morpho Documentation

Release v2.3.2-0-g65a74ca

The Project 8 Collaboration

Apr 17, 2019

Contents

Contents:

CHAPTER 1

What's New

Morpho version 2 is live! You can have a look at how this works in the "Morpho 2" pages.

No future updates are planned for Morpho 1, but Morpho 1 information is still included at the end of the documentation.

CHAPTER 2

Introduction

Morpho is an analysis tool that organizes data inflow to and outflow from Stan, a platform for Bayesian statistical modeling and computation, and RooFit, a toolkit for modeling probability distributions.

It is especially useful for

1) Generating **pseudo data**, and
2) Performing **Bayesian statistical analyses** of real or fake data—that is, extracting posterior distributions for parameters of interest using data and a model.
3) Performing **chi2 fits of data**.

Morpho interfaces with Stan using PyStan, but it is designed to be employed by general Stan users (not only PyStan users).

For more information, also see:

Stan: http://mc-stan.org

PyStan: https://pystan.readthedocs.io/en/latest/index.html

RooFit: https://root.cern.ch/guides/roofit-manual

2.1 Why morpho?

- Morpho **streamlines analyses**. It enables users to load data, run Stan or RooFit, save results, perform convergence diagnostic tests, and create plots of posteriors and their correlations—all as part of one individual analysis. Users can control some or all of these processes using a single configuration file.
- Morpho helps users organize and run multiple related Stan models (for example, models that share input data and Stan functions).
- Morpho **minimizes the need to recompile** Stan models by using cache files.
- Morpho automatically **performs convergence checks** after running Stan, and it provides additional options for convergence analysis and plotting.

- Morpho reads and saves files in either **R**, **JSON/YAML**, **CVS**, or **ROOT**.

2.2 Stan vs Roofit

Stan uses a Hamiltonian Markov Chain Monte Carlo (HMCMC) algorithm in order to explore probability distributions. HMCMC uses the geometry of a distribution in order to efficiently explore distributions with a large number of parameters. Using HMCMC, however, places the constraint that all elements of the probability distribution must be expressed analytically.

RooFit is a toolkit to represent probability distributions and perform simple fits, such as unbinned maximum likelihood fits, or simple MCMC functionality with the Metropolis-Hastings algorithm. RooFit places less constraints on the form of the probability distribution, and is capable of interfacing with PDFs generated by simulations or other external code.

In general, Stan will be more efficient, and is capable of working with a very large number of parameters. RooFit is slower, but it offers more flexibility in the way probability distributions can be defined.

CHAPTER 3

Morpho 2: a new framework

Morpho is an analysis framework based on the Stan/PyStan Markov Chain Monte Carlo package and the ROOT/RooFit C++ library.

Similarly to Morpho 1, Morpho 2 is intended as a meta-analysis tool to fit or generate data, organize inflow and outflow of data and models.

3.1 A new underlying framework

Morpho 2 uses a framework similar to Nymph: it uses classes called *processors* to act on the data. All classes inherites from a *BaseProcessor* class where all the common behaviors are encoded. However the exchange of informations between processors is less constraint than the Katydid implementation of Nymph. The *output* of a processor is contained into an internal variable of the processor, and is generally a dictionary.

The connection between processors is usually defined into a configuration file, but can be done *manually* using the morpho python API. An example of both implementation can be found here *Example*.

3.2 An extensible module

Morpho is intended to be a generic analysis framework. It contains processors that users can find useful, regardless of their field. Suggestions of new processors and features are welcome and can be submitted via issue posting on Github.

When processors are needed by users for a specific processor (e.g. a processor that reads files with a specific formatting), it is recommended to set these into an *extension*. Extensions would then contain all the processors and be installed along with morpho and used via the main *morpho* executable which would look for the needed processors.

An example of such extension is mermithid: it contains processors related to the file formatting needed by the Project 8 collaboration. It also implements RootFit sampling and fitting processors that makes use of custom beta decay spectrum shapes. The associated pdf are compiled (via CMake) and the libraries appended to the *PYTHONPATH* before the installation of the module Finally a plotting processor (generating Kurie plots) specific to this experiment is kept there.

3.3 An interface with external software

Thanks to this new framework and the extensitvity of the package, it is easy to interface with other softwares. Several ways of implementing such interfacing are possible and should be implemented depending on how complex the interfacing is:

1. If the new piece of code is contain into a simple function into a python script, one can use as a first step the ProcessorAssistant to wrap the function into a processor (this does require the creation of an extension). Eventually, for production usage, a new processor with the desired behavior should be created (this might require the creation of an extension).
2. If morpho needs to interface with an external library (e.g. some C++ code), an extension is highly recommended. The libraries can be built before the installation of the extension. An example of such implementation is mermithid.

CHAPTER 4

Install

4.1 Dependencies

The following dependencies should be installed (via a package manager) before installing morpho:

- python 3.x (python 2 not supported)
- python-pip
- git
- root (ensure that the same version of python is enabled for morpho and ROOT)

4.2 Virtual environment-based installation

We recommend installing morpho using pip inside a python virtual environment. Doing so will automatically install dependencies beyond the four listed above, including PyStan 2.15.

If necessary, install virtualenv, then execute:

```
# Use a flag for virtualenv to specify python3 if necessary: --python /path/to/python3
virtualenv ~/path/to/the/virtualenvironment
source ~/path/to/the/virtualenvironment/bin/activate #Activate the environment
pip install -U pip #Update pip to >= 7.0.0
cd ~/path/to/morpho
pip install .
# When done with morpho, use "bash deactivate" to exit the virtual environment
```

4.3 Docker installation

If you would like to modify your local installation of morpho (to add features or resolve any bugs), we recommend you use a Docker container instead of a python virtual environment. To do so:

1. Install Docker: https://docs.docker.com/engine/installation/.
2. Clone and pull the latest master version of morpho.
3. Inside the morpho folder, execute `` `docker-compose run morpho` ``. A new terminal prompter (for example, `` `root@413ab10d7a8f:` ``) should appear. You may make changes to morpho either inside or outside of the Docker container. If you wish to work outside of the container, move morpho to the `` `morpho_share` `` directory that is mounted under the `` `/host` `` folder created by docker-compose. Once inside the container, run `` `source /setup.sh` `` to be able to access morpho libraries.
4. You can remove the container image using `` `docker rmi morpho_morpho` ``.
5. If the morpho Docker image gets updated, you can update the morpho image using `` `docker pull morpho` ``.

If you develop new features or identify bugs, please open a GitHub issue.

CHAPTER 5

Use

5.1 Configuration Files

Morpho primarly reads a **configuration file** (.json or .yaml) written by the user (it can also be used via the python interface). The file defines the actions ("processors") the user wants to perform and the order in which these should be done. The file also specifies input parameters that the user may wish to change on a run-to-run basis, such as the desired number of Stan iterations, or Stan initialization and data-block values.

