inknest

Release 1.0

Gianluca Becuzzi

Sep 21, 2021

Contents

1 Usage

1.1 Model Definition

To define a model create a class based on `model.Model` .

The `set_parameters()` method must specify the space bounds as a list `[ [inf1,sup1] , [inf2,sup2], .. ]`. If a list of names is specified for certain variables, they can be accessed by name indexing (see `varenv()`). The parameters for which the name is not specified are called automatically `var<n>`. After the model initialization `model.names` will contain all the names.

```
>>> import model
>>> class MyModel(model.Model):
>>>
>>>     def set_parameters(self):
>>>             self.bounds = [[0,1], [0,1], [0,42]]
>>>             self.names  = ['A','mu','sigma']
```

The model can have other attributes. To add them override `__init__(self)` and make sure to call the parent __init__():

```
>>>     def __init__(self,data):
>>>             self.bounds = [[0,1], [0,1], [0,42]]
>>>             self.names  = ['A','mu','sigma']
>>>             self.data   = data
>>>             super().__init__()
```

The logarithm of likelihood and prior have to be specified as class methods:

```
>>> ...
>>>     def log_prior(self, x):
>>>             return
```

The `log_prior()` and `log_likelihood()` methods must be capable to manage (*, space_dim)-shaped arrays and return a (*)-shaped array.

If names are not specified, all operations must be preformed over the last axis.

```
>>>     def log_prior(self,x):
>>>             return -0.5*np.sum(x**2, axis = -1)
```

If names are specified, it is possible to use `model.Model.varenv()`:

```
>>> @model.Model.varenv
>>> def log_prior(self,x):
>>>     return -(x['A'] - self.data[0])**2 - x['mu']**2
```

Finally, to automatically bound a function inside the model domain use the `auto_bound()` decorator:

```
>>> @model.Model.auto_bound
>>> @model.Model.varenv
>>> def log_prior(self,x):
>>>     return -(x['A'] - self.data[0])**2 - x['mu']**2
```

Warning: `varenv` must be the first decorator applied

The data type used in the models is `['position', 'logL', 'logP']`

```
>>> x['position']['A'][time,walker]
>>> x['logL'][time,walker]
```

in case it is necessary to reduce the data structure use `numpy.lib.recfunctions.structured_to_unstructured`.

1.2 Samplers usage

The available samplers are contained in `samplers` module. The first argument is a `model.Model` subclass instance. The second argument is the chain length.

```
>>> import sampler
>>> sampler = sampler.AIESampler(MyModel(), 500 , nwalkers=100)
```

To sample a function, define it as a `log_prior` and use `sample_prior` method of a `Sampler` subclass. After the chain is filled it is accessible as an attribute:

```
>>> x = sampler.chain
```

To join the chains of each particle after removing a `burn_in` use:

```
>>> x = sampler.join_chains(burn_in = 0.3)
```

1.3 Nested Sampling usage

After having defined a model, create an instance of `NestedSampling.NestedSampler` specifying:

1. the model
2. the number of live points
3. the number of sampling steps the live points undergo before getting accepted

Other options are:

- `npoints` stops the computation after having generated a fixed number of points
- `relative_precision`
- `load_old` loads the save of the same run (if it exists). If `filename` is not specified, an *almost* unique code for the run is generated based on the features of the model and the NSampler run
- `filename` to specify a save file
- `evo_progress` to display the progress bar for the evolutin process

The run is performed by `ns.run()`, after that every computed feature is stored as an attribute of the nested sampler:

```
>>> ns = NestedSampling.NestedSampler(model, nlive=1000, evosteps=1000, load_
↪old=False)
>>> ns.run()
>>> print(ns.Z, ns.Z_error, ns.points)
```

1.4 Multiprocess Nested Sampling

It is performed by `mpNestedSampler`. The arguments are the same of `NestedSampler`.

Runs `multiprocessing.cpu_count()` copies of nested sampling, then merges them using the dynamic nested sampling[1] merge algorithm.

After running, the instance contains the merged computed variables (`logX`, `logZ`, ecc.) and the single run variables through `nested_samplers` attribute:

```
>>> mpns = mpNestedSampler(model_, nlive = 500, evosteps = 1200, load_old=False)
>>> mpns.run()
>>> print(f'Z = {mpns.Z} +- {mpns.Z_error})
>>> single_runs = mpns.nested_samplers
>>> for ns in single_runs:
>>>     print(f'Z = {ns.Z} +- {ns.Z_error})
```

[1] https://arxiv.org/abs/1704.03459

2 Generated code description

Code description generated automatically from docstrings.

