PRISM documentation

Ellert van der Velden

Mar 25, 2020

User Documentation

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:

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.

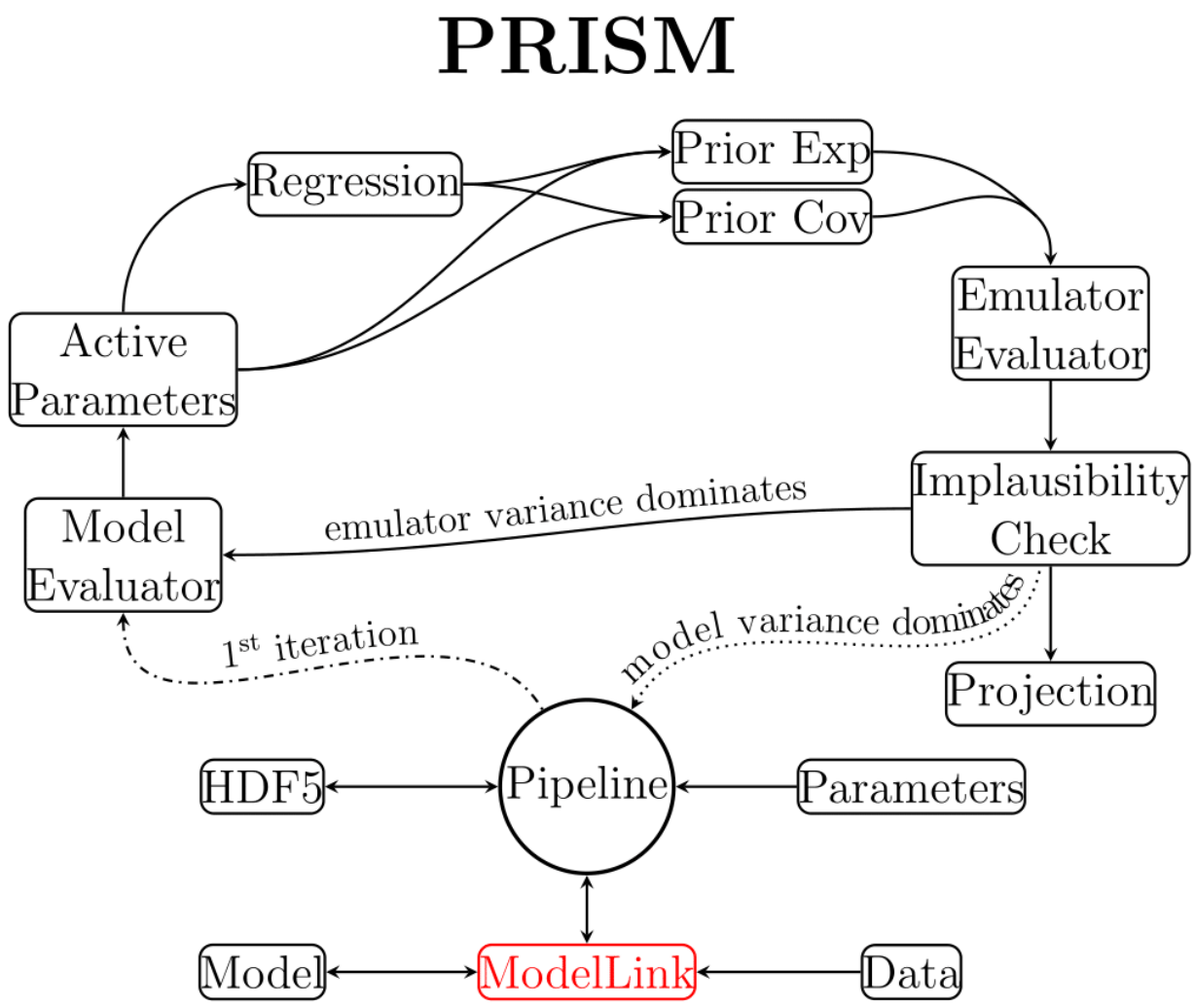

Fig. 3.1: The structure of the *PRISM* pipeline.

