dyPolyChord Documentation

Release 0.0.0

Edward Higson

Aug 17, 2019

CONTENTS

1 Documentation contents **3**
- 1.1 Installation . . . 3
- 1.2 dyPolyChord Demo . . . 4
- 1.3 Detailed API documentation . . . 8
- 1.4 Performance . . . 12
- 1.5 Example python likelihoods and priors . . . 13

2 Attribution **21**

3 Changelog **23**

4 Contributions **25**

5 Authors & License **27**

Python Module Index **29**

Index **31**

Nested sampling is a numerical method for Bayesian computation which simultaneously calculates posterior samples and an estimate of the Bayesian evidence for a given likelihood and prior. The approach is popular in scientific research, and performs well compared to Markov chain Monte Carlo (MCMC)-based sampling for multi-modal or degenerate posterior distributions.

`dyPolyChord` implements dynamic nested sampling using the efficient `PolyChord` sampler to provide state-of-the-art nested sampling performance. Any likelihoods and priors which work with `PolyChord` can be used (Python, C++ or Fortran), and the output files produced are in the `PolyChord` format.

To get started, see the installation instructions and the demo. N.B. `dyPolyChord` requires `PolyChord` v1.14 or higher.

For more details about dynamic nested sampling, see the dynamic nested sampling paper (Higson et al., 2019). For a discussion of `dyPolyChord`'s performance, see the performance section of the documentation.

CHAPTER

ONE

DOCUMENTATION CONTENTS

1.1 Installation

`dyPolyChord` is compatible with python 2.7 and >=3.4, and can be installed with pip:

```
pip install dyPolyChord
```

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

```
git clone https://github.com/ejhigson/dyPolyChord.git
cd dyPolyChord
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).

1.1.1 Dependencies

`dyPolyChord` requires:

- `PolyChord` >=v1.14;
- `numpy` >=v1.13;
- `scipy` >=v1.0.0;
- `nestcheck` >=v0.1.8.

`PolyChord` is available at https://github.com/PolyChord/PolyChordLite and has its own installation and licence instructions; see the link for more information. Note running `dyPolyChord` with compiled C++ or Fortran likelihoods does not require the installation of `PolyChord`'s Python interface.

1.1.2 Tests

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

```
nosetests
```

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

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

Note that these tests will run without `PolyChord`. This is to allow all the `dyPolyChord` code (including code specifically for Python or compiled likelihoods) to be tested without the need for the user to compile any executables or install `pypolychord`. If `pypolychord` is installed, the tests will also run calculations using Python likelihoods and check their results (otherwise these tests are skipped).

Note:

You can download this demo as a Jupyter notebook here and run it interactively yourself.

1.2 dyPolyChord Demo

The main user-facing function is `dyPolyChord.run_dypolychord`, which performs dynamic nested sampling.

Likelihoods and priors are specified within a Python callable, which can be used to run PolyChord on the likelihood and prior with an input settings dictionary. Tools for making such a callable are provided in `pypolychord_utils.py` (python likelihoods and priors) and `polychord_utils.py` (compiled C++ and Fortran likelihoods and priors).

In addition the user can specify PolyChord settings (as a dictionary), and can choose whether to prioritise parameter estimation or evidence calculation via the `dynamic_goal` argument - see the dynamic nested sampling paper (Higson et al., 2019) for an explanation.

1.2.1 Compiled (C++ or Fortran) likelihoods and priors

C++ and Fortran likelihoods used by PolyChord can also be used by `dyPolyChord` (they must be able to read settings from .ini files). These must be compiled to executables within the PolyChord directory, via commands such as

```
$ make gaussain  # PolyChord gaussian example
```

or

```
$ make polychord_CC_ini  # PolyChord template C++ likelihood which reads .ini file
```

See the PolyChord README for more details. `dyPolyChord` simply needs the file path to the executable, which it runs via `os.system` - settings are specified by writing temporary .ini files.

[ ]:
```
import dyPolyChord.polychord_utils
import dyPolyChord


# Definte the distribution to sample (likelihood, prior, number of dimensions)
ex_command = './gaussian'  # path to compiled executable likelihood
# The prior must be specified as strings via .ini files. get_prior_block_str provides␣
↪a
# convenient function for making such PolyChord-formatted strings. See its docstring␣
↪and
# the PolyChord documentation for more details
ndim = 10
prior_str = dyPolyChord.polychord_utils.get_prior_block_str(
    'gaussian',  # name of prior - see PolyChord for a list of allowed priors
    [0.0, 10.0],  # parameters of the prior
    ndim)
```

(continues on next page)

(continued from previous page)

```
# Make a callable for running PolyChord
my_callable = dyPolyChord.polychord_utils.RunCompiledPolyChord(
    ex_command, prior_str)

# Specify sampler settings (see run_dynamic_ns.py documentation for more details)
dynamic_goal = 1.0  # whether to maximise parameter estimation or evidence accuracy.
ninit = 100          # number of live points to use in initial exploratory run.
nlive_const = 500   # total computational budget is the same as standard nested␣
↪sampling with nlive_const live points.
settings_dict = {'file_root': 'gaussian',
                 'base_dir': 'chains',
                 'seed': 1}

