perfectns Documentation

Edward Higson

Aug 26, 2019

CONTENTS

Nested sampling (Skilling, 2006) 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. Dynamic nested sampling (Higson et al., 2019a) is a generalisation of the nested sampling algorithm which can provide order-of-magnitude increases in computational efficiency.

`perfectns` performs dynamic nested sampling and standard nested sampling for spherically symmetric likelihoods and priors, and analyses the samples produced. The spherical symmetry allows the nested sampling algorithm to be followed "perfectly" - i.e. without additional errors due to correlations between samples, which are present in other nested sampling software.

The specialised methods used by `perfectns` make it highly effective and reliable for spherically symmetric calculations. The software is also intended for use in research into the statistical properties of nested sampling, and to provide a benchmark for testing the performance of nested sampling software packages used for practical problems - which rely on numerical techniques to produce approximately uncorrelated samples. For details of the theory behind the software and of how it performs perfect nested sampling, see the see the theory section of the documentation.

To get started, see the installation instructions and the demo. For more examples of `perfectns`'s use, see the code used to make the results and figures in the dynamic nested sampling paper (https://github.com/ejhigson/dns).

CHAPTER

ONE

DOCUMENTATION CONTENTS

1.1 Theory

Nested sampling is a method for Bayesian computation which given some likelihood $\mathcal{L}(\boldsymbol{\theta})$ and prior $\pi(\boldsymbol{\theta})$ will generate posterior samples and an estimate of the Bayesian evidence $\mathcal{Z}$. The algorithm works by sampling some number of points randomly from the prior then iteratively replacing the point with the lowest likelihood with another point sampled from the region of the prior with a higher likelihood.

Generating uncorrelated samples within the likelihood constraint is computationally challenging and can only be done approximately by software such as `MultiNest` and `PolyChord`. This can lead to additional errors; for example due to correlated samples or missing a mode in a multimodal posterior (see Higson et al., (2019), for a detailed discussion). `perfectns` uses specially symmetric cases where uncorrelated samples can be easily drawn from within some iso-likelihood contour to perform nested sampling perfectly in the manner described by Keeton (2011).

1.1.1 Likelihoods and Priors

`perfectns` uses only spherical likelihoods and priors $\mathcal{L}(r)$ and $\pi(r)$, where the radial coordinate $r = |\boldsymbol{\theta}|$. Any perfect nested sampling evidence or parameter estimation calculation is equivalent to some problem with spherically symmetric likelihoods and priors (see Section 3 of Higson et al., 2018, for more details), so with suitable choices of $\mathcal{L}(r)$, $\pi(r)$ and the parameter estimation quantity a wide variety of tests can be performed with this package. $\mathcal{L}(r)$ must be a monotone decreasing function of the radius r, so that the likelihood increases as r decreases to a maximum at $r = 0$.

Nested statistically estimating the shrinkage in the fraction of the prior volume remaining X. This package generates nested sampling runs by sampling from the known distribution of prior volumes X, then using the prior to find the corresponding radial coordinates $r(X)$ and the likelihood to find the likelihood values $\mathcal{L}(r(X))$. Parameter vectors $\boldsymbol{\theta}_i$ for each point i are sampled from the hyper-spherical shell with radius r_i.

New likelihoods and priors can be added to `likelihoods.py` and `priors.py`, and new parameter estimation quantities can be added to `estimators.py`. See the documentation in each file for more details.

1.2 Installation

`perfectns` is compatible with python >=3.5; for a list of its dependencies see the `setup.py` file. Installation can be carried out with pip:

```
pip install perfectns
```

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

```
git clone https://github.com/ejhigson/perfectns.git
cd perfectns
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 `perfectns`'s dependencies which are not already satisfied by your system.

1.2.1 Tests

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

```
nosetests
```

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

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

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

1.3 Demo

This notebook demonstrates the basic functionality of the `perfectns` package; for background see the dynamic nested sampling paper (Higson at al., 2019a).

1.3.1 Running nested sampling calculations

The likelihood $\mathcal{L}(\theta)$, prior $\pi(\theta)$ and calculation settings are specified in a PerfectNSSettings object. For this example we will use a 10-dimensional spherically symmetric Gaussian likelihood with size $\sigma_{\mathcal{L}} = 1$ and a Gaussian prior with size $\sigma_{\pi} = 10$.

[1]:
```
import perfectns.settings
import perfectns.likelihoods as likelihoods
import perfectns.priors as priors

# Input settings
settings = perfectns.settings.PerfectNSSettings()
settings.likelihood = likelihoods.Gaussian(likelihood_scale=1)
settings.prior = priors.Gaussian(prior_scale=10)
settings.n_dim = 10
settings.ninit = 10
settings.nlive_const = 100
```

The "dynamic_goal" setting determines if dynamic nested sampling should be used and, if so, how to split the computational effort between increasing parameter estimation accuracy and evidence calculation accuracy. dynamic_goal=1 optimises purely for parameter estimation and dynamic_goal=0 optimises purely for calculating the Bayesian evidence $\mathcal{Z}$.

Lets try running standard nested sampling and dynamic nested sampling calculation:

[2]:
```
import perfectns.nested_sampling as nested_sampling

# Perform standard nested sampling
settings.dynamic_goal = None
standard_ns_run = nested_sampling.generate_ns_run(settings, random_seed=0)  # set␣
↪random_seed for reproducible results
# Perform dynamic nested sampling
settings.dynamic_goal = 1  # optimise for parameter estimation accuracy
dynamic_ns_run = nested_sampling.generate_ns_run(settings, random_seed=0)  # set␣
↪random_seed for reproducible results
```

We can now make posterior inferences using the samples generated by the nested sampling calculations using the utility functions from `nestcheck`. Here we calculate:

1. the log Bayesian evidence $\log \mathcal{Z} = \log\left(\int \mathcal{L}(\theta)\pi(\theta)\mathrm{d}\theta\right)$,
2. the mean of the first parameter θ_1,
3. the second moment of the posterior distribution of θ_1,
4. the median of θ_1,
5. the 84% one-tailed credible interval on θ_1.

For the Gaussian likelihood and prior we can calculate the posterior distribution analytically, so we first calculate the analytic values of each quantity for comparison. The results are displayed in a `pandas` DataFrame.

[3]:
```
import perfectns.estimators as e
import nestcheck.ns_run_utils
import pandas as pd

estimator_list = [e.LogZ(),
                  e.ParamMean(),
                  e.ParamSquaredMean(),
                  e.ParamCred(0.5),
                  e.ParamCred(0.84)]
estimator_names = [est.latex_name for est in estimator_list]
results = pd.DataFrame([nestcheck.ns_run_utils.run_estimators(standard_ns_run,␣
↪estimator_list),
                        nestcheck.ns_run_utils.run_estimators(dynamic_ns_run,␣
↪estimator_list)],
                       columns=estimator_names, index=['standard run', 'dynamic run'])
# Add true values for comparison
results.loc['true values'] = e.get_true_estimator_values(estimator_list, settings)
results
```

[3]:
```
              $\mathrm{log} \mathcal{Z}$  $\overline{\theta_{\hat{1}}}$  \
standard run                  -32.289475                      -0.029820
dynamic run                   -33.053897                      -0.054664
true values                   -32.264988                       0.000000

              $\overline{\theta^2_{\hat{1}}}$  \
standard run                         0.971677
dynamic run                          0.931194
true values                          0.990080

              $\mathrm{median}(\theta_{\hat{1}})$  \
standard run                            -0.046113
dynamic run                             -0.093203
true values                              0.000000
```

(continues on next page)

(continued from previous page)

