littlemcmc

Release 0.2.2

Apr 22, 2021

Contents

CHAPTER 1

Installation

Note: LittleMCMC is developed for Python 3.6 or later.

LittleMCMC is a pure Python library, so it can be easily installed by using `pip` or directly from source.

1.1 Using `pip`

LittleMCMC can be installed using pip.

```
pip install littlemcmc
```

1.2 From Source

The source code for LittleMCMC can be downloaded from GitHub by running

```
git clone https://github.com/eigenfoo/littlemcmc.git
cd littlemcmc/
python setup.py install
```

1.3 Testing

To run the unit tests, install pytest and then, in the root of the project directory, execute:

```
pytest -v
```

All of the tests should pass. If any of the tests don't pass and you can't figure out why, please open an issue on GitHub.

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

CHAPTER 2

LittleMCMC Quickstart

LittleMCMC is a lightweight and performant implementation of HMC and NUTS in Python, spun out of the PyMC project. In this quickstart tutorial, we will walk through the main use case of LittleMCMC, and outline the various modules that may be of interest.

2.1 Table of Contents

2.2 Who should use LittleMCMC?

LittleMCMC is a fairly barebones library with a very niche use case. Most users will probably find that PyMC3 will satisfy their needs, with better strength of support and quality of documentation.

There are two expected use cases for LittleMCMC. Firstly, if you:

1. Have a model with only continuous parameters,
2. Are willing to manually transform all of your model's parameters to the unconstrained space (if necessary),
3. Have a Python function/callable that:
   1. computes the log probability of your model and its derivative
   2. is pickleable
   3. outputs an array with the same shape as its input

4. And all you need is an implementation of HMC/NUTS (preferably in Python) to sample from the posterior,

then you should consider using LittleMCMC.

Secondly, if you want to run algorithmic experiments on HMC/NUTS (in Python), without having to develop around the heavy machinery that accompanies other probabilistic programming frameworks (like PyMC3, TensorFlow Probability or Stan), then you should consider running your experiments in LittleMCMC.

2.3 How to Sample

```
import numpy as np
import scipy
import littlemcmc as lmc
```

```
def logp_func(x, loc=0, scale=1):
    return np.log(scipy.stats.norm.pdf(x, loc=loc, scale=scale))


def dlogp_func(x, loc=0, scale=1):
    return -(x - loc) / scale


def logp_dlogp_func(x, loc=0, scale=1):
    return logp_func(x, loc=loc, scale=scale), dlogp_func(x, loc=loc, scale=scale)
```

```
# By default: 4 chains in 4 cores, 500 tuning steps and 1000 sampling steps.
trace, stats = lmc.sample(
    logp_dlogp_func=logp_dlogp_func,
    model_ndim=1,
    progressbar=None,  # HTML progress bars don't render well in RST.
)
```

```
/home/george/littlemcmc/venv/lib/python3.6/site-packages/ipykernel_launcher.py:2:␣
↪RuntimeWarning: divide by zero encountered in log
```

2.3.1 Inspecting the Output of `lmc.sample`

```
# Shape is (num_chains, num_samples, num_parameters)
trace.shape
```

```
(4, 1000, 1)
```

```
# The first 2 samples across all chains and parameters
trace[:, :2, :]
```

```
array([[[ 0.92958231],
        [ 0.92958231]],

       [[-1.06231693],
        [-1.11589309]],

       [[-0.73177109],
```

(continues on next page)

(continued from previous page)

```
        [-0.66975061]],

       [[ 0.8923907 ],
        [ 0.97253646]]])
```

```
stats.keys()
```

```
dict_keys(['depth', 'step_size', 'tune', 'mean_tree_accept', 'step_size_bar', 'tree_
↪size', 'diverging', 'energy_error', 'energy', 'max_energy_error', 'model_logp'])
```

```
# Again, shape is (num_chains, num_samples, num_parameters)
stats["depth"].shape
```

```
(4, 1000, 1)
```

```
# The first 2 tree depths across all chains and parameters
stats["depth"][:, :2, :]
```

```
array([[[2],
        [1]],

       [[1],
        [1]],

       [[2],
        [1]],

       [[2],
        [1]]])
```

2.4 Customizing the Default NUTS Sampler

By default, `lmc.sample` samples using NUTS with sane defaults. These defaults can be override by either:

1. Passing keyword arguments from `lmc.NUTS` into `lmc.sample`, or
2. Constructing an `lmc.NUTS` sampler, and passing that to `lmc.sample`. This method also allows you to choose to other samplers, such as `lmc.HamiltonianMC`.

