inknest

Release 1.0

Gianluca Becuzzi

Sep 14, 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__``()`

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**

`__init__()`
Initialise and checks the model

`class model.ToyGaussian(dim=1)`

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

`class model.UniformJeffreys`

`__init__()`
Initialise and checks the model

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=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

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

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(Lthreshold=None)`
Single step of AIESampler.

Parameters **Lthreshold** (`float, optional`) – The threshold of likelihood below which a point is set as impossible to reach

`__init__(model, mcmc_length, nwalkers=10, space_scale=None, 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)`
Returns a new different point from prior given likelihood threshold

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

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

Returns (new , correct) one of the evolved points and all the generated points

Return type tuple

`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.

`tail_to_head`()
Helper function for doing continuous sampling.

Sets the end of the chain as the head and restarts elapsed time.

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 $\log\left(e^{x_1} - e^{x_2}\right)$

Parameters

- **`x1`** (`float`) –

- **x2** (float) –

`utils.`**`logsumexp`**`(`*arg*`)`

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

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

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

Return type float

Python Module Index