returned in a dict. This allows the user to either let *PRISM* create the projection figures using the standard template or create the figures themselves.

The *align* argument controls the alignment of the subplots in every projection figure. By default, it aligns the subplots in a column (*'col'*), as shown in the figures above. Aligning the subplots in a row (*'row'*) would give Fig. 5.1 as the figure below.

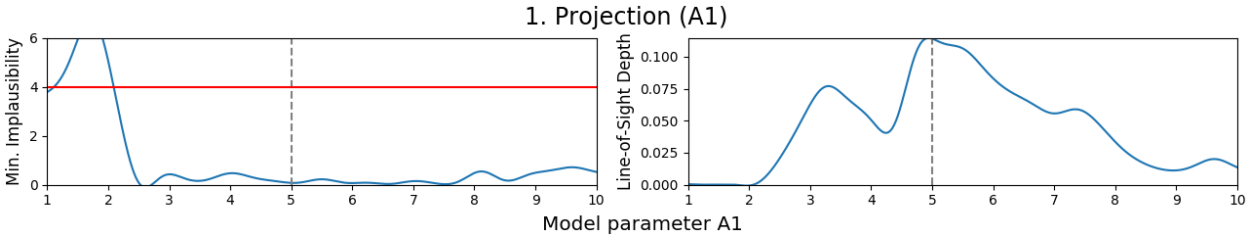


Fig. 5.3: 2D projection figure of model parameter A_1 with the *'row'* alignment.

New in version 1.1.2: The *show_cuts* argument is also a bool, that determines whether to show all implausibility cut-off values in 2D projections (`True`) or only the first cut-off value (`False`, default). In some cases, this may be useful when the first cut-off is not definitive in accepting or rejecting parameter values (as explained below for the *smooth* parameter).

The *smooth* argument is yet another bool, that determines what to do if a grid point in the projection figure contains no plausible samples, but does contain a minimum implausibility value below the first non-wildcard cut-off. If `False`, which is the default, these values are kept in the figure, which may show up as artifact-like features. If `True`, these values are set to the first cut-off, basically removing them from the projection figure. This may however also remove interesting features. Below are two identical projections, one that is smoothed and one that is not, to showcase this difference (these projections are from the second iteration, since this effect rarely occurs in the first iteration).

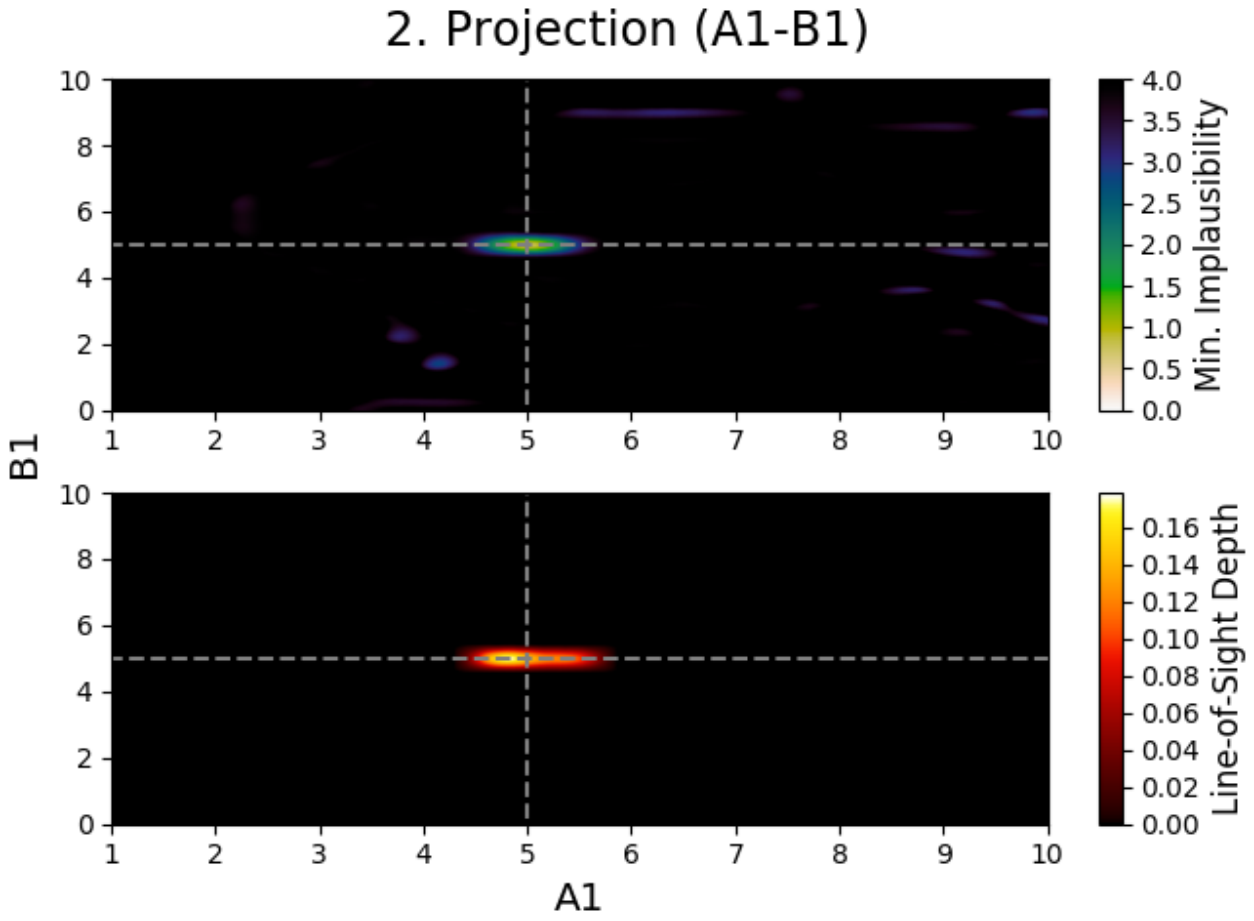


Fig. 5.4: Non-smoothed 3D projection figure of model parameters A_1 and B_1.

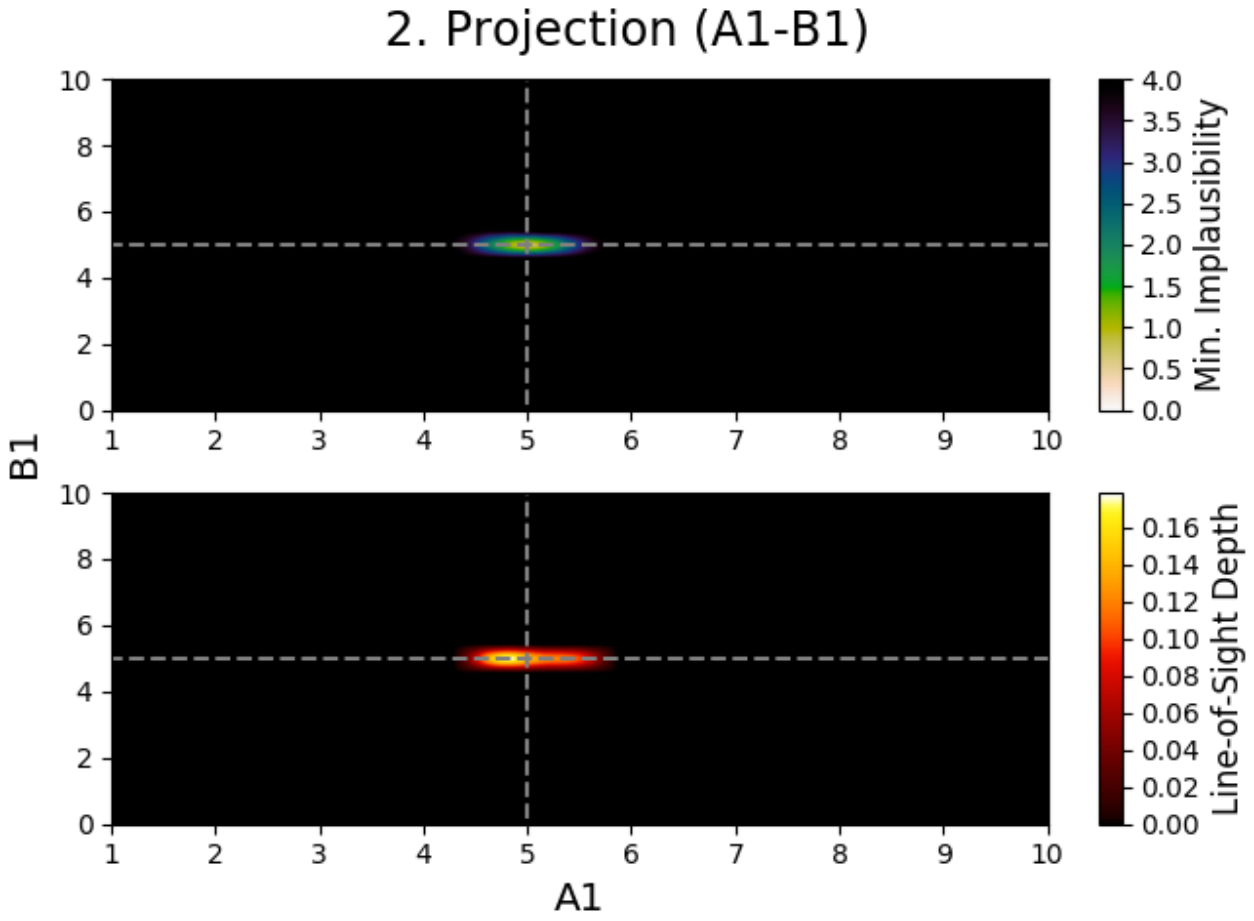


Fig. 5.5: Smoothed 3D projection figure of model parameters A_1 and B_1.

In these figures, one can see that the non-smoothed projection shows many features in the upper subplot that look like artifacts. These features are however not artifacts, but caused by a sample (or samples) having its highest implausibility value being below the first implausibility cut-off, but still being implausible due to failing a later cut-off. For example, if the implausibility cut-offs are `[4.0, 3.7, 3.5]` and a sample has implausibility values `[3.9, 3.8, 3.2]`, it is found implausible due to failing to meet the second cut-off. However, since the first value is still the highest implausibility value, that value is used in the projection figure. Smoothing figures usually allows for 3D projections (2D projections rarely show this) to become less crowded, but they do throw away information. It should therefore only be used when necessary.

New in version 1.2.3: The *use_par_space* argument is also a bool, which controls whether the axes limits should be given by the model parameter space limits (`True`) or by the parameter space over which the emulator iteration is defined (`False`, default). If `False`, only the parameter space in which the emulator is defined is shown, which more often than not allows for more details to be seen in the projection figure. If `True`, the full model parameter space is used for the axes limits, allowing projection figures from different iterations to be compared with each other more easily.

New in version 1.2.4: The *full_impl_rng* argument is a bool, which controls whether the lower limit for the minimum implausibility plot should be set to zero (`True`) or to the lowest implausibility value plotted (`False`), with the latter being the default. If `False`, only the implausibility range in which plausible samples are plotted will be used, which increases the amount of detail in the minimum implausibility plot. If `True`, the entire implausibility range that can hold plausible samples is used, allowing for the relative implausibility difference between samples to be interpreted correctly. Set this value to `True` if one wishes to have the same plotting behavior as in versions earlier than v1.2.4.

The *force* argument is a bool, which controls what to do if a projection is requested for which data already exists. If `False` (default), it will use the previously acquired projection data to create the projection figure if it does not exist, skip if it does or return the figure data if *figure* is `False`. If `True`, the projection data and all associated projection figures will be deleted, and the projection will be recalculated.

The remaining eight arguments are keyword argument dicts, that need to be passed to the various different plotting functions that are used for creating the projection figures. The *fig_kwargs* dict is passed to the `figure()` function when creating the projection figure instance. The *impl_kwargs_2D* and *los_kwargs_2D* dicts are passed to the `plot()` function when making the minimum implausibility and line-of-sight depth subplots, respectively, for the 2D projections. Similarly, the *impl_kwargs_3D* and *los_kwargs_3D* dicts are passed to the `hexbin()` function for 3D projections. The *line_kwargs_est* and *line_kwargs_cut* dicts are passed to the `draw()` function for drawing the parameter estimate and implausibility cut-off lines. And, finally, the *arrow_kwargs_est* dict is passed to the `arrow()` function for drawing the parameter estimate arrow in case its line is out-of-range.

5.2.3 Crystal (GUI)

New in version 1.2.0.

PRISM also has an internal GUI (graphical user-interface) for creating; viewing; comparing; and analyzing projection figures, called *Crystal*. *Crystal* can be started from any `Pipeline` object by using the `crystal()` method.

5.3 Dual nature (normal/worker mode)

PRISM features a high-level MPI implementation, as described in *MPI implementation*: all user-methods and most major methods are to be executed by all MPI ranks at the same time, and *PRISM* will automatically distribute the work among the available ranks within this function/method. This allows for *PRISM* to be used with both serial and parallel models, by setting the `MPI_call` flag accordingly, while also allowing for the same code to be used in serial and parallel. However, given that the emulator of *PRISM* can be very useful for usage in other routines, like *Hybrid sampling*, an external code will call *PRISM*'s methods. In order to use *PRISM* in parallel with a parallelized model, this code would have to call *PRISM* with all MPI ranks simultaneously at all times, which may not always be possible (e.g., when using MCMC methods).

Therefore, *PRISM* has a *dual execution/call nature*, where it can be switched between two different modes. In the default mode, *PRISM* works as described before, where all MPI ranks call the same user-code. However, by using the *WorkerMode* context manager, accessed through *worker_mode()*, all code within will be executed in worker mode. When in worker mode, all worker ranks are continously listening for calls from the controller rank, made with the *_make_call()* and *_make_call_workers()* methods. They will continue to do so until the controller exits *WorkerMode* with *__exit__()*. Manually exiting should solely be done in advanced use-cases.

In *worker_mode*, one uses the following structure (assuming that the *Pipeline* instance is called `pipe`):

```
# Code to be executed in default mode

with pipe.worker_mode:
    if pipe.is_controller:
        # Code to be executed in worker mode

# More code to be executed in default mode
```

Note: All code that is inside the *worker_mode* context manager should solely be executed by the controller rank. If not, all worker ranks will execute this code after the controller ranks exits the context manager. Currently, it is not possible to make a context manager handle this automatically (the rejected **PEP 377** describes this perfectly).

The *_make_call()* method accepts almost anything that can be called. It can also be used when not in *worker_mode*, in which case it works the exact same way for all MPI ranks. Its sole limitation is that all supplied arguments must be pickleable (e.g., compiled code objects are NOT pickleable due to safety reasons), both when used in *worker_mode* and outside of it. The `copyreg` module can be used to register specific objects to become pickleable (including compiled code objects).

The *worker_mode* can be used in a variety of ways, as described below. It can be used to access any attribute of the *Pipeline* instance:

```
with pipe.worker_mode:
    if pipe.is_controller:
        # Construct first emulator iteration
        pipe._make_call('construct', 1)

        # Print latest constructed emulator iteration
        print(pipe._make_call('emulator._get_emul_i', 1, 0))

        # Make a specific projection with the 'row' alignment
        pipe._make_call('project', 1, (0, 1), align='row')
```

which is equivalent to:

```
# Construct first emulator iteration
pipe.construct(1)

# Print latest constructed emulator iteration
print(pipe.emulator._get_emul_i(1, 0))

# Make a specific projection with the 'row' alignment
pipe.project(1, (0, 1), align='row')
```

The above two code snippets are equal to each other, and the *worker_mode* will most likely be used very rarely in this fashion. However, by supplying the *_make_call()* method with a callable function (that can be pickled), externally defined functions can be executed:

```
# Enable worker mode
with pipe.worker_mode:
    if pipe.is_controller:
        # Import print function that prepends MPI rank to message
        from prism._internal import rprint

        # Make call to use this function
        # Equivalent to 'rprint("Reporting in.")'
        pipe._make_call(rprint, "Reporting in.")
```

This is especially useful when one combines a serial code with *PRISM*, but wants *PRISM* to execute in MPI. An application example of this is *Hybrid sampling*.

Changed in version 1.2.0: It is also possible to make a call that is solely executed by the workers, by using the *_make_call_workers()* method.

Changed in version 1.2.0: If any positional or keyword argument is a string written as `'pipe.XXX'`, it is assumed that `'XXX'` refers to a *Pipeline* attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

Changed in version 1.2.0: Initializing a worker mode within an already existing worker mode is possible and will function properly. An example of this is using the *construct()* or *crystal()* method within worker mode, as both use one themselves as well.

5.4 Hybrid sampling

A common problem when using MCMC methods is that it can often take a very long time for MCMC to find its way on the posterior probability distribution function, which is often referred to as the *burn-in phase*. This is because, when considering a parameter set, there is usually no prior information that this parameter set is (un)likely to result into a desirable model realization. This means that such a parameter set must first be evaluated in the model before any probabilities can be calculated. However, by constructing an emulator of the model, one can use it as an additional prior for the posterior probability calculation. Therefore, although *PRISM* is primarily designed to make analyzing models much more efficient and accessible than normal MCMC methods, it is also very capable of enhancing them. This process is called *hybrid sampling*, which can be performed easily with the *utils* module and will be explained below. Note that an interactive version of this section can be found in the tutorials.

5.4.1 Algorithm

Hybrid sampling allows one to use *PRISM* to first analyze a model's behavior, and later use the gathered information to speed up parameter estimations (by using the emulator as an additional prior in a Bayesian analysis). Hybrid sampling works in the following way:

1. Whenever an MCMC walker proposes a new sample, it is first passed to the emulator of the model;
2. If the sample is not within the defined parameter space, it automatically receives a prior probability of zero (or $-\infty$ in case of logarithmic probabilities). Else, it will be evaluated in the emulator;
3. If the sample is labeled as *implausible* by the emulator, it also receives a prior probability of zero. If it is plausible, the sample is evaluated in the same way as for normal sampling;
4. Optionally, a scaled value of the first implausibility cut-off is used as an exploratory method by adding an additional (non-zero) prior probability. This can be enabled by using the *impl_prior* input argument for the *get_hybrid_lnpost_fn()* function.

Since the emulator that *PRISM* makes of a model is not defined outside of the parameter space given by *par_rng*, the second step is necessary to make sure the results are valid. There are several advantages of using hybrid sampling over normal sampling:

- Acceptable samples are guaranteed to be within plausible space;
- This in turn makes sure that the model is only evaluated for plausible samples, which heavily reduces the number of required evaluations;
- No burn-in phase is required, as the starting positions of the MCMC walkers are chosen to be in plausible space;
- As a consequence, varying the number of walkers tends to have a much lower negative impact on the convergence probability and speed;
- Samples with low implausibility values can optionally be favored.

5.4.2 Usage

In order to help the user with combining *PRISM* with MCMC to use hybrid sampling, the *utils* module provides two functions: *get_walkers()* and *get_hybrid_lnpost_fn()*. The *get_walkers()* function analyzes a set of proposed *init_walkers* and returns the positions that are plausible (and the number of positions that are plausible). By default, it uses the available *impl_sam* of the last constructed iteration, but it can also be supplied with a custom set of proposed walkers or an integer stating how many proposed positions the function should check:

```
>>> # Use impl_sam if it is available
>>> n, p0 = get_walkers(pipe)

>>> # Request 2000 proposed samples
>>> n_walkers = 2000
>>> n, p0 = get_walkers(pipe, init_walkers=n_walkers)

>>> # Use custom init_walkers
>>> from e13tools.sampling import lhd
>>> init_walkers = lhd(n_walkers, pipe.modellink.n_par, pipe.modellink.par_rng)
>>> n, p0 = get_walkers(pipe, init_walkers=init_walkers)

>>> # Request 100 plausible starting positions (requires v1.1.4 or later)
>>> n, p0 = get_walkers(pipe, req_n_walkers=100)
```

As *PRISM*'s sampling methods operate in parameter space, the *get_walkers()* function automatically assumes that all starting positions are defined in parameter space. However, as some sampling methods use unit space, normalized starting positions can be requested by setting the *unit_space* input argument to `True`. One has to keep in mind that, because of the way the emulator works, there is no guarantee for a specific number of plausible starting positions to be obtained. Having the desired emulator iteration already analyzed may give an indication how many starting positions in total need to be proposed to be left with a specific number.

Changed in version 1.2.0: It is now possible to request a specific number of plausible starting positions by using the *req_n_walkers* input argument. This will use a custom Metropolis-Hastings sampling algorithm to obtain the required number of starting positions, using the plausible samples in *init_walkers* as the start of every MCMC chain.

When the initial positions of the MCMC walkers have been determined, one can use them in an MCMC parameter estimation algorithm, avoiding the burn-in phase. This in itself can already be very useful, but it does not allow for hybrid sampling yet. Most MCMC methods require the definition of an *lnpost()* function, which takes a parameter set and returns the corresponding natural logarithm of the posterior probability. In order to do hybrid sampling, this *lnpost()* function must have the algorithm described above implemented.

The *get_hybrid_lnpost_fn()* function factory provides exactly that. It takes a user-defined *lnpost()* function (as *lnpost_fn*) and a *Pipeline* object, and returns a function definition `hybrid_lnpost(par_set, *args,`

`**kwargs)`. This *hybrid_lnpost()* function first analyzes a proposed *par_set* in the emulator, passes *par_set* (along with any additional arguments) to *lnpost()* if the sample is plausible, or returns $-\infty$ if it is not. The return-value of the *lnpost()* function is then returned by the *hybrid_lnpost()* function as well. To make sure that the *hybrid_lnpost()* function can be used in both execution modes (see *Dual nature (normal/worker mode)*), all parallel calls to the `Pipeline` object are done with the `_make_call()` method.

The use of a function factory here allows for all input arguments to be validated once and then saved as local variables for the *hybrid_lnpost()* function. Not only does this avoid that all arguments have to be provided and validated for every individual call, but it also ensures that the same arguments are used every time, as local variables of a function cannot be modified by anything. Since users most likely use `get_walkers()` and `get_hybrid_lnpost_fn()` frequently together, the `get_walkers()` function allows for the *lnpost_fn* argument to be supplied to it. This will automatically call the `get_hybrid_lnpost_fn()` function factory using the provided *lnpost_fn* and the same input arguments given to `get_walkers()`, and return the obtained *hybrid_lnpost()* function in addition to the starting positions of the MCMC walkers.

5.4.3 Application

Using the information above, using hybrid sampling on a model of choice can be done quite easily. For performing the MCMC analysis, we will be using the emcee package in this example.

Assume that we want to first analyze and then optimize the Gaussian model given by the `GaussianLink` class. So, we first have to make an emulator of the model:

```
>>> from prism import Pipeline
>>> from prism.modellink import GaussianLink
>>> model_data = {3: [3.0, 0.1], 5: [5.0, 0.1], 7: [3.0, 0.1]}
>>> modellink_obj = GaussianLink(model_data=model_data)
>>> pipe = Pipeline(modellink_obj)
>>> pipe.construct()
```

Using the constructed emulator, we can perform a model parameter optimization using hybrid sampling. For this, we need to define an *lnpost()* function, for which we will use a simple Gaussian probability function:

```
def lnpost(par_set, pipe):
    # Create parameter dict for call_model
    par_dict = dict(zip(pipe.modellink.par_name, par_set))

    # Use wrapped model to obtain model output
    mod_out = pipe.modellink.call_model(pipe.emulator.emul_i,
                                        par_dict,
                                        pipe.modellink.data_idx)

    # Get the model and data variances
    # Since the value space is linear, the data error is centered
    md_var = pipe.modellink.get_md_var(pipe.emulator.emul_i,
                                       par_dict,
                                       pipe.modellink.data_idx)
    data_var = [err[0]**2 for err in pipe.modellink.data_err]

    # Calculate the posterior probability and return it
    sigma2 = md_var+data_var
    diff = pipe.modellink.data_val-mod_out
    return(-0.5*(np.sum(diff**2/sigma2)))
```

Since the `Pipeline` object already has the model wrapped and linked, we used that to evaluate the model. The `GaussianLink` class has a centered data error, therefore we can take the upper bound for every error when calculating the variance. However, for more complex models, this is probably not true.

Next, we have to obtain the starting positions for the MCMC walkers. Since we want to do hybrid sampling, we can obtain the *hybrid_lnpost()* function at the same time as well:

```
>>> from prism.utils import get_walkers
>>> n, p0, hybrid_lnpost = get_walkers(pipe, unit_space=False,
                                       lnpost_fn=lnpost, impl_prior=True)
```

By setting *impl_prior* to `True`, we use the implausibility cut-off value as an additional prior. Now we only still need the `EnsembleSampler` class and NumPy (for the *lnpost()* function):

```
>>> import numpy as np
>>> from emcee import EnsembleSampler
```

Now we have everything that is required to perform a hybrid sampling analysis. In most cases, MCMC methods require to be executed on only a single MPI rank, so we will use the *worker_mode*:

```
# Activate worker mode
with pipe.worker_mode:
    if pipe.is_controller:
        # Create EnsembleSampler object
        sampler = EnsembleSampler(n, pipe.modellink.n_par,
                                  hybrid_lnpost, args=[pipe])

        # Run mcmc for 1000 iterations
        sampler.run_mcmc(p0, 1000)

        # Execute any custom operations here
        # For example, saving the chain data or plotting the results
```

And that is basically all that is required for using *PRISM* together with MCMC. For a normal MCMC approach, the same code can be used, except that one has to use *lnpost()* instead of *hybrid_lnpost()* (and, obtain the starting positions of the walkers in a different way).

5.5 General usage rules

Below is a list of general usage rules that apply to *PRISM*.

- Unless specified otherwise in the documentation, any input argument in the *PRISM* package that accepts...
  - a bool (`True`/`False`) also accepts 0/1 as a valid input;
  - `None` indicates a default value or operation for obtaining this input argument. In most of these cases, the default value depends on the current state of the *PRISM* pipeline, and therefore a small operation is required for obtaining this value;

    Example

    Providing `None` to *pot_active_par*, where it indicates that all model parameters should be potentially active.

  - the names of model parameters also accepts the internal indices of these model parameters. The index is the order in which the parameter names appear in the *par_name* list or as they appear in the output of the *details()* method;
  - a parameter/sample set will accept a 1D/2D array-like or a dict of sample(s). As with the previous rule, the columns in an array-like are in the order in which the parameter names appear in the *par_name* list;

  - a sequence of integers, floats and/or strings will accept (almost) any formatting including most special characters as separators as long as they do not have any meaning (like a dot for floats or valid escape sequences for strings). Keep in mind that providing `'1e3'` (or equivalent) will be converted to `1000.0`, as per Python standards;

    Example

    The following sequences are equal:

    ```
    * A, 1, 20.0, B;
    * [A,1,2e1,B];
    * "A 1 20. B";
    * "'["  (A / }| \n; <1{}) ,,">20.000000 !! \t< )?%\B `.
    ```

  - the path to a data file (*PRISM* parameters, model parameters, model data) will read in all the data from that file as a Python dict, with a colon `:` acting as the separator between the key and value.

- Depending on the used emulator type, state of loaded emulator and the *PRISM* parameter values, it is possible that providing values for certain *PRISM* parameters has no influence on the outcome of the pipeline. This can be either because they have non-changeable default values or are simply not used anywhere (given the current state of the pipeline);

  Examples

  - If *method* != `'gaussian'`, it causes *sigma* to have no use in the pipeline;
  - Switching the bool value for *use_mock* while loading a constructed emulator has no effect, since the mock data is generated (or not) when constructing a new emulator and cannot be changed or swapped out afterward.

- All docstrings in *PRISM* are written in RST (reStructuredText) and are therefore best viewed in an editor that supports it (like Spyder);
- All class attributes that hold data specific to an emulator iteration, start with index 1 instead of index 0. So, for example, to access the sample set that was used to construct iteration 1, one would use `pipe.emulator.sam_set[1]` (given that the *Pipeline* object is called `pipe`).

5.6 External data files

When using *PRISM*, there are three different cases where the path to an external data file can be provided. As mentioned in *General usage rules*, all external files are read-in as a Python dict, with the colon being the separator between the key and value. Additionally, all lines are read as strings and converted back when assigned in memory, to allow for many different mark-ups to be used. Depending on which of the three files is read-in, the keys and values have different meanings. Here, the three different files are described.