For example, suppose you want to increase the `target_accept` from the default `0.8` to `0.9`. The following two cells are equivalent:

```
trace, stats = lmc.sample(
    logp_dlogp_func=logp_dlogp_func,
    model_ndim=1,
    target_accept=0.9,
    progressbar=None,
)
```

```
step = lmc.NUTS(logp_dlogp_func=logp_dlogp_func, model_ndim=1, target_accept=0.9)
trace, stats = lmc.sample(
    logp_dlogp_func=logp_dlogp_func,
```

(continues on next page)

(continued from previous page)

```
    model_ndim=1,
    step=step,
    progressbar=None,
)
```

```
/home/george/littlemcmc/venv/lib/python3.6/site-packages/ipykernel_launcher.py:2:␣
↪RuntimeWarning: divide by zero encountered in log
```

For a list of keyword arguments that `lmc.NUTS` accepts, please refer to the *API reference for ‘‘lmc.NUTS‘* <https://littlemcmc.readthedocs.io/en/latest/generated/littlemcmc.NUTS.html#littlemcmc.NUTS>‘__.

2.5 Other Modules

LittleMCMC exposes:

1. Two step methods (a.k.a. samplers): `littlemcmc.HamiltonianMC (Hamiltonian Monte Carlo) <https://littlemcmc.readthedocs.io/en/latest/generated/littlemcmc.HamiltonianMC.html#littlemcmc.HamiltonianMC>‘__ and the `littlemcmc.NUTS (No-U-Turn Sampler) <https://littlemcmc.readthedocs.io/en/latest/generated/littlemcmc.NUTS.html#littlemcmc.NUTS>‘__
2. Various quadpotentials (a.k.a. mass matrices or inverse metrics) in `littlemcmc.quadpotential <https://littlemcmc.readthedocs.io/en/latest/api.html#quadpotentials-a-k-a-mass-matrices>‘__, along with mass matrix adaptation routines
3. Dual-averaging step size adaptation in `littlemcmc.step_sizes <https://littlemcmc.readthedocs.io/en/latest/generated/littlemcmc.step_sizes.DualAverageAdaptation.html#littlemcmc.step_sizes.DualAverageAdaptation>‘__
4. A leapfrog integrator in `littlemcmc.integration <https://littlemcmc.readthedocs.io/en/latest/generated/littlemcmc.integration.CpuLeapfrogIntegrator.html#littlemcmc.integration.CpuLeapfrogIntegrator>‘__

These modules should allow for easy experimentation with the sampler. Please refer to the API Reference for more information.

Note: This tutorial was generated from an IPython notebook that can be downloaded here.

CHAPTER 3

Framework Cookbook

`littlemcmc` only needs a `logp_dlogp_func`, which is framework-agnostic. To illustrate this, this cookbook implements linear in multiple frameworks, and samples them with `littlemcmc`. At the end of this notebook, we load the inference traces and sampler statistics into ArviZ and do some basic visualizations.

```
import littlemcmc as lmc
```

3.1 Create and Visualize Data

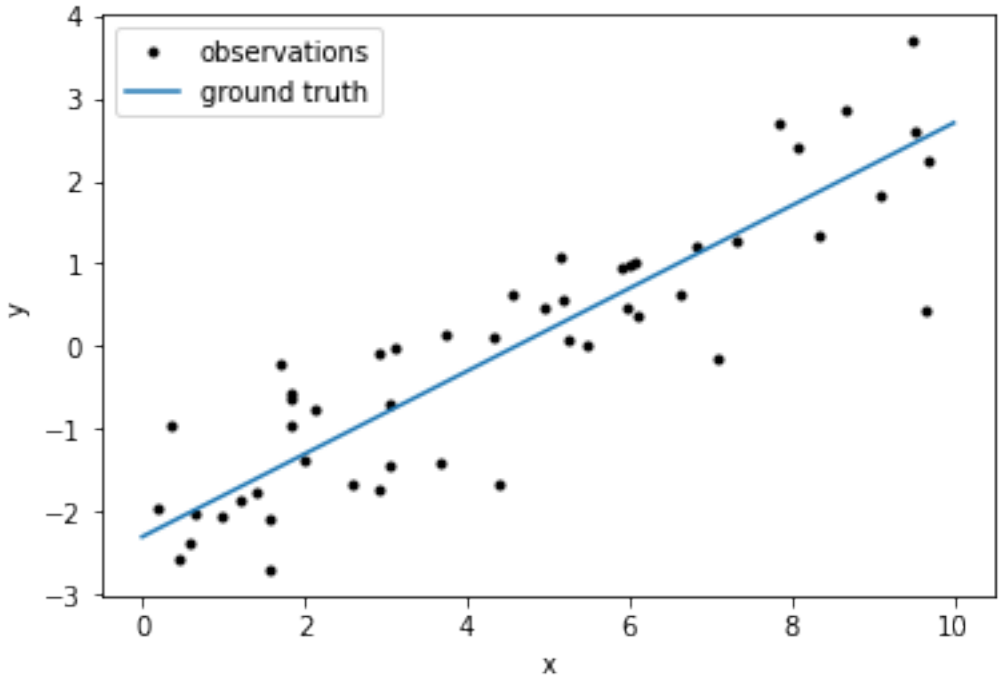
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