See this example and morpho's documentation for more information.

We recommend modeling the organization of your configuration files, Stan models and data files after the **examples** folder in morpho. Your directory structure should be of the form:

```
examples
|
+---functions_dir
|   |
|   +---Stan_funcs1.functions
|   +---Stan_funcs2.functions
|   +---Stan_funcs3.functions
|
+---analysis_dir1
|   |
|   +---data_dir
|   |   |
|   |   +---fileA.data
|   |   +---fileB.data
|   |
|   +---model_dir
|   |   |
|   |   +---modelA.stan
|   |   +---modelB.stan
|   |
|   +---scripts_dir
```

(continues on next page)

(continued from previous page)

```
|       |
|       +---configA.yaml
|       +---configB.yaml
|
+---analysis_dir2
|   |
```

The files in the optional `functions_dir` directory contain Stan functions (written in the Stan language) that are used in multiple Stan models.

5.2 Running Morpho

5.2.1 Using config files

Once the relevant data, model and configuration files are at your disposal, run morpho by executing:

```
morpho --config  /path/to/json_or_yaml_config_file --other_options
```

You can find and run an example in the examples/linear_fit directory:

```
morpho --config scripts/morpho_linear_fit.yaml
```

"Help will always be given to those who ask for it":

```
morpho --help
```

5.2.2 Using morpho API

The morpho python API allows you to run custom and more modulable scripts. In a python script, the processors should be created, configured and run. Connections between processors are made by setting a internal varible of a processor (like "results" for PyStanSamplingProcessor) as the internal variable of another variable. Examples of such python scripts can be found in the examples folder.

```
python linear_fit/scripts/pystan_test.py
```

CHAPTER 6

Example

The `linear_fit` analysis serves as an example of how to use morpho, and specifically, how to prepare a configuration file, Stan model and data file for a morpho run. See Use for more details regarding analysis file organization.

Run `linear_fit` from the `examples` folder by executing:

```
morpho --config linear_fit/scripts/morpho_linear_fit.yaml
```

Equivalently, you can run the same example using the python API:

```
python linear_fit/scripts/pystan_test.py
```

6.1 Model

The `linear_fit/models` folder contains two examples Stan models `model_linear_generator.stan` and `model_linear_fit.stan`. The first model will generate a set of points normally distributed along a line. The data are saved into a R file The data points are extracted from the file, Stan code model inputs these data points and it extracts posteriors for the line's slope and y-intercept, as well as the variance of the normal distribution. Convenience plots are then produced: a a posteriori distribution plot of the model parameters and the time series.

6.2 Executing the example

The example exists in two forms:

- A yaml configuration file
- A python script

6.2.1 Configuration File

The configuration file `linear_fit/scripts/morpho_linear_fit.yaml` specifies the processors that should be used, how they should be connected together, how they are individually configured and in which order they should be run. The content of the file possesses 2 main structures:

- The *processors-toolbox* dictionary
- The processors *configurations*

The structure of the configuration file is very similar to the Katydid software.

6.2.2 `Processors-toolbox` Block

This block defines the processors to be used and assigns these a name. It also provide the connections between processors (which variable of a processor will be set as variable of another processor) and defines the order in which the processors will be executed.

```
processors-toolbox:
  # Define the processors and their names
    processors:
        - type: morpho:PyStanSamplingProcessor
          name: generator
        - type: IORProcessor
          name: writer
        - type: IORProcessor
          name: reader
        - type: morpho:PyStanSamplingProcessor
          name: analyzer
        - type: APosterioriDistribution
          name: posterioriDistrib
        - type: TimeSeries
          name: timeSeries
  # Define in which order the processors should be run and how connections should be␣
↪made
    connections:
        - signal: "generator:results"
          slot: "writer:data"
        - signal: "reader:data"
          slot: "analyzer:data"
        - signal: "analyzer:results"
          slot: "posterioriDistrib:data"
        - signal: "analyzer:results"
          slot: "timeSeries:data"
```

The block is composed of two structures:

- *processors* defines the processors to be used and their names. The type defines which class/processor should be used. For example, we will use *PyStanSamplingProcessor* from the *morpho* package. It is possible to import classes/processors from other packages (for example mermithid) by setting using *type: mermithid:ProcessorX* instead of *type: morpho:ProcessorY*. If no package is given (for example: *type: TimeSeries*), it will look for the default *morpho* package.
- *connections* defines the order in which the processors are run. In the example, it will be *generator* -> *writer* -> *reader* -> *analyzer* -> *posterioriDistrib* -> *timeSeries*. It also defines how processors are connected together: for example the internal variable *results* of *generator* (called *signal*) containing the MC samples as a dictionary will be given to *writer* as *data* (called *slot*). It is important that the signal and slot types match.

6.2.3 Processors configurations

The following dictionaries defines the properties of each processor:

```
# Configure generator
generator:
  model_code: "linear_fit/models/model_linear_generator.stan"
  input_data:
      slope: 1
      intercept: -2
      xmin: 1
      xmax: 10
      sigma: 1.6
  iter: 530
  warmup: 500
  interestParams: ['x','y','residual']
  delete: False
```

Documentation about each processor parameters can be found in the source code in each class.

6.3 Python script

Similarly it is possible to create, configure and run processors using the morpho python API. An example can be found in `linear_fit/scripts/pystan_test.py`. This example should do the exact same thing as the script above.

The python API is an alternative way of using morpho. It can be used when the object must be modified between two processors and this cannot be done using a processor (or the ProcessorAssistant). It is also useful to test new features. However it is not the recommended method for production analyses.

CHAPTER 7

How to create new processors

At that point you might be thinking that morpho is great, but it does not have the feature or a processor you want. Before going any further, you should go in the morpho issue tracker to see if someone else is not working on this feature. If you see something similar there, you should say so there and/or on the morphoorg Slack. If you don't, then you are in the right place to know how to create your own processor.

7.1 Generalities about processors

As you might have read there, processors are objects that act on data and produce an output. Generally processors actions are intended to be simple in order to keep things as modular as possible. For example, you would prefer a processor that reads a file and one that acts on these compared with one that does both of these at once.

Morpho already provides a set of processors that could serve as a basis for your new processor. For example, there exists a input/output base class that defines base methods for any processor reading/writing a specific file format. If that is the case, you should consider using this class as a base class for your own. If there is not such class but one could with some modifications, you should consider the possibility of doing these modifications so you could use this class as a base for your new processor. If really none of the existing classes is of any help for you, creating a new processor from scratch is the way to go.