2.1 Model

class model.**Gaussian**(*dim=1*)

__init__(*dim=1*)
Initialise and checks the model

class model.**Model**(**args*)
Class to describe models

log_prior
the logarithm of the prior pdf

Type function

log_likelihood
the logarithm of the likelihood function

Type function

space_bounds
the coordinate of the two vertices of the hyperrectangle defining the bounds of the parameters

Type 2-tuple of `np.ndarray`

Note: The `log_prior` and `log_likelihood` functions are user defined and must have **one argument only**.

They also must be capable of managing (*,*, .., space_dimension)-shaped arrays, so make sure every operation is done on the **-1 axis of input** or use `varenv()`.

If input is a single point of shape (space_dimension,) both the functions must return a float (not a (1,)-shaped array)

__init__(**args*)
Initialise and checks the model

auto_bound()

Decorator to bound functions.

Parameters **log_func** (function) – A function for which `self.log_func(x)` is valid.

Returns the bounded function `log_func(x) + log_chi(x)`

Return type function

Example

```
>>> class MyModel(model.Model):
>>>
>>>     @model.Model.auto_bound
>>>     def log_prior(x):
>>>         return x
```

is_inside_bounds(*points*)

Checks if a point is inside the space bounds.

Parameters **points** (np.ndarray) – point to be checked. Must have shape (*,space_dim,).

Returns

True if all the coordinates lie between bounds

False if at least one is outside.

Return type np.ndarray

log_chi(*points*)

Logarithm of the characteristic function of the domain. Is equivalent to

```
>>> np.log(model.is_inside_bounds(point).astype(float))
```

Parameters **points** (np.ndarray) – point to be checked. Must have shape (*,space_dim,).

Returns

0 if all the coordinates lie between bounds

-np.inf if at least one is outside

Return type np.ndarray

varenv()

Helper function to index the variables by name inside user-defined functions.

Uses the names defined in the constructor of the model + var0,var1, … for the one which are left unspecified.

Warning: When using with `@auto_bound`, it must be first:

```
>>> @auto_bound
>>> @varenv
>>> def f(self,x):
```

```
>>>      u = x['A']+x['mu']
>>>      ... do stuff
```

class model.**RosenBrock**(**args*)

class model.**UniformJeffreys**(**args*)

2.2 Samplers

Module containing the samplers used in main calculations.

Since almost every sampler is defined by a markov chain, basic attributes are the model and the length of the chain.

Since is intended to be used in nested sampling, each sampler should support likelihood constrained prior sampling (LCPS).

class samplers.**AIESampler**(*model*, *mcmc_length*, *nwalkers=10*, *space_scale=None*, *verbosity=0*)

The Affine-Invariant Ensemble sampler (Goodman, Weare, 2010).

After a uniform initialisation step, for each particle k selects a *pivot* particle an then proposes

$$\begin{aligned} j &= k + random(0 \to n) \\ z &\sim g(z) \\ y &= x_j + z(x_k - x_j) \end{aligned}$$

and then executes a MH-acceptance over y (more information here[2]).

AIEStep(*Lthreshold=None*, *continuous=False*)

Single step of AIESampler.

Parameters

- **Lthreshold** (`float, optional`) – The threshold of likelihood below which a point is set as impossible to reach
- **continuous** (`bool`, optional) – If true use modular index assignment, overwriting past values as `self.elapsed_time_index` > `self.length`

__init__(*model*, *mcmc_length*, *nwalkers=10*, *space_scale=None*, *verbosity=0*)

Initialise the chain uniformly over the space bounds.

get_stretch(*size=1*)

Generates the stretch values given the scale_parameter `a`.

Output is distibuted as $\frac{1}{\sqrt{z}}$ in $[1/a, a]$. Uses inverse transform sampling

join_chains(*burn_in=0.02*)

Joins the chains for the ensemble after removing `burn_in` % of each single_particle chain.

Parameters **burn_in** (`float, optional`) – the burn_in percentage.

Must be `burn_in` > 0 and `burn_in` < 1.

sample_prior(*Lthreshold=None*, *progress=False*)

Fills the chain by sampling the prior.

[2] https://msp.org/camcos/2010/5-1/camcos-v5-n1-p04-p.pdf

class samplers.**AIEevolver**(*model*, *steps*, *length=None*, *nwalkers=10*, *verbosity=0*)
Class to override some functionalities of the sampler in case only the final state is of interest.

The main difference from AIESampler is that `lenght` and `steps` can be different

__init__(*model*, *steps*, *length=None*, *nwalkers=10*, *verbosity=0*)
Initialise the chain uniformly over the space bounds.

bring_over_threshold(*logLthreshold*)
Brings the sampler over threshold.

It is necessary to initialise the sampler before sampling over threshold.

Parameters **Lthreshold** (`float`) – the logarithm of the likelihood.

get_new(*Lmin*, *start_ensemble=None*, *progress=False*, *allow_resize=True*)
Returns `nwalkers` *different* point from prior given likelihood threshold.

As for AIEStep, needs that every point is in a valid region (the border is included).

If the length of the sampler is not enough to ensure that all points are different stretches it doubling `self.steps` each time. The stretch is *permanent*.

Parameters **Lmin** (`float`) – the threshold likelihood that a point must have to be accepted

Returns new generated points

Return type np.ndarray

class samplers.**Sampler**(*model*, *mcmc_length*, *nwalkers*, *verbosity=0*)
Produces samples from model.

It is intended as a base class that has to be further defined. For generality the attribute *nwalkers* is present, but it can be one for not ensamble-based samplers.

model
Model defined as the set of (log_prior, log_likelihood , bounds)

Type *model.Model*

mcmc_lenght
the lenght of the single markov chain

Type int

nwalkers
the number of walkers the ensamble is made of

Type int

__init__(*model*, *mcmc_length*, *nwalkers*, *verbosity=0*)
Initialise the chain uniformly over the space bounds.

2.3 NestedSampling.py

The nested sampling module. In the first (and probably only) version it is mainly tailored onto the AIEsampler class, so there's no choice for the sampler.

class NestedSampling.**NestedSampler**(*model*, *nlive=1000*, *npoints=numpy.inf*, *evosteps=150*, *relative_precision=0.1*, *load_old=None*, *filename=None*, *evo_progress=True*)

Class performing nested sampling

check_prior_sampling(*logL*, *evosteps*, *nsamples*)

Samples the prior over a given likelihood threshold with different steps of evolution.

Can be useful in estimating the steps necessary for convergence to the real distribution.

At first it brings the sample from being uniform to being over threshold, then it evolves sampling over the threshold for `evosteps[i]` times, then takes the last generated points (`sampler.chain[sampler.elapsed_time_index]`) and adds them to the sample to be returned.

Parameters

- **logL** (`float`) – the log of likelihood threshold
- **evosteps** (`int` or array of `int`) – the steps of evolution performed for sampling
- **nsamples** (`int`) – the number of samples taken from `sampler.chain[sampler.elapsed_time_index]`

Returns samples

Return type np.ndarray

estimate_Zerror()

Estimates the error sampling t.

run()

Performs nested sampling.

update()

Updates the value of Z given the current state.

The number of live points is of the form:

`nlive`,(jump) `2nlive-1`, `2nlive-2`, … , `nlive` , (jump) `2nlive-1`, ecc.

Integration is performed between the two successive times at which `N = nlive` (extrema included), then one extremum is excluded when saving to `self.N`.

varenv_points()

Gives usable fields to `self.points['position']` based on `model.names`

NestedSampling.**log_worst_t_among**(*N*)

Helper function to generate shrink factors

Since max({t}) with t in [0,1], len({t}) = N is distributed as Nt**(N-1), the cumulative function is y = t**(N) and sampling uniformly over y gives the desired sample.

Therefore, max({t}) is equivalent to (unif)**(1/N)

and log(unif**(1/N)) = 1/N*log(unif)

class NestedSampling.**mpNestedSampler**(**args*, ***kwargs*)

Multiprocess version of nested sampling.

Runs `multiprocess.cpu_count()` instances of `NestedSampler` and joins them.

logX

Type np.ndarray(dtype=np.float64)

logL

Type np.ndarray(dtype=np.float64)

N

Type np.ndarray(dtype=np.int)

logZ

Type np.float64

Z

Type np.float64

logZ_error

Type np.float64

Z_error

Type np.float64

logZ_samples

Type np.ndarray(dtype=np.float64)

nested_samplers
The individual runs. Each nested sampler has completely defined attributes.

Type list of `NestedSampler`

run_time
The time required to perform the runs and merge them.

Type np.float64

error_estimate_time
The time required to perform error estimate on `logZ`

Type np.float64

how_many_at_given_logL(*N*, *logLs*, *givenlogL*)
Helper function that does what the name says.

See dynamic nested sampling[3].

[3] https://arxiv.org/abs/1704.03459

2.4 Utility routines

utils.**hms**(*secs*)
Returns time in `hour minute seconds` fromat given time in seconds.

utils.**logsubexp**(*x1*, *x2*)
Helper function to execute $\log(e^{x_1} - e^{x_2})$

Parameters

- **x1** (`float`) –
- **x2** (`float`) –

utils.**logsumexp**(*arg*)
Utility to sum over log_values. Given a vector [a1,a2,a3, …] returns $\log(e^{a1} + e^{a2} + ...)$

Parameters **arg** (`np.ndarray`) – the array of values to be log-sum-exponentiated

Returns $\log(e^{a1} + e^{a2} + ...)$

Return type float

Python Module Index

Index

Non-alphabetical

A

B

C

E

G

H

I

J

L

M

N

R

S

U

V

Z