true_params = np.array([0.5, -2.3, -0.23])

N = 50
t = np.linspace(0, 10, 2)
x = np.random.uniform(0, 10, 50)
y = x * true_params[0] + true_params[1]
y_obs = y + np.exp(true_params[-1]) * np.random.randn(N)

plt.plot(x, y_obs, ".k", label="observations")
plt.plot(t, true_params[0]*t + true_params[1], label="ground truth")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```



3.2 PyTorch

```
import torch

class LinearModel(torch.nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.m = torch.nn.Parameter(torch.tensor(0.0, dtype=torch.float64))
        self.b = torch.nn.Parameter(torch.tensor(0.0, dtype=torch.float64))
        self.logs = torch.nn.Parameter(torch.tensor(0.0, dtype=torch.float64))

    def forward(self, x, y):
        mean = self.m * x + self.b
        loglike = -0.5 * torch.sum((y - mean) ** 2 * torch.exp(-2 * self.logs) + 2 *␣
↪self.logs)
        return loglike

torch_model = torch.jit.script(LinearModel())
torch_params = [torch_model.m, torch_model.b, torch_model.logs]
args = [torch.tensor(x, dtype=torch.double), torch.tensor(y_obs, dtype=torch.double)]

def torch_logp_dlogp_func(x):
    for i, p in enumerate(torch_params):
        p.data = torch.tensor(x[i])
        if p.grad is not None:
            p.grad.detach_()
            p.grad.zero_()

    result = torch_model(*args)
    result.backward()
```

(continues on next page)

(continued from previous page)

```
    return result.detach().numpy(), np.array([p.grad.numpy() for p in torch_params])
```

```
298 µs ± 43.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Please see `sample_pytorch_logp_dlogp_func.py <https://github.com/eigenfoo/littlemcmc/tree/master/docs/_static/scripts/sample_pytorch_logp_dlogp_func.py>`__ for a working example. Theoretically, however, all that's needed is to run the following snippet:

```
trace, stats = lmc.sample(
    logp_dlogp_func=torch_logp_dlogp_func, model_ndim=3, tune=500, draws=1000,␣
↪chains=4,
)
```

3.3 JAX

```
from jax.config import config
config.update("jax_enable_x64", True)

import jax
import jax.numpy as jnp

def jax_model(params):
    mean = params[0] * x + params[1]
    loglike = -0.5 * jnp.sum((y_obs - mean) ** 2 * jnp.exp(-2 * params[2]) + 2 *␣
↪params[2])
    return loglike

@jax.jit
def jax_model_and_grad(x):
    return jax_model(x), jax.grad(jax_model)(x)


def jax_logp_dlogp_func(x):
    v, g = jax_model_and_grad(x)
    return np.asarray(v), np.asarray(g)
```

```
/Users/george/miniconda3/lib/python3.7/site-packages/jax/lib/xla_bridge.py:125:␣
↪UserWarning: No GPU/TPU found, falling back to CPU.
  warnings.warn('No GPU/TPU found, falling back to CPU.')
```

```
269 µs ± 48.6 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Please see `sample_jax_logp_dlogp_func.py <https://github.com/eigenfoo/littlemcmc/tree/master/docs/_static/scripts/sample_jax_logp_dlogp_func.py>`__ for a working example. Theoretically, however, all that's needed is to run the following snippet:

```
trace, stats = lmc.sample(
    logp_dlogp_func=jax_logp_dlogp_func, model_ndim=3, tune=500, draws=1000, chains=4,
)
```