7.2 Structure and requirements for a new processor

Let's have a look at a basic example: the GaussianSamplingProcessor.

```
from morpho.utilities import morphologging, reader
from morpho.processors import BaseProcessor
logger = morphologging.getLogger(__name__)

__all__ = []
__all__.append(__name__)
```

(continues on next page)

(continued from previous page)

```
class GaussianSamplingProcessor(BaseProcessor):
        '''
        Sampling processor that will generate a simple gaussian distribution
        using TRandom3.
        Does not require input data nor model (as they are define in the class itself)
        Parameters:
                iter (required): total number of iterations (warmup and sampling)
                mean: mean of the gaussian (default=0)
                width: width of the gaussian (default=0)
        Input:
                None
        Results:
                results: dictionary containing the result of the sampling of the␣
↪parameters of interest
        '''

        def InternalConfigure(self, input):
                self.iter = int(reader.read_param(input, 'iter', "required"))
                self.mean = reader.read_param(input, "mean", 0.)
                self.width = reader.read_param(input, "width", 1.)
                if self.width <= 0.:
                        raise ValueError("Width is negative or null!")
                return True

        def InternalRun(self):
                from ROOT import TRandom3
                ran = TRandom3()
                data = []
                for _ in range(self.iter):
                        data.append(ran.Gaus(self.mean, self.width))
                self.results = {'x': data}
                return True
```

This processor aims at generating random values following a normal distribution using `TRandom3` from ROOT. Processors all inherite from the BaseProcessor class that defines very basic behaviors. BaseProcessor defines two methods `InternalConfigure` and `InternalRun`. `InternalConfigure` is used to configure the processor: here the number of values to generate (`iter`), the mean (`mean`) and the width (`width`) are given to the processor from the configuration dictionary (this dictionary is extracted from the configuration file). This method makes sure that all the given parameters are okay so the execution will work fine: for example we make sure the width is positive. `InternalRun` is used for the actual execution: it produces the samples from the normal distribution. The result of this sampling is saved inside a membre variable of the class (`results` in this case) in the shape of a dictionary. Contrary to Katydid, there are no defined data class defined in this framework for containing the intermediate results. We use python defined objects such as float, string, list or dictionary: we try to avoid using objects defined by external packages (such as ROOT or PyStan).

CHAPTER 8

Morpho 1

8.1 Introduction

Morpho is a python interface to the Stan/PyStan Markov Chain Monte Carlo package.

Morpho is intended as a meta-analysis tool to fit or generate data, organize inflow and outflow of data and models.

For more information, also see:

Stan: http://mc-stan.org

PyStan: https://pystan.readthedocs.io/en/latest/index.html

8.2 Install

Dependencies

The following dependencies should be installed (via a package manager) before installing morpho:

- python (2.7.x; 3.x not supported)
- python-pip
- git
- python-matplotlib

Morpho reads and saves files in either **R** or **ROOT.** If you would like to use root, install root-system or see https://root.cern (and ensure that the same version of python is enabled for morpho and ROOT).

Virtual environment-based installation

We recommend installing morpho using pip inside a python virtual environment. Doing so will automatically install dependencies beyond the four listed above, including PyStan 2.15.

If necessary, install [virtualenv](https://virtualenv.pypa.io/en/stable/), then execute: ‘‘‘bash

virtualenv ~/path/to/the/virtualenvironment source ~/path/to/the/virtualenvironment/bin/activate #Activate the environment #Use “bash deactivate” to exit the environment pip install -U pip #Update pip to >= 7.0.0 cd ~/path/to/morpho pip install . pip install .[all]

“‘

Docker installation

If you would like to modify your local installation of morpho (to add features or resolve any bugs), we recommend you use a [Docker container](https://docs.docker.com/get-started/) instead of a python virtual environment. To do so:

1. Install Docker: https://docs.docker.com/engine/installation/.
2. Clone and pull the latest master version of morpho.
3. Inside the morpho folder, execute `docker-compose run morpho`. A new terminal prompter (for example, `root@413ab10d7a8f:`) should appear. You may make changes to morpho either inside or outside of the Docker container. If you wish to work outside of the container, move morpho to the `morpho_share` directory that is mounted under the `/host` folder created by docker-compose.
4. You can remove the container image using `docker rmi morpho_morpho`.

If you develop new features or identify bugs, please open a GitHub issue.

8.3 Running Morpho

Once the relevant data, model and configuration files are at your disposal, run morpho by executing: “‘bash

morpho –config /path/to/json_or_yaml_config_file –other_options

“‘

You can test morpho using the example in the morpho_test directory: “‘bash

morpho –config morpho_test/scripts/morpho_linear_fit.yaml

“‘

8.4 An Example File

The format allows the user to execute Stan using standarized scripts. Let us now take apart an example file to illustrate how morpho functions. You can find the example file in

```
morpho/examples/morpho_test/scripts/morpho_linear_fit.yaml
```

Let us start with the initiation portion of the configuration.

```
morpho:
 do_preprocessing: False
 do_stan: True
 do_postprocessing: False
 do_plots: True
```

Under the morpho block, you can select how the processors will be run. In this case, it will run the main Stan function and produce plots at the end of processing.

Next, we come to the main Stan configuration block, where both running conditions, data and parameters can be fed into the Stan model.

```
stan:
name: "morpho_test"
model:
   file: "./morpho_test/models/morpho_linear_fit.stan"
   function_file: None
   cache: "./morpho_test/cache"
 data:
   files:
   - name: "./morpho_test/data/input.data"
      format: "R"
   parameters:
    - N: 30
 run:
   algorithm: "NUTS"
   iter: 4000
   warmup: 1000
   chain: 12
   n_jobs: 2
   init:
    - slope : 2.0
      intercept : 1.0
      sigma: 1.0
 output:
   name: "./morpho_test/results/morpho_linear_fit"
   format: "root"
   tree: "morpho_test"
   inc_warmup: False
   branches:
   - variable: "slope"
     root_alias: "a"
   - variable: "intercept"
     root_alias: "b"
```

The model block allows you to load in your Stan model file (for more on Stan models, see PyStan or Stan documentations). The compiled code can be cached to reduce running time. It is also possible to load in *external* functions located in separated files elsewhere.

The next block, the data block, reads in data. File formats include R and root. One can also load in parameters directly using the parameters block, as we do for the variable *N*.

The next block, the run block, allows one to control how Stan is run (number of chains, warmup, algorithms, etc.). Initializations can also be set here. This block feeds directly into PyStan.

The last block within the Stan block is the output. In this example, we save to a root file, and maintain two variables, *a* and *b*.

Since we specified the configure file to also make some plots, we can set up those conditions as well. In our example again, we have:

```
plot:
 which_plot:
  - method_name: histo
    module_name: histo
    title: "histo"
    input_file_name : "./morpho_test/results/morpho_linear_fit.root"
    input_tree: "morpho_test"
    output_path: ./morpho_test/results/
    data:
      - a
```

The plot saves a PDF of the variable *a* based on the root file results.