5.6.1 PRISM parameters file

This file contains the non-default values that must be used for the *PRISM parameters*. These parameters control various different functionalities of *PRISM*. It is provided as the *prism_par* argument when initializing the *Pipeline* class and stored in the *prism_dict* property (a dict or array-like can be provided instead as well). When certain parameters are set depends on their type:

- Emulator parameters: Whenever a new emulator is created;
- Pipeline parameters: When the *Pipeline* class is initialized;
- Implausibility parameters: When the *analyze()* method is called (saved to HDF5) or when an emulator iteration is loaded that has not been analyzed yet (not saved to HDF5);
- Projection parameters: When the *project()* method is called.

The default *PRISM* parameters file can be found in the `prism/data` folder and is shown below:

```
n_sam_init          : 500                   # Number of initial model evaluation␣
↪samples
proj_res            : 25                    # Number of projected grid points per␣
↪model parameter
proj_depth          : 250                   # Number of emulator evaluation samples␣
↪per projected grid point
base_eval_sam       : 800                   # Base number for growth in number of␣
↪model evaluation samples
sigma               : 0.8                   # Gaussian sigma/standard deviation (only␣
↪required if method == 'gaussian')
l_corr              : 0.3                   # Gaussian correlation length(s)
f_infl              : 0.2                   # Residual variance inflation factor
impl_cut            : [0.0, 4.0, 3.8, 3.5]  # List of implausibility cut-off values
criterion           : None                  # Criterion for constructing LHDs
method              : 'full'                # Method used for constructing the␣
↪emulator
use_regr_cov        : False                 # Use regression covariance
poly_order          : 3                     # Polynomial order for regression
n_cross_val         : 5                     # Number of cross-validations for␣
↪regression
do_active_anal      : True                  # Perform active parameter analysis
freeze_active_par   : True                  # Active parameters always stay active
pot_active_par      : None                  # List of potentially active parameters
use_mock            : False                 # Use mock data
```

In this file, the key is the name of the parameter that needs to be changed, and the value what it needs to be changed to. *PRISM* itself does not require this default file, as all of the default values are hard-coded, and is therefore never read-in. An externally provided *PRISM* parameters file is only required to have the non-default values. The contents of this file is equal to providing the following as *prism_par*:

```
# As a dict
prism_par = {'n_sam_init': 500,
             'proj_res': 25,
             'proj_depth': 250,
             'base_eval_sam': 800,
             'sigma': 0.8,
             'l_corr': 0.3,
             'impl_cut': [0.0, 4.0, 3.8, 3.5],
             'criterion': None,
             'method': 'full',
             'use_regr_cov': False,
             'poly_order': 3,
             'n_cross_val': 5,
             'do_active_anal': True,
             'freeze_active_par': True,
             'pot_active_par': None,
             'use_mock': False}
```

(continues on next page)

(continued from previous page)

```
# As an array_like
prism_par = [['n_sam_init', 500],
             ['proj_res', 25],
             ['proj_depth', 250],
             ['base_eval_sam', 800],
             ['sigma', 0.8],
             ['l_corr', 0.3],
             ['impl_cut', [0.0, 4.0, 3.8, 3.5]],
             ['criterion', None],
             ['method', 'full'],
             ['use_regr_cov', False],
             ['poly_order', 3],
             ['n_cross_val', 5],
             ['do_active_anal', True],
             ['freeze_active_par', True],
             ['pot_active_par', None],
             ['use_mock', False]]
```

Note that it is also possible to set any parameter besides *Emulator* parameters by using the corresponding class property.

5.6.2 Model parameters file

This file contains the non-default model parameters to use for a model. It is provided as the *model_parameters* input argument when initializing the *ModelLink* subclass (a dict or array-like can be provided instead as well). Keep in mind that the *ModelLink* subclass may not have default model parameters defined.

An example of the various different ways model parameter information can be provided is given below:

```
# name      : lower_bndry       upper_bndry     estimate
  A         : 1                 5               3
  Bravo     : 2                 7               None
  C42       : 3                 6.74
```

In this file, the key is the name of the model parameter and the value is a sequence of integers or floats, specifying the *lower* and *upper* boundaries of the parameter and, optionally, its estimate. Similarly to the *PRISM* parameters, one can provide the following equivalent as *model_parameters* during initialization of a *ModelLink* subclass:

```
# As a dict
model_parameters = {'A': [1, 5, 3],
                    'Bravo': [2, 7, None],
                    'C42': [3, 6.74]}

# As an array_like
model_parameters = [['A', [1, 5, 3]],
                    ['Bravo', [2, 7, None]],
                    ['C42', [3, 6.74]]]

# As two array_likes zipped
model_parameters = zip(['A', 'Bravo', 'C42'],
                       [[1, 5, 3], [2, 7, None], [3, 6.74]])
```

Providing `None` as the parameter estimate or not providing it at all, implies that no parameter estimate (for the corresponding parameter) should be used in the projection figures. If required, one can use the *convert_parameters()* function to validate their parameters formatting before using it to initialize a *ModelLink* subclass.

5.6.3 Model data file

This file contains the non-default model comparison data points to use for a model. It is provided as the *model_data* input argument when initializing the *ModelLink* subclass (a dict or array-like can be provided instead as well). Keep in mind that the *ModelLink* subclass may not have default model comparison data defined.

An example of the various different ways model comparison data information can be provided is given below:

```
# data_idx   : data_val          data_err        data_spc
  1, 2       : 1                 0.05    0.05    'lin'
  3.0        : 2                 0.05            'log'
  ['A']      : 3                 0.05    0.15
  1, A, 1.0 : 4                  0.05
```

Here, the key is the full sequence of the data identifier of a data point, where any character that is not a letter, number, minus/plus or period acts as a separator between the elements of the data identifier. The corresponding value specifies the data value, data error(s) and data value space. Braces, parentheses, brackets and many other characters can be used as mark-up in the data identifier, to make it easier for the user to find a suitable file lay-out. A full list of all characters that can be used for this can be found in `prism.aux_char_set` and can be freely edited.

Similarly to the model parameters, the following is equal to the contents of this file:

```
# As a dict
model_data = {(1, 2): [1, 0.05, 0.05, 'lin'],
              3.0: [2, 0.05, 'log'],
              ('A'): [3, 0.05, 0.15],
              (1, 'A', 1.0): [4, 0.05]}

# As an array_like
model_data = [[(1, 2), [1, 0.05, 0.05, 'lin']],
              [3.0, [2, 0.05, 'log']],
              [('A'), [3, 0.05, 0.15]],
              [(1, 'A', 1.0), [4, 0.05]]]

# As two array_likes zipped
model_data = zip([(1, 2), 3.0, ('A'), (1, 'A', 1.0)],
                 [[1, 0.05, 0.05, 'lin'], [2, 0.05, 'log'], [3, 0.05, 0.15], [4, 0.
↪05]])
```

It is necessary for the data value to be provided at all times. The data error can be given as either a single value, where it assumed that the data point has a centered 1σ-confidence interval, or as two values, where they describe the *upper* and *lower* bounds of the 1σ-confidence interval. The data value space can be given as a string or omitted, in which case it is assumed that the value space is linear. Keep in mind that, as mentioned in *Data identifiers (data_idx)*, providing a single element data identifier causes it to be saved as a scalar instead of a tuple. Therefore, `['A']` or `('A')` is the same as `'A'`. If required, one can use the *convert_data()* function to validate their data formatting before using it to initialize a *ModelLink* subclass.

Note: The parameter value bounds are given as [*lower bound*, *upper bound*], whereas the data errors are given as [*upper error*, *lower error*]. The reason for this is that, individually, the order for either makes the most sense. Together however, it may cause some confusion, so extra care needs to be taken.

CHAPTER 6

Descriptions

6.1 Terminology

Below is a list of the most commonly used terms/abbreviations in *PRISM* and their meaning.

Active emulator system An emulator system that has a data point assigned to it.

Active parameters The set of model parameters that are considered to have significant influence on the output of the model and contribute at least one polynomial term to one/the regression function.

Adjusted expectation The prior expectation of a parameter set, with the adjustment term taken into account. It is equal to the prior expectation if the emulator system has perfect accuracy.

Adjusted values The adjusted expectation and variance values of a parameter set.

Adjusted variance The prior variance of a parameter set, with the adjustment term taken into account. It is zero if the emulator system has perfect accuracy.

Adjustment term The extra term (as determined by the BLA) that is added to the prior expectation and variance values that describes all additional correlation knowledge between model realization samples.

Analysis

Analyze The process of evaluating a set of emulator evaluation samples in the last emulator iteration and determining which samples should be used to construct the next iteration.

BLA Abbreviation of *Bayes linear approach*.

Construct

Construction The process of calculating all necessary components to describe an iteration of the emulator.

Construction check A list of keywords determining which components of which emulator systems are still required to finish the construction of a specified emulator iteration.

Controller

Controller rank An MPI process that controls the flow of operations in *PRISM* and distributes work to all workers and itself. By default, a controller also behaves like a worker, although is not identified as such.

Covariance matrix

Inverted covariance matrix The (inverted) matrix of prior covariances between all model realization samples and itself.

Covariance vector The vector of prior covariances between all model realization samples and a given parameter set.

Data error The 1σ-confidence interval of a model comparison data point, often a measured/calculated observational error.

Data identifier

Data point identifier The unique identifier of a model comparison data point, often a sequence of integers, floats and strings that describe the operations required to extract it.

Data point A collection of all the details (value, error, space and identifier) about a specific model comparison data point that is used to constrain the model with.

Data space

Data value space The value space (linear, logarithmic or exponential) in which a model comparison data point is defined.

Data value The value of a model comparison data point, often an observed/measured value.

Emulation method The specific method (Gaussian, regression or both) that needs to be used to construct an emulator.

Emulator The collection of all emulator systems together, provided by an `Emulator` object.

Emulator evaluation samples The sample set (to be) used for evaluating the emulator.

Emulator iteration

Iteration A single, specified step in the construction of the emulator.

Emulator system The emulated version of a single model output/comparison data point in a single iteration.

Emulator type The type of emulator that needs to be constructed. This is used to make sure different emulator types are not mixed together by accident.

Evaluate

Evaluation The process of calculating the adjusted values of a parameter set in all emulator systems starting at the first iteration, determining the corresponding implausibility values and performing an implausibility check. This process is repeated in the next iteration if the check was successful and the requested iteration has not been reached.

External model realization set A set of externally calculated and provided model realization samples and their outputs.

Frozen parameters

Frozen active parameters The set of model parameters that, once considered active, will always stay active if possible.

FSLR Abbreviation of *forward stepwise linear regression*.

Gaussian correlation length The maximum distance between two values of a specific model parameter within which the Gaussian contribution to the correlation between the values is still significant.

Gaussian sigma The standard deviation of the Gaussian function. It is not required if regression is used.

HDF5 Abbreviation of *Hierarchical Data Format version 5*.

Hybrid sampling The process of performing a best parameter estimation of a model with MCMC sampling, while using its emulator as an additional Bayesian prior. This process is explained in *Hybrid sampling*.

Implausibility check

Implausibility cut-off check The process of determining whether or not a given set of implausibility values satisfy the implausibility cut-offs of a specific emulator iteration.

Implausibility cut-offs The maximum implausibility values an evaluated parameter set is allowed to generate, to be considered plausible in a specific emulator iteration.

Implausibility value

Univariate implausibility value The minimum σ-confidence level (standard deviations) that the real model realization cannot explain the comparison data. It takes into account all variances associated with the parameter set, which are the observational variance (given by *data_err*), adjusted emulator variance (*adj_var*) and the model discrepancy variance (*md_var*).

Implausibility wildcard A maximum implausibility value, preceding the implausibility cut-offs, that is not taken into account during the implausibility cut-off check. It is denoted as 0 in provided implausibility parameters lists.

LHD Abbreviation of *Latin-Hypercube design*.

Master file

Master HDF5 file (Path to) The HDF5-file in which all important data about the currently loaded emulator is stored. A master file is usually accompanied by several emulator system (HDF5) files, which store emulator system specific data and are externally linked to the master file.

MCMC Abbreviation of *Markov chain Monte Carlo*.

Mock data The set of comparison data points that has been generated by evaluating the model for a random parameter set and perturbing the output by the model discrepancy variance.

Model A *black box* that takes a parameter set, performs a sequence of operations and returns a unique collection of values corresponding to the provided parameter set.

Note: This is how *PRISM* 'sees' a model, not the used definition of one.

2D model A model that has/takes 2 model parameters.

2+D model

nD model A model that has/takes more than 2 model parameters.

ModelLink

ModelLink subclass The user-provided wrapper around the model that needs to be emulated, provided by a `ModelLink` object.

Model data The set of all data points that are provided to a `ModelLink` subclass, to be used to constrain the model with.

Model discrepancy variance A user-defined value that includes all contributions to the overall variance on a model output that is created/caused by the model itself. More information on this can be found in *Model discrepancy variance (md_var)*.

Model evaluation samples The sample set (to be) used for evaluating the model.

Model output

Model outputs The model output(s) corresponding to a single (set of) model realization/evaluation sample(s).

Model parameter

Model parameters The (set of) details about every (all) degree(s)-of-freedom that a model has and whose value range(s) must be explored by the emulator.

Model realization samples Same as model evaluation samples.

Model realizations

Model realization set The combination of model realization/evaluation samples and their corresponding model outputs.

MPI Abbreviation of *Message Passing Interface*.

MPI rank An MPI process that is used by any *PRISM* operation, either being a controller or a worker.

MSE Abbreviation of *mean squared error*.

OLS Abbreviation of *ordinary least-squares*.

Parameter set

Sample A single combination/set of model parameter values, used to evaluate the emulator/model once.

Passive parameters The set of model parameters that are not considered active, and therefore are considered to not have a significant influence on the output of the model.

Pipeline

PRISM Pipeline The main *PRISM* framework that orchestrates all operations, provided by a `Pipeline` object.

Plausible region The region of model parameter space that still contains plausible samples.

Plausible samples A subset of a set of emulator evaluation samples that satisfied the implausibility checks.

Polynomial order Up to which order polynomial terms need to be taken into account for all regression processes.

Potentially active parameters A user-provided set of model parameters that are allowed to become active. Any model parameter that is not potentially active will never become active, even if it should.

PRISM The acronym for *Probabilistic Regression Instrument for Simulating Models*. It is also a one-word description of what *PRISM* does (splitting up a model into individually emulated model outputs).

Prior covariance The covariance value between two parameter sets as determined by an emulator system.

Prior expectation The expectation value of a parameter set as determined by an emulator system, without taking the adjustment term (from the BLA) into account. It is a measure of how much information is captured by an emulator system. It is zero if regression is not used, as no information is captured.

Prior variance The variance value of a parameter set as determined by an emulator system, without taking the adjustment term (from the BLA) into account.

Project

Projection The process of analyzing a specific set of active parameters in an iteration to determine the correlation between the parameters.

Projection figure The visual representation of a projection.

Regression The process of determining the important polynomial terms of the active parameters and their coefficients, by using an FSLR algorithm.

Regression covariances The covariances between all polynomial coefficients of the regression function. By default, they are not calculated and it is empty if regression is not used.

Residual variance The variance that has not been captured during the regression process. It is empty if regression is not used.

Root directory (Path to) The directory/folder on the current machine in which all *PRISM* working directories are located. It also acts as the base for all relative paths.

Sample set

Evaluation set A set of samples.

Worker

Worker rank An MPI process that receives its calls/orders from a controller and performs the heavy-duty operations in *PRISM*.

Working directory (Path to) The directory/folder on the current machine in which the *PRISM* master file, log-file and all projection figures of the currently loaded emulator are stored.

Worker mode A mode initialized by `worker_mode`, where all workers are continuously listening for calls made by the controller rank and execute the received messages. This allows for serial codes to be combined more easily with *PRISM*. See *Dual nature (normal/worker mode)* for more information.

6.2 PRISM parameters

Below are descriptions of all the parameters that can be provided to *PRISM* in a text-file when initializing the `Pipeline` class (using the *prism_par* input argument).

Changed in version 1.1.2: Input argument *prism_file* was renamed to *prism_par*. A dictionary with *PRISM* parameters instead of a file can additionally be provided to the `Pipeline` class. All `Pipeline` parameters can also be changed by setting the corresponding class property.

---

n_sam_init **(Default: 500)** Number of model evaluation samples that is used to construct the first iteration of the emulator. This value must be a positive integer.

proj_res **(Default: 25)** Number of emulator evaluation samples that is used to generate the grid for the projection figures (it defines the resolution of the projection). This value must be a positive integer.

proj_depth **(Default: 250)** Number of emulator evaluation samples that is used to generate the samples in every projected grid point (it defines the accuracy/depth of the projection). This value must be a positive integer.

base_eval_sam **(Default: 800)** Base number of emulator evaluation samples that is used to analyze an iteration of the emulator. It is multiplied by the iteration number and the number of model parameters to generate the true number of emulator evaluations, in order to ensure an increase in emulator accuracy. This value must be a positive integer.

sigma **(Default: 0.8)** The Gaussian sigma/standard deviation that is used when determining the Gaussian contribution to the overall emulator variance. This value is only required when `method == 'gaussian'`, as the Gaussian sigma is obtained from the residual variance left after the regression optimization if regression is included. This value must be non-zero.

l_corr **(Default: 0.3)** The normalized amplitude(s) of the Gaussian correlation length. This number is multiplied by the difference between the upper and lower value boundaries of the model parameters to obtain the Gaussian correlation length for every model parameter. This value must be positive, normalized and either a scalar or a list of `n_par` scalars (where the values correspond to the sorted list of model parameters).

f_infl **(Default: 0.2)** New in version 1.2.2.

The residual variance inflation factor. The variance values for all known samples in an emulator iteration are inflated by this number multiplied by `rsdl_var`. This can be used to adjust for the underestimation of the emulator variance. Setting this to zero causes no variance inflation to be performed. This value must be non-negative.

impl_cut **(Default: [0.0, 4.0, 3.8, 3.5])** A list of implausibility cut-off values that specifies the maximum implausibility values a parameter set is allowed to have to be considered 'plausible'. A zero can be used as a filler value, either taking on the preceding value or acting as a wildcard if the preceding value is a wildcard or non-existent. Zeros are appended at the end of the list if the length is less than the number of comparison data points, while extra values are ignored if the length is more. This must be a sorted list of positive values (excluding zeros).

criterion **(Default: None)** The criterion to use for determining the quality of the LHDs that are used, represented by an integer, float, string or `None`. This parameter is the only non-*PRISM* parameter. Instead, it is used in the `lhd()`-function of the e13Tools package. By default, `None` is used.

method **(Default: 'full')** The method to use for constructing the emulator. `'gaussian'` will only include Gaussian processes (no regression), which is much faster, but also less accurate. `'regression'` will only include regression processes (no Gaussian), which is more accurate than Gaussian only, but underestimates the emulator variance by multiple orders of magnitude. `'full'` includes both Gaussian and regression processes, which is slower than Gaussian only, but by far the most accurate both in terms of expectation and variance values.

`'gaussian'` can be used for faster exploration especially for simple models. `'regression'` should only be used when the polynomial representation of a model is important and enough model realizations are available. `'full'` should be used by default.

> **Warning:** When using *PRISM* on a model that can be described completely by the regression function (anything that has an analytical, polynomial form up to order *poly_order* like a straight line or a quadratic function), use the `'gaussian'` method unless unavoidable (in which case *n_sam_init* and *base_eval_sam* must be set to very low values).
>
> When using the regression method on such a model, *PRISM* will be able to capture the behavior of the model perfectly given enough samples, in which case the residual (unexplained) variance will be approximately zero and therefore *sigma* as well. This can occassionally cause floating point errors when calculating emulator variances, which in turn can give unexplainable artifacts in the adjustment terms, therefore causing answers to be incorrect.
>
> Since *PRISM*'s purpose is to identify the characteristics of a model and therefore it does not know anything about its workings, it is not possible to automatically detect such problems.

use_regr_cov **(Default: False)** Whether or not the regression variance should be taken into account for the variance calculations. The regression variance is the variance on the regression process itself and is only significant if a low number of model realizations (*n_sam_init* and *base_eval_sam*) is used to construct the emulator systems. Including it usually only has a very small effect on the overall variance value, while it can slow down the emulator evaluation rate by as much as a factor of 3. This value is not required if *method* == `'gaussian'` and is automatically set to `True` if *method* == `'regression'`. This value must be a bool.

poly_order **(Default: 3)** Up to which order all polynomial terms of all model parameters should be included in the active parameters and regression processes. This value is not required if *method* == `'gaussian'` and *do_active_anal* is `False`. This value must be a positive integer.

n_cross_val **(Default: 5)** Number of (k-fold) cross-validations that must be used for determining the quality of the active parameters analysis and regression process fits. If this parameter is zero, cross-validations are not used. This value is not required if *method* == `'gaussian'` and *do_active_anal* is `False`. This value must be a non-negative integer and not equal to 1.

do_active_anal **(Default: True)** Whether or not an active parameters analysis must be carried out for every iteration of every emulator system. If `False`, all potentially active parameters listed in *pot_active_par* will be active. This value must be a bool.

freeze_active_par **(Default: True)** Whether or not active parameters should be frozen in their active state. If `True`, parameters that have been considered active in a previous iteration of an emulator system, will automatically be active again (and skip any active parameters analysis). This value must be a bool.

pot_active_par **(Default: None)** A list of parameter names that indicate which parameters are potentially active. Potentially active parameters are the only parameters that will enter the active parameters analysis (or will all be active if *do_active_anal* is `False`). Therefore, all parameters not listed will never be considered active. If all parameters should be potentially active, then a `None` can be given. This must either be a list of parameter names or `None`.

use_mock **(Default: False)** Whether or not mock data must be used as comparison data when constructing a new emulator. Mock data is calculated by evaluating the model for a specific set of parameter values, and adding the model discrepancy variances as noise to the returned data values. This set of parameter values is either the provided set, or a randomly chosen one if not. When using mock data for an emulator, it is not possible to change the comparison data in later emulator iterations. This value must be a bool or a list of *n_par* scalars (where the values correspond to the sorted list of model parameters).

6.3 HDF5

Whenever *PRISM* constructs an emulator, it automatically stores all the calculated data for it in an HDF5-file named `'prism.hdf5'` in the designated working directory. This file contains all the data that is required in order to recreate all emulator systems that have been constructed for the emulator belonging to this run. If the *Pipeline* class is initialized by using an HDF5-file made by *PRISM*, it will load in this data and return a *Pipeline* object in the same state as described in the file.

Below is a short overview of all the data that can be found inside a *PRISM* master HDF5-file. HDF5-files can be viewed freely by the user using the HDFView application made available by The HDFGroup.

The general file contains:

- Attributes (11/12): Describe the general non-changeable properties of the emulator, which include:
  - Emulator type and method;
  - Gaussian parameters;
  - Name of used *ModelLink* subclass;
  - Used *PRISM* version;
  - Regression parameters;
  - Bools for using mock data or regression covariance;
  - Mock data parameters if mock data was used.
- Every emulator iteration has its own data group with the iteration number as its name. This data group stores all data/information specific to that iteration.

An iteration data group (`'i'`) contains:

- Attributes (9): Describe the general properties and results of this iteration, including:
  - Active parameters for this emulator iteration;
  - Implausibility cut-off parameters;
  - Number of emulated data points, emulator systems, emulator evaluation samples, plausible samples and model realization samples;
  - Bool stating whether this emulator iteration used an external model realization set.

- `'emul_n'`: The data group that contains all data for a specific emulator system in this iteration. The value of `'n'` indicates which emulator system it is, not the data point. See below for its contents;
- `'emul_space'`: The boundaries of the hypercube that encloses the parameter space in which this iteration is defined. This is always equal to the plausible space of the previous iteration;
- `'impl_sam'`: The set of emulator evaluation samples that survived the implausibility checks and will be used to construct the next iteration;
- `'proj_hcube'`: The data group that contains all data for the (created) projections for this iteration, if at least one has been made. See below for its contents;
- `'sam_set'`: The set of model realization samples that were used to construct this iteration. In every iteration after the first, this is the `'impl_sam'` of the previous iteration;
- `'statistics'`: An empty data set that stores several different types of statistics as its attributes, including:
  - Size of the MPI communicator during various construction steps;
  - Average evaluation rate/time of the emulator and model;
  - Total time cost of most construction steps (note that this value may be incorrect if a construction was interrupted);
  - Percentage of parameter space that is still plausible within the iteration.

An emulator system data group (`'i/emul_n'`) contains:

- Attributes (7+): List the details about the model comparison data point used in this emulator system, including:
  - Active parameters for this emulator system;
  - Data errors, identifiers, value space and value;
  - Regression score and residual variance if regression was used;
  - The active and passive contributions to the residual variance (obtained from either the regression residual variance or the Gaussian sigma).
- `'cov_mat'`: The pre-calculated covariance matrix of all model evaluation samples in this emulator system. This data set is never used in *PRISM* and stored solely for user-convenience;
- `'cov_mat_inv'`: The pre-calculated inverse of `'cov_mat'`;
- `'exp_dot_term'`: The pre-calculated second expectation adjustment dot-term $(\mathrm{Var}\,(D)^{-1} \cdot (D - \mathrm{E}(D)))$ of all model evaluation samples in this emulator system.
- `'mod_set'`: The model outputs for the data point in this emulator system corresponding to the `'sam_set'` used in this iteration;
- `'poly_coef'` (if regression is used): The non-zero coefficients for the polynomial terms in the regression function in this emulator system;
- `'poly_coef_cov'` (if regression and regr_cov are used): The covariances for all polynomial coefficients `'poly_coef'`;
- `'poly_idx'` (if regression is used): The indices of the polynomial terms with non-zero coefficients if all active parameters are converted to polynomial terms;
- `'poly_powers'` (if regression is used): The powers of the polynomial terms corresponding to `'poly_idx'`. Both `'poly_idx'` and `'poly_powers'` are required since different methods of calculating the polynomial terms are used depending on the number of required terms and samples;

- `'prior_exp_sam_set'`: The pre-calculated prior expectation values of all model evaluation samples in this emulator system. This data set is also never used in *PRISM*.

A projections data group (`'i/proj_hcube'`) contains individual projection data groups (`'i/proj_hcube/<name>'`), whic

- Attributes (4): List the general properties with which this projection was made, including:
  - Implausibility cut-off parameters (they can differ from the iteration itself);
  - Projection depth and resolution.
- `'impl_los'`: The calculated line-of-sight depth for all grid points in this projection;
- `'impl_min'`: The calculated minimum implausibility values for all grid points in this projection;
- `'proj_space'`: The boundaries of the hypercube that encloses the defined parameter space of this projection.

CHAPTER 7

FAQ

7.1 How do I contribute?

Contributing to *PRISM* is done through pull requests in the repository. If you have an idea on what to contribute, it is recommended to open a GitHub issue about it, such that the maintainers can give their advice or help. If you want to contribute but do not really know what, then you can take a look at the ToDos that are scattered throughout the code. When making a contribution, please keep in mind that it must be compatible with all Python versions that *PRISM* supports (3.5+), and preferably with all operating systems as well.

7.2 How do I report a bug/problem?

By opening a GitHub issue and using the *Bug report* template.

7.3 What does the term ... mean?

A list of the most commonly used terms in *PRISM* can be found on the *Terminology* page.

7.4 Where can I get PRISM's colormaps?

The *rainforest* and *freeze* colormaps that are used for drawing *PRISM*'s projection figures, are freely available in the e13Tools package. Importing e13Tools will automatically add both colormaps (and their reverses) to the list of available colormaps in Matplotlib. One can then access them directly in the `cm` module or by using the `get_cmap()` function. More information on the colormaps in e13Tools can be found in its documentation.

7.5 Which OSs are supported?

PRISM should be compatible with all Windows, Mac OS and UNIX-based machines, as long as they support one of the required Python versions. Compatibility is currently tested for Linux 16.04 (Xenial)/Mac OS-X using Travis CI, Windows 32-bit/64-bit using AppVeyor and all OSs using Azure Pipelines.

