nestcheck Documentation

Release 0.0.0

Edward Higson

Jul 08, 2023

CONTENTS

Nested sampling is a popular numerical method for Bayesian computation, which simultaneously generates samples from the posterior distribution and an estimate of the Bayesian evidence for a given likelihood and prior. `nestcheck` provides Python utilities for analysing samples produced by nested sampling, and estimating uncertainties on nested sampling calculations (which have different statistical properties to calculations using other numerical methods). This includes implementations of the diagnostic tests and plots described in:

- "Sampling errors in nested sampling parameter estimation" (Higson et al., 2018);
- "nestcheck: diagnostic tests for nested sampling calculations" (Higson et al., 2019).

To get started, see the installation instructions and the quickstart demo. More examples of `nestcheck`'s use can be found in the code used to make the results and plots in Higson et al. (2019) at https://github.com/ejhigson/diagnostic.

CHAPTER

ONE

COMPATIBLE NESTED SAMPLING SOFTWARE

Currently `nestcheck.data_processing` has functions to load results from:

- PolyChord >=v1.14;
- MultiNest >=v3.11;
- dyPolyChord (same output format as PolyChord);
- dynesty;
- perfectns.

You can easily add input functions for other nested sampling software packages. Note that `nestcheck` requires information about the iso-likelihood contours within which dead points were sampled ("born"), which is needed to split nested sampling runs into their constituent single live point runs ("threads"); see Higson et al. (2018) for more details. `nestcheck` is fully compatible with dynamic nested sampling, in which the number of live points is varied to increase calculation accuracy.

CHAPTER

TWO

DOCUMENTATION CONTENTS

2.1 Installation

`nestcheck` is compatible with Python 2.7 and >=3.4, and can be installed with pip:

```
pip install nestcheck
```

Alternatively, you can download the latest version and install it by cloning the github repository:

```
git clone https://github.com/ejhigson/nestcheck.git
cd nestcheck
python setup.py install
```

Note that the github repository may include new changes which have not yet been released on PyPI (and therefore will not be included if installing with pip). Both of these methods also automatically install any of `nestcheck`'s dependencies which are not already satisfied by your system.

2.1.1 Dependencies

`nestcheck` requires:

- `numpy` >=1.13;
- `scipy` >=1.0.0;
- `matplotlib` >=2.1.0;
- `fgivenx` >=2.1.11;
- `pandas` >=0.21.0;
- `tqdm` >=4.11.

Note also that producing the birth contour output files needed for `nestcheck` analysis using `MultiNest` requires v3.11 or later, and using `PolyChord` requires v1.14 or later and the setting "write_dead"=True (its default value).

2.1.2 Tests

You can run the test suite with nose. From the root `nestcheck` directory, run:

```
nosetests
```

To also get code coverage information (this requires the `coverage` package), use:

```
nosetests --with-coverage --cover-erase --cover-package=nestcheck
```

If all the tests pass, the install should be working.

Note:

You can download this demo as a Jupyter notebook here and run it interactively yourself. The PolyChord nested sampling run data it uses can be downloaded at https://github.com/ejhigson/nestcheck_demo_data (this also contains runs using a few other likelihoods).

2.2 Quickstart demo

This is a brief demonstration covering loading nested sampling run data, performing calculations, running error analysis and diagnostic tests, and making plots. For detailed explanations of the diagnostic tests and plots, see the papers which introduce them (Higson et al. 2018, Higson et al. 2019).

More information about `nestcheck`'s code and and functionality can be found in the documentation. For more examples of `nestcheck`'s usage, including on a variety of likelihoods, see code used to make the results and diagrams in the diagnostic tests paper (Higson et al. 2019) (available at https://github.com/ejhigson/diagnostic).

2.2.1 Loading nested sampling runs

For this demo we will use some PolyChord nested sampling runs with a simple 2-dimensional Gaussian likelihood

$$\mathcal{L}(\theta) = \frac{1}{2\pi\sigma^2} \exp\left(\frac{|\theta|^2}{2\sigma^2}\right), \tag{2.1}$$

with $\sigma = 0.5$ and a uniform prior on each parameter in $[-10, 10]$. For more information about the dictionary format and keys `nestcheck` uses to store nested sampling runs, see the API documentation.

For example, a PolyChord run can be loaded as follows:

[1]:
```
import nestcheck.data_processing

base_dir = 'polychord_chains'  # directory containing run (PolyChord's 'base_dir'␣
↪setting)
file_root = 'gaussian_2d_100nlive_5nrepeats_1'  # output files' name root (PolyChord's
↪'file_root' setting)
run = nestcheck.data_processing.process_polychord_run(file_root, base_dir)
```

`nestcheck.data_processing` also has functions for loading nested sampling data from a variety of nested sampling software packages (including `MultiNest`), and you can add your own method to load data from other sources.

Data from multiple runs can be loaded and processed together (with optional parallelisation):

[2]:
```
file_roots = ['gaussian_2d_100nlive_5nrepeats_' + str(i) for i in range(1, 11)]
run_list = nestcheck.data_processing.batch_process_data(
    file_roots, base_dir=base_dir, parallel=True,
    process_func=nestcheck.data_processing.process_polychord_run)
```

```
HBox(children=(IntProgress(value=0, max=10), HTML(value='')))
```

2.2.2 Evidence and parameter estimation calculations from runs

Nested sampling runs in the `nestcheck` format can be easily used to make posterior inferences (see `estimators.py` for more example functions)

[3]:
```
import nestcheck.estimators as e

print('The log evidence estimate using the first run is',
      e.logz(run_list[0]))
print('The estimateed the mean of the first parameter is',
      e.param_mean(run_list[0], param_ind=0))
```

```
The log evidence estimate using the first run is -5.941675954408192
The estimateed the mean of the first parameter is 0.003249535192363336
```

You can get a pandas DataFrame of results for a list of quantities and a list of runs as follows:

[4]:
```
import nestcheck.diagnostics_tables

estimator_list = [e.logz, e.param_mean, e.param_squared_mean, e.r_mean]
# Use nestcheck's stored LaTeX format estimator names
estimator_names = [e.get_latex_name(est) for est in estimator_list]
vals_df = nestcheck.diagnostics_tables.estimator_values_df(
    run_list, estimator_list, estimator_names=estimator_names)
vals_df
```

```
HBox(children=(IntProgress(value=0, max=10), HTML(value='')))
```

[4]:
```
     $\mathrm{log} \mathcal{Z}$  $\overline{\theta_{\hat{1}}}$  \
run
0                     -5.934184                      -0.000376
1                     -5.941676                       0.003250
2                     -5.997774                      -0.017127
3                     -5.848036                      -0.026617
4                     -5.761255                      -0.004223
5                     -5.874927                      -0.012669
6                     -5.784370                       0.011934
7                     -5.880606                       0.019924
8                     -6.297974                       0.031247
9                     -5.863086                      -0.031087

     $\overline{\theta^2_{\hat{1}}}$  $\overline{|\theta|}$
run
0                          0.242457               0.625360
```

(continues on next page)

(continued from previous page)

```
1                          0.238832                0.594131
2                          0.263610                0.629681
3                          0.279740                0.654915
4                          0.230528                0.608089
5                          0.224504                0.598055
6                          0.249281                0.599508
7                          0.253596                0.616933
8                          0.264749                0.663862
9                          0.257504                0.648641
```

2.2.3 Bootstrap sampling error estimates

The sampling error on calculations from nested sampling runs can be estimated using the bootstrap approach method introduced in Higson et al. (2018) Section 4 (see the paper for more details).

[5]:
```
import pandas as pd
import nestcheck.error_analysis

bs_error_df = pd.DataFrame(columns=estimator_names)
for i, run in enumerate(run_list[:2]):  # just use the first two runs as an example
    bs_error_df.loc[i] = nestcheck.error_analysis.run_std_bootstrap(run, estimator_list,␣
↪n_simulate=100)
bs_error_df.index.name = 'run'
print('Run boostrap error estimates:')
bs_error_df
```

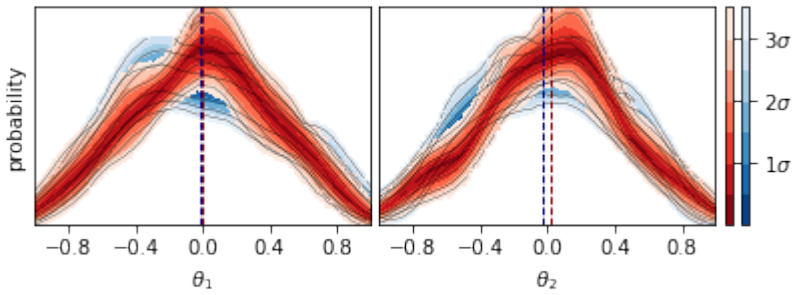
```
Run boostrap error estimates:
```

[5]:
```
     $\mathrm{log} \mathcal{Z}$  $\overline{\theta_{\hat{1}}}$  \
run
0                       0.193681                        0.026585
1                       0.214706                        0.025536

     $\overline{\theta^2_{\hat{1}}}$  $\overline{|\theta|}$
run
0                            0.016430               0.022041
1                            0.019318               0.020765
```

2.2.4 Diagrams of uncertainties on posterior distributions using bootstrap resamples

Bootstrap resamples of nested sampling runs can be used to plot numerical uncertainties on whole posterior distributions (rather than just scalar quantities) using `nestcheck`'s `bs_param_dists` function.