# Run dyPolyChord
dyPolyChord.run_dypolychord(my_callable, dynamic_goal, settings_dict,
                            ninit=ninit, nlive_const=nlive_const)
```

1.2.2 Python likelihoods and priors

Python likelihoods and priors must be defined as functions or callable classes, just as for running pypolychord (PolyChord's python wrapper). Otherwise the process is very similar to that with compiled likelihoods.

Note that pypolychord used to be called PyPolyChord before PolyChord v1.15. `dyPolyChord` is compatible with both the new and old names; if pypolychord cannot be imported then we try importing PyPolyChord instead.

[ ]:
```
import dyPolyChord.python_likelihoods as likelihoods  # Import some example python␣
↪likelihoods
import dyPolyChord.python_priors as priors  # Import some example python priors
import dyPolyChord.pypolychord_utils
import dyPolyChord


# Definte the distribution to sample (likelihood, prior, number of dimensions)
ndim = 10
likelihood = likelihoods.Gaussian(sigma=1.0)
prior = priors.Gaussian(sigma=10.0)

# Make a callable for running PolyChord
my_callable = dyPolyChord.pypolychord_utils.RunPyPolyChord(
    likelihood, prior, ndim)

# Specify sampler settings (see run_dynamic_ns.py documentation for more details)
dynamic_goal = 1.0  # whether to maximise parameter estimation or evidence accuracy.
ninit = 100          # number of live points to use in initial exploratory run.
nlive_const = 500   # total computational budget is the same as standard nested␣
↪sampling with nlive_const live points.
settings_dict = {'file_root': 'gaussian',
                 'base_dir': 'chains',
                 'seed': 1}

# Run dyPolyChord
dyPolyChord.run_dypolychord(my_callable, dynamic_goal, settings_dict,
                            ninit=ninit, nlive_const=nlive_const)
```

1.2.3 Parallelisation

Compiled likelihoods and priors

To run compiled likelihoods in parallel with MPI, specify an mpirun command in the `mpi_str` argument when initializing your `RunPyPolyChord` object. For example to run with 8 processes, use

```
[ ]: my_callable = dyPolyChord.polychord_utils.RunCompiledPolyChord(
         ex_command, prior_str, mpi_str='mpirun -np 8')
```

The callable can then be used with `run_dypolychord` as normal.

Python likelihoods and priors

You must import `mpi4py`, create an `MPI.COMM_WORLD` object and pass it to `run_dypolychord` as an argument.

```
[ ]: from mpi4py import MPI
comm = MPI.COMM_WORLD

dyPolyChord.run_dypolychord(my_callable, dynamic_goal, settings_dict,
                            ninit=ninit, nlive_const=nlive_const, comm=comm)
```

You can then run your script with mpirun:

```
$ mpirun -np 8 my_dypolychord_script.py
```

Repeated runs

If you want to perform a number of independent `dyPolyChord` calculations (such as repeating the same calculation many times) then, as this is "embarrassingly parallel", you don't need MPI and can instead perform many `dyPolyChord` runs in parallel using python's `concurrent.futures`. This also allows reliable random seeding for reproducible results, which is not possible with MPI due to the unpredictable order in which slave processes are called by PolyChord. Note that for this to work PolyChord must be installed without MPI.

For an example of this type of usage, see the code used to make the results for the dynamic nested sampling paper (https://github.com/ejhigson/dns).

1.2.4 Checking the output

Running `dyPolyChord` produces PolyChord-format output files in `settings['base_dir']`. These output files can be analysed in the same way as other PolyChord output files.

One convenient package for doing this in python is `nestcheck` (http://nestcheck.readthedocs.io/en/latest/). We can use it to load and analyse the results from the 10-dimensional Gaussian produced by running the second cell in this demo (the example python likelihood and prior).