CHAPTER 8

Community guidelines

PRISM is an open-source and free-to-use software package (and it always will be), provided under the BSD-3 license (see below for the full license).

Users are highly encouraged to make contributions to the package or request new features by opening a GitHub issue. If you would like to contribute to the package, but do not know what, then there are quite a few ToDos in the code that may give you some inspiration. As with contributions, if you find a problem or issue with *PRISM*, please do not hesitate to open a GitHub issue about it or post it on Gitter.

And, finally, if you use *PRISM* as part of your workflow in a scientific publication, please consider including an acknowledgement like *"Parts of the results in this work were derived using the PRISM Python package."* and citing the *PRISM* pipeline paper using the BibTeX-entry below.

8.1 License

```
BSD 3-Clause License

Copyright (c) 2019-2020, Ellert van der Velden
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of the copyright holder nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.
```

(continues on next page)

(continued from previous page)

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

8.2 Citation

This BibTeX-entry is also available in *PRISM* using *get_bibtex()* (available as `prism.get_bibtex()`).

```
@ARTICLE{2019ApJS..242...22V,
       author = {{van der Velden}, Ellert and {Duffy}, Alan R. and {Croton}, Darren␣
↪and
         {Mutch}, Simon J. and {Sinha}, Manodeep},
        title = "{Model Dispersion with PRISM: An Alternative to MCMC for Rapid␣
↪Analysis of Models}",
      journal = {\apjs},
     keywords = {methods: data analysis, methods: numerical,
         Astrophysics - Instrumentation and Methods for Astrophysics, Physics -␣
↪Computational Physics},
         year = 2019,
        month = jun,
       volume = {242},
       number = {2},
          eid = {22},
        pages = {22},
          doi = {10.3847/1538-4365/ab1f7d},
archivePrefix = {arXiv},
       eprint = {1901.08725},
 primaryClass = {astro-ph.IM},
       adsurl = {https://ui.adsabs.harvard.edu/abs/2019ApJS..242...22V},
      adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

8.3 Additions

Below are some bigger ideas/improvements that may be added to *PRISM* if there is demand:

- Add a developer's guide to the docs, describing the inner workings and structures of *PRISM*;
- Low-level MPI implementation (probably by using D2O);

  With 6 emulator systems and 4 processes, the three different MPI levels would be:

  - No level: 6-0-0-0;
  - High-level: 2-2-1-1;

  - Low-level: 1.5-1.5-1.5-1.5.

- Dynamic implausibility cut-offs;
- Allow for a master projection figure to be made (kind of like a double corner plot);
- Allow for user-provided methods in the *ModelLink* subclass to be executed at specific points in the emulator construction;
- Implement multi-variate implausibilities;
- Allow for no *ModelLink* object to be provided, which blocks construction but enables everything emulator-only related;
- Allow for old *PRISM* master files to be provided when making a new emulator, recycling work done previously;
- If *MPI_call* is `False` for the *ModelLink* subclass, use all MPI ranks to evaluate a part of `sam_set` in the model simultaneously. This will require a check or flag that the model can be called in multiple instances simultaneously (to accommodate for models that, for example, need to read files during evaluations). Added benefit of this is that it would become possible to add the option for the user to set a preferred number of MPI processes calling the model (in MPI), allowing *PRISM* to split up the available processes if more efficient;
- GPU acceleration;
- Adding the theory behind *PRISM* to the docs;
- Adding the possibility to evaluate the derivatives of the emulated model outputs, which could be used as approximations of the gradient field of a model for certain MCMC methods;
- Replace the *list of lists* data system with a *list of dicts* system. This would remove the need for converting global indices to/from local indices in several cases, and make it easier for users to understand. However, as indexing dicts is more complicated, this may require a lot of rewriting;
- Code objects can be made pickleable by importing the `codeutil` module. This package could be added to the requirements or an equivalent function could be written, which is then automatically imported/executed upon importing *PRISM*;

CHAPTER 9

Pipeline

class prism.**Pipeline**(*modellink_obj*, ***, *root_dir=None*, *working_dir=None*, *prefix=None*, *prism_par=None*, *emul_type=None*, *comm=None*, ***kwargs*)

Defines the `Pipeline` class of the *PRISM* package.

The `Pipeline` class is the main user class of the *PRISM* package and provides a user-friendly environment that gives access to all operations within the pipeline.

_Projection__analyze_proj_hcube(*hcube*)

Analyzes an emulator projection hypercube *hcube*.

Parameters **hcube** (*1D array_like of int of length {2, 3}*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube.

Returns

- **impl_min_hcube** (1D `ndarray` object) – List containing the lowest implausibility value that can be reached in every single grid point on the given hypercube.
- **impl_los_hcube** (1D `ndarray` object) – List containing the fraction of the total amount of evaluated samples in every single grid point on the given hypercube, that still satisfied the implausibility cut-off criterion.

_Projection__draw_2D_proj_fig(*hcube*)

Draws the 2D projection figure for the provided *hcube*.

Parameters **hcube** (*1D array_like of int of length 2*) – Array containing the internal integer identifiers of the main model parameters that require a projection figure.

_Projection__draw_3D_proj_fig(*hcube*)

Draws the 3D projection figure for the provided *hcube*.

Parameters **hcube** (*1D array_like of int of length 3*) – Array containing the internal integer identifiers of the main model parameters that require a projection figure.

_Projection__draw_estimate_arrowline(*par_est1*, *par_est2*, *ax*)

Draws parameter estimate arrow and lines in the given *ax*, for the provided estimates *par_est1* and *par_est2*. Whether an arrow and/or a line is drawn, and what their layout and position is, depends on the given estimate values.

Parameters

- **par_est1, par_est2** (*float or None*) – The parameter estimates for which an arrow or line must be drawn. If the estimate is within the axes limits, a line is drawn. If the estimate is outside of the axes limits, an arrow is drawn in the direction of *(par_est1, par_est2)*. If *None*, a default value is used for the respective argument.
- **ax** (`Axes` object) – The Axes object in which the arrow and lines must be drawn.

Note: Because the arrow is drawn using relative DPI-coordinates, the axes limits and positions of *ax* must be final before calling this function.

_Projection__get_default_input_arguments()
Generates a dict containing default values for all input arguments.

Returns **kwargs_dict** (*dict*) – Dict containing all default input argument values.

_Projection__get_default_parameters()
Generates a dict containing default values for all projection parameters.

Returns **par_dict** (*dict*) – Dict containing all default projection parameter values.

_Projection__get_fig_path(*hcube*)
Determines the absolute path of a projection figure corresponding to a provided projection hypercube *hcube* and returns it.

Parameters **hcube** (*1D array_like of int of length {2, 3} or str*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube. If str, the name of *hcube* instead (*`_Projection__get_hcube_name()`*).

Returns

- **fig_path** (*str*) – The absolute path to the requested projection figure.
- **fig_path_s** (*str*) – The absolute path to the smoothed version.

_Projection__get_grid_idx(*hcube*)
Returns the indices of all parameters that are in the grid points of the given projection hypercube *hcube*.

Parameters **hcube** (*1D array_like of int of length {2, 3}*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube.

Returns **grid_idx** (*list of int*) – Indices of all grid point parameters.

_Projection__get_hcube_name(*hcube*)
Determines the name of a provided projection hypercube *hcube* and returns it.

Parameters **hcube** (*1D array_like of int of length {2, 3}*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube.

Returns **hcube_name** (*str*) – The name of this projection hypercube.

_Projection__get_proj_data(*hcube*)
Returns the projection data belonging to the provided hypercube *hcube*.

Parameters **hcube** (*1D array_like of int of length {2, 3}*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube.

Returns **proj_data** (*dict*) – Dict containing all the data associated with the specified *hcube*.

_Projection__get_proj_hcube(*hcube*)
Generates a projection hypercube *hcube* containing emulator evaluation samples The output of this function depends on the requested projection type.

Parameters **hcube** (*1D array_like of int of length {2, 3}*) – Array containing the internal integer identifiers of the main model parameters that require a projection hypercube.

Returns **proj_hcube** (3D `ndarray` object) – 3D projection hypercube of emulator evaluation samples. For 3D projections, the grid uses matrix indexing (second parameter varies the fastest).

_Projection__get_proj_space(*emul_i*)
Returns the boundaries of the hypercube that encloses the parameter space in which the projection space of the provided emulator iteration *emul_i* is defined.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **proj_space** (2D `ndarray` object) – The requested hypercube boundaries.

_Projection__get_req_hcubes(*emul_i*, *proj_par*)
Determines which projection hypercubes have been requested by the user. Also checks if these projection hypercubes have been calculated before, and depending on the value of `force`, either skips them or recreates them.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **proj_par** (*1D array_like of {int; str} or None*) – For which model parameters to construct the projection figures. If 1D array_like, construct projection figures for all combinations of provided model parameters that are active, with a string referring to the name of the model parameter and an integer referring to the position in which the model parameter is shown in the *`details()`* method. If *None*, projection figures are made for all active model parameters.

Generates

hcubes [list of lists] List containing the parameter indices of the requested projection hypercubes.

create_hcubes [list of lists] List containing the parameter indices of the requested projection hypercubes that need to be created first.

_Projection__prepare_projections(*emul_i*, *proj_par*, ***kwargs*)
Prepares the pipeline for the creation of the requested projections.

Parameters

- **emul_i** (*int or None*) – Number indicating the requested emulator iteration.
- **proj_par** (*1D array_like of {int; str} or None*) – For which model parameters to construct the projection figures. If 1D array_like, construct projection figures for all combinations of provided model parameters that are active, with a string referring to the name of the model parameter and an integer referring to the position in which the model parameter is shown in the *`details()`* method. If *None*, projection figures are made for all active model parameters.
- **kwargs** (*keyword arguments*) – Keyword arguments that were provided to *`project()`*.

_Projection__process_input_arguments(*emul_i*, ***kwargs*)
Processes the input arguments given to the *`project()`* method.

Parameters

- **emul_i** (*int or None*) – Number indicating the requested emulator iteration.
- **kwargs** (*keyword arguments*) – Keyword arguments that were provided to *`project()`*.

Generates

All default and provided arguments are saved to their respective properties.

_Projection__read_proj_space(*hcube_group*)
Reads in and transforms the projection parameter space hypercube that is stored in the provided *hcube_group*.

Parameters **hcube_group** (`Group` object) – The HDF5-group from which the projection parameter space hypercube needs to be read in.

Returns **proj_space** (2D `ndarray` object) – The read-in hypercube boundaries.

_Projection__save_data(*emul_i*, *data_dict*)
Saves a given data dict `{keyword: data}` at the emulator iteration this class was initialized for, to the HDF5-file.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **data_dict** (*dict*) – Dict containing the data that needs to be saved to the HDF5-file.

Keyword Arguments

- **keyword** (*{'nD_proj_hcube'}*) – String specifying the type of data that needs to be saved.
- **data** (*{int; float; str; array_like} or dict*) – The actual data that needs to be saved at data keyword *keyword*. If dict, save every item individually.

Generates

The specified data is saved to the HDF5-file.

_Projection__set_parameters()
Sets the Projection parameters from the *prism_dict* property and saves them in the current `Projection` instance.

__call__(*emul_i=None*, ***, *force=False*)
Calls the *construct()* method to start the construction of the given iteration of the emulator and creates the projection figures right afterward if this construction was successful.

Other Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the next iteration of the emulator will be constructed.
- **force** (*bool. Default: False*) – Controls what to do if the specified emulator iteration *emul_i* already (partly) exists. If *False*, finish construction of the specified iteration or skip it if already finished. If *True*, reconstruct the specified iteration entirely.

__init__(*modellink_obj*, ***, *root_dir=None*, *working_dir=None*, *prefix=None*, *prism_par=None*, *emul_type=None*, *comm=None*, ***kwargs*)
Initialize an instance of the *Pipeline* class.

Parameters **modellink_obj** (*ModelLink* object) – Instance of the *ModelLink* subclass that links the emulated model to this *Pipeline* instance.

Other Parameters

- **root_dir** (*str or None. Default: None*) – String containing the absolute path of the root directory where all working directories are stored. If *None*, root directory will be set to the directory this class was initialized in.

- **working_dir** (*str, bool or None. Default: None*) – String containing the name of the working directory of the emulator in *root_dir*. If *True*, a new working directory will be created in *root_dir*. If *None* or *False*, working directory is set to the last one that was created in *root_dir* that starts with the given *prefix*. Note that providing an integer instead of a bool will not work here. If no directories are found, one will be created.
- **prefix** (*str or None. Default: None*) – String containing a prefix that is used for naming new working directories or scan for existing ones. If *None*, all directories in *root_dir* are considered working directories and 'prism_' will be used as a prefix for new ones.
- **prism_par** (*array_like, dict, str or None. Default: None*) – A dict containing the values for the *PRISM* parameters that need to be changed from their default values. If array_like, dict(*prism_par*) must generate a dict with the correct lay-out. If str, the string is the absolute or relative path to the file that contains the keys in the first column and the dict values in the second column. If a relative path is given, its path must be relative to *root_dir* or the current directory. If *None*, no changes will be made to the default parameters.
- **emul_type** (`Emulator` subclass or None. Default: None) – The type of `Emulator` to use in this `Pipeline` instance. If *None*, use the default emulator instead.
- **comm** (`Intracomm` object or None. Default: None) – The MPI intra-communicator to use in this `Pipeline` instance or `MPI.COMM_WORLD` if *comm* is *None*.

_call_model(*emul_i*, *par_set*, *data_idx*)

Obtain the output corresponding to the provided *data_idx* that is generated by the model for a given model parameter sample *par_set*. The current emulator iteration *emul_i* is also provided in case it is required by the `ModelLink` subclass.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **par_set** (*1D array_like*) – Model parameter sample to calculate the model output for.
- **data_idx** (*list of tuples*) – The list of data identifiers for which the model is requested to return the corresponding data values.

Returns **mod_out** (1D `ndarray` object) – Model output corresponding to given *par_set*.

_compile_code_snippets()

Compiles all pre-defined built-in code snippets to code objects and saves them to `code_objects`. These code objects are used for performing standard operations in the `_evaluate_sam_set()` method.

_do_impl_check(*emul_i*, *uni_impl_vals*)

Performs an implausibility cut-off check on the provided implausibility values *uni_impl_vals* at emulator iteration *emul_i*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **uni_impl_vals** (*2D array_like*) – Array containing all univariate implausibility values corresponding to all parameter sets for all data points.

Returns

- **impl_check_vals** (1D `ndarray` object) – Array containing whether the check was successful for that sample.
- **impl_cut_vals** (1D `ndarray` object) – Array containing the implausibility value at the first real implausibility cut-off for every sample.

_evaluate_model(*emul_i*, *sam_set*, *data_idx*)
Evaluates the model for provided evaluation sample set *sam_set* at given data points *data_idx*.

This method automatically distributes the samples according to the various flags set in the *ModelLink* subclass.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **sam_set** (*1D or 2D array_like*) – Parameter/sample set to evaluate in the model.
- **data_idx** (*list of tuples*) – The list of data identifiers for which the model is requested to return the corresponding data values.

Returns

- **sam_set** (2D `ndarray` object of shape `(n_sam, n_par)`) – Array containing the sample set used to evaluate the model.
- **mod_set** (2D `ndarray` object of shape `(n_sam, n_data)`) – Array containing the data values corresponding to the requested data points.

_evaluate_sam_set(*emul_i*, *sam_set*, *exec_code*)
Evaluates a provided set of emulator evaluation samples *sam_set* at a given emulator iteration *emul_i*. The provided tuple of code snippets *exec_code* are executed using Python's `exec()` function at specific points during the analysis.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **sam_set** (2D `ndarray` object) – Array containing model parameter value sets to be evaluated in all emulator systems in emulator iteration *emul_i*.
- **exec_code** (*{'analyze', 'evaluate', 'hybrid', 'project'} or tuple*) – Tuple of five code snippets `(pre_code, eval_code, anal_code, post_code, exit_code)` to be executed at specific points during the analysis. If string, use one of the built-in tuples in *code_objects* instead.

Other Parameters

- **pre_code** (*str or code object*) – Code snippet to be executed before the evaluation of *sam_set* starts.
- **eval_code** (*str or code object*) – Code snippet to be executed after the evaluation of each sample in *sam_set*.
- **anal_code** (*str or code object*) – Code snippet to be executed after the analysis of each sample in *sam_set*. This code snippet is only executed by the controller.
- **post_code** (*str or code object*) – Code snippet to be executed after the evaluation of *sam_set* ends.
- **exit_code** (*str or code object*) – Code snippet to be executed before returning the results of the evaluation of *sam_set*. This code snippet is only executed by the controller.

Returns **results** (*object*) – The object that is assigned to a local variable called *results*, which is defaulted to *None* if no code snippet sets it. Preferably, the execution of *post_code* and/or *exit_code* sets this variable. All MPI ranks return it.

Notes

If any of the code snippets is provided as a string, it will be compiled into a code object before starting the evaluation.

_get_default_parameters()
Generates a dict containing default values for all pipeline parameters.

Returns **par_dict** (*dict*) – Dict containing all default pipeline parameter values.

_get_eval_sam_set(*emul_i*)
Generates an emulator evaluation sample set to be used for analyzing an emulator iteration. Currently uses the `lhd()` function.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **eval_sam_set** (2D `ndarray` object) – Array containing the evaluation samples.

_get_ext_real_set(*emul_i*, *ext_real_set*)
Processes an externally provided model realization set *ext_real_set*, containing the used sample set and the corresponding data value set, to be used for the provided *emul_i*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **ext_real_set** (*str, list, dict or None*) – String indicating the suffix of the backup file that needs to be used; list of dicts containing an externally calculated set of model evaluation samples and its data values; a dict with keys `['sam_set', 'mod_set']` containing these dicts; or *None* if no external set needs to be used.

Returns

- **ext_sam_set** (1D or 2D `ndarray` object) – Array containing the externally provided model evaluation samples.
- **ext_mod_set** (1D or 2D `ndarray` object) – Array containing the model outputs of all specified externally provided model evaluation samples.

_get_f_impl(*emul_i*)
Returns the fraction of parameter space that passed the implausibility checks during the analysis of the provided emulator iteration *emul_i*.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **f_impl** (*float*) – The fraction of parameter space that is still plausible.

_get_impl_space(*emul_i*)
Returns the boundaries of the hypercube that encloses the parameter space in which the plausible space of the provided emulator iteration *emul_i* is defined.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **impl_space** (2D `ndarray` object) – The requested hypercube boundaries.

Note: The parameter space over which plausible space is defined is always equal to the emulator space of the next iteration. This means that reanalyzing an iteration can change the result of this function.

_get_iteration_data(*emul_i*, *sam_set*, *ext_sam_set*, *ext_mod_set*)
Obtains the model realization data for given emulator iteration *emul_i* by evaluating the provided *sam_set* in the model and distributing model outputs to the correct emulator systems.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **sam_set** (2D `ndarray` object) – Array containing the model evaluation samples.
- **ext_sam_set** (1D or 2D `ndarray` object) – Array containing the externally provided model evaluation samples.
- **ext_mod_set** (1D or 2D `ndarray` object) – Array containing the model outputs of all specified externally provided model evaluation samples.

Generates

sam_set [2D `ndarray` object] Array containing the model evaluation samples for emulator iteration *emul_i*.

mod_set [2D `ndarray` object] Array containing the model outputs of all specified model evaluation samples for emulator iteration *emul_i*.

_get_md_var(*emul_i*, *par_set*)

Retrieves the model discrepancy variances, which includes all variances that are created by the model provided by the *ModelLink* instance. This method tries to call the *get_md_var()* method, and assumes a default model discrepancy variance of `1/6th` the data value if it cannot be called. If the data value space is not linear, then this default value is calculated such to reflect that.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **par_set** (1D `ndarray` object) – Model parameter value set to calculate the model discrepancy variances for.

Returns **var_md** (2D `ndarray` object) – Variance of the model discrepancy.

_get_mock_data(*mock_par*)

Generates mock data and loads it into the *ModelLink* object that was provided during class initialization. This function overwrites the *ModelLink* properties holding the parameter estimates, data values and data errors.

Parameters **mock_par** (*1D array_like or None*) – If 1D array_like, use the provided parameter estimates to create the mock data. If *None*, a random parameter set will be generated as parameter estimates.

Generates

Overwrites the corresponding *ModelLink* class properties with the generated values.

_get_n_eval_sam(*emul_i*)

This function calculates the total number of emulator evaluation samples at a given emulator iteration *emul_i* from *base_eval_sam*.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **n_eval_sam** (*int*) – Total number of emulator evaluation samples.

_get_paths(*root_dir*, *working_dir*, *prefix*)

Obtains the path for the root directory, working directory and parameters file for *PRISM*.

Parameters

- **root_dir** (*str or None*) – String containing the absolute path of the root directory where all working directories are stored. If *None*, root directory will be set to the directory this class was initialized in.
- **working_dir** (*str, bool or None*) – String containing the name of the working directory of the emulator in *root_dir*. If *True*, a new working directory will be created in *root_dir*. If *None* or *False*, working directory is set to the last one that was created in *root_dir* that starts with the given *prefix*. Note that providing an integer instead of a bool will not work here. If no directories are found, one will be created.
- **prefix** (*str or None*) – String containing a prefix that is used for naming new working directories or scan for existing ones. If *None*, all directories in *root_dir* are considered working directories and 'prism_' will be used as a prefix for new ones.

Generates

The absolute paths to the root directory, working directory, emulator master HDF5-file and *PRISM* parameters file.

_get_uni_impl(*emul_i*, *par_set*, *adj_exp_val*, *adj_var_val*)

Calculates the univariate implausibility values at a given emulator iteration *emul_i* for specified expectation and variance values *adj_exp_val* and *adj_var_val*, corresponding to given *par_set*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **par_set** (1D `ndarray` object) – Model parameter value set to calculate the univariate implausibility values for. Only used to pass to the `get_md_var()` method.
- **adj_exp_val, adj_var_val** (*1D array_like*) – The adjusted expectation and variance values to calculate the univariate implausibility for.

Returns **uni_impl_val** (1D `ndarray` object) – Univariate implausibility value for all requested emulator systems.

_load_data()

Loads in all the important pipeline data into memory for the controller rank. If it is detected that the last emulator iteration has not been analyzed yet, the implausibility analysis parameters are taken from the *PRISM* parameters dict and temporarily stored in memory.

Generates

All relevant pipeline data up to the last emulator iteration is loaded into memory.

_make_call(*exec_fn*, **args*, ***kwargs*)

Sends the provided *exec_fn* to all worker ranks, if they are listening for calls, and tells them to execute it using the provided *args* and *kwargs*. All ranks that call this function will execute *exec_fn* as well.

If used within the `WorkerMode` context manager, this function should only be called by the controller. If not, it should be called by all valid ranks that must execute *exec_fn*.

Parameters

- **exec_fn** (*str or callable*) – If string, a callable attribute of this `Pipeline` instance or a callable object that the workers should execute if not.
- **args** (*positional arguments*) – Positional arguments that need to be provided to *exec_fn*.
- **kwargs** (*keyword arguments*) – Keyword arguments that need to be provided to *exec_fn*.

Returns out (*object*) – The object returned by executing *exec_fn*. Note that only ranks that directly call this function return, as workers in worker mode cannot do so.

Note: Changed in version 1.2.0: If any entry in *args* or *kwargs* is a string written as 'pipe.XXX', it is assumed that 'XXX' refers to a `Pipeline` attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

_make_call_workers(*exec_fn*, **args*, ***kwargs*)

Sends the provided *exec_fn* to all worker ranks, if they are listening for calls, and tells them to execute it using the provided *args* and *kwargs*.

If used within the `WorkerMode` context manager, this function should only be called by the controller. If not, it should be called by all valid ranks that must execute *exec_fn*.

Parameters

- **exec_fn** (*str or callable*) – If string, a callable attribute of this `Pipeline` instance or a callable object that the workers should execute if not.
- **args** (*positional arguments*) – Positional arguments that need to be provided to *exec_fn*.
- **kwargs** (*keyword arguments*) – Keyword arguments that need to be provided to *exec_fn*.

Returns out (*object*) – The object returned by executing *exec_fn*. Note that only ranks that directly call this function return, as workers in worker mode cannot do so.

Note: Changed in version 1.2.0: If any entry in *args* or *kwargs* is a string written as 'pipe.XXX', it is assumed that 'XXX' refers to a `Pipeline` attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

_multi_call_model(*emul_i*, *sam_set*, *data_idx*)

Obtain the output set corresponding to the provided *data_idx* that is generated by the model for a given model parameter sample set *sam_set*. The current emulator iteration *emul_i* is also provided in case it is required by the `ModelLink` subclass.

This is a multi-version of `_call_model()`.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **sam_set** (*2D array_like*) – Model parameter sample set to calculate the model output for.
- **data_idx** (*list of tuples*) – The list of data identifiers for which the model is requested to return the corresponding data values.

Returns mod_set (2D `ndarray` object) – Model output set corresponding to given *sam_set*.

_read_parameters(*prism_par*)

Reads in all parameters in the provided *prism_par* and saves them as a dict in the current `Pipeline` instance.

_save_data(*data_dict*)

Saves a given data dict `{keyword: data}` at the last emulator iteration to the HDF5-file and as an data attribute to the current `Pipeline` instance.

Parameters data_dict (*dict*) – Dict containing the data that needs to be saved to the HDF5-file.

Keyword Arguments

- **keyword** (*{'impl_par'; 'impl_sam'; 'n_eval_sam'}*) – String specifying the type of data that needs to be saved.
- **data** (*{int; float; str; array_like} or dict*) – The actual data that needs to be saved at data keyword *keyword*. If dict, save every item individually.

Generates

The specified data is saved to the HDF5-file.

_save_statistics(*emul_i*, *stat_dict*)
Saves a given statistics dict `{keyword: [value, unit]}` at emulator iteration *emul_i* to the HDF5-file. The provided values are always saved as strings.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Keyword Arguments

- **keyword** (*str*) – String containing the name/keyword of the statistic that is being saved.
- **value** (*int, float or str*) – The value of the statistic.
- **unit** (*str*) – The unit of the statistic.