```
              $\mathrm{C.I.}_{84\%}(\theta_{\hat{1}})$
standard run                                  0.945738
dynamic run                                   0.913430
true values                                   0.989523
```

1.3.2 Estimating sampling errors

You can estimate the numerical uncertainties on these results by calculating the standard deviation of the sampling errors distributions each run using the bootstrap resampling approach described in Higson et al. (2018) (implemented in `nestcheck`).

[5]:
```
import numpy as np
import nestcheck.error_analysis
np.random.seed(0)
results.loc['standard unc'] = nestcheck.error_analysis.run_std_bootstrap(
    standard_ns_run, estimator_list, n_simulate=200)
results.loc['dynamic unc'] = nestcheck.error_analysis.run_std_bootstrap(
    dynamic_ns_run, estimator_list, n_simulate=200)
results.loc[['standard unc', 'dynamic unc']]
```

[5]:
```
              $\mathrm{log} \mathcal{Z}$  $\overline{\theta_{\hat{1}}}$  \
standard unc                    0.366472                       0.031321
dynamic unc                     1.204710                       0.017033

              $\overline{\theta^2_{\hat{1}}}$  \
standard unc                         0.049564
dynamic unc                          0.030143

              $\mathrm{median}(\theta_{\hat{1}})$  \
standard unc                               0.043201
dynamic unc                                0.020610

              $\mathrm{C.I.}_{84\%}(\theta_{\hat{1}})$
standard unc                                  0.050531
dynamic unc                                   0.027900
```

This approach works for both dynamic and standard nested sampling. In addition as `perfectns` can perform the nested sampling algorithm "perfectly" there are no additional errors from implementation-specific effects such as correlated samples (see Higson et al., 2019b for a detailed discussion).

1.3.3 Generating and analysing runs in parallel

Multiple nested sampling runs can be generated and analysed in parallel (using `parallel_utils` from `nestcheck`).

[6]:
```
import numpy as np
import nestcheck.parallel_utils as pu
import nestcheck.pandas_functions as pf

# Generate 100 nested sampling runs
run_list = nested_sampling.get_run_data(settings, 100, save=False, load=False, random_
↪seeds=list(range(100)))
# Calculate posterior inferences for each run
```

(continues on next page)

(continued from previous page)

```
values = pu.parallel_apply(nestcheck.ns_run_utils.run_estimators, run_list,
                           func_args=(estimator_list,))
# Show the mean and standard deviation of the calculation results
multi_run_tests = pf.summary_df_from_list(values, estimator_names)
multi_run_tests
```

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

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

[6]:

		$\mathrm{log} \mathcal{Z}$	$\overline{\theta_{\hat{1}}}$	$\overline{\theta^2_{\hat{1}}}$	$\mathrm{median}(\theta_{\hat{1}})$	$\mathrm{C.I.}_{84\%}(\theta_{\hat{1}})$
calculation type	result type					
mean	value	-32.139762	-0.000028	0.992171	-0.001141	0.992247
	uncertainty	0.114168	0.001896	0.002937	0.002478	0.002826
std	value	1.141684	0.018956	0.029367	0.024777	0.028265
	uncertainty	0.081136	0.001347	0.002087	0.001761	0.002009

1.3.4 Comparing dynamic and standard nested sampling performance

Lets now compare the performance of dynamic and standard nested sampling, using the 10-dimensional Gaussian likelihood and prior.

[7]:
```
import perfectns.results_tables as rt

# Input settings
```

(continues on next page)

(continued from previous page)

```
settings = perfectns.settings.PerfectNSSettings()
settings.likelihood = likelihoods.Gaussian(likelihood_scale=1)
settings.prior = priors.Gaussian(prior_scale=10)
settings.ninit = 10
settings.nlive_const = 100
settings.n_dim = 10
# Run results settings
dynamic_results_table = rt.get_dynamic_results(100, [0, 1], estimator_list, settings,␣
↪save=False, load=False)
dynamic_results_table[estimator_names]
```

```
dynamic_goal=None n_samples_max=None
```

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

```

```

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

```
dynamic_goal=0 n_samples_max=2992
```

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

```

```

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

```
dynamic_goal=1 n_samples_max=3007
```

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

```

```

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

```

```

[7]:

			$\mathrm{log} \mathcal{Z}$	\
calculation type	dynamic settings	result type		
mean	standard	value	-32.259097	
		uncertainty	0.040170	
	dynamic $G=0$	value	-32.218098	
		uncertainty	0.034715	
	dynamic $G=1$	value	-32.143640	
		uncertainty	0.113207	
std	standard	value	0.401704	
		uncertainty	0.028548	
	dynamic $G=0$	value	0.347150	
		uncertainty	0.024671	
	dynamic $G=1$	value	1.132066	
		uncertainty	0.080452	
std efficiency gain	dynamic $G=0$	value	1.338991	
		uncertainty	0.269147	
	dynamic $G=1$	value	0.125912	
		uncertainty	0.025309	

			$\overline{\theta_{\hat{1}}}$	\
calculation type	dynamic settings	result type		
mean	standard	value	-0.004415	
		uncertainty	0.003470	
	dynamic $G=0$	value	-0.002052	
		uncertainty	0.004263	
	dynamic $G=1$	value	-0.000484	

(continues on next page)

(continued from previous page)

```
                                    uncertainty                      0.001923
std                 standard        value                            0.034697
                                    uncertainty                      0.002466
                    dynamic $G=0$   value                            0.042629
                                    uncertainty                      0.003029
                    dynamic $G=1$   value                            0.019231
                                    uncertainty                      0.001367
std efficiency gain dynamic $G=0$   value                            0.662480
                                    uncertainty                      0.133164
                    dynamic $G=1$   value                            3.255211
                                    uncertainty                      0.654322

                                                  $\overline{\theta^2_{\hat{1}}}$  \
calculation type    dynamic settings result type
mean                standard        value                              0.984532
                                    uncertainty                        0.004505
                    dynamic $G=0$   value                              0.979013
                                    uncertainty                        0.006417
                    dynamic $G=1$   value                              0.992710
                                    uncertainty                        0.002890
std                 standard        value                              0.045052
                                    uncertainty                        0.003202
                    dynamic $G=0$   value                              0.064168
                                    uncertainty                        0.004560
                    dynamic $G=1$   value                              0.028904
                                    uncertainty                        0.002054
std efficiency gain dynamic $G=0$   value                              0.492935
                                    uncertainty                        0.099084
                    dynamic $G=1$   value                              2.429505
                                    uncertainty                        0.488349

                                                  $\mathrm{median}(\theta_{\hat{1}})$␣
↪ \
calculation type    dynamic settings result type
mean                standard        value                              0.001289
                                    uncertainty                        0.004431
                    dynamic $G=0$   value                             -0.001307
                                    uncertainty                        0.005667
                    dynamic $G=1$   value                             -0.001467
                                    uncertainty                        0.002494
std                 standard        value                              0.044311
                                    uncertainty                        0.003149
                    dynamic $G=0$   value                              0.056671
                                    uncertainty                        0.004027
                    dynamic $G=1$   value                              0.024941
                                    uncertainty                        0.001772
std efficiency gain dynamic $G=0$   value                              0.611363
                                    uncertainty                        0.122889
                    dynamic $G=1$   value                              3.156470
                                    uncertainty                        0.634474

                                                  $\mathrm{C.I.}_{84\%}(\theta_{\hat
↪{1}})$
calculation type    dynamic settings result type
mean                standard        value                                    0.
↪981203
                                    uncertainty                              0.
↪005580
```

(continues on next page)

(continued from previous page)