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.

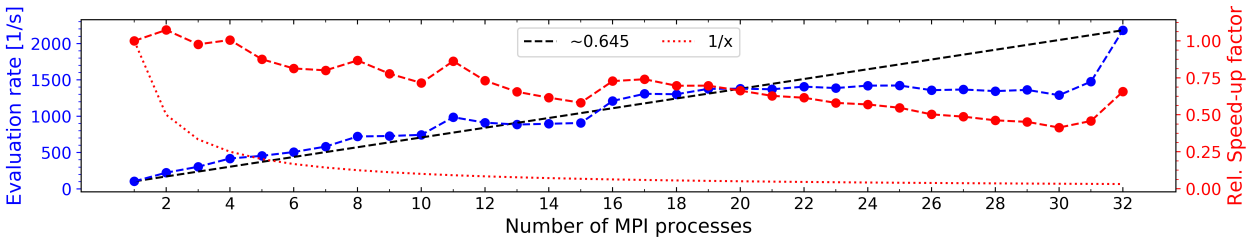

Fig. 3.2: Figure showing the MPI scaling of *PRISM* using the emulator of a simple Gaussian model with 32 emulator systems. The tests involved analyzing a Latin-Hypercube design of $3 \cdot 10^6$ samples in the emulator, determining the average evaluation rate and executing this a total of 20 times using 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):