The flow is thus as follows. Morpho is told to execute Stan and its plotting features. The Stan execution reads in external data and sets the running in much the same way as PyStan does. Results are then saved to the results folder (in this case, under root files). Plots are also executed to ensure the quality of results.

8.5 Preprocessing

Preprocessing functions are applied to data in advance of executing the fitter. Typically this is done to prepare the data in some state in advance of fitting.

Preprocessing can be set as a flag in the beginning of the configuration file. As an example

```
morpho:
  do_preprocessing: true
```

Later in the configuration file, you can set up the commands to pre-process data

```
preprocessing:
 which_pp:
  - method_name: bootstrapping
    module_name: resampling
    input_file_name: ./my_spectrum.root
    input_tree: input
    output_file_name: ./my_fit_data.root
    output_tree: bootstrapped_data
    option: "RECREATE"
    number_data: 5000
```

In the above example, it will randomly sample 5000 data points from the root file "my_spectrum.root" (with tree input) and save it to a new data file called "./my_fit_data.root" with tree name " bootstrapped_data".

8.6 Postprocessing

Postprocessing functions are applied to data after executing the fitter. Typically this is done examine the parameter information and check for convergence.

Postprocessing can be set as a flag in the beginning of the configuration file. As an example

```
morpho:
  do_postprocessing: true
```

Later in the configuration file, you can set up the commands to post-process data. For example, to reduce the data into bins

```
preprocessing:
 which_pp:
  - method_name: general_data_reducer
    module_name: general_data_reducer
    input_file_name: ./my_spectrum.root
    input_file_format: root
    input_tree: spectrum
    data:
     -Kinetic_Energy
```

(continues on next page)

(continued from previous page)

```
    minX:
     -18500.
    maxX:
     -18600.
    nBinHisto:
     -1000
    output_file_name: ./my_binned_data.root
    output_file_format: root
    output_tree: bootstrapped_data
    option: "RECREATE"
```

In the above example, it will take data from the root file saved in the *Kinetic_Energy* parameter and rebin it in a 1000-bin histogram.

8.7 Plots

Plotting is a useful set of routines to make quick plots and diagnostic tests, usualluy after the Stan main executable has been run.:

```
morpho:
  do_plots: true
```

Later in the configuration file, you can set up the commands to plot data after the fitter is complete.

```
plot:
which_plot:
 - method_name: histo
    title: "histo"
    input_file_name : "./morpho_test/results/morpho_linear_fit.root"
    input_tree: "morpho_test"
    output_path: ./morpho_test/results/
    data:
     - a
```

In the above example, it will take data from the root file saved in the *a* parameter plot and save it to ./morpho_test/results/histo_a.pdf

We have plotting schemes that cover a number of functions:

1. Plotting contours, densities, and matricies (often to look for correlations).
2. Time series to study convergences.

8.8 Example Script

The following are example yaml scripts for important Preprocessing, Postprocessing, and Plot routines in Morpho 1. The format of the yaml script for other methods can be obtained from the documentation for that method.

8.9 Preprocessing

“do_preprocessing : true” must be in the morpho dictionary. The dictionaries below should be placed in a “which_pp” dictionary inside the “preprocessing” dictionary.

8.9.1 bootstrapping

Resamples the contents of a tree. Instead of regenerating a fake data set on every sampler, one can generate a larger data set, then extract subsets.

```
- method_name: "boot_strapping"
  module_name: "resampling"
  input_file_name: "input.root" # Name of file to access
                                # Must be a root file
  input_tree: "tree_name" # Name of tree to access
  output_file_name: "output.root" # Name of the output file
                                  # The default is the same the input_file_name
  output_tree: "tree_name" # Tree output name
                           # Default is same as input.
  number_data: int # Number of sub-samples the user wishes to extract.
  option: "RECREATE" # Option for saving root file (default = RECREATE)
```

8.10 Postprocessing

"do_postprocessing : true" must be in the morpho dictionary. The dictionaries below should be placed in a "which_pp" dictionary inside the "postprocessing" dictionary.

8.10.1 general_data_reducer

Tranform a function defining a spectrum into a histogram of binned data points.

```
- method_name: "general_data_reducer"
  module_name: "general_data_reducer"
  input_file_name: "input.root" # Path to the root file that contains the raw data
  input_file_format: "root" # Format of the input file
                            # Currently only root is supported
  input_tree: "spectrum" #  Name of the root tree containing data of interest
  data: ["KE"] # Optional list of names of branches of the data to be binned
  minX:[18500.] # Optional list of minimum x axis values of the data to be binned
  maxX:[18600.] # Optional list of maximum x axis values of the data to be binned
  nBinHisto:[50] # List of desired number of bins in each histogram
  output_file_name: "out.root", # Path to the file where the binned data will be saved
  output_file_format: "root", # Format of the output file
  output_file_option: RECREATE # RECREATE will erase and recreate the output file
                               # UPDATE will open a file (after creating it, if it␣
↪does not exist) and update the file.
```

8.11 Plot

"do_plots : true" must be in the morpho dictionary. The dictionaries below should be placed in a "which_plot" dictionary inside the "plot" dictionary.

8.11.1 contours

contours creates a matrix of contour plots using a stanfit object

```
- method_name: "contours"
  module_name: "contours"
  read_cache_name: "cache_name_file.txt" # File containing path to stan model cache
  input_fit_name: "analysis_fit.pkl"# pickle file containing stan fit object
  output_path: "./results/" # Directory to save results in
  result_names: ["param1", "param2", "param3"] # Names of parameters to plot
  output_format: "pdf"
```

8.11.2 histo

Plot a 1D histogram using a list of data

```
- method_name: "histo"
  module_name: "histo"
```

8.11.3 spectra

Plot a 1D histogram using 2 lists of data giving an x point and the corresponding bin contents

```
- method_name: "spectra"
  module_name: "histo"
  title: "histo"
  input_file_name : "input.root"
  input_tree: "tree_name"
  output_path: "output.root"
  data:
      - param_name
```

8.11.4 histo2D

Plot a 2D histogram using 2 lists of data

```
- method_name: "histo2D"
  module_name: "histo"
  input_file_name : "input.root"
  input_tree: "tree_name"
  root_plot_option: "contz"
  data:
    - list_x_branch
    - list_y_branch
```

8.11.5 histo2D_divergence

Plot a 2D histogram with divergence indicated by point color

```
- method_name: "histo2D_divergence"
  module_name: "histo"
  input_file_name : "input.root"
  input_tree: "tree_name"
  root_plot_option: "contz"
```

(continues on next page)

(continued from previous page)

```
  data:
    - list_x_branch
    - list_y_branch
```