3.4 PyMC3

```
import pymc3 as pm
import theano

with pm.Model() as pm_model:
    pm_params = pm.Flat("pm_params", shape=3)
    mean = pm_params[0] * x + pm_params[1]
    pm.Normal("obs", mu=mean, sigma=pm.math.exp(pm_params[2]), observed=y_obs)

pm_model_and_grad = pm_model.fastfn([pm_model.logpt] + theano.grad(pm_model.logpt, pm_
↪model.vars))

def pm_logp_dlogp_func(x):
    return pm_model_and_grad(pm_model.bijection.rmap(x))
```

```
46.3 µs ± 3.94 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
trace, stats = lmc.sample(
    logp_dlogp_func=pm_logp_dlogp_func,
    model_ndim=3,
    tune=500,
    draws=1000,
    chains=4,
    progressbar=False,  # Progress bars don't render well in reStructuredText docs...
)
```

3.5 Visualize Traces with ArviZ

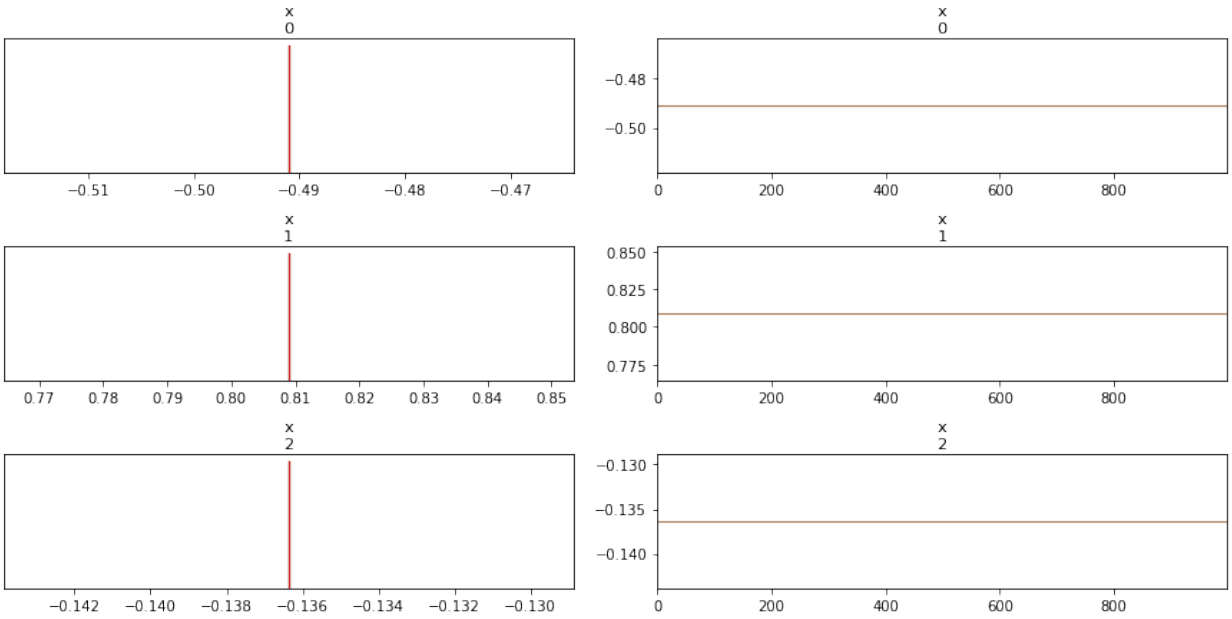
Just to sanity check our results, let's visualize all the traces using ArviZ. At the time of writing there's no way to easily load the `np.ndarrays` arrays that `littlemcmc` returns into an `az.InferenceDataset`. Hopefully one day we'll have an `az.from_littlemcmc` method... but until then, please use this code snippet!

```
def arviz_from_littlemcmc(trace, stats):
    return az.InferenceData(
        posterior=az.dict_to_dataset({"x": trace}),
        sample_stats=az.dict_to_dataset({k: v.squeeze() for k, v in stats.items()})
    )
```

```
import arviz as az

dataset = arviz_from_littlemcmc(trace, stats)

az.plot_trace(dataset)
plt.show()
```


x
0
−0.51
−0.50
−0.49
−0.48
−0.47
x
0
−0.48
−0.50
0
200
400
600
800
x
1
0.77
0.78
0.79
0.80
0.81
0.82
0.83
0.84
0.85
x
1
0.850
0.825
0.800
0.775
0
200
400
600
800
x
2
−0.142
−0.140
−0.138
−0.136
−0.134
−0.132
−0.130
x
2
−0.130
−0.135
−0.140
0
200
400
600
800

CHAPTER 4

API Reference

4.1 Sampling

sample(logp_dlogp_func, Tuple[numpy.ndarray, ...)	Draw samples from the posterior using the given step methods.