```
# 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.

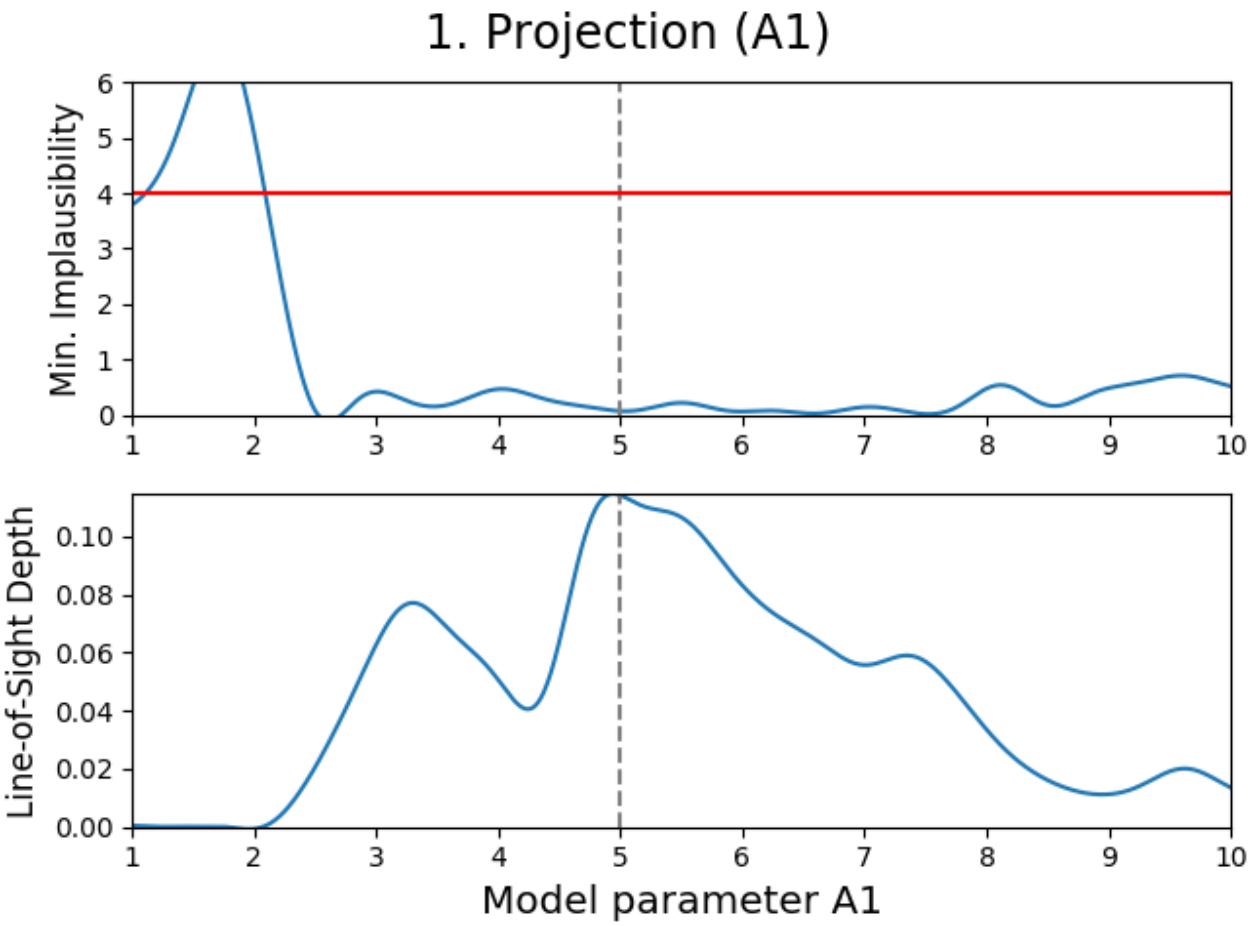

Fig. 5.1: 2D projection figure of model parameter A_1. The vertical dashed line shows the parameter estimate of A_1, whereas the horizontal red line shows the first implausibility cut-off value.

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

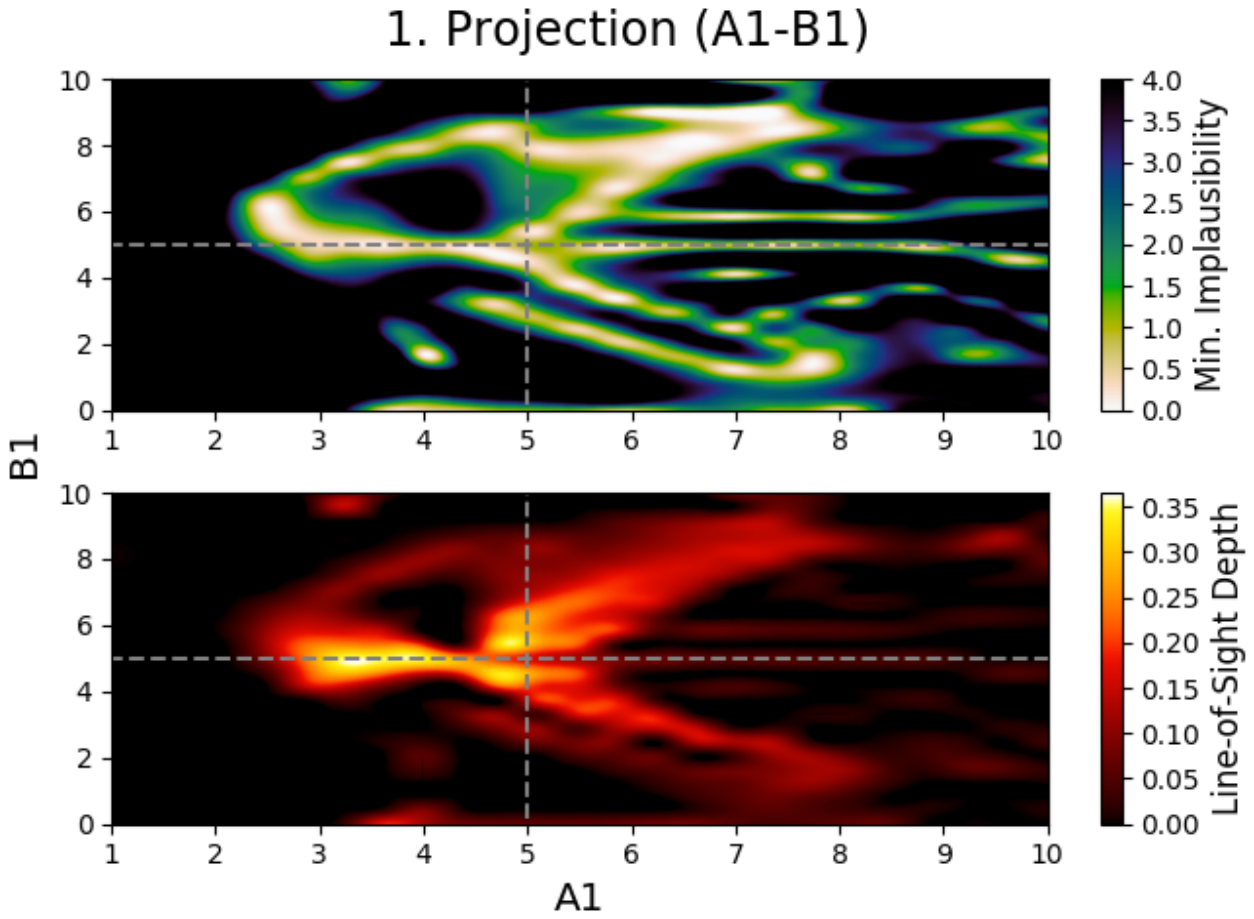