_set_impl_par(*impl_cut*)
Sets the *impl_cut* and *cut_idx* properties for implausibility evaluations using *prism_dict* and the provided *impl_cut*.

Parameters **impl_cut** (*list of float or None*) – Incomplete, shortened impl_cut-offs list to be used during the analysis of this emulator iteration. If *None*, use *prism_dict* instead.

Generates

impl_cut [1D ndarray object] Full list containing the impl_cut-offs for all data points provided to the emulator.

cut_idx [int] Index of the first impl_cut-off in the impl_cut list that is not 0.

_set_parameters()
Sets the Pipeline parameters from the *prism_dict* property and saves them in the current *Pipeline* instance.

analyze(*, *impl_cut=None*)
Analyzes the emulator at the last emulator iteration for a large number of emulator evaluation samples. All samples that survive the implausibility checks set by the provided *impl_cut*, are used in the construction of the next emulator iteration.

Other Parameters **impl_cut** (*list of float or None. Default: None*) – Incomplete, shortened implausibility cut-offs list to be used during the analysis of this emulator iteration. If *None*, the currently set implausibility cut-off values (*impl_cut*) will be used.

Generates

impl_sam [2D ndarray object] Array containing all emulator evaluation samples that survived the implausibility checks.

construct(*emul_i=None*, *, *analyze=True*, *ext_real_set=None*, *force=False*)

Constructs the emulator at the specified emulator iteration *emul_i*, and performs an implausibility analysis on the emulator iteration right afterward if requested (`analyze()`).

Other Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the next iteration of the emulator will be constructed.
- **analyze** (*bool. Default: True*) – Bool indicating whether or not to perform an analysis after the specified emulator iteration has been successfully constructed, which is required for making projections (`project()`) and constructing the next iteration.
- **ext_real_set** (*str, list, dict or None. Default: None*) – String indicating the suffix of the backup file that needs to be used; list of dicts containing an externally calculated set of model evaluation samples and its data values; a dict with keys `['sam_set', 'mod_set']` containing these dicts; or *None* if no external set needs to be used.
- **force** (*bool. Default: False*) – Controls what to do if the specified emulator iteration *emul_i* already (partly) exists. If *False*, finish construction of the specified iteration or skip it if already finished. If *True*, reconstruct the specified iteration entirely.

Generates

A new HDF5-group with the emulator iteration as its name, in the loaded emulator master file, containing emulator data required for this emulator iteration.

Notes

Using an emulator iteration that has been (partly) constructed before, will finish construction or skip it if already finished when *force* is *False*; or it will delete that and all following iterations, and reconstruct the specified iteration when *force* is *True*. Using *emul_i* = 1 and *force* is *True* is equivalent to reconstructing the entire emulator.

If no implausibility analysis is requested, then the implausibility parameters are taken from the *PRISM* parameters dict and temporarily stored in memory in order to enable the usage of the `evaluate()` method.

---

crystal()

Creates an instance of `QApplication` or retrieves it if one already exists, and starts *Crystal*, *PRISM*'s Projection GUI.

Crystal provides an interactive way of creating projection figures, as opposed to the static and linear method provided by `project()`. It is made to make it easier to create; view; compare; and analyze large numbers of projection figures. All options available in the `project()` method can also be accessed through *Crystal*.

As with all `Pipeline` user methods, this function must be called by all MPI ranks.

New in version 1.2.0.

details(*emul_i=None*)

Prints the details/properties of the currently loaded `Pipeline` instance at given emulator iteration *emul_i*. See `Props` for detailed descriptions of all printed properties.

Other Parameters **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the last iteration of the emulator will be used.

Props

Working directory The relative path to the working directory of the emulator starting at the current working directory.

Emulator type The type of this emulator, corresponding to the `emul_type` of the provided *emul_type* during `Pipeline` initialization. If no emulator type was provided during initialization, this is 'default'.

ModelLink subclass Name of the `ModelLink` subclass used to construct this emulator.

Emulation method Indicates the combination of regression and Gaussian emulation methods that have been used for this emulator.

Mock data used? Whether or not mock data has been used to construct this emulator. If so, the printed estimates for all model parameters are the parameter values used to create the mock data.

Emulator iteration The iteration of the emulator this details overview is about. By default, this is the last (partly) constructed iteration.

Construction completed? Whether or not the construction of this emulator iteration is completed. If not, the missing components for each emulator system are listed and the remaining information of this iteration is not printed.

Plausible regions? Whether or not plausible regions have been found during the analysis of this emulator iteration. If no analysis has been done yet, "N/A" will be printed.

Projections available? Whether or not projections have been created for this emulator iteration. If projections are available and analysis has been done, but with different implausibility cut-offs, a "desynced" note is added. Also prints number of available projections versus maximum number of projections in parentheses.

# of model evaluation samples The total number of model evaluation samples used to construct all emulator iterations up to this iteration, with the number for every individual iteration in parentheses.

# of plausible/analyzed samples The number of emulator evaluation samples that passed the implausibility check out of the total number of analyzed samples in this emulator iteration. This is the number of model evaluation samples that was/will be used for the construction of the next emulator iteration. If no analysis has been done, the numbers show up as "-".

% of parameter space remaining The percentage of the total number of analyzed samples that passed the implausibility check in this emulator iteration. If no analysis has been done, the number shows up as "-".

# of active/total parameters The number of model parameters that was considered active during the construction of this emulator iteration, compared to the total number of model parameters defined in the used `ModelLink` subclass.

# of emulated data points The number of data points that have been emulated in this emulator iteration.

# of emulator systems The total number of emulator systems that are required in this emulator. The number of active emulator systems is equal to the number of data points.

Parameter space Lists the name, lower and upper value boundaries and estimate (if provided) of all model parameters defined in the used `ModelLink` subclass. An asterisk is printed in front of the parameter name if this model parameter was considered active during the construction of this emulator iteration. A question mark is used instead if the construction of this emulator iteration is not finished.

evaluate(*sam_set*, *emul_i=None*)

Evaluates the given model parameter sample set *sam_set* up to given emulator iteration *emul_i*. The output of this function depends on the number of dimensions in *sam_set*. The output is always provided on the controller rank.

Parameters **sam_set** (*1D or 2D array_like or dict*) – Array containing model parameter value sets to be evaluated in the emulator up to emulator iteration *emul_i*.

Other Parameters **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the last iteration of the emulator will be used.

Returns

- **impl_check** (*list of bool*) – List of bool indicating whether or not the given samples passed the implausibility check at the given emulator iteration *emul_i*.
- **emul_i_stop** (*list of int*) – List containing the last emulator iterations at which the given samples are still within the plausible region of the emulator.
- **adj_exp_val** (2D `ndarray` object) – Array containing the adjusted expectation values for all given samples.
- **adj_var_val** (2D `ndarray` object) – Array containing the adjusted variance values for all given samples.
- **uni_impl_val** (2D `ndarray` object) – Array containing the univariate implausibility values for all given samples.

Prints (if 1D *sam_set*)

emul_i_stop [int] Last emulator iteration at which the given sample is still within the plausible region of the emulator.

adj_exp_val [1D `ndarray` object] The adjusted expectation values for the given sample.

adj_var_val [1D `ndarray` object] The adjusted variance values for the given sample.

sigma_val [1D `ndarray` object] The corresponding sigma value for the given sample.

uni_impl_val [1D `ndarray` object] The univariate implausibility values for the given sample.

Notes

If given emulator iteration *emul_i* has been analyzed before, the implausibility parameters of the last analysis are used. If not, then the values are used that were read in when the emulator was loaded or that have been set by the user.

---

project(*emul_i=None*, *proj_par=None*, ***kwargs*)

Analyzes the emulator iteration *emul_i* and constructs a series of projection figures detailing the behavior of the model parameters corresponding to the given *proj_par*. The input and output depend on the number of model parameters *n_par*.

All optional keyword arguments (except *force*) control various aspects of drawing the projection figures and do not affect the projection data that is saved to HDF5. This is instead influenced by the *proj_res* and *proj_depth* properties.

Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the last iteration of the emulator will be used.

- **proj_par** (*1D array_like of {int; str} or None. Default: None*) – For which model parameters to construct the projection figures. If 1D array_like, construct projection figures for all combinations of provided model parameters that are active, with a string referring to the name of the model parameter and an integer referring to the position in which the model parameter is shown in the `details()` method. If *None*, projection figures are made for all active model parameters.

Keyword Arguments

- **proj_type** (*{'2D'; '3D'; '2D+3D'}. Default: '2D' (2D), '2D+3D' (nD)*) – String indicating which projection type to create for all supplied active parameters. If `n_par` == 2, this is always '2D' (and cannot be modified).
- **figure** (*bool. Default: True*) – Whether or not to create the projection figures. If *True*, the figures are calculated, drawn and saved. If *False*, the figures are calculated and their data is returned in a dict.
- **align** (*{'row'/'horizontal'; 'col'/'column'/'vertical'}. Default: 'col'*) – If *figure* is *True*, string indicating how to position the two subplots. If 'row'/'horizontal', the subplots are positioned on a single row. If 'col'/'column'/'vertical', the subplots are positioned on a single column.
- **show_cuts** (*bool. Default: False*) – If *figure* is *True* and *proj_type* is not '3D', whether to show all implausibility cut-offs in the 2D projections (*True*) or only the first cut-off (*False*).
- **smooth** (*bool. Default: False*) – Controls what to do if a grid point contains no plausible samples, but does contain a minimum implausibility value below the first non-wildcard cut-off. If *False*, these values are kept, which can show up as artifact-like features in the projection figure. If *True*, these values are set to the first cut-off, removing them from the projection figure. Doing this may also remove interesting features. This does not affect the projection data saved to HDF5. Smoothed figures have an '_s' string appended to their filenames.
- **use_par_space** (*bool. Default: False*) – Controls whether to use the model parameter space (*True*) or the parameter space in which emulator iteration *emul_i* is defined (*False*) as the axes limits in the projection figure. If this is *True* and a parameter estimate is outside of the axes limit, a parameter estimate arrow is used instead of a line.
- **full_impl_rng** (*bool. Default: False*) – Controls whether to use zero (*True*) or the lowest plotted value (*False*) as the lower bound for the minimum implausibility plot. Set this to *True* for the same plotting behavior as in versions earlier than v1.2.4.

  New in version 1.2.4.
- **force** (*bool. Default: False*) – Controls what to do if a projection hypercube has been calculated at the emulator iteration *emul_i* before. If *False*, it will use the previously acquired projection data to create the projection figure. If *True*, it will recalculate all the data required to create the projection figure. Note that this will also delete all associated projection figures.
- **fig_kwargs** (*dict. Default: {'dpi': 100}*) – Dict of keyword arguments to be used when creating the subplots figure. It takes all optional arguments that can be provided to the `figure()` function.
- **impl_kwargs_2D** (*dict. Default: {}*) – Dict of keyword arguments to be used for making the minimum implausibility (top/left) plot in the 2D projection figures. It takes all optional arguments that can be provided to the `plot()` function.

- **impl_kwargs_3D** (*dict. Default: {'cmap': 'cmr.rainforest_r'}*) – Dict of keyword arguments to be used for making the minimum implausibility (top/left) plot in the 3D projection figures. It takes all optional arguments that can be provided to the `hexbin()` function.
- **los_kwargs_2D** (*dict. Default: {}*) – Dict of keyword arguments to be used for making the line-of-sight (bottom/right) plot in the 2D projection figures. It takes all optional arguments that can be provided to the `plot()` function.
- **los_kwargs_3D** (*dict. Default: {'cmap': 'cmr.freeze'}*) – Dict of keyword arguments to be used for making the line-of-sight (bottom/right) plot in the 3D projection figures. It takes all optional arguments that can be provided to the `hexbin()` function.
- **line_kwargs_est** (*dict. Default: {'linestyle': '–', 'color': 'grey'}*) – Dict of keyword arguments to be used for drawing the parameter estimate lines in both plots. It takes all optional arguments that can be provided to the `plot()` function.
- **arrow_kwargs_est** (*dict. Default: {'color': 'grey', 'fh_arrowlength': 0.015, 'fh_arrowwidth': 0.025, 'ft_arrowlength': 1.5, 'ft_arrowwidth': 0.003, 'rel_xpos': 0.5, 'rel_ypos': 0.5}*) – Dict of keyword arguments to be used for drawing the parameter estimate arrows in both plots. It takes a special set of arguments that are described in the **'Notes'_**.
- **line_kwargs_cut** (*dict. Default: {'color': 'r'}*) – Dict of keyword arguments to be used for drawing the implausibility cut-off line(s) in the top/left plot in the 2D projection figures. It takes all optional arguments that can be provided to the `plot()` function.

Returns **fig_data** (*dict of dicts*) – Dict containing the data for every requested projection figure, split up into the 'impl_min' and 'impl_los' dicts. For 2D projections, every dict contains a list with the x and y values. For 3D projections, it contains the x, y and z values. Note that due to the figures being interpolations, the y/z values can be below zero or the line-of-sight values being above unity.

Generates (if *figure* is *True*)

A series of projection figures detailing the behavior of the model. The lay-out and output of the projection figures depend on the type of figure:

2D projection figure: The output will feature a figure with two subplots for every active model parameter (`n_par`). Every figure gives details about the behavior of the corresponding model parameter, by showing the minimum implausibility value (top/left) and the line-of-sight depth (bottom/right) obtained at the specified parameter value, independent of the values of the other parameters.

3D projection figure (only if *`n_par`* > 2): The output will feature a figure with two subplots for every combination of two active model parameters that can be made (`n_par*(n_par-1)/2`). Every figure gives details about the behavior of the corresponding model parameters, as well as their dependency on each other. This is done by showing the minimum implausibility (top/left) and the line-of-sight depth (bottom/right) obtained at the specified parameter values, independent of the values of the remaining model parameters.

Notes

All colormaps defined in the `cmasher` package are loaded automatically when *PRISM* is imported and can be used.

Some *kwargs* dicts have a subset of arguments that are reserved for this function (like *x/y* for *impl_kwargs_2D* and co.). These arguments are ignored when provided.

The *arrow_kwargs_est* argument takes all optional arguments that `arrow()` takes, plus a few special arguments that are described below:

fh_arrowlength [float. Default: 0.015] The fraction of the appropriate axis limit used as the length of the arrow head.

fh_arrowwidth [float. Default: 0.03] The fraction of the appropriate axis limit used as the width of the arrow head.

ft_arrowlength [float. Default: 1.5] The relative length of the arrow tail compared to its head.

ft_arrowwidth [float. Default: 0.002] The fraction of the appropriate axis limit used as the width of the arrow tail.

rel_xpos [float. Default: 0.5] The relative x-position of the arrow. Only used if the arrow is drawn vertically and has no anchor in 3D projections.

rel_ypos [float. Default: 0.5] The relative y-position of the arrow. Only used if the arrow is drawn horizontally and has no anchor in 3D projections.

---

run(*emul_i=None*, ***, *force=False*)

Calls the *construct()* method to start the construction of the given iteration of the emulator and creates the projection figures right afterward if this construction was successful.

Other Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the next iteration of the emulator will be constructed.
- **force** (*bool. Default: False*) – Controls what to do if the specified emulator iteration *emul_i* already (partly) exists. If *False*, finish construction of the specified iteration or skip it if already finished. If *True*, reconstruct the specified iteration entirely.

start_gui()

Creates an instance of `QApplication` or retrieves it if one already exists, and starts *Crystal*, *PRISM*'s Projection GUI.

Crystal provides an interactive way of creating projection figures, as opposed to the static and linear method provided by *project()*. It is made to make it easier to create; view; compare; and analyze large numbers of projection figures. All options available in the *project()* method can also be accessed through *Crystal*.

As with all *Pipeline* user methods, this function must be called by all MPI ranks.

New in version 1.2.0.

File

Custom `File` class that has added logging and automatically uses *hdf5_file* as the HDF5-file to open.

Type `File`

base_eval_sam

Base number of emulator evaluations used to analyze the emulator systems. This number is scaled up by the number of model parameters to generate the true number of emulator evaluations (*n_eval_sam*).

Type int

code_objects

Collection of pre-compiled built-in code snippets that are used in the *_evaluate_sam_set()* method.

Type dict of code object

comm
The MPI intra-communicator that is used in this *Pipeline* instance. By default, this is `MPI.COMM_WORLD`.

Type `Intracomm`

criterion
Value indicating which criterion to use in the `lhd()` function.

Type str, float or None

cut_idx
The index of the first non-wildcard in a list of implausibility values. This is equivalent to the number of wildcards leading the cut-off values in *impl_cut*.

Type int

do_active_anal
Whether or not to do an active parameters analysis during the construction of the emulator systems.

Type bool

do_logging
Whether or not to save all logging messages. If *False*, all logging messages of level `INFO` and below are ignored. It also enables/disables the progress bar when making projections.

Type bool

emulator
The *Emulator* instance created during *Pipeline* initialization.

Type *Emulator*

freeze_active_par
Whether or not previously active parameters always stay active if possible.

Type bool

hdf5_file
Absolute path to the loaded master HDF5-file.

Type str

impl_cut
The non-wildcard univariate implausibility cut-off values for an emulator iteration. Setting it with the reduced implausibility cut-off list will change the values of *cut_idx* and this property at the last emulator iteration.

Type list of float

impl_sam
The model evaluation samples that will be added to the next emulator iteration.

Type list of dict

is_controller
Whether or not this MPI process is a controller rank. If no MPI is used, this is always *True*.

Type bool

is_worker
Whether or not this MPI process is a worker rank. If no MPI is used, this is always *False*.

Type bool

modellink
The *ModelLink* instance provided during *Pipeline* initialization.

Type *ModelLink*

n_eval_sam
The number of evaluation samples used to analyze an emulator iteration of the emulator systems. The number of plausible evaluation samples is stored in *n_impl_sam*. It is zero if the specified iteration has not been analyzed yet.

Type int

n_impl_sam
Number of model evaluation samples that passed the implausibility checks during the analysis of an emulator iteration. It is zero if the specified iteration has not been analyzed yet or has no plausible samples.

Type int

n_sam_init
Number of evaluation samples that will be used to construct the first iteration of the emulator systems.

Type int

pot_active_par
The potentially active parameters. Only parameters from this list can become active during the active parameters analysis. If *do_active_anal* is *False*, all parameters in this list will be active.

Type list of str

prism_dict
Dictionary containing all *PRISM* parameters that were provided during *Pipeline* initialization.

Type dict

proj_depth
Number of emulator evaluations that will be used to generate the samples in every grid point for the projection figures. Note that when making 2D projections of nD models, the used depth will be this number multiplied by *proj_res*.

Type int

proj_res
Number of emulator evaluations that will be used to generate the grid for the projection figures.

Type int

rank
The rank of this MPI process in *comm*. If no MPI is used, this is always 0.

Type int

root_dir
Absolute path to the root directory.

Type str

size
The number of MPI processes in *comm*. If no MPI is used, this is always 1.

Type int

worker_mode
Special context manager within which all code is executed in worker mode. In worker mode, all worker ranks are continuously listening for calls from the controller rank made with *_make_call()* or *_make_call_workers()*.

Note that all code within the context manager is executed by all ranks, with the worker ranks executing it after the controller rank exits. It is therefore advised to use an if-statement inside to make sure only the controller rank executes the code.

Using this context manager allows for easier use of *PRISM* in combination with serial/OpenMP codes (like MCMC methods). It also makes it easier to write long complex code that is mostly executed on the controller rank (but the worker ranks sometimes need to execute something).

All worker modes are independent of each other and can be created in a nested fashion.

Type *WorkerMode*

working_dir
Absolute path to the working directory.

Type str

class prism._pipeline.**WorkerMode**(*pipeline_obj*)

__enter__()
The provided *Pipeline* object enters worker mode, making all worker ranks start listening for calls from the controller rank until this context manager exits.

__exit__(*etype*, *value*, *tb*)
The provided *Pipeline* object exits worker mode, making all worker ranks stop listening for calls from the controller rank and resume normal code execution.

__init__(*pipeline_obj*)
Initialize the *WorkerMode* class using the MPI ranks defined in the provided *pipeline_obj*. This class should solely be initialized and finalized through the *Pipeline* class.

New in version 1.2.0.

Parameters **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that is enabling this worker mode.

static _process_call(*pipeline_obj*, *exec_fn*, *args*, *kwargs*)
Processes a call that was made with the *make_call()* or *make_call_workers()* method.

This function should solely be called through either of these methods and never directly.

Parameters

- **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that is making this call.
- **exec_fn** (*str or callable*) – If string, a callable attribute of this *Pipeline* instance or a callable object that should be executed if not.
- **args** (*tuple*) – Positional arguments that need to be provided to *exec_fn*.
- **kwargs** (*dict*) – Keyword arguments that need to be provided to *exec_fn*.

Returns **out** (*object*) – The object returned by executing *exec_fn*.

Note: If any entry in *args* or *kwargs* is a string written as 'pipe.XXX', it is assumed that 'XXX' refers to a *Pipeline* attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

---

static _process_call_str(*pipeline_obj*, *string*)
Processes a provided *string* that was provided as an argument value to *_process_call()*.

Parameters

- **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that is making this call.
- **string** (*str*) – String value that must be processed.

Returns **out** (*str or object*) – If *string* starts with 'pipe.', the corresponding *Pipeline* attribute will be returned. Else, *string* is returned.

listen_for_calls()

All worker ranks in the *comm* communicator start listening for calls from the corresponding controller rank and will attempt to execute the received message. Listening for calls continues until this context manager exits (*__exit__()* is called).

This method is automatically initialized and finalized when using the *worker_mode* context manager.

static make_call(*pipeline_obj*, *exec_fn*, **args*, ***kwargs*)

Sends the provided *exec_fn* to all worker ranks, if they are listening for calls, and tells them to execute it using the provided *args* and *kwargs*. All ranks that call this function will execute *exec_fn* as well.

If used within the *WorkerMode* context manager, this function should only be called by the controller. If not, it should be called by all valid ranks that must execute *exec_fn*.

Parameters

- **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that is making this call.
- **exec_fn** (*str or callable*) – If string, a callable attribute of this *Pipeline* instance or a callable object that the workers should execute if not.
- **args** (*positional arguments*) – Positional arguments that need to be provided to *exec_fn*.
- **kwargs** (*keyword arguments*) – Keyword arguments that need to be provided to *exec_fn*.

Returns **out** (*object*) – The object returned by executing *exec_fn*. Note that only ranks that directly call this function return, as workers in worker mode cannot do so.

Note: Changed in version 1.2.0: If any entry in *args* or *kwargs* is a string written as 'pipe.XXX', it is assumed that 'XXX' refers to a *Pipeline* attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

---

static make_call_workers(*pipeline_obj*, *exec_fn*, **args*, ***kwargs*)

Sends the provided *exec_fn* to all worker ranks, if they are listening for calls, and tells them to execute it using the provided *args* and *kwargs*.

If used within the *WorkerMode* context manager, this function should only be called by the controller. If not, it should be called by all valid ranks that must execute *exec_fn*.

Parameters

- **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that is making this call.
- **exec_fn** (*str or callable*) – If string, a callable attribute of this *Pipeline* instance or a callable object that the workers should execute if not.
- **args** (*positional arguments*) – Positional arguments that need to be provided to *exec_fn*.
- **kwargs** (*keyword arguments*) – Keyword arguments that need to be provided to *exec_fn*.

Returns out (*object*) – The object returned by executing *exec_fn*. Note that only ranks that directly call this function return, as workers in worker mode cannot do so.

Note: Changed in version 1.2.0: If any entry in *args* or *kwargs* is a string written as 'pipe.XXX', it is assumed that 'XXX' refers to a `Pipeline` attribute of the MPI rank receiving the call. It will be replaced with the corresponding attribute before *exec_fn* is called.

__weakref__
list of weak references to the object (if defined)

CHAPTER 10

Emulator

10.1 Classes

10.1.1 Emulator

class prism.emulator.**Emulator**(*pipeline_obj*, *modellink_obj*)

Defines the *Emulator* base class of the *PRISM* package.

Description

The *Emulator* class is the backbone of the *PRISM* package, holding all tools necessary to construct, load, save and evaluate the emulator of a model. It performs many checks to see if the provided *ModelLink* object is compatible with the current emulator, advises the user on alternatives when certain operations are requested, automatically takes care of distributing emulator systems over MPI ranks and more.

Even though the purpose of the *Emulator* class is to hold only information about the emulator and therefore does not require any details about the provided *ModelLink* object, it will keep track of changes made to it. This is to allow the user to modify the properties of the *ModelLink* subclass without causing any desynchronization problems by accident.

The *Emulator* class requires to be linked to an instance of the *Pipeline* class and will automatically attempt to do so when initialized. By default, this class should only be initialized from within a *Pipeline* object.

__init__(*pipeline_obj*, *modellink_obj*)

Initialize an instance of the *Emulator* class.

Parameters

- **pipeline_obj** (*Pipeline* object) – The *Pipeline* instance this *Emulator* instance should be linked to.
- **modellink_obj** (*ModelLink* object) – The *ModelLink* instance that should be linked to *pipeline_obj*.

_assign_data_idx (*emul_i*)

Determines the emulator system each data point in the provided emulator iteration *emul_i* should be assigned to, in order to make sure that recurring data points have the same emulator system index as in the previous emulator iteration. If multiple options are possible, data points are assigned such to spread them as much as possible.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns

- **data_to_emul_s** (*list of int*) – The index of the emulator system that each data point should be assigned to.
- **n_emul_s** (*int*) – The total number of active and passive emulator systems there will be in the provided emulator iteration.

Examples

If the number of data points is less than the previous iteration:

```
>>> emul_i = 2
>>> self._data_idx[emul_i-1]
['A', 'B', 'C', 'D', 'E']
>>> self._modellink._data_idx
['B', 'F', 'G', 'E']
>>> self._assign_data_idx(emul_i)
([1, 3, 2, 4], 5)
```

If the number of data points is more than the previous iteration:

```
>>> emul_i = 2
>>> self._data_idx[emul_i-1]
['A', 'B', 'C', 'D', 'E']
>>> self._modellink._data_idx
['B', 'F', 'G', 'E', 'A', 'C']
>>> self._assign_data_idx(emul_i)
([1, 5, 3, 4, 0, 2], 6)
```

If there is no previous iteration:

```
>>> emul_i = 1
>>> self._data_idx[emul_i-1]
[]
>>> self._modellink._data_idx
['B', 'F', 'G', 'E', 'A', 'C']
>>> self._assign_data_idx(emul_i)
([5, 4, 3, 2, 1, 0], 6)
```

_assign_emul_s (*emul_i*)

Determines which emulator systems (files) should be assigned to which MPI rank in order to balance the number of active emulator systems on every rank for every iteration up to the provided emulator iteration *emul_i*. If multiple choices can achieve this, the emulator systems are automatically spread out such that the total number of active emulator systems on a single rank is also balanced as much as possible.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **emul_s_to_core** (*list of lists*) – A list containing the emulator systems that have been assigned to the corresponding MPI rank by the controller.

Notes

Currently, this function only uses high-level MPI. Additional speed can be obtained by also implementing low-level MPI, which will potentially be done in the future.

_check_future_compat(*req_version*, *dep_version*)
Checks if the version of this emulator is compatible with the provided *req_version*. If not, raises a `FutureWarning`, indicating that the given *dep_version* will no longer support this emulator.

Parameters

- **req_version** (*str*) – The version in which an incompatible change was introduced.
- **dep_version** (*str*) – The version in which the backward compatibility for this change will be removed.

_cleanup_emul_files(*emul_i*)
Opens all emulator HDF5-files and removes the provided emulator iteration *emul_i* and subsequent iterations from them. Also removes any related projection figures that have default names. If *emul_i* == 1, all emulator HDF5-files are removed instead.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

_construct_iteration(*emul_i*)
Constructs the emulator iteration corresponding to the provided *emul_i*, by performing the given emulation method and pre-calculating the prior expectation and variance values of the used model evaluation samples.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Generates

All data sets that are required to evaluate the emulator at the constructed iteration.

_create_new_emulator()
Creates a new master HDF5-file that holds all the information of a new emulator and writes all important emulator details to it. Afterwards, resets all loaded emulator data and prepares the HDF5-file and emulator for the construction of the first emulator iteration.

Generates

A new master HDF5-file 'prism.hdf5' contained in the working directory specified in the *Pipeline* instance, holding all information required to construct the first iteration of the emulator.

_do_regression(*emul_i*, *emul_s_seq*)
Performs a forward stepwise linear regression for all requested emulator systems *emul_s_seq* in the provided emulator iteration *emul_i*. Calculates what the expectation values of all polynomial coefficients are. The polynomial order that is used in the regression depends on *poly_order*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

Generates (for every emulator system)

rsdl_var [float] Residual variance of the regression function.

regr_score [float] Fit-score of the regression function.

poly_coef [1D `ndarray` object] Array containing the expectation values of the non-zero polynomial coefficients.

poly_powers [2D `ndarray` object] Array containing the powers of the non-zero polynomial terms in the regression function.

poly_idx [1D `ndarray` object] Array containing the indices of the non-zero polynomial terms in the regression function.

poly_coef_cov [1D `ndarray` object (if `use_regr_cov` is *True*)] Array containing the covariance values of the non-zero polynomial coefficients.

_evaluate(*emul_i*, *par_set*)

Evaluates the emulator systems *emul_s_seq* at iteration *emul_i* for given *par_set*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.
- **par_set** (1D `ndarray` object) – Model parameter value set to evaluate the emulator at.

Returns

- **adj_exp_val** (1D `ndarray` object) – Adjusted emulator expectation value for all requested emulator systems on this MPI rank.
- **adj_var_val** (1D `ndarray` object) – Adjusted emulator variance value for all requested emulator systems on this MPI rank.

_get_active_par(*emul_i*, *emul_s_seq*)

Determines the active parameters to be used for every emulator system listed in *emul_s_seq* in the provided emulator iteration *emul_i*. Uses backwards stepwise elimination to determine the set of active parameters. The polynomial order that is used in the stepwise elimination depends on `poly_order`.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

Generates (for every emulator system)

active_par_data [1D `ndarray` object] Array containing the indices of all the parameters that are active in the emulator iteration *emul_i*.

_get_adj_exp(*emul_i*, *emul_s_seq*, *par_set*, *cov_vec*)

Calculates the adjusted emulator expectation values for requested emulator systems *emul_s_seq* at a given emulator iteration *emul_i* for specified parameter set *par_set* and corresponding covariance vector *cov_vec*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

- **par_set** (1D ndarray object) – Model parameter value set to calculate the adjusted emulator expectation for.
- **cov_vec** (2D ndarray object) – Covariance vector corresponding to *par_set*.

Returns **adj_exp_val** (1D ndarray object) – Adjusted emulator expectation value for all requested emulator systems on this MPI rank.

_get_adj_var(*emul_i*, *emul_s_seq*, *par_set*, *cov_vec*)
Calculates the adjusted emulator variance values for requested emulator systems *emul_s_seq* at a given emulator iteration *emul_i* for specified parameter set *par_set* and corresponding covariance vector *cov_vec*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.
- **par_set** (1D ndarray object) – Model parameter value set to calculate the adjusted emulator variance for.
- **cov_vec** (2D ndarray object) – Covariance vector corresponding to *par_set*.

Returns **adj_var_val** (1D ndarray object) – Adjusted emulator variance value for all requested emulator systems on this MPI rank.

_get_cov(*emul_i*, *emul_s_seq*, *par_set1*, *par_set2*)
Calculates the full emulator covariances for requested emulator systems *emul_s_seq* at emulator iteration *emul_i* for given parameter sets *par_set1* and *par_set2*. The contributions to these covariances depend on *method*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.
- **par_set1, par_set2** (1D ndarray object or None) – If *par_set1* and *par_set2* are both not *None*, calculate covariances for *par_set1* with *par_set2*. If *par_set1* is not *None* and *par_set2* is *None*, calculate covariances for *par_set1* with *sam_set* (covariance vector). If *par_set1* and *par_set2* are both *None*, calculate covariances for *sam_set* (covariance matrix). When not *None*, *par_set* is the model parameter value set to calculate the covariances for.

Returns **cov** (1D, 2D or 3D ndarray object) – Depending on the arguments provided, a covariance value, vector or matrix for requested emulator systems.

_get_cov_matrix(*emul_i*, *emul_s_seq*)
Calculates the (inverse) matrix of covariances between known model evaluation samples for requested emulator systems *emul_s_seq* at emulator iteration *emul_i*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

Generates

cov_mat [3D ndarray object] Matrix containing the covariances between all known model evaluation samples for requested emulator systems.

cov_mat_inv [3D ndarray object] Inverse of covariance matrix for requested emulator systems.

_get_data_idx_flat(*emul_i*)
Obtains the data point identifiers *data_idx* that are assigned to each MPI rank for the provided emulator iteration *emul_i* and returns them as a single list. The number of data points assigned to each MPI rank is also returned.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns

- **n_data_rank** (*list of int*) – The number of data points each MPI rank has assigned to it for given *emul_i*. These values can be used to split *data_idx_flat* up again into MPI rank-specific data point identifiers.
- **data_idx_flat** (*list of tuples*) – The data point identifiers that are assigned to an MPI rank for given *emul_i*.

_get_default_parameters()
Generates a dict containing default values for all emulator parameters.

Returns **par_dict** (*dict*) – Dict containing all default emulator parameter values.

_get_emul_i(*emul_i*, *cur_iter=True*)
Checks if the provided emulator iteration *emul_i* can be requested or replaces it if *None* was provided. This method requires all MPI ranks to call it simultaneously.

Parameters **emul_i** (*int or None*) – Number indicating the requested emulator iteration.

Other Parameters **cur_iter** (*bool*) – Bool determining whether the current (*True*) or the next (*False*) emulator iteration is requested.

Returns **emul_i** (*int*) – The requested emulator iteration that passed the check.

_get_emul_space(*emul_i*)
Returns the boundaries of the hypercube that encloses the parameter space in which the provided emulator iteration *emul_i* is defined.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **emul_space** (2D `ndarray` object) – The requested hypercube boundaries. If *emul_i* == 1, this is equal to the defined model parameter space.

Note: The parameter space over which an emulator iteration is defined is always equal to the plausible space of the previous iteration.

_get_exp_dot_term(*emul_i*, *emul_s_seq*)
Pre-calculates the second expectation adjustment dot-term for requested emulator systems *emul_s_seq* at a given emulator iteration *emul_i* for all model evaluation samples and saves it for later use.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

Generates

exp_dot_term [2D `ndarray` object] 2D array containing the pre-calculated values for the second adjustment dot-term of the adjusted expectation for requested emulator systems.

_get_inv_matrix(*matrix*)
Calculates the inverse of a given *matrix*. Right now only uses the `pinv()` function.

Parameters **matrix** (*2D array_like*) – Matrix to be inverted.

Returns **matrix_inv** (2D `ndarray` object) – Inverse of the given *matrix*.

_get_poly_term_str(*active_par*, *poly_power*)
Returns the string representation of a polynomial term given by the provided *active_par* and *poly_power*.

Parameters

- **active_par** (*list of int*) – List containing the indices of the parameters whose polynomial powers are given in *poly_power*.
- **poly_power** (*list of int*) – List with the powers of the requested polynomial term.

Returns **poly_term** (*str*) – String representation of the requested polynomial term.

_get_prior_exp(*emul_i*, *emul_s_seq*, *par_set*)
Calculates the prior expectation value for requested emulator systems *emul_s_seq* at a given emulator iteration *emul_i* for specified parameter set *par_set*. This expectation depends on *method*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.
- **par_set** (1D `ndarray` object or None) – If *None*, calculate the prior expectation values of sam_set. If not *None*, calculate the prior expectation value for the given model parameter value set.

Returns **prior_exp** (1D or 2D `ndarray` object) – Prior expectation values for either sam_set or *par_set* for requested emulator systems.

_get_regr_cov(*emul_i*, *emul_s_seq*, *par_set1*, *par_set2*)
Calculates the covariances of the regression function for requested emulator systems *emul_s_seq* at emulator iteration *emul_i* for given parameter sets *par_set1* and *par_set2*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.
- **par_set1, par_set2** (1D `ndarray` object or None) – If *par_set1* and *par_set2* are both not *None*, calculate regression covariances for *par_set1* with *par_set2*. If *par_set1* is not *None* and *par_set2* is *None*, calculate regression covariances for *par_set1* with *sam_set* (covariance vector). If *par_set1* and *par_set2* are both *None*, calculate regression covariances for *sam_set* (covariance matrix). When not *None*, *par_set* is the model parameter value set to calculate the regression covariances for.

Returns **regr_cov** (1D, 2D or 3D `ndarray` object) – Depending on the arguments provided, a regression covariance value, vector or matrix for requested emulator systems.

_get_rsdl_var(*emul_i*, *emul_s_seq*)
Splits up the calculated residual variances for requested emulator systems *emul_s_seq* at emulator iteration *emul_i* into active and passive contributions.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **emul_s_seq** (*list of int*) – List of numbers indicating the requested emulator systems.

Generates

act_rsdl_var [list of float] List containing the active portions of the residual variances.

pas_rsdl_var [list of float] List containing the passive portions of the residual variances. If `f_infl` is not zero, this also includes the inflated residual variance values.

_load_data(*emul_i*)
Loads in all the important emulator data up to emulator iteration *emul_i* into memory.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Generates

All relevant emulator data up to emulator iteration *emul_i* is loaded into memory.

_load_emulator(*modellink_obj*)
Checks if the provided working directory contains a constructed emulator and loads in the emulator data accordingly.

Parameters **modellink_obj** (`ModelLink` object) – Instance of the `ModelLink` class that links the emulated model to this `Pipeline` object.

_prepare_new_iteration(*emul_i*)
Prepares the emulator for the construction of a new iteration *emul_i*. Checks if this iteration can be prepared or if it has been prepared before, and acts accordingly.

Parameters **emul_i** (*int*) – Number indicating the requested emulator iteration.

Returns **reload** (*bool*) – Bool indicating whether or not the controller rank of the `Pipeline` instance needs to reload its data.

Generates

A new group in the master HDF5-file with the emulator iteration as its name, containing subgroups corresponding to all emulator systems that will be used in this iteration.

Notes

Preparing an iteration that has been prepared before, causes that and all subsequent iterations of the emulator to be deleted. A check is carried out to see if it was necessary to reprepare the requested iteration and a warning is given if this check fails.

_read_data_idx(*emul_s_group*)
Reads in and combines the parts of the data point identifier that is assigned to the provided *emul_s_group*.

Parameters **emul_s_group** (`Group` object) – The HDF5-group from which the data point identifier needs to be read in.

Returns **data_idx** (*tuple of {int, float, str}*) – The combined data point identifier.

_retrieve_parameters()
Reads in the emulator parameters from the provided working directory and saves them in the current `Emulator` instance.

_save_data(*emul_i*, *lemul_s*, *data_dict*)
Saves a given data dict `{keyword: data}` at the given emulator iteration *emul_i* and local emulator system *lemul_s* to the HDF5-file and as an data attribute to the current `Emulator` instance.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **lemul_s** (*int or None*) – Number indicating the requested local emulator system. If *None*, use the master emulator file instead.
- **data_dict** (*dict*) – Dict containing the data that needs to be saved to the HDF5-file.

Keyword Arguments

- **keyword** (*{'active_par'; 'active_par_data'; 'cov_mat'; 'exp_dot_term'; 'mod_real_set'; 'regression', 'rsdl_var'}*) – String specifying the type of data that needs to be saved.
- **data** (*{int; float; str; array_like} or dict*) – The actual data that needs to be saved at data keyword *keyword*. If dict, save every item individually.

Generates

The specified data is saved to the HDF5-file.

_set_mock_data()
Loads previously used mock data into the *ModelLink* object, overwriting the parameter estimates, data values, data errors, data spaces and data identifiers with their mock equivalents.

Generates

Overwrites the corresponding *ModelLink* class properties with the previously used values (taken from the first emulator iteration).

_set_modellink(*modellink_obj*, *modellink_loaded*)
Sets the *ModelLink* object that will be used for constructing this emulator. If a constructed emulator is present, checks if provided *modellink_obj* argument matches the *ModelLink* subclass used to construct it.

Parameters

- **modellink_obj** (*ModelLink* object) – Instance of the *ModelLink* class that links the emulated model to this *Pipeline* object. The provided *ModelLink* object must match the one used to construct the loaded emulator.
- **modellink_loaded** (*str or None*) – If str, the name of the *ModelLink* subclass that was used to construct the loaded emulator. If *None*, no emulator is loaded.

_set_parameters()
Sets the Emulator parameters from the *prism_dict* property and saves them in the current *Emulator* instance.

_set_sam_set_data(*emul_i*, *sam_set*)
Sets the provided *sam_set* as the iteration data at the given emulator iteration *emul_i*.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **sam_set** (2D ndarray object) – Array containing the model evaluation samples for emulator iteration *emul_i*.

_write_data_idx(*emul_s_group*, *data_idx*)
Splits a given *data_idx* up into individual parts and saves it as an attribute to the provided *emul_s_group*.

Parameters

- **emul_s_group** (`Group` object) – The HDF5-group to which the data point identifier needs to be saved.
- **data_idx** (*tuple of {int, float, str}*) – The data point identifier to be saved.

__weakref__
list of weak references to the object (if defined)

act_rsdl_var
The active contribution of the residual variance of every emulator system on this MPI rank. Obtained from either *rsdl_var* (regression) or *sigma* (Gaussian).

Type dict of float

active_emul_s
The indices of the emulator systems on this MPI rank that are active.

Type list of int

active_par
The model parameter names that are considered active. Only available on the controller rank.

Type list of str

active_par_data
The model parameter names that are considered active for every emulator system on this MPI rank.

Type dict of lists

ccheck
The emulator system components that are still required to complete the construction of an emulator iteration on this MPI rank. The controller rank additionally lists the required components that are emulator iteration specific ('mod_real_set' and 'active_par').

Type list of str

cov_mat_inv
The inverses of the covariance matrices for every emulator system on this MPI rank.

Type dict of `ndarray`

data_idx_to_core
List of the data identifiers that were assigned to the emulator systems listed in *emul_s_to_core*. Only available on the controller rank.

Type list of lists

emul_i
The last emulator iteration that is fully constructed for all emulator systems on this MPI rank.

Type int

emul_load
Whether or not a previously constructed emulator is currently loaded.

Type bool

emul_s
The indices of the emulator systems that are assigned to this MPI rank.

Type list of int

emul_s_to_core
List of the indices of the emulator systems that are assigned to every MPI rank. Only available on the controller rank.

Type list of lists

emul_space
The boundaries of the hypercube that encloses the parameter space in which the specified emulator iteration is defined. This is always equal to the plausible space of the previous iteration.

Type dict of ndarray

emul_type
The type of emulator that is currently loaded. This determines the way in which the *Pipeline* instance will treat this *Emulator* instance.

Type str

exp_dot_term
The second expectation adjustment dot-term values of all model evaluation samples for every emulator system on this MPI rank.

Type dict of ndarray

f_infl
The residual variance inflation factor. The prior variance of all known samples is inflated by this factor multiplied with *rsdl_var* (regression) or *sigma* (Gaussian). If this value is zero, no variance inflation is performed.

Type float

l_corr
The Gaussian correlation lengths for all model parameters, which is defined as the maximum distance between two values of a specific model parameter within which the Gaussian contribution to the correlation between the values is still significant.

Type dict of float

method
The emulation method to use for constructing the emulator. Possible are 'gaussian', 'regression' and 'full'.

Type str

mod_set
The model outputs corresponding to the samples in *sam_set* for every emulator system on this MPI rank.

Type dict of ndarray

n_cross_val
Number of (k-fold) cross-validations that are used for determining the quality of the regression process. It is set to zero if cross-validations are not used. If *method* == 'gaussian' and *do_active_anal* is *False*, this number is not required.

Type int

n_emul_s
Number of emulator systems assigned to this MPI rank.

Type int

n_emul_s_tot
Total number of emulator systems assigned to all MPI ranks combined. Only available on the controller rank.

Type int

n_sam
Number of model evaluation samples that have been/will be used to construct an emulator iteration.

Type int

pas_rsdl_var
The passive contribution of the residual variance of every emulator system on this MPI rank. If *f_infl* is not zero, this also includes the inflated residual variance value. Obtained from either *rsdl_var* (regression) or *sigma* (Gaussian).

Type dict of float

poly_coef
The non-zero coefficients of the polynomial terms in the regression function for every emulator system on this MPI rank. Empty if *method* == 'gaussian'.

Type list of ndarray

poly_coef_cov
The covariances of all coefficients in *poly_coef* for every emulator system on this MPI rank. Empty if *method* == 'gaussian' or *use_regr_cov* is *False*.

Type list of ndarray

poly_idx
The indices of all polynomial terms with non-zero coefficients in the regression function for every emulator system on this MPI rank. Empty if *method* == 'gaussian'.

Type list of ndarray

poly_order
Polynomial order that is considered for the regression process. If *method* == 'gaussian' and *do_active_anal* is *False*, this number is not required.

Type int

poly_powers
The powers of all polynomial terms with non-zero coefficients in the regression function for every emulator system on this MPI rank. Empty if *method* == 'gaussian'.

Type list of ndarray

poly_terms
Overview of all polynomial terms with non-zero coefficients in the regression function for every emulator system on this MPI rank. Empty if *method* == 'gaussian'.

This is basically a human-readable representation of *poly_coef* plus *poly_powers*. Given its formatting, it is not advised to use this for any operations.

Type dict of dict

prism_version
The version of *PRISM* that was used to construct the emulator that is currently loaded.

Type str

rsdl_var
The residual variance of every emulator system on this MPI rank. Obtained from regression process and replaces the Gaussian sigma. Empty if *method* == 'gaussian'.

Type dict of float

sam_set
The model evaluation samples that have been/will be used to construct the specified emulator iteration.

Type list of dict

sigma
Value of the Gaussian sigma. If *method* != 'gaussian', this value is not required, since it is obtained from the regression process instead.

Type float

use_mock
Whether or not mock data has been used for the construction of this emulator instead of actual data. If *True*, changes made to the data in the provided *ModelLink* object are ignored.

Type bool

use_regr_cov
Whether or not to take into account the regression covariance when calculating the covariance of the emulator, in addition to the Gaussian covariance. If *method* == 'gaussian', this bool is not required. If *method* == 'regression', this bool is always set to *True*.

Type bool

CHAPTER 11

ModelLink

11.1 Classes

11.1.1 GaussianLink

class prism.modellink.**GaussianLink**(*n_gaussians=1*, **args*, ***kwargs*)

`ModelLink` class wrapper for a simple Gaussian model, used for testing the functionality of the *PRISM* pipeline.

Formatting data_idx

x [float] The value that needs to be used for x in the function $\sum_i A_i \exp\left(-\frac{(x-B_i)^2}{2C_i^2}\right)$ to obtain the data value.

__init__(*n_gaussians=1*, **args*, ***kwargs*)

Initialize an instance of the `GaussianLink` class.

Other Parameters **n_gaussians** (*int. Default: 1*) – The number of Gaussians to use for the Gaussian model in this instance. The resulting number of model parameters `n_par` will be $3 * n_{gaussians}$.

n_gaussians

Number of Gaussians used in this `GaussianLink` instance.

Type int

11.1.2 ModelLink

class prism.modellink.**ModelLink**(*, *model_parameters=None*, *model_data=None*)

Provides an abstract base class definition that allows the `Pipeline` class to be linked to any model/test object of choice. Every model wrapper used in the `Pipeline` class must be an instance of the `ModelLink` class.

Description

The *ModelLink* class is an abstract base class, which forms the base for wrapping a model and allowing *PRISM* to use it effectively. Because it is mandatory for every model to be wrapped in a user-made *ModelLink* subclass, several tools are provided to the user to make this as versatile as possible.

The *ModelLink* class uses three properties that define the way the subclass will be used by *PRISM*: *name*, *call_type* and *MPI_call*. The first defines what the name of the subclass is, which is used by *PRISM* to identify the subclass with and check if one did not use a different subclass by accident. The other two are flags that determine how the *call_model()* method should be used. These three properties can be set anywhere during the initialization of the *ModelLink* subclass, or are set to a default value if they are not modified.

Every *ModelLink* subclass needs to be provided with two different data sets: *model parameters* and *model data*. The model parameters define which parameters the model can take, what their names are and in what value range each parameter must be. The model data on the other hand, states where in a model realization a data value must be retrieved and compared with a provided observational value. One can think of the model data as the observational constraints used to calculate the likelihood in a Bayesian analysis.

The model parameters and model data can be set in two different ways. They can be hard-coded into the *ModelLink* subclass by altering the *get_default_model_parameters()* and *get_default_model_data()* methods or set by providing them during class initialization. A combination of both is also possible. More details on this can be found in *__init__()*.

The *ModelLink* class has two abstract methods that must be overridden before the subclass can be initialized. The *call_model()* method is the most important method, as it provides *PRISM* with a way of calling the model wrapped in the *ModelLink* subclass. The *get_md_var()* method allows for *PRISM* to calculate the model discrepancy variance.

Notes

The *__init__()* method may be extended by the *ModelLink* subclass, but the superclass version must always be called.

If required, one can use the *test_subclass()* function to test a *ModelLink* subclass on correct functionality.

_ModelLink__set_model_data(*add_model_data*)

Generates the model data properties from the default model data and the additional input argument *add_model_data*.

Parameters **add_model_data** (*array_like, dict, str or None*) – Anything that can be converted to a dict that provides non-default model data information or *None* if only default data is used from *get_default_model_data()*.

Generates

n_data [int] Number of provided data points.

data_val [list] List with values of provided data points.

data_err [list of lists] List with upper and lower 1σ-confidence levels of provided data points.

data_spc [list] List with types of value space ({'lin', 'log', 'ln'}) of provided data points.

data_idx [list of tuples] List with user-defined data point identifiers.

_ModelLink__set_model_parameters(*add_model_parameters*)
Generates the model parameter properties from the default model parameters and the additional input argument *add_model_parameters*.

Parameters **add_model_parameters** (*array_like, dict, str or None*) – Anything that can be converted to a dict that provides non-default model parameters information or *None* if only default information is used from *get_default_model_parameters()*.

Generates

n_par [int] Number of model parameters.

par_name [list] List with model parameter names.

par_rng [ndarray object] Array containing the lower and upper values of the model parameters.

par_est [list] List containing user-defined estimated values of the model parameters. Contains *None* in places where estimates were not provided.

__init__(*, *model_parameters=None*, *model_data=None*)
Initialize an instance of the *ModelLink* subclass.

Other Parameters **model_parameters, model_data** (*array_like, dict, str or None. Default: None*) – Anything that can be converted to a dict that provides non-default model parameters/data information or *None* if only default information is used from *get_default_model_parameters()* or *get_default_model_data()*. For more information on the lay-out of these dicts, see Notes.

If array_like, dict(*model_parameters*/*model_data*) must generate a dict with the correct lay-out. If dict, the dict itself must have the correct lay-out. If str, the string must be the path to a file containing the dict keys in the first column and the dict values in the second column, which combined generate a dict with the correct lay-out.

Notes (model_parameters)

The model parameters provides this *ModelLink* subclass with the names, ranges and estimates of all model parameters that need to be explored.

The model parameters dict requires to have the name of the parameters as the keyword, and a 1D list containing the lower bound, the upper bound and, if applicable, the estimate of this parameter. It is not required to provide an estimate for every parameter. The estimates are used to draw illustrative lines when making projection figures. An example of a model parameters file can be found in the 'data' folder of the *PRISM* package. If required, one can use the *convert_parameters()* function to validate their formatting.

Formatting : `{par_name:  [lower_bnd, upper_bnd, par_est]}`

Notes (model_data)

The model data provides this *ModelLink* subclass with the observational data points that need to be used to constrain this model with.

The model data dict requires to have the data identifiers (*data_idx*) as the keyword, and a 1D list containing the data value (*data_val*); the data errors (*data_err*) and the data space (*data_spc*).

If the data errors are given with one value, then the data points are assumed to have a centered 1σ-confidence interval. If the data errors are given with two values, then the data points are assumed to have a 1σ-confidence interval defined by the provided upper and lower errors.

The data spaces are one of five strings ({'lin', 'log' or 'log_10', 'ln' or 'log_e'}) indicating in which of the three value spaces (linear, log, ln) the data values are. It defaults to 'lin' if it is not provided.

The data identifier is a sequence of bools, ints, floats and strings that is unique for every data point. *PRISM* uses it to identify a data point with, which is required in some cases (like MPI), while the model itself can use it as a description of the operations required to extract the data point from the model output. It can be provided as any sequence of any length for any data point. If any sequence contains a single element, it is replaced by just that element instead of a tuple.

A simple example of a data identifier is $f($

_check_md_var(*md_var*, *name*)

Checks validity of provided set of model discrepancy variances *md_var* in this *ModelLink* instance.

Parameters

- **md_var** (*1D or 2D array_like or dict*) – Model discrepancy variance set to validate in this *ModelLink* instance.
- **name** (*str*) – The name of the model discrepancy set, which is used in the error message if the validation fails.

Returns **md_var** (2D ndarray object) – The (converted) provided *md_var* if the validation was successful. If *md_var* was a dict, it will be converted to a ndarray object.

_check_mod_set(*mod_set*, *name*)

Checks validity of provided set of model outputs *mod_set* in this *ModelLink* instance.

Parameters

- **mod_set** (*1D or 2D array_like or dict*) – Model output (set) to validate in this *ModelLink* instance.
- **name** (*str*) – The name of the model output (set), which is used in the error message if the validation fails.

Returns **mod_set** (1D or 2D ndarray object) – The provided *mod_set* if the validation was successful. If *mod_set* was a dict, it will be converted to a ndarray object (sorted on *data_idx*).

_check_sam_set(*sam_set*, *name*)

Checks validity of provided set of model parameter samples *sam_set* in this *ModelLink* instance.

Parameters