```
                    dynamic $G=0$    value                                    0.
↪978076
                                     uncertainty                              0.
↪007123
                    dynamic $G=1$    value                                    0.
↪992193
                                     uncertainty                              0.
↪002868
std                 standard         value                                    0.
↪055795
                                     uncertainty                              0.
↪003965
                    dynamic $G=0$    value                                    0.
↪071234
                                     uncertainty                              0.
↪005062
                    dynamic $G=1$    value                                    0.
↪028678
                                     uncertainty                              0.
↪002038
std efficiency gain dynamic $G=0$    value                                    0.
↪613510
                                     uncertainty                              0.
↪123320
                    dynamic $G=1$    value                                    3.
↪785308
                                     uncertainty                              0.
↪760876
```

Looking at the `std efficiency gain` rows, you should see that dynamic nested sampling targeted at parameter estimation (dynamic goal=1) has an efficiency gain (equivalent computational speedup) for parameter estimation (columns other than $\log \mathcal{Z}$) of factor of around 3 compared to standard nested sampling.

Similar results tables for different likelihoods can be found in the dynamic nested sampling paper (Higson at al., 2019a). For more information about the get_dynamic_results function look at its documentation.

1.3.5 Comparing bootstrap error estimates to observed distributions of results

We can check if the bootstrap estimates of parameter estimation sampling errors are accurate, using a 3d Gaussian likelihood and Gaussian prior.

[8]:
```
settings.likelihood = likelihoods.Gaussian(likelihood_scale=1)
settings.prior = priors.Gaussian(prior_scale=10)
settings.n_dim = 3
bootstrap_results_table = rt.get_bootstrap_results(50, 50, # 100, 200,
                                                   estimator_list, settings,
                                                   n_run_ci=20,
                                                   n_simulate_ci=200,  # n_simulate_
↪ci=1000,
                                                   add_sim_method=False,
                                                   cred_int=0.95,
                                                   ninit_sep=True,
                                                   parallel=True)
bootstrap_results_table
```

```
HBox(children=(IntProgress(value=0, max=50), HTML(value='')))

HBox(children=(IntProgress(value=0, max=50), HTML(value='')))

HBox(children=(IntProgress(value=0, max=50), HTML(value='')))

HBox(children=(IntProgress(value=0, max=20), HTML(value='')))
```

[8]:

		$\mathrm{log} \mathcal{Z}$	$\overline{\theta_{\hat{1}}}$	$\overline{\theta^2_{\hat{1}}}$
calculation type	result type			
repeats mean	value	-9.625403	0.002816	0.987749
	uncertainty	0.036342	0.007468	0.010398
repeats std	value	0.256977	0.052807	0.073526
	uncertainty	0.025959	0.005334	0.007427
bs std / repeats std	value	0.892930	0.873771	0.925045
	uncertainty	0.091285	0.089760	0.095384
bs estimate % variation	value	11.112636	13.206385	14.631458
	uncertainty	1.122546	1.334046	1.478000
bs 0.95 CI	value	-9.220209	0.070315	1.114863
	uncertainty	0.044147	0.012680	0.018326
bs +-1std % coverage	value	72.000000	62.000000	52.000000
bs 0.95 CI % coverage	value	94.000000	90.000000	96.000000

(continues on next page)

(continued from previous page)

```
                                    $\mathrm{median}(\theta_{\hat{1}})$  \
calculation type        result type
repeats mean            value                                  0.007308
                        uncertainty                            0.009250
repeats std             value                                  0.065408
                        uncertainty                            0.006607
bs std / repeats std    value                                  0.972450
                        uncertainty                            0.100413
bs estimate % variation value                                 15.134609
                        uncertainty                            1.528826
bs 0.95 CI              value                                  0.097298
                        uncertainty                            0.016915
bs +-1std % coverage    value                                 62.000000
bs 0.95 CI % coverage   value                                 92.000000