[6]:
```
import nestcheck.plots
%matplotlib inline

fig = nestcheck.plots.bs_param_dists(run_list[:2])
```

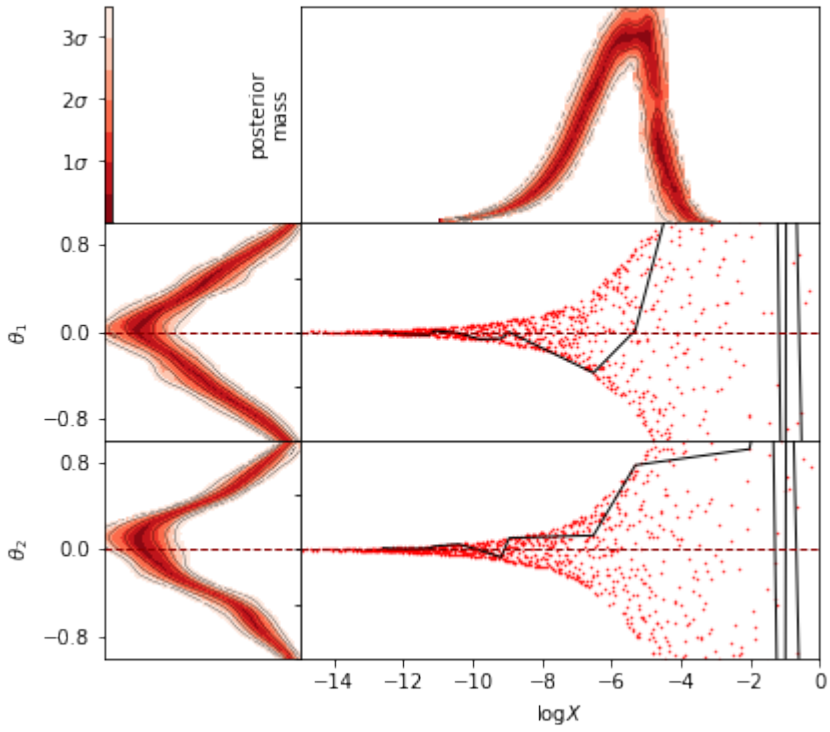


Here the dashed dark red and dark blue lines mark the estimates of the mean of each parameter for the red and blue runs respectively. For a detailed explanation of this type of diagram and its uses, see Higson et al. (2019) Section 4.1 and Figure 3.

2.2.5 Diagrams of samples in log X

The `param_logx_diagram` function plots nested sampling diagrams of the type proposed in Higson et al. (2019) Section 4.2 and shown in Figures 4 and 5.

[7]:
```
fig = nestcheck.plots.param_logx_diagram(run_list[0], logx_min=-15)
```



These diagrams illustrate the nested sampling algorithm's exponential compression of the prior by plotting samples in $\log X$, where $X(\mathcal{L}) \in [0, 1]$ is the fraction of the prior with likelihood greater than $\mathcal{L}$. The algorithm iterates towards higher likelihoods (towards lower $\log X$ values). The plots on the left are similar the distributions plotted in the previous cell, and the top right plot shows the relative posterior mass at each $\log X$ value. See Higson et al. (2019) Section 4.2 for a detailed explanation and more examples.

2.2.6 Calculating errors due to implementation-specific effects

Nested sampling software used for practical problems, such as MultiNest and PolyChord, uses numerical techniques to produce approximately uncorrelated samples within some iso-likelihood contour. However, for challenging problems - such as those with multimodal or degenerate posteriors - the software may fail to do this accurately, causing additional errors due to *implementation-specific effects* (see Higson et al. 2019 for more details).

The error due to implementation-specific effects can be estimated with `nestcheck` using the method described in Section 5 of Higson et al. (2019). This involves computing the standard deviation of results from repeated calculations, and therefore requires multiple runs (the more runs used, the more precise the results).

[8]:
```
df = nestcheck.diagnostics_tables.run_list_error_summary(run_list, estimator_list,␣
↪estimator_names, 100)
df
```

```
HBox(children=(IntProgress(value=0, max=10), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='bs values', max=10), HTML(value='')))
```

[8]:

		$\mathrm{log} \mathcal{Z}$ \
calculation type	result type	
values mean	value	-5.918389
	uncertainty	0.047744
values std	value	0.150980
	uncertainty	0.035586
bootstrap std mean	value	0.217638
	uncertainty	0.010354
implementation std	value	0.000000
	uncertainty	0.055952
implementation std frac	value	0.000000
	uncertainty	6.940037

		$\overline{\theta_{\hat{1}}}$ \
calculation type	result type	
values mean	value	-0.002574
	uncertainty	0.006330
values std	value	0.020019
	uncertainty	0.004718
bootstrap std mean	value	0.026168
	uncertainty	0.000704
implementation std	value	0.000000
	uncertainty	0.009373
implementation std frac	value	0.000000
	uncertainty	2.363760

		$\overline{\theta^2_{\hat{1}}}$ \
calculation type	result type	
values mean	value	0.250480
	uncertainty	0.005339
values std	value	0.016884
	uncertainty	0.003980
bootstrap std mean	value	0.018131

(continues on next page)

(continued from previous page)

```
                         uncertainty                          0.000481
implementation std       value                                0.000000
                         uncertainty                          0.010334
implementation std frac  value                                0.000000
                         uncertainty                          1.141478

                                        $\overline{|\theta|}$
calculation type         result type
values mean              value                       0.623917
                         uncertainty                 0.007926
values std               value                       0.025065
                         uncertainty                 0.005908
bootstrap std mean       value                       0.022129
                         uncertainty                 0.000441
implementation std       value                       0.011771
                         uncertainty                 0.014222
implementation std frac  value                       0.469616
                         uncertainty                 1.763814
```

The 2-dimensional Gaussian likelihood is unimodal and easy for PolyChord to sample, so as expected we see that the standard deviation of the result values is close to the mean bootstrap standard deviation. Consequently the estimated errors due to implementation-specific effects are low.

2.2.7 Tests for implementation specific effects using only 2 nested sampling runs

It is impossible to tell *a priori* if implementation-specific effects are present in a single nested sampling run without some additional knowledge of what the results should be. However diagnostic tests to determine if significant implementation-specific effects are present using only two nested sampling runs are proposed in Higson et al. (2019).

1. The first test divides nested sampling runs into their constituent single live point runs ("threads") and assesses whether threads within each run are correlated with each other using the Kolmogorov–Smirnov test. This yields a p-value, with $p \approx 0$ indicating implementation-specific effects are almost certainly present.
2. Secondly, the statistical distance between the uncertainty distributions (calculated from bootstrap replications) of quantities such as $\log X$ or parameter means for the two runs can be calculated. We use the Kolmogorov–Smirnov statistic as a distance measure; if this is close to 1 then there little or is no overlap between the distributions and additional errors from implementation-specific effects are likely present.

For a full explanation see the diagnostic tests paper (Higson et al. 2019)).

These statistcs can be computed for pairs of runs using `nestcheck` as:

[9]:
```
# perform error analysis on two runs
error_vals_df = nestcheck.diagnostics_tables.run_list_error_values(
    run_list[:2], estimator_list, estimator_names, thread_pvalue=True, bs_stat_dist=True,
↪ n_simulate=100)
# select only rows containing pairwise tests to output
error_vals_df.loc[pd.IndexSlice[['thread ks pvalue', 'bootstrap ks distance'], :], :]
```

```
HBox(children=(IntProgress(value=0, max=2), HTML(value='')))
```

```

```

```
HBox(children=(IntProgress(value=0, description='bs values', max=2), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='thread values', max=2), HTML(value='')))
```

[9]:
```
                              $\mathrm{log} \mathcal{Z}$  \
calculation type      run
thread ks pvalue      (1, 0)                    0.343886
bootstrap ks distance (1, 0)                    0.190000

                              $\overline{\theta_{\hat{1}}}$  \
calculation type      run
thread ks pvalue      (1, 0)                       0.556017
bootstrap ks distance (1, 0)                       0.290000

                              $\overline{\theta^2_{\hat{1}}}$  \
calculation type      run
thread ks pvalue      (1, 0)                          0.67662
bootstrap ks distance (1, 0)                          0.58000

                              $\overline{|\theta|}$
calculation type      run
thread ks pvalue      (1, 0)                0.260553
bootstrap ks distance (1, 0)                0.570000
```

As expected, the p-values are not particularly low nor are the KS distances particularly high - indicating there are no significant implementation-specific effects for the simple 2-dimensional Gaussian.

If more than two runs are provided, the above function will calculate the diagnostics for each pairwise combination.

2.3 Detailed API documentation

This page documents the different modules and functions in the `nestcheck` package.

2.3.1 data_processing

Module containing functions for loading and processing output files produced by nested sampling software.

Background: threads

`nestcheck`'s error estimates and diagnostics rely on the decomposition of a nested sampling run into multiple runs, each with a single live point. We refer to these constituent single live point runs as *threads*. See "Sampling Errors In Nested Sampling Parameter Estimation" (Higson et al. 2018) for a detailed discussion, including an algorithm for dividing nested sampling runs into their constituent threads.

Nested sampling run format

nestcheck stores nested sampling runs in a standard format as python dictionaries. For a run with n_{samp} samples, the keys are:

logl: 1d numpy array
Loglikelihood values (floats) for each sample. Shape is (n_{samp},).

thread_labels: 1d numpy array
Integer label for each point representing which thread each point belongs to. Shape is (n_{samp},). For some thread label k, the thread's start (birth) log-likelihood and end log-likelihood are given by thread_min_max[k, :].

thread_min_max: 2d numpy array
Shape is (n_{threads}, 2). Each row with index k contains the logl from within which the first point in the thread with label k was sampled (the "birth contour") and the logl of the final point in the thread. The birth contour is -inf if the thread began by sampling from the whole prior.

theta: 2d numpy array
Parameter values for samples - each row represents a sample. Shape is (n_{samp}, d) where d is number of dimensions.

nlive_array: 1d numpy array
Number of live points present between the previous point and this point.

output: dict (optional)
Dict containing extra information about the run.