8.11.6 aposteriori_distribution

Plot a grid of 2D histograms

```
- method_name: "aposteriori_distribution"
  module_name: "histo"
  input_file_name : "input.root"
  input_tree: "tree_name"
  root_plot_option: "cont"
  output_path: output.root
  title: "aposteriori_plots"
  output_format: pdf
  output_width: 12000
  output_height: 1100
  data:
    - param1
    - param2
    - param3
```

8.11.7 correlation_factors

Plot a grid of correlation factors

```
- method_name: "correlation_factors"
  module_name: "histo"
  input_file_name : "input.root"
  input_tree: "tree_name"
  root_plot_option: "cont"
  output_path: output.root
  title: "aposteriori_plots"
  output_format: pdf
  output_width: 12000
  output_height: 1100
  data:
    - param1
    - param2
    - param3
```

CHAPTER 9

Contribute

9.1 Branching Model

Morpho uses the git flow branching model, as described here. In summary, the master branch is reserved for numbered releases of morpho. The only branches that may branch off of master are hotfixes. All development should branch off of the develop branch, and merge back into the develop branch when complete. Once the develop branch is ready to go into a numbered release, a release branch is created where any final testing and bug fixing is carried out. This release branch is then merged into master, and the resulting commit is tagged with the number of the new release.

9.2 Style

Morpho loosely follows the style suggested in the Style Guide for Python (PEP 8).

Every package, module, class, and function should contain a docstring, that is, a comment beginning and ending with three double quotes. We use the Google format, because the docstrings can then be automatically formatted by sphinx and shown in the API.

Every docstring should start with a single line (<=72 characters) summary of the code. This is followed by a blank line, then further description is in paragraphs separated by blank lines. Functions should contain "Args:", "Returns:", and if necessary, "Raises" sections to specify the inputs, outputs, and exceptions for the function. All text should be wrapped to around 72 characters to improve readability.

9.3 Other Conventions

- __init__.py files:

In morpho 1, __init__.py files are set up such that

```
from package import *
```

will import all functions from all subpackages and modules into the namespace. If a package contains the subpackages "subpackage1" and "subpackage2", and the modules "module1" and "module2", then the __init__.py file should include imports of the form:

```
from . import subpackage1
from . import subpackage2
from ./module1 import *
from ./module2 import *
```

In morpho 2, __init__.py files are set up such that

```
from package import *
```

will import all modules into the namespace, but it will not directly import the functions into the namespace. For our package containing "subpackage1", "subpackage2", "module1", and "module2", __init__.py should be of the form:

```
__all__ = ["module1", "module2"]
```

In this case, functions would be called via module1.function_name(). If one wants all of the functions from module1 in the namespace, then they can include "from package.module1 import *" at the top of their code. This change to more explicit imports should prevent any issues with function names clashing as Morpho grows.

CHAPTER 10

Validation Log

10.1 Log

10.1.1 Version: v2.3.1

Release Date: April 17th 2018

New Hotfixes:

- Downgrade pystan to v2.17.1

10.1.2 Version: v2.3.1

Release Date: March 28th 2018

New Features:

- Debug section in README.md
- Upgrade pystan to v2.18.1

10.1.3 Version: v2.3.0

Release Date: November 13 2018

New Features:

- **RooFit base interface processor:**
  - All RooFit processors now inherit from RooFitInterfaceProcessor

    - Allow to do sampling, likelihood sampling and fitting by defining the model only once
- Python API example: gaussian model

10.1.4 Version: v2.2.1

Release Date: Thursday November 8th 2018

Fixes:

- Fixing the import of RootCanvas and RootHistogram in Histogram

10.1.5 Version: v2.2.0

Release Date: Sunday November 4th 2018

New Features:

- Possibility to generate several histograms on the same RootCanvas
- A huge effort in documenting the code and on RTD!

10.1.6 Version: v2.1.5

Release Date: Friday September 28th 2018

New Features:

- Add access to processors properties from ToolBox
- Travis: adding linux via Docker

Fixes:

- **Documentation update:**
    - Adding docstring for processors
    - Update example
    - Adding descriptions about morpho 2, reworking the morpho 1's
- Issue tracker: adding template issues
- Plotting: better RootCanvas class, more RootHistogram methods

10.1.7 Version: v2.1.4

Release Date: Tues. July 31st 2018

Fixes:

- Travis fix: switch to XCode 9.4

10.1.8 Version: v2.1.3

Release Date: Thur. July 26th 2018

Fixes:

- **RTD**
  - Changed CPython to 3
  - Edited conf.py to use better_apidoc
  - Defined try/except for additional modules like ROOT and pystan
- Dependencies cleanup (matplotlib)

10.1.9 Version: v2.1.2

Release Date: Thur. July 19th 2018

Fixes:

- Update dependencies to support python 3.7

10.1.10 Version: v2.1.1

Release Date: Fri. June 29th 2018

Fixes:

- Debug of the docker and docker-compose

10.1.11 Version: v2.1.0

Release Date: Wed. June 27th 2018

New Features:

- **Morpho executable:**
  - Use configuration file similar to Katydid: configuration can be edited via the CLI
  - Toolbox that creates, configures, runs and connects processors
  - Can import processors from other modules (mermithid tested)
  - Add main executable

Fixes:

10.1.12 Version: v2.0.0

Release Date: Sat. June 9th 2018

New Features:

- **Upgrade to morpho2:**
  - **Create basic processors for**
    - sampling (PyStan and RooFit)
    - plotting
    - IO (ROOT, csv, json, yaml, R)
  - Added tests scripts and main example

Fixes:

- Use brew instead of conda for Travis CI

10.2 Guidelines

- All new features incorporated into a tagged release should have their validation documented. * Document the new feature. * Perform tests to validate the new feature. * If the feature is slated for incorporation into an official analysis, perform tests to show that the overall analysis works and benefits from this feature. * Indicate in this log where to find documentation of the new feature. * Indicate in this log what tests were performed, and where to find a writeup of the results.
- Fixes to existing features should also be validated. * Perform tests to show that the fix solves the problem that had been indicated. * Perform tests to show that the fix does not cause other problems. * Indicate in this log what tests were performed and how you know the problem was fixed.

10.3 Template

10.3.1 Version:

Release Date:

New Features:

- **Feature 1**
  - Details
- **Feature 2**
  - Details

Fixes:

- **Fix 1**
  - Details
- **Fix 2**
  - Details

CHAPTER 11

morpho

11.1 morpho package

All modules and packages used by morpho

Subpackages:

- preprocessing: Process inputs before passing to stan
- loader: Load data for use by stan
- plot: Create plots from stan outputs
- postprocessing: Process stan outputs before or after plotting

Subpackages:

11.1.1 morpho.processors package

Submodules:

morpho.processors.BaseProcessor module

Some template vars

Members: BaseProcessor

Functions:

Classes:

Base processor for sampling-type operations Authors: J. Johnston, M. Guigue, T. Weiss Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.BaseProcessor.**BaseProcessor**(*name*, **args*, ***kwargs*)
Bases: object

Base Processor All Processors will be implemented in a child class where the specifics are encoded by overwriting Configure and Run.