4.1.1 littlemcmc.sample

littlemcmc.**sample**(*logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], model_ndim: int, draws: int = 1000, tune: int = 1000, step: Union[littlemcmc.nuts.NUTS, littlemcmc.hmc.HamiltonianMC] = None, init: str = 'auto', chains: Optional[int] = None, cores: Optional[int] = None, start: Optional[numpy.ndarray] = None, progressbar: Union[bool, str] = True, random_seed: Union[int, List[int], None] = None, discard_tuned_samples: bool = True, chain_idx: int = 0, callback=None, mp_ctx=None, pickle_backend: str = 'pickle', **kwargs*)

Draw samples from the posterior using the given step methods.

Parameters

logp_dlogp_func: Python callable Python callable that returns a tuple of the model joint log probability and its derivative, in that order.

model_ndim: int The number of parameters of the model.

draws: int The number of samples to draw. Defaults to 1000. The number of tuned samples are discarded by default. See `discard_tuned_samples`.

tune: int Number of iterations to tune, defaults to 1000. Samplers adjust the step sizes, scalings or similar during tuning. Tuning samples will be drawn in addition to the number specified in the `draws` argument, and will be discarded unless `discard_tuned_samples` is set to False.

step: function A step function. By default the NUTS step method will be used.

init: str

Initialization method to use for auto-assigned NUTS samplers.

- auto: Choose a default initialization method automatically. Currently, this is `jitter+adapt_diag`, but this can change in the future. If you depend on the exact behaviour, choose an initialization method explicitly.
- adapt_diag: Start with a identity mass matrix and then adapt a diagonal based on the variance of the tuning samples.
- jitter+adapt_diag: Same as `adapt_diag`, but add uniform jitter in [-1, 1] to the starting point in each chain.
- adapt_full: Same as *'adapt_diag'*, but adapt a dense mass matrix using the sample covariances.

chains: int The number of chains to sample. Running independent chains is important for some convergence statistics and can also reveal multiple modes in the posterior. If `None`, then set to either `cores` or 2, whichever is larger.

cores: int The number of chains to run in parallel. If `None`, set to the number of CPUs in the system, but at most 4.

start: dict, or array of dict Starting point in parameter space. Initialization methods for NUTS (see `init` keyword) can overwrite the default.

progressbar: bool, optional default=True Whether or not to display a progress bar in the command line. The bar shows the percentage of completion, the sampling speed in samples per second (SPS), and the estimated remaining time until completion ("expected time of arrival"; ETA).

random_seed: int or list of ints A list is accepted if `cores` is greater than one.

discard_tuned_samples: bool Whether to discard posterior samples of the tune interval.

Returns

trace: np.array An array that contains the samples.

stats: dict A dictionary that contains sampler statistics.

Notes

Optional keyword arguments can be passed to `sample` to be delivered to the ‘‘step_method‘‘s used during sampling. In particular, the NUTS step method accepts a number of arguments. You can find a full list of arguments in the docstring of the step methods. Common options are:

- target_accept: float in [0, 1]. The step size is tuned such that we approximate this acceptance rate. Higher values like 0.9 or 0.95 often work better for problematic posteriors.
- max_treedepth: The maximum depth of the trajectory tree.
- step_scale: float, default 0.25. The initial guess for the step size scaled

down by $1/n * *(1/4)$.

4.2 Step Methods

HamiltonianMC(logp_dlogp_func, ... [, potential])	A sampler for continuous variables based on Hamiltonian mechanics.
NUTS(logp_dlogp_func, Tuple[numpy.ndarray, ...)	A sampler for continuous variables based on Hamiltonian mechanics.