Samples are arranged in ascending order of logl.

Processing nested sampling software output

To process output files for a nested sampling run into the format described above, the following information is required:

- Samples' loglikelihood values;
- Samples' parameter values;
- Information allowing decomposition into threads and identifying each thread's birth contour (starting logl).

The first two items are self-explanatory, but the latter is more challenging as it can take different formats and may not be provided by all nested sampling software packages.

Sufficient information for thread decomposition and calculating the number of live points (including for dynamic nested sampling) is provided by a list of the loglikelihoods from within which each point was sampled (the points' birth contours). This is output by `PolyChord` >= v1.13 and `MultiNest` >= v3.11, and is used in the output processing for these packages via the `birth_inds_given_contours` and `threads_given_birth_inds` functions. Also sufficient is a list of the indexes of the point which was removed at the step when each point was sampled ("birth indexes"), as this can be mapped to the birth contours and vice versa.

`process_dynesty_run` does not require the `birth_inds_given_contours` and `threads_given_birth_inds` functions as `dynesty` results objects already include thread labels via their `samples_id` property. If the `dynesty` run is dynamic, the `batch_bounds` property is need to determine the threads' starting birth contours.

Adding a new processing function for another nested sampling package

You can add new functions to process output from other nested sampling software, provided the output files include the required information for decomposition into threads. Depending on how this information is provided you may be able to adapt `process_polychord_run` or `process_dynesty_run`. If thread decomposition information if provided in a different format, you will have to write your own helper functions to process the output into the `nestcheck` dictionary format described above.

`nestcheck.data_processing.`**`batch_process_data`**(*file_roots*, ***kwargs*)

Process output from many nested sampling runs in parallel with optional error handling and caching.

The result can be cached using the 'save_name', 'save' and 'load' kwargs (by default this is not done). See save_load_result docstring for more details.

Remaining kwargs passed to parallel_utils.parallel_apply (see its docstring for more details).

Parameters

file_roots: list of strs
file_roots for the runs to load.

base_dir: str, optional
path to directory containing files.

process_func: function, optional
function to use to process the data.

func_kwargs: dict, optional
additional keyword arguments for process_func.

errors_to_handle: error or tuple of errors, optional
which errors to catch when they occur in processing rather than raising.

save_name: str or None, optional
See nestcheck.io_utils.save_load_result.

save: bool, optional
See nestcheck.io_utils.save_load_result.

load: bool, optional
See nestcheck.io_utils.save_load_result.

overwrite_existing: bool, optional
See nestcheck.io_utils.save_load_result.

Returns

list of ns_run dicts
List of nested sampling runs in dict format (see the module docstring for more details).

`nestcheck.data_processing.`**`birth_inds_given_contours`**(*birth_logl_arr*, *logl_arr*, ***kwargs*)

Maps the iso-likelihood contours on which points were born to the index of the dead point on this contour.

MultiNest and PolyChord use different values to identify the inital live points which were sampled from the whole prior (PolyChord uses -1e+30 and MultiNest -0.17976931348623157E+309). However in each case the first dead point must have been sampled from the whole prior, so for either package we can use

init_birth = birth_logl_arr[0]

If there are many points with the same logl_arr and dup_assert is False, these points are randomly assigned an order (to ensure results are consistent, random seeding is used).

Parameters

logl_arr: 1d numpy array
logl values of each point.

birth_logl_arr: 1d numpy array
Birth contours - i.e. logl values of the iso-likelihood contour from within each point was sampled (on which it was born).

dup_assert: bool, optional
See ns_run_utils.check_ns_run_logls docstring.

dup_warn: bool, optional
See ns_run_utils.check_ns_run_logls docstring.

Returns

birth_inds: 1d numpy array of ints
Step at which each element of logl_arr was sampled. Points sampled from the whole prior are assigned value -1.

nestcheck.data_processing.**process_dynesty_run**(*results*)

Transforms results from a dynesty run into the nestcheck dictionary format for analysis. This function has been tested with dynesty v9.2.0.

Note that the nestcheck point weights and evidence will not be exactly the same as the dynesty ones as nestcheck calculates logX volumes more precisely (using the trapezium rule).

This function does not require the birth_inds_given_contours and threads_given_birth_inds functions as dynesty results objects already include thread labels via their samples_id property. If the dynesty run is dynamic, the batch_bounds property is need to determine the threads' starting birth contours.

Parameters

results: dynesty results object
N.B. the remaining live points at termination must be included in the results (dynesty samplers' run_nested method does this if add_live_points=True - its default value).

Returns

ns_run: dict
Nested sampling run dict (see the module docstring for more details).

nestcheck.data_processing.**process_error_helper**(*root*, *base_dir*, *process_func*, *errors_to_handle=()*, ***func_kwargs*)

Wrapper which applies process_func and handles some common errors so one bad run does not spoil the whole batch.

Useful errors to handle include:

OSError: if you are not sure if all the files exist AssertionError: if some of the many assertions fail for known reasons; for example is there are occasional problems decomposing runs into threads due to limited numerical precision in logls.

Parameters

root: str
File root.

base_dir: str
Directory containing file.

process_func: func
Function for processing file.

errors_to_handle: error type or tuple of error types
Errors to catch without throwing an exception.

func_kwargs: dict
Kwargs to pass to process_func.

Returns

run: dict
Nested sampling run dict (see the module docstring for more details) or, if an error occured, a dict containing its type and the file root.

nestcheck.data_processing.**process_multinest_run**(*file_root*, *base_dir*, ***kwargs*)

Loads data from a MultiNest run into the nestcheck dictionary format for analysis.

N.B. producing required output file containing information about the iso-likelihood contours within which points were sampled (where they were "born") requies MultiNest version 3.11 or later.

Parameters

file_root: str
Root name for output files. When running MultiNest, this is determined by the nest_root parameter.

base_dir: str
Directory containing output files. When running MultiNest, this is determined by the nest_root parameter.

kwargs: dict, optional
Passed to ns_run_utils.check_ns_run (via process_samples_array)

Returns

ns_run: dict
Nested sampling run dict (see the module docstring for more details).

nestcheck.data_processing.**process_polychord_run**(*file_root*, *base_dir*, *process_stats_file=True*, ***kwargs*)

Loads data from a PolyChord run into the nestcheck dictionary format for analysis.

N.B. producing required output file containing information about the iso-likelihood contours within which points were sampled (where they were "born") requies PolyChord version v1.13 or later and the setting write_dead=True.

Parameters

file_root: str
Root for run output file names (PolyChord file_root setting).

base_dir: str
Directory containing data (PolyChord base_dir setting).

process_stats_file: bool, optional
Should PolyChord's <root>.stats file be processed? Set to False if you don't have the <root>.stats file (such as if PolyChord was run with write_stats=False).

kwargs: dict, optional
Options passed to ns_run_utils.check_ns_run.

Returns

ns_run: dict
Nested sampling run dict (see the module docstring for more details).

nestcheck.data_processing.**process_polychord_stats**(*file_root*, *base_dir*)

Reads a PolyChord <root>.stats output file and returns the information contained in a dictionary.

Parameters

file_root: str
Root for run output file names (PolyChord file_root setting).

base_dir: str
Directory containing data (PolyChord base_dir setting).

Returns

output: dict
See PolyChord documentation for more details.

nestcheck.data_processing.**process_samples_array**(*samples*, ***kwargs*)

Convert an array of nested sampling dead and live points of the type produced by PolyChord and MultiNest into a nestcheck nested sampling run dictionary.

Parameters

samples: 2d numpy array
Array of dead points and any remaining live points at termination. Has #parameters + 2 columns: param_1, param_2, ... , logl, birth_logl

kwargs: dict, optional
Options passed to birth_inds_given_contours

Returns

ns_run: dict
Nested sampling run dict (see the module docstring for more details). Only contains information in samples (not additional optional output key).

nestcheck.data_processing.**sample_less_than_condition**(*choices_in*, *condition*)

Creates a random sample from choices without replacement, subject to the condition that each element of the output is greater than the corresponding element of the condition array.

condition should be in ascending order.

nestcheck.data_processing.**threads_given_birth_inds**(*birth_inds*)

Divides a nested sampling run into threads, using info on the indexes at which points were sampled. See "Sampling errors in nested sampling parameter estimation" (Higson et al. 2018) for more information.

Parameters

birth_inds: 1d numpy array
Indexes of the iso-likelihood contours from within which each point was sampled ("born").

Returns

thread_labels: 1d numpy array of ints
labels of the thread each point belongs to.

2.3.2 ns_run_utils

Functions for performing basic operations on nested sampling runs; such as working out point weights and splitting and combining runs.

Nested sampling runs are stored in a standard format as python dictionaries (see the `data_processing` module docstring for more details).

`nestcheck.ns_run_utils.array_given_run(ns_run)`

Converts information on samples in a nested sampling run dictionary into a numpy array representation. This allows fast addition of more samples and recalculation of nlive.

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

Returns

samples: 2d numpy array
Array containing columns [logl, thread label, change in nlive at sample, (thetas)] with each row representing a single sample.

`nestcheck.ns_run_utils.check_ns_run(run, dup_assert=False, dup_warn=False)`

Checks a nestcheck format nested sampling run dictionary has the expected properties (see the data_processing module docstring for more details).

Parameters

run: dict
nested sampling run to check.

dup_assert: bool, optional
See check_ns_run_logls docstring.

dup_warn: bool, optional
See check_ns_run_logls docstring.

Raises

AssertionError
if run does not have expected properties.

`nestcheck.ns_run_utils.check_ns_run_logls(run, dup_assert=False, dup_warn=False)`

Check run logls are unique and in the correct order.