Fig. 5.2: 3D projection figure of model parameters A_1 and B_1. The dashed lines show the estimates of both parameters.

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.

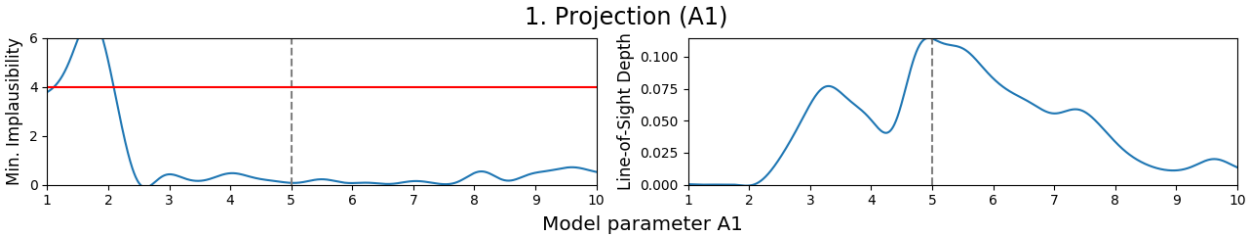

Fig. 5.3: 2D projection figure of model parameter A_1 with the *'row'* alignment.

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).

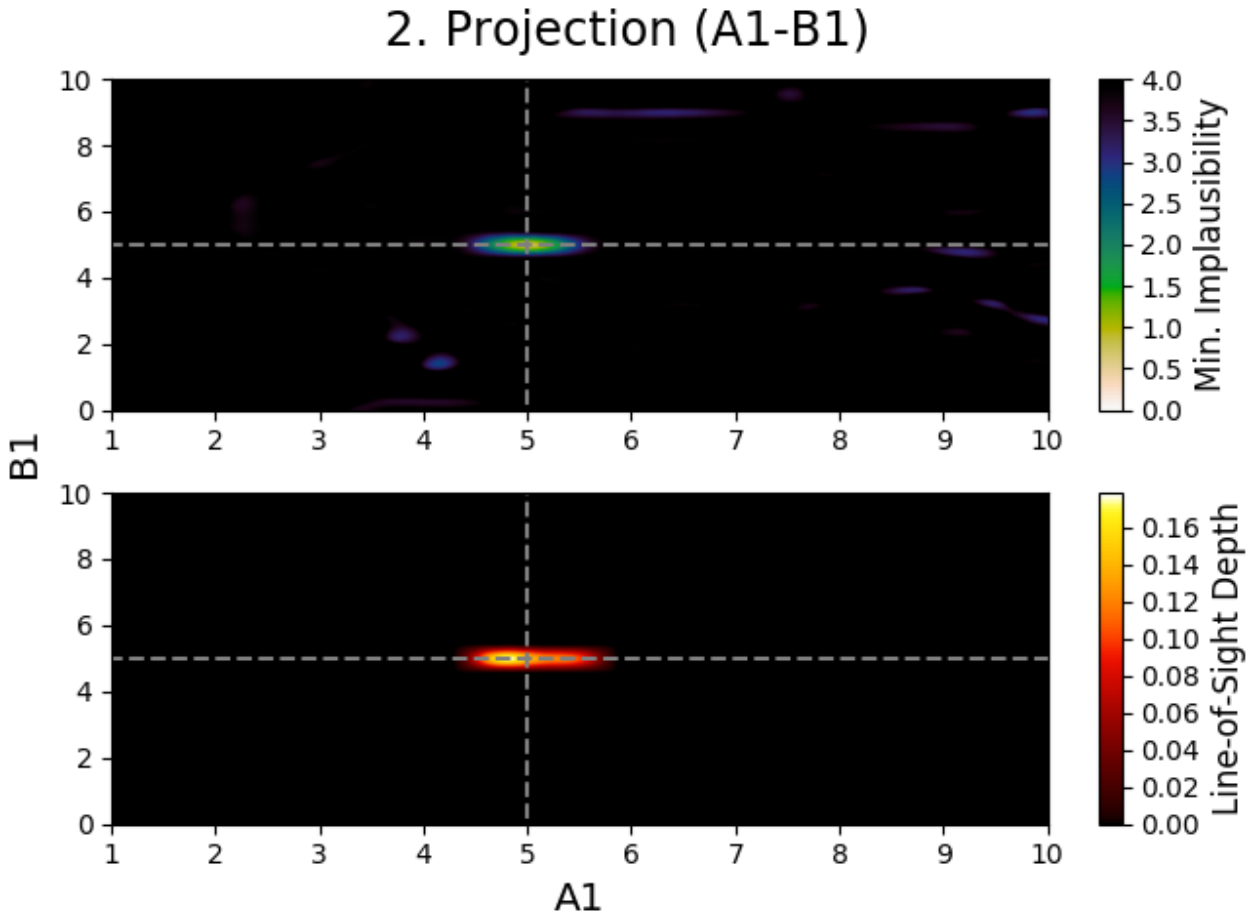

Fig. 5.4: Non-smoothed 3D projection figure of model parameters A_1 and B_1.

