## CONTENTS

1 Getting Started  
1.1 Installation .......................................................... 3  
1.2 Requirements ......................................................... 3  
1.3 Loading data .......................................................... 3  
1.4 Training the emulator ............................................... 5  
1.5 Making predictions ............................................... 5  

2 Analysis tools  
2.1 Cross checks ......................................................... 8  
2.2 Sweeps Of Parameter Space ....................................... 9  
2.3 Model Parameters Features ...................................... 10  
2.4 Checking Hyperparameters ..................................... 12  

3 Emulator Options  
3.1 Mean models ......................................................... 16  
3.2 Emulating Bin-by-Bin ............................................. 16  
3.3 1D Emulation ....................................................... 17  

4 Interaction with SWIFT  
4.1 Experimental Design ............................................. 19  
4.2 Loading SWIFT data ............................................. 20  

5 Comparing With Data  
5.1 Penalty Functions .................................................. 25  
5.2 Defining New Penalty Functions ................................ 26  

6 API Documentation  
6.1 swiftemulator package .......................................... 29  

7 Indices and tables  

8 Indices and tables  

Python Module Index  

Index
The SWIFT-Emulator is a python toolkit for using Gaussian Process machine learning to produce synthetic simulation data by interpolating between base outputs. It excels at creating synthetic scaling relations across large swathes of model parameter space, as it was created to model galaxy scaling relations as a function of galaxy formation model parameters for calibration purposes.

It includes functionality to:

- Generate parameters to perform ground truth runs with in an efficient way as a latin hypercube.
- Train machine learning models, including linear models and Gaussian Process Regression models (with mean models), on this data in a very clean way.
- Generate densely populated synthetic data across the original parameter space, and tools to generate complex model discrepancy descriptions (known here as penalty functions).
- Generate sweeps across model parameter space for the emulated scaling relations to assist in physical insight, as well as sensitivity analysis tools based upon raw and synthetic data.
- Validate predictions through cross validation.
- Visualise the resulting penalty data to assist in model choice decisions.
- Produce inputs and read outputs from the cosmological code SWIFT that processed by VELOCIraptor and the swift-pipeline.

Information about SWIFT can be found here, Information about VELOCIraptor can be found here and Information about the SWIFT-pipeline can be found here.

By combining a selection of SWIFT-io and GP analysis tools, the SWIFT-Emulator serves to make emulation of SWIFT outputs very easy, while staying flexible enough to emulate anything, given a good set of training data.
In this section the basics of using the SWIFT-Emulator will be explained, with examples of how to make your first GP predictions.

1.1 Installation

The package can be installed easily from PyPI under the name swiftemulator, so:

```
pip3 install swiftemulator
```
This will install all necessary dependencies.

The package can be installed from source, by cloning the repository and then using `pip install -e .` for development purposes.

1.2 Requirements

The package requires a number of numerical and experimental design packages. These have been tested (and are continuously tested) using GitHub actions CI to use the latest versions available on PyPI. See `requirements.txt` for details for the packages required to develop SWIFT-Emulator. The packages will be installed automatically by `pip` when installing from PyPI.

1.3 Loading data

In the way we set up the emulator, loading the data is the most cumbersome part of emulation. Once everything is in the right format the emulation itself will be very easy.

At the basis of the SWIFT-Emulator lies the ability to train a Gaussian process (GP) based on a set of training data. As the main goal is emulating scaling relations on the back of hydro simulations you should think of the emulation being in the following form

\[
GP(y, x, \theta),
\]

where we want to predict the dependent \( y \) as a function of the independent \( x \) and model parameters \( \theta \). The distinction between \( x \) and \( \theta \) is made to distinguish the relation that can be obtained from a single simulation output (Like the number density of galaxies as a function of their stellar mass, where \( y \) is the number density and \( x \) the stellar mass) from the parameters that span different outputs (Like redshift, or AGN feedback strength). For this example we will predict the stellar mass function using some data generated with a Schecter function.
import swiftemulator as se
import numpy as np

def log_schecter_function(log_M, log_M_star, alpha):
    M = 10 ** log_M
    M_star = 10 ** log_M_star
    return np.log10((1 / M_star) * (M / M_star) ** alpha * np.exp(-M / M_star))

where we set the normalisation to unity. In this case we will use log M as the independent, while M_star and alpha are our model parameters. The choice to emulate in log space is important, this massively decreases the dynamic range which makes it a lot easier for a Gaussian process to emulate accurately.

In order to get the data in the correct form we need to define three containers. We start by specifying some of the basic information of our model. This is done via swiftemulator.backend.model_specification().

model_specification = se.ModelSpecification(
    number_of_parameters=2,
    parameter_names=['log_M_star','alpha'],
    parameter_limits=[[11.,12.],[-1.,-3.]],
    parameter_printable_names=['Mass at knee','Low mass slope'],
)

The mode specification is used to store some of the metadata of the training set.

Lets assume our training set consists of 100 simulations where our model parameters are randomly sampled

log_M_star = np.random.uniform(11., 12., 100)
alpha = np.random.uniform(-1., -3., 100)

This can be used to set up the second container, which contains the model parameters. For each unique model, named by unique_identifier, we store the values in a dictionary

modelparameters = {}
for unique_identifier in range(100):
    modelparameters[unique_identifier] = {'log_M_star': log_M_star[unique_identifier],
                                          'alpha': alpha[unique_identifier]}
model_parameters = se.ModelParameters(model_parameters=modelparameters)

The unique_identifier is really important, as this will be used to link the model parameters to the model values. There are some major advantages to splitting this up. By splitting the model from the scaling relation we only have to define the model once, allowing us to attach as many relations to it as we want.

Final thing is adding the values of the function we want to emulate. This is done in a similar way to how we add the model parameters, except that we now attach a complete array for each model.

modelvalues = {}
for unique_identifier in range(100):
    independent = np.linspace(10,12,10)
    dependent = log_schecter_function(independent,
                                      log_M_star[unique_identifier],
                                      alpha[unique_identifier])
    dependent_error = 0.02 * dependent
    modelvalues[unique_identifier] = {'independent': independent,
                                       'dependent': dependent,
                                       'dependent_error': dependent_error}
1.4 Training the emulator

After setting up the model containers, training the emulator becomes very simple. First we create an empty GP, which we can then train on the data we have just loaded.

```python
from swiftemulator.emulators import gaussian_process
schecter_emulator = gaussian_process.GaussianProcessEmulator()
schecter_emulator.fit_model(model_specification=model_specification,
                           model_parameters=model_parameters,
                           model_values=model_values)
```

This might take a little bit of time. At this point the GP is fully trained and can be used to make predictions. There are a lot more options when setting up the GP, like inducing a model for mean, but if your input is smooth this is likely all you will need.

1.5 Making predictions

The real reason to use an emulator is to eventually predict the shape of the scaling relation continuously over the parameterspace. Just like training the emulator, making predictions is extremely simple.

```python
predictparams = {
    "log_M_star": 11.5,
    "alpha": -2
}
predict_x = np.linspace(10, 12, 100)
pred, pred_var = schecter_emulator.predict_values(predict_x, predictparams)
```

The main thing to keep in mind is that you give the model parameters as a dictionary again, with the same names as how they are defined in the `model_parameters`. In this case we can directly compare with the original model.

```python
import matplotlib.pyplot as plt
plt.plot(predict_x, pred, label="Emulator")
plt.plot(predict_x, log_schecter_function(predict_x, predictparams["log_M_star"], predictparams["alpha"])
          ,color="black",ls=":",label="Model")
plt.xlabel("Stellar mass")
plt.ylabel("dn/dlogM")
plt.legend()
```

Which shows that the emulator can predict the model with high accuracy.
This covers the most basic way to use SWIFT-Emulator and should give a good baseline for using some of the additional features it offers.
Here we will outline some of the available tools that can help inspect the performance of the emulator. The example data will be the Schecter function example:

```python
import swiftemulator as se
from swiftemulator.emulators import gaussian_process
import numpy as np

def log_schecter_function(log_M, log_M_star, alpha):
    M = 10 ** log_M
    M_star = 10 ** log_M_star
    return np.log10((1 / M_star) * (M / M_star) ** alpha * np.exp(- M / M_star ))

model_specification = se.ModelSpecification(
    number_of_parameters=2,
    parameter_names=["log_M_star","alpha"],
    parameter_limits=[[11.,12.],[-1.,-3.]],
    parameter_printable_names=["Mass at knee","Low mass slope"],
)

log_M_star = np.random.uniform(11., 12., 100)
alpha = np.random.uniform(-1., -3., 100)

modelparameters = {}
for unique_identifier in range(100):
    modelparameters[unique_identifier] = {"log_M_star": log_M_star[unique_identifier],
                                            "alpha": alpha[unique_identifier]}

model_parameters = se.ModelParameters(model_parameters=modelparameters)

modelvalues = {}
for unique_identifier in range(100):
    independent = np.linspace(10,12,10)
    dependent = log_schecter_function(independent,
                                      log_M_star[unique_identifier],
                                      alpha[unique_identifier])
    dependent_error = 0.02 * dependent
    modelvalues[unique_identifier] = {"independent": independent,
                                        "dependent": dependent,
                                        "dependent_error": dependent_error}
```

(continues on next page)
model_values = se.ModelValues(model_values=model_values)

schecter_emulator = gaussian_process.GaussianProcessEmulator()
schecter_emulator.fit_model(model_specification=model_specification,
model_parameters=model_parameters,
model_values=model_values)

2.1 Cross checks

To set up a cross check, the emulator is trained on all but one of the input data-sets. The resulting emulator can then be compared against the model that was left out.

Cross checks are the main way of quantifying emulator performance in the absence of validation data. When emulating via cosmological simulations it is likely to be very expensive to generate a validation dataset of sufficient size. For cases like this SWIFT-Emulator has an easy way of setting up cross-checks.

The `swiftemulator.sensitivity.cross_check()` object acts identically to `swiftemulator.emulators.gaussian_process()` and takes the same inputs. By setting the cross-checks up in this way you can directly compare the results with the main GP that you use for predictions.

```python
from swiftemulator.sensitivity import cross_check

schecter_ccheck = cross_check.CrossCheck()
schecter_ccheck.build_emulators(model_specification=model_specification,
model_parameters=model_parameters,
model_values=model_values)
```

In this case `build_emulators` takes the place of `fit_model`. Note that `build_emulators` now creates N independent trained emulators, where N is the number of models, so this can take quite a long time. For this example the amount of models was reduced from 100 to 20.

Once the emulators have been build there are some inherent tools to have a look at the result (see `swiftemulator.sensitivity.cross_check()`). We will use `build_mocked_model_values_original_independent()` to compare the cross-check predictions with the original data.

```python
import matplotlib.pyplot as plt
data_by_cc = schecter_ccheck.build_mocked_model_values_original_independent()

for unique_identifier in range(20):
    cc_over_og = data_by_cc[unique_identifier]["dependent"] / \
                 model_values[unique_identifier]["dependent"]
    plt.plot(data_by_cc[unique_identifier]["independent"], cc_over_og)
    plt.xlabel("Mass")
    plt.ylabel("Cross-check / Truth")
plt.savefig("Cross_check_accuracy.png", dpi=200)
```
Just with a few line we are able to quantify how accurate the emulator is. Also note that any \textit{ModelValues} container can be parsed as if it is a dictionary.

### 2.2 Sweeps Of Parameter Space

One of the advantages of using emulators is that it supplies you with a fully continuous model of the given function. Besides fitting the parameters it is often interesting to see the effect of changing a single parameter, by doing a sweep. This is implemented into the SWIFT-Emulator with \texttt{swiftemulator.mocking.mock_sweep()}.

```python
from swiftemulator.mocking import mock_sweep

center = {"log_M_star": 11.5, "alpha": -2.0}

Mock_values, Mock_parameters = mock_sweep(schecter_emulator, model_specification, 6, "alpha", center)

for mock_name in Mock_values.keys():
    plt.plot(Mock_values[mock_name]["independent"],
             Mock_values[mock_name]["dependent"],
             label = "Alpha = " + str(Mock_parameters[mock_name]["alpha"])[:4])

plt.xlabel("Stellar mass")
plt.ylabel("$dn/d\log M$")
plt.legend()
plt.savefig("parameter_sweep.png", dpi=200)
```

2.2. Sweeps Of Parameter Space
mock_sweep returns the values and parameter of the sweep as `ModelValues` and `ModelParameters` containers, that are easy to parse.

### 2.3 Model Parameters Features

This highlights two small functions that are attached to the `swiftemulator.backend.model_parameters()` object. The first is the ability to generate a quick plot of the experimental design using corner.

```python
model_parameters.plot_model(model_specification)
```
Note that the axis label used here are the one passed to the model specification. This can be used to have a quick look at whether your space is well sampled.

After finding a set of best fit model parameters it is sometimes useful to see if there are any individual model that has similar values. `find_closest_model` takes a dictionary of input values and finds the training model that is closest to those values.

```python
best_model = {"log_M_star": 11.3, "alpha": -2.1}
model_parameters.find_closest_model(best_model,number_of_close_models=5)
```

which outputs

```
([2, 12, 18, 19, 3],
 [{'log_M_star': 11.26347510702813, 'alpha': -1.9614226414699145},
  {'log_M_star': 11.507944778215956, 'alpha': -1.9818583963792449},
  {'log_M_star': 11.19527147203741, 'alpha': -1.8330160108907092},
  {'log_M_star': 11.033961506507945, 'alpha': -2.275313906753826},
  {'log_M_star': 11.67912812994198, 'alpha': -2.0664526312834353}])
```

It returns a list with the `unique_identifier` of each close model, and the model parameters belonging to that model. This can be used to explore the models close to your best fit model, for example to check how well sampled that part of parameter space is.

### 2.3. Model Parameters Features
2.4 Checking Hyperparameters

In general one should not look at the hyperparameters. They should only be used as a diagnostic when the emulator is giving strange results. The SWIFT-Emulator provides an easy way to check the parameterspace of the hyperparameters. The hyperparameters are optimised to using the marginalised likelihood, so we can inspect how well converged they are by looking at the probability distribution of each individual hyperparameter. This is done via `swiftemulator.emulators.gaussian_process_mcmc()`. In this case MCMC implies the use of Markov chain Monte Carlo (via `emcee`) to find the best hyperparameters, allowing us to look at the complete parameter space.

```python
from swiftemulator.emulators import gaussian_process_mcmc
schecter_emulator_mcmc = gaussian_process_mcmc.GaussianProcessEmulatorMCMC(burn_in_steps=1, mcmc_steps=1000)
schecter_emulator_mcmc.fit_model(model_specification=model_specification, model_parameters=model_parameters, model_values=model_values)
schecter_emulator_mcmc.plot_hyperparameter_distribution()
```
This method is a lot slower than the default hyperparameter optimisation, and may take some time to compute. The main takeaway from plots like this is to see whether the hyperparameters are converged, and whether they are consistent with the faster optimisation method.
For any smooth function with low dynamic range the default gaussian process will provide all the accuracy that is necessary for most purposes. However, it is not hard to imagine certain situations where taking some extra care before invoking the GP can lead to more accuracy. Here, some of the extra options that are available for the emulation are highlighted. The example data will be the Schecter function example:

```python
import swiftemulator as se
from swiftemulator.emulators import gaussian_process
import numpy as np

def log_schecter_function(log_M, log_M_star, alpha):
    M = 10 ** log_M
    M_star = 10 ** log_M_star
    return np.log10( (1 / M_star) * (M / M_star) ** alpha * np.exp(- M / M_star ))

model_specification = se.ModelSpecification(
    number_of_parameters=2,
    parameter_names=["log_M_star","alpha"],
    parameter_limits=[[11.,12.],[-1.,-3.]],
    parameter_printable_names=["Mass at knee","Low mass slope"],
)

log_M_star = np.random.uniform(11., 12., 100)
alpha = np.random.uniform(-1., -3., 100)

modelparameters = {}
for unique_identifier in range(100):
    modelparameters[unique_identifier] = {"log_M_star": log_M_star[unique_identifier],
                                          "alpha": alpha[unique_identifier]}

model_parameters = se.ModelParameters(model_parameters=modelparameters)

modelvalues = {}
for unique_identifier in range(100):
    independent = np.linspace(10,12,10)
    dependent = log_schecter_function(independent, log_M_star[unique_identifier],
                                       alpha[unique_identifier])
    dependent_error = 0.02 * dependent
    modelvalues[unique_identifier] = {"independent": independent,
                                       "dependent": dependent,
                                       "error": dependent_error}
```

(continues on next page)
3.1 Mean models

The most basic addition to a GP is to exchange the constant (or zero) mean for a more complete model. For the SWIFT-Emulator these can be found under `swiftemulator.mean_models()`. All currently implement models come in the form of different order polynomials. Much like the GP you have to define your model first. The model can then be passed to the GP which will fit the coefficients and use that as a mean model. The GP will then be used to model the residuals between the polynomial fit and the data.

```python
from swiftemulator.mean_models.polynomial import PolynomialMeanModel
polynomial_model = PolynomialMeanModel(degree=1)

schecter_emulator = gaussian_process.GaussianProcessEmulator(mean_model=polynomial_model)
schecter_emulator.fit_model(model_specification=model_specification,
                            model_parameters=model_parameters,
                            model_values=model_values)
```

The polynomial model fits a polynomial surface to all parameters. This includes not just the polynomial coefficients for each parameter but also the linear combinations up to the degree of the model. Be careful picking a degree that is very large, as it can quickly lead to over-fitting.

3.2 Emulating Bin-by-Bin

The scaling relations obtained from simulations are often binned relations. For the all-purpose emulator we use the `independent` as an additional parameter which the emulator uses for prediction, but there are situations where you would prefer modeling the response at each `independent` bin separately, instead of modeling it all at once.

When emulating bin-to-bin the main difference is that your GP now comes from `swiftemulator.emulators.gaussian_process_bins()`. Each bin will have a unique emulator, that is trained on all data available for that bin. A bin is created for each unique value of the `independent` found in the `ModelValues` container. It is extremely important that each bin of the original data-set has exactly the same value for the `independent`. However, the individual models do not need the same sample of bins. If some models are missing values for some of the bins, this is not a problem.

Using the binned emulator is as simple as

```python
from swiftemulator.emulators import gaussian_process_bins

schecter_emulator_binned = gaussian_process_bins.GaussianProcessEmulatorBins()
schecter_emulator_binned.fit_model(model_specification=model_specification,
                                    model_parameters=model_parameters,
                                    model_values=model_values)
```
Which has the same prediction functionality as the standard `gaussian_process`. Note that there is also a binned version of the cross checks, `swiftemulator.sensitivity.cross_check_bins()`, which acts the same as the normal `cross_check` but instead uses the binned emulator, making it easy to compare the two methods.

### 3.3 1D Emulation

Sometimes the emulation problem is better solved as

\[ f(\vec{\theta}) \]

In this case we only have the model parameters. The emulator won’t be a function of an additional x parameter stored in the model values. In this case the use can use `swiftemulator.emulators.gaussian_process_one_dim()`. This method has similar functionality as the other emulator types. It will still need a ModelValues container. Here is an example of how such a container should look like:

```python
modelvalues = {}
for unique_identifier in range(100):
    dependent = func(a_arr[unique_identifier], b_arr[unique_identifier])
    dependent_error = 0.02 * dependent
    modelvalues[unique_identifier] = {
        "independent": [None],
        "dependent": [dependent],
        "dependent_error": [dependent_error]
    }
```

In order to make use of the general emulator containers, it is still required to provide the values as list. In this case the lists will only contain a single value. The independent value will not be read. When your data is in the correct format the emulator can be trained like all the other methods.

```python
from swiftemulator.emulators import gaussian_process_one_dim
schecter_emulator_one_dim = gaussian_process_one_dim.GaussianProcessEmulator1D()
schecter_emulator_one_dim.fit_model(model_specification=model_specification, model_parameters=model_parameters, model_values=model_values)
```

The only other thing of note is that while `predict_values` retains the same functionality, you are no longer required to specify any independent values. The prediction is now based purely of the given values of the model parameters.
The SWIFT-Emulator is designed to easily generate SWIFT parameter files, and read swift-pipeline outputs. Here you can find a description of how to use the SWIFT-Emulator’s tools to set up an experimental design, and then load the resulting scaling relations.

4.1 Experimental Design

One part of SWIFT-Emulator’s i/o features is the ability to generate Latin Hypercube (LH) designs and save them to SWIFT parameter files, one file for each set of parameters. For this example you will need to download some data from http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/emulator_output.zip.

We do this by combining the swiftemulator.design() with swiftemulator.io.swift(). First we have to specify what parameter we want to vary.

```python
from swiftemulator.design import latin
from swiftemulator.io.swift import write_parameter_files
from swiftemulator import ModelSpecification

spec = ModelSpecification(
    number_of_parameters=5,
    parameter_names=[
        "EAGLEFeedback:SNII_energy_fraction_min",
        "EAGLEFeedback:SNII_energy_fraction_max",
        "EAGLEFeedback:SNII_energy_fraction_n_Z",
        "EAGLEFeedback:SNII_energy_fraction_n_0_H_p_cm3",
        "EAGLEFeedback:SNII_energy_fraction_n_n",
    ],
    parameter_printable_names=[
        "$f_{\text{E, min}}$",
        "$f_{\text{E, max}}$",
        "$n_{Z}$",
        "$\log_{10}(n_{0, H_p \text{ cm}^{-3}})$",
        "$n_{n}$",
    ],
    parameter_limits=[
        [0.0, 1.0],
        [1.0, 7.0],
        [-0.5, 5.0],
        [-1.0, 1.5],
        [-0.5, 5.0],
    ]
)
```

(continues on next page)
In this case it is important that your \textit{parameter_names} are identical to the names in the SWIFT parameter file. The parameter file is a \texttt{.yml} file so the individual parameters should be named in that format.

In this case the fourth parameter, \textit{EAGLEFeedback:SNII\textunderscore energy\textunderscore fraction\textunderscore n\textunderscore 0\textunderscore H\textunderscore p\textunderscore cm3} is sampled in log-space. The \textit{parameter\_limits} are given in log-space as well in this case, but you need to define the transformation needed when going from the design space, to the value you want to put in the parameter file.

Generating the LH can then be done with \texttt{swiftemulator.design.latin.create\_hypercube()}.

\begin{verbatim}
number_of_simulations = 30
model_parameters = latin.create_hypercube(
    model_specification=spec,
    number_of_samples=number_of_simulations,
)
\end{verbatim}

Now we can use the SWIFT i/o to write these to a set of parameter files. You will have noticed that we only need to provide the parameters that we want to vary. This is because we provide \texttt{write\_parameter\_files} with a base parameter file. This file should hold the base values for all other parameters. Example parameter files can be found on the main SWIFT repository. For this example we will use one of the parameter files from the example hypercube.

\begin{verbatim}
base_parameter_file = "emulator\_output/input\_data/1.yml"
output_path = "."
write_parameter_files(
    filenames={'\{}\output\_path\} / \{key\}.yml"
        for key in model_parameters.model_parameters.keys()\},
    model_parameters=model_parameters,
    parameter_transforms=parameter_transforms,
    base_parameter_file=base_parameter_file,
)
\end{verbatim}

This writes 30 files to the current directory. These files can then be used to run SWIFT for each of the models.

### 4.2 Loading SWIFT data

In order for this example to work you will need to download some data from \url{http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/emulator_output.zip}. This will contain a set of parameter files and a set of data files. The parameters files will be used to retrieve the parameters of the Latin Hypercube, while the data files contain the results for the scaling relations for each model.

It is advised to use the SWIFT-iо options if you want to compare directly with observational data. The main advantage being that loading the data in this way will ensure that you use the correct units.

What will be required to load the data is a list of all the parameter files, and a list for all the data files. This can easily be obtained using \texttt{glob} and \texttt{Path}.
from glob import glob
from pathlib import Path

parameter_files = [Path(x) for x in glob("./emulator_output/input_data/*.yml")]
parameter_filenames = {filename.stem: filename for filename in parameter_files}

data_files = [Path(x) for x in glob("./emulator_output/output_data/*.yml")]
data_filenames = {filename.stem: filename for filename in data_files}

For the parameter we use `swiftemulator.io.swift.load_parameter_files()`. This reads in the parameters and returns both a `ModelSpecification` and a `ModelParameters` container to pass to the emulator.

```python
from swiftemulator.io.swift import load_parameter_files
spec, parameters = load_parameter_files(
    filenames=parameter_filenames,
    parameters=[
        "EAGLEFeedback:SNII_energy_fraction_min",
        "EAGLEFeedback:SNII_energy_fraction_max",
        "EAGLEFeedback:SNII_energy_fraction_n_Z",
        "EAGLEFeedback:SNII_energy_fraction_n_0_H_p_cm3",
        "EAGLEFeedback:SNII_energy_fraction_n_n",
        "EAGLEAGN:coupling_efficiency",
        "EAGLEAGN:viscous_alpha",
        "EAGLEAGN:AGN_delta_T_K",
    ],
    log_parameters=[
        "EAGLEAGN:AGN_delta_T_K",
        "EAGLEAGN:viscous_alpha",
        "EAGLEAGN:coupling_efficiency",
        "EAGLEFeedback:SNII_energy_fraction_n_0_H_p_cm3",
    ],
    parameter_printable_names=[
        "$f_\{rm E, min}\$",
        "$f_\{rm E, max}\$",
        "$n_\{Z\}$",
        "$\log_{10}\$ $n_\{rm H, 0}\$",
        "$n_\{n\}$",
        "$\log_{10}\$ $C_\{rm eff}\$",
        "$\log_{10}\$ $\alpha_\{\rm V}\$",
        "$\log_{10}\$ $\Delta T$",
    ],
)
```

Just like for the experimental design, it is important that the name used for `parameters` is the same as the one used in the parameter file. Note also that you have to supply a list of parameters that where sampled in log-space. These are then trasformed to log-space before being stored in a `ModelParameters` container.

To read the `ModelValues` the function `swiftemulator.io.swift.load_pipeline_outputs()` is used. In this case you have to supply the filenames, and the name(s) of the scaling relation(s). These names can be easily found in the data file and are set by the config used for the pipeline. `log_independent` and `log_dependent` will cause the x or y to be loaded in log-space.

4.2. Loading SWIFT data
from swiftemulator.io.swift import load_pipeline_outputs

values, units = load_pipeline_outputs(
    filenames=data_filenames,
    scaling_relations=["stellar_mass_function_100"],
    log_independent=["stellar_mass_function_100"],
    log_dependent=["stellar_mass_function_100"],
)

scaling_relation = values["stellar_mass_function_100"]
scaling_relation_units = units["stellar_mass_function_100"]

load_pipeline_outputs can return as many scaling relations as required. values is dictionary that contains a ModelValues container for each requested scaling relation. A ModelValues container for a single relation can be obtained by parsing it with the correct name.

At this point the data is loaded and you can build and train your emulator.

def from swiftemulator.emulators import gaussian_process

def emulator = gaussian_process.GaussianProcessEmulator()
emulator.fit_model(model_specification=spec,
    model_parameters=parameters,
    model_values=scaling_relation,
)
CHAPTER FIVE

COMPARING WITH DATA

To unlock the full power of emulation it is often useful to compare your results with observational data. With SWIFT-Emulator you can directly compare the emulated outputs with the observational data stored in velociraptor-comparison-data which can be found here. You can also download just the required Vernon.hdf5 file instead.

In this case we will again use the data from http://virgodb.cosma.dur.ac.uk/swift-webstorage/IOExamples/emulator_output.zip First we have to set up the emulator

```python
from swiftemulator.io.swift import load_parameter_files, load_pipeline_outputs
from swiftemulator.emulators.gaussian_process import GaussianProcessEmulator
from swiftemulator.mean_models import LinearMeanModel
from velociraptor.observations import load_observations
from glob import glob
from pathlib import Path
from tqdm import tqdm
from matplotlib.colors import Normalize
import matplotlib.pyplot as plt
import numpy as np
import corner
import os

files = [Path(x) for x in glob("./emulator_output/input_data/*.yml")]
filenames = {filename.stem: filename for filename in files}

spec, parameters = load_parameter_files(
    filenames=filenames,
    parameters=[
        "EAGLEFeedback:SNII_energy_fraction_min",
        "EAGLEFeedback:SNII_energy_fraction_max",
        "EAGLEFeedback:SNII_energy_fraction_n_Z",
        "EAGLEFeedback:SNII_energy_fraction_n_0_H_p_cm3",
        "EAGLEAGN:AGN_energyкцион fraction_n",
        "EAGLEAGN:coupling_efficiency",
        "EAGLEAGN:viscous_alpha",
        "EAGLEAGN:AGN_delta_T_K",
    ],
    log_parameters=[
        "EAGLEAGN:AGN_delta_T_K",
    ]
)```
In this case we are gonna look at the stellar mass function. To compare we load the calibration SMF for EAGLE-XL.

```python
observation = load_observations(
    path="../velociraptor-comparison-data/data/GalaxyStellarMassFunction/Vernon.hdf5"
)[0]
```
5.1 Penalty Functions

There is a large selection of “Penalty” functions available. We define a penalty function as an analogous to a likelihood.

\[ \mathcal{L} = 1 - P(x, \theta), \]

where \( \mathcal{L} \) is the likelihood and \( P(x, \theta) \) is the accompanying penalty function.

As an example we will use an L2 norm. This will calculate the mean squared distance between the emulator and the data.

```python
from swiftemulator.comparison.penalty import L2PenaltyCalculator
from unyt import Msun, Mpc

L2_penalty = L2PenaltyCalculator(offset = 0.5, lower=9, upper=12)
L2_penalty.register_observation(observation, log_independent=True,
    log_dependent=True,
    independent_units=Msun,
    dependent_units=Mpc**-3)

L2_penalty.plot_penalty(9,12,-6,-1,"penalty_example",x_label="Stellar mass",y_label="dn/dlogM")
```

Now we can combine this with the emulator to compare models in terms of how good they fit the data. Without using the emulator we can use interpolation to be able to quickly check which node of the parameter space best fits the data via swiftemulator.comparison.penalty.L2PenaltyCalculator.penalties()

```python
all_penalties = L2_penalty.penalties(emulator.model_values, np.mean)
all_penalties_array = []
node_number = []
```

(continues on next page)
for key in all_penalties.keys():
    all_penalties_array.append(all_penalties[key])
    node_number.append(int(key))

print("Best fit node = ",node_number[np.argmin(all_penalties_array)])

Best fit node = 107

If we want to check the simulation that is best without rerunning anything we can use node 107. In general we can use this to check not just models at the nodes, but use the emulator to check the complete parameter range. Starting with node 107, let’s see if we can improve the fit by changing one of the parameters.

dictparams = emulator.model_parameters["107"].copy()
x_to_predict = np.log10(L2_penalty.observation.x.value)
pred, pred_var = emulator.predict_values(x_to_predict, predictparams)

print("Mean Penalty of node 107 = ",np.mean(L2_penalty.penalty(x_to_predict,pred)))

#Let’s change one of the parameters and see if it improves the fit
predictparams["EAGLEFeedback:SNII_energy_fraction_max"] = 1
x_to_predict = np.log10(L2_penalty.observation.x.value)
pred, pred_var = emulator.predict_values(x_to_predict, predictparams)

print("Mean after change = ",np.mean(L2_penalty.penalty(x_to_predict,pred)))

Mean Penalty of node 107 = 0.21988119507121354 
Mean after change = 0.3344361855742612

This change makes the fit worse, so no luck. In general you would not do this by hand, but use for example MCMC to sample all the parameters.

5.2 Defining New Penalty Functions

What you want out of these penalty functions can vary wildly, but it is very easy to define your own. There is a large set of functions available within swiftemulator.comparison.penalty(). It is also possible to add your own functions. The base class swiftemulator.comparison.penalty.PenaltyCalculator() covers the most important part, which is loading and interpolating the data. You can then add whichever calculation of the penalties you want. In the example below we create a function that is Gaussian weighted, with a constant error term.

from swiftemulator.comparison.penalty import PenaltyCalculator
import unyt

class ExamplePenaltyCalculator(PenaltyCalculator):
    def penalty(self, independent, dependent, dependent_error):
        # We can use the observational data from the base class.
        # We calculate the observational y-values to compare with
        # from the interpolated observations.
        obs_dependent = self.interpolator_values(independent)
penalties = np.exp(-np.abs(dependent - obs_dependent)**2/0.1)
return penalties

my_penalty = ExamplePenaltyCalculator()
my_penalty.register_observation(observation, log_independent=True, log_dependent=True,
independent_units=Msun, dependent_units=Mpc**-3)

my_penalty.plot_penalty(9,12,-6,-1,"my_penalty",x_label="Stellar mass",y_label="dn/dlogM")

For the simplest models you can also still use the \texttt{plot\_penalty} functionality. There are also PF's available that use the errors on the data, for example \texttt{swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator()}. When creating new penalty functions you can use different parts of already existing ones to make the process very easy.
6.1 swiftemulator package

6.1.1 Subpackages

swiftemulator.backend package

Submodules

swiftemulator.backend.emulator_generator module

Emulator generation object.


Bases: object

Generator for emulators for individual scaling relations.

Parameters

- model_specification (ModelSpecification) – Full instance of the model specification.
- model_parameters (ModelParameters) – Full instance of the model parameters.

Notes

The required initialisation parameters are shared amongst all emulators that the emulator generator produces.

model_specification: swiftemulator.backend.model_specification.ModelSpecification

model_parameters: swiftemulator.backend.model_parameters.ModelParameters

create_gaussian_process_emulator(model_values: swiftemulator.backend.model_values.ModelValues) -> swiftemulator.emulators.gaussian_process.GaussianProcessEmulator

Creates an individual emulator for an individual scaling relation described by the provided model_values.

Parameters
• **model_values** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

• **ModelValues** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

Returns The built and trained emulator ready for prediction steps.

Return type emulator, *GaussianProcessEmulator*

```python
create_gaussian_process_emulator_mcmc(model_values: swiftemulator.backend.model_values.ModelValues) -> swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC
```

Creates the object needed for the hyperparameter_investigator function

Parameters

• **model_values** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

• **ModelValues** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

Returns The built emulator ready for analysis of hyperparameters

Return type emulator, *GaussianProcessEmulatorMCMC*

```python
create_gaussian_process_emulator_bins(model_values: swiftemulator.backend.model_values.ModelValues) -> swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins
```

Creates the object needed for the binned emulator

Parameters

• **model_values** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

• **ModelValues** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

Returns The built emulator ready for analysis of hyperparameters

Return type emulator, *GaussianProcessEmulatorMCMC*

```python
create_linear_model_emulator(model_values: swiftemulator.backend.model_values.ModelValues) -> swiftemulator.emulators.linear_model.LinearModelEmulator
```

Creates an individual emulator for an individual scaling relation described by the provided `model_values`.

Parameters

• **model_values** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

• **ModelValues** – The model values structure for this given scaling relation. This specifies the training data for the emulator.

Returns The built and trained emulator ready for prediction steps.

Return type emulator, *LinearModelEmulator*
swiftemulator.backend.model_parameters module

Model parameters container, contains only the parameters that are part of the differences between models (i.e. not anything about the individual scaling relations!).

```python
class swiftemulator.backend.model_parameters.ModelParameters(model_parameters: Dict[Hashable, Dict[str, float]])
```

Bases: object

Class that contains the parameters of the models, i.e. this does not contain any information about individual scaling relations. Performs validation on the dictionary that is given.

**Parameters**

- **model_parameters** – Free parameters of the underlying model. This is specified as a dictionary with the following structure: `{unique_run_identifier: {parameter_name: parameter_value}}`. Here the unique run identifier can be anything, but it must be unique between runs. An example could be just an integer defining a run number. The parameter names must match with what is defined in the :class:`ModelSpecification`, with the parameter values the specific values taken for that individual simulation run. Note that all models must have each parameter present, and this is checked at the creation time of the ModelParameters object.

- **Dict[Hashable]** – Free parameters of the underlying model. This is specified as a dictionary with the following structure: `{unique_run_identifier: {parameter_name: parameter_value}}`. Here the unique run identifier can be anything, but it must be unique between runs. An example could be just an integer defining a run number. The parameter names must match with what is defined in the :class:`ModelSpecification`, with the parameter values the specific values taken for that individual simulation run. Note that all models must have each parameter present, and this is checked at the creation time of the ModelParameters object.

- **Dict[str]** – Free parameters of the underlying model. This is specified as a dictionary with the following structure: `{unique_run_identifier: {parameter_name: parameter_value}}`. Here the unique run identifier can be anything, but it must be unique between runs. An example could be just an integer defining a run number. The parameter names must match with what is defined in the :class:`ModelSpecification`, with the parameter values the specific values taken for that individual simulation run. Note that all models must have each parameter present, and this is checked at the creation time of the ModelParameters object.

- **float]** – Free parameters of the underlying model. This is specified as a dictionary with the following structure: `{unique_run_identifier: {parameter_name: parameter_value}}`. Here the unique run identifier can be anything, but it must be unique between runs. An example could be just an integer defining a run number. The parameter names must match with what is defined in the :class:`ModelSpecification`, with the parameter values the specific values taken for that individual simulation run. Note that all models must have each parameter present, and this is checked at the creation time of the ModelParameters object.

**Raises** `AttributeError` – When the parameters do not match between all models.

```python
model_parameters: Dict[Hashable, Dict[str, float]]
```

items()

keys()
values()

**find_closest_model**(*comparison_parameters*: Dict[str, float], *number_of_close_models*: int = 1) → Tuple[List[Hashable], List[Dict[str, float]]]

Finds the closest model currently in this instance of ModelParameters to the set of provided comparison_parameters, with the option to return the closest ‘n’ sets to the input.

**Parameters**

- `comparison_parameters` – Set of comparison parameters. The closest parameters, and unique identifier, of the run within the current set of model_parameters to this point in n-dimensional parameter space will be returned.
- `Dict[str]` – Set of comparison parameters. The closest parameters, and unique identifier, of the run within the current set of model_parameters to this point in n-dimensional parameter space will be returned.
- `float` – Set of comparison parameters. The closest parameters, and unique identifier, of the run within the current set of model_parameters to this point in n-dimensional parameter space will be returned.
- `number_of_close_models` – Number of closest model that will be returned
- `int` – Number of closest model that will be returned

**Returns**

- `unique_identifier, List[Hashable]` – Unique identifier of the closest run(s).
- `closest_parameters, List[Dict[str, float]]` – Model parameters of the closest run(s).


Plots the model parameters based on the model specification given. Can either be saved to file, or show.

**Parameters**

- `model_specification` – Model specification object for this set of parameters.
- `ModelSpecification` – Model specification object for this set of parameters.
- `filename` – Name for the file to which the plot is saved. Optional, if None it will show the image.
- `Union[str]` – Name for the file to which the plot is saved. Optional, if None it will show the image.
- `Path` – Name for the file to which the plot is saved. Optional, if None it will show the image.
- `optional` – Name for the file to which the plot is saved. Optional, if None it will show the image.
- `corner_kwargs` (Dict[str, Any], optional) – Optional key word arguments to pass to corner for the plotting.

**to_yaml**(*filename*: pathlib.Path)

Write the model parameters to a YAML file.

**Parameters**

- `filename` (Path) – The path to write the file to. This should be a Path object, but if it is a string it will be automatically converted.
classmethod from_yaml(filename: pathlib.Path) → swiftemulator.backend.model_parameters.ModelParameters

Generate an instance of ModelParameters from a YAML file, written to disk using to_yaml.

Parameters
filename (Path) – The path to read the file from. This should be a Path object, but if it is a string it will be automatically converted.

Returns model_parameters – Instance of ModelParameters restored from disk.

Return type ModelParameters
to_json(filename: pathlib.Path)

Write the model parameters to a JSON file. Preferred over YAML as this is much faster for large datasets.

Parameters
filename (Path) – The path to write the file to. This should be a Path object, but if it is a string it will be automatically converted.

classmethod from_json(filename: pathlib.Path) → swiftemulator.backend.model_parameters.ModelParameters

Generate an instance of ModelParameters from a JSON file, written to disk using to_json.

Parameters
filename (Path) – The path to read the file from. This should be a Path object, but if it is a string it will be automatically converted.

Returns model_parameters – Instance of ModelParameters restored from disk.

Return type ModelParameters

swiftemulator.backend.model_specification module

Model specification for the emulator.
Contains the :class:`ModelSpecification` class, used to input the parameter names and limits.


Bases: object

Base specification for the model. Contains information about the names of parameters and their ranges.

Parameters

• number_of_parameters – Total number of variable parameters in your model.
• int – Total number of variable parameters in your model.
• parameter_names – The names of parameters in your model; these will be used to access the parameters in :class:`ModelParameters`.
• List[str] – The names of parameters in your model; these will be used to access the parameters in :class:`ModelParameters`.
• parameter_limits – The lower and upper limit of the input parameters. Should be the same length as parameter_names, but each item is a list of length two, with a lower and upper bound. For example, in a two parameter model `[[0.0, 1.0], [8.3, 9.3]]` would mean that the first parameter would vary between 0.0 and 1.0, with the second parameter varying between 8.3 and 9.3.
List[List[float]] – The lower and upper limit of the input parameters. Should be the same length as parameter_names, but each item is a list of length two, with a lower and upper bound. For example, in a two parameter model 

```
[[0.0, 1.0], [8.3, 9.3]]
```

would mean that the first parameter would vary between 0.0 and 1.0, with the second parameter varying between 8.3 and 9.3.

parameter_printable_names – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.

List[str] – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.

optional – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.

Raises AttributeError – When the number of parameters in all of the required attributes are not equal (e.g. a different number of names has been provided compared to the number of limits).

Raises AttributeError – When the number of parameters in all of the required attributes are not equal (e.g. a different number of names has been provided compared to the number of limits).

number_of_parameters: int
parameter_names: List[str]
parameter_limits: List[List[float]]
parameter_printable_names: Optional[List[str]]

property salib_problem
Generates the SALib problem dictionary.

swiftemulator.backend.model_values module

Object describing the model values, i.e. this uniquely describes the set of scaling relations present.

class swiftemulator.backend.model_values.ModelValues(model_values: Dict[Hashable, Dict[str, Optional[np.array]]])

Bases: object

Set of model values for a given scaling relation.

Parameters

- model_values – Model values for this given scaling relation. Must have the following structure: 

```
{unique_identifier: {"independent": np.array, "dependent": np.array, "dependent_errors": Optional[ np.array]}},
```

with the unique_identifiers the same as those that were specified within the ModelParameters object. The dependent errors are optional, but if present must be of the same length as the dependent variables. Both independent and dependent are required for every model value that is present. If a unique identifier is present in the model parameters, but not in the values, it is simply left out of the emulation step.

- Dict[Hashable] – Model values for this given scaling relation. Must have the following structure: 

```
{unique_identifier: {"independent": np.array, "dependent": np.array, "dependent_errors": Optional[ np.array]}},
```

with the unique_identifiers the same as those that were specified within the ModelParameters object. The dependent errors are optional, but if present must be of the same length as the dependent variables. Both independent and dependent are required for every model value that is present. If a unique identifier is present in the model parameters, but not in the values, it is simply left out of the emulation step.
• Dict[str] – Model values for this given scaling relation. Must have the following structure: `{unique_identifier: {"independent": np.array, "dependent": np.array, "dependent_errors": Optional[np.array]}}`, with the unique_identifiers the same as those that were specified within the ModelParameters object. The dependent errors are optional, but if present must be of the same length as the dependent variables. Both independent and dependent are required for every model value that is present. If a unique identifier is present in the model parameters, but not in the values, it is simply left out of the emulation step.

• Optional[np.array]]] – Model values for this given scaling relation. Must have the following structure: `{unique_identifier: {"independent": np.array, "dependent": np.array, "dependent_errors": Optional[np.array]}}`, with the unique_identifiers the same as those that were specified within the ModelParameters object. The dependent errors are optional, but if present must be of the same length as the dependent variables. Both independent and dependent are required for every model value that is present. If a unique identifier is present in the model parameters, but not in the values, it is simply left out of the emulation step.

Raises AttributeError – If the model values do not conform to the required specification.

model_values: Dict[Hashable, Dict[str, Optional[numpy.array]]]

items()
keys()
values()

property number_of_variables: int
    Total number of variables present in all models.

to_yaml(filename: pathlib.Path)
    Write the model values to a YAML file.

    Parameters filename (Path) – The path to write the file to. This should be a Path object, but if it is a string it will be automatically converted.

    classmethod from_yaml(filename: pathlib.Path) → swiftemulator.backend.model_values.ModelValues
    Generate an instance of ModelValues from a YAML file, written to disk using to_yaml.

    Parameters filename (Path) – The path to read the file from. This should be a Path object, but if it is a string it will be automatically converted.

    Returns model_values – Instance of ModelValues restored from disk.

    Return type ModelValues

to_json(filename: pathlib.Path)
    Write the model values to a JSON file. Preferred to YAML as this is much faster for large datasets.

    Parameters filename (Path) – The path to write the file to. This should be a Path object, but if it is a string it will be automatically converted.

    classmethod from_json(filename: pathlib.Path) → swiftemulator.backend.model_values.ModelValues
    Generate an instance of ModelValues from a JSON file, written to disk using to_json.

    Parameters filename (Path) – The path to read the file from. This should be a Path object, but if it is a string it will be automatically converted.

    Returns model_values – Instance of ModelValues restored from disk.

    Return type ModelValues
**swiftemulator.comparison package**

Comparison to observational data, including visualisation and penalty functions.

**Submodules**

**swiftemulator.comparison.penalty module**

Penalty functions and their specifications.

```python
class swiftemulator.comparison.penalty.PenaltyCalculator
    Bases: object

    Base class for the penalty functions.

    Extend this with your own, following the following pattern:
    1. Configuration parameters for the penalty function are taken as initialisation parameters to the class.
    2. The observational data set to be matched to is passed to register_observation.
    3. The penalty function is calculated using the penalty() method, taking the exact arguments that are taken here, for an individual model.

    Provided for convenience is penalties() which calculates the penalty function for all data in a ModelValues container.

    `observation`: velociraptor.observations.objects.ObservationalData
    `interpolator_values`: scipy.interpolate.interpolate.interp1d
    `interpolator_error`: Optional[scipy.interpolate.interpolate.interp1d]
    `log_independent`: bool
    `log_dependent`: bool
    `independent_units`: unyt.array.unyt_quantity
    `dependent_units`: unyt.array.unyt_quantity

    register_observation(`observation`: velociraptor.observations.objects.ObservationalData, `log_independent`: bool, `log_dependent`: bool, `independent_units`: unyt.array.unyt_quantity, `dependent_units`: unyt.array.unyt_quantity) → None

    Registers the observation for use in penalty with the class.

    **Parameters**
    - `observation` *(ObservationalData)* – Instance of the velociraptor observational data used for comparisons.
    - `log_independent` *(bool)* – Take the base-10 log of the independent data before comparison?
    - `log_dependent` *(bool)* – Take the base-10 log of the dependent data before comparison?
    - `independent_units` *(unyt.unyt_quantity)* – The units that the model was calculated in (independent)
    - `dependent_units` *(unyt.unyt_quantity)* – The units that the model was calculated in (dependent)
```

Chapter 6. API Documentation
observation_interpolation()

Produce the interpolation for the internal observation.

penalty(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → List[float]

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

independent: np.array The independent data.
dependent: np.array The dependent data for comparison.
dependent_error: np.array, optional The dependent errors, for comparison.

Returns The penalties for this model, between 0 and 1 each.

Return type: penalty, List[float]

penalties(model: swiftemulator.backend.model_values.ModelValues, collate_with: Callable) → Dict[Hashable, float]

Calculate the penalty function for all models in the model values container.

It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

Parameters
- model (ModelValues) – The set of model (values) to calculate the penalty function for.
- collate_with (Callable) – A function that takes a numpy array and returns the ‘global’ penalty for a model given the input for all of the valid points in the array. Examples could be np.max, np.mean, np.median.

Returns penalties – Penalty functions for each of the models, with the key being the unique identifier.

Return type: Dict[Hashable, float]


Create a figure of the penalty function, over the limits given. Limits are given in log space if logged, linear space if not (with units required in that case).

class swiftemulator.comparison.penalty.L1PenaltyCalculator(offset: Union[unyt.array.unyt_quantity, float], lower: Union[unyt.array.unyt_quantity, float], upper: Union[unyt.array.unyt_quantity, float])

Bases: swiftemulator.comparison.penalty.PenaltyCalculator

Penalty calculator for an L1-type norm, i.e. a linear penalty function away from the data. This penalty function is capped after a (vertical) distance, provided with units if the provided observation is used in linear space, or provided as a logarithmic offset in dex if in log space.

Parameters
• **offset** *(Union[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum. This is required as the penalty function is not allowed to be unlimited.*

• **lower** *(Union[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.*

• **upper** *(Union[unyt.unyt_quantity, float]) – The highest independent value to calculate the model offset at.*

**offset**: Union[unyt.array.unyt_quantity, float]

**lower**: Union[unyt.array.unyt_quantity, float]

**upper**: Union[unyt.array.unyt_quantity, float]

**observation_interpolation()**

Produces the interpolation for the internal observation.

**penalty**(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → float

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array The independent data.

**dependent**: np.array The dependent data for comparison.

**dependent_error**: np.array, optional The dependent errors, for comparison.

**Returns** The penalties for this model, between 0 and 1 each.

**Return type** penalty, List[float]

class swiftemulator.comparison.penalty.L1PenaltyCalculatorOneSided(offset: Union[unyt.array.unyt_quantity, float], maximum_penalty: str, lower: Union[unyt.array.unyt_quantity, float], upper: Union[unyt.array.unyt_quantity, float])

**Bases**: swiftemulator.comparison.penalty.PenaltyCalculator

Penalty calculator for an L1-type norm, i.e. a linear penalty function away from the data, but one-sided only. Values above/below the line are given a maximal penalty.

This penalty function is capped after a (vertical) distance, provided with units if the provided observation is used in linear space, or provided as a logarithmic offset in dex if in log space.

**Parameters**

• **offset** *(Union[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum. This is required as the penalty function is not allowed to be unlimited.*

• **maximum_penalty** *(str) – Give the maximum penalty above or below the line. Accepted values are “above” or “below” as strings.*
• **lower** (*Union*[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.

• **upper** (*Union*[unyt.unyt_quantity, float]) – The highest independent value to calculate the model offset at.

**offset**: Union[unyt.array.unyt_quantity, float]

**maximum_penalty**: str

**lower**: Union[unyt.array.unyt_quantity, float]

**upper**: Union[unyt.array.unyt_quantity, float]

**observation_interpolation()**

Produces the interpolation for the internal observation.


Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array The independent data.

**dependent**: np.array The dependent data for comparison.

**dependent_error**: np.array, optional The dependent errors, for comparison.

**Returns** The penalties for this model, between 0 and 1 each.

**Return type** penalty, List[float]

class swiftemulator.comparison.penalty.L2PenaltyCalculator(offset: Union[unyt.array.unyt_quantity, float], lower: Union[unyt.array.unyt_quantity, float], upper: Union[unyt.array.unyt_quantity, float])

Bases: swiftemulator.comparison.penalty.PenaltyCalculator

Penalty calculator for an L2-type norm, i.e. a square penalty function away from the data. This penalty function is capped after a (vertical) distance, provided with units if the provided observation is used in linear space, or provided as a logarithmic offset in dex if in log space.

**Parameters**

• **offset** (*Union*[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum. This is required as the penalty function is not allowed to be unlimited.

• **lower** (*Union*[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.

• **upper** (*Union*[unyt.unyt_quantity, float]) – The highest independent value to calculate the model offset at.

**offset**: Union[unyt.array.unyt_quantity, float]

**lower**: Union[unyt.array.unyt_quantity, float]
upper: Union[unyt.array.unyt_quantity, float]

observation_interpolation()

Produces the interpolation for the internal observation.

penalty(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → float

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

independent: np.array The independent data.

dependent: np.array The dependent data for comparison.

dependent_error: np.array, optional The dependent errors, for comparison.

Returns The penalties for this model, between 0 and 1 each.

Return type penalty, List[float]

class swiftemulator.comparison.penalty.L1VariablePenaltyCalculator(offset_lower: Union[unyt.unyt_quantity, float], offset_upper: Union[unyt.unyt_quantity, float], offset_transition: Union[unyt.unyt_quantity, float], transition_width: Union[unyt.unyt_quantity, float], lower: Union[unyt.unyt_quantity, float], upper: Union[unyt.unyt_quantity, float], offset_below_above_ratio: float = 1.0)

Bases: swiftemulator.comparison.penalty.PenaltyCalculator

Penalty calculator for an L1-type norm, i.e. a linear penalty function away from the data. This penalty function is capped after a (vertical) distance, provided with units if the provided observation is used in linear space, or provided as a logarithmic offset in dex if in log space.

In this version the offset is variable, using a logistic curve.

Parameters

- offset_lower (Union[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum, at the lower end of the independent range.
- offset_upper (Union[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum, at the upper end of the independent range.
- offset_transition (Union[unyt.unyt_quantity, float]) – The independent variable at which you would like the offset to transition from offset_lower to offset_upper.
- transition_width (Union[unyt.unyt_quantity, float]) – The width of the transition between offsets, centered around offset_transition.
- lower (Union[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.
• **upper** *(Union [unyt.unyt_quantity, float])* – The highest independent value to calculate the model offset at.

• **offset_below_above_ratio** *(float, optional)* – Ratio of the allowed offset below or above the data. If this takes a value of less than 1.0, models below the data are penalised more (by that factor) than models above. Default: 1.0

**offset_lower**: Union [unyt.array.unyt_quantity, float]

**offset_upper**: Union [unyt.array.unyt_quantity, float]

**offset_transition**: Union [unyt.array.unyt_quantity, float]

**transition_width**: Union [unyt.array.unyt_quantity, float]

**lower**: Union [unyt.array.unyt_quantity, float]

**upper**: Union [unyt.array.unyt_quantity, float]

**offset_below_above_ratio**: float

**observation_interpolation()**

Produces the interpolation for the internal observation.

**penalty** *(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → float*

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array The independent data.

**dependent**: np.array The dependent data for comparison.

**dependent_error**: np.array, optional The dependent errors, for comparison.

**Returns** The penalties for this model, between 0 and 1 each.

**Return type** penalty, List[float]

class **swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator** *(offset_squeeze: Union [unyt.array.unyt_quantity, float], offset_normal: Union [unyt.array.unyt_quantity, float], offset_transition: Union [unyt.array.unyt_quantity, float], transition_width: Union [unyt.array.unyt_quantity, float], lower: Union [unyt.array.unyt_quantity, float], upper: Union [unyt.array.unyt_quantity, float], offset_below_above_ratio: float = 1.0)*

**Bases**: swiftemulator.comparison.penalty.PenaltyCalculator
Penalty calculator for an L1-type norm, i.e. a linear penalty function away from the data. This penalty function is capped after a (vertical) distance, provided with units if the provided observation is used in linear space, or provided as a logarithmic offset in dex if in log space.

In this version the offset is variable, with it being ‘squeezed’ at a point over a width.

**Parameters**

- **offset_squeeze** (Union[unyt.unyt_quantity, float]) – The vertical offset at which to set the L1 norm to the maximum at the pinch point.
- **offset_normal** (Union[unyt.unyt_quantity, float]) – The usual vertical offset at which to set the L1 norm to the maximum.
- **offset_transition** (Union[unyt.unyt_quantity, float]) – The independent variable at which you would like the offset to transition from offset_lower to offset_upper.
- **transition_width** (Union[unyt.unyt_quantity, float]) – The width of the transition between offsets, centered around offset_transition.
- **lower** (Union[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.
- **upper** (Union[unyt.unyt_quantity, float]) – The highest independent value to calculate the model offset at.
- **offset_below_above_ratio** (float, optional) – Ratio of the allowed offset below or above the data. If this takes a value of less than 1.0, models below the data are penalised more (by that factor) than models above. Default: 1.0

**observation_interpolation()**

Produces the interpolation for the internal observation.

**penalty** *(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → float*

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

- **independent**: np.array The independent data.
- **dependent**: np.array The dependent data for comparison.
- **dependent_error**: np.array, optional The dependent errors, for comparison.

**Returns** The penalties for this model, between 0 and 1 each.

**Return type** penalty, List[float]
class swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator(
    sigma_max:
    Union[unyt.array.unyt_quantity, float],
    lower:
    Union[unyt.array.unyt_quantity, float],
    upper:
    Union[unyt.array.unyt_quantity, float])

Bases: swiftemulator.comparison.penalty.PenaltyCalculator

Penalty calculator for observations that include errors. This penalty function uses a Gaussian distribution around the data, based on the observational errors. Capped at a input number of sigmas away from the data.

Parameters

- **sigma_max** (Union[unyt.unyt_quantity, float]) – The number of sigmas at which the function is capped.

- **lower** (Union[unyt.unyt_quantity, float]) – The lowest independent value to calculate the model offset at.

- **upper** (Union[unyt.unyt_quantity, float]) – The highest independent value to calculate the model offset at.

**sigma_max**: Union[unyt.array.unyt_quantity, float]

**lower**: Union[unyt.array.unyt_quantity, float]

**upper**: Union[unyt.array.unyt_quantity, float]

**error_interpolator_values**: scipy.interpolate.interpolate.interp1d

observation_interpolation()

Produces the interpolation for the internal observation.

**penalty**(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → List[float]

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array The independent data.

**dependent**: np.array The dependent data for comparison.

**dependent_error**: np.array, optional The dependent errors, for comparison.

Returns The penalties for this model, between 0 and 1 each.

Return type penalty, List[float]

class swiftemulator.comparison.penalty.GaussianPercentErrorsPenaltyCalculator(
    percent_error:
    float,
    sigma_max:
    float, lower:
    Union[unyt.array.unyt_quantity, float], upper:
    Union[unyt.array.unyt_quantity, float])
Penalty calculator that that uses Gaussian errors with widths based on the percentages difference between the model and the data.

**Parameters**

- **percent_error** *(float)* – percent error that sets the one-sigma deviation, in units of percent (0-100).
- **sigma_max** *(float)* – The number of sigmas at which the function is capped.
- **lower** *(Union[unyt.unyt_quantity, float])* – The lowest independent value to calculate the model offset at.
- **upper** *(Union[unyt.unyt_quantity, float])* – The highest independent value to calculate the model offset at.

**percent_error**: float

**sigma_max**: float

**lower**: Union[unyt.array.unyt_quantity, float]

**upper**: Union[unyt.array.unyt_quantity, float]

**observation_interpolation**()

Produces the interpolation for the internal observation.

**penalty**(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → List[float]

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array The independent data.

**dependent**: np.array The dependent data for comparison.

**dependent_error**: np.array, optional The dependent errors, for comparison.

**Returns** The penalties for this model, between 0 and 1 each.

**Return type** penalty, List[float]

**class** swiftemulator.comparison.penalty.GaussianDataErrorsPercentFloorPenaltyCalculator

Refer to the original source for the complete documentation on the **GaussianDataErrorsPercentFloorPenaltyCalculator** class.
Penalty calculator that uses Gaussian errors based on the observational data. Includes a floor based on a percent error. It will pick the worst out of the two. This is meant as a way to not fit better then the emulator allows, while also not constraining stronger than observations.

Parameters

- **percent_error (float)** – percent error that sets the one-sigma deviation, in units of percent (0-100).
- **sigma_max (float)** – The number of sigmas at which the function is capped.
- **lower (Union[unyt.unyt_quantity, float])** – The lowest independent value to calculate the model offset at.
- **upper (Union[unyt.unyt_quantity, float])** – The highest independent value to calculate the model offset at.

percent_error: float

sigma_max: float

lower: Union[unyt.array.unyt_quantity, float]

upper: Union[unyt.array.unyt_quantity, float]

error_interpolator_values: scipy.interpolate.interpolate.interp1d

observation_interpolation()

Produces the interpolation for the internal observation.

penalty (independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → List[float]

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

independent: np.array The independent data.

dependent: np.array The dependent data for comparison.

dependent_error: np.array, optional The dependent errors, for comparison.

Returns The penalties for this model, between 0 and 1 each.

Return type penalty, List[float]

class swiftemulator.comparison.penalty.GaussianWeightedDataErrorsPercentFloorPenaltyCalculator (percent_error: float, sigma_max: float, weight: float, lower: Union[unyt.array.unyt_quantity, float], upper: Union[unyt.array.unyt_quantity, float])
Penalty calculator that that uses Gaussian errors based on the observational data. Includes a floor based on a percent error. It will pick the worst out of the two. This is meant as a way to not fit better then the emulator allows, while also not constraining stronger than observations.

**Parameters**

- **percent_error** *(float)* – percent error that sets the one-sigma deviation, in units of percent (0-100).
- **sigma_max** *(float)* – The number of sigmas at which the function is capped.
- **lower** *(Union[unyt.unyt_quantity, float])* – The lowest independent value to calculate the model offset at.
- **upper** *(Union[unyt.unyt_quantity, float])* – The highest independent value to calculate the model offset at.
- **weight** *(A general weight that scales the entire range of)* – errors, but keeps relative weights intact.

**percent_error**: float

**sigma_max**: float

**weight**: float

**lower**: Union[unyt.array.unyt_quantity, float]

**upper**: Union[unyt.array.unyt_quantity, float]

**error_interpolator_values**: scipy.interpolate.interpolate.interp1d

**observation_interpolation**()

Produces the interpolation for the internal observation.

**penalty**(independent: numpy.array, dependent: numpy.array, dependent_error: Optional[numpy.array] = None) → List[float]

Calculate the penalty function, relative to the observational data, for this model. It is highly recommended that you evaluate the model at the same independent variables as the observational data. The observational data is linearly interpolated to find a prediction at the independent variables that you provide.

**independent**: np.array  The independent data.

**dependent**: np.array  The dependent data for comparison.

**dependent_error**: np.array, optional  The dependent errors, for comparison.

**Returns**  The penalties for this model, between 0 and 1 each.

**Return type**  penalty, List[float]
Visualisation functions for comparison datasets.

Allows you to project a plausibility region for each parameter cross-correlation.

**visualise_penalties_mean**

```python
```

Visualises the penalties using SPH smoothing for each individual model point.

**Parameters**

- **model_specification** (*ModelSpecification*) – The appropriate model specification. Used for the limits of the figure.
- **model_parameters** (*ModelParameters*) – Parameters of the model, with the appropriate unique IDs.
- **penalties** (*Dict[Hashable, float]*) – Penalties for all parameters in *model_parameters*, with the key in this dictionary being the unique IDs.
- **norm** (*Normalize, optional*) – A *matplotlib* normalisation object. By default this uses vmin=0.2 and vmax=0.7.
- **remove_ticks** (*bool, optional*) – Remove the axes ticks? This is recommended, as the plot can become very cluttered if you don’t do this. Default: True.
- **figsize** (*Tuple[float], optional*) – The figure size to use. Defaults to 7 inches by 7 inches, the size for a figure* in the MNRAS template.
- **use_parameters** (*Iterable[str], optional*) – The parameters to include in the figure. If not provided, all parameters in the *model_specification* are used.
- **use_colorbar** (*Bool, optional*) – Include a colorbar? Default: False
- **highlight_model** (*Hashable, optional*) – The model unique ID to highlight. If not provided, no model is highlighted.

**Returns**

- **fig** (*Figure*) – The figure object.
• **axes** (*np.ndarray[Axes]*) – The individual axes.

**Notes**

You can either change how the figure looks by using the figure and axes objects that are returned, or by modifying the matplotlib stylesheet you are currently using.

```python
```

Visualises the penalties using basic binning.

**Parameters**

- **model_specification** (*ModelSpecification*) – The appropriate model specification. Used for the limits of the figure.

- **model_parameters** (*ModelParameters*) – Parameters of the model, with the appropriate unique IDs.

- **penalties** (*Dict[Hashable, float]*) – Penalties for all parameters in model_parameters, with the key in this dictionary being the unique IDs.
• **statistic** *(str, optional)* – The statistic that you would like to compute. Allowed values are the same as for *scipy.stats.binned_statistic_2d*. Defaults to *mean*.

• **norm** *(Normalize, optional)* – A *matplotlib* normalisation object. By default this uses $v_{min}=0.2$ and $v_{max}=0.7$.

• **remove_ticks** *(bool, optional)* – Remove the axes ticks? This is recommended, as the plot can become very cluttered if you don’t do this. Default: True.

• **figsize** *(Tuple[float], optional)* – The figure size to use. Defaults to 7 inches by 7 inches, the size for a figure* in the MNRAS template.

• **use_parameters** *(Iterable[str], optional)* – The parameters to include in the figure. If not provided, all parameters in the *model_specification* are used.

• **use_colorbar** *(Bool, optional)* – Include a colorbar? Default: False.

• **highlight_model** *(Hashable, optional)* – The model unique ID to highlight. If not provided, no model is highlighted.

**Returns**

• **fig** *(Figure)* – The figure object.

• **axes** *(np.ndarray[Axes])* – The individual axes.

**Notes**

You can either change how the figure looks by using the figure and axes objects that are returned, or by modifying the *matplotlib* stylesheet you are currently using.

**swiftemulator.design package**

**Submodules**

**swiftemulator.design.latin module**


Creates a Latin Hypercube model design.

**Parameters**

• **model_specification** *(ModelSpecification)* – Model specification for which to create a latin hypercube from.

• **number_of_samples** *(int)* – The number of samples to draw; this will be the number of input simulations that you wish to create.

• **correlation_retries** *(int, optional)* – Number of times to re-try creating a random hypercube, to minimize the correlation coefficient even further. Default: 32.

• **prefix_unique_id** *(str, optional)* – An optional prefix for the newly generated unique IDs. Defaults to no prefix.
**Returns**  
`model_parameters` – A model values container with the prepared latin hypercube. Contains methods to visualise the output hypercube.

**Return type**  
`ModelParameters`

**Notes**

Uses :mod:`pyDOE`’s `lhs()` function, with the `maximin` method.

### swiftemulator.design.random module

Generates a completely random design, by using the numpy random command.

```python
swiftemulator.design.random.create_cube(model_specification:
swiftemulator.backend.model_specification.ModelSpecification,
number_of_samples: int, prefix_unique_id: Optional[str] = None) →
swiftemulator.backend.model_parameters.ModelParameters
```

Creates a random hypercube model design.

**Parameters**

- `model_specification` (*ModelSpecification*) – Model specification for which to create a latin hypercube from.
- `number_of_samples` (*int*) – The number of samples to draw; this will be the number of input simulations that you wish to create.
- `prefix_unique_id` (*str, optional*) – An optional prefix for the newly generated unique IDs. Defaults to no prefix.

**Returns**  
`model_parameters` – A model values container with the prepared latin hypercube. Contains methods to visualise the output hypercube.

**Return type**  
`ModelParameters`

**Notes**

Uses numpy’s random methods to generate a completely random (i.e. no guarantee of a nice even distribution) hypercube.

### swiftemulator.design.transform module

Transformer from an ND array to a model specification object.

```python
swiftemulator.design.transform.transform_to_model_spec(input_array: numpy.ndarray,
model_specification: swiftemulator.backend.model_specification.ModelSpecification,
prefix_unique_id: Optional[str] = None) →
swiftemulator.backend.model_parameters.ModelParameters
```

Transforms the input nd array (which is of shape n models by n parameters) to a model parameters object, by re-scaling the parameters according to the specification.

**Parameters**
• **input_array** (*np.ndarray*) – Input array, of shape (number of samples, number of parameters).

• **model_specification** (*ModelSpecification*) – Model specification used to rescale the array.

• **prefix_unique_id** (*str, optional*) – An optional prefix for the newly generated unique IDs. Defaults to no prefix.

**Returns**  
**model_parameters** – A model values container with the re-scaled parameters and associated metadata.

**Return type**  
*ModelParameters*

---

**swiftemulator.emulators package**

**Submodules**

**swiftemulator.emulators.base module**

Base for all emulators. This *BaseEmulator* class is a specification for how all emulators should be implemented, but does not implement a specific model itself (all methods raise *NotImplementedError* by default). Controls on the behaviour of the emulator are passed as initialiser arguments, and the data is passed as arguments to *fit_model*.

```python
class swiftemulator.emulators.base.BaseEmulator

    Bases: object

    Base emulator for training models. Initialisation parameters are used in *fit_model* to specify additional parameters to the model.

    ordering: Optional[List[Hashable]] = None

    parameter_order: Optional[List[str]] = None

    model_specification: Optional[swiftemulator.backend.model_specification.ModelSpecification] = None

    model_parameters: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None

    model_values: Optional[swiftemulator.backend.model_values.ModelValues] = None

    independent_variables: Optional[numpy.array] = None

    dependent_variables: Optional[numpy.array] = None

    dependent_variable_errors: Optional[numpy.array] = None

    emulator = None

    fit_model(model_specification: swiftemulator.backend.model_specification.ModelSpecification,
              model_parameters: swiftemulator.backend.model_parameters.ModelParameters,
              model_values: swiftemulator.backend.model_values.ModelValues)

        Fits a model to the independent and dependent variables given by the model spec, parameters, and values.
```

---

**6.1. swiftemulator package**
**predict_values** *(independent: numpy.array, model_parameters: Dict[str, float]) → numpy.array*

Predict values from the trained emulator contained within this object.

**Parameters**

- **independent** – Independent continuous variables to evaluate the emulator at. If the emulator is discrete, these are only allowed to be the discrete independent variables that the emulator was trained at (disregarding the additional ‘independent’ model parameters, below.)

- **np.array** – Independent continuous variables to evaluate the emulator at. If the emulator is discrete, these are only allowed to be the discrete independent variables that the emulator was trained at (disregarding the additional ‘independent’ model parameters, below.)

- **model_parameters** *(Dict[str, float]) – The point in model parameter space to create predicted values at.*

**Returns**

- **dependent_predictions, np.array** – Array of predictions, if the emulator is a function \( f \), these are the predicted values of \( f(independent) \) evaluated at the position of the input model_parameters.

- **dependent_prediction_errors, np.array** – Errors on the model predictions. For models where the errors are unconstrained, this is an array of zeroes.

**Raises** **AttributeError** – When the model has not been trained before trying to make a prediction, or when attempting to evaluate the model at disallowed independent variables.
- **optional** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).

```python
kernel: Optional[george.kernels.Kernel]
mean_model: Optional[swiftemulator.mean_models.base.MeanModel]
model_specification:
    Optional[swiftemulator.backend.model_specification.ModelSpecification] = None
model_parameters: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None
model_values: Optional[swiftemulator.backend.model_values.ModelValues] = None
ordering: Optional[List[Hashable]] = None
parameter_order: Optional[List[str]] = None
independent_variables: Optional[numpy.array] = None
dependent_variables: Optional[numpy.array] = None
dependent_variable_errors: Optional[numpy.array] = None
emulator: Optional[george.gp.GP] = None
```

```python
fit_model(model_specification: swiftemulator.backend.model_specification.ModelSpecification,
          model_parameters: swiftemulator.backend.model_parameters.ModelParameters,
          model_values: swiftemulator.backend.model_values.ModelValues)
```

Fits the gaussian process model, as determined by the initialiser variables of the class (i.e. the kernel and the mean model).

### Parameters

- **model_specification** *(ModelSpecification) – Full instance of the model specification.*
- **model_parameters** *(ModelParameters) – Full instance of the model parameters.*
- **model_values** * (ModelValues) – Full instance of the model values describing this individual scaling relation.*

### Notes

This method uses copies of the internal kernel and mean model objects, as those objects contain slightly unhelpful state information.

```python
predict_values(independent: numpy.array, model_parameters: Dict[str, float]) → numpy.array
```

Predict values from the trained emulator contained within this object.

### Parameters

- **independent** – Independent continuous variables to evaluate the emulator at.
- **np.array** – Independent continuous variables to evaluate the emulator at.
- **model_parameters** *(Dict[str, float]) – The point in model parameter space to create predicted values at.*

### Returns
• `dependent_predictions`, `np.array` – Array of predictions, if the emulator is a function f, these are the predicted values of f(independent) evaluated at the position of the input `model_parameters`.

• `dependent_prediction_errors`, `np.array` – Errors on the model predictions.

**swiftemulator.emulators.gaussian_process_bins module**

Gaussian Process Emulator using an emulator for each bins

```python
```

Bases: `swiftemulator.emulators.base.BaseEmulator`

Generator for emulators for individual scaling relations. Uses a GP for each separate bin.

**Parameters**

- **kernel** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **`george.kernels.Kernel`** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **optional** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **mean_model** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).

- **MeanModel** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).

- **optional** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).

**kernel**: Optional[`george.kernels.Kernel`]

**mean_model**: Optional[`swiftemulator.mean_models.base.MeanModel`]

**model specification**: Optional[`swiftemulator.backend.model_specification.ModelSpecification`] = None

**model parameters**: Optional[`swiftemulator.backend.model_parameters.ModelParameters`] = None

**model_values**: Optional[`swiftemulator.backend.model_values.ModelValues`] = None

**ordering**: Optional[List[Hashable]] = None

**parameter order**: Optional[List[str]] = None

**independent variables**: Optional[`numpy.array`] = None
dependent_variables:  Optional[numpy.array] = None
dependent_variable_errors:  Optional[numpy.array] = None
n_bins:  int = None
bin_model_values:  List[Dict[str, numpy.array]] = None
bin_centers:  List[float] = None
bin_gaussian_process:  List[george.gp.GP] = None


Fits the gaussian process model, as determined by the initialiser variables of the class (i.e. the kernel and the mean model).

Parameters

• model_specification (ModelSpecification) – Full instance of the model specification.
• model_parameters (ModelParameters) – Full instance of the model parameters.
• model_values (ModelValues) – Full instance of the model values describing this individual scaling relation.

Notes

This method uses copies of the internal kernel and mean model objects, as those objects contain slightly unhelpful state information.

predict_values(independent: numpy.array, model_parameters: Dict[str, float]) → numpy.array

Predict values from the trained emulator contained within this object.

Parameters

• independent – Independent continuous variables to evaluate the emulator at. If the emulator is discrete, these are the only allowed to be the discrete independent variables that the emulator was trained at (disregarding the additional ‘independent’ model parameters, below). These can be found in this object in the bin_centers attribute.
• np.array – Independent continuous variables to evaluate the emulator at. If the emulator is discrete, these are only allowed to be the discrete independent variables that the emulator was trained at (disregarding the additional ‘independent’ model parameters, below). These can be found in this object in the bin_centers attribute.
• model_parameters (Dict[str, float]) – The point in model parameter space to create predicted values at.

Returns

• dependent_predictions, np.array – Array of predictions, if the emulator is a function f, these are the predicted values of f(independent) evaluated at the position of the input model_parameters.
• dependent_prediction_errors, np.array – Errors on the model predictions. For models where the errors are unconstrained, this is an array of zeroes.
**Raises** `AttributeError` – When the model has not been trained before trying to make a prediction, or when attempting to evaluate the model at disallowed independent variables.

### swiftemulator.emulators.gaussian_process_mcmc module

Gaussian Process Emulator


**Bases:** `swiftemulator.emulators.base.BaseEmulator`

Generator for emulators for individual scaling relations.

**Parameters**

- **kernel** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **george.kernels.Kernel** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **optional** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **mean_model** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **MeanModel** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **optional** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **burn_in_steps** – Optional: Number of steps used for the burn-in part of the MCMC chain. Defaults to 50 for small intial test.
- **int** – Optional: Number of steps used for the burn-in part of the MCMC chain. Defaults to 50 for small intial test.
- **optional** – Optional: Number of steps used for the burn-in part of the MCMC chain. Defaults to 50 for small intial test.
• **mcmc_steps** – Optional: Number of steps used for sampling the likelihood by the MCMC chain. Defaults to 100 for small initial tests.

• **int** – Optional: Number of steps used for sampling the likelihood by the MCMC chain. Defaults to 100 for small initial tests.

• **optional** – Optional: Number of steps used for sampling the likelihood by the MCMC chain. Defaults to 100 for small initial tests.

• **walkers** – Optional: Number of walkers used by the MCMC. Defaults to 40. Should (statistically) be at least 2 times the number of free parameters

• **int** – Optional: Number of walkers used by the MCMC. Defaults to 40. Should (statistically) be at least 2 times the number of free parameters

• **optional** – Optional: Number of walkers used by the MCMC. Defaults to 40. Should (statistically) be at least 2 times the number of free parameters

**use_hyperparameter_error, bool, optional** Switch for including errors originating from uncertain hyperparameters in the prediction outputs, (defaults to False).

**samples_for_error, int, optional** Number of MCMC samples to use for hyperparameter error estimation if **use_hyperparameter_error** is True, defaults to 100.

**kernel**: Optional[george.kernels.Kernel]

**mean_model**: Optional[swiftemulator.mean_models.base.MeanModel]

**burn_in_steps**: int

**mcmc_steps**: int

**walkers**: int

**use_hyperparameter_error**: bool

**samples_for_error**: int

**model_specification**: Optional[swiftemulator.backend.model_specification.ModelSpecification] = None

**model_parameters**: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None

**model_values**: Optional[swiftemulator.backend.model_values.ModelValues] = None

**ordering**: Optional[List[Hashable]] = None

**parameter_order**: Optional[List[str]] = None

**independent_variables**: Optional[numpy.array] = None

**dependent_variables**: Optional[numpy.array] = None

**dependent_variable_errors**: Optional[numpy.array] = None

**emulator**: Optional[george.gp.GP] = None

Fits the gaussian process model, as determined by the initialiser variables of the class (i.e. the kernel and the mean model).

**Parameters**

- **model_specification** (*ModelSpecification*) – Full instance of the model specification.
- **model_parameters** (*ModelParameters*) – Full instance of the model parameters.
- **model_values** (*ModelValues*) – Full instance of the model values describing this individual scaling relation.

**Notes**

This method uses copies of the internal kernel and mean model objects, as those objects contain slightly unhelpful state information.

**plot_hyperparameter_distribution** (*filename=None, labels=None*)

Makes a cornerplot of the MCMC samples obtained when fitting the model

**Parameters**

- **filename** – Name for the file to which the plot is saved. Optional, if None it will show the image.
- **None** – Name for the file to which the plot is saved. Optional, if None it will show the image.
- **str** – Name for the file to which the plot is saved. Optional, if None it will show the image.
- **labels** – labels to add to the different plots. Optional, if None it will take the kernel names
- **None** – labels to add to the different plots. Optional, if None it will take the kernel names
- **list[Hashable]** – labels to add to the different plots. Optional, if None it will take the kernel names

**Note:** By using this function you solemnly swear to never try to infer anything from the hyperparameters, except whether they are converged.

**predict_values** (*independent: numpy.array, model_parameters: Dict[str, float]*) → `numpy.array`

Predict values from the trained emulator contained within this object.

**Parameters**

- **independent** – Independent continuous variables to evaluate the emulator at.
- **np.array** – Independent continuous variables to evaluate the emulator at.
- **model_parameters** (*Dict[str, float]*) – The point in model parameter space to create predicted values at.

**Returns**

- **dependent_predictions, np.array** – Array of predictions, if the emulator is a function f, these are the predicted values of f(independent) evaluated at the position of the input model_parameters.
• dependent_prediction_errors, np.array – Variance on the model predictions.

swiftemulator.emulators.gaussian_process_one_dim module

Gaussian Process Emulator for emulating single values

```python

Bases: swiftemulator.emulators.base.BaseEmulator
```

Generator for emulators for individual values. In this case no independent values are used. The prediction is based on the model parameters and model values only.

Parameters

- **kernel** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George
- **george.kernels.Kernel** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George
- **optional** – The `george` kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George
- **mean_model** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).
- **MeanModel** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).
- **optional** – A mean model conforming to the `swiftemulator` mean model protocol (several pre-made models are available in the `swiftemulator.mean_models` module).

```python
kernel: Optional[george.kernels.Kernel]
mean_model: Optional[swiftemulator.mean_models.base.MeanModel]
model_specification:
Optional[swiftemulator.backend.model_specification.ModelSpecification] = None
model_parameters: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None
model_values: Optional[swiftemulator.backend.model_values.ModelValues] = None
ordering: Optional[List[Hashable]] = None
parameter_order: Optional[List[str]] = None
independent_variables: Optional[numpy.array] = None
dependent_variables: Optional[numpy.array] = None
```
dependent_variable_errors: Optional[numpy.array] = None
emulator: Optional[george.gp.GP] = None

fit_model(model_specification: swiftemulator.backend.model_specification.ModelSpecification, 
model_parameters: swiftemulator.backend.model_parameters.ModelParameters, 
model_values: swiftemulator.backend.model_values.ModelValues)

Fits the gaussian process model, as determined by the initialiser variables of the class (i.e. the kernel and the mean model).

Parameters

- model_specification (ModelSpecification) – Full instance of the model specification.
- model_parameters (ModelParameters) – Full instance of the model parameters.
- model_values (ModelValues) – Full instance of the model values describing this individual scaling relation.

Notes

This method uses copies of the internal kernel and mean model objects, as those objects contain slightly unhelpful state information.

predict_values(model_parameters: Dict[str, float]) → numpy.array

Predict a value from the trained emulator contained within this object. returns the value at the input model parameters.

Parameters model_parameters (Dict[str, float]) – The point in model parameter space to create predicted values at.

Returns

- dependent_prediction, float – Value of predictions, if the emulator is a function f, this is the predicted value of f(independent) evaluated at the position of the input model_parameters.
- dependent_prediction_error, float – Error on the model prediction.

swiftemulator.emulators.linear_model module

Linear Model Emulator

class swiftemulator.emulators.linear_model.LinearModelEmulator(lasso_model_alpha: float = 0.0)

Bases: swiftemulator.emulators.base.BaseEmulator

Emulator that builds an internal linear model (either a basic linear model or a Lasso model), and fits it to the data provided in the model specification, parameters, and values containers.

Parameters lasso_model_alpha (float) – Alpha for the Lasso model. If this is 0.0 (the default) basic linear regression is used.

lasso_model_alpha: float
ordering: Optional[List[Hashable]] = None
parameter_order: Optional[List[str]] = None
independent_variables: Optional[numpy.array] = None

dependent_variables: Optional[numpy.array] = None

dependent_variable_errors: Optional[numpy.array] = None

model_specification: Optional[swiftemulator.backend.model_specification.ModelSpecification] = None

model_parameters: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None

model_values: Optional[swiftemulator.backend.model_values.ModelValues] = None

emulator: Optional[sklearn.linear_model._base.LinearRegression] = None


Fits the linear model, given the specification, parameters, and values of the space.

Parameters

• model_specification (ModelSpecification) – Full instance of the model specification.

• model_parameters (ModelParameters) – Full instance of the model parameters.

• model_values (ModelValues) – Full instance of the model values describing this individual scaling relation.

predict_values(independent: numpy.array, model_parameters: Dict[str, float]) → numpy.array

Predict values from the trained emulator contained within this object.

Parameters

• independent – Independent continuous variables to evaluate the emulator at.

• np.array – Independent continuous variables to evaluate the emulator at.

• model_parameters (Dict[str, float]) – The point in model parameter space to create predicted values at.

Returns

• dependent_predictions, np.array – Array of predictions, if the emulator is a function f, these are the predicted values of f(independent) evaluated at the position of the input model_parameters.

• dependent_prediction_errors, np.array – Errors on the model predictions. For the linear model these are all zeroes, as the errors are unconstrained.
A gaussian process emulator that uses _multiple_ internal emulators to better predict functions that contain a ‘break’.

```python
class swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator:
    kernel: Optional[george.kernels.Kernel] = None,
    mean_model: Optional[swiftemulator.mean_models.base.MeanModel] = None,
    independent_regions: Optional[List[List[float]]] = [[None, None]]
```

**Bases:** *swiftemulator.emulators.base.BaseEmulator*

Generator for emulators for individual scaling relations, using multiple trained gaussian processes regression instances and linear models under the hood.

**Parameters**

- **kernel** – The *george* kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **george.kernels.Kernel** – The *george* kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **optional** – The *george* kernel to use. The GPE here uses a copy of this instance. By default, this is the `ExpSquaredKernel` in George

- **mean_model** – A mean model conforming to the *swiftemulator* mean model protocol (several pre-made models are available in the *swiftemulator.mean_models* module).

- **MeanModel** – A mean model conforming to the *swiftemulator* mean model protocol (several pre-made models are available in the *swiftemulator.mean_models* module).

- **optional** – A mean model conforming to the *swiftemulator* mean model protocol (several pre-made models are available in the *swiftemulator.mean_models* module).

- **independent_regions** – The regions over which to construct independent emulators. None can be used in the first and last element to specify there are no boundaries to overlap. Must be monotonically increasing. Overlaps between regions are allowed and predicted values will be a weighted linear combination of both. For example, you could use `[[None, 1.0], [0.7, 2.0], [1.7, None]]` for data that ran from 0.0 to 3.0 in the independent variable. Regions should not overlap more than once. This isn’t checked, but will break the code.

- **List[List[float]]** – The regions over which to construct independent emulators. None can be used in the first and last element to specify there are no boundaries to overlap. Must be monotonically increasing. Overlaps between regions are allowed and predicted values will be a weighted linear combination of both. For example, you could use `[[None, 1.0],`
for data that ran from 0.0 to 3.0 in the independent variable. Regions should not overlap more than once. This isn’t checked, but will break the code.

kernel: Optional[george.kernels.Kernel]

mean_model: Optional[swiftemulator.mean_models.base.MeanModel]

independent_regions: Optional[List[List[float]]]

model_specification: Optional[swiftemulator.backend.model_specification.ModelSpecification] = None

model_parameters: Optional[swiftemulator.backend.model_parameters.ModelParameters] = None

model_values: Optional[swiftemulator.backend.model_values.ModelValues] = None

model_values_regions: Optional[List[swiftemulator.backend.model_values.ModelValues]] = None

emulators: Optional[List[george.gp.GP]] = None

fit_model(model_specification: swiftemulator.backend.model_specification.ModelSpecification,
model_parameters: swiftemulator.backend.model_parameters.ModelParameters,
model_values: swiftemulator.backend.model_values.ModelValues)

Fits the gaussian process model, as determined by the initialiser variables of the class (i.e. the kernel and the mean model).

Parameters

• model_specification (ModelSpecification) – Full instance of the model specification.

• model_parameters (ModelParameters) – Full instance of the model parameters.

• model_values (ModelValues) – Full instance of the model values describing this individual scaling relation.

Notes

This method uses copies of the internal kernel and mean model objects, as those objects contain slightly unhelpful state information.

predict_values(independent: numpy.array, model_parameters: Dict[str, float]) → numpy.array

Predict values from the trained emulator contained within this object.

Parameters

• independent – Independent continuous variables to evaluate the emulator at.

• np.array – Independent continuous variables to evaluate the emulator at.

• model_parameters (Dict[str, float]) – The point in model parameter space to create predicted values at.

Returns

• dependent_predictions, np.array – Array of predictions, if the emulator is a function f, these are the predicted values of f(independent) evaluated at the position of the input model_parameters.

• dependent_prediction_errors, np.array – Errors on the model predictions.
Notes

This will use the originally defined regions and overlaps will be calculated by using the weighted linear sum corresponding to the independent variable’s distance to the adjacent boundary. The errors use a weighted square sum.

swiftemulator.io package

Submodules

swiftemulator.io.swift module

I/O functions for reading in SWIFT simulation data.

Includes functions to read parameters to instances of :class:`ModelParameters`, and functions to read model data to instances of :class:`ModelValues`.

Also includes functions to write out :class:`ModelParameters` as files, based on a base parameter file.

```
swiftemulator.io.swift.load_pipeline_outputs(filenames: Dict[Hashable, pathlib.Path],
scaling_relations: List[str], log_independent: Optional[List[str]] = None, log_dependent: Optional[List[str]] = None) → Tuple[Dict[str, swiftemulator.backend.model_values.ModelValues], Dict[str, Dict[str, Union[str, bool]]]]
```

Loads the pipeline outputs from the provided files, for the given specified scaling relations, into a :class:`ModelValues` container.

Parameters

- **filenames** – Paths to files to load data from, with the keys in the dictionary the unique identifiers for the models to use throughout.
- **Dict[Hashable] – Paths to files to load data from, with the keys in the dictionary the unique identifiers for the models to use throughout.
- **Path] – Paths to files to load data from, with the keys in the dictionary the unique identifiers for the models to use throughout.
- **scaling_relations** (List[str]) – Top-level name for the scaling relations (i.e. the top-level item in the yaml file, e.g. `stellar_mass_function_100`).
- **log_independent** (List[str], optional) – Scaling relations (the same as in `scaling_relations`) where the independent values (given by `centers` in the yaml files) should be log-scaled (uses `log10`).
- **log_dependent** (List[str], optional) – Scaling relations (the same as in `scaling_relations`) where the dependent values (given by `values` in the yaml files) should be log-scaled (uses `log10`).

Returns

- **model_values**, Dict[str, ModelValues] – Dictionary of `ModelValues` containers for each scaling relation, read from the files. The keys are the names of the scaling relations.

Chapter 6. API Documentation

Loads information from the parameter files and returns the associated model specification and model parameters instances.

**Parameters**

- **filenames** – Paths to the parameter files, keyed by their unique identifiers (i.e. those also used in load_pipeline_outputs()).
- **Dict[Hashable]** – Paths to the parameter files, keyed by their unique identifiers (i.e. those also used in load_pipeline_outputs()).
- **Path** – Paths to the parameter files, keyed by their unique identifiers (i.e. those also used in load_pipeline_outputs()).
- **parameters** – Parameters to load from the yaml files. Should be specified in the same way as the --param option in SWIFT, i.e. in the format SectionName:ParameterName.
- **List[str]** – Parameters to load from the yaml files. Should be specified in the same way as the --param option in SWIFT, i.e. in the format SectionName:ParameterName.
- **log_parameters** – Which parameters in the list above should be scaled logarithmically.
- **List[str]** – Which parameters in the list above should be scaled logarithmically.
- **optional** – Which parameters in the list above should be scaled logarithmically.
- **parameter_printable_names** – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.
- **List[str]** – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.
- **optional** – Optional ‘fancy’ names for your parameters. These strings will be used on any figures generated through swift-emulator. Can include LaTeX formatting as in matplotlib.
- **parameter_limits** – The lower and upper limit of the input parameters. Should be the same length as parameters, but each item is a list of length two, with a lower and upper bound. For example, in a two parameter model [[0.0, 1.0], [8.3, 9.3]] would mean that the first parameter would vary between 0.0 and 1.0, with the second parameter varying between 8.3 and 9.3. If not provided, these will be inferred from the data.
- **List[List[float]]** – The lower and upper limit of the input parameters. Should be the same length as parameters, but each item is a list of length two, with a lower and upper bound. For example, in a two parameter model [[0.0, 1.0], [8.3, 9.3]] would mean that the first parameter would vary between 0.0 and 1.0, with the second parameter varying between 8.3 and 9.3. If not provided, these will be inferred from the data.
- **optional** – The lower and upper limit of the input parameters. Should be the same length as parameters, but each item is a list of length two, with a lower and upper bound. For example, in a two parameter model [[0.0, 1.0], [8.3, 9.3]] would mean that the first parameter would vary between 0.0 and 1.0, with the second parameter varying between 8.3 and 9.3. If not provided, these will be inferred from the data.
Returns

- `model_specification, ModelSpecification` – Specification for the model based on the parameters that have been passed to this function.
- `model_parameters, ModelParameters` – Model parameter container corresponding to the SWIFT parameter files.

```
swiftemulator.io.swift.write_parameter_files(filenames: Dict[Hashable, pathlib.Path],
model_parameters: swiftemulator.backend.model_parameters.ModelParameters,
parameter_transforms: Optional[Dict[str, Callable]] = None,
base_parameter_file: Optional[pathlib.Path] = None)
```

Writes parameter files, containing the parameters from a `ModelParameters` instance, based on a base parameter file.

**Parameters**

- `filenames (Dict[Hashable, Path])` – Dictionary stating where to write each parameter file, based upon the unique identifiers of each run in `model_parameters`.
- `model_parameters (ModelParameters)` – Varied parameters to write in the output files.
- `parameter_transforms (Dict[str, Callable], optional)` – Parameter transformation functions for transforming parameter values before writing. Parameters may be generated (and emulated) in a space that is very different to their meaning in the code. Hence, this parameter allows for a transformation (for instance, a logarithmic transformation). For each parameter that should be transformed (keys) there should be a function taking the emulated value, transforming it into the code value. For instance, if a parameter is emulated in logarithmic space, this should be `lambda x: 10**x`.
- `base_parameter_file (Path, optional)` – Base parameter file to read. The parameters specified in `model_parameters` will be overwritten when writing each individual file, but the rest will remain the same.

**Notes**

Also changes the value of `MetaData:run_name` to the unique identifiers.

```
swiftemulator.mean_models package
```

Mean models, usually used to create a basic parametrisation of the space to enable the GPE to more efficiently model residuals.

**Submodules**

```
swiftemulator.mean_models.base module
```

A base model to be overloaded in further implementations.

```
class swiftemulator.mean_models.base.MeanModel
    Bases: object
```

A base mean model which describes the basic layout. Each function must be overloaded.
**train**(*independent: numpy.ndarray, dependent: numpy.ndarray*) → None

Trains the mean model to predict dependent(*independent*).

By convention, if there is an underlying model object, this should be stored in **self.model**.

**Parameters**

- **independent** (*np.ndarray*) – Independent variables. Should be in the same format as is passed to **george**.
- **dependent** (*np.ndarray*) – Dependent variables, to be predicted from the independent variables. Should be in the same format as is passed to **george**.

**Raises**

- **NotImplementedError** – If not implemented.

**predict**(*independent: numpy.ndarray*) → numpy.ndarray

Predicts dependent variables from independent variables, using the predictive model.

**Parameters**

- **independent** (*np.ndarray*) – Independent variables to predict dependent variables from, in the same format as is passed to **george**.

**Returns**

- **dependent** – Dependent variables that are in the same format that **george** expects.

**Return type**

- np.ndarray

**Raises**

- **NotImplementedError** – If not implemented
- **AttributeError** – If the **model** is not trained.

**property george_model**: george.modeling.CallableModel

Get the **george CallableModel** instance that corresponds to this system. Returns a copy of the predict function associated with the current instance.

**copy**() → **swiftemulator.mean_models.base.MeanModel**

Copy self to a new version of the model. Required should you wish to re-use a version of this object later on, as otherwise the **model** parameter will be mutated.

**swiftemulator.mean_models.linear module**

A basic linear model based on the scikit-learn multidimensional mean model.

**class** swiftemulator.mean_models.linear.LinearMeanModel(*lasso_model_alpha: float = 0.0*)

**Bases**: **swiftemulator.mean_models.base.MeanModel**

A linear mean model; fits a linear model to the multidimensional parameter space.

Under the hood, this uses the sklearn.linear_model.lm.

**Parameters**

- **lasso_model_alpha** (*float, optional*) – alpha for the Lasso model. If this is zero (the default) we fit a basic linear regression model is used for performance reasons.

- **lasso_model_alpha**: float

- **model**: Optional[Union[sklearn.linear_model._base.LinearRegression, sklearn.linear_model._coordinate_descent.Lasso]] = None

- **train**(*independent: numpy.ndarray, dependent: numpy.ndarray*) → None

Train the model. See **MeanModel** for more information.
predict(independent: numpy.ndarray) → numpy.ndarray
Predict using the model. See MeanModel for more information.

swiftemulator.mean_models.offset module

Offset mean model. Basic 0th order model.

class swiftemulator.mean_models.offset.OffsetMeanModel
    Bases: swiftemulator.mean_models.base.MeanModel
    A basic offset mean model. Simply takes the mean of all of the dependent variables. Not likely to be useful in practice, but more of an example of using the protocol.
    model: Optional[float] = None
    train(independent: numpy.ndarray, dependent: numpy.ndarray) → None
        Train the model. See MeanModel for more information.
    predict(independent: numpy.ndarray) → numpy.ndarray
        Predict using the model. See MeanModel for more information.

swiftemulator.mean_models.polynomial module

A basic polynomial model based on the scikit-learn multidimensional mean model.

class swiftemulator.mean_models.polynomial.PolynomialMeanModel(degree: int = 1)
    Bases: swiftemulator.mean_models.base.MeanModel
    A polynomial mean model; fits a linear model to the multidimensional parameter space.
    Under the hood, this uses the sklearn.linear_model.lm.
    Parameters degree (int, optional) – Maximal degree of the polynomial surface, default 1 (linear in each parameter).
    degree: int
    model: Optional[Union[sklearn.linear_model._base.LinearRegression, sklearn.linear_model._coordinate_descent.Lasso]] = None
    train(independent: numpy.ndarray, dependent: numpy.ndarray) → None
        Train the model. See MeanModel for more information.
    predict(independent: numpy.ndarray) → numpy.ndarray
        Predict using the model. See MeanModel for more information.

swiftemulator.mocking package

Sub-module that uses the created emulators to re-sample the parameter space completely, effectively creating a higher ‘resolution’ (in sub-grid parameters) hypercube.

swiftemulator.mocking.mock_hypercube(emulator, model_specification:
    swiftemulator.backend.model_specification.ModelSpecification,
samples: int, predict_values_kwargs: Optional[Dict[str, Any]] = None) → Tuple[swiftemulator.backend.model_values.ModelValues,
    swiftemulator.backend.model_parameters.ModelParameters]
Create a mocked version of the cube, interpolated at random points using the emulator.

**Parameters**

- **emulator** – An emulator object that provides a `predict_values` function.
- **model_spec** ([ModelSpecification](#)) – A model specification for your chosen model. The cube will be generated for points within the ranges specified here.
- **samples** (int) – Number of samples to create within your model specification.
- **predict_value_kwargs** (dict, optional) – Keyword arguments to pass to `predict_values` on the emulator object.

**Returns**

- **values** ([ModelValues](#)) – Model values container with the predictions from the provided emulator within a new cube.
- **parameters** ([ModelParameters](#)) – New model parameters generated in a random cube, corresponding to the unique identifiers in `values`.

**Notes**

The unique identifiers for the new simulations are prefixed with `emulated_` to prevent confusion when comparing with 'real' data. Samples are generated at all of the independent variable points present within the provided emulator’s data.

```python
```

Create a mocked version of the cube, interpolated evenly spaced along one dimension.

**Parameters**

- **emulator** – An emulator object that provides a `predict_values` function.
- **model_spec** ([ModelSpecification](#)) – A model specification for your chosen model. The cube will be generated for points within the ranges specified here.
- **samples** (int) – Number of samples to create within your model specification.
- **sweep_parameter** (str) – Parameter to sweep along, from minimum to maximum in the `model_spec`.
- **center_point** (str) – Model parameters for the center point of the sweep. The other model parameters will remain as these values throughout the sweep.
- **predict_value_kwargs** (dict, optional) – Keyword arguments to pass to `predict_values` on the emulator object.

**Returns**

- **values** ([ModelValues](#)) – Model values container with the predictions from the provided emulator within a new cube.
- **parameters** ([ModelParameters](#)) – New model parameters generated in a sweep, corresponding to the unique identifiers in `values`.
Notes

The unique identifiers for the new simulations are prefixed with emulated_ to prevent confusion when comparing with 'real' data. Samples are generated at all of the independent variable points present within the provided emulator’s data.

swiftemulator.sensitivity package

Submodules

swiftemulator.sensitivity.basic module

Basic sensitivity analysis based purely on the model values at consistent values in the space. No emulation is used to determine the sensitivity.

A different sensitivity analysis is ran for each dependent variable, so it is important to ensure that the functions are evaluated at consistent values.


Creates a binwise sensitivity analysis dictionary.

For each bin in dependent variable a hash is created; these are the keys in the returned dictionary.

Parameters

- **specification** (ModelSpecification) – Model spec; parameter limits must be valid as these feed into the sensitivity analysis.
- **parameters** (ModelParameters) – Parameters; these feed in as the independent variables in the sensitivity analysis.
- **values** (ModelValues) – Dependent variables in the sensitivity analysis.

Returns sensitivity – Binwise sensitivity analysis, with each array corresponding to the parameters in the order as specified by the specification. This is the “S1” vector from the RBD FAST method.

Return type  Dict[str, np.array]

Create a figure and axis displaying the output of the sensitivity analysis.

swiftemulator.sensitivity.cross_check module

Basic test for any emulator. Leave one simulation out and test how well the emulator fits those values.


Bases: object

Generator for emulators for leave one out checks.

Parameters

- **kernel** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **george.kernels** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **mean_model** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **MeanModel** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **optional** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **hide_progress** (bool) – Option to display a tqdm bar when creating the emulators, Default is to hide progress bar.

kernel: Optional[george.kernels.Kernel]
mean_model: Optional[swiftemulator.mean_models.base.MeanModel]
hide_progress: bool
model_specification: swiftemulator.backend.model_specification.ModelSpecification
model_parameters: swiftemulator.backend.model_parameters.ModelParameters
model_values: `swiftemulator.backend.model_values.ModelValues`

leave_out_order: Optional[List[int]] = None

cross_emulators: Optional[Dict[Hashable, george.gp.GP]] = None


Build a dictionary with an emulator for each simulation where the data of that simulation is left out

Note: this can take a long time

Parameters

- **model_specification** (*ModelSpecification*) – Full instance of the model specification.
- **model_parameters** (*ModelParameters*) – Full instance of the model parameters.
- **model_values** (*ModelValues*) – Full instance of the model values describing this individual scaling relation.

build_mocked_model_values_original_independent() → `swiftemulator.backend.model_values.ModelValues`

" Builds a mocked `ModelValues` container, using the cross emulators. The emulators are evaluated at the same independent variables that were ‘left out’.

**Returns model_values** – The model values container with each leave-one-out scaling relation predicted. This is also set as `cross_model_values`.

**Return type** `ModelValues`

build_mocked_model_values(*emulate_at: numpy.array*) → `swiftemulator.backend.model_values.ModelValues`

Builds a mocked `ModelValues` container, using the cross emulators. Similar to `build_mocked_model_value_original_independent` but evaluates all emulators at a consistent set of independent variables.

Parameters **emulate_at** (*np.array*) – independent array where the emulator is evaluated.

**Returns model_values** – The model values container with each leave-one-out scaling relation predicted. This is also set as `cross_model_values`.

**Return type** `ModelValues`

plot_results(*emulate_at: numpy.array, output_path: Optional[Union[pathlib.Path, str]] = None, xlabel: Optional[str] = None, ylabel: Optional[str] = None*)

Make a plot of each of the leave_out emulators vs the original data.

Parameters

- **emulate_at** (*np.array*) – independent array where the emulator is evaluated.
- **output_path** (*Union[str, Path], optional*) – Optional, name of the folder where you want to save the figures.
- **xlabel** (*str, optional*) – Label for horizontal axis on the resultant figure.
- **ylabel** (*str, optional*) – Label for vertical axis on the resultant figure.
get_mean_squared(
    use_dependent_error: bool = False,
    use_y_as_error: bool = False,
    use_squared_difference: bool = True)

Calculates the mean squared per simulation and the total mean squared of the entire set of left-out simulations.

Parameters

- **use_dependent_error** (boolean) – Use the simulation errors as weights for the mean squared calculation. Default is false.
- **use_y_as_error** (boolean) – Use the model y values as the weights for the calculation.
- **use_squared_difference** (boolean) – Use the simulation errors as weights for the mean squared calculation. Default is false.

Returns

- **total_square_mean** (float) – Mean (square) error across the bins.
- **mean_squared_dict** (Dict[Hashable, float]) – Error per unique identifier.

swiftemulator.sensitivity.cross_check_bins module

Basic test for any emulator. Leave one simulation out and test how well the emulator fits those values.

class swiftemulator.sensitivity.cross_check_bins.CrossCheckBins(
    kernel: Optional[george.kernels.Kernel] = None,
    mean_model: Optional[swiftemulator.mean_models.base.MeanModel] = None,
    hide_progress: bool = True)

Bases: object

Generator for emulators for leave one out checks.

Parameters

- **kernel** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **george.kernels** – The george kernel to use. The GPE here uses a copy of this instance. By default, this is the ExpSquaredKernel in George
- **mean_model** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **MeanModel** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **optional** – A mean model conforming to the swiftemulator mean model protocol (several pre-made models are available in the swiftemulator.mean_models module).
- **hide_progress** (bool) – Option to display a tqdm bar when creating the emulators, Default is to hide progress bar.

kernel: Optional[george.kernels.Kernel]
mean_model: Optional[swiftemulator.mean_models.base.MeanModel]
hide_progress: bool
model_specification:  `swiftemulator.backend.model_specification.ModelSpecification`
model_parameters:  `swiftemulator.backend.model_parameters.ModelParameters`
model_values:  `swiftemulator.backend.model_values.ModelValues`
leave_out_order:  Optional[List[int]] = None
cross_emulators:  Optional[Dict[Hashable, george.gp.GP]] = None

build_emulators(model_specification: swiftemulator.backend.model_specification.ModelSpecification,
model_parameters: swiftemulator.backend.model_parameters.ModelParameters,
model_values: swiftemulator.backend.model_values.ModelValues)

Build a dictionary with an emulator for each simulation where the data of that simulation is left out

Note: this can take a long time

Parameters

• `model_specification` (ModelSpecification) – Full instance of the model specification.

• `model_parameters` (ModelParameters) – Full instance of the model parameters.

• `model_values` (ModelValues) – Full instance of the model values describing this individual scaling relation.

plot_results(output_path: Optional[Union[pathlib.Path, str]] = None, xlabel: Optional[str] = None, ylabel: Optional[str] = None)

Make a plot of each of the leave_out emulators vs the original data.

Parameters

• `output_path` (Union[str, Path], optional) – Optional, name of the folder where you want to save the figures.

• `xlabel` (str, optional) – Label for horizontal axis on the resultant figure.

• `ylabel` (str, optional) – Label for vertical axis on the resultant figure.

get_mean_squared(use_dependent_error: bool = False, use_y_as_error: bool = False, use_squared_difference: bool = True)

Calculates the mean squared per simulation and the total mean squared of the entire set of left-out simulations.

Parameters

• `use_dependent_error` (bool) – Use the simulation errors as weights for the mean squared calculation. Default is false.

• `use_y_as_error` (boolean) – Use the model y values as the weights for the calculation.

• `use_squared_difference` (boolean) – Use the simulation errors as weights for the mean squared calculation. Default is false.

Returns

• `total_square_mean` (float) – Mean (square) error across the bins.

• `mean_squared_dict` (Dict[Hashable, float]) – Error per unique identifier.
CHAPTER
SEVEN

INDICES AND TABLES

• genindex
• modindex
• search
INDICES AND TABLES

- genindex
- modindex
- search
## PYTHON MODULE INDEX

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>swiftemulator</td>
<td>29</td>
</tr>
<tr>
<td>swiftemulator.backend</td>
<td>29</td>
</tr>
<tr>
<td>swiftemulator.backend.emulator_generator</td>
<td>29</td>
</tr>
<tr>
<td>swiftemulator.backend.model_parameters</td>
<td>31</td>
</tr>
<tr>
<td>swiftemulator.backend.model_specification</td>
<td>33</td>
</tr>
<tr>
<td>swiftemulator.backend.model_values</td>
<td>34</td>
</tr>
<tr>
<td>swiftemulator.comparison</td>
<td>36</td>
</tr>
<tr>
<td>swiftemulator.comparison.penalty</td>
<td>36</td>
</tr>
<tr>
<td>swiftemulator.comparison.visualisation</td>
<td>47</td>
</tr>
<tr>
<td>swiftemulator.design</td>
<td>49</td>
</tr>
<tr>
<td>swiftemulator.design.latin</td>
<td>49</td>
</tr>
<tr>
<td>swiftemulator.design.random</td>
<td>50</td>
</tr>
<tr>
<td>swiftemulator.design.transform</td>
<td>50</td>
</tr>
<tr>
<td>swiftemulator.emulators</td>
<td>51</td>
</tr>
<tr>
<td>swiftemulator.emulators.base</td>
<td>51</td>
</tr>
<tr>
<td>swiftemulator.emulators gaussian_process</td>
<td>52</td>
</tr>
<tr>
<td>swiftemulator.emulators gaussian_process_bins</td>
<td>54</td>
</tr>
<tr>
<td>swiftemulator.emulators gaussian_process_mcmc</td>
<td>56</td>
</tr>
<tr>
<td>swiftemulator.emulators gaussian_process_one_dim</td>
<td>59</td>
</tr>
<tr>
<td>swiftemulator.emulators.linear_model</td>
<td>60</td>
</tr>
<tr>
<td>swiftemulator.emulators.multi_gaussian_process</td>
<td>62</td>
</tr>
<tr>
<td>swiftemulator.io</td>
<td>64</td>
</tr>
<tr>
<td>swiftemulator.io.swift</td>
<td>64</td>
</tr>
<tr>
<td>swiftemulator.mean_models</td>
<td>66</td>
</tr>
<tr>
<td>swiftemulator.mean_models.base</td>
<td>66</td>
</tr>
<tr>
<td>swiftemulator.mean_models.linear</td>
<td>67</td>
</tr>
<tr>
<td>swiftemulator.mean_models.offset</td>
<td>68</td>
</tr>
<tr>
<td>swiftemulator.mean_models.polynomial</td>
<td>68</td>
</tr>
<tr>
<td>swiftemulator.mocking</td>
<td>68</td>
</tr>
<tr>
<td>swiftemulator.sensitivity</td>
<td>70</td>
</tr>
<tr>
<td>swiftemulator.sensitivity.basic</td>
<td>70</td>
</tr>
<tr>
<td>swiftemulator.sensitivity.cross_check</td>
<td>71</td>
</tr>
<tr>
<td>swiftemulator.sensitivity.cross_check_bins</td>
<td>73</td>
</tr>
</tbody>
</table>
INDEX

B

BaseEmulator (class in swiftemulator.emulators.base), 51

bin_centers (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 55

bin_gaussian_process (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 55

bin_model_values (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 55

binwise_sensitivity() (in module swiftemulator.sensitivity.basic), 70

build_emulators() (swiftemulator.sensitivity.cross_check.CrossCheck method), 72

build_emulators() (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins method), 74

build_emulators() (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins method), 72

build_mocked_model_values() (swiftemulator.sensitivity.cross_check.CrossCheck method), 72

build_mocked_model_values_original_independent() (swiftemulator.sensitivity.cross_check.CrossCheck method), 72

burn_in_steps (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57

C

copy() (swiftemulator.mean_models.base.MeanModel method), 67

create_cube() (in module swiftemulator.design.random), 50

create_gaussian_process_emulator() (swiftemulator.backend.emulator_generator.EmulatorGenerator method), 29

create_gaussian_process_emulator_bins() (swiftemulator.backend.emulator_generator.EmulatorGenerator method), 30

create_gaussian_process_emulator_mcmc() (swiftemulator.backend.emulator_generator.EmulatorGenerator method), 30

cross_emulators (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 72

cross_emulators (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 74

CrossCheck (class in swiftemulator.sensitivity.cross_check), 71

CrossCheckBins (class in swiftemulator.sensitivity.cross_check_bins), 73

D

degree (swiftemulator.mean_models.polynomial.PolynomialMeanModel attribute), 68

dependent_units (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36

dependent_variable_errors (swiftemulator.emulators.base.BaseEmulator attribute), 51

dependent_variable_errors (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53

dependent_variable_errors (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 55

dependent_variable_errors (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57

dependent_variable_errors (swiftemulator.emulators.gaussian_process_one_dint.GaussianProcessEmulator attribute), 59

dependent_variable_errors (swiftemulator.emulators.linear_model.LinearModelEmulator attribute), 61
### SWIFT-Emulator, Release 1.2.0

<table>
<thead>
<tr>
<th>dependent_variables</th>
<th>fit_model()</th>
<th>from_json()</th>
<th>from_yaml()</th>
</tr>
</thead>
<tbody>
<tr>
<td>(swiftemulator.emulators.base.BaseEmulator attribute)</td>
<td>(swiftemulator.emulators.base.BaseEmulator method), 51</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 35</td>
<td></td>
</tr>
<tr>
<td>(swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator attribute)</td>
<td>(swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator method), 60</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 33</td>
<td></td>
</tr>
<tr>
<td>(swiftemulator.emulators.linear_model.LinearModelEmulator attribute)</td>
<td>(swiftemulator.emulators.linear_model.LinearModelEmulator method), 61</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 33</td>
<td></td>
</tr>
<tr>
<td>(swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute)</td>
<td>(swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator method), 63</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 35</td>
<td></td>
</tr>
<tr>
<td>(swiftemulator.backend.model_parameters.ModelParameters property)</td>
<td>(swiftemulator.backend.model_parameters.ModelParameters class method), 33</td>
<td>(swiftemulator.backend.model_parameters.ModelParameters class method), 33</td>
<td></td>
</tr>
<tr>
<td>(swiftemulator.backend.model_values.ModelValues class)</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 35</td>
<td>(swiftemulator.backend.model_values.ModelValues class method), 35</td>
<td></td>
</tr>
</tbody>
</table>

### GaussianDataErrorsPenaltyCalculator

- GaussianDataErrorsPenaltyCalculator (class in swiftemulator.comparison.penalty), 42
- GaussianPercentErrorsPenaltyCalculator (class in swiftemulator.comparison.penalty), 44

### GaussianProcessEmulator

- GaussianProcessEmulator (class in swiftemulator.emulators.gaussian_process), 52
- GaussianProcessEmulatorBins (class in swiftemulator.emulators.gaussian_process_bins), 59
- GaussianProcessEmulatorMCMC (class in swiftemulator.emulators.gaussian_process_mcmc), 56

### GaussianWeightedDataErrorsPercentFloorPenaltyCalculator

- GaussianWeightedDataErrorsPercentFloorPenaltyCalculator (class in swiftemulator.comparison.penalty), 45

### HigherEmulator

- george_model (swiftemulator.mean_models.base.MeanModel property), 67
get_mean_squared() (swiftemulator.sensitivity.cross_check.CrossCheck method), 72
get_mean_squared() (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins method), 74

H
hide_progress (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 71
hide_progress (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 73

I
independent_regions (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute), 63
independent_units (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36
independent_variables (swiftemulator.emulators.base.BaseEmulator attribute), 51
independent_variables (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53
independent_variables (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 54
independent_variables (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57
independent_variables (swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator1D attribute), 59
independent_variables (swiftemulator.emulators.linear_model.LinearModelEmulator attribute), 60
independent_variables (swiftemulator.mean_models.linear.LinearMeanModel attribute), 67
independent_variables (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 72
independent_variables (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 74

interpolator_error (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36
interpolator_values (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36

items() (swiftemulator.backend.model_parameters.ModelParameters method), 31
items() (swiftemulator.backend.model_values.ModelValues method), 35

K
kernel (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53
kernel (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 54
kernel (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57
kernel (swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator1D attribute), 59
kernel (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute), 63
kernel (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 71
kernel (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 73
keys() (swiftemulator.backend.model_parameters.ModelParameters method), 31
keys() (swiftemulator.backend.model_values.ModelValues method), 35

L
L1PenaltyCalculator (class in swiftemulator.comparison.penalty), 37
L1PenaltyCalculatorOneSided (class in swiftemulator.comparison.penalty), 38
L1SqueezePenaltyCalculator (class in swiftemulator.comparison.penalty), 41
L1VariablePenaltyCalculator (class in swiftemulator.comparison.penalty), 40
L2PenaltyCalculator (class in swiftemulator.comparison.penalty), 39
lasso_model_alpha (swiftemulator.linear_model.LinearModelEmulator attribute), 60
lasso_model_alpha (swiftemulator.mean_models.linear.LinearMeanModel attribute), 67
leave_out_order (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36
leave_out_order (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 72
leave_out_order (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 74
LinearMeanModel (class in swiftemulator.mean_models.linear), 67
LinearModelEmulator (class in swiftemulator.emulators.linear_model), 60
load_parameter_files() (in module swiftemulator.io.swift), 64
load_pipeline_outputs() (in module swiftemulator.io.swift), 64
log_dependent (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36
log_independent (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36

Index
lower (swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator.attribute), 43
lower (swiftemulator.comparison.penalty.GaussianDataErrorsPercentFloorPenaltyCalculator.attribute), 45
lower (swiftemulator.comparison.penalty.GaussianPercentErrorsPenaltyCalculator.attribute), 44
lower (swiftemulator.comparison.penalty.GaussianWeightedDataErrorsPercentFloorPenaltyCalculator.attribute), 46
lower (swiftemulator.comparison.penalty.L1PenaltyCalculator.attribute), 38
lower (swiftemulator.comparison.penalty.L1PenaltyCalculatorOneSided.attribute), 39
lower (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator.attribute), 42
lower (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator.attribute), 41
lower (swiftemulator.comparison.penalty.L2PenaltyCalculator.attribute), 39
M
maximum_penalty (swiftemulator.comparison.penalty.L1PenaltyCalculatorOneSided.attribute), 39
mcmc_steps (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC.attribute), 57
mean_model (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator.attribute), 53
mean_model (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins.attribute), 54
mean_model (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC.attribute), 57
mean_model (swiftemulator.emulators.linear_model.LinearModelEmulator.attribute), 61
mean_model (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator.attribute), 63
mean_model (swiftemulator.sensitivity.cross_check.CrossCheck.attribute), 71
MeanModel (class in swiftemulator.mean_models.base), 66
mock_hypercube() (in module swiftemulator.mocking), 68
mock_sweep() (in module swiftemulator.mocking), 69
model (swiftemulator.mean_models.linear.LinearMeanModel.attribute), 67
model_parameters (swiftemulator.backend.emulator_generator.EmulatorGenerator.attribute), 29
model_parameters (swiftemulator.backend.model_parameters.ModelParameters.attribute), 31
model_parameters (swiftemulator.emulators.base.BaseEmulator.attribute), 51
model_parameters (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator.attribute), 53
model_parameters (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins.attribute), 54
model_parameters (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC.attribute), 57
model_parameters (swiftemulator.emulators.linear_model.LinearModelEmulator.attribute), 61
model_parameters (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator.attribute), 63
model_parameters (swiftemulator.sensitivity.cross_check.CrossCheck.attribute), 71
model_specification (swiftemulator.backend.emulator_generator.EmulatorGenerator.attribute), 29
model_specification (swiftemulator.emulators.base.BaseEmulator.attribute), 51
model_specification (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator.attribute), 53
model_specification (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins.attribute), 54
model_specification (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC.attribute), 57
model_specification (swiftemulator.emulators.linear_model.LinearModelEmulator.attribute), 61
model_specification (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator.attribute), 63
model_specification (swiftemulator.sensitivity.cross_check.CrossCheck.attribute), 71

84 Index
model_specification (swiftemulator.emulators.linear_model.LinearModelEmulator attribute), 61
model_specification (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute), 63
model_specification (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 71
model_specification (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 73
model_values (swiftemulator.backend.model_values.ModelValues attribute), 35
model_values (swiftemulator.emulators.base.BaseEmulator attribute), 51
model_values (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53
model_values (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 54
model_values (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57
model_values (swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator1D attribute), 59
model_values (swiftemulator.emulators.linear_model.LinearModelEmulator attribute), 60
model_values (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute), 62
model_values (swiftemulator.sensitivity.cross_check.CrossCheck attribute), 71
model_values (swiftemulator.sensitivity.cross_check_bins.CrossCheckBins attribute), 74
model_values_regions (swiftemulator.emulators.multi_gaussian_process.MultipleGaussianProcessEmulator attribute), 63

ModelParameters (class in swiftemulator.backend.model_parameters), 31
ModelSpecification (class in swiftemulator.backend.model_specification), 33
ModelValues (class in swiftemulator.backend.model_values), 34
module swiftemulator, 29

n_bins (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 55
number_of_parameters (swiftemulator.backend.model_specification.ModelSpecification attribute), 34
number_of_variables (swiftemulator.backend.model_values.ModelValues attribute), 34
property), 35

observation (swiftemulator.comparison.penalty.PenaltyCalculator attribute), 36
observation_interpolation() (swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator method), 43
observation_interpolation() (swiftemulator.comparison.penalty.GaussianDataErrorsPercentFloorPenaltyCalculator method), 45
observation_interpolation() (swiftemulator.comparison.penalty.GaussianPercentErrorsPenaltyCalculator method), 44
observation_interpolation() (swiftemulator.comparison.penalty.GaussianWeightedDataErrorsPercentFloorPenaltyCalculator method), 46
observation_interpolation() (swiftemulator.comparison.penalty.L1PenaltyCalculator method), 38
observation_interpolation() (swiftemulator.comparison.penalty.L1PenaltyCalculatorOneSided method), 39
observation_interpolation() (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator method), 42
observation_interpolation() (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator method), 41
offset (swiftemulator.comparison.penalty.L1PenaltyCalculator attribute), 38
offset (swiftemulator.comparison.penalty.L1PenaltyCalculatorOneSided attribute), 39
offset (swiftemulator.comparison.penalty.L2PenaltyCalculator attribute), 40
offset_squeeze (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator attribute), 42
offset_transition (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator attribute), 42
offset_upper (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator attribute), 41
offset_transition (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator attribute), 41
offset_below_above_ratio (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator attribute), 42
offset_below_above_ratio (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator attribute), 41
offset_lower (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator attribute), 41
offset_normal (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator attribute), 42
offset_normal (swiftemulator.comparison.penalty.L1VariablePenaltyCalculator attribute), 41
OffsetMeanModel (class in swiftemulator.mean_models.offset), 68
ordering (swiftemulator.emulators.base.BaseEmulator attribute), 51
ordering (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53
ordering (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 54
ordering (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57
ordering (swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator attribute), 59
ordering (swiftemulator.emulators.linear_model.LinearModelEmulator attribute), 60
parameter_limits (swiftemulator.backend.model_specification.ModelSpecification attribute), 34
parameter_names (swiftemulator.backend.model_specification.ModelSpecification attribute), 34
parameter_order (swiftemulator.emulators.base.BaseEmulator attribute), 51
parameter_order (swiftemulator.emulators.gaussian_process.GaussianProcessEmulator attribute), 53
parameter_order (swiftemulator.emulators.gaussian_process_bins.GaussianProcessEmulatorBins attribute), 54
parameter_order (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57
parameter_order (swiftemulator.emulators.gaussian_process_one_dim.GaussianProcessEmulator attribute), 59
parameter_order  (swiftemulator.emulators.linear_model.LinearModelEmulator.plot_model()  (swiftemulator.backend.model_parameters.ModelParameters attribute), 60

parameter_printable_names  (swiftemulator.backend.model_specification.ModelSpecification.plot_penalty()  (swiftemulator.comparison.penalty.PenaltyCalculator method), 32

penalties()  (swiftemulator.comparison.penalty.PenaltyCalculator.plot_penalty()  (swiftemulator.comparison.penalty.PenaltyCalculator method), 37

penalty()  (swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator.plot_results()  (swiftemulator.sensitivity.cross_check.CrossCheck method), 72

penalty()  (swiftemulator.comparison.penalty.GaussianPercentErrorsPenaltyCalculator.plot_results()  (swiftemulator.sensitivity.cross_check.Bins.CrossCheckBins method), 74

penalty()  (poly_model (class in swiftemulator.mean_models.polynomial), 68

penalty()  (predict()  (swiftemulator.mean_models.base.MeanModel method), 67

penalty()  (predict()  (swiftemulator.mean_models.linear.LinearMeanModel method), 67

penalty()  (predict()  (swiftemulator.mean_models.offset.OffsetMeanModel method), 68

penalty()  (predict()  (swiftemulator.mean_models.polynomial.PolynomialMeanModel method), 68

penalty()  (predict()  (swiftemulator.emulators.gaussian_process.MultipleGaussianProcessEmulator method), 63

plot_binwise_sensitivity()  (plot_binwise_sensitivity()  (swiftemulator.sensitivity.basic), 70

plot_hyperparameter_distribution()  (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC register_observation()  (swiftemulator.comparison.penalty.PenaltyCalculator method), 36

Index 87
SWIFT-Emulator, Release 1.2.0

S

salib_problem
(swiftemulator.backend.model_specification.ModelSpecification
property), 34

samples_for_error
(swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC
attribute), 57

sigma_max
(swiftemulator.comparison.penalty.GaussianDataErrorsPenaltyCalculator
attribute), 43

sigma_max
(swiftemulator.comparison.penalty.GaussianDataErrorsPercentFloorPenaltyCalculator
attribute), 45

sigma_max
(swiftemulator.comparison.penalty.GaussianPercentErrorsPenaltyCalculator
attribute), 44

sigma_max
(swiftemulator.comparison.penalty.GaussianWeightedDataErrorsPercentFloorPenaltyCalculator
attribute), 46

swiftemulator
module, 29

swiftemulator.backend
module, 29

swiftemulator.backend.emulator_generator
module, 29

swiftemulator.backend.model_parameters
module, 31

swiftemulator.backend.model_specification
module, 33

swiftemulator.backend.model_values
module, 34

swiftemulator.comparison
module, 36

swiftemulator.comparison.penalty
module, 36

swiftemulator.comparison.visualisation
module, 47

swiftemulator.design
module, 49

swiftemulator.design.latin
module, 49

swiftemulator.design.random
module, 50

swiftemulator.design.transform
module, 50

swiftemulator.emulators
module, 51

swiftemulator.emulators.base
module, 51

swiftemulator.emulators.gaussian_process
module, 52

swiftemulator.emulators.gaussian_process_bins
module, 54

swiftemulator.emulators.gaussian_process_mcmc
module, 56

swiftemulator.emulators.gaussian_process_one_dim
module, 59

swiftemulator.emulators.linear_model
module, 60

swiftemulator.emulators.multi_gaussian_process
module, 62

swiftemulator.io
module, 64

swiftemulator.io.swift
module, 64

swiftemulator.mean_models
module, 66

swiftemulator.mean_models.base
module, 66

swiftemulator.mean_models.linear
module, 67

swiftemulator.mean_models.offset
module, 68

swiftemulator.mean_models.polynomial
module, 68

swiftemulator.mocking
module, 68

swiftemulator.sensitivity
module, 70

swiftemulator.sensitivity.basic
module, 70

swiftemulator.sensitivity.cross_check
module, 71

swiftemulator.sensitivity.cross_check_bins
module, 73

T

to_json()
(swiftemulator.backend.model_parameters.ModelParameters
method), 33

to_json()
(swiftemulator.backend.model_values.ModelValues
method), 35

to_yaml()
(swiftemulator.backend.model_parameters.ModelParameters
method), 32

to_yaml()
(swiftemulator.backend.model_values.ModelValues
method), 35

train()
(swiftemulator.mean_models.base.MeanModel
method), 66

train()
(swiftemulator.mean_models.linear.LinearMeanModel
method), 67

train()
(swiftemulator.mean_models.offset.OffsetMeanModel
method), 68

train()
(swiftemulator.mean_models.polynomial.PolynomialMeanModel
method), 68
transform_to_model_spec() (in module swiftemulator.design.transform), 50

transition_width (swiftemulator.comparison.penalty.L1SqueezePenaltyCalculator attribute), 42

visualise_penalties_generic_statistic() (in module swiftemulator.comparison.visualisation), 48

visualise_penalties_mean() (in module swiftemulator.comparison.visualisation), 47

use_hyperparameter_error (swiftemulator.emulators.gaussian_process_mcmc.GaussianProcessEmulatorMCMC attribute), 57

values() (swiftemulator.backend.model_parameters.ModelParameters method), 31

values() (swiftemulator.backend.model_values.ModelValues method), 35

write_parameter_files() (in module swiftemulator.io.swift), 66