Parameters

run: dict
nested sampling run to check.

dup_assert: bool, optional
Whether to raise and AssertionError if there are duplicate logl values.

dup_warn: bool, optional
Whether to give a UserWarning if there are duplicate logl values (only used if dup_assert is False).

Raises

AssertionError
if run does not have expected properties.

nestcheck.ns_run_utils.**check_ns_run_members**(*run*)

Check nested sampling run member keys and values.

Parameters

run: dict
nested sampling run to check.

Raises

AssertionError
if run does not have expected properties.

nestcheck.ns_run_utils.**check_ns_run_threads**(*run*)

Check thread labels and thread_min_max have expected properties.

Parameters

run: dict
Nested sampling run to check.

Raises

AssertionError
If run does not have expected properties.

nestcheck.ns_run_utils.**combine_ns_runs**(*run_list_in*, ***kwargs*)

Combine a list of complete nested sampling run dictionaries into a single ns run.

Input runs must contain any repeated threads.

Parameters

run_list_in: list of dicts
List of nested sampling runs in dict format (see data_processing module docstring for more details).

kwargs: dict, optional
Options for check_ns_run.

Returns

run: dict
Nested sampling run dict (see data_processing module docstring for more details).

nestcheck.ns_run_utils.**combine_threads**(*threads*, *assert_birth_point=False*)

Combine list of threads into a single ns run. This is different to combining runs as repeated threads are allowed, and as some threads can start from log-likelihood contours on which no dead point in the run is present.

Note that if all the thread labels are not unique and in ascending order, the output will fail check_ns_run. However provided the thread labels are not used it will work ok for calculations based on nlive, logl and theta.

Parameters

threads: list of dicts
List of nested sampling run dicts, each representing a single thread.

assert_birth_point: bool, optional
Whether or not to assert there is exactly one point present in the run with the log-likelihood at which each point was born. This is not true for bootstrap resamples of runs, where birth points may be repeated or not present at all.

Returns

run: dict
Nested sampling run dict (see data_processing module docstring for more details).

nestcheck.ns_run_utils.**dict_given_run_array**(*samples*, *thread_min_max*)

Converts an array of information about samples back into a nested sampling run dictionary (see data_processing module docstring for more details).

N.B. the output dict only contains the following keys: 'logl', 'thread_label', 'nlive_array', 'theta'. Any other keys giving additional information about the run output cannot be reproduced from the function arguments, and are therefore ommitted.

Parameters

samples: numpy array
Numpy array containing columns [logl, thread label, change in nlive at sample, (thetas)] with each row representing a single sample.

thread_min_max': numpy array, optional
2d array with a row for each thread containing the likelihoods at which it begins and ends. Needed to calculate nlive_array (otherwise this is set to None).

Returns

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

nestcheck.ns_run_utils.**get_logw**(*ns_run*, *simulate=False*)

Calculates the log posterior weights of the samples (using logarithms to avoid overflow errors with very large or small values).

Uses the trapezium rule such that the weight of point i is

$$w_i = \mathcal{L}_i(X_{i-1} - X_{i+1})/2$$

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

simulate: bool, optional
Should log prior volumes logx be simulated from their distribution (if false their expected values are used).

Returns

logw: 1d numpy array
Log posterior masses of points.

nestcheck.ns_run_utils.**get_logx**(*nlive*, *simulate=False*)

Returns a logx vector showing the expected or simulated logx positions of points.

The shrinkage factor between two points

$$t_i = X_{i-1}/X_i$$

is distributed as the largest of n_i uniform random variables between 1 and 0, where n_i is the local number of live points.

We are interested in

$$\log(t_i) = \log X_{i-1} - \log X_i$$

which has expected value $-1/n_i$.

Parameters

nlive_array: 1d numpy array
Ordered local number of live points present at each point's iso-likelihood contour.

simulate: bool, optional
Should log prior volumes logx be simulated from their distribution (if False their expected values are used).

Returns

logx: 1d numpy array
log X values for points.

nestcheck.ns_run_utils.**get_run_threads**(*ns_run*)

Get the individual threads from a nested sampling run.

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

Returns

threads: list of numpy array
Each thread (list element) is a samples array containing columns [logl, thread label, change in nlive at sample, (thetas)] with each row representing a single sample.

nestcheck.ns_run_utils.**get_w_rel**(*ns_run*, *simulate=False*)

Get the relative posterior weights of the samples, normalised so the maximum sample weight is 1. This is calculated from get_logw with protection against numerical overflows.

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

simulate: bool, optional
See the get_logw docstring for more details.

Returns

w_rel: 1d numpy array
Relative posterior masses of points.

nestcheck.ns_run_utils.**log_subtract**(*loga*, *logb*)

Numerically stable method for avoiding overflow errors when calculating $\log(a-b)$, given $\log(a)$, $\log(a)$ and that $a > b$.

See https://hips.seas.harvard.edu/blog/2013/01/09/computing-log-sum-exp/ for more details.

Parameters

loga: float
logb: float
Must be less than loga.

Returns

log(a - b): float

nestcheck.ns_run_utils.**run_estimators**(*ns_run*, *estimator_list*, *simulate=False*)

Calculates values of list of quantities (such as the Bayesian evidence or mean of parameters) for a single nested sampling run.

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

estimator_list: list of functions for estimating quantities from nested
sampling runs. Example functions can be found in estimators.py. Each should have arguments: func(ns_run, logw=None).

simulate: bool, optional
See get_logw docstring.

Returns

output: 1d numpy array
Calculation result for each estimator in estimator_list.

2.3.3 error_analysis

Sampling error estimates and diagnostic tests for nested sampling runs.

nestcheck.error_analysis.**bootstrap_resample_run**(*ns_run*, *threads=None*, *ninit_sep=False*, *random_seed=False*)

Bootstrap resamples threads of nested sampling run, returning a new (resampled) nested sampling run.

Get the individual threads for a nested sampling run.

Parameters

ns_run: dict
Nested sampling run dictionary.

threads: None or list of numpy arrays, optional
ninit_sep: bool
For dynamic runs: resample initial threads and dynamically added threads separately. Useful when there are only a few threads which start by sampling the whole prior, as errors occur if none of these are included in the bootstrap resample.

random_seed: None, bool or int, optional
Set numpy random seed. Default is to use None (so a random seed is chosen from the computer's internal state) to ensure reliable results when multiprocessing. Can set to an integer or to False to not edit the seed.

Returns

ns_run_temp: dict
Nested sampling run dictionary.

nestcheck.error_analysis.**implementation_std**(*vals_std*, *vals_std_u*, *bs_std*, *bs_std_u*, ***kwargs*)

Estimates varaition of results due to implementation-specific effects. See 'nestcheck: diagnostic tests for nested sampling calculations' (Higson et al. 2019) for more details.

Uncertainties on the output are calculated numerically using the fact that (from central limit theorem) our uncertainties on vals_std and bs_std are (approximately) normally distributed. This is needed as results from standard error propagation techniques are not valid when the uncertainties are not small compared to the result.

Parameters

vals_std: numpy array
Standard deviations of results from repeated calculations.

vals_std_u: numpy array
1σ uncertainties on vals_std_u.

bs_std: numpy array
Bootstrap error estimates. Each element should correspond to the same element in vals_std.

bs_std_u: numpy array
1σ uncertainties on vals_std_u.

nsim: int, optional
Number of simulations to use to numerically calculate the uncertainties on the estimated implementation-specific effects.

random_seed: int or None, optional
Numpy random seed. Use to get reproducible uncertainties on the output.

Returns

imp_std: numpy array
Estimated standard deviation of results due to implementation-specific effects.

imp_std_u: numpy array
1σ uncertainties on imp_std.

imp_frac: numpy array
imp_std as a fraction of vals_std.

imp_frac_u:
1σ uncertainties on imp_frac.

nestcheck.error_analysis.**pairwise_distances**(*dist_list*, *earth_mover_dist=True*, *energy_dist=True*)

Applies statistical_distances to each unique pair of distribution samples in dist_list.

Parameters

dist_list: list of 1d arrays
earth_mover_dist: bool, optional
Passed to statistical_distances.

energy_dist: bool, optional
Passed to statistical_distances.

Returns

ser: pandas Series object
Values are statistical distances. Index levels are: calculation type: name of statistical distance. run: tuple containing the index in dist_list of the pair of samples arrays from which the statistical distance was computed.

nestcheck.error_analysis.**run_bootstrap_values**(*ns_run*, *estimator_list*, ***kwargs*)

Uses bootstrap resampling to calculate an estimate of the standard deviation of the distribution of sampling errors (the uncertainty on the calculation) for a single nested sampling run.

For more details about bootstrap resampling for estimating sampling errors see 'Sampling errors in nested sampling parameter estimation' (Higson et al. 2018).

Parameters

ns_run: dict
Nested sampling run dictionary.

estimator_list: list of functions for estimating quantities (such as the
Bayesian evidence or mean of parameters) from nested sampling runs. Example functions can be found in estimators.py. Each should have arguments: func(ns_run, logw=None)

n_simulate: int
ninit_sep: bool, optional
For dynamic runs: resample initial threads and dynamically added threads separately. Useful when there are only a few threads which start by sampling the whole prior, as errors occur if none of these are included in the bootstrap resample.

flip_skew: bool, optional
Determine if distribution of bootstrap values should be flipped about its mean to better represent our probability distribution on the true value - see "Bayesian astrostatistics: a backward look to the future" (Loredo, 2012 Figure 2) for an explanation. If true, the samples X are mapped to $2\mu - X$, where μ is the mean sample value. This leaves the mean and standard deviation unchanged.

random_seeds: list, optional
list of random_seed arguments for bootstrap_resample_run. Defaults to range(n_simulate) in order to give reproducible results.