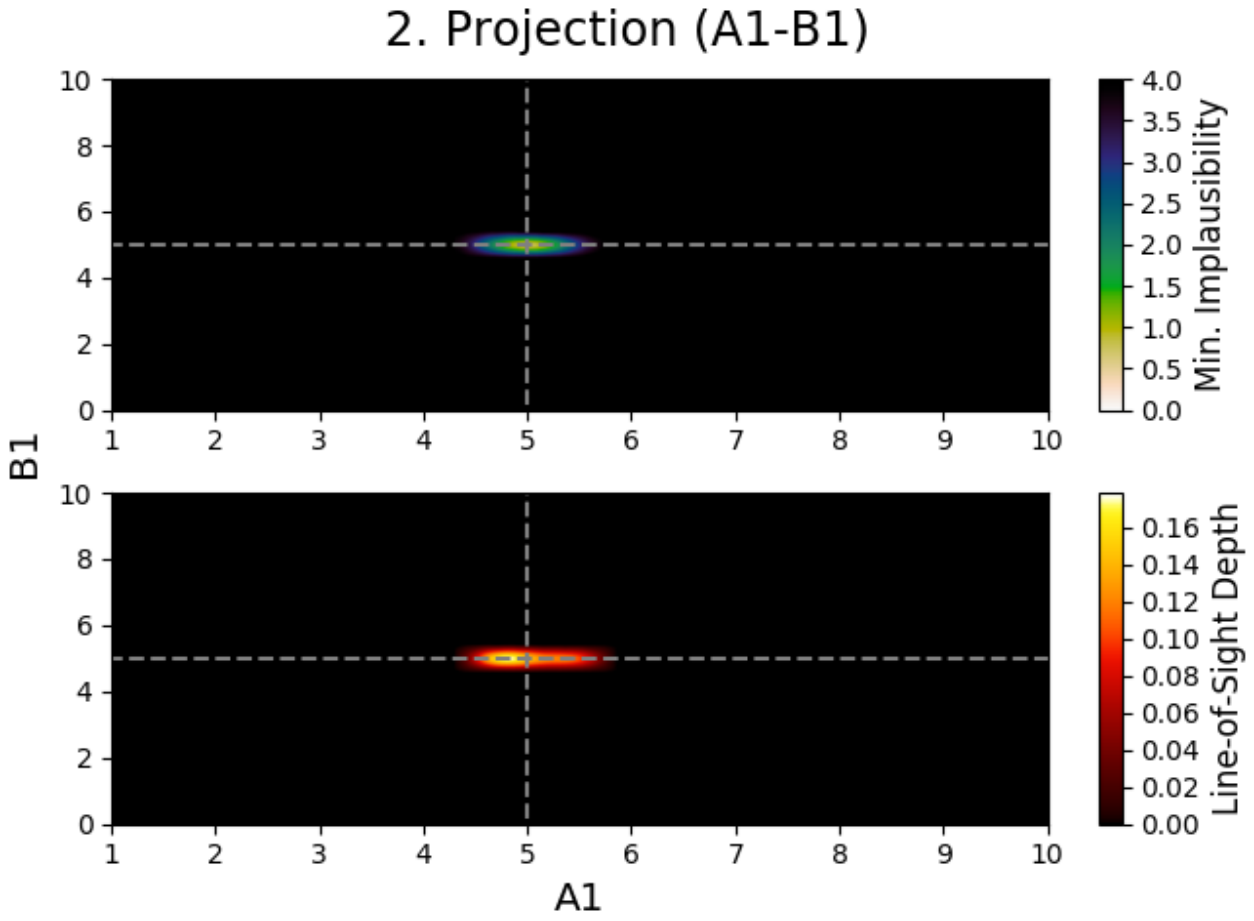

Fig. 5.5: Smoothed 3D projection figure of model parameters A_1 and B_1.

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.

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 seven 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. And, finally, 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.

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
```