- **sam_set** (*1D or 2D array_like or dict*) – Parameter/sample set to validate in this *ModelLink* instance.
- **name** (*str*) – The name of the parameter/sample set, which is used in the error message if the validation fails.

Returns **sam_set** (1D or 2D ndarray object) – The provided *sam_set* if the validation was successful. If *sam_set* was a dict, it will be converted to a ndarray object.

_get_backup_path(*emul_i*, *suffix*)

Returns the absolute path to a backup file made by this *ModelLink* instance, using the provided *emul_i* and *suffix*.

This method is used by the *_make_backup()* and *_read_backup()* methods, and should not be called directly.

Parameters

- **emul_i** (*int*) – The emulator iteration for which a backup filepath is needed.
- **suffix** (*str or None*) – If str, determine path to associated backup file using provided *suffix*. If *suffix* is empty, obtain last created backup file. If *None*, create a new path to a backup file.

Returns **filepath** (*str*) – Absolute path to requested backup file.

_get_model_par_seq(*par_seq*, *name*)

Converts a provided sequence *par_seq* of model parameter names and indices to a list of indices, removes duplicates and checks if every provided name/index is valid.

Parameters

- **par_seq** (*1D array_like of {int, str}*) – A sequence of integers and strings determining which model parameters need to be used for a certain operation.
- **name** (*str*) – A string stating the name of the variable the result of this method will be stored in. Used for error messages.

Returns **par_seq_conv** (*list of int*) – The provided sequence *par_seq* converted to a sorted list of model parameter indices.

_get_sam_dict(*sam_set*)

Converts a provided set of model parameter samples *sam_set* to a dict for use in this `ModelLink` instance.

This dict can be used as the *par_set* argument in the `call_model()` and `get_md_var()` methods.

Parameters **sam_set** (*1D or 2D array_like*) – Parameter/sample set to convert for this `ModelLink` instance.

Returns **sam_dict** (*dict of list*) – Dict of parameter samples.

_get_sam_space(*sam_set*)

Returns the boundaries of the hypercube that encloses the parameter space in which the provided *sam_set* is defined.

The main use for this function is to determine what part of model parameter space was likely sampled from in order to obtain the provided *sam_set*. Because of this, extra spacing is added to the boundaries to reduce the effect of the used sampling method.

Parameters **sam_set** (*1D or 2D array_like or dict*) – Parameter/sample set for which an enclosing hypercube is requested.

Returns **sam_space** (2D `ndarray` object) – The requested hypercube boundaries.

_make_backup(**args*, ***kwargs*)

WARNING: This is an advanced utility method and probably will not work unless used properly. Use with caution!

Creates an HDF5-file backup of the provided *args* and *kwargs* when called by the `call_model()` method or any of its inner functions. Additionally, the backup will contain the *emul_i*, *par_set* and *data_idx* values that were passed to the `call_model()` method. It also contains the version of *PRISM* that made the backup. The backup can be restored using the `_read_backup()` method.

If it is detected that this method is used incorrectly, a `RequestWarning` is raised (and the method returns) rather than a `RequestError`, in order to not disrupt the call to `call_model()`.

Parameters

- **args** (*positional arguments*) – All positional arguments that must be stored in the backup file.

- **kwargs** (*keyword arguments*) – All keyword arguments that must be stored in the backup file.

Notes

The name of the created backup file contains the value of *emul_i*, `name` and a random string to avoid replacing an already existing backup file.

The saved *emul_i*, *par_set* and *data_idx* are the values these variables have locally in the `call_model()` method at the point this method is called. Because of this, making any changes to them may cause problems and is therefore heavily discouraged. If changes are necessary, it is advised to copy them to a different variable first.

`_read_backup`(*emul_i*, *, *suffix=None*)

Reads in a backup HDF5-file created by the `_make_backup()` method, using the provided *emul_i* and the value of `name`.

Parameters **emul_i** (*int*) – The emulator iteration that was provided to the `call_model()` method when the backup was made.

Other Parameters **suffix** (*str or None. Default: None*) – The suffix of the backup file (everything between parentheses) that needs to be read. If *None* or empty, the last created backup will be read.

Returns

- **filename** (*str*) – The absolute path to the backup file that has been read.
- **data** (dict with keys *('emul_i', 'prism_version', 'par_set', 'data_idx', 'args', 'kwargs')*) – A dict containing the data that was provided to the `_make_backup()` method.

`_to_par_space`(*sam_set*)

Converts provided *sam_set* from unit space ([0, 1]) to parameter space ([lower_bnd, upper_bnd]).

`_to_unit_space`(*sam_set*)

Converts provided *sam_set* from parameter space ([lower_bnd, upper_bnd]) to unit space ([0, 1]).

`call_model`(*emul_i*, *par_set*, *data_idx*)

Calls the model wrapped in this `ModelLink` subclass at emulator iteration *emul_i* for model parameter values *par_set* and returns the data points corresponding to *data_idx*.

This method is called with solely keyword arguments.

This is an abstract method and must be overridden by the `ModelLink` subclass.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **par_set** (dict of `float64`) – Dict containing the values for all model parameters corresponding to the requested model realization(s). If model is single-called, dict item is formatted as `{par_name: par_val}`. If multi-called, it is formatted as `{par_name: [par_val_1, par_val_2, ..., par_val_n]}`.
- **data_idx** (*list of tuples*) – List containing the user-defined data point identifiers corresponding to the requested data points.

Returns **data_val** (*1D or 2D array_like or dict*) – Array containing the data values corresponding to the requested data points generated by the requested model realization(s). If model is multi-called, *data_val* is of shape `(n_sam, n_data)`. If dict, it has the identifiers in *data_idx* as its keys with either scalars or 1D array_likes as its values.

Note: If this model is multi-called, then the parameter sets in the provided *par_set* dict will be sorted in order of parameter name (e.g., sort on first parameter first, then on second parameter, etc.).

get_default_model_data()
Returns the default model data to use for every instance of this *ModelLink* subclass. By default, returns *_default_model_data*.

get_default_model_parameters()
Returns the default model parameters to use for every instance of this *ModelLink* subclass. By default, returns *_default_model_parameters*.

get_md_var(*emul_i*, *par_set*, *data_idx*)
Calculates the linear model discrepancy variance at a given emulator iteration *emul_i* for model parameter values *par_set* and given data points *data_idx* for the model wrapped in this *ModelLink* subclass.

This method is always single-called by one MPI rank with solely keyword arguments.

This is an abstract method and must be overridden by the *ModelLink* subclass.

Parameters

- **emul_i** (*int*) – Number indicating the requested emulator iteration.
- **par_set** (dict of `float64`) – Dict containing the values for all model parameters corresponding to the requested model realization.
- **data_idx** (*list of tuples*) – List containing the user-defined data point identifiers corresponding to the requested data points.

Returns **md_var** (*1D or 2D array_like*) – Array containing the linear model discrepancy variance values corresponding to the requested data points. If 1D array_like, data is assumed to have a centered one sigma confidence interval. If 2D array_like, the values determine the upper and lower variances and the array is of shape `(n_data, 2)`. If dict, it has the identifiers in *data_idx* as its keys with either scalars or 1D array_likes of length 2 as its values.

Notes

The returned model discrepancy variance values must be of linear form, even for those data values that are returned in logarithmic form by the *call_model()* method. If not, the possibility exists that the emulation process will not converge properly.

get_str_repr()
Returns a list of string representations of all additional input arguments with which this *ModelLink* subclass was initialized.

MPI_call
Whether *call_model()* can/should be called by all MPI ranks simultaneously instead of by the controller. By default, only the controller rank calls the model (False).

Type bool

__weakref__
list of weak references to the object (if defined)

_default_model_data
The default model data to use for every instance of this *ModelLink* subclass.

Type dict

_default_model_parameters
The default model parameters to use for every instance of this *ModelLink* subclass.

Type dict

call_type
String indicating whether *call_model()* should be supplied with a single evaluation sample ('single') or a set of samples ('multi'), or can be supplied with both ('hybrid'). By default, single model calls are requested ('single').

Type str

data_err
The upper and lower 1σ-confidence levels of provided data points.

Type dict of float

data_idx
The user-defined data point identifiers.

Type list of tuples

data_spc
The types of value space ({'lin', 'log', 'ln'}) of provided data points.

Type dict of str

data_val
The values of provided data points.

Type dict of float

model_data
The dict of model data points as used by this *ModelLink* instance.

This dict can be used as the *model_data* argument when initializing this *ModelLink* subclass.

Type dict of list

model_parameters
The dict of model parameters as used by this *ModelLink* instance.

This dict can be used as the *model_parameters* argument when initializing this *ModelLink* subclass.

Type dict of list

multi_call
Whether *call_model()* can/should be supplied with a set of evaluation samples. At least one of *single_call* and *multi_call* must be *True*. By default, single model calls are requested (False).

Type bool

n_data
Number of provided data points.

Type int

n_par
Number of model parameters.

Type int

name
Name associated with an instance of this *ModelLink* subclass. By default, it is set to the name of this *ModelLink* subclass. Can be manually manipulated to allow for more user control.

Type str

par_est
The user-defined estimated values of the model parameters. Contains *None* in places where estimates were not provided.

Type dict of {float, None}

par_name
List with model parameter names.

Type list of str

par_rng
The lower and upper values of the model parameters.

Type dict of ndarray

single_call
Whether *call_model()* can/should be supplied with a single evaluation sample. At least one of *single_call* and *multi_call* must be *True*. By default, single model calls are requested (True).

Type bool

11.1.3 PolyLink

class prism.modellink.**PolyLink**(*order=2*, **args*, ***kwargs*)
ModelLink class wrapper for a simple polynomial model, used for testing the functionality of the *PRISM* pipeline.

Formatting data_idx

x [float] The value that needs to be used for x in the function $\sum_i C_i x^i$ to obtain the data value.

__init__(*order=2*, **args*, ***kwargs*)
Initialize an instance of the *PolyLink* class.

Other Parameters **order** (*int. Default: 2*) – The polynomial order to use for the polynomial model in this instance. The resulting number of model parameters *n_par* will be $1 + \text{order}$.

order
Polynomial order used in this *PolyLink* instance.

Type int

11.1.4 SineWaveLink

class prism.modellink.**SineWaveLink**(**args*, ***kwargs*)
ModelLink class wrapper for a simple sine wave model, used for testing the functionality of the *PRISM* pipeline.

Formatting data_idx

x [float] The value that needs to be used for x in the function $A + 0.1 * B * \sin(C * x + D)$ to obtain the data value.

11.2 Utilities

prism.modellink.**convert_data**(*model_data*)

Converts the provided *model_data* into a full data dict, taking into account all formatting options, and returns it.

This function can be used externally to check how the provided *model_data* would be interpreted when provided to the `ModelLink` subclass. Its output can be used for the 'model_data' input argument.

Parameters **model_data** (*array_like, dict or str*) – Anything that can be converted to a dict that provides model data information.

Returns **data_dict** (*dict*) – Dict with the provided *model_data* converted to its full format.

prism.modellink.**convert_parameters**(*model_parameters*)

Converts the provided *model_parameters* into a full parameters dict, taking into account all formatting options, and returns it.

This function can be used externally to check how the provided *model_parameters* would be interpreted when provided to the `ModelLink` subclass. Its output can be used for the 'model_parameters' input argument.

Parameters **model_parameters** (*array_like, dict or str*) – Anything that can be converted to a dict that provides model parameters information.

Returns **par_dict** (*dict*) – Dict with the provided *model_parameters* converted to its full format.

prism.modellink.**test_subclass**(*subclass*, **args*, ***kwargs*)

Tests a provided `ModelLink` *subclass* by initializing it with the given *args* and *kwargs* and checking if all required methods can be properly called.

This function needs to be called by all MPI ranks.

Parameters

- **subclass** (`ModelLink` subclass) – The `ModelLink` subclass that requires testing.
- **args** (*positional arguments*) – Positional arguments that need to be provided to the constructor of the *subclass*.
- **kwargs** (*keyword arguments*) – Keyword arguments that need to be provided to the constructor of the *subclass*.

Returns **modellink_obj** (`ModelLink` object) – Instance of the provided *subclass* if all tests pass successfully. Specific exceptions are raised if a test fails.

Note: Depending on the complexity of the model wrapped in the given *subclass*, this function may take a while to execute.

CHAPTER 12

Utilities

Provides a collection of functions useful for using/mixing *PRISM* with other applications.

prism.utils.**get_hybrid_lnpost_fn**(*lnpost_fn*, *pipeline_obj*, *, *emul_i=None*, *unit_space=False*, *impl_prior=True*, *par_dict=False*)

Returns a function definition `hybrid_lnpost(par_set, *args, **kwargs)`.

This *hybrid_lnpost()* function can be used to calculate the natural logarithm of the posterior probability, which analyzes a given *par_set* first in the provided *pipeline_obj* at iteration *emul_i* and passes it to *lnpost_fn* if it is plausible.

This function needs to be called by all MPI ranks.

Parameters

- **lnpost_fn** (*function*) – Function definition that needs to be called if the provided *par_set* is plausible in iteration *emul_i* of *pipeline_obj*. The used call signature is `lnpost_fn(par_set, *args, **kwargs)`. All MPI ranks will call this function unless called within the *worker_mode* context manager.
- **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class that needs to be used for determining the validity of the proposed sampling step.

Other Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the last iteration of the emulator will be used.
- **unit_space** (*bool. Default: False*) – Bool determining whether or not *par_set* will be given in unit space.
- **impl_prior** (*bool. Default: True*) – Bool determining whether or not the *hybrid_lnpost()* function should use the implausibility values of a given *par_set* as an additional prior.
- **par_dict** (*bool. Default: False*) – Bool determining whether or not *par_set* will be an array_like (*False*) or a dict (*True*).

Returns **hybrid_lnpost** (*function*) – Definition of the function `hybrid_lnpost(par_set, *args, **kwargs)`.

See also:

get_walkers() Analyzes proposed *init_walkers* and returns valid *p0_walkers*.

worker_mode Special context manager within which all code is executed in worker mode.

Note: The input arguments *unit_space* and *par_dict* state in what form *par_set* will be provided to the *hybrid_lnpost()* function, such that it can be properly converted to the format used in `Pipeline`. The *par_set* that is passed to *lnpost_fn* is unchanged.

Warning: Calling this function factory will disable all regular logging in *pipeline_obj* (`do_logging` set to *False*), in order to avoid having the same message being logged every time *hybrid_lnpost()* is called.

prism.utils.**get_walkers**(*pipeline_obj*, *, *emul_i=None*, *init_walkers=None*, *req_n_walkers=None*, *unit_space=False*, *lnpost_fn=None*, ***kwargs*)

Analyzes proposed *init_walkers* and returns plausible *p0_walkers*.

Analyzes sample set *init_walkers* in the provided *pipeline_obj* at iteration *emul_i* and returns all samples that are plausible to be used as starting positions for MCMC walkers. The provided samples and returned walkers should be/are given in unit space if *unit_space* is *True*.

If *init_walkers* is *None*, returns `impl_sam` instead if it is available.

This function needs to be called by all MPI ranks.

Parameters **pipeline_obj** (`Pipeline` object) – The instance of the `Pipeline` class that needs to be used for determining the plausibility of the proposed starting positions.

Other Parameters

- **emul_i** (*int or None. Default: None*) – If int, number indicating the requested emulator iteration. If *None*, the last iteration of the emulator will be used.
- **init_walkers** (*2D array_like, dict, int or None. Default: None*) – Sample set of proposed initial MCMC walker positions. All plausible samples in *init_walkers* will be returned. If int, generate an LHD of provided size and return all plausible samples. If *None*, return `impl_sam` corresponding to iteration *emul_i* instead.
- **req_n_walkers** (*int or None. Default: None*) – The minimum required number of plausible starting positions that should be returned. If *None*, all plausible starting positions in *init_walkers* are returned instead.

  New in version 1.2.0.
- **unit_space** (*bool. Default: False*) – Bool determining whether or not the provided samples and returned walkers are given in unit space.
- **lnpost_fn** (*function or None. Default: None*) – If function, call `get_hybrid_lnpost_fn()` using *lnpost_fn* and the same values for *pipeline_obj*, *emul_i* and *unit_space*, and return the resulting function definition *hybrid_lnpost()*. Any additionally provided *kwargs* are also passed to it.

Returns

- **n_walkers** (*int*) – Number of returned MCMC walkers. Note that this number can be higher than *req_n_walkers* if not *None*.

- **p0_walkers** (2D `ndarray` object or dict) – Array containing plausible starting positions of valid MCMC walkers. If *init_walkers* was provided as a dict, *p0_walkers* will be a dict.
- **hybrid_lnpost** (function (if *lnpost_fn* is a function)) – The function returned by `get_hybrid_lnpost_fn()` using *lnpost_fn*, *pipeline_obj*, *emul_i*, *unit_space* and *kwargs* as the input values.

See also:

get_hybrid_lnpost_fn() Returns a function definition `hybrid_lnpost(par_set, *args, **kwargs)`.

worker_mode Special context manager within which all code is executed in worker mode.

Notes

If *init_walkers* is *None* and emulator iteration *emul_i* has not been analyzed yet, a `RequestError` will be raised.

If *req_n_walkers* is not *None*, a custom Metropolis-Hastings sampling algorithm is used to generate the required number of starting positions. All plausible samples in *init_walkers* are used as the start of every MCMC chain. Note that if the number of plausible samples in *init_walkers* is small, it is possible that the returned *p0_walkers* are not spread out properly over parameter space.

CHAPTER 13

Projection GUI

13.1 Classes

13.1.1 MainViewerWindow

class prism._gui.widgets.**MainViewerWindow**(*pipeline_obj*, **args*, ***kwargs*)

Defines the *MainViewerWindow* class for the Projection GUI.

This class provides the main window for the GUI and combines all other widgets; layouts; and elements together.

__init__(*pipeline_obj*, **args*, ***kwargs*)

Initialize an instance of the *MainViewerWindow* class.

Parameters **pipeline_obj** (*Pipeline* object) – Instance of the *Pipeline* class for which the GUI needs to be initialized.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the QMainWindow class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the QMainWindow class.

about()

Displays a small section with information about the GUI.

This function acts as a Qt slot.

all_call_proj_attr(*name*, **args*, ***kwargs*)

Calls the requested Projection attribute *name* using the provided *args* and *kwargs* on all ranks.

all_set_proj_attr(*name*, *value*)

Sets the requested Projection attribute *name* to *value* on all ranks.

api_reference()
Opens the API reference documentation of the GUI in a webbrowser.

This function acts as a Qt slot.

call_proj_attr(*name*, **args*, ***kwargs*)
Calls the requested `Projection` attribute *name* using the provided *args* and *kwargs* on the controller rank.

closeEvent(**args*, ***kwargs*)
Special `closeEvent()` event that automatically performs some clean-up operations before the main window closes.

create_menubar()
Creates the top-level menubar of the main window.

Other widgets can modify this menubar to add additional actions to it.

create_statusbar()
Creates the bottom-level statusbar of the main window, primarily used for displaying extended descriptions of actions.

get_default_dock_positions()
Returns the default positions of all dock widgets connected to the main window.

get_proj_attr(*name*)
Gets the value of the requested `Projection` attribute *name* on the controller rank.

init()
Sets up the main window after it has been initialized.

This function is mainly responsible for initializing all other widgets that are required to make the GUI work, and connecting them together.

set_default_dock_positions()
Sets the positions of all dock widgets connected to the main window to their default positions.

This function acts as a Qt slot.

set_proj_attr(*name*, *value*)
Sets the requested `Projection` attribute *name* to *value* on the controller rank.

show_pipeline_details_overview()
Creates and shows a dialog containing the output of the *details()* method for all emulator iterations.

This function acts as a Qt slot.

13.1.2 OverviewDockWidget

class prism._gui.widgets.**OverviewDockWidget**(*main_window_obj*, **args*, ***kwargs*)
Defines the *OverviewDockWidget* class for the Projection GUI.

This class provides the user with the ability to quickly create; draw; view; and save projection figures.

__init__(*main_window_obj*, **args*, ***kwargs*)
Initialize an instance of the *OverviewDockWidget* class.

Parameters **main_window_obj** (*MainViewerWindow* object) – Instance of the *MainViewerWindow* class that acts as the parent of this dock widget.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `QDockWidget` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `QDockWidget` class.

_create_projection_figure(*list_item*)
Creates the projection figure requested in the provided *list_item*, calculating its projection data.

This function is used iteratively by *create_projection_figures()*.

Parameters **list_item** (`QListWidgetItem` object) – The item that contains the requested projection figure.

_details_projection_figure(*list_item*)
Creates and shows a details dialog for the projection figure requested in the provided *list_item*.

Parameters **list_item** (`QListWidgetItem` object) – The item that contains the requested projection figure.

_draw_projection_figure(*list_item*)
Draws the projection figure requested in the provided *list_item*, creating its `Figure` instance.

This function is used iteratively by *draw_projection_figures()*.

Parameters **list_item** (`QListWidgetItem` object) – The item that contains the requested projection figure.

closeEvent(**args*, ***kwargs*)
Special `closeEvent()` event that automatically performs some clean-up operations before the overview dock widget closes.

close_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and closes their `Figure` objects.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

create_available_context_menu()
Creates the context (right-click) menu for the 'Available' overview list.

This menu contains all actions that are available for created (but not drawn) projection figures.

create_draw_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items*, calculates their projection data and draws their `Figure` objects afterward.

This function is basically a combination of *create_projection_figures()* and *draw_projection_figures()*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

create_draw_save_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items*, calculates their projection data, draws their `Figure` objects and saves them afterward.

This function is basically a combination of *create_projection_figures()*; *draw_projection_figures()* and *save_projection_figures()*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

create_drawn_context_menu()
Creates the context (right-click) menu for the 'Drawn' overview list.

This menu contains all actions that are available for drawn projection figures.

create_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and creates them, calculating their corresponding projection data.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

create_unavailable_context_menu()
Creates the context (right-click) menu for the 'Unavailable' overview list.

This menu contains all actions that are available for non-existing projection figures.

delete_projection_figures(*list_items=None*, *, *skip_warning=False*)
Retrieves the projection figures requested in the provided *list_items* and delete them, permanently removing their corresponding projection data.

This function acts as a Qt slot.

Other Parameters

- **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.
- **skip_warning** (*bool. Default: False*) – Whether or not to skip showing the warning asking the user if they are sure they want to permanently delete all items in *list_items*. If *True*, the answer is taken to be *True*.

details_available_projection_figure(*list_item=None*)
Retrieves the projection figure requested in the provided *list_item*, gathers its properties and shows a details dialog listing them.

This function is used for projections in the 'Available' list.

This function acts as a Qt slot.

Other Parameters **list_item** (`QListWidgetItem` object or None. Default: None) – The item that contains the requested projection figure. If *None*, the currently selected list item is used instead.

details_drawn_projection_figure(*list_item=None*)
Retrieves the projection figure requested in the provided *list_item*, gathers its properties and shows a details dialog listing them.

This function is used for projections in the 'Drawn' list.

This function acts as a Qt slot.

Other Parameters **list_item** (`QListWidgetItem` object or None. Default: None) – The item that contains the requested projection figure. If *None*, the currently selected list item is used instead.

draw_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and draws them, creating their `Figure` instances.

If the *auto_show* option is *True*, drawn figures will be shown automatically as well.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

draw_save_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items*, draws their `Figure` objects and saves them afterward.

This function is basically a combination of *draw_projection_figures()* and *save_projection_figures()*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

init()
Sets up the projection overview dock widget after it has been initialized.

This function is mainly responsible for creating the different overview lists and menus that allow the user to manipulate projection figures.

recreate_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and recreates them, permanently removing their corresponding projection data and recalculating it.

This function is basically a combination of *delete_projection_figures()* and *create_projection_figures()*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

redraw_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and redraws them, closing and recreating their `Figure` objects.

This function is basically a combination of *close_projection_figures()* and *draw_projection_figures()*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

save_as_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and saves their `Figure` objects, asking the user where to save each one.

This function basically calls *save_projection_figures()* with *choose* set to *True*.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

save_projection_figures(*list_items=None*, *, *choose=False*)
Retrieves the projection figures requested in the provided *list_items* and saves their `Figure` objects.

This function acts as a Qt slot.

Other Parameters

- **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.
- **choose** (*bool. Default: False*) – Whether or not the user is allowed to choose where the projection figure is saved to. If *False*, it uses the default filename as defined by *_Projection__get_fig_path()*.

show_available_context_menu()
Shows the 'Available' context menu, giving the user access to its actions.

This function acts as a Qt slot.

show_drawn_context_menu()
Shows the 'Drawn' context menu, giving the user access to its actions.

This function acts as a Qt slot.

show_projection_figures(*list_items=None*)
Retrieves the projection figures requested in the provided *list_items* and shows them in the projection viewing area.

This function acts as a Qt slot.

Other Parameters **list_items** (list of `QListWidgetItem` objects or None. Default: None) – The list of items that contains the requested projection figures. If *None*, all currently selected list items are used instead.

show_unavailable_context_menu()
Shows the 'Unavailable' context menu, giving the user access to its actions.

This function acts as a Qt slot.

use_progress_dialog(*label*, *func*, **iterables*)
Creates a progress dialog with the given *label*, and executes the requested *func* using the provided *iterables*.

Depending on the current settings, this function will either create a *ThreadedProgressDialog* object that allows the user to abort the operation (but is slower), or a static dialog that cannot be interrupted.

Parameters

- **label** (*str*) – The label that is used as the description of what operation is currently being executed.
- **func** (*function*) – The function that must be called iteratively using the arguments provided in *iterables*.

- **iterables** (*positional arguments*) – All iterables that must be used to call *func* with.

Returns **result** (*bool*) – Whether or not the operations ended successfully, which can be used by other functions to determine if it should continue.

13.1.3 ViewingAreaDockWidget

class prism._gui.widgets.**ViewingAreaDockWidget**(*main_window_obj*, **args*, ***kwargs*)
Defines the *ViewingAreaDockWidget* class for the Projection GUI.

This class provides the user with an MDI (Multiple Document Interface) area using the QMdiArea class. All drawn projection figures live in this area and can be interacted with.

__init__(*main_window_obj*, **args*, ***kwargs*)
Initialize an instance of the *ViewingAreaDockWidget* class.

Parameters **main_window_obj** (*MainViewerWindow* object) – Instance of the *MainViewerWindow* class that acts as the parent of this dock widget.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the QDockWidget class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the QDockWidget class.

closeEvent(**args*, ***kwargs*)
Special closeEvent() event that automatically performs some clean-up operations before the viewing area closes.

create_projection_toolbar()
Creates the top-level toolbar of the viewing area, primarily used for manipulating the area subwindows.

get_default_dock_positions()
Returns the default positions of all dock widgets connected to the viewing area.

init()
Sets up the projection viewing area dock widget after it has been initialized.

This function is mainly responsible for enabling the *OverviewDockWidget* to properly interact and control the projection figures that have been drawn.

save_view()
Saves the current view of the viewing area to file.

This function acts as a Qt slot.

set_default_dock_positions()
Sets the postions of all dock widgets connected to the viewing area to their default positions.

This function acts as a Qt slot.

13.2 Widgets

13.2.1 GUI Base Layout Classes

Provides a collection of custom QLayout base classes that allow for certain layouts to be standardized.

13.2.2 GUI Base Widget Classes

Provides a collection of custom `QWidget` base classes that allow for certain widgets to be standardized.

class prism._gui.widgets.base_widgets.**QW_QAction**(*parent*, *text*, *, *shortcut=None*, *tooltip=None*, *statustip=None*, *icon=None*, *triggered=None*, *role=None*)

Defines the *QW_QAction* class.

This class provides default settings and extra options for the `QAction` class.

__init__(*parent*, *text*, *, *shortcut=None*, *tooltip=None*, *statustip=None*, *icon=None*, *triggered=None*, *role=None*)

Initializes the *QW_QAction* class.

Parameters