Returns

output: 1d numpy array
Sampling error on calculation result for each estimator in estimator_list.

nestcheck.error_analysis.**run_ci_bootstrap**(*ns_run*, *estimator_list*, ***kwargs*)

Uses bootstrap resampling to calculate credible intervals on the distribution of sampling errors (the uncertainty on the calculation) for a single nested sampling run.

For more details about bootstrap resampling for estimating sampling errors see 'Sampling errors in nested sampling parameter estimation' (Higson et al. 2018).

Parameters

ns_run: dict
Nested sampling run dictionary.

estimator_list: list of functions for estimating quantities (such as the
Bayesian evidence or mean of parameters) from nested sampling runs. Example functions can be found in estimators.py. Each should have arguments: func(ns_run, logw=None)

cred_int: float
n_simulate: int
ninit_sep: bool, optional

Returns

output: 1d numpy array
Credible interval on sampling error on calculation result for each estimator in estimator_list.

nestcheck.error_analysis.**run_std_bootstrap**(*ns_run*, *estimator_list*, ***kwargs*)

Uses bootstrap resampling to calculate an estimate of the standard deviation of the distribution of sampling errors (the uncertainty on the calculation) for a single nested sampling run.

For more details about bootstrap resampling for estimating sampling errors see 'Sampling errors in nested sampling parameter estimation' (Higson et al. 2018).

Parameters

ns_run: dict
Nested sampling run dictionary.

estimator_list: list of functions for estimating quantities (such as the
Bayesian evidence or mean of parameters) from nested sampling runs. Example functions can be found in estimators.py. Each should have arguments: func(ns_run, logw=None)

kwargs: dict
kwargs for run_bootstrap_values

Returns

output: 1d numpy array
Sampling error on calculation result for each estimator in estimator_list.

nestcheck.error_analysis.**run_std_simulate**(*ns_run*, *estimator_list*, *n_simulate=None*)

Uses the 'simulated weights' method to calculate an estimate of the standard deviation of the distribution of sampling errors (the uncertainty on the calculation) for a single nested sampling run.

Note that the simulated weights method is not accurate for parameter estimation calculations.

For more details about the simulated weights method for estimating sampling errors see 'Sampling errors in nested sampling parameter estimation' (Higson et al. 2018).

Parameters

ns_run: dict
Nested sampling run dictionary.

estimator_list: list of functions for estimating quantities (such as the
bayesian evidence or mean of parameters) from nested sampling runs. Example functions can be found in estimators.py. Each should have arguments: func(ns_run, logw=None)

n_simulate: int

Returns

output: 1d numpy array
Sampling error on calculation result for each estimator in estimator_list.

nestcheck.error_analysis.**run_thread_values**(*run*, *estimator_list*)

Helper function for parallelising thread_values_df.

Parameters

ns_run: dict
Nested sampling run dictionary.

estimator_list: list of functions

Returns

vals_array: numpy array
Array of estimator values for each thread. Has shape (len(estimator_list), len(theads)).

nestcheck.error_analysis.**statistical_distances**(*samples1*, *samples2*, *earth_mover_dist=True*, *energy_dist=True*)

Compute measures of the statistical distance between samples.

Parameters

samples1: 1d array
samples2: 1d array
earth_mover_dist: bool, optional
Whether or not to compute the Earth mover's distance between the samples.

energy_dist: bool, optional
Whether or not to compute the energy distance between the samples.

Returns

1d array

2.3.4 diagnostics_tables

High-level functions for calculating results of error analysis and diagnostic tests for batches of nested sampling runs.

nestcheck.diagnostics_tables.**bs_values_df**(*run_list*, *estimator_list*, *estimator_names*, *n_simulate*, ***kwargs*)

Computes a data frame of bootstrap resampled values.

Parameters

run_list: list of dicts
List of nested sampling run dicts.

estimator_list: list of functions
Estimators to apply to runs.

estimator_names: list of strs
Name of each func in estimator_list.

n_simulate: int
Number of bootstrap replications to use on each run.

kwargs:
Kwargs to pass to parallel_apply.

Returns

bs_values_df: pandas data frame
Columns represent estimators and rows represent runs. Each cell contains a 1d array of bootstrap resampled values for the run and estimator.

nestcheck.diagnostics_tables.**error_values_summary**(*error_values*, ***summary_df_kwargs*)

Get summary statistics about calculation errors, including estimated implementation errors.

Parameters

error_values: pandas DataFrame
Of format output by run_list_error_values (look at it for more details).

summary_df_kwargs: dict, optional
See pandas_functions.summary_df docstring for more details.

Returns

df: pandas DataFrame
Table showing means and standard deviations of results and diagnostics for the different runs. Also contains estimated numerical uncertainties on results.

nestcheck.diagnostics_tables.**estimator_values_df**(*run_list*, *estimator_list*, ***kwargs*)

Get a dataframe of estimator values.

NB when parallelised the results will not be produced in order (so results from some run number will not necessarily correspond to that number run in run_list).

Parameters

run_list: list of dicts
List of nested sampling run dicts.

estimator_list: list of functions
Estimators to apply to runs.

estimator_names: list of strs, optional
Name of each func in estimator_list.

parallel: bool, optional
Whether or not to parallelise - see parallel_utils.parallel_apply.

save_name: str or None, optional
See nestcheck.io_utils.save_load_result.

save: bool, optional
See nestcheck.io_utils.save_load_result.

load: bool, optional
See nestcheck.io_utils.save_load_result.

overwrite_existing: bool, optional
See nestcheck.io_utils.save_load_result.

Returns

df: pandas DataFrame
Results table showing calculation values and diagnostics. Rows show different runs. Columns have titles given by estimator_names and show results for the different functions in estimators_list.

nestcheck.diagnostics_tables.**pairwise_dists_on_cols**(*df_in*, *earth_mover_dist=True*, *energy_dist=True*)

Computes pairwise statistical distance measures.

Parameters

df_in: pandas data frame
Columns represent estimators and rows represent runs. Each data frane element is an array of values which are used as samples in the distance measures.

earth_mover_dist: bool, optional
Passed to error_analysis.pairwise_distances.

energy_dist: bool, optional
Passed to error_analysis.pairwise_distances.

Returns

df: pandas data frame with kl values for each pair.

nestcheck.diagnostics_tables.**run_list_error_summary**(*run_list*, *estimator_list*, *estimator_names*, *n_simulate*, ***kwargs*)

Wrapper which runs run_list_error_values then applies error_values summary to the resulting dataframe. See the docstrings for those two funcions for more details and for descriptions of parameters and output.

nestcheck.diagnostics_tables.**run_list_error_values**(*run_list*, *estimator_list*, *estimator_names*, *n_simulate=100*, ***kwargs*)

Gets a data frame with calculation values and error diagnostics for each run in the input run list.

NB when parallelised the results will not be produced in order (so results from some run number will not nessesarily correspond to that number run in run_list).

Parameters

run_list: list of dicts
List of nested sampling run dicts.

estimator_list: list of functions
Estimators to apply to runs.

estimator_names: list of strs
Name of each func in estimator_list.

n_simulate: int, optional
Number of bootstrap replications to use on each run.

thread_pvalue: bool, optional
Whether or not to compute KS test diaganostic for correlations between threads within a run.

bs_stat_dist: bool, optional
Whether or not to compute statistical distance between bootstrap error distributions diaganostic.

parallel: bool, optional
Whether or not to parallelise - see parallel_utils.parallel_apply.

save_name: str or None, optional
See nestcheck.io_utils.save_load_result.

save: bool, optional
See nestcheck.io_utils.save_load_result.

load: bool, optional
See nestcheck.io_utils.save_load_result.

overwrite_existing: bool, optional
See nestcheck.io_utils.save_load_result.

Returns

df: pandas DataFrame
Results table showing calculation values and diagnostics. Rows show different runs (or pairs of runs for pairwise comparisons). Columns have titles given by estimator_names and show results for the different functions in estimators_list.

nestcheck.diagnostics_tables.**thread_values_df**(*run_list*, *estimator_list*, *estimator_names*, ***kwargs*)

Calculates estimator values for the constituent threads of the input runs.

Parameters

run_list: list of dicts
List of nested sampling run dicts.

estimator_list: list of functions
Estimators to apply to runs.

estimator_names: list of strs
Name of each func in estimator_list.

kwargs:
Kwargs to pass to parallel_apply.

Returns

df: pandas data frame
Columns represent estimators and rows represent runs. Each cell contains a 1d numpy array with length equal to the number of threads in the run, containing the results from evaluating the estimator on each thread.

2.3.5 plots

Functions for diagnostic plots of nested sampling runs.

Includes functions for plots described 'nestcheck: diagnostic tests for nested sampling calculations' (Higson et al. 2019).

nestcheck.plots.**alternate_helper**(*x*, *alt_samps*, *func=None*)

Helper function for making fgivenx plots of functions with 2 array arguments of variable lengths.

nestcheck.plots.**average_by_key**(*dict_in*, *key*)

Helper function for plot_run_nlive.

Try returning the average of dict_in[key] and, if this does not work or if key is None, return average of whole dict.

Parameters

dict_in: dict
Values should be arrays.

key: str

Returns

average: float

nestcheck.plots.**bs_param_dists**(*run_list*, ***kwargs*)

Creates posterior distributions and their bootstrap error functions for input runs and estimators.

For a more detailed description and some example use cases, see 'nestcheck: diagnostic tests for nested sampling calculations' (Higson et al. 2019).