                                    $\mathrm{C.I.}_{84\%}(\theta_{\hat{1}})$
calculation type        result type
repeats mean            value                                  0.999308
                        uncertainty                            0.012202
repeats std             value                                  0.086281
                        uncertainty                            0.008716
bs std / repeats std    value                                  0.920943
                        uncertainty                            0.096081
bs estimate % variation value                                 18.444304
                        uncertainty                            1.863156
bs 0.95 CI              value                                  1.127381
                        uncertainty                            0.018304
bs +-1std % coverage    value                                 64.000000
bs 0.95 CI % coverage   value                                 92.000000
```

Note that every second column gives an estimated numerical uncertainty on the values in the previous column.

You should see that the ratio of the bootstrap error estimates to bootstrap_results the standard deviation of results (row 4 of bootstrap_results_table) has values close to 1 given the estimated numerical uncertainties. Similar results are given in the appendix of the dynamic nested sampling paper (Higson, 2019a); see the paper and the get_bootstrap_results function's documentation for more details.

1.4 Detailed API documentation

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

1.4.1 settings

Defines base class which holds settings for performing perfect nested sampling. This includes likelihood and prior objects as well as the parameters controlling how the calculation is performed - for example the number of live points and whether or not dynamic nested sampling is to be used.

class perfectns.settings.**PerfectNSSettings**(***kwargs*)

Controls how Perfect Nested Sampling is performed.

For more details of the standard nested sampling and dynamic nested sampling algorithms see:

1) 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al. 2019). 2) 'Sampling errors in nested sampling parameter estimation' (Higson et al. 2018).

Parameters

n_dim: int Dimension of the likelihood and prior.

prior: object A prior object. See priors.py for more details.

likelihood: object A likelihood object. See likelihoods.py for more details.

nlive_const: int The number of live points for standard nested sampling.

termination_fraction: float Standard nested sampling runs will terminate when the posterior mass remaining (estimated from the live points) is less than termination_fraction times the posterior mass (evidence) in the dead points.

dynamic_goal: float or None Determines is dynamic nested sampling is used, and if so how improvements in evidence and parameter estimation accuracy are weighted.

n_samples_max: int or None Number of samples after which dynamic nested sampling will terminate. If this is None the number of samples to use is chosen using nlive_const.

ninit: int Number of live points used in dynamic nested sampling for the initial exploratory standard nested sampling run.

nbatch: int Number of threads dynamic nested sampling adds before each recalculation of points' importances to the calculation.

dynamic_fraction: float Dynamic nested sampling adds more samples wherever points' importances are greater than dynamic_fraction times the largest importance.

tuned_dynamic_p: bool Determines if dynamic nested sampling is tuned for a specific parameter estimation problem.

Methods

get_settings_dict(self)	Returns a dictionary containing settings information.
logl_given_logx(self, logx)	Maps input logx values to loglikelihoods.
logl_given_r(self, r)	Maps input radial coordinates to loglikelihood values.
logx_given_logl(self, logl)	Maps input loglikelihoods to logx values.
logx_given_r(self, r)	Maps input radial coordinates to logx values.
logz_analytic(self)	If available gives an analytically calculated value for the log evidence for the likelihood and prior (useful for checking results).
r_given_logl(self, logl)	Maps input loglikelihood values to radial coordinates.
r_given_logx(self, logx)	Maps input logx values to radial coordinates.
save_name(self[, include_dg, ...])	Make a standard save name format for a given settings configoration.

get_settings_dict(*self*)
Returns a dictionary containing settings information. The names and parameters of the likelihoods and priors are stored instead of the objects themselves so they can be saved with pickle.

Returns

settings_dict: dict

logl_given_logx(*self*, *logx*)
Maps input logx values to loglikelihoods.

Parameters

logx: float or numpy array

n_dim: int

Returns

logl: same type and size as logx

logl_given_r (*self*, *r*)
Maps input radial coordinates to loglikelihood values.

Parameters

r: float or numpy array

n_dim: int

Returns

logl: same type and size as r

logx_given_logl (*self*, *logl*)
Maps input loglikelihoods to logx values.

Parameters

logl: float or numpy array

n_dim: int

Returns

logx: same type and size as logl

logx_given_r (*self*, *r*)
Maps input radial coordinates to logx values.

Parameters

r: float or numpy array

n_dim: int

Returns

logx: same type and size as r

logz_analytic (*self*)
If available gives an analytically calculated value for the log evidence for the likelihood and prior (useful for checking results).

This functionality is stored in the likelihood object. If it has not been set up then an error message is printed and nothing is returned.

Returns

float or None True value of logz or None if it is not available.

r_given_logl (*self*, *logl*)
Maps input loglikelihood values to radial coordinates.

Parameters

r: float or numpy array

n_dim: int

Returns

logl: same type and size as r

r_given_logx (*self*, *logx*)
Maps input logx values to radial coordinates.

Parameters

logx: float or numpy array

n_dim: int

Returns

r: same type and size as logx

save_name (*self*, *include_dg=True*, *include_samples_max=False*)
Make a standard save name format for a given settings configoration.

Parameters

include_dg: bool, optional Whether or not to include the dynamic_goal setting in save_name.

include_samples_max: bool, optional Whether or not to include the n_samples_max setting in save_name.

Returns

save_name: str

1.4.2 nested_sampling

Functions which perform standard and dynamic nested sampling runs and generate samples for use in evidence calculations and parameter estimation.

Nested sampling runs are stored a format compatible with the nestcheck package.

perfectns.nested_sampling.**dict_given_samples_array** (*samples*, *thread_min_max*)
Converts an array of information about samples back into a dictionary.

Parameters

samples: numpy array Numpy array containing columns [logl, r, logx, 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 dictionary corresponding to the samples array. Contains keys: 'logl', 'r', 'logx', 'thread_label', 'nlive_array', 'theta' N.B. this does not contain a record of the run's settings.

perfectns.nested_sampling.**generate_dynamic_run** (*settings*)
Generate a dynamic nested sampling run. For details of the dynamic nested sampling algorithm, see 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019).

The run terminates when the number of samples reaches some limit settings.n_samples_max. If this is not set, the function will estimate the number of samples that a standard nested sampling run with settings.nlive_const would use from the number of samples in the initial exploratory run.

Parameters

settings: PerfectNSSettings object settings.dynamic_goal controls whether the algorithm aims to increase parameter estimation accuracy (dynamic_goal=1), evidence accuracy (dynamic_goal=0) or places some weight on both.

Returns

dict Nested sampling run dictionary containing information about the run's posterior samples and a record of the settings used. See docstring for generate_ns_run for more details.

perfectns.nested_sampling.**generate_ns_run**(*settings*, *random_seed=None*)

Performs perfect nested sampling calculation and returns a nested sampling run in the form of a dictionary.

This function is just a wrapper around the generate_standard_run (performs standard nested sampling) and generate_dynamic_run (performs dynamic nested sampling) which are chosen depending on the input settings.

Parameters

settings: PerfectNSSettings object

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

dict Nested sampling run dictionary containing information about the run's posterior samples and a record of the settings used. These are as separate arrays giving values for points in order of increasing likelihood. Keys:

'settings': dict recording settings used. 'logl': 1d array of log likelihoods. 'r': 1d array of radial coordinates. 'logx': 1d array of logx coordinates. 'theta': 2d array, each row is sample coordinate. The number of

co-ordinates saved is controlled by settings.dims_to_sample.

'nlive_array': 1d array of the local number of live points at each sample.

'thread_min_max': 2d array containing likelihoods at which each nested sampling thread begins and ends.

'thread_labels': 1d array listing the threads each sample belongs to.

perfectns.nested_sampling.**generate_single_thread**(*settings*, *logx_end*, *thread_label*, *logx_start=0*, *keep_final_point=True*)

Generates a samples array for a thread (single live point run) between logx_start and logx_end. Settings argument specifies how the calculation is done.

Parameters

settings: PerfectNSSettings object