```
[1]: import nestcheck.data_processing
import nestcheck.estimators as e

# load the run
run = nestcheck.data_processing.process_polychord_run(
    'gaussian',  # = settings['file_root']
    'chains')    # = settings['base_dir']
```

(continues on next page)

(continued from previous page)

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

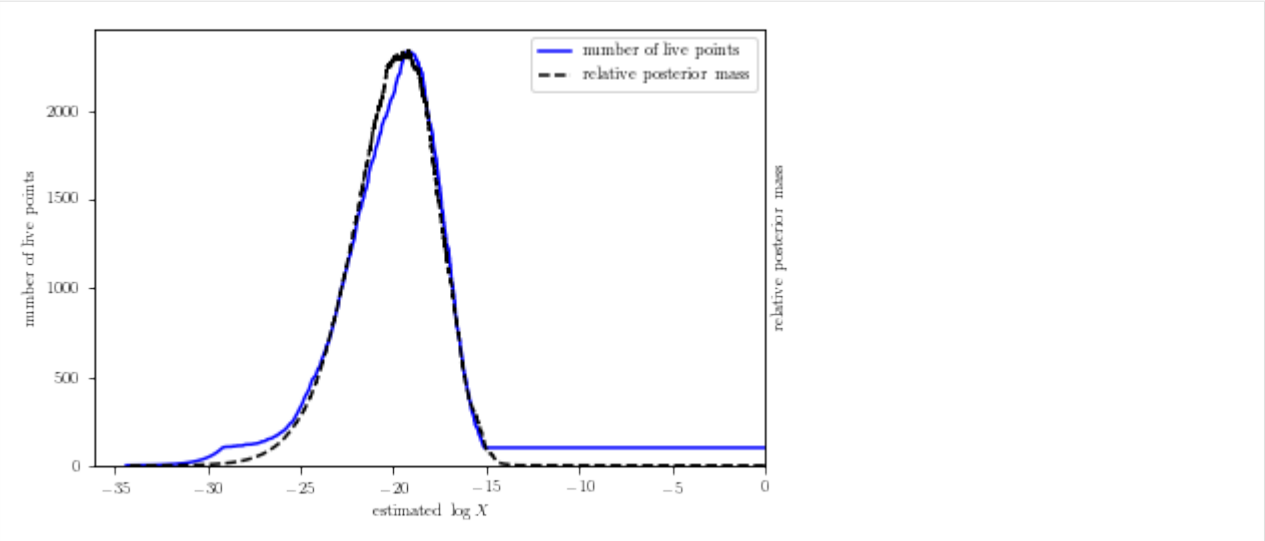
```
The log evidence estimate using the first run is -31.97320739404335
The estimateed the mean of the first parameter is -0.009077055705321546
```

As an illustration, lets use `nestcheck` to check `dyPolyChord`'s allocation of live points roughly matches the distribution of posterior mass, which it should do when the dynamic goal setting equals 1. For a detailed explanation of this type of plot, see Figure 4 in the dynamic nested sampling paper (Higson et al., 2019) and its caption.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import nestcheck.ns_run_utils
%matplotlib inline

# get the sample's estimated logX co-ordinates and relative posterior mass
logx = nestcheck.ns_run_utils.get_logx(run['nlive_array'])
logw = logx + run['logl']
w_rel = np.exp(logw - logw.max())

# plot nlive and w_rel on same axis
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax2 = ax1.twinx()
l1 = ax1.plot(logx, run['nlive_array'], label='number of live points', color='blue')
l2 = ax2.plot(logx, w_rel, label='relative posterior mass', color='black', linestyle=
↪'dashed')
lines = l1 + l2
ax1.legend(lines, [l.get_label() for l in lines], loc=0)
ax1.set_xlabel('estimated $\log X$')
ax1.set_xlim(right=0.0)
ax1.set_ylim(bottom=0.0)
ax1.set_ylabel('number of live points')
ax2.set_ylim(bottom=0.0)
ax2.set_yticks([])
ax2.set_ylabel('relative posterior mass')
plt.show()
```



It looks like `dyPolyChord`'s allocation of samples closely matches the regions with high posterior mass, as expected.

Note that this plot is only approximate as the $\log X$ values (x coordinates) are estimated from just the one run, and are correlated with the posterior mass estimates. For a more accurate version, see Figure 4 of Higson et al., (2019).

1.3 Detailed API documentation

This page contains documentation for the user-facing parts of the `dyPolyChord` package. For details of the internal workings of the package, look at the source code and its documentation at https://github.com/ejhigson/dyPolyChord.

1.3.1 run_dynamic_ns

Contains main function for running dynamic nested sampling.

`dyPolyChord.run_dynamic_ns.`**`run_dypolychord`**(*run_polychord*, *dynamic_goal*, *settings_dict_in*, ***kwargs*)

Performs dynamic nested sampling using the algorithm described in Appendix F of "Dynamic nested sampling: an improved algorithm for parameter estimation and evidence calculation" (Higson et al., 2019). This proceeds in 4 steps:

1) Generate an initial run with a constant number of live points n_{init}. This process is run in chuncks using PolyChord's max_ndead setting to allow periodic saving of .resume files so the initial run can be resumed at different points.

2) Calculate an allocation of the number of live points at each likelihood for use in step 3. Also cleans up resume files and saves relevant information.

3) Generate dynamic nested sampling run using the calculated live point allocation.

4) Combine the initial and dynamic runs and write output files in the PolyChord format, and remove the intermediate output files produced.

The output files are of the same format produced by `PolyChord`, and contain posterior samples and an estimate of the Bayesian evidence. Further analysis, including estimating uncertainties, can be performed with `nestcheck`.

Like for `PolyChord`, the output files are saved in base_dir (specified in settings_dict_in, default value is 'chains'). Their names are determined by file_root (also specified in settings_dict_in). `dyPolyChord` ensures the following following files are always produced:

- [base_dir]/[file_root].stats: run statistics including an estimate of the Bayesian evidence;
- [base_dir]/[file_root]_dead.txt: posterior samples;
- [base_dir]/[file_root]_dead-birth.txt: as above but with an extra column containing information about when points were sampled.

For more information about the output format, see `PolyChord`'s documentation. Note that `dyPolyChord` is not able to produce all of the types of output files made by `PolyChord` - see check_settings's documentation for more information. In addition, a number of intermediate files are produced during the dynamic nested sampling process which are removed by default when the process finishes. See clean_extra_output's documentation for more details.