4.2.1 littlemcmc.HamiltonianMC

class littlemcmc.**HamiltonianMC**(*logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], model_ndim: int, scaling: Optional[numpy.ndarray] = None, is_cov: bool = False, potential=None, target_accept: float = 0.8, Emax: float = 1000, adapt_step_size: bool = True, step_scale: float = 0.25, gamma: float = 0.05, k: float = 0.75, t0: int = 10, step_rand: Optional[Callable[[float], float]] = None, path_length: float = 2.0, max_steps: int = 1024*)

A sampler for continuous variables based on Hamiltonian mechanics.

See NUTS sampler for automatically tuned stopping time and step size scaling.

__init__(*logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], model_ndim: int, scaling: Optional[numpy.ndarray] = None, is_cov: bool = False, potential=None, target_accept: float = 0.8, Emax: float = 1000, adapt_step_size: bool = True, step_scale: float = 0.25, gamma: float = 0.05, k: float = 0.75, t0: int = 10, step_rand: Optional[Callable[[float], float]] = None, path_length: float = 2.0, max_steps: int = 1024*)

Set up the Hamiltonian Monte Carlo sampler.

Parameters

logp_dlogp_func [Python callable] Python callable that returns the log-probability and derivative of the log-probability, respectively.

model_ndim [int] Total number of parameters. Dimensionality of the output of `logp_dlogp_func`.

scaling [1 or 2-dimensional array-like] Scaling for momentum distribution. 1 dimensional arrays are interpreted as a matrix diagonal.

is_cov [bool] Treat scaling as a covariance matrix/vector if True, else treat it as a precision matrix/vector

potential [littlemcmc.quadpotential.Potential, optional] An object that represents the Hamiltonian with methods `velocity`, `energy`, and `random` methods. Only one of `scaling` or `potential` may be non-None.

target_accept [float] Adapt the step size such that the average acceptance probability across the trajectories are close to target_accept. Higher values for target_accept lead to smaller step sizes. Setting this to higher values like 0.9 or 0.99 can help with sampling from difficult posteriors. Valid values are between 0 and 1 (exclusive).

Emax [float] The maximum allowable change in the value of the Hamiltonian. Any trajectories that result in changes in the value of the Hamiltonian larger than `Emax` will be declared divergent.

adapt_step_size [bool, default=True] If True, performs dual averaging step size adaptation. If False, `k`, `t0`, `gamma` and `target_accept` are ignored.

step_scale [float] Size of steps to take, automatically scaled down by 1 / (`size` ** 0.25).

gamma [float, default .05]

k [float, default .75] Parameter for dual averaging for step size adaptation. Values between 0.5 and 1 (exclusive) are admissible. Higher values correspond to slower adaptation.

t0 [int, default 10] Parameter for dual averaging. Higher values slow initial adaptation.

step_rand [Python callable] Callback for step size adaptation. Called on the step size at each iteration immediately before performing dual-averaging step size adaptation.

path_length [float, default=2] total length to travel

max_steps [int, default=1024] The maximum number of leapfrog steps.

Methods

`__init__`(logp_dlogp_func, ... [, potential])	Set up the Hamiltonian Monte Carlo sampler.
`reset`(start)	Reset quadpotential and begin retuning.
`reset_tuning`(start)	Reset quadpotential and step size adaptation, and begin retuning.
`stop_tuning`()	Stop tuning.
`warnings`()	Generate warnings from HMC sampler.

Attributes

`generates_stats`
`name`
`stats_dtypes`

4.2.2 littlemcmc.NUTS

class littlemcmc.**NUTS**(*logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], model_ndim: int, scaling: Optional[numpy.ndarray] = None, is_cov: bool = False, potential=None, target_accept: float = 0.8, Emax: float = 1000, adapt_step_size: bool = True, step_scale: float = 0.25, gamma: float = 0.05, k: float = 0.75, t0: int = 10, step_rand: Optional[Callable[[float], float]] = None, path_length: float = 2.0, max_treedepth: int = 10, early_max_treedepth: int = 8*)

A sampler for continuous variables based on Hamiltonian mechanics.

NUTS automatically tunes the step size and the number of steps per sample. A detailed description can be found at [1], "Algorithm 6: Efficient No-U-Turn Sampler with Dual Averaging".

NUTS provides a number of statistics that can be accessed with *trace.get_sampler_stats*:

- *mean_tree_accept*: The mean acceptance probability for the tree that generated this sample. The mean of these values across all samples but the burn-in should be approximately *target_accept* (the default for this is 0.8).
- *diverging*: Whether the trajectory for this sample diverged. If there are any