(continues on next page)

(continued from previous page)

```
# 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
    sigma_2 = 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
```

(continues on next page)

(continued from previous page)

```
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}

# 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],
```

(continues on next page)

(continued from previous page)

```
            ['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;
- `'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 (5+): 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.
- `'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.

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:

```
@ARTICLE{2019ApJS..242...22V,
    author = {{van der Velden}, E. and {Duffy}, A.~R. and {Croton}, D. and
        {Mutch}, S.~J. and {Sinha}, M.},
    title = "{Model dispersion with PRISM; an alternative to MCMC for rapid analysis␣
↪of models}",
    journal = {\apjs},
    keywords = {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 = {http://adsabs.harvard.edu/abs/2019ApJS..242...22V},
    adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

8.1 License

8.2 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

CHAPTER 10

Emulator

10.1 Classes

10.1.1 Emulator

CHAPTER 11

ModelLink

11.1 Classes

11.1.1 GaussianLink

11.1.2 ModelLink

11.1.3 SineWaveLink

11.2 Utilities

CHAPTER 12

Projection GUI

12.1 Classes

12.1.1 MainViewerWindow

12.1.2 OverviewDockWidget

12.1.3 ViewingAreaDockWidget

12.2 Widgets

12.2.1 GUI Preferences

12.3 Functions

CHAPTER 13

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.

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

U

W