Parameters

run_list: dict or list of dicts
Nested sampling run(s) to plot.

fthetas: list of functions, optional
Quantities to plot. Each must map a 2d theta array to 1d ftheta array - i.e. map every sample's theta vector (every row) to a scalar quantity. E.g. use lambda x: x[:, 0] to plot the first parameter.

labels: list of strs, optional
Labels for each ftheta.

ftheta_lims: list, optional
Plot limits for each ftheta.

n_simulate: int, optional
Number of bootstrap replications to be used for the fgivenx distributions.

random_seed: int, optional
Seed to make sure results are consistent and fgivenx caching can be used.

figsize: tuple, optional
Matplotlib figsize in (inches).

nx: int, optional
Size of x-axis grid for fgivenx plots.

ny: int, optional
Size of y-axis grid for fgivenx plots.

cache: str or None
Root for fgivenx caching (no caching if None).

parallel: bool, optional
fgivenx parallel option.

rasterize_contours: bool, optional
fgivenx rasterize_contours option.

tqdm_kwargs: dict, optional
Keyword arguments to pass to the tqdm progress bar when it is used in fgivenx while plotting contours.

Returns

fig: matplotlib figure

nestcheck.plots.**kde_plot_df**(*df*, *xlims=None*, ***kwargs*)

Plots kde estimates of distributions of samples in each cell of the input pandas DataFrame.

There is one subplot for each dataframe column, and on each subplot there is one kde line.

Parameters

df: pandas data frame
Each cell must contain a 1d numpy array of samples.

xlims: dict, optional
Dictionary of xlimits - keys are column names and values are lists of length 2.

num_xticks: int, optional
Number of xticks on each subplot.

figsize: tuple, optional
Size of figure in inches.

nrows: int, optional
Number of rows of subplots.

ncols: int, optional
Number of columns of subplots.

normalize: bool, optional
If true, kde plots are normalized to have the same area under their curves. If False, their max value is set to 1.

legend: bool, optional
Should a legend be added?

legend_kwargs: dict, optional
Additional kwargs for legend.

Returns

fig: matplotlib figure

nestcheck.plots.**param_logx_diagram**(*run_list*, ***kwargs*)

Creates diagrams of a nested sampling run's evolution as it iterates towards higher likelihoods, expressed as a function of log X, where X(L) is the fraction of the prior volume with likelihood greater than some value L.

For a more detailed description and some example use cases, see 'nestcheck: diagnostic tests for nested sampling calculations" (Higson et al. 2019).

Parameters

run_list: dict or list of dicts
Nested sampling run(s) to plot.

fthetas: list of functions, optional
Quantities to plot. Each must map a 2d theta array to 1d ftheta array - i.e. map every sample's theta vector (every row) to a scalar quantity. E.g. use lambda x: x[:, 0] to plot the first parameter.

labels: list of strs, optional
Labels for each ftheta.

ftheta_lims: dict, optional
Plot limits for each ftheta.

plot_means: bool, optional
Should the mean value of each ftheta be plotted?

n_simulate: int, optional
Number of bootstrap replications to use for the fgivenx distributions.

random_seed: int, optional
Seed to make sure results are consistent and fgivenx caching can be used.

logx_min: float, optional
Lower limit of logx axis.

figsize: tuple, optional
Matplotlib figure size (in inches).

colors: list of strs, optional
Colors to plot run scatter plots with.

colormaps: list of strs, optional
Colormaps to plot run fgivenx plots with.

npoints: int, optional
How many points to have in the logx array used to calculate and plot analytical weights.

cache: str or None
Root for fgivenx caching (no caching if None).

parallel: bool, optional
fgivenx parallel optional

point_size: float, optional
size of markers on scatter plot (in pts)

thin: float, optional
factor by which to reduce the number of samples before plotting the scatter plot. Must be in half-closed interval (0, 1].

rasterize_contours: bool, optional
fgivenx rasterize_contours option.

tqdm_kwargs: dict, optional
Keyword arguments to pass to the tqdm progress bar when it is used in fgivenx while plotting contours.

Returns

fig: matplotlib figure

nestcheck.plots.**plot_bs_dists**(*run*, *fthetas*, *axes*, ***kwargs*)

Helper function for plotting uncertainties on posterior distributions using bootstrap resamples and the fgivenx module. Used by bs_param_dists and param_logx_diagram.

Parameters

run: dict
Nested sampling run to plot.

fthetas: list of functions
Quantities to plot. Each must map a 2d theta array to 1d ftheta array - i.e. map every sample's theta vector (every row) to a scalar quantity. E.g. use lambda x: x[:, 0] to plot the first parameter.

axes: list of matplotlib axis objects
ftheta_lims: list, optional
Plot limits for each ftheta.

n_simulate: int, optional
Number of bootstrap replications to use for the fgivenx distributions.

colormap: matplotlib colormap
Colors to plot fgivenx distribution.

mean_color: matplotlib color as str
Color to plot mean of each parameter. If None (default) means are not plotted.

nx: int, optional
Size of x-axis grid for fgivenx plots.

ny: int, optional
Size of y-axis grid for fgivenx plots.

cache: str or None
Root for fgivenx caching (no caching if None).

parallel: bool, optional
fgivenx parallel option.

rasterize_contours: bool, optional
fgivenx rasterize_contours option.

smooth: bool, optional
fgivenx smooth option.

flip_axes: bool, optional
Whether or not plot should be rotated 90 degrees anticlockwise onto its side.

tqdm_kwargs: dict, optional
Keyword arguments to pass to the tqdm progress bar when it is used in fgivenx while plotting contours.

Returns

cbar: matplotlib colorbar
For use in higher order functions.

nestcheck.plots.**plot_run_nlive**(*method_names*, *run_dict*, ***kwargs*)

Plot the allocations of live points as a function of logX for the input sets of nested sampling runs of the type used in the dynamic nested sampling paper (Higson et al. 2019). Plots also include analytically calculated distributions of relative posterior mass and relative posterior mass remaining.

Parameters

method_names: list of strs
run_dict: dict of lists of nested sampling runs.
Keys of run_dict must be method_names.

logx_given_logl: function, optional
For mapping points' logl values to logx values. If not specified the logx coordinates for each run are estimated using its numbers of live points.

logl_given_logx: function, optional
For calculating the relative posterior mass and posterior mass remaining at each logx coordinate.

logx_min: float, optional
Lower limit of logx axis. If not specified this is set to the lowest logx reached by any of the runs.

ymax: bool, optional
Maximum value for plot's nlive axis (yaxis).

npoints: int, optional
Number of points to have in the fgivenx plot grids.

figsize: tuple, optional
Size of figure in inches.

post_mass_norm: str or None, optional
Specify method_name for runs use form normalising the analytic posterior mass curve. If None, all runs are used.

cum_post_mass_norm: str or None, optional
Specify method_name for runs use form normalising the analytic cumulative posterior mass remaining curve. If None, all runs are used.

Returns

fig: matplotlib figure

nestcheck.plots.**rel_posterior_mass**(*logx*, *logl*)

Calculate the relative posterior mass for some array of logx values given the likelihood, prior and number of dimensions. The posterior mass at each logX value is proportional to L(X)X, where L(X) is the likelihood. The weight is returned normalized so that the integral of the weight with respect to logX is 1.

Parameters

logx: 1d numpy array
Logx values at which to calculate posterior mass.

logl: 1d numpy array
Logl values corresponding to each logx (same shape as logx).

Returns

w_rel: 1d numpy array
Relative posterior mass at each input logx value.

nestcheck.plots.**weighted_1d_gaussian_kde**(*x*, *samples*, *weights*)

Gaussian kde with weighted samples (1d only). Uses Scott bandwidth factor.

When all the sample weights are equal, this is equivalent to

kde = scipy.stats.gaussian_kde(theta) return kde(x)

When the weights are not all equal, we compute the effective number of samples as the information content (Shannon entropy)

nsamp_eff = exp(- sum_i (w_i log(w_i)))

Alternative ways to estimate nsamp_eff include Kish's formula

nsamp_eff = (sum_i w_i) ** 2 / (sum_i w_i ** 2)

See https://en.wikipedia.org/wiki/Effective_sample_size and "Effective sample size for importance sampling based on discrepancy measures" (Martino et al. 2017) for more information.

Parameters

x: 1d numpy array
Coordinates at which to evaluate the kde.

samples: 1d numpy array
Samples from which to calculate kde.

weights: 1d numpy array of same shape as samples
Weights of each point. Need not be normalised as this is done inside the function.

Returns

result: 1d numpy array of same shape as x
Kde evaluated at x values.

2.3.6 estimators

Functions for estimating quantities from nested sampling runs. Each estimator function should have arguments:

```
def estimator_func(self, ns_run, logw=None, simulate=False):
    ...
```

Any additional arguments required for the function should be keyword arguments.

The `logw` argument allows the log weights for the points in the run to be provided - this is useful if many estimators are being calculated from the same run as it allows `logw` to only be calculated once. If it is not specified, `logw` is calculated from the run when required.

The simulate argument is passed to `ns_run_utils.get_logw`, and is only used if the function needs to calculate `logw`.

nestcheck.estimators.**count_samples**(*ns_run*, ***kwargs*)

Number of samples in run.

Unlike most estimators this does not require log weights, but for convenience will not throw an error if they are specified.

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

Returns

int

nestcheck.estimators.**evidence**(*ns_run*, *logw=None*, *simulate=False*)

Bayesian evidence $\log \mathcal{Z}$.

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

Returns

float

nestcheck.estimators.**get_latex_name**(*func_in*, ***kwargs*)

Produce a latex formatted name for each function for use in labelling results.

Parameters

func_in: function
kwargs: dict, optional
Kwargs for function.

Returns

latex_name: str
Latex formatted name for the function.

nestcheck.estimators.**logz**(*ns_run*, *logw=None*, *simulate=False*)

Natural log of Bayesian evidence $\log \mathcal{Z}$.