Parameters

run_polychord: callable Callable which runs PolyChord with the desired likelihood and prior, and takes a settings dictionary as its argument.

dynamic_goal: float or int Number in (0, 1) which determines how to allocate computational effort between parameter estimation and evidence calculation. See the dynamic nested sampling paper for more details.

settings_dict: dict PolyChord settings to use (see check_settings for information on allowed and default settings).

nlive_const: int, optional Used to calculate total number of samples if max_ndead not specified in settings. The total number of samples used is the estimated number that would be taken by a nested sampling run with a constant number of live points nlive_const.

ninit: int, optional Number of live points to use for the initial exporatory run (Step 1).

ninit_step: int, optional Number of samples taken between saving .resume files in Step 1.

seed_increment: int, optional If seeding is used (PolyChord seed setting >= 0), this increment is added to PolyChord's random seed each time it is run to avoid repeated points. When running in parallel using MPI, PolyChord hashes the seed with the MPI rank using IEOR. Hence you need seed_increment to be > number of processors to ensure no two processes use the same seed. When running repeated results you need to increment the seed used for each run by some number >> seed_increment.

smoothing_filter: func, optional Smoothing to apply to the nlive allocation (if any).

stats_means_errs: bool, optional Whether to include estimates of logZ and parameter mean values and their uncertainties in the .stats file. This is passed to nestcheck's write_run_output; see its documentation for more details.

clean: bool, optional Clean the additional output files made by dyPolyChord, leaving only output files for the combined run in PolyChord format. When debugging this can be set to False to allow inspection of intermediate output.

resume_dyn_run: bool, optional Resume a partially completed dyPolyChord run using its cached output files. Resuming is only possible if the initial exploratory run finished and the process reached the dynamic run stage. If the run is resumed with different settings to what were used the first time then this may give unexpected results.

`dyPolyChord.run_dynamic_ns.`**`check_settings`**(*settings_dict_in*)

Checks the input dictionary of PolyChord settings. Issues warnings where these are not appropriate, and adds default values.

Parameters

settings_dict_in: dict PolyChord settings to use.

Returns

settings_dict: dict Updated settings dictionary including default and mandatory values.

output_settings: dict Settings for writing output files which are saved until the final output files are calculated at the end.

1.3.2 polychord_utils

Functions for running dyPolyChord using compiled PolyChord C++ or Fortran likelihoods.

class dyPolyChord.polychord_utils.**RunCompiledPolyChord**(*executable_path*, *prior_str*, ***kwargs*)

Object for running a compiled PolyChord executable with specified inputs.

__call__(*self*, *settings_dict*, *comm=None*)

Run PolyChord with the input settings by writing a .ini file then using the compiled likelihood specified in executable_path.

See the PolyChord documentation for more details.

Parameters

settings_dict: dict Input PolyChord settings.

comm: None, optional Not used. Included only so __call__ has the same arguments as the equivalent python function (which uses the comm argument for runnign with MPI).

__init__(*self*, *executable_path*, *prior_str*, ***kwargs*)

Specify path to executable, priors and derived parameters.

Parameters

executable_path: str Path to compiled likelihood. If this is in the directory from which dyPolyChord is being run, you may need to prepend "./" to the executable name for it to work.

prior_str: str String specifying prior in the format required for PolyChord .ini files (see get_prior_block_str for more details).

config_str: str, optional String to be written to [root].cfg file if required.

derived_str: str or None, optional String specifying prior in the format required for PolyChord .ini files (see prior_str for more details).

mpi_str: str or None, optional Optional mpi command to preprend to run command. For example to run with 8 processors, use mpi_str = 'mprun -np 8'. Note that PolyChord must be installed with MPI enabled to allow running with MPI.

__weakref__

list of weak references to the object (if defined)

ini_string(*self*, *settings*)

Get a PolyChord format .ini file string based on the input settings.

Parameters

settings: dict

Returns

string: str

dyPolyChord.polychord_utils.**format_setting**(*setting*)

Return setting as string in the format needed for PolyChord's .ini files. These use 'T' for True and 'F' for False, and require lists of numbers written separated by spaces and without commas or brackets.

Parameters

setting: (can be any type for which str(settings) works)

Returns

str

dyPolyChord.polychord_utils.**get_prior_block_str**(*prior_name*, *prior_params*, *nparam*, ***kwargs*)

Returns a PolyChord format prior block for inclusion in PolyChord .ini files.

See the PolyChord documentation for more details.

Parameters

prior_name: str Name of prior. See the PolyChord documentation for a list of currently available priors and details of how to add your own.

prior_params: str, float or list of strs and floats Parameters for the prior function.

nparam: int Number of parameters.

start_param: int, optional Where to start param numbering. For when multiple prior blocks are being used.

block: int, optional Number of block (only needed when using multiple prior blocks).

speed: int, optional Use to specify fast and slow parameters if required. See the PolyChord documentation for more details.

Returns

block_str: str PolyChord format prior block string for ini file.

dyPolyChord.polychord_utils.**python_block_prior_to_str**(*bp_obj*)

As for python_prior_to_str, but for BlockPrior objects of the type defined in python_priors.py. python_prior_to_str is called seperately on every block.

Parameters

prior_obj: python prior object Of the type defined in python_priors.py.

kwargs: dict, optional Passed to get_prior_block_str (see its docstring for more details).