logx_end: float

thread_label: int Index labelling the thread.

logx_start: float, optional

keep_final_point: bool, optional See generate_thread_logx docstring.

perfectns.nested_sampling.**generate_standard_run**(*settings*, *is_dynamic_initial_run=False*)

Performs standard nested sampling using the likelihood and prior specified in settings.

The run terminates when the estimated posterior mass contained in the live points is less than settings.termination_fraction. The evidence in the remaining live points is estimated as

Z_{live} = average likelihood of live points * prior volume remaining

Parameters

settings: PerfectNSSettings object

is_dynamic_initial_run: bool, optional Set to True if this is the initial exploratory run in dynamic nested sampling.

Returns

run: dict Nested sampling run dictionary containing information about the run's posterior samples and a record of the settings used. See docstring for generate_ns_run for more details.

perfectns.nested_sampling.**generate_thread_logx**(*logx_end*, *logx_start=0*, *keep_final_point=True*)

Generate logx co-ordinates of a new nested sampling thread (single live point run).

Parameters

logx_end: float Logx value at which run terminates.

logx_start: float, optional. Logx value at which run starts. 0 corresponds to sampling from the whole prior.

keep_final_point: bool, optional If False, the final point with logx less than logx_end is removed.

Returns

logx_list: list of floats

perfectns.nested_sampling.**get_run_data**(*settings*, *n_repeat*, ***kwargs*)

Tests if runs with the specified settings are already cached. If not the runs are generated and saved.

Parameters

settings: PerfectNSSettings object

n_repeat: int Number of nested sampling runs to generate.

parallel: bool, optional Should runs be generated in parallel?

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 and therefore there will be no speedup from multiple nodes (must specify manually in this case).

load: bool, optional Should previously saved runs be loaded? If False, new runs are generated.

save: bool, optional Should any new runs generated be saved?

cache_dir: str, optional Directory for caching

overwrite_existing: bool, optional if a file exists already but we generate new run data, should we overwrite the existing file when saved?

check_loaded_settings: bool, optional if we load a cached file, should we check if the loaded file's settings match the current settings (and generate fresh runs if they do not)?

random_seeds: list, optional random_seed arguments for each call of generate_ns_run.

Returns

run_list list of n_repeat nested sampling runs.

perfectns.nested_sampling.**min_max_importance**(*importance*, *samples*, *settings*)

Find the logl and logx values at which to start and end additional dynamic nested sampling threads.

Parameters

importance: 1d numpy array Relative importances of samples.

samples: 2d numpy array See dict_given_samples_arrry docstring for details of columns.

settings: PerfectNSSettings object

Returns

list of two floats Contains logl_min and logl_max defining the start and end of the region from which new points should be sampled.

list of two floats Logx values corresponding to logl_min and logl_max.

perfectns.nested_sampling.**p_importance**(*theta*, *w_relative*, *tuned_dynamic_p=False*, *tuning_type='theta1'*)

Calculate the relative importance of each point for parameter estimation.

For more details see 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019).

Parameters

theta: 2d numpy array Each row gives parameter values of samples.

w_relative: 1d numpy array Relative point weights.

tuned_dynamic_p: bool, optional Whether or not to tune for a specific parameter estimation problem. See the dynamic nested sampling paper for more details.

tuning_type: str, optional Which parameter estimation problem to tune for. Only used if tuned_dynamic_p is True. So far only set up to tune for the mean of the first parameter.

Returns

importance: 1d numpy array Relative point importances. Normalised so the biggest value in the array is equal to 1.

perfectns.nested_sampling.**point_importance**(*samples*, *thread_min_max*, *settings*, *simulate=False*)

Calculate the relative importance of each point for use in the dynamic nested sampling algorithm.

For more details see 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019).

Parameters

samples: 2d numpy array See dict_given_samples_arrry docstring for details of columns.

thread_min_max: 2d numpy array First column is starting logl of each thread and second column is ending logl.

settings: PerfectNSSettings object

simulate: bool, optional Passed to nestcheck.ns_run_utils.get_logw.

Returns

importance: 1d numpy array Relative point importances of the rows of the input samples array. Normalised so the biggest value in the array is equal to 1.

perfectns.nested_sampling.**samples_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 dictionary. Contains keys: 'logl', 'r', 'logx', 'thread_label', 'nlive_array', 'theta'

Returns

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

perfectns.nested_sampling.**z_importance**(*w_relative*, *nlive*)

Calculate the relative importance of each point for evidence calculation.

For more details see 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019).

Parameters

w_relative: 1d numpy array Relative point weights.

nlive: 1d numpu array Number of live points.

Returns

importance: 1d numpy array Relative point importances. Normalised so the biggest value in the array is equal to 1.

1.4.3 likelihoods

Classes representing spherically symmetric likelihoods.

Each likelihood class must contain a member function giving the log likelihood as a function of the radial coordinate r = **|theta|** and the number of dimensions

def logl_given_r(self, r, n_dim): ...

The number of dimensions is not stored in this class but in the PerfectNSSettings object to ensure it is the same for the likelihood and the prior.

Likelihood classes may also optionally contain the inverse function

def r_given_logl(self, logl, n_dim): ...

(although this is not needed to generate nested sampling runs). Another optional function is the analytic value of the log evidence for some given prior and dimension, which is useful for checking results:

def logz_analytic(self, prior, n_dim): ...

class perfectns.likelihoods.**Cauchy**(*likelihood_scale=1*)

Spherically symmetric Cauchy likelihood.

Methods

logl_given_r(self, r, n_dim)	Get loglikelihood values for input radial coordinates.
r_given_logl(self, logl, n_dim)	Get the radial coordinates corresponding to the input loglikelihood values.

logl_given_r (*self*, *r*, *n_dim*)
Get loglikelihood values for input radial coordinates.

Parameters

r: float or numpy array Radial coordinates.

n_dim: int Number of dimensions.

Returns

logl: same type and size as r Loglikelihood values.

r_given_logl (*self*, *logl*, *n_dim*)
Get the radial coordinates corresponding to the input loglikelihood values.

Parameters

logl: float or numpy array Loglikelihood values.

n_dim: int Number of dimensions.

Returns

r: same type and size as logl Radial coordinates.

class perfectns.likelihoods.**ExpPower** (*likelihood_scale=1*, *power=2*)
Spherically symmetric exponential power likelihood. When power=1, this is the same as the Gaussian likelihood.

Methods

logl_given_r(self, r, n_dim)	Get loglikelihood values for input radial coordinates.
r_given_logl(self, logl, n_dim)	Get the radial coordinates corresponding to the input loglikelihood values.

logl_given_r (*self*, *r*, *n_dim*)
Get loglikelihood values for input radial coordinates.

Parameters

r: float or numpy array Radial coordinates.

n_dim: int Number of dimensions.

Returns

logl: same type and size as r Loglikelihood values.

r_given_logl (*self*, *logl*, *n_dim*)
Get the radial coordinates corresponding to the input loglikelihood values.

Parameters

logl: float or numpy array Loglikelihood values.

n_dim: int Number of dimensions.

Returns

r: same type and size as logl Radial coordinates.

class perfectns.likelihoods.**Gaussian**(*likelihood_scale=1.0*)
Spherically symmetric Gaussian likelihood.

Methods

logl_given_r(self, r, n_dim)	Get loglikelihood values for input radial coordinates.
logz_analytic(self, prior, n_dim)	Returns analytic value of the log evidence for the input prior and dimension if it is available.
r_given_logl(self, logl, n_dim)	Get the radial coordinates corresponding to the input loglikelihood values.

logl_given_r(*self*, *r*, *n_dim*)
Get loglikelihood values for input radial coordinates.

Parameters

r: float or numpy array Radial coordinates.

n_dim: int Number of dimensions.

Returns

logl: same type and size as r Loglikelihood values.

logz_analytic(*self*, *prior*, *n_dim*)
Returns analytic value of the log evidence for the input prior and dimension if it is available.

If not set up for this prior an AssertionError is thrown (this is caught in functions which check analytical values where they are available).

Parameters

prior: object

n_dim: int Number of dimensions.

Returns

float Analytic value of log Z for this likelihood given the prior and number of dimensions.

r_given_logl(*self*, *logl*, *n_dim*)
Get the radial coordinates corresponding to the input loglikelihood values.

Parameters

logl: float or numpy array Loglikelihood values.

n_dim: int Number of dimensions.

Returns

r: same type and size as logl Radial coordinates.

1.4.4 priors

Classes representing spherically symmetric priors.

Each prior class must contain a member function giving the radial coordinate r = |theta| as a function of the log fraction of the prior volume remaining and the dimension