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

Returns

float

nestcheck.estimators.**param_cred**(*ns_run*, *logw=None*, *simulate=False*, *probability=0.5*, *param_ind=0*)

One-tailed credible interval on the value of a single parameter (component of theta).

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

probability: float, optional
Quantile to estimate - must be in open interval (0, 1). For example, use 0.5 for the median and 0.84 for the upper 84% quantile. Passed to weighted_quantile.

param_ind: int, optional
Index of parameter for which the credible interval should be calculated. This corresponds to the column of ns_run['theta'] which contains the parameter.

Returns

float

nestcheck.estimators.**param_mean**(*ns_run*, *logw=None*, *simulate=False*, *param_ind=0*, *handle_indexerror=False*)

Mean of a single parameter (single component of theta).

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

param_ind: int, optional
Index of parameter for which the mean should be calculated. This corresponds to the column of ns_run['theta'] which contains the parameter.

handle_indexerror: bool, optional
Make the function function return nan rather than raising an IndexError if param_ind >= ndim. This is useful when applying the same list of estimators to data sets of different dimensions.

Returns

float

nestcheck.estimators.**param_squared_mean**(*ns_run*, *logw=None*, *simulate=False*, *param_ind=0*)

Mean of the square of single parameter (second moment of its posterior distribution).

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

param_ind: int, optional
Index of parameter for which the second moment should be calculated. This corresponds to the column of ns_run['theta'] which contains the parameter.

Returns

float

nestcheck.estimators.**r_cred**(*ns_run*, *logw=None*, *simulate=False*, *probability=0.5*)

One-tailed credible interval on the value of the radial coordinate (magnitude of theta vector).

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

probability: float, optional
Quantile to estimate - must be in open interval (0, 1). For example, use 0.5 for the median and 0.84 for the upper 84% quantile. Passed to weighted_quantile.

Returns

float

nestcheck.estimators.**r_mean**(*ns_run*, *logw=None*, *simulate=False*)

Mean of the radial coordinate (magnitude of theta vector).

Parameters

ns_run: dict
Nested sampling run dict (see the data_processing module docstring for more details).

logw: None or 1d numpy array, optional
Log weights of samples.

simulate: bool, optional
Passed to ns_run_utils.get_logw if logw needs to be calculated.

Returns

float

nestcheck.estimators.**weighted_quantile**(*probability*, *values*, *weights*)

Get quantile estimate for input probability given weighted samples using linear interpolation.

Parameters

probability: float
Quantile to estimate - must be in open interval (0, 1). For example, use 0.5 for the median and 0.84 for the upper 84% quantile.

values: 1d numpy array
Sample values.

weights: 1d numpy array
Corresponding sample weights (same shape as values).

Returns

quantile: float

2.3.7 io_utils

Helper functions for saving, loading and input/output.

nestcheck.io_utils.**pickle_load**(*name*, *extension='.pkl'*)

Load data with pickle.

Parameters

name: str
Path to save to (includes dir, excludes extension).

extension: str, optional
File extension.

Returns

Contents of file path.

nestcheck.io_utils.**pickle_save**(*data*, *name*, ***kwargs*)

Saves object with pickle.

Parameters

data: anything picklable
Object to save.

name: str
Path to save to (includes dir, excludes extension).

extension: str, optional
File extension.

overwrite existing: bool, optional
When the save path already contains file: if True, file will be overwritten, if False the data will be saved with the system time appended to the file name.

nestcheck.io_utils.**save_load_result**(*func*)

Saves and/or loads func output (must be picklable).

nestcheck.io_utils.**timing_decorator**(*func*)

Prints the time func takes to execute.

2.3.8 parallel_utils

Parallel wrapper functions using the concurrent.futures module.

nestcheck.parallel_utils.**parallel_apply**(*func*, *arg_iterable*, ***kwargs*)

Apply function to iterable with parallelisation and a tqdm progress bar.

Roughly equivalent to

```
>>> [func(*func_pre_args, x, *func_args, **func_kwargs) for x in
     arg_iterable]
```

but will **not** necessarily return results in input order.

Parameters

func: function
Function to apply to list of args.

arg_iterable: iterable
argument to iterate over.

func_args: tuple, optional
Additional positional arguments for func.

func_pre_args: tuple, optional
Positional arguments to place before the iterable argument in func.

func_kwargs: dict, optional
Additional keyword arguments for func.

parallel: bool, optional
To turn off parallelisation if needed.

parallel_warning: bool, optional
To turn off warning for no parallelisation if needed.

max_workers: int or None, optional
Number of processes. If max_workers is None then concurrent.futures.ProcessPoolExecutor defaults to using the number of processors of the machine. N.B. If max_workers=None and running on supercomputer clusters with multiple nodes, this may default to the number of processors on a single node.

Returns

results_list: list of function outputs

nestcheck.parallel_utils.**parallel_map**(*func*, **arg_iterable*, ***kwargs*)

Apply function to iterable with parallel map, and hence returns results in order. functools.partial is used to freeze func_pre_args and func_kwargs, meaning that the iterable argument must be the last positional argument.

Roughly equivalent to

```
>>> [func(*func_pre_args, x, **func_kwargs) for x in arg_iterable]
```

Parameters

func: function
Function to apply to list of args.

arg_iterable: iterable
argument to iterate over.

chunksize: int, optional
Perform function in batches

func_pre_args: tuple, optional
Positional arguments to place before the iterable argument in func.

func_kwargs: dict, optional
Additional keyword arguments for func.

parallel: bool, optional
To turn off parallelisation if needed.

parallel_warning: bool, optional
To turn off warning for no parallelisation if needed.

max_workers: int or None, optional
Number of processes. If max_workers is None then concurrent.futures.ProcessPoolExecutor defaults to using the number of processors of the machine. N.B. If max_workers=None and running on supercomputer clusters with multiple nodes, this may default to the number of processors on a single node.

Returns

results_list: list of function outputs

nestcheck.parallel_utils.**select_tqdm**()

If running in a jupyter notebook, then returns tqdm_notebook. Otherwise returns a regular tqdm progress bar.

Returns

progress: function

2.3.9 pandas_functions

Useful transformations and operations on pandas DataFrames.

nestcheck.pandas_functions.**array_ratio_std**(*values_n*, *sigmas_n*, *values_d*, *sigmas_d*)

Gives error on the ratio of 2 floats or 2 1-dimensional arrays given their values and uncertainties. This assumes the covariance = 0, and that the input uncertainties are small compared to the corresponding input values. _n and _d denote the numerator and denominator respectively.

Parameters

values_n: float or numpy array
Numerator values.

sigmas_n: float or numpy array
1σ uncertainties on values_n.

values_d: float or numpy array
Denominator values.

sigmas_d: float or numpy array
1σ uncertainties on values_d.

Returns

std: float or numpy array
1σ uncertainty on values_n / values_d.

nestcheck.pandas_functions.**efficiency_gain_df**(*method_names*, *method_values*, *est_names*, ***kwargs*)

Calculated data frame showing

$$\text{efficiency gain} = \frac{\text{Var[base method]}}{\text{Var[new method]}}$$

See the dynamic nested sampling paper (Higson et al. 2019) for more details.

The standard method on which to base the gain is assumed to be the first method input.

The output DataFrame will contain rows:

mean [dynamic goal]: mean calculation result for standard nested
sampling and dynamic nested sampling with each input dynamic goal.

std [dynamic goal]: standard deviation of results for standard
nested sampling and dynamic nested sampling with each input dynamic goal.

gain [dynamic goal]: the efficiency gain (computational speedup)
from dynamic nested sampling compared to standard nested sampling. This equals (variance of standard results) / (variance of dynamic results); see the dynamic nested sampling paper for more details.

Parameters

method names: list of strs
method values: list
Each element is a list of 1d arrays of results for the method. Each array must have shape (len(est_names),).

est_names: list of strs
Provide column titles for output df.

true_values: iterable of same length as estimators list
True values of the estimators for the given likelihood and prior.

Returns

results: pandas data frame
Results data frame.

nestcheck.pandas_functions.**get_eff_gain**(*base_std*, *base_std_unc*, *meth_std*, *meth_std_unc*, *adjust=1*)

Calculates efficiency gain for a new method compared to a base method. Given the variation in repeated calculations' results using the two methods, the efficiency gain is:

$$\text{efficiency gain} = \frac{\text{Var[base method]}}{\text{Var[new method]}}$$

The uncertainty on the efficiency gain is also calculated.

See the dynamic nested sampling paper (Higson et al. 2019) for more details.

Parameters

base_std: 1d numpy array
base_std_unc: 1d numpy array
Uncertainties on base_std.

meth_std: 1d numpy array
meth_std_unc: 1d numpy array
Uncertainties on base_std.

Returns

gain: 1d numpy array
gain_unc: 1d numpy array
Uncertainties on gain.

nestcheck.pandas_functions.**paper_format_efficiency_gain_df**(*eff_gain_df*)

Transform efficiency gain data frames output by nestcheck into the format shown in the dynamic nested sampling paper (Higson et al. 2019).

Parameters

eff_gain_df: pandas DataFrame
DataFrame of the from produced by efficiency_gain_df.

Returns

paper_df: pandas DataFrame

nestcheck.pandas_functions.**rmse_and_unc**(*values_array*, *true_values*)

Calculate the root meet squared error and its numerical uncertainty.

With a reasonably large number of values in values_list the uncertainty on sq_errors should be approximately normal (from the central limit theorem). Uncertainties are calculated via error propagation: if σ is the error on X then the error on $\sqrt{X}$ is $\frac{\sigma}{2\sqrt{X}}$.

Parameters

values_array: 2d numpy array
Array of results: each row corresponds to a different estimate of the quantities considered.

true_values: 1d numpy array
Correct values for the quantities considered.