- **parent** (`QWidget` object or None) – The parent widget for this dialog or *None* for no parent.
- **text** (*str*) – The label that this action must have.

Other Parameters

- **shortcut** (`QKeySequence` or None. Default: None) – The key sequence that must be set as the shortcut for this action. If *None*, no shortcut will be set.
- **tooltip** (*str or None. Default: None*) – The text that must be set as the tooltip for this action. If *None*, the tooltip is set to *text*. If *shortcut* is not *None*, the tooltip will also include the shortcut.
- **statustip** (*str or None. Default: None*) – The text that must be set as the statustip for this action. If *None*, the statustip is set to *tooltip*.
- **icon** (`QIcon` object or None. Default: None) – The icon that must be set as the icon for this action. If *None*, no icon will be set.
- **triggered** (*function or None. Default: None*) – The Qt slot function that must be called whenever this action is triggered. If *None*, no slot will connected to this action's signal.
- **role** (`MenuRole` object or None. Default: None) – The menu role that must be set as the role of this action. If *None*, it is set to `NoRole`.

setDetails(*, *shortcut=None*, *tooltip=None*, *statustip=None*)

Uses the provided *shortcut*; *tooltip*; and *statustip* to set the details of this action.

Parameters

- **shortcut** (`QKeySequence` or None. Default: None) – The key sequence that must be set as the shortcut for this action. If *None*, no shortcut will be set.
- **tooltip** (*str or None. Default: None*) – The text that must be set as the tooltip for this action. If *None*, the tooltip is set to *text*. If *shortcut* is not *None*, the tooltip will also include the shortcut.
- **statustip** (*str or None. Default: None*) – The text that must be set as the statustip for this action. If *None*, the statustip is set to *tooltip*.

class prism._gui.widgets.base_widgets.**QW_QComboBox**

Defines the *QW_QComboBox* class.

This class provides default settings and extra options for the `QComboBox` class.

class prism._gui.widgets.base_widgets.**QW_QEditableComboBox**(**args*, ***kwargs*)
Defines the *QW_QEditableComboBox* class.

This class makes the *QW_QComboBox* class editable.

class prism._gui.widgets.base_widgets.**QW_QLabel**
Defines the *QW_QLabel* class.

This class provides default settings and extra options for the QLabel class.

class prism._gui.widgets.base_widgets.**QW_QMenu**(*parent*, *title*)
Defines the *QW_QMenu* class.

This class provides default settings and extra options for the QMenu class.

class prism._gui.widgets.base_widgets.**QW_QToolBar**(*parent*, *window_title*)
Defines the *QW_QToolBar* class.

This class provides default settings and extra options for the QToolBar class.

13.2.3 GUI Widgets Core

Provides a collection of utility functions and the *BaseBox* class definition, which are core to the functioning of all GUI widgets.

class prism._gui.widgets.core.**BaseBox**
Defines the *BaseBox* base class.

This class is used by many custom QWidget classes as their base. It defines the modified signal, which is automatically connected to any widget that changes its state.

childEvent(*event*)
Special childEvent() event that automatically connects the default modified signal of any widget that becomes a child of this widget.

connect_box(*box*)
Connect the default modified signal of the provided *box* to this widget's modified signal.

get_box_value()
Obtain the value of this widget and return it.

set_box_value(*value*)
Set the value of this widget to *value*.

prism._gui.widgets.core.**get_box_value**(*box*)
Retrieves the value of the provided widget *box* and returns it.

prism._gui.widgets.core.**get_modified_box_signal**(*box*)
Retrieves the default modified signal of the provided widget *box* and returns it.

prism._gui.widgets.core.**set_box_value**(*box*, *value*)
Sets the value of the provided widget *box* to *value*.

13.2.4 GUI Widget Helpers

Provides a collection of custom QWidget subclasses that provide specific functionalities.

class prism._gui.widgets.helpers.**ExceptionDialog**(*parent*, *etype*, *value*, *tb*)
Defines the *ExceptionDialog* class for the Projection GUI.

This class takes a set of exception details and converts it into a format that can be shown using a dialog.

__init__ (*parent*, *etype*, *value*, *tb*)
Initialize an instance of the *ExceptionDialog* class.

Parameters

- **parent** (QWidget object or None) – The parent widget for this dialog or *None* for no parent.
- **etype** (Exception class) – The Exception class that is associated with this error.
- **value** (Exception object) – The Exception instance that is associated with this error.
- **tb** (*traceback object*) – The corresponding traceback object.

create_traceback_box()
Creates a special box for the exception dialog that contains the traceback information and returns it.

format_exception()
Formats the exception provided during initialization and returns it.

format_traceback()
Formats the traceback provided during initialization and returns it.

init()
Sets up the exception dialog after it has been initialized.

This function is mainly responsible for gathering all required information; formatting it; and drawing the dialog.

toggle_traceback_box()
Toggles the visibility of the traceback box and updates the dimensions of the exception dialog accordingly.

This function acts as a Qt slot.

update_size()
Updates the dimensions of the exception dialog depending on its current state (traceback box visibility).

class prism._gui.widgets.helpers.**OverviewListWidget**(**args*, ***kwargs*)
Defines the *OverviewListWidget* class.

This class defines the overview lists that are used by the *OverviewDockWidget* class.

__init__(**args*, ***kwargs*)
Initialize an instance of the *OverviewListWidget* class.

Parameters

- **args** (*positional arguments*) – The positional arguments that need to be passed to *init()*.
- **kwargs** (*keyword arguments*) – The keyword arguments that need to be passed to *init()*.

init(*, *hcubes_list*, *status_tip*, *context_menu*, *activated*)
Sets up the overview list after it has been initialized.

This function is mainly responsible for creating the list; adding the items to it; and setting some properties.

Parameters

- **hcubes_list** (*list of str*) – List of projection hypercube names that must be used to initialize this overview list with.
- **statustip** (*str*) – The statustip that will be displayed in the statusbar whenever this overview list is hovered.
- **context_menu** (*function*) – The function that must be called whenever the context menu is requested.

- **activated** (*function*) – The function that must be called whenever an item in this overview list is activated. This corresponds to the default action.

class prism._gui.widgets.helpers.**ThreadedProgressDialog**(*main_window_obj*, **args*, ***kwargs*)

Defines the *ThreadedProgressDialog* class for the Projection GUI.

This class provides a QProgressDialog class that automatically executes a provided operation on a separate thread, allowing for the user to interrupt it.

__call__()
Calls and returns the result of *open()*.

__init__(*main_window_obj*, **args*, ***kwargs*)
Initialize an instance of the *ThreadedProgressDialog* class.

Parameters

- **main_window_obj** (*MainViewerWindow* object) – Instance of the *MainViewerWindow* class that acts as the parent of progress dialog.
- **args** (*positional arguments*) – The positional arguments that need to be passed to *init()*.
- **kwargs** (*keyword arguments*) – The keyword arguments that need to be passed to *init()*.

init(*label*, *func*, **iterables*)
Sets up the progress dialog after it has been initialized.

This function is mainly responsible for preparing the dialog to be opened and the *func* function to be executed.

Parameters

- **label** (*str*) – The label that is used as the description of what operation is currently being executed.
- **func** (*function*) – The function that must be called iteratively using the arguments provided in *iterables*.
- **iterables** (*positional arguments*) – All iterables that must be used to call *func* with.

kill_threads()
Terminates all currently running threads besides the main thread (on all MPI ranks) and returns control to the main thread.

This function is the sole way to abort the operation.

This function acts as a Qt slot.

open()
Opens the progress dialog and starts the execution of the requested operation.

Returns **result** (*bool*) – Whether or not the operations ended successfully, which can be used by other functions to determine if it should continue.

raise_exception(*exception*)
Qt slot that raises a provided exception.

set_successful_finish()
Qt slot that marks the operation as 'successful'.

prism._gui.widgets.helpers.**show_exception_details**(*parent*, **args*, ***kwargs*)
Creates an instance of the *ExceptionDialog* class and shows it.

Parameters **parent** (QWidget object or None) – The parent widget for this dialog or *None* for no parent.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the *ExceptionDialog* class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the *ExceptionDialog* class.

13.2.5 GUI Preferences

GUI Options

Provides the main `QDialog` subclass that creates the preferences menu and keeps track of all internally saved options. The window used for the kwargs dicts is defined in *kwargs_dicts*.

class prism._gui.widgets.preferences.options.**OptionsDialog**(*main_window_obj*, **args*, ***kwargs*)

Defines the *OptionsDialog* class for the Projection GUI.

This class provides both the 'Preferences' dialog and the functions that are required to load; save; set; and change them.

__call__()
Qt slot that shows the options dialog in the center of the main window.

__init__(*main_window_obj*, **args*, ***kwargs*)
Initialize an instance of the *OptionsDialog* class.

Parameters **main_window_obj** (*MainViewerWindow* object) – Instance of the *MainViewerWindow* class that acts as the parent of this dialog.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `QDialog` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `QDialog` class.

closeEvent(**args*, ***kwargs*)
Special `closeEvent()` event that makes sure that all dialogs will be closed related to the options menu, and discards all changes made.

create_entry(*name*, *box*, *default*)
Creates a new *OptionsEntry* instance, using the provided *name*, *box* and *default*, and registers it in the options dialog.

Parameters

- **name** (*str*) – The name of this options entry.
- **box** (`QWidget` object) – The widget that will hold the values of this entry.
- **default** (*object*) – The default value of this entry.

create_group(*name*, *options_list*)
Creates a new option group with the given *name* and adds the options defined in *options_list* to it.

This function acts as a base function called by *create_group_* functions.

Parameters

- **name** (*str*) – The name of this option group.

- **options_list** (*list of str*) – A list containing the names of all options that need to be added to this group.

Returns **group** (`QGroupBox` object) – The created option group.

create_group_buttons(*window_layout*)
Creates the button box that is shown at the bottom of the options dialog and registers it in the provided *window_layout*.

create_group_fonts()
Creates the 'Fonts' group and returns it.

create_group_interface()
Creates the 'Interface' group and returns it.

create_group_proj_grid()
Creates the 'Projection grid' group and returns it.

create_group_proj_kwargs()
Creates the 'Projection keywords' group and returns it.

create_option_align()
Creates the 'align' option and returns it.

This option sets the value of the 'align' projection parameter.

create_option_auto_show()
Creates the 'auto_show' option and returns it.

This option sets whether the projection subwindows are automatically shown whenever created.

create_option_auto_tile()
Creates the 'auto_tile' option and returns it.

This option sets whether the projection subwindows are automatically tiled.

create_option_dpi()
Creates the 'dpi' option and returns it.

This option allows for the DPI used in the GUI to be modified.

create_option_full_impl_rng()
Creates the 'full_impl_rng' option and returns it.

This option sets the value of the 'full_impl_rng' projection parameter.

create_option_kwargs_dicts()
Creates the 'kwargs_dicts' option and returns it.

This option allows for the `KwargsDictDialog` to be shown to the user. This dialog is able to set the values of all 'XXX_kwargs' projection parameters.

create_option_progress_dialog()
Creates the 'progress_dialog' option and returns it.

This option sets whether a threaded progress dialog is used for some operations.

create_option_proj_depth()
Creates the 'proj_depth' option and returns it.

This option sets the value of the 'proj_depth' projection parameter.

create_option_proj_res()
Creates the 'proj_res' option and returns it.

This option sets the value of the 'proj_res' projection parameter.

create_option_show_cuts()
Creates the 'show_cuts' option and returns it.

This option sets the value of the 'show_cuts' projection parameter.

create_option_smooth()
Creates the 'smooth' option and returns it.

This option sets the value of the 'smooth' projection parameter.

create_option_text_fonts()
Creates the 'text_fonts' option and returns it.

This option allows for the fonts used in the GUI to be modified.

create_option_use_par_space()
Creates the 'use_par_space' option and returns it.

This option sets the value of the 'use_par_space' projection parameter.

create_tab(*name*, *groups_list*)
Creates a new options tab with the given *name* and adds the groups defined in *groups_list* to it.

This function acts as a base function called by *create_tab_* functions.

Parameters

- **name** (*str*) – The name of this options tab.
- **groups_list** (*list of str*) – A list containing the names of all option groups that need to be added to this tab.

Returns

- **tab** (`QWidget` object) – The created options tab.
- **name** (*str*) – The name of this options tab as provided with *name*. This variable is mainly returned such that it is easier to pass tab names between functions.

create_tab_appearance()
Creates the 'Appearance' tab and returns it.

create_tab_general()
Creates the 'General' tab and returns it.

disable_save_button()
Qt slot that disables the save button at the bottom of the options dialog. The save button is disabled whenever no changes have been made to any option entry.

discard_options()
Discards the current values of all option entries and sets them back to their saved values.

This function acts as a Qt slot.

enable_save_button()
Qt slot that enables the save button at the bottom of the options dialog. The save button is enabled if at least one change has been made to any option entry.

get_option(*name*)
Returns the value of the option entry associated with the given *name*.

init()
Sets up the options dialog after it has been initialized.

This function is mainly responsible for initializing all option entries that the GUI has, and creating a database for them. It also creates the layout of the options dialog.

reset_options()
Resets the saved and current values of all option entries back to their default values.

This function acts as a Qt slot.

save_options()
Saves all current values of all option entries.

Option entries that affect projection parameters are automatically modified as well.

This function acts as a Qt slot.

class prism._gui.widgets.preferences.options.**OptionsEntry**(*parent*, *name*, *box*, *default*)
Defines the *OptionsEntry* class.

This class is used as a container for making option entries in the *OptionsDialog* class.

__init__(*parent*, *name*, *box*, *default*)
Initialize an instance of the *OptionsEntry* class.

Parameters

- **parent** (QWidget object) – The widget to use as the parent of this entry.
- **name** (*str*) – The name of this options entry.
- **box** (QWidget object) – The widget that will hold the values of this entry.
- **default** (*object*) – The default value of this entry.

discard_value()
Qt slot that discards the current value and sets it back to its saved value.

init()
Sets up the options entry after it has been initialized.

This function is mainly responsible for making sure that the current and saved values of this entry are set to its default value.

reset_value()
Qt slot that resets the current value of this options entry to its default value.

save_value()
Qt slot that saves the current value of this options entry.

box
The widget box that contains this options entry.

Type QWidget object

default
The default value of this options entry.

Type object

name
The name of this options entry.

Type str

value
The currently saved value of this options entry.

Type object

GUI Kwargs Dicts Options

Provides a custom `QDialog` subclass that allows for the projection keyword argument dicts to be modified properly in the Projection GUI preferences.

class prism._gui.widgets.preferences.kwargs_dicts.**KwargsDictBoxLayout**(*options_dialog_obj*, **args*, ***kwargs*)

Defines the *KwargsDictBoxLayout* class for the preferences window.

This class provides the options entry box that gives the user access to a separate window, where the various different keyword dicts can be modified.

__init__(*options_dialog_obj*, **args*, ***kwargs*)

Initialize an instance of the *KwargsDictBoxLayout* class.

Parameters **options_dialog_obj** (`OptionsDialog` object) – Instance of the `OptionsDialog` class that acts as the parent of the *KwargsDictDialog* this layout creates.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `QHBoxLayout` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `QHBoxLayout` class.

add_dict(**args*, ***kwargs*)

Adds a new kwargs dict to the box layout, by calling the *add_page()* method using the provided *args* and *kwargs*.

init()

Sets up the box layout after it has been initialized.

This function is mainly responsible for initializing the *KwargsDictDialog* class and binding it.

class prism._gui.widgets.preferences.kwargs_dicts.**KwargsDictDialog**(*options_dialog_obj*, **args*, ***kwargs*)

Defines the *KwargsDictDialog* class for the preferences window.

This class provides the 'Projection keyword argument dicts' dialog, which allows for the various different kwargs dicts to be modified by the user.

__call__()

Qt slot that shows the kwargs dict dialog in the center of the preferences window.

__init__(*options_dialog_obj*, **args*, ***kwargs*)

Initialize an instance of the *KwargsDictDialog* class.

Parameters **options_dialog_obj** (`OptionsDialog` object) – Instance of the `OptionsDialog` class that acts as the parent of this dialog.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `QDialog` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `QDialog` class.

add_page(*name*, *option_key*, *tooltip*, **args*, ***kwargs*)

Initializes a new *KwargsDictDialogPage* object with name *name* and adds it to this dialog.

Parameters

- **name** (*str*) – The name that this kwargs dict page will have.
- **option_key** (*str*) – The name of the options entry that this page will create. The value of *option_key* must correspond to the name the associated dict has in the `project()` method.
- **tooltip** (*str*) – The tooltip that must be used for this kwargs dict page.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `KwargsDictDialogPage` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `KwargsDictDialogPage` class.

init()
Sets up the kwargs dict dialog after it has been initialized.

This function is mainly responsible for setting up the layout of the dialog, and making sure that new kwargs dict pages can be added.

class prism._gui.widgets.preferences.kwargs_dicts.**KwargsDictDialogPage**(*kwargs_dict_dialog_obj*, *name*, *std_entries*, *banned_entries*, **args*, ***kwargs*)
Defines the `KwargsDictDialogPage` class for the kwargs dict dialog.

This class provides the tab/page in the kwargs dict dialog where the items of the associated kwargs dict can be viewed and modified by the user.

__init__(*kwargs_dict_dialog_obj*, *name*, *std_entries*, *banned_entries*, **args*, ***kwargs*)
Initialize an instance of the `KwargsDictDialogPage` class.

Parameters

- **kwargs_dict_dialog_obj** (`KwargsDictDialog` object) – Instance of the `KwargsDictDialog` class that initialized this kwargs dict page.
- **name** (*str*) – The name of this kwargs dict page.
- **std_entries** (*list of str*) – A list of all standard entry types that this kwargs dict should accept.
- **banned_entries** (*list of str*) – A list of all entry types that this kwargs dict should not accept. Usually, these entry types are used by *PRISM* and therefore should not be modified by the user.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the `BaseBox` class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the `BaseBox` class.

add_editable_entry()
Adds a new editable entry to the kwargs dict page, which allows for the user to edit the contents of the kwargs dict.

This function acts as a Qt slot.

create_type_alpha()
Creates the 'alpha' entry and returns it.

create_type_cmap()
Creates the 'cmap' entry and returns it.

create_type_color()
Creates the 'color' entry and returns it.

create_type_dpi()
Creates the 'dpi' entry and returns it.

create_type_fh_arrowlength()
Creates the 'fh_arrowlength' entry and returns it.

create_type_fh_arrowwidth()
Creates the 'fh_arrowwidth' entry and returns it.

create_type_ft_arrowlength()
Creates the 'ft_arrowlength' entry and returns it.

create_type_ft_arrowwidth()
Creates the 'ft_arrowwidth' entry and returns it.

create_type_linestyle()
Creates the 'linestyle' entry and returns it.

create_type_linewidth()
Creates the 'linewidth' entry and returns it.

create_type_marker()
Creates the 'marker' entry and returns it.

create_type_markersize()
Creates the 'markersize' entry and returns it.

create_type_rel_xpos()
Creates the 'rel_xpos' entry and returns it.

create_type_rel_ypos()
Creates the 'rel_ypos' entry and returns it.

create_type_scale(*axis*)
Base function for creating the entry types 'xscale' and 'yscale'.

create_type_xscale()
Creates the 'xscale' entry and returns it.

create_type_yscale()
Creates the 'yscale' entry and returns it.

entry_type_selected(*entry_type*, *kwargs_box*)
Qt slot that modifies the field box associated with the provided *kwargs_box* to given *entry_type*.

Parameters

- **entry_type** (*str*) – The entry type that is requested for the field box.
- **kwargs_box** (*QW_QEditableComboBox* object) – The combobox that is used for setting the entry type of this entry.

get_box_value()
Returns the current value of the kwargs dict page.

Returns **page_dict** (*dict*) – A dict containing all valid entries that are currently on this kwargs dict page. Any invalid entries (banned or empty ones) are ignored.

init()
Sets up the kwargs dict page after it has been initialized.

This function is mainly responsibe for creating the layout of the page; determining what entry types are available; and preparing for the user to add entries.

remove_editable_entry(*kwargs_box*)
Removes the editable entry associated with the provided *kwargs_box*.

This function acts as a Qt slot.

Parameters **kwargs_box** (*QW_QEditableComboBox* object) – The combobox that is used for setting the entry type of this entry.

set_box_value(*page_dict*)
Sets the current value of the kwargs dict page to *page_dict*.

Parameters **page_dict** (*dict*) – A dict containing all entries that this kwargs dict page must have. Current entries that are also in *page_dict* will be reused, otherwise they are deleted.

GUI Custom Option Boxes

Provides a collection of `QWidget` subclasses to be used as custom option entry boxes in the `OptionsDialog` class or `KwargsDictDialogPage` class.

class prism._gui.widgets.preferences.custom_boxes.**ColorBox**(**args*, ***kwargs*)
Defines the *ColorBox* class.

This class is used for making the 'color' entry in the `KwargsDictDialogPage` class.

__init__(**args*, ***kwargs*)
Initialize an instance of the *ColorBox* class.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the *BaseBox* class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the *BaseBox* class.

static convert_to_mpl_color(*qcolor*)
Converts a provided `QColor` object *color* to a matplotlib color.

Parameters **qcolor** (`QColor` object) – The instance of the `QColor` class must be converted to a matplotlib color.

Returns **color** (*str*) – The corresponding matplotlib color. The returned *color* is always written in HEX.

static convert_to_qcolor(*color*)
Converts a provided matplotlib color *color* to a `QColor` object.

Parameters **color** (*str*) – The matplotlib color that must be converted. If *color* is a float string, an error will be raised, as Qt5 does not accept those.

Returns **qcolor** (`QColor` object) – The instance of the `QColor` class that corresponds to the provided *color*.

create_color_combobox()
Creates a combobox that holds all default colors accepted by matplotlib and returns it.

create_color_label()
Creates a special label that shows the currently selected or hovered color, and returns it.

static create_color_pixmap(*color*, *size*)
Creates a `QPixmap` object consisting of the given *color* with the provided *size*.

Parameters

- **color** (*str*) – The matplotlib color that must be used for the pixmap.
- **size** (*tuple*) – The width and height dimension values of the pixmap to be created.

Returns **pixmap** (`QPixmap` object) – The instance of the `QPixmap` class that was created from the provided *color* and *size*.

get_box_value()
Returns the current (valid) color value of the color combobox.

Returns **color** (*str*) – The current valid matplotlib color value.

init()
Sets up the color box entry after it has been initialized.

This function is mainly responsible for creating the color wheel and color label, that allow the user to quickly cycle through different color options.

set_box_value(*value*)
Sets the current (default) color value to *value*.

Parameters **value** (*str*) – The matplotlib color value that must be set for this colorbox.

set_color(*color*)
Sets the current color to the provided *color*, and updates the entry in the combobox and the label accordingly.

This function acts as a Qt slot.

Parameters **color** (*str*) – The color that needs to be used as the current color. The provided *color* can be any string that is accepted as a color by matplotlib. If *color* is invalid, it is set to the current default color instead.

set_color_label(*color*)
Sets the current color label to the provided *color*.

This function acts as a Qt slot.

Parameters **color** (*str*) – The color that needs to be used as the current color label. The provided *color* can be any string that is accepted as a color by matplotlib. If *color* is invalid, it is set to the current default color instead.

Returns **default_flag** (*bool*) – Whether or not the color label is currently set to the default color. This happens when *color* is an invalid color.

show_colorpicker()
Shows the colorwheel picker dialog to the user, allowing for any color option to be selected.

This function acts as a Qt slot.

class prism._gui.widgets.preferences.custom_boxes.**ColorMapBox**(**args*, ***kwargs*)
Defines the *ColorMapBox* class.

This class is used for making the 'cmap' entry in the `KwargsDictDialogPage` class.

__init__(**args*, ***kwargs*)
Initialize an instance of the *ColorMapBox* class.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the *BaseBox* class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the *BaseBox* class.

cmap_selected(*cmap*)
Qt slot that checks a provided *cmap* and shows an error message if *cmap* is a terrible colormap.

static create_cmap_icon(*cmap*, *size*)
Creates a `QIcon` object of the given *cmap* with the provided *size*.

Parameters

- **cmap** (`Colormap` object or str) – The colormap for which an icon needs to be created.
- **size** (*tuple*) – A tuple containing the width and height dimension values of the icon to be created.

Returns **icon** (`QIcon` object) – The instance of the `QIcon` class that was created from the provided *cmap* and *size*.

get_box_value()
Returns the current colormap of the colormap box.

Returns **cmap** (`Colormap` object) – The currently selected colormap.

set_box_value(*cmap*)
Sets the current colormap to *cmap*.

Parameters **cmap** (`Colormap` object) – The colormap that must be used for this colormap box.

class prism._gui.widgets.preferences.custom_boxes.**DefaultBox**(**args*, ***kwargs*)
Defines the *DefaultBox* class.

This class is used for making a non-standard entry in the `KwargsDictDialogPage` class. It currently supports inputs of type bool; float; int; and str.

__init__(**args*, ***kwargs*)
Initialize an instance of the *DefaultBox* class.

Other Parameters

- **args** (*positional arguments*) – The positional arguments that must be passed to the constructor of the *BaseBox* class.
- **kwargs** (*keyword arguments*) – The keyword arguments that must be passed to the constructor of the *BaseBox* class.

create_field_box(*value_type*)
Creates a field box for the provided type *value_type* and replaces the current field box with it.

This function acts as a Qt slot.

Parameters **value_type** (*{'bool'; 'float'; 'int'; 'str'}*) – The string that defines what type of field box is requested.

create_type_bool()
Creates the field box for values of type 'bool' and returns it.

create_type_float()
Creates the field box for values of type 'float' and returns it.

create_type_int()
Creates the field box for values of type 'int' and returns it.

create_type_str()
Creates the field box for values of type 'str' and returns it.

get_box_value()
Returns the current value of the field box.

Returns **value** (*bool, float, int or str*) – The current value of this default box.

init()
Sets up the non-standard default box entry after it has been initialized.

This function is mainly responsible for creating the type combobox and allowing for different field boxes to be used for different value types.

set_box_value(*value*)
Sets the value type to *type(value)* and the field value to *value*.

Parameters **value** (*bool, float, int or str*) – The value to use for this default box. The type of *value* determines which field box must be used.

13.3 Functions

prism._gui.**start_gui**(*pipeline_obj*)
Creates an instance of `QApplication` or retrieves it if one already exists, and starts *Crystal*, *PRISM*'s Projection GUI.

Crystal provides an interactive way of creating projection figures, as opposed to the static and linear method provided by *project()*. It is made to make it easier to create; view; compare; and analyze large numbers of projection figures. All options available in the *project()* method can also be accessed through *Crystal*.

As with all *Pipeline* user methods, this function must be called by all MPI ranks.

New in version 1.2.0.

Parameters **pipeline_obj** (*Pipeline* object) – The instance of the *Pipeline* class for which *Crystal* must be initialized.

Returns **main_window_obj** (*MainViewerWindow* object) – The instance of the *MainViewerWindow* class that was created for drawing *Crystal*. Can be used for debugging purposes.

Note: This function can also be accessed through the *crystal()* method.

CHAPTER 14

Internal

Contains a collection of support classes/functions for the *PRISM* package.

exception prism._internal.**FeatureWarning**
Generic warning raised for experimental features in *PRISM*.

General purpose warning class, raised whenever a feature is used that should be considered experimental. Its behavior and API are subject to change, or the entire feature may be removed without a deprecation period.

__weakref__
list of weak references to the object (if defined)

exception prism._internal.**RequestError**
Generic exception raised for invalid action requests in the *PRISM* pipeline.

General purpose exception class, raised whenever a requested action cannot be executed due to it not being allowed or possible in the current state of the *Pipeline* instance.

__weakref__
list of weak references to the object (if defined)

exception prism._internal.**RequestWarning**
Generic warning raised for (future) action requests in the *PRISM* pipeline that may not be useful.

General purpose warning class, raised whenever a requested action may not produce appropriate or expected results due to the current state of the *Pipeline* instance. It is also raised if an obtained result can lead to such an action in the future.

__weakref__
list of weak references to the object (if defined)

class prism._internal.**CFilter**(*MPI_rank*)
Custom Filter class that only allows the controller rank to log messages to the logfile. Calls from worker ranks are ignored.

class prism._internal.**PRISM_Logger**(**args*, ***kwargs*)
Special Logger class that allows for special filters to be set more easily.

class prism._internal.**RFilter**(*MPI_rank*)
Custom `Filter` class that prepends the world rank of the MPI process that calls it to the logging message. If the size of `MPI.COMM_WORLD` is 1, this filter does nothing.

prism._internal.**check_compatibility**(*emul_version*)
Checks if the provided *emul_version* is compatible with the current version of *PRISM*. Raises a *RequestError* if *False* and indicates which version of *PRISM* still supports the provided *emul_version*.

prism._internal.**check_vals**(*values*, *name*, **args*)