def r_given_logx(self, logx, n_dim): ...

The number of dimensions is not stored in this class but in the PerfectNSSettings object to ensure it is the same for the likelihood and the prior.

Prior classes may also optionally contain the inverse function

def logx_given_r(self, r, n_dim): ...

(although this is not needed to generate nested sampling runs).

class perfectns.priors.**Gaussian**(*prior_scale*)

Spherically symmetric Gaussian prior.

Methods

logx_given_r(self, r, n_dim)	Maps input radial coordinates to logx values.
r_given_logx(self, logx, n_dim)	Maps input logx values to radial coordinates.

logx_given_r(*self*, *r*, *n_dim*)

Maps input radial coordinates to logx values.

Parameters

r: float or numpy array

n_dim: int

Returns

logx: same type and size as r

r_given_logx(*self*, *logx*, *n_dim*)

Maps input logx values to radial coordinates.

Parameters

logx: float or numpy array

n_dim: int

Returns

r: same type and size as logx

class perfectns.priors.**GaussianCached**(*prior_scale*, ***kwargs*)

Spherically symmetric uniform prior. The scipy inverse gamma function is not numerically stable so cache r_given_logx by using the mpmath logx_given_r and linearly interpolating.

Methods

check_cache(self, n_dim)	Helper function which checks that the input dimension matches that of the cached interpolation function, and if needed recalculates it.
logx_given_r(self, r, n_dim)	Maps input radial coordinates to logx values.
r_given_logx(self, logx, n_dim)	Maps input logx values to radial coordinates.

check_cache (*self*, *n_dim*)
Helper function which checks that the input dimension matches that of the cached interpolation function, and if needed recalculates it.

Parameters

n_dim: int Number of dimensions

logx_given_r (*self*, *r*, *n_dim*)
Maps input radial coordinates to logx values.

Parameters

r: float or numpy array

n_dim: int

Returns

logx: same type and size as r

r_given_logx (*self*, *logx*, *n_dim*)
Maps input logx values to radial coordinates.

Parameters

logx: float or numpy array

n_dim: int

Returns

r: same type and size as logx

class perfectns.priors.**Uniform**(*prior_scale*)
Spherically symmetric uniform prior.

Methods

logx_given_r(self, r, n_dim)	Maps input radial coordinates to logx values.
r_given_logx(self, logx, n_dim)	Maps input logx values to radial coordinates.

logx_given_r (*self*, *r*, *n_dim*)
Maps input radial coordinates to logx values.

Parameters

r: float or numpy array

n_dim: int

Returns

logx: same type and size as r

r_given_logx (*self*, *logx*, *n_dim*)
Maps input logx values to radial coordinates.

Parameters

logx: float or numpy array

n_dim: int

Returns

r: same type and size as logx

1.4.5 estimators

Contains classes representing quantities which can be calculated from nested sampling run.

Each estimator class should contain a mandatory member function returning the value of the estimator for a nested sampling run:

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

This allows logw to be provided if many estimators are being calculated from the same run so logw is only calculated once. Otherwise logw is calculated from the run if required.

They may also optionally contain a function giving its analytical value for some given set of calculation settings (for use in checking results):

def analytical(self, settings): ...

as well as helper functions. Estimators should also contain class variables:

name: str used for results tables.

latex_name: str used for plotting results diagrams.

class perfectns.estimators.**CountSamples**
Number of samples in run.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.

class perfectns.estimators.**EstimatorBase**(*func*, ***kwargs*)
Base class for estimators.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.

class perfectns.estimators.**LogZ**
Natural log of Bayesian evidence.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.
analytical(settings)	Returns analytical value of estimator given settings.

static analytical(*settings*)
Returns analytical value of estimator given settings.

class perfectns.estimators.**ParamCred**(*probability*, *param_ind=0*)
One-tailed credible interval on the value of a single parameter (component of theta). By symmetry all parameters have the same distribution.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.
analytical(self, settings)	Returns analytical value of estimator given settings.

analytical (*self*, *settings*)
Returns analytical value of estimator given settings.

class perfectns.estimators.**ParamMean** (*param_ind=0*)
Mean of a single parameter (single component of theta). By symmetry all parameters have the same distribution.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.
analytical(settings)	Returns analytical value of estimator given settings.
ftilde(logx, settings)	Helper function for finding the analytic value of the estimator.

static analytical (*settings*)
Returns analytical value of estimator given settings.

static ftilde (*logx*, *settings*)
Helper function for finding the analytic value of the estimator. See "check_by_integrating" docstring for more details.

ftilde(X) is mean of f(theta) on the iso-likelihood contour L(theta) = L(X).

class perfectns.estimators.**ParamSquaredMean** (*param_ind=0*)
Mean of the square of single parameter (second moment of its posterior distribution). By symmetry all parameters have the same distribution.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.
analytical(self, settings)	Returns analytical value of estimator given settings.
ftilde(logx, settings)	Helper function for finding the analytic value of the estimator.

analytical (*self*, *settings*)
Returns analytical value of estimator given settings.

static ftilde (*logx*, *settings*)
Helper function for finding the analytic value of the estimator. See "check_by_integrating" docstring for more details.

ftilde(X) is mean of f(theta) on the iso-likelihood contour L(theta) = L(X).

class perfectns.estimators.**RCred** (*probability*, *from_theta=False*)
One-tailed credible interval on the value of **|theta|**.

Methods

__call__(self, ns_run[, logw, simulate])	Returns estimator value for run.

class perfectns.estimators.**RMean**(*from_theta=False*)
Mean of |thetal (the radial distance from the centre).

Methods

__call__(self, ns_run[, logw, simulate])	Overwrite __call__ from nestcheck r_mean to allow use of perfectns' 'r' key if from_theta=False, and to check all the dimensions have been sampled if from_theta=True.
analytical(self, settings)	Returns analytical value of estimator given settings.
ftilde(logx, settings)	Helper function for finding the analytic value of the estimator.

analytical(*self*, *settings*)
Returns analytical value of estimator given settings.

static ftilde(*logx*, *settings*)
Helper function for finding the analytic value of the estimator. See "check_by_integrating" docstring for more details.

ftilde(X) is mean of f(theta) on the iso-likelihood contour L(theta) = L(X).

class perfectns.estimators.**Z**
Bayesian evidence.

Methods

__call__(self, *args, **kwargs)	Returns estimator value for run.
analytical(settings)	Returns analytical value of estimator given settings.

static analytical(*settings*)
Returns analytical value of estimator given settings.

perfectns.estimators.**check_by_integrating**(*ftilde*, *settings*)
Return the true value of the estimator using numerical integration.

Chopin and Robert (2010) show that the expectation of some function f(theta) is given by the integral

int L(X) X ftilde(X) dX / Z,

where ftilde(X) is mean of f(theta) on the iso-likelihood contour L(theta) = L(X).

Parameters

ftilde: function

settings: PerfectNSSettings object

Returns

float The estimator's true value.

perfectns.estimators.**check_integrand**(*logx*, *ftilde*, *settings*)

Helper function to return integrand L(X) X ftilde(X) for checking estimator values by numerical integration. Note that the integral must be normalised by multiplying by a factor of (1/Z).

Parameters

logx: 1d numpy array Values on which to evaluate integrand.

ftilde: function See check_by_integrating docstring for more details.

settings: PerfectNSSettings object

Returns

1d numpy array Integrand evaluated at input logx coordinates.

perfectns.estimators.**get_true_estimator_values**(*estimators*, *settings*)

Calculates analytic values for estimators given the likelihood and prior in settings. If there is no method for calculating the values then np.nan is returned.

Parameters

estimators: estimator object or list of estimator objects

settings: PerfectNSSettings object

Returns

output: np.array of size (len(estimators),) if estimators is a list, float otherwise.

1.4.6 plots

Plotting functions.

perfectns.plots.**cdf_given_logx**(*estimator*, *value*, *logx*, *settings*)

Calculate CDF at where each column represents the CDF of the distribution of ftheta values on some iso-likelihood contour L = L(X).

Parameters

estimator: estimator object Function whose values we are getting the CDF of.

value: numpy array Function values at which to evaluate the CDF.

logx: numpy array of same size and shape as value. Logx values specifying contours - we calculate the CDF on each contour.

settings: PerfectNSSettings object

Returns

cdf: numpy array of same size and shape as value and logx

perfectns.plots.**plot_dynamic_nlive**(*dynamic_goals*, *settings_in*, ***kwargs*)

Plot the allocations of live points as a function of logX for different dynamic_goal settings. Plots also include analytically calculated distributions of relative posterior mass and relative posterior mass remaining.