Returns

rmse: 1d numpy array
Root-mean-squared-error for each quantity.

rmse_unc: 1d numpy array
Numerical uncertainties on each element of rmse.

nestcheck.pandas_functions.**summary_df**(*df_in*, ***kwargs*)

Make a panda data frame of the mean and std devs of an array of results, including the uncertainties on the values.

This is similar to pandas.DataFrame.describe but also includes estimates of the numerical uncertainties.

The output DataFrame has multiindex levels:

'calculation type': mean and standard deviations of the data. 'result type': value and uncertainty for each quantity.

calculation type result type column_1 column_2 ... mean value mean uncertainty std value std uncertainty

Parameters

df_in: pandas DataFrame
true_values: array
Analytical values if known for comparison with mean. Used to calculate root mean squared errors (RMSE).

include_true_values: bool, optional
Whether or not to include true values in the output DataFrame.

include_rmse: bool, optional
Whether or not to include root-mean-squared-errors in the output DataFrame.

Returns

df: MultiIndex DataFrame

nestcheck.pandas_functions.**summary_df_from_array**(*results_array*, *names*, *axis=0*, ***kwargs*)

Make a panda data frame of the mean and std devs of an array of results, including the uncertainties on the values.

This function converts the array to a DataFrame and calls summary_df on it.

Parameters

results_array: 2d numpy array
names: list of str
Names for the output df's columns.

axis: int, optional
Axis on which to calculate summary statistics.

Returns

df: MultiIndex DataFrame
See summary_df docstring for more details.

nestcheck.pandas_functions.**summary_df_from_list**(*results_list*, *names*, ***kwargs*)

Make a panda data frame of the mean and std devs of each element of a list of 1d arrays, including the uncertainties on the values.

This just converts the array to a DataFrame and calls summary_df on it.

Parameters

results_list: list of 1d numpy arrays
Must have same length as names.

names: list of strs
Names for the output df's columns.

kwargs: dict, optional
Keyword arguments to pass to summary_df.

Returns

df: MultiIndex DataFrame
See summary_df docstring for more details.

nestcheck.pandas_functions.**summary_df_from_multi**(*multi_in*, *inds_to_keep=None*, ***kwargs*)

Apply summary_df to a multiindex while preserving some levels.

Parameters

multi_in: multiindex pandas DataFrame
inds_to_keep: None or list of strs, optional
Index levels to preserve.

kwargs: dict, optional
Keyword arguments to pass to summary_df.

Returns

df: MultiIndex DataFrame
See summary_df docstring for more details.

2.3.10 dummy_data

Create dummy nested sampling run data for testing.

nestcheck.dummy_data.**get_dummy_dynamic_run**(*nsamples*, ***kwargs*)

Generate dummy data for a dynamic nested sampling run.

Loglikelihood values of points are generated from a uniform distribution in (0, 1), sorted, scaled by logl_range and shifted by logl_start (if it is not -np.inf). Theta values of each point are each generated from a uniform distribution in (0, 1).

Parameters

nsamples: int
Number of samples in thread.

nthread_init: int
Number of threads in the inital run (starting at logl=-np.inf).

nthread_dyn: int
Number of threads in the inital run (starting at randomly chosen points in the initial run).

ndim: int, optional
Number of dimensions.

seed: int, optional
If not False, the seed is set with np.random.seed(seed).

logl_start: float, optional
logl at which thread starts.

logl_range: float, optional
Scale factor applied to logl values.

nestcheck.dummy_data.**get_dummy_run**(*nthread*, *nsamples*, ***kwargs*)

Generate dummy data for a nested sampling run.

Log-likelihood values of points are generated from a uniform distribution in (0, 1), sorted, scaled by logl_range and shifted by logl_start (if it is not -np.inf). Theta values of each point are each generated from a uniform distribution in (0, 1).

Parameters

nthreads: int
Number of threads in the run.

nsamples: int
Number of samples in thread.

ndim: int, optional
Number of dimensions.

seed: int, optional
If not False, the seed is set with np.random.seed(seed).

logl_start: float, optional
logl at which thread starts.

logl_range: float, optional
Scale factor applied to logl values.

nestcheck.dummy_data.**get_dummy_thread**(*nsamples*, ***kwargs*)

Generate dummy data for a single nested sampling thread.

Log-likelihood values of points are generated from a uniform distribution in (0, 1), sorted, scaled by logl_range and shifted by logl_start (if it is not -np.inf). Theta values of each point are each generated from a uniform distribution in (0, 1).

Parameters

nsamples: int
Number of samples in thread.

ndim: int, optional
Number of dimensions.

seed: int, optional
If not False, the seed is set with np.random.seed(seed).

logl_start: float, optional
logl at which thread starts.

logl_range: float, optional
Scale factor applied to logl values.

2.3.11 write_polychord_output

Functions for writing PolyChord-format output files given a nested sampling run dictionary stored in the nestcheck format.

nestcheck.write_polychord_output.**run_dead_birth_array**(*run*, *logl_init=-1e+30*, ***kwargs*)

Converts input run into an array of the format of a PolyChord <root>_dead-birth.txt file. Note that this in fact includes live points remaining at termination as well as dead points.

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

logl_init: float, optional
Value used to identify the inital live points which were sampled from the whole prior. Default value is set to -1e30 as in PolyChord.

kwargs: dict, optional
Options for check_ns_run.

Returns

samples: 2d numpy array
Array of dead points and any remaining live points at termination. Has #parameters + 2 columns: param_1, param_2, ... , logl, birth_logl

nestcheck.write_polychord_output.**write_run_output**(*run*, ***kwargs*)

Writes PolyChord output files corresponding to the input nested sampling run. The file root is

```
root = os.path.join(run['output']['base_dir'],
                    run['output']['file_root'])
```

Output files which can be made with this function (see the PolyChord documentation for more information about what each contains):

- [root].stats
- [root].txt
- [root]_equal_weights.txt
- [root]_dead-birth.txt
- [root]_dead.txt

Files produced by PolyChord which are not made by this function:

- [root].resume: for resuming runs part way through (not relevant for a completed run).
- [root]_phys_live.txt and [root]phys_live-birth.txt: for checking runtime progress (not relevant for a completed run).
- [root].paramnames: for use with getdist (not needed when calling getdist from within python).

Parameters

ns_run: dict
Nested sampling run dict (see data_processing module docstring for more details).

write_dead: bool, optional
Whether or not to write [root]_dead.txt and [root]_dead-birth.txt.

write_stats: bool, optional
Whether or not to write [root].stats.

posteriors: bool, optional
Whether or not to write [root].txt.

equals: bool, optional
Whether or not to write [root]_equal_weights.txt.

stats_means_errs: bool, optional
Whether or not to calculate mean values of $\log \mathcal{Z}$ and each parameter, and their uncertainties.

fmt: str, optional
Formatting for numbers written by np.savetxt. Default value is set to make output files look like the ones produced by PolyChord.

n_simulate: int, optional
Number of bootstrap replications to use when estimating uncertainty on evidence and parameter means.

logl_init: float, optional
Value used to identify the inital live points which were sampled from the whole prior. Default value is set to -1e30 as in PolyChord.

nestcheck.write_polychord_output.**write_stats_file**(*run_output_dict*)

Writes a dummy PolyChord format .stats file for tests functions for processing stats files. This is written to:

base_dir/file_root.stats

Also returns the data in the file as a dict for comparison.

Parameters

run_output_dict: dict
Output information to write to .stats file. Must contain file_root and base_dir. If other settings are not specified, default values are used.

Returns

output: dict
The expected output of nestcheck.process_polychord_stats(file_root, base_dir)

CHAPTER

THREE

ATTRIBUTION

If nestcheck is useful for your academic research, please cite the three papers introducing the software and the methods it implements. The BibTeX is:

```
@article{higson2019diagnostic,
title={nestcheck: diagnostic tests for nested sampling calculations},
author={Higson, Edward and Handley, Will and Hobson, Mike and Lasenby, Anthony},
journal={Monthly Notices of the Royal Astronomical Society},
year={2019}
volume={483},
number={2},
pages={2044--2056},
doi={10.1093/mnras/sty3090},
url={http://doi.org/10.1093/mnras/sty3090},
archivePrefix={arXiv},
arxivId={1804.06406}}

@article{higson2018sampling,
title={Sampling Errors in Nested Sampling Parameter Estimation},
author={Higson, Edward and Handley, Will and Hobson, Mike and Lasenby, Anthony},
year={2018}
journal={Bayesian Analysis},
number={3},
volume={13},
pages={873--896},
doi={10.1214/17-BA1075},
url={https://doi.org/10.1214/17-BA1075}}

@article{higson2018nestcheck,
title={nestcheck: error analysis, diagnostic tests and plots for nested sampling␣
↪calculations},
author={Higson, Edward},
year={2018},
journal={Journal of Open Source Software},
number={29},
pages={916},
volume={3},
doi={10.21105/joss.00916},
url={http://joss.theoj.org/papers/10.21105/joss.00916}}
```

CHAPTER

FOUR

CHANGELOG

The changelog for each release can be found at https://github.com/ejhigson/nestcheck/releases.

CHAPTER

FIVE

CONTRIBUTIONS

Contributions are welcome! Development takes place on github:

- source code: https://github.com/ejhigson/nestcheck;
- issue tracker: https://github.com/ejhigson/nestcheck/issues.

When creating a pull request, please try to make sure the tests pass and use numpy-style docstrings.

If you have any questions or suggestions please get in touch (e.higson@mrao.cam.ac.uk).

CHAPTER

SIX

AUTHORS & LICENSE

PYTHON MODULE INDEX

n

INDEX

N

P

R

S

T

W