Parameters **delete** – do delete processor after running

Input: None

Results: None

name

delete

Configure(*params*)
This method will be called by nymph to configure the processor

InternalConfigure(*params*)
Method called by Configure() to set up the object. Must be overridden by child class.

Run()
This method will be called by nymph to run the processor

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

Subpackages:

morpho.processors.IO package

Submodules:

morpho.processors.IO.IOCVSProcessor module

Some template vars

Members: IOCVSProcessor

Functions:

Classes:

CVS IO Processor Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.IO.IOCVSProcessor.**IOCVSProcessor**(*name*, **args*, ***kwargs*)
Bases: *morpho.processors.IO.IOProcessor.IOProcessor*

Base IO CVS Processor The CVS Reader and Writer

Parameters

- **filename** (*required*) – path/name of file
- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")

Input: None

Results: data: dictionary containing the data

Reader()
Need to be defined by the child class

Writer()
Need to be defined by the child class

morpho.processors.IO.IOJSONProcessor module

Some template vars

Members: IOJSONProcessor IOYAMLProcessor

Functions:

Classes:

JSON/Yaml IO processors Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.IO.IOJSONProcessor.**IOJSONProcessor**(*name*)
Bases: *morpho.processors.IO.IOProcessor.IOProcessor*

Base IO JSON Processor

Parameters

- **filename** (*required*) – path/name of file
- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")

Input: None

Results: data: dictionary containing the data

module_name = 'json'

dump_kwargs = {'indent': 4}

Reader()
Need to be defined by the child class

Writer()
Need to be defined by the child class

class morpho.processors.IO.IOJSONProcessor.**IOYAMLProcessor**(*name*)
Bases: *morpho.processors.IO.IOJSONProcessor.IOJSONProcessor*

IO YAML Processor: uses IOJSONProcessor as basis

Parameters

- **filename** (*required*) – path/name of file
- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")

Input: None

Results: data: dictionary containing the data

module_name = 'yaml'

morpho.processors.IO.IOProcessor module

Some template vars

Members: IOProcessor

Functions:

Classes:

Base input/output processor for reading and writing operations Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.IO.IOProcessor.**IOProcessor**(*name*, **args*, ***kwargs*)
Bases: *morpho.processors.BaseProcessor.BaseProcessor*

IO_Processor All Processors will be implemented in a child class where the specifics are encoded by overwriting Configure and Run.

Parameters

- **filename** (*required*) – path/name of file

- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")

Input: None

Results: data: dictionary containing the data

Reader()
Need to be defined by the child class

Writer()
Need to be defined by the child class

InternalConfigure(*params*)
This method will be called by nymph to configure the processor

InternalRun()
This method will read or write an file

morpho.processors.IO.IOROOTProcessor module

Some template vars

Members: IOROOTProcessor

Functions:

Classes:

ROOT IO processor Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.IO.IOROOTProcessor.**IOROOTProcessor**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.IO.IOProcessor.IOProcessor*

Base IO ROOT Processor The ROOT Reader and Writer

Parameters

- **filename** (*required*) – path/name of file
- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")
- **tree_name** (*required*) – name of the tree
- **file_option** – option for the file (default=Recreate)

Input: None

Results: data: dictionary containing the data

InternalConfigure(*params*)
This method will be called by nymph to configure the processor

Reader()
Read the content of a TTree in a ROOT File. Note the use of the uproot package. The variables should be a list of the "variable" to read.

Writer()
Write the data into a TTree in a ROOT File. The variables should be a list of dictionaries where

- "variable" is the variable name in the input dictionary,
- "root_alias" is the name of the branch in the tree,
- "type" is the type of data to be saved.

morpho.processors.IO.IORProcessor module

Some template vars

Members: IORProcessor

Functions:

Classes:

R IO processor Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.IO.IORProcessor.**IORProcessor**(*name*, **args*, ***kwargs*)
Bases: *morpho.processors.IO.IOProcessor.IOProcessor*

Base IO R Processor The R Reader and Writer use pystan.misc package

Parameters

- **filename** (*required*) – path/name of file
- **variables** (*required*) – variables to extract
- **action** – read or write (default="read")

Input: None

Results: data: dictionary containing the data

Reader()
Need to be defined by the child class

Writer()
Need to be defined by the child class

morpho.processors.diagnostics package

Submodules:

morpho.processors.diagnostics.StanDiagnostics module

Some template vars

Members: StanDiagnostics

Functions:

Classes:

Creates Stan diagnostic plots. Authors: T. Weiss Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.diagnostics.StanDiagnostics.**StanDiagnostics**(**args*, ***kwargs*)

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Describe.

InternalConfigure(*params*)
Configures by reading in list of names of divergence plots to be created and dictionary containing fit object

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.misc package

Submodules:

morpho.processors.misc.ProcessorAssistant module

Some template vars

Members: ProcessorAssistant

Functions:

Classes:

Create a wrapping processor from a function given in a python script Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.misc.ProcessorAssistant.**ProcessorAssistant**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Convenience wrapper that creates a processor around a function from an external python script The parameters of the function are given in the same configuration dictionary.

Parameters

- **module_name** (*required*) – path/name of the python script
- **function_name** (*required*) – name of the function to execute

Input: None

Results: results: dictionary containing the result of the function

InternalConfigure(*config_dict*)
Method called by Configure() to set up the object. Must be overridden by child class.

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.plots package

Submodules:

morpho.processors.plots.APosterioriDistribution module

Some template vars

Members: APosterioriDistribution

Functions:

Classes:

Plot a posteriori distribution of the variables of interest Authors: J. Jonhston, M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.plots.APosterioriDistribution.**APosterioriDistribution**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Generates an a posterior distribution for all the parameters of interest TODO: - Use the RootHistogram class instead of TH1F itself... :param n_bins_y: number of bins (default=100) :param n_bins_y: number of bins (default=100) :param variables: name(s) of the variable in the data :type variables: required :param width: window width (default=600) :param height: window height (default=400) :param title: canvas title :param x_title: title of the x axis :param y_title: title of the y axis :param options: other options (logy, logx) :param root_plot_option: root plot option (default=contz) :param output_path: where to save the plot :param output_pformat: plot format (default=pdf)

data

InternalConfigure(*param_dict*)
Configure

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.plots.Histogram module

Some template vars

Members: Histogram

Functions:

Classes:

Plot an histogram of the variables of interest Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.plots.Histogram.**Histogram**(*name*, **args*, ***kwargs*)
Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Processor that generates a canvas and a histogram and saves it. TODO: - Add the possibility to plot several histograms with the same binning on the same canvas - Generalize this processor so it understands if if should be a 1D or a 2D histogram

Parameters

- **n_bins_x** – number of bins (default=100)
- **range** – range of x (list)
- **variables** (*required*) – name(s) of the variable in the data
- **width** – window width (default=600)
- **height** – window height (default=400)
- **title** – canvas title
- **x_title** – title of the x axis
- **y_title** – title of the y axis

- **options** – other options (logy, logx)
- **output_path** – where to save the plot
- **output_pformat** – plot format (default=pdf)

Input: data: dictionary containing model input data

Results: None

InternalConfigure(*params*)
Configure

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.plots.RootCanvas module

Some template vars

Members: RootCanvas

Functions:

Classes:

Root-based canvas class Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.plots.RootCanvas.**RootCanvas**(*input_dict*, *optStat='emr'*)
Bases: object

Create default ROOT canvas object.

Parameters

- **width** – window width (default=600)
- **height** – window height (default=400)
- **title** – canvas title
- **x_title** – title of the x axis
- **y_title** – title of the y axis
- **options** – other options (logy, logx)
- **output_path** – where to save the plot
- **output_pformat** – plot format (default=pdf)

cd(*number=0*)
Go to frame 'number' of the TCanvas

Divide(*cols*, *rows*)
Divide the TCanvas

Draw()
Draw the TCanvas

Save()
Save the TCanvas

morpho.processors.plots.RootHistogram module

Some template vars

Members: RootHistogram

Functions:

Classes:

Root-based histogram class Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.plots.RootHistogram.**RootHistogram**(*input_dict*, *optStat='emr'*)

Bases: object

Create default ROOT histogram object.

Parameters

- **n_bins_x** – number of bins (default=100)
- **range** – range of x (list)
- **variables** (*required*) – parameters to be put in the histogram
- **title** – plot title
- **x_title** – title of the x axis

GetNbinsX()

Fill(*input_data*)

SetBinsError(*a_list*)

SetBinsContent(*a_list*)

SetLineColor(*value*, *n=1*)

Draw(*arg='hist'*)

Write()

morpho.processors.plots.TimeSeries module

Some template vars

Members: TimeSeries

Functions:

Classes:

Plot a time series of the variables of interest Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.plots.TimeSeries.**TimeSeries**(*name*, **args*, ***kwargs*)
Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Time series plot generator. Display the value for each parameter (variables) as a time series. The red points are warmup part of the chain.

Parameters

- **variables** (*required*) – name(s) of the variable in the data
- **width** – window width (default=600)
- **height** – window height (default=400)
- **title** – canvas title
- **x_title** – title of the x axis
- **y_title** – title of the y axis
- **options** – other options (logy, logx)
- **output_path** – where to save the plot
- **output_pformat** – plot format (default=pdf)

Input: data: dictionary containing model input data

Results: None

data

InternalConfigure(*param_dict*)
Method called by Configure() to set up the object. Must be overridden by child class.

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.sampling package

Submodules:

morpho.processors.sampling.GaussianRooFitProcessor module

Some template vars

Members: GaussianRooFitProcessor

Functions:

Classes:

Processor for linear fitting Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.sampling.GaussianRooFitProcessor.**GaussianRooFitProcessor**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.sampling.RooFitInterfaceProcessor.RooFitInterfaceProcessor*

Linear fit of data using RootFit Likelihood sampler. We redefine the _defineDataset method as this analysis requires datapoints in a 2D space. Users should feel free to change this method as they see fit.

Parameters

- **varName** (*required*) – name(s) of the variable in the data
- **nuisanceParams** (*required*) – parameters to be discarded at end of sampling
- **interestParams** (*required*) – parameters to be saved in the results variable
- **iter** (*required*) – total number of iterations (warmup and sampling)
- **warmup** – number of warmup iterations (default=iter/2)
- **chain** – number of chains (default=1)
- **n_jobs** – number of parallel cores running (default=1)
- **binned** – should do binned analysis (default=false)
- **options** – other options
- **a** (*required*) – range of slopes (list)
- **b** (*required*) – range of intercepts (list)
- **x** (*required*) – range of x (list)
- **y** (*required*) – range of y (list)
- **witdh** (*required*) – range of width (list)

Input: data: dictionary containing model input data

Results: results: dictionary containing the result of the sampling of the parameters of interest

InternalConfigure(*config_dict*)
Method called by Configure() to set up the object. Must be overridden by child class.

definePdf(*wspace*)
Define the model which is that the residual of the linear fit should be normally distributed.

morpho.processors.sampling.GaussianSamplingProcessor module

Some template vars

Members: GaussianSamplingProcessor

Functions:

Classes:

Gaussian distribution sampling processor Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.sampling.GaussianSamplingProcessor.**GaussianSamplingProcessor**(*name*, **args*, ***kwarg*

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Sampling processor that will generate a simple gaussian distribution using TRandom3. Does not require input data nor model (as they are define in the class itself)

Parameters

- **iter** (*required*) – total number of iterations (warmup and sampling)
- **mean** – mean of the gaussian (default=0)
- **width** – width of the gaussian (default=0)

Input: None

Results: results: dictionary containing the result of the sampling of the parameters of interest

InternalConfigure(*input*)
Method called by Configure() to set up the object. Must be overridden by child class.

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.sampling.LinearFitRooFitProcessor module

Some template vars

Members: LinearFitRooFitProcessor

Functions:

Classes:

Processor for linear fitting Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.sampling.LinearFitRooFitProcessor.**LinearFitRooFitProcessor**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.sampling.RooFitInterfaceProcessor.RooFitInterfaceProcessor*

Linear fit of data using RootFit Likelihood sampler. We redefine the _defineDataset method as this analysis requires datapoints in a 2D space. Users should feel free to change this method as they see fit.

Parameters

- **varName** (*required*) – name(s) of the variable in the data
- **nuisanceParams** (*required*) – parameters to be discarded at end of sampling
- **interestParams** (*required*) – parameters to be saved in the results variable
- **iter** (*required*) – total number of iterations (warmup and sampling)
- **warmup** – number of warmup iterations (default=iter/2)
- **chain** – number of chains (default=1)
- **n_jobs** – number of parallel cores running (default=1)
- **binned** – should do binned analysis (default=false)
- **options** – other options
- **a** (*required*) – range of slopes (list)
- **b** (*required*) – range of intercepts (list)
- **x** (*required*) – range of x (list)
- **y** (*required*) – range of y (list)
- **witdh** (*required*) – range of width (list)

Input: data: dictionary containing model input data

Results: results: dictionary containing the result of the sampling of the parameters of interest

InternalConfigure(*config_dict*)
Method called by Configure() to set up the object. Must be overridden by child class.

definePdf(*wspace*)
Define the model which is that the residual of the linear fit should be normally distributed.

morpho.processors.sampling.PyStanSamplingProcessor module

Some template vars

Members: PyStanSamplingProcessor

Functions:

Classes:

PyStan sampling processor Authors: J. Formaggio, J. Johnston, M. Guigue, T. Weiss Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.sampling.PyStanSamplingProcessor.**PyStanSamplingProcessor**(*name*)

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Sampling processor that will call PyStan.