Parameters

dynamic_goals: list of ints or None dynamic_goal setting values to plot.

settings_in: PerfectNSSettings object

tuned_dynamic_ps: list of bools, optional tuned_dynamic_ps settings corresponding to each dynamic goal settings. Defaults to False for all dynamic goals.

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.

load: bool, optional Should the nested sampling runs be loaded from cache if available?

save: bool, optional Should the nested sampling runs be cached?

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

n_run: int, optional How many runs to plot for each dynamic goal.

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

figsize: tuple, optional Size of figure in inches.

Returns

fig: matplotlib figure

perfectns.plots.**plot_parameter_logx_diagram**(*settings*, *ftheta*, ***kwargs*)

Plots parameter estimation diagram of the type described in Section 3.1 and shown in Figure 3 of "Sampling errors in nested sampling parameter estimation" (Higson et al., 2018).

Parameters

settings: PerfectNSSettings object

ftheta: estimator object function of parameters to plot

ymin: float, optional y axis (ftheta) min.

ymax: float, optional y axis (ftheta) max.

ylabel: str, optional y axis (ftheta) label.

logx_min: float, optional Lower limit of logx axis.

x_points: int, optional How many logx points to use in the plots.

y_points: int, optional

figsize: tuple, optional Size of figure in inches.

Returns

fig: matplotlib figure

perfectns.plots.**plot_rel_posterior_mass**(*likelihood_list*, *prior*, *dim_list*, *logx*, ***kwargs*)

Plot analytic distributions of the relative posterior mass for different likelihoods and dimensionalities as a function of logx (this is proportional to L(X)X, where L(X) is the likelihood).

Parameters

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

likelihood_list: list of perfectns likelihood objects Likelihoods to plot.

prior: perfectns prior object

dim_list: list of ints Dimensions to plot for each likelihood.

figsize: tuple, optional Size of figure in inches.

linestyles: list, optional List of different linestyles to use for each likelihood.

Returns

fig: matplotlib figure Figure showing relative posterior masses of input likelihoods

perfectns.plots.**posterior_cdf**(*estimator*, *values*, *logx*, *settings*)

Calculates the 1d posterior cumulative distribution function (CDF) for some estimator given the likelihood and prior settings.

Parameters

estimator: estimator object Function whose values we are getting the CDF of.

value: 1d numpy array Function values at which to evaluate the CDF.

logx: 1d numpy array Logx values over which to numericallly marginalise the probability distribution.

settings: PerfectNSSettings object

Returns

cdf: numpy array of same size and shape as values

perfectns.plots.**sigma_given_cdf**(*cdf*)

Maps cdf values in [0,1] to number of standard deviations from the median. scipy.special.erfinv is defined in [-1,+1] mapping to [-inf,+inf]. We want to map a CDF to the number of sigma from the median - i.e. from [0,+1] to [0,+inf] - so we need the argument (2*cdf - 1), and to take abs to get a positive answer. erf is definted in terms of int e^(-t^2) which corresponds to a Gaussian with sigma = 1/sqrt(2). So we must multiply sigma_temp by sqrt(2)

Parameters

cdf: float or numpy array

Returns

float or numpy array Same type and shape as input cdf.

1.4.7 results_tables

Functions used to generate results tables.

Used for results in 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019).

perfectns.results_tables.**get_bootstrap_results**(*n_run*, *n_simulate*, *estimator_list*, *settings*, ***kwargs*)

Generate data frame showing the standard deviations of the results of repeated calculations and estimated sampling errors from bootstrap resampling.

This function was used for Table 5 in 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019). See the paper for more details.

Parameters

n_run: int how many runs to use

n_simulate: int how many times to resample the nested sampling run in each bootstrap standard deviation estimate.

estimator_list: list of estimator objects

settings: PerfectNSSettings object

load: bool, optional should run data and results be loaded if available?

save: bool, optional should run data and results be saved?

parallel: bool, optional

cache_dir: str, optional Directory to use for caching.

add_sim_method: bool, optional should we also calculate standard deviations using the simulated weights method for comparison with bootstrap resampling? This method is inaccurate for parameter estimation.

n_simulate_ci: int, optional how many times to resample the nested sampling run in each bootstrap credible interval estimate. These may require more simulations than the standard deviation estimate.

run_random_seeds: list, optional list of random seeds to use for generating runs.

n_run_ci: int, optional how many runs to use for each credible interval estimate. You may want to set this to lower than n_run if n_simulate_ci is large as otherwise the credible interval estimate may take a long time.

cred_int: float, optional one-tailed credible interval to calculate

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 and therefore there will be no speedup from multiple nodes (must specify manually in this case).

Returns

results: pandas data frame results data frame. Contains two columns for each estimator - the second column (with '_unc' appended to the title) shows the numerical uncertainty in the first column. Contains rows:

true values: analytical values of estimators for this likelihood and posterior if available

repeats mean: mean calculation result repeats std: standard deviation of calculation results bs std / repeats std: mean bootstrap standard deviation estimate as

a fraction of the standard deviation of repeated results.

bs estimate % variation: standard deviation of bootstrap estimates as a percentage of the mean estimate.

[only if add sim method is True]:

sim std / repeats std: as for 'bs std / repeats std' but with simulation method standard deviation estimates.

sim estimate % variation: as for 'bs estimate % variation' but with simulation method standard deviation estimates.

bs [cred_int] CI: mean bootstrap credible interval estimate. bs +-1std % coverage: % of calculation results falling within +- 1

mean bootstrap standard deviation estimate of the mean.

bs [cred_int] CI % coverage: % of calculation results which are less than the mean bootstrap credible interval estimate.

perfectns.results_tables.**get_dynamic_results**(*n_run*, *dynamic_goals_in*, *estimator_list_in*, *settings_in*, ***kwargs*)

Generate data frame showing the standard deviations of the results of repeated calculations and efficiency gains (ratios of variances of results calculations) from different dynamic goals. To make the comparison fair, for

dynamic nested sampling settings.n_samples_max is set to slightly below the mean number of samples used by standard nested sampling.

This function was used for Tables 1, 2, 3 and 4, as well as to generate the results shown in figures 6 and 7 of 'Dynamic nested sampling: an improved algorithm for nested sampling parameter estimation and evidence calculation' (Higson et al., 2019). See the paper for a more detailed description.

Parameters

n_run: int how many runs to use

dynamic_goals_in: list of floats which dynamic goals to test

estimator_list_in: list of estimator objects

settings_in: PerfectNSSettings object

load: bool, optional should run data and results be loaded if available?

save: bool, optional should run data and results be saved?

overwrite_existing: bool, optional if a file exists already but we generate new run data, should we overwrite the existing file when saved?

run_random_seeds: list, optional list of random seeds to use for generating runs.

parallel: bool, optional

cache_dir: str, optional Directory to use for caching.

tuned_dynamic_ps: list of bools, same length as dynamic_goals_in, optional

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 and therefore there will be no speedup from multiple nodes (must specify manually in this case).

Returns

results: pandas data frame results data frame. Contains 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.

perfectns.results_tables.**merged_dynamic_results**(*dim_scale_list*, *likelihood_list*, *settings*, *estimator_list*, ***kwargs*)

Wrapper for running get_dynamic_results for many different likelihood, dimension and prior scales, and merging the output into a single data frame. See get_dynamic_results doccumentation for more details.

Parameters

dim_scale_list: list of tuples (dim, prior_scale) pairs to run

likelihood_list: list of likelihood objects

settings_in: PerfectNSSettings object

estimator_list: list of estimator objects

n_run: int, optional number of runs for use with each setting.

dynamic_goals_in: list of floats, optional which dynamic goals to test

(remaining kwargs passed to get_dynamic_results)

Returns

results: pandas data frame

1.4.8 maths_functions

Mathematical functions.

perfectns.maths_functions.**analytic_logx_terminate**(*settings*)

Find logx_terminate analytically by assuming all likelihood at very low X approximately equals the maximum likelihood. This approximation breaks down in very high dimensions.

Parameters

settings: PerfectNSSettings object

perfectns.maths_functions.**gaussian_logx_given_r**(*r*, *sigma*, *n_dim*)

Returns logx coordinate corresponding to r values for a Gaussian prior with the specificed standard deviation and dimension