Checks if all values in provided input argument *values* with *name* meet all criteria given in *args*. If no criteria are given, it is checked if all values are finite. Returns *values* (0 or 1 in case of bool) if *True* and raises a `ValueError` or `TypeError` if *False*.

Parameters

- **values** (*array_like of {bool; complex; float; int; str}*) – The values to be checked against all given criteria in *args*. It must be possible to convert *values* to a `ndarray` object.
- **name** (*str*) – The name of the input argument, which is used in the error message if a criterion is not met.
- **args** (*positional arguments in {'bool'; 'complex'; 'float'; 'int'; 'neg'; 'nneg'; 'normal'; 'npos'; 'nzero'; 'pos'; 'str'}*) – Sequence of strings determining the criteria that *values* must meet. If *args* is empty, it is checked if *values* are finite.

Returns **return_values** (*array_like of {complex; float; int; str}*) – If *args* contained 'bool', returns 0s or 1s. Else, returns *values*.

Notes

If *values* contains integers, but *args* contains 'float', *return_values* will be cast as float.

---

prism._internal.**get_bibtex**()
Prints a string that gives the BibTeX entry for citing the *PRISM* paper (Van der Velden et al. 2019, ApJS, 242, 22).

prism._internal.**get_PRISM_File**(*prism_hdf5_file*)
Returns a class definition `PRISM_File(mode, emul_s=None, **kwargs)`.

This class definition is a specialized version of the `File` class with the filename automatically set to *prism_hdf5_file* and added logging to the constructor and destructor methods.

Parameters **prism_hdf5_file** (*str*) – Absolute path to the master HDF5-file that is used in a *Pipeline* instance.

Returns **PRISM_File** (*class*) – Definition of the class `PRISM_File(mode, emul_s=None, **kwargs)`.

prism._internal.**get_formatter**()
Returns a `Formatter` object containing the default logging formatting.

prism._internal.**get_handler**(*filename*)
Returns a `Handler` object containing the default logging handling settings.

prism._internal.**get_info**()
Prints a string that gives an overview of all information relevant to the *PRISM* package distribution.

prism._internal.**getCLogger**(*name=None*)
Creates a *PRISM_Logger* instance with *name*, adds the *CFilter* to it and returns it.

prism._internal.**getLogger**(*name=None*, *filters=None*)
Creates a *PRISM_Logger* instance with *name* and adds the provided *filters* to it. The returned *PRISM_Logger* instance is a child of the base *PRISM_Logger* created with *set_base_logger()*, but has its name changed (such that the parent name does not show up in the log-file).

Other Parameters

- **name** (*str or None. Default: None*) – The name of the *PRISM_Logger* instance to create. If *None*, initialize the base *PRISM_Logger* instead.
- **filters** (*list of str or None. Default: None*) – List of strings naming the filters that must be applied to the created *PRISM_Logger* instance. If *None* or the *PRISM_Logger* instance already existed, no filters will be applied.

Returns **logger** (*PRISM_Logger* object) – The created *PRISM_Logger* instance.

prism._internal.**getRLogger**(*name=None*)
Creates a *PRISM_Logger* instance with *name*, adds the *RFilter* to it and returns it.

prism._internal.**move_logger**(*working_dir*)
Moves the base *PRISM_Logger* from the current working directory to the given *working_dir*, and then restarts it again.

Parameters **working_dir** (*str*) – String containing the directory the log-file needs to be moved to.

prism._internal.**np_array**(*obj*, **args*, ***kwargs*)
Returns `np.array(obj, *args, copy=False, **kwargs)`.

prism._internal.**set_base_logger**(*filename=None*)
Initializes the base *PRISM_Logger*, from which all other *PRISM_Logger* instances are derived.

Other Parameters **filename** (*str or None. Default: None*) – String containing the name of the log-file that is opened. If *None*, a new log-file will be created.

CHAPTER 15

Acknowledgements

Special thanks to Alan Duffy, Darren Croton, Simon Mutch and Manodeep Sinha for providing many valuable suggestions and constructive feedback points. Huge thanks to James Josephides for making the *PRISM* logo.

Python Module Index

_

prism._gui.widgets.base_layouts, 113
prism._gui.widgets.base_widgets, 113
prism._gui.widgets.core, 115
prism._gui.widgets.helpers, 115
prism._gui.widgets.preferences.custom_boxes, 125
prism._gui.widgets.preferences.kwargs_dicts, 121
prism._gui.widgets.preferences.options, 118
prism._internal, 128

u

prism.utils, 102

Index

Symbols

_ModelLink__set_model_data() (*prism.modellink.ModelLink method*), 94
_ModelLink__set_model_parameters() (*prism.modellink.ModelLink method*), 94
_Projection__analyze_proj_hcube() (*prism.Pipeline method*), 57
_Projection__draw_2D_proj_fig() (*prism.Pipeline method*), 57
_Projection__draw_3D_proj_fig() (*prism.Pipeline method*), 57
_Projection__draw_estimate_arrowline() (*prism.Pipeline method*), 57
_Projection__get_default_input_arguments() (*prism.Pipeline method*), 58
_Projection__get_default_parameters() (*prism.Pipeline method*), 58
_Projection__get_fig_path() (*prism.Pipeline method*), 58
_Projection__get_grid_idx() (*prism.Pipeline method*), 58
_Projection__get_hcube_name() (*prism.Pipeline method*), 58
_Projection__get_proj_data() (*prism.Pipeline method*), 58
_Projection__get_proj_hcube() (*prism.Pipeline method*), 58
_Projection__get_proj_space() (*prism.Pipeline method*), 59
_Projection__get_req_hcubes() (*prism.Pipeline method*), 59
_Projection__prepare_projections() (*prism.Pipeline method*), 59
_Projection__process_input_arguments() (*prism.Pipeline method*), 59
_Projection__read_proj_space() (*prism.Pipeline method*), 60
_Projection__save_data() (*prism.Pipeline method*), 60
_Projection__set_parameters() (*prism.Pipeline method*), 60
__call__() (*prism.Pipeline method*), 60
__call__() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
__call__() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictD method*), 122
__call__() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 118
__enter__() (*prism._pipeline.WorkerMode method*), 76
__exit__() (*prism._pipeline.WorkerMode method*), 76
__init__() (*prism.Pipeline method*), 60
__init__() (*prism._gui.widgets.MainViewerWindow method*), 107
__init__() (*prism._gui.widgets.OverviewDockWidget method*), 108
__init__() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
__init__() (*prism._gui.widgets.base_widgets.QW_QAction method*), 114
__init__() (*prism._gui.widgets.helpers.ExceptionDialog method*), 115
__init__() (*prism._gui.widgets.helpers.OverviewListWidget method*), 116
__init__() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
__init__() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 125
__init__() (*prism._gui.widgets.preferences.custom_boxes.ColorMapBo method*), 126
__init__() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 127
__init__() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictB method*), 122
__init__() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictD method*), 122
__init__() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictD method*), 123

__init__() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 118
__init__() (*prism._gui.widgets.preferences.options.OptionsEntry method*), 121
__init__() (*prism._pipeline.WorkerMode method*), 76
__init__() (*prism.emulator.Emulator method*), 79
__init__() (*prism.modellink.GaussianLink method*), 93
__init__() (*prism.modellink.ModelLink method*), 95
__init__() (*prism.modellink.PolyLink method*), 101
__weakref__ (*prism._internal.FeatureWarning attribute*), 129
__weakref__ (*prism._internal.RequestError attribute*), 129
__weakref__ (*prism._internal.RequestWarning attribute*), 129
__weakref__ (*prism._pipeline.WorkerMode attribute*), 78
__weakref__ (*prism.emulator.Emulator attribute*), 88
__weakref__ (*prism.modellink.ModelLink attribute*), 99
_assign_data_idx() (*prism.emulator.Emulator method*), 79
_assign_emul_s() (*prism.emulator.Emulator method*), 80
_call_model() (*prism.Pipeline method*), 61
_check_future_compat() (*prism.emulator.Emulator method*), 81
_check_md_var() (*prism.modellink.ModelLink method*), 96
_check_mod_set() (*prism.modellink.ModelLink method*), 96
_check_sam_set() (*prism.modellink.ModelLink method*), 96
_cleanup_emul_files() (*prism.emulator.Emulator method*), 81
_compile_code_snippets() (*prism.Pipeline method*), 61
_construct_iteration() (*prism.emulator.Emulator method*), 81
_create_new_emulator() (*prism.emulator.Emulator method*), 81
_create_projection_figure() (*prism._gui.widgets.OverviewDockWidget method*), 109
_default_model_data (*prism.modellink.ModelLink attribute*), 99
_default_model_parameters (*prism.modellink.ModelLink attribute*), 99
_details_projection_figure() (*prism._gui.widgets.OverviewDockWidget method*), 109
_do_impl_check() (*prism.Pipeline method*), 61
_do_regression() (*prism.emulator.Emulator method*), 81
_draw_projection_figure() (*prism._gui.widgets.OverviewDockWidget method*), 109
_evaluate() (*prism.emulator.Emulator method*), 82
_evaluate_model() (*prism.Pipeline method*), 61
_evaluate_sam_set() (*prism.Pipeline method*), 62
_get_active_par() (*prism.emulator.Emulator method*), 82
_get_adj_exp() (*prism.emulator.Emulator method*), 82
_get_adj_var() (*prism.emulator.Emulator method*), 83
_get_backup_path() (*prism.modellink.ModelLink method*), 96
_get_cov() (*prism.emulator.Emulator method*), 83
_get_cov_matrix() (*prism.emulator.Emulator method*), 83
_get_data_idx_flat() (*prism.emulator.Emulator method*), 84
_get_default_parameters() (*prism.Pipeline method*), 63
_get_default_parameters() (*prism.emulator.Emulator method*), 84
_get_emul_i() (*prism.emulator.Emulator method*), 84
_get_emul_space() (*prism.emulator.Emulator method*), 84
_get_eval_sam_set() (*prism.Pipeline method*), 63
_get_exp_dot_term() (*prism.emulator.Emulator method*), 84
_get_ext_real_set() (*prism.Pipeline method*), 63
_get_f_impl() (*prism.Pipeline method*), 63
_get_impl_space() (*prism.Pipeline method*), 63
_get_inv_matrix() (*prism.emulator.Emulator method*), 84
_get_iteration_data() (*prism.Pipeline method*), 63
_get_md_var() (*prism.Pipeline method*), 64
_get_mock_data() (*prism.Pipeline method*), 64
_get_model_par_seq() (*prism.modellink.ModelLink method*), 97
_get_n_eval_sam() (*prism.Pipeline method*), 64
_get_paths() (*prism.Pipeline method*), 64
_get_poly_term_str() (*prism.emulator.Emulator method*), 85
_get_prior_exp() (*prism.emulator.Emulator method*), 85
_get_regr_cov() (*prism.emulator.Emulator method*), 85
_get_rsdl_var() (*prism.emulator.Emulator method*), 85
_get_sam_dict() (*prism.modellink.ModelLink*

method), 97
_get_sam_space() (*prism.modellink.ModelLink method*), 97
_get_uni_impl() (*prism.Pipeline method*), 65
_load_data() (*prism.Pipeline method*), 65
_load_data() (*prism.emulator.Emulator method*), 86
_load_emulator() (*prism.emulator.Emulator method*), 86
_make_backup() (*prism.modellink.ModelLink method*), 97
_make_call() (*prism.Pipeline method*), 65
_make_call_workers() (*prism.Pipeline method*), 66
_multi_call_model() (*prism.Pipeline method*), 66
_prepare_new_iteration() (*prism.emulator.Emulator method*), 86
_process_call() (*prism._pipeline.WorkerMode static method*), 76
_process_call_str() (*prism._pipeline.WorkerMode static method*), 76
_read_backup() (*prism.modellink.ModelLink method*), 98
_read_data_idx() (*prism.emulator.Emulator method*), 86
_read_parameters() (*prism.Pipeline method*), 66
_retrieve_parameters() (*prism.emulator.Emulator method*), 86
_save_data() (*prism.Pipeline method*), 66
_save_data() (*prism.emulator.Emulator method*), 86
_save_statistics() (*prism.Pipeline method*), 67
_set_impl_par() (*prism.Pipeline method*), 67
_set_mock_data() (*prism.emulator.Emulator method*), 87
_set_modellink() (*prism.emulator.Emulator method*), 87
_set_parameters() (*prism.Pipeline method*), 67
_set_parameters() (*prism.emulator.Emulator method*), 87
_set_sam_set_data() (*prism.emulator.Emulator method*), 87
_to_par_space() (*prism.modellink.ModelLink method*), 98
_to_unit_space() (*prism.modellink.ModelLink method*), 98
_write_data_idx() (*prism.emulator.Emulator method*), 87
2+D model, **43**
2D model, **43**

A

about() (*prism._gui.widgets.MainViewerWindow method*), 107
act_rsdl_var (*prism.emulator.Emulator attribute*), 88
Active emulator system, **41**
Active parameters, **41**
active_emul_s (*prism.emulator.Emulator attribute*), 88
active_par (*prism.emulator.Emulator attribute*), 88
active_par_data (*prism.emulator.Emulator attribute*), 88
add_dict() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictB method*), 122
add_editable_entry() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialog method*), 123
add_page() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictD method*), 122
Adjusted expectation, **41**
Adjusted values, **41**
Adjusted variance, **41**
Adjustment term, **41**
all_call_proj_attr() (*prism._gui.widgets.MainViewerWindow method*), 107
all_set_proj_attr() (*prism._gui.widgets.MainViewerWindow method*), 107
Analysis, **41**
Analyze, **41**
analyze() (*prism.Pipeline method*), 67
api_reference() (*prism._gui.widgets.MainViewerWindow method*), 107

B

base_eval_sam (*prism.Pipeline attribute*), 73
BaseBox (*class in prism._gui.widgets.core*), 115
BLA, **41**
box (*prism._gui.widgets.preferences.options.OptionsEntry attribute*), 121

C

call_model() (*prism.modellink.ModelLink method*), 98
call_proj_attr() (*prism._gui.widgets.MainViewerWindow method*), 108
call_type (*prism.modellink.ModelLink attribute*), 100
ccheck (*prism.emulator.Emulator attribute*), 88
CFilter (*class in prism._internal*), 129
check_compatibility() (*in module prism._internal*), 130
check_vals() (*in module prism._internal*), 130
childEvent() (*prism._gui.widgets.core.BaseBox method*), 115
close_projection_figures() (*prism._gui.widgets.OverviewDockWidget*

method), 109
closeEvent() (*prism._gui.widgets.MainViewerWindow method*), 108
closeEvent() (*prism._gui.widgets.OverviewDockWidget method*), 109
closeEvent() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 118
closeEvent() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
cmap_selected() (*prism._gui.widgets.preferences.custom_boxes.ColorMapBox method*), 127
code_objects (*prism.Pipeline attribute*), 73
ColorBox (*class in prism._gui.widgets.preferences.custom_boxes*), 125
ColorMapBox (*class in prism._gui.widgets.preferences.custom_boxes*), 126
comm (*prism.Pipeline attribute*), 73
connect_box() (*prism._gui.widgets.core.BaseBox method*), 115
Construct, **41**
construct() (*prism.Pipeline method*), 67
Construction, **41**
Construction check, **41**
Controller, **41**
Controller rank, **42**
convert_data() (*in module prism.modellink*), 102
convert_parameters() (*in module prism.modellink*), 102
convert_to_mpl_color() (*prism._gui.widgets.preferences.custom_boxes.ColorBox static method*), 125
convert_to_qcolor() (*prism._gui.widgets.preferences.custom_boxes.ColorBox static method*), 125
cov_mat_inv (*prism.emulator.Emulator attribute*), 88
Covariance matrix, **42**
Covariance vector, **42**
create_available_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 109
create_cmap_icon() (*prism._gui.widgets.preferences.custom_boxes.ColorMapBox static method*), 127
create_color_combobox() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 125
create_color_label() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 126
create_color_pixmap() (*prism._gui.widgets.preferences.custom_boxes.ColorBox static method*), 126
create_draw_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 109
create_draw_save_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 109
create_drawn_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 110
create_entry() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 118
create_field_box() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 127
create_group() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 118
create_group_buttons() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_group_fonts() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_group_interface() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_group_proj_grid() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_group_proj_kwargs() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_menubar() (*prism._gui.widgets.MainViewerWindow method*), 108
create_option_align() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_auto_show() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_auto_tile() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_dpi() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_full_impl_rng() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_kwargs_dicts() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_progress_dialog() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_proj_depth()

(*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_proj_res() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_show_cuts() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 119
create_option_smooth() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_option_text_fonts() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_option_use_par_space() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 110
create_projection_toolbar() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
create_statusbar() (*prism._gui.widgets.MainViewerWindow method*), 108
create_tab() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_tab_appearance() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_tab_general() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
create_traceback_box() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116
create_type_alpha() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_bool() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 127
create_type_cmap() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_color() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_dpi() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_fh_arrowlength() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_fh_arrowwidth() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_float() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 127
create_type_ft_arrowlength() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_ft_arrowwidth() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_int() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 128
create_type_linestyle() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_linewidth() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_marker() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_markersize() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_rel_xpos() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_rel_ypos() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_scale() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_str() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 128
create_type_xscale() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_type_yscale() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 124
create_unavailable_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 110
criterion (*prism.Pipeline attribute*), 74
crop() (*prism.Pipeline method*), 68
cut_idx (*prism.Pipeline attribute*), 74

D

Data error, **42**
Data identifier, **42**
Data point, **42**
Data point identifier, **42**
Data space, **42**
Data value, **42**
Data value space, **42**
data_err (*prism.modellink.ModelLink attribute*), 100
data_idx (*prism.modellink.ModelLink attribute*), 100
data_idx_to_core (*prism.emulator.Emulator attribute*), 88
data_spc (*prism.modellink.ModelLink attribute*), 100
data_val (*prism.modellink.ModelLink attribute*), 100
default (*prism._gui.widgets.preferences.options.OptionsEntry attribute*), 121
DefaultBox (*class in prism._gui.widgets.preferences.custom_boxes*), 127
delete_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 110
details() (*prism.Pipeline method*), 68
details_available_projection_figure() (*prism._gui.widgets.OverviewDockWidget method*), 110
details_drawn_projection_figure() (*prism._gui.widgets.OverviewDockWidget method*), 110
disable_save_button() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
discard_options() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
discard_value() (*prism._gui.widgets.preferences.options.OptionsEntry method*), 121
do_active_anal (*prism.Pipeline attribute*), 74
do_logging (*prism.Pipeline attribute*), 74
draw_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 111
draw_save_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 111

E

emul_i (*prism.emulator.Emulator attribute*), 88
emul_load (*prism.emulator.Emulator attribute*), 88
emul_s (*prism.emulator.Emulator attribute*), 88
emul_s_to_core (*prism.emulator.Emulator attribute*), 88
emul_space (*prism.emulator.Emulator attribute*), 89
emul_type (*prism.emulator.Emulator attribute*), 89
Emulation method, **42**
Emulator, **42**
Emulator (*class in prism.emulator*), 79
emulator (*prism.Pipeline attribute*), 74
Emulator evaluation samples, **42**
Emulator iteration, **42**
Emulator system, **42**
Emulator type, **42**
enable_save_button() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
entry_type_selected() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialog method*), 124
Evaluate, **42**
evaluate() (*prism.Pipeline method*), 69
Evaluation, **42**
Evaluation set, **45**
ExceptionDialog (*class in prism._gui.widgets.helpers*), 115
exp_dot_term (*prism.emulator.Emulator attribute*), 89
External model realization set, **42**

F

f_infl (*prism.emulator.Emulator attribute*), 89
FeatureWarning, 129
File (*prism.Pipeline attribute*), 73
format_exception() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116
format_traceback() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116
freeze_active_par (*prism.Pipeline attribute*), 74
Frozen active parameters, **42**
Frozen parameters, **42**
FSLR, **42**

G

Gaussian correlation length, **42**
Gaussian sigma, **42**
GaussianLink (*class in prism.modellink*), 93
get_bibtex() (*in module prism._internal*), 130
get_box_value() (*in module prism._gui.widgets.core*), 115
get_box_value() (*prism._gui.widgets.core.BaseBox method*), 115
get_box_value() (*prism._gui.widgets.preferences.custom_boxes.Color method*), 126
get_box_value() (*prism._gui.widgets.preferences.custom_boxes.Color method*), 127
get_box_value() (*prism._gui.widgets.preferences.custom_boxes.Defau method*), 128

get_box_value() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictBoxLayout method*), 124
get_default_dock_positions() (*prism._gui.widgets.MainViewerWindow method*), 108
get_default_dock_positions() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
get_default_model_data() (*prism.modellink.ModelLink method*), 99
get_default_model_parameters() (*prism.modellink.ModelLink method*), 99
get_formatter() (*in module prism._internal*), 130
get_handler() (*in module prism._internal*), 130
get_hybrid_lnpost_fn() (*in module prism.utils*), 103
get_info() (*in module prism._internal*), 130
get_md_var() (*prism.modellink.ModelLink method*), 99
get_modified_box_signal() (*in module prism._gui.widgets.core*), 115
get_option() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
get_PRISM_File() (*in module prism._internal*), 130
get_proj_attr() (*prism._gui.widgets.MainViewerWindow method*), 108
get_str_repr() (*prism.modellink.ModelLink method*), 99
get_walkers() (*in module prism.utils*), 104
getCLogger() (*in module prism._internal*), 130
getLogger() (*in module prism._internal*), 130
getRLogger() (*in module prism._internal*), 131

H

HDF5, **42**
hdf5_file (*prism.Pipeline attribute*), 74
Hybrid sampling, **43**

I

impl_cut (*prism.Pipeline attribute*), 74
impl_sam (*prism.Pipeline attribute*), 74
Implausibility check, **43**
Implausibility cut-off check, **43**
Implausibility cut-offs, **43**
Implausibility value, **43**
Implausibility wildcard, **43**
init() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116
init() (*prism._gui.widgets.helpers.OverviewListWidget method*), 116
init() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
init() (*prism._gui.widgets.MainViewerWindow method*), 108
init() (*prism._gui.widgets.OverviewDockWidget method*), 111
init() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 126
init() (*prism._gui.widgets.preferences.custom_boxes.DefaultBox method*), 128
init() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictBoxLayout method*), 122
init() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialog method*), 123
init() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 125
init() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
init() (*prism._gui.widgets.preferences.options.OptionsEntry method*), 121
init() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
Inverted covariance matrix, **42**
is_controller (*prism.Pipeline attribute*), 74
is_worker (*prism.Pipeline attribute*), 74
Iterations, **42**

K

kill_threads() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
KwargsDictBoxLayout (*class in prism._gui.widgets.preferences.kwargs_dicts*), 122
KwargsDictDialog (*class in prism._gui.widgets.preferences.kwargs_dicts*), 122
KwargsDictDialogPage (*class in prism._gui.widgets.preferences.kwargs_dicts*), 123

L

l_corr (*prism.emulator.Emulator attribute*), 89
LHD, **43**
listen_for_calls() (*prism._pipeline.WorkerMode method*), 77

M

MainViewerWindow (*class in prism._gui.widgets*), 107
make_call() (*prism._pipeline.WorkerMode static method*), 77
make_call_workers() (*prism._pipeline.WorkerMode static method*), 77
Master file, **43**
Master HDF5 file, **43**
MCMC, **43**
method (*prism.emulator.Emulator attribute*), 89

Mock data, **43**
mod_set (*prism.emulator.Emulator attribute*), 89
Model, **43**
Model data, **43**
Model discrepancy variance, **43**
Model evaluation samples, **43**
Model output, **43**
Model outputs, **43**
Model parameter, **43**
Model parameters, **44**
Model realization samples, **44**
Model realization set, **44**
Model realizations, **44**
model_data (*prism.modellink.ModelLink attribute*), 100
model_parameters (*prism.modellink.ModelLink attribute*), 100
ModelLink, **43**
ModelLink (*class in prism.modellink*), 93
modellink (*prism.Pipeline attribute*), 74
ModelLink subclass, **43**
move_logger() (*in module prism._internal*), 131
MPI, **44**
MPI rank, **44**
MPI_call (*prism.modellink.ModelLink attribute*), 99
MSE, **44**
multi_call (*prism.modellink.ModelLink attribute*), 100

N

n_cross_val (*prism.emulator.Emulator attribute*), 89
n_data (*prism.modellink.ModelLink attribute*), 100
n_emul_s (*prism.emulator.Emulator attribute*), 89
n_emul_s_tot (*prism.emulator.Emulator attribute*), 89
n_eval_sam (*prism.Pipeline attribute*), 75
n_gaussians (*prism.modellink.GaussianLink attribute*), 93
n_impl_sam (*prism.Pipeline attribute*), 75
n_par (*prism.modellink.ModelLink attribute*), 100
n_sam (*prism.emulator.Emulator attribute*), 89
n_sam_init (*prism.Pipeline attribute*), 75
name (*prism._gui.widgets.preferences.options.OptionsEntry attribute*), 121
name (*prism.modellink.ModelLink attribute*), 100
nD model, **43**
np_array() (*in module prism._internal*), 131

O

OLS, **44**
open() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
OptionsDialog (*class in prism._gui.widgets.preferences.options*), 118
OptionsEntry (*class in prism._gui.widgets.preferences.options*), 121
order (*prism.modellink.PolyLink attribute*), 101
OverviewDockWidget (*class in prism._gui.widgets*), 108
OverviewListWidget (*class in prism._gui.widgets.helpers*), 116

P

par_est (*prism.modellink.ModelLink attribute*), 101
par_name (*prism.modellink.ModelLink attribute*), 101
par_rng (*prism.modellink.ModelLink attribute*), 101
Parameter set, **44**
pas_rsdl_var (*prism.emulator.Emulator attribute*), 90
Passive parameters, **44**
Pipeline, **44**
Pipeline (*class in prism*), 57
Plausible region, **44**
Plausible samples, **44**
poly_coef (*prism.emulator.Emulator attribute*), 90
poly_coef_cov (*prism.emulator.Emulator attribute*), 90
poly_idx (*prism.emulator.Emulator attribute*), 90
poly_order (*prism.emulator.Emulator attribute*), 90
poly_powers (*prism.emulator.Emulator attribute*), 90
poly_terms (*prism.emulator.Emulator attribute*), 90
PolyLink (*class in prism.modellink*), 101
Polynomial order, **44**
pot_active_par (*prism.Pipeline attribute*), 75
Potentially active parameters, **44**
Prior covariance, **44**
Prior expectation, **44**
Prior variance, **44**
PRISM, **44**
PRISM Pipeline, **44**
prism._gui.widgets.base_layouts (*module*), 113
prism._gui.widgets.base_widgets (*module*), 113
prism._gui.widgets.core (*module*), 115
prism._gui.widgets.helpers (*module*), 115
prism._gui.widgets.preferences.custom_boxes (*module*), 125
prism._gui.widgets.preferences.kwargs_dicts (*module*), 121
prism._gui.widgets.preferences.options (*module*), 118
prism._internal (*module*), 128
prism.utils (*module*), 102

prism_dict (*prism.Pipeline attribute*), 75
PRISM_Logger (*class in prism._internal*), 129
prism_version (*prism.emulator.Emulator attribute*), 90
proj_depth (*prism.Pipeline attribute*), 75
proj_res (*prism.Pipeline attribute*), 75
Project, **44**
project() (*prism.Pipeline method*), 70
Projection, **44**
Projection figure, **44**
Python Enhancement Proposals
    PEP 377, 31

Q

QW_QAction (*class in prism._gui.widgets.base_widgets*), 114
QW_QComboBox (*class in prism._gui.widgets.base_widgets*), 114
QW_QEditableComboBox (*class in prism._gui.widgets.base_widgets*), 114
QW_QLabel (*class in prism._gui.widgets.base_widgets*), 115
QW_QMenu (*class in prism._gui.widgets.base_widgets*), 115
QW_QToolBar (*class in prism._gui.widgets.base_widgets*), 115

R

raise_exception() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
rank (*prism.Pipeline attribute*), 75
recreate_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 111
redraw_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 111
Regression, **44**
Regression covariances, **44**
remove_editable_entry() (*prism._gui.widgets.preferences.kwargs_dicts.KwargsDictDialogPage method*), 125
RequestError, 129
RequestWarning, 129
reset_options() (*prism._gui.widgets.preferences.options.OptionsDialog method*), 120
reset_value() (*prism._gui.widgets.preferences.options.OptionsEntry method*), 121
Residual variance, **44**
RFilter (*class in prism._internal*), 129
Root directory, **45**
root_dir (*prism.Pipeline attribute*), 75
rsdl_var (*prism.emulator.Emulator attribute*), 90
run() (*prism.Pipeline method*), 73

S

sam_set (*prism.emulator.Emulator attribute*), 90
Sample, **44**
Sample set, **45**
save_as_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 111
save_options() (*prism._gui.widgets.preferences.options.OptionsDialo method*), 121
save_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 112
save_value() (*prism._gui.widgets.preferences.options.OptionsEntry method*), 121
save_view() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
set_base_logger() (*in module prism._internal*), 131
set_box_value() (*in module prism._gui.widgets.core*), 115
set_box_value() (*prism._gui.widgets.core.BaseBox method*), 115
set_box_value() (*prism._gui.widgets.preferences.custom_boxes.Colo method*), 126
set_box_value() (*prism._gui.widgets.preferences.custom_boxes.Colo method*), 127
set_box_value() (*prism._gui.widgets.preferences.custom_boxes.Defa method*), 128
set_box_value() (*prism._gui.widgets.preferences.kwargs_dicts.Kwarg method*), 125
set_color() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 126
set_color_label() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 126
set_default_dock_positions() (*prism._gui.widgets.MainViewerWindow method*), 108
set_default_dock_positions() (*prism._gui.widgets.ViewingAreaDockWidget method*), 113
set_proj_attr() (*prism._gui.widgets.MainViewerWindow method*), 108
set_successful_finish() (*prism._gui.widgets.helpers.ThreadedProgressDialog method*), 117
setDetails() (*prism._gui.widgets.base_widgets.QW_QAction method*), 114
show_available_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 112

show_colorpicker() (*prism._gui.widgets.preferences.custom_boxes.ColorBox method*), 126
show_drawn_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 112
show_exception_details() (*in module prism._gui.widgets.helpers*), 117
show_pipeline_details_overview() (*prism._gui.widgets.MainViewerWindow method*), 108
show_projection_figures() (*prism._gui.widgets.OverviewDockWidget method*), 112
show_unavailable_context_menu() (*prism._gui.widgets.OverviewDockWidget method*), 112
sigma (*prism.emulator.Emulator attribute*), 90
SineWaveLink (*class in prism.modellink*), 101
single_call (*prism.modellink.ModelLink attribute*), 101
size (*prism.Pipeline attribute*), 75
start_gui() (*in module prism._gui*), 128
start_gui() (*prism.Pipeline method*), 73

T

test_subclass() (*in module prism.modellink*), 102
ThreadedProgressDialog (*class in prism._gui.widgets.helpers*), 117
toggle_traceback_box() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116

U

Univariate implausibility value, **43**
update_size() (*prism._gui.widgets.helpers.ExceptionDialog method*), 116
use_mock (*prism.emulator.Emulator attribute*), 91
use_progress_dialog() (*prism._gui.widgets.OverviewDockWidget method*), 112
use_regr_cov (*prism.emulator.Emulator attribute*), 91

V

value (*prism._gui.widgets.preferences.options.OptionsEntry attribute*), 121
ViewingAreaDockWidget (*class in prism._gui.widgets*), 113

W

Worker, **45**
Worker mode, **45**
Worker rank, **45**
worker_mode (*prism.Pipeline attribute*), 75
WorkerMode (*class in prism._pipeline*), 76
Working directory, **45**
working_dir (*prism.Pipeline attribute*), 76