Returns

block_str: str PolyChord format prior block string for ini file.

dyPolyChord.polychord_utils.**python_prior_to_str**(*prior*, ***kwargs*)

Utility function for mapping python priors (of the type in python_priors.py) to ini file format strings used for compiled (C++ or Fortran) likelihoods.

The input prior must correspond to a prior function set up in PolyChord/src/polychord/priors.f90. You can easily add your own too. Note that some of the priors are only available in PolyChord >= v1.15.

Parameters

prior_obj: python prior object Of the type defined in python_priors.py

kwargs: dict, optional Passed to get_prior_block_str (see its docstring for more details).

Returns

block_str: str PolyChord format prior block string for ini file.

1.3.3 pypolychord_utils

Functions for running dyPolyChord using pypolychord (PolyChord's built-in python wrapper) using with python likelihoods and priors.

Note that pypolychord was called PyPolyChord before PolyChord v1.15; if pypolychord cannot be imported then we try importing it using its old name instead for backwards compatibility.

class dyPolyChord.pypolychord_utils.**RunPyPolyChord**(*likelihood*, *prior*, *ndim*, *nderived=0*)

Callable class for running PolyChord in Python with specified the settings.

__call__(*self*, *settings_dict*, *comm=None*)

Runs pypolychord with specified inputs and writes output files. See the pypolychord documentation for more details.

Parameters

settings_dict: dict Input PolyChord settings.

comm: None or mpi4py MPI.COMM object, optional For MPI parallelisation.

__init__(*self*, *likelihood*, *prior*, *ndim*, *nderived=0*)

Specify likelihood, prior and number of dimensions in calculation.

Parameters

likelihood: func

prior: func

ndim: int

nderived: int, optional

__weakref__

list of weak references to the object (if defined)

1.4 Performance

dyPolyChord uses PolyChord to perform dynamic nested sampling by running it from within a python wrapper. Dynamic nested sampling allows significant increases in performance over standard nested sampling, with the largest efficiency gains for high-dimensional parameter estimation (PolyChord and dyPolyChord can handle calculations with over 100 dimensions).

Like PolyChord, dyPolyChord is optimized for calculations where the main computational cost is sampling new live points. For empirical tests of dyPolyChord's performance, see the dynamic nested sampling paper (Higson et al., 2017). These tests can be reproduced using the code at https://github.com/ejhigson/dns.

dyPolyChord uses a version of the dynamic nested sampling algorithm designed to minimise the computational overhead of allocating additional samples, so this should typically be a small part of the total computational cost. However this overhead may become significant for calculations where likelihood evaluations are fast and a large number of MPI processes are used (the saving, loading and processing of the initial exploratory samples is not currently fully parallelised).

It is also worth noting that `PolyChord`'s slice sampling-based implementation is less efficient than `MultiNest` (which uses rejection sampling) for low dimensional problems; see Handley et al. (2015) for more details. However for calculations using `dyPolyChord` this is may be offset by efficiency gains from dynamic nested sampling.

1.5 Example python likelihoods and priors

Some example PolyChord python likelihoods and priors. You can also add your own!

1.5.1 python_likelihoods

Loglikelihood functions for use with PolyChord's python interface.

PolyChord >= v1.14 requires likelihoods to be callables with parameter and return signatures:

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood.

phi: list of length nderived Any derived parameters.

We use classes with the loglikelihood defined in the __call__ property, as this provides convenient way of storing other information such as hyperparameter values. These objects can be used in the same way as functions due to python's "duck typing" (alternatively you can define likelihoods using functions).

class dyPolyChord.python_likelihoods.**Gaussian**(*sigma=1.0*, *nderived=0*)

Symmetric Gaussian loglikelihood centered on the origin.

__call__(*self*, *theta*)

Calculate loglikelihood(theta), as well as any derived parameters.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *sigma=1.0*, *nderived=0*)

Set up likelihood object's hyperparameter values.

Parameters

sigma: float, optional Standard deviation of Gaussian (the same for each parameter).

nderived: int, optional Number of derived parameters.

class dyPolyChord.python_likelihoods.**GaussianMix**(*sep=4*, *weights=(0.4*, *0.3*, *0.2*, *0.1)*, *sigma=1*, *nderived=0*)

Gaussian mixture likelihood in ≥ 2 dimensions with up to 4 compoents.

Each component has the same standard deviation σ, and they their centres respectively have (θ_1, θ_2) coordinates:

(0, sep), (0, -sep), (sep, 0), (-sep, 0).

__call__(*self*, *theta*)

Calculate loglikelihood(theta), as well as any derived parameters.

N.B. this loglikelihood is not normalised.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *sep=4*, *weights=(0.4*, *0.3*, *0.2*, *0.1)*, *sigma=1*, *nderived=0*)

Define likelihood hyperparameter values.

Parameters

sep: float Distance from each Gaussian to the origin.

weights: iterable of floats weights of each Gaussian component.

sigma: float Stanard deviation of Gaussian components.

nderived: int, optional Number of derived parameters.

class dyPolyChord.python_likelihoods.**GaussianShell**(*sigma=0.2*, *rshell=2*, *nderived=0*)

Gaussian Shell loglikelihood centred on the origin.

__call__(*self*, *theta*)

Calculate loglikelihood(theta), as well as any derived parameters.

N.B. this loglikelihood is not normalised.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *sigma=0.2*, *rshell=2*, *nderived=0*)

Set up likelihood object's hyperparameter values.

Parameters

sigma: float, optional Standard deviation of Gaussian (the same for each parameter).

rshell: float, optional Distance of shell peak from origin.

nderived: int, optional Number of derived parameters.

class dyPolyChord.python_likelihoods.**LogGammaMix**(*nderived=0*)

The loggamma mix used in Beaujean and Caldwell (2013) and Feroz et al. (2013).

__call__(*self*, *theta*)

Calculate loglikelihood(theta), as well as any derived parameters.

N.B. this loglikelihood is not normalised.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *nderived=0*)
Define likelihood hyperparameter values.

Parameters

nderived: int, optional Number of derived parameters.

class dyPolyChord.python_likelihoods.**Rastrigin**(*a=10*, *nderived=0*)
Rastrigin loglikelihood as described in "PolyChord: next-generation nested sampling" (Handley et al., 2015).

__call__(*self*, *theta*)
Calculate loglikelihood(theta), as well as any derived parameters.

N.B. this loglikelihood is not normalised.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *a=10*, *nderived=0*)
Set up likelihood object's hyperparameter values.

Parameters

a: float, optional

nderived: int, optional Number of derived parameters.

class dyPolyChord.python_likelihoods.**Rosenbrock**(*a=1*, *b=100*, *nderived=0*)
Rosenbrock loglikelihood as described in "PolyChord: next-generation nested sampling" (Handley et al., 2015).

__call__(*self*, *theta*)
Calculate loglikelihood(theta), as well as any derived parameters.

N.B. this loglikelihood is not normalised.

Parameters

theta: float or 1d numpy array

Returns

logl: float Loglikelihood

phi: list of length nderived Any derived parameters.

__init__(*self*, *a=1*, *b=100*, *nderived=0*)
Define likelihood hyperparameter values.

Parameters

theta: 1d numpy array Parameters.

a: float, optional

b: float, optional

nderived: int, optional Number of derived parameters.

dyPolyChord.python_likelihoods.**log_gaussian_pdf**(*theta*, *sigma=1*, *mu=0*, *ndim=None*)

Log of uncorrelated Gaussian pdf.

Parameters

theta: float or 1d numpy array

sigma: float, optional

mu: float, optional

ndim: int, optional

Returns

logl: float Loglikelihood.

dyPolyChord.python_likelihoods.**log_loggamma_pdf**(*theta*, *alpha=1*, *beta=1*)

Loglikelihood for multidimensional loggamma distribution, with each component of theta having an independent loggamma distribution.

Always returns a float.

Parameters

theta: float or numpy array

alpha: float, optional

beta: float, optional

Returns

logl: float Loglikelihood.

dyPolyChord.python_likelihoods.**log_loggamma_pdf_1d**(*theta*, *alpha=1*, *beta=1*)

1d gamma distribution, with each component of theta independently having PDF:

$$f(x) = \frac{\mathrm{e}^{\beta x}\mathrm{e}^{-\mathrm{e}^{x}/\alpha}}{\alpha^{\beta}\Gamma(\beta)}$$

This function returns $\log f(x)$.

Parameters

theta: float or array Where to evaluate $\log f(x)$. Values $\in (-\inf, \inf)$.

alpha: float, > 0 Scale parameter

beta: float, > 0 Shape parameter

Returns

logl: same type as theta

1.5.2 python_priors

Python priors for use with PolyChord.

PolyChord v1.14 requires priors to be callables with parameter and return types:

Parameters

hypercube: float or 1d numpy array Parameter positions in the prior hypercube.

Returns

theta: float or 1d numpy array Corresponding physical parameter coordinates.

Input hypercube values numpy array is mapped to physical space using the inverse CDF (cumulative distribution function) of each parameter. See the PolyChord papers for more details.

This module use classes with the prior defined in the cube_to_physical function and called with __call__ property. This provides convenient way of storing other information such as hyperparameter values. The objects be used in the same way as functions due to python's "duck typing" (or alternatively you can just define prior functions).

The BlockPrior class allows convenient use of different priors on different parameters.

Inheritance of the BasePrior class allows priors to:

1. have parameters' values sorted to give an enforced order. Useful when the parameter space is symmetric under exchange of variables as this allows the space to be explored to be contracted by a factor of N! (where N is the number of such parameters);

2. adaptively select the number of parameters to use.

You can ignore these if you don't need them.

class dyPolyChord.python_priors.**BasePrior**(*adaptive=False*, *sort=False*, *nfunc_min=1*)

Base class for Priors.

__call__(*self*, *cube*)

Evaluate prior on hypercube coordinates.

Parameters

cube: 1d numpy array Point coordinate on unit hypercube (in probabily space). Note this variable cannot be edited else PolyChord throws an error.

Returns

theta: 1d numpy array Physical parameter values for prior.

__init__(*self*, *adaptive=False*, *sort=False*, *nfunc_min=1*)

Set up prior object's hyperparameter values.

Parameters

adaptive: bool, optional

sort: bool, optional

nfunc_min: int, optional

cube_to_physical(*self*, *cube*)

Map hypercube values to physical parameter values.

Parameters

cube: 1d numpy array Point coordinate on unit hypercube (in probabily space). See the PolyChord papers for more details.

Returns

theta: 1d numpy array Physical parameter values corresponding to hypercube.

class dyPolyChord.python_priors.**BlockPrior**(*prior_blocks*, *block_sizes*)
Prior object which applies a list of priors to different blocks within the parameters.

__call__(*self*, *cube*)
Map hypercube values to physical parameter values.

Parameters

hypercube: 1d numpy array Point coordinate on unit hypercube (in probabily space). See the PolyChord papers for more details.

Returns

theta: 1d numpy array Physical parameter values corresponding to hypercube.

__init__(*self*, *prior_blocks*, *block_sizes*)
Store prior and size of each block.

class dyPolyChord.python_priors.**Exponential**(*lambd=1.0*, ***kwargs*)
Exponential prior.

__init__(*self*, *lambd=1.0*, ***kwargs*)
Set up prior object's hyperparameter values.

Parameters

lambd: float

kwargs: dict, optional See BasePrior.__init__ for more infomation.

cube_to_physical(*self*, *cube*)
Map hypercube values to physical parameter values.

Parameters

cube: 1d numpy array

Returns

theta: 1d numpy array

class dyPolyChord.python_priors.**Gaussian**(*sigma=10.0*, *half=False*, *mu=0.0*, ***kwargs*)
Symmetric Gaussian prior centred on the origin.

__init__(*self*, *sigma=10.0*, *half=False*, *mu=0.0*, ***kwargs*)
Set up prior object's hyperparameter values.

Parameters

sigma: float Standard deviation of Gaussian prior in each parameter.

half: bool, optional Half-Gaussian prior - nonzero only in the region greater than mu. Note that in this case mu is no longer the mean and sigma is no longer the standard deviation of the prior.

mu: float, optional Mean of Gaussian prior.

kwargs: dict, optional See BasePrior.__init__ for more infomation.

cube_to_physical(*self*, *cube*)
Map hypercube values to physical parameter values.

Parameters

cube: 1d numpy array Point coordinate on unit hypercube (in probabily space). See the PolyChord papers for more details.

Returns

theta: 1d numpy array Physical parameter values corresponding to hypercube.

class dyPolyChord.python_priors.**PowerUniform**(*minimum=0.1*, *maximum=2.0*, *power=-2*, ***kwargs*)

Uniform in theta ** power

__init__(*self*, *minimum=0.1*, *maximum=2.0*, *power=-2*, ***kwargs*)

Set up prior object's hyperparameter values.

Prior is uniform in [minimum, maximum] in each parameter.

Parameters

minimum: float

maximum: float

power: float or int

kwargs: dict, optional See BasePrior.__init__ for more infomation.

cube_to_physical(*self*, *cube*)

Map hypercube values to physical parameter values.

Parameters

cube: 1d numpy array

Returns

theta: 1d numpy array

class dyPolyChord.python_priors.**Uniform**(*minimum=0.0*, *maximum=1.0*, ***kwargs*)

Uniform prior.

__init__(*self*, *minimum=0.0*, *maximum=1.0*, ***kwargs*)

Set up prior object's hyperparameter values.

Prior is uniform in [minimum, maximum] in each parameter.

Parameters

minimum: float

maximum: float

kwargs: dict, optional See BasePrior.__init__ for more infomation.

cube_to_physical(*self*, *cube*)

Map hypercube values to physical parameter values.

Parameters

cube: 1d numpy array

Returns

theta: 1d numpy array

dyPolyChord.python_priors.**adaptive_transform**(*cube*, *sort=True*, *nfunc_min=1*)

Tranform first parameter (nfunc) to uniform in (nfunc_min, nfunc_max) and, if required, perform forced identifiability transform on the next nfunc parameters only.

Parameters

cube: 1d numpy array Point coordinate on unit hypercube (in probabily space).

Returns

ad_cube: 1d numpy array First element is physical coordinate of nfunc parameter, other elements are cube coordinates with any forced identifiability transform already applied.

dyPolyChord.python_priors.**forced_identifiability**(*cube*)

Transform hypercube coordinates to enforce identifiability.

For more details see: "PolyChord: next-generation nested sampling" (Handley et al. 2015).

Parameters

cube: 1d numpy array Point coordinate on unit hypercube (in probabily space).

Returns

ordered_cube: 1d numpy array

CHAPTER

TWO

ATTRIBUTION

If you use `dyPolyChord` in your academic research, please cite the two papers introducing the software and the dynamic nested sampling algorithm it uses (the BibTeX is below). Note that `dyPolyChord` runs use `PolyChord`, which also requires its associated papers to be cited.

```
@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}}

@article{higson2018dypolychord,
title={dyPolyChord: dynamic nested sampling with PolyChord},
author={Higson, Edward},
year={2018},
journal={Journal of Open Source Software},
number={29},
pages={916},
volume={3},
doi={10.21105/joss.00965},
url={http://joss.theoj.org/papers/10.21105/joss.00965}}
```

CHAPTER

THREE

CHANGELOG

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

CHAPTER

FOUR

CONTRIBUTIONS

Contributions are welcome! Development takes place on github:

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

d

dyPolyChord.polychord_utils, 10
dyPolyChord.pypolychord_utils, 12
dyPolyChord.python_likelihoods, 13
dyPolyChord.python_priors, 16
dyPolyChord.run_dynamic_ns, 8

INDEX

Symbols

__call__() (*dyPolyChord.polychord_utils.RunCompiledPolyChord method*), 10
__call__() (*dyPolyChord.pypolychord_utils.RunPyPolyChord method*), 12
__call__() (*dyPolyChord.python_likelihoods.Gaussian method*), 13
__call__() (*dyPolyChord.python_likelihoods.GaussianMix method*), 14
__call__() (*dyPolyChord.python_likelihoods.GaussianShell method*), 14
__call__() (*dyPolyChord.python_likelihoods.LogGammaMix method*), 14
__call__() (*dyPolyChord.python_likelihoods.Rastrigin method*), 15
__call__() (*dyPolyChord.python_likelihoods.Rosenbrock method*), 15
__call__() (*dyPolyChord.python_priors.BasePrior method*), 17
__call__() (*dyPolyChord.python_priors.BlockPrior method*), 18
__init__() (*dyPolyChord.polychord_utils.RunCompiledPolyChord method*), 10
__init__() (*dyPolyChord.pypolychord_utils.RunPyPolyChord method*), 12
__init__() (*dyPolyChord.python_likelihoods.Gaussian method*), 13
__init__() (*dyPolyChord.python_likelihoods.GaussianMix method*), 14
__init__() (*dyPolyChord.python_likelihoods.GaussianShell method*), 14
__init__() (*dyPolyChord.python_likelihoods.LogGammaMix method*), 15
__init__() (*dyPolyChord.python_likelihoods.Rastrigin method*), 15
__init__() (*dyPolyChord.python_likelihoods.Rosenbrock method*), 15
__init__() (*dyPolyChord.python_priors.BasePrior method*), 17
__init__() (*dyPolyChord.python_priors.BlockPrior method*), 18
__init__() (*dyPolyChord.python_priors.Exponential method*), 18
__init__() (*dyPolyChord.python_priors.Gaussian method*), 18
__init__() (*dyPolyChord.python_priors.PowerUniform method*), 19
__init__() (*dyPolyChord.python_priors.Uniform method*), 19
__weakref__ (*dyPolyChord.polychord_utils.RunCompiledPolyChord attribute*), 10
__weakref__ (*dyPolyChord.pypolychord_utils.RunPyPolyChord attribute*), 12

A

adaptive_transform() (*in module dyPolyChord.python_priors*), 19

B

BasePrior (*class in dyPolyChord.python_priors*), 17
BlockPrior (*class in dyPolyChord.python_priors*), 17

C

check_settings() (*in module dyPolyChord.run_dynamic_ns*), 9

cube_to_physical() (*dyPolyChord.python_priors.BasePrior method*), 17
cube_to_physical() (*dyPolyChord.python_priors.Exponential method*), 18
cube_to_physical() (*dyPolyChord.python_priors.Gaussian method*), 18
cube_to_physical() (*dyPolyChord.python_priors.PowerUniform method*), 19
cube_to_physical() (*dyPolyChord.python_priors.Uniform method*), 19

D

dyPolyChord.polychord_utils (*module*), 10
dyPolyChord.pypolychord_utils (*module*), 12
dyPolyChord.python_likelihoods (*module*), 13
dyPolyChord.python_priors (*module*), 16
dyPolyChord.run_dynamic_ns (*module*), 8

E

Exponential (*class in dyPolyChord.python_priors*), 18

F

forced_identifiability() (*in module dyPolyChord.python_priors*), 20
format_setting() (*in module dyPolyChord.polychord_utils*), 11

G

Gaussian (*class in dyPolyChord.python_likelihoods*), 13
Gaussian (*class in dyPolyChord.python_priors*), 18
GaussianMix (*class in dyPolyChord.python_likelihoods*), 13
GaussianShell (*class in dyPolyChord.python_likelihoods*), 14
get_prior_block_str() (*in module dyPolyChord.polychord_utils*), 11

I

ini_string() (*dyPolyChord.polychord_utils.RunCompiledPolyChord method*), 10

L

log_gaussian_pdf() (*in module dyPolyChord.python_likelihoods*), 16
log_loggamma_pdf() (*in module dyPolyChord.python_likelihoods*), 16
log_loggamma_pdf_1d() (*in module dyPolyChord.python_likelihoods*), 16
LogGammaMix (*class in dyPolyChord.python_likelihoods*), 14

P

PowerUniform (*class in dyPolyChord.python_priors*), 19
python_block_prior_to_str() (*in module dyPolyChord.polychord_utils*), 11
python_prior_to_str() (*in module dyPolyChord.polychord_utils*), 11

R

Rastrigin (*class in dyPolyChord.python_likelihoods*), 15
Rosenbrock (*class in dyPolyChord.python_likelihoods*), 15
run_dypolychord() (*in module dyPolyChord.run_dynamic_ns*), 8
RunCompiledPolyChord (*class in dyPolyChord.polychord_utils*), 10
RunPyPolyChord (*class in dyPolyChord.pypolychord_utils*), 12

U

Uniform (*class in dyPolyChord.python_priors*), 19