Uses mpmath package for arbitary precision.

Parameters

r: float or numpy array Radial coordinates at which to evaluate logx.

sigma: float Standard deviation of Gaussian.

n_dim: int Number of dimensions.

Returns

logx: float or numpy array Logx coordinates corresponding to input radial coordinates.

perfectns.maths_functions.**gaussian_r_given_logx**(*logx*, *sigma*, *n_dim*)

Returns r coordinate corresponding to logx values for a Gaussian prior with the specificed standard deviation and dimension.

This uses scipy.special.gammaincinv and requires exponentiating logx, so numerical errors occur with very low logx values.

Parameters

logx: float or numpy array Logx coordinates at which to work out r.

sigma: float Standard deviation of Gaussian.

n_dim: int Number of dimensions.

Returns

float or numpy array Radial coordinates corresponding to input log X values.

perfectns.maths_functions.**log_cauchy_given_r**(*r*, *sigma*, *n_dim*)

Returns the natural log of a normalised, uncorrelated Cauchy distribution with 1 degree of freedom.

Parameters

r: float or numpy array

sigma: float

n_dim: int

Returns

logl: float or numpy array Loglikelihood values corresponding to input radial coordinates.

perfectns.maths_functions.**log_exp_power_given_r**(*r*, *sigma*, *n_dim*, *b=0.5*)
Returns the natural log of an exponential power distribution. This equals a Gaussian distribution when b=1.

Parameters

r: float or numpy array

sigma: float

n_dim: int

b: float, optional

Returns

logl: float or numpy array Loglikelihood values corresponding to input radial coordinates.

perfectns.maths_functions.**log_gaussian_given_r**(*r*, *sigma*, *n_dim*)
Returns the natural log of a normalised, uncorrelated Gaussian likelihood with equal variance in all n_dim dimensions.

Parameters

r: float or numpy array

sigma: float

n_dim: int

Returns

logl: float or numpy array Loglikelihood values corresponding to input radial coordinates.

perfectns.maths_functions.**logx_terminate_bound**(*logl_max*, *termination_fraction*, *logz_analytic*)
Find a lower bound logx_terminate analytically by assuming all likelihood at very low X approximately equals the maximum likelihood. This approximation breaks down in very high dimensions and the true logx terminate required will be larger.

We want:

Z_term = termination_fraction * Z_analytic = int_0^Xterm L(X) dX approx= Xterm L_max,

so logx_term = log(termination_fraction) + log(Z_analytic) - logl_max

Parameters

logl_max: float Maximum loglikelihood.

termination_frac: float Fraction of posterior mass remaining at termination.

logz_analytic: float True value of log evidence.

Returns

float

perfectns.maths_functions.**nsphere_logvol**(*dim*, *radius=1.0*)
Returns the natural log of the hypervolume of a unit nsphere of specified dimension. Useful for very high dimensions. Formulae is from https://en.wikipedia.org/wiki/N-sphere#Volume_and_surface_area

Parameters

dim: int

radius: float, optional

Returns

float

perfectns.maths_functions.**nsphere_logx_given_r**(*r*, *r_max*, *n_dim*)

Finds logx assuming the prior is an n-dimensional sphere co-centred with a spherically symmetric likelihood. This will return an answer of the same type (float or numpy array) as the input {r}.

Parameters

r: float or numpy array

r_max: float

n_dim: int

Returns

logx: float or numpy array. Logx coordinates corresponding to input radial coordinates.

perfectns.maths_functions.**nsphere_r_given_logx**(*logx*, *r_max*, *n_dim*)

Finds r coordinates given input logx values for a uniform prior within an n-dimensional sphere co-centred with a spherically symmetric likelihood. This will return an answer of the same type (float or numpy array) as the input {logx}.

Parameters

logx: float or numpy array

r_max: float Boundry of the spherically symmetric prior.

n_dim: int Number of dimensions

Returns

r: float or numpy array Radial coordinates corresponding to logx.

perfectns.maths_functions.**r_given_log_cauchy**(*logl*, *sigma*, *n_dim*)

Returns the radius for a given logl of a normalised, uncorrelated Cauchy distribution with 1 degree of freedom.

Parameters

logl: float or numpy array

sigma: float

n_dim: int

Returns

r: float or numpy array Radial coordinates corresponding to input logl values.

perfectns.maths_functions.**r_given_log_exp_power**(*logl*, *sigma*, *n_dim*, *b=0.5*)

Returns the natural log of an exponential power distribution. This equals a Gaussian distribution when b=1.

Parameters

logl: float or numpy array

sigma: float

n_dim: int

b: float, optional

Returns

r: float or numpy array Radial coordinates corresponding to input logl values.

perfectns.maths_functions.**r_given_log_gaussian**(*logl*, *sigma*, *n_dim*)

Returns the radius of a given logl for a normalised, uncorrelated Gaussian with equal variance in all n_dim dimensions.

Parameters

logl: float or numpy array

sigma: float

n_dim: int

Returns

r: float or numpy array Radial coordinates corresponding to input logl values.

perfectns.maths_functions.**sample_nsphere_shells**(*r*, *n_dim*, *n_sample*)

Wrapper calling either sample_nsphere_shells_normal or sample_nsphere_shells_beta depending on the dimension and number of samples required.

See the docstrings of sample_nsphere_shells_normal and sample_nsphere_shells_beta for more information.

perfectns.maths_functions.**sample_nsphere_shells_beta**(*r*, *n_dim*, *n_sample=None*)

Given some 1d numpy array of radial coordinates r, return a numpy array of sample coordinates on the hyperspherical shells with each radial coordinate.

Sample single parameters on n_dim-dimensional sphere independently, as as described in Section 3.2 of 'Sampling Errors in nested sampling parameter estimation' (Higson et al., 2018).

NB if each parameter is sampled independently and combined into a vector, that vector will not have a magnitude r.

The n_sample argument can be used to set the number of parameters for which samples are returned (by symmetry they all have the same distribution). This is useful for saving memory in high dimensions.

This function is much quicker than sample_nsphere_shells_normal when n_dim is high and n_sample is low.

Parameters

r: 1d numpy array

n_dim: int

n_sample: int or None, optional Number of parameters to include in output for each sample.

Returns

thetas: 2d numpy array Each row is a random sample from the shells defined by the corresponding input r coordinate.

perfectns.maths_functions.**sample_nsphere_shells_normal**(*r*, *n_dim*, *n_sample=None*)

Given some 1d numpy array of radial coordinates r, return a numpy array of sample coordinates on the hyperspherical shells with each radial coordinate.

This works by using the symmetry of the normal distribution to sample isotropically, then normalising each set of samples to lie on a spherical shell of the correct radius.

The n_sample argument can be used to set the number of parameters for which samples are returned (by symmetry they all have the same distribution). This is useful for saving memory in high dimensions.

Parameters

r: 1d numpy array

n_dim: int

n_sample: int or None, optional Number of parameters to include in output for each sample.

Returns

thetas: 2d numpy array Each row is a random sample from the shells defined by the corresponding input r coordinate.

1.4.9 cached_gaussian_prior

Contains helper functions for the 'gaussian_cached' prior.

This is needed as the 'gaussian' prior suffers from overflow errors and numerical instability for very low values of X, which are reached in high dimensional problems.

`perfectns.cached_gaussian_prior.`**`interp_r_logx_dict`**(*n_dim*, *prior_scale*, ***kwargs*)

Generate a dictionary containing arrays of logx and r values for use in interpolation, as well as a record of the settings used.

Parameters

n_dim: int Number of dimensions.

prior_scale: float Gaussian prior's standard deviation.

logx_min: float Values for interpolation are generated between logx_min and logx_max.

save_dict: bool, optional Whether or not to cache the output dictionary.

cache_dir: str, optional Directory in which to cache output if save_dict is True.

interp_density: float Number of points to include in interpolation arrays per unit of logx (the number of points is cast to int so this can be a fraction).

logx_max: float, optional Values for interpolation are generated between logx_min and logx_max.

Returns

interp_dict: dict

CHAPTER

TWO

ATTRIBUTION

If `perfectns` is useful for your academic research, please cite the two papers introducing the software and the dynamic nested sampling algorithm. The BibTeX is:

```
@article{Higson2018perfectns,
author = {Higson, Edward},
title = {perfectns: perfect dynamic and standard nested sampling for spherically␣
↪symmetric likelihoods and priors},
doi = {10.21105/joss.00985},
journal = {Journal of Open Source Software},
number = {30},
pages = {985},
volume = {3},
year = {2018}}

@article{Higson2019dynamic,
author={Higson, Edward and Handley, Will and Hobson, Michael and Lasenby, Anthony},
title={Dynamic nested sampling: an improved algorithm for parameter estimation and␣
↪evidence calculation},
year={2019},
journal={Statistics and Computing},
doi={10.1007/s11222-018-9844-0},
url={https://doi.org/10.1007/s11222-018-9844-0},
archivePrefix={arXiv},
arxivId={1704.03459}}
```

CHAPTER

THREE

CHANGELOG

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

CHAPTER

FOUR

CONTRIBUTIONS

Contributions are welcome! Development takes place on github:

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

FIVE

AUTHORS & LICENSE

PYTHON MODULE INDEX

p

INDEX

M

N

P

R

S

U

Z