inknest

Release 1.0

Gianluca Becuzzi

Sep 06, 2021

Contents

1 Usage

1.1 Model Definition

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

The `__init__` method must specify the space bounds as a 2-tuple `(inf_bound , sup_bound)`.

After that call the superclass initialisation method to check the model.

```
>>> import model
>>> class MyModel(model.Model):
>>>
>>>     def __init__(self):
>>>             self.bounds = ([0,0], [1,1])
>>>             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.

All operation must be performed onto the last axis. To separate the variables use `unpack_variables()`

```
>>>     def log_prior(self,x):
>>>             x0,x1 = model.unpack_variables(x)
```

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

```
>>>     @model.Model.auto_bound
>>>     def log_prior(self,x):
>>>             x0,x1 = model.unpack_variables(x)
>>>             return -0.5*x0**2
```

1.2 Samplers usage

The available samplers are contained in `samplers` module. The first argument is a `model.Model` user-defined subclass instance.

The second argument is the chain lentgth. Once defined, a sampler has a definite length.

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

To sample a function use the `sample_function` method of a `Sampler` subclass. The function is not necessarily a `Model.log_prior` or a `Model.log_likelihood`, but the sampling bounds are inherited from the model onto which the sampler is instantiated.

```
>>> def log_foo(x):
>>>     ...
>>> sampler.sample_function(log_foo)
```

At this point the sampler fills its chain. For the ensamble samplers the chain has shape `(Niter, Nwalkers, Model.space_dim)`.

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

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

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

2 Generated code description

Code description generated automatically from docstrings.

2.1 Model

class model.**Model**

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 logg_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 `Model.unpack_variables()`.

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

__init__()

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.

The returned array has shape (*,) = `utils.pointshape(point)`

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

The returned array has shape (*,) = `utils.pointshape(point)`

Return type np.ndarray

new_is_inside_bounds(*points*)

Same as `is_inside_bounds`.

Shorter but slower (allegedly due to high processing time of numpy broadcasting).

pointshape(*x*)

`self` shorthand for `utils.pointshape(x, dim = self.space_dim)`

model.**unpack_variables**(*x*)

Helper function that performs values shapecasting.

Given a `np.ndarray` of shape `(n1,n2,--, space_dim)` returns an unpackable array of shape `(space_dim, n1,n2, --)`.

Note: if any of the n1, n2, – other dimension is equal to 1, it gets squeezed as it is an unnecessary nesting. Indeed

- `(x,y,space_dim)` -> `(space_dim,x,y)` is fine
- `(x,y,1)` ~ `(x,y)` -> `(x,y)` (1D case)

Parameters **x** (`np.ndarray`) – the array to be casted

Returns an unpackable array

Return type tuple

Example

It can be used to define models:

```
>>> def log_prior(x):
>>>     x1,x2,x3 = unpack_variables(x)
>>>     return x1/x2*x3
```

Warning: It may be computationally expensive. Check for improvements.

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.

Each sampler shoud be capable of tackling with discontinuous functions.

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=4*, *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

$$j = k + random(0 \to n)$$
$$z \sim g(z)$$
$$y = x_j + z(x_k - x_j)$$

and then executes a MH-acceptance over y (more information at <https://msp.org/camcos/2010/5-1/camcos-v5-n1-p04-p.pdf>).

AIEStep(*log_function*)

Single step of AIESampler

Parameters **log_function** (`function`) –

__init__(*model*, *mcmc_length*, *nwalkers=10*, *space_scale=4*, *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_function(*log_function*)

Samples function.

The real problem for being used in NS is that it is not clear how to treat an Ensemble of particles.

In vanilla NS one has a set of points {x1,—,xn}, chooses the worse, replace with another.

Here the evolution of the single particle itself depends on what others are doing.

One option could be (must be confirmed by theoretical calculations) taking the currentlive points and consider them as the ensemble, then generate a new point like so.

The problem is that this sampler produces `nwalker` particles at a time, which means that the process would be:

- generate nlive from prior (can use this func)
- take worst
- generate OTHER (nlive - 1) points
- pick one of theese at random

which doesn't seem a reasonable way to follow.

Well, i could by the way proceed like this:

- generate nlive from prior (can use this func)
- take worst -> do stuff
- generate a bunch of points (say M)
- take M worst point -> do stuff

but i don't think it is how the vanilla NS should work, because nlive is variable throughout the process. Check dynamic NS.

Returns the chain obtained

Return type np.ndarray

sample_over_threshold(*Lmin*)

Performs likelihood-constrained prior sampling.

The NS algorithm starts with a set $heta_i$ of points distributed as $\pi(heta)$

After excluding the worst (L_w), a new point of likelihood greater than L_w has to be generated. One has freedom to choose the way this new point is yielded, as long as the new point has pdf:

$$p(heta_{new})dheta_{new} = \pi(heta_{new})(L(heta_{new}) > L_w)p(heta_{new})dheta_{new} = 0(L(heta_{new}) > L_w)$$

This function uses the AIE sampler on the current live points (nlive -1) and evolves them in the likelihood-constrained prior to generate other (nlive - 1) points, then takes one at random.

Furthermore, the newly generated point is forced to have likelihood different from all the initial ones.

Note: (at the moment) sampler has to be initialised to points already inside bounds

To solve: conflict nlive -> nlive - 1

---

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

2.4 Utility routines

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

utils.**pointshape**(*x*, *dim=None*)

Gives the shape of an array of points of dimension space_dim. Basically pops the last item of x.shape and checks whether it's fine.

Parameters

- **x** (np.ndarray) –
- **dim** (int, optional) – the space dimension

Returns the shape of x considering last axis made of ()-shaped items.

Return type tuple

Python Module Index