Parameters

- **model_code** (*required*) – location of the Stan model
- **function_files_location** – location of the Stan functions
- **model_name** – name of the cached model
- **cache_dir** – location of the cache folder (containing cached models)
- **input_data** – dictionary containing model input data
- **iter** (*required*) – total number of iterations (warmup and sampling)
- **warmup** – number of warmup iterations (default=iter/2)
- **chain** – number of chains (default=1)
- **n_jobs** – number of parallel cores running (default=1)
- **interestParams** – parameters to be saved in the results variable
- **no_cache** – don't create cache
- **force_recreate** – force the cache regeneration
- **init** – initial values for the parameters
- **control** – PyStan sampling settings

Input: data: dictionary containing model input data

Results: results: dictionary containing the result of the sampling of the parameters of interest

data

gen_arg_dict()

InternalConfigure(*params*)
Method called by Configure() to set up the object. Must be overridden by child class.

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

morpho.processors.sampling.RooFitInterfaceProcessor module

Some template vars

Members: RooFitInterfaceProcessor

Functions:

Classes:

Base processor for RooFit-based samplers Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.processors.sampling.RooFitInterfaceProcessor.**RooFitInterfaceProcessor**(*name*, **args*, ***kwargs*)

Bases: *morpho.processors.BaseProcessor.BaseProcessor*

Base class for RooFit-based sampling. A new class should inheritate from this one and have its own version of "definePdf". The input data are given via the attribute "data".

Parameters

- **varName** (*required*) – name(s) of the variable in the data
- **nuisanceParams** (*required*) – parameters to be discarded at end of sampling
- **interestParams** (*required*) – parameters to be saved in the results variable
- **iter** (*required*) – total number of iterations (warmup and sampling)
- **warmup** – number of warmup iterations (default=iter/2)
- **chain** – number of chains (default=1)
- **n_jobs** – number of parallel cores running (default=1)
- **binned** – should do binned analysis (default=false)
- **options** – other options

Input: data: dictionary containing model input data

Results: results: dictionary containing the result of the sampling of the parameters of interest

definePdf(*wspace*)
Defines the Pdf that RooFit will sample and add it to the workspace. The Workspace is then returned by the user. Users should always create their own method.

data

InternalConfigure(*config_dict*)
Method called by Configure() to set up the object. Must be overridden by child class.

InternalRun()
Method called by Run() to run the object. Must be overridden by child class.

11.1.2 morpho.utilities package

Submodules:

morpho.utilities.morphologging module

Some template vars

Members: getLogger

Functions:

Classes:

Morpho logging utilities Authors: J. Johnston, M. Guigue Date: 02/22/18

Summary

Data:

Reference

morpho.utilities.morphologging.**getLogger**(*name*, *stderr_lb=40*, *level=10*, *propagate=False*)
Return a logger object with the given settings that prints messages greater than or equal to a given level to stderr instead of stdout name: Name of the logger. Loggers are conceptually arranged

in a namespace hierarchy using periods as separators. For example, a logger named morpho is the parent of a logger named morpho.plot, and by default the child logger will display messages with the same settings as the parent

stderr_lb: Messages with level equal to or greaterthan stderr_lb will be printed to stderr instead of stdout

level: Initial level for the logger propagate: Whether messages to this logger should be passed to

the handlers of its ancestor

morpho.utilities.parser module

Some template vars

Members: change_and_format merge parse_args update_from_arguments

Functions:

Classes:

Definitions for parsing the CLI and updating the Toolbox configuration dictionary Authors: J. Johnston, M. Guigue, T. Weiss Date: 06/26/18

Summary

Data:

Reference

morpho.utilities.parser.**parse_args**()
Parse the command line arguments provided to morpho :param None:

Returns Namespace containing the arguments

Return type namespace

morpho.utilities.parser.**update_from_arguments**(*the_dict*, *args*)
Update a dictionary :param the_dict: Dictionary to update :param args: Dictionary to merge into the_dict

Returns Dictionary with args merged into the_dict

Return type dict

morpho.utilities.parser.**change_and_format**(*b*)
Try to convert a string into a boolean or float :param b: String containing a boolean or float

Returns If b == 'True' or 'False', then the corresponding boolean is returns. Otherwise, if b can be converted into a float, the float is returned. Otherwise b is returned.

Return type bool, float, or str

morpho.utilities.parser.**merge**(*a*, *b*, *path=None*)
Merge two dictionaries :param a: Base dictionary :param b: Dictionary to merge into a :param path: Location to merge b at

Returns Merged dictionary

Return type dict

morpho.utilities.plots module

Some template vars

Members:

Functions:

Classes:

Definitions for plots Authors: J. Johnston, M. Guigue, T. Weiss Date: 06/26/18

morpho.utilities.pystanLoader module

Some template vars

Members: extract_data_from_outputdata

Functions:

Classes:

Definitions for interfacing with pyStan IO Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

morpho.utilities.pystanLoader.**extract_data_from_outputdata**(*conf*, *theOutput*)

morpho.utilities.reader module

Some template vars

Members: add_dict_param read_param

Functions:

Classes:

Interface between config files and processors config dictionaries Authors: J. Johnston, M. Guigue, T. Weiss Date: 06/26/18

Summary

Data:

Reference

morpho.utilities.reader.**read_param**(*yaml_data*, *node*, *default*)

morpho.utilities.reader.**add_dict_param**(*dictionary*, *key*, *value*)

This method checks if a key already exists in a dictionary, and if not, it adds the key and its corresponding value to the dictionary.

Could be changed to take as input a list of tuples (key, value), so multiple parameters may be added at once.

morpho.utilities.toolbox module

Some template vars

Members: ToolBox

Functions:

Classes:

Toolbox class: create, configure and run processors Authors: M. Guigue Date: 06/26/18

Summary

Data:

Reference

class morpho.utilities.toolbox.**ToolBox**(*args*)

Manages processors requested by the user at run-time. Via a configuration file, the user defines which processor to use, how to configure them and how to connect them.

Run()

GetProcessor()

GetProcAttr(*varName*)

11.1.3 Summary

Data:

11.1.4 Reference

Python Module Index

m

Index

I

L

M

N

P

R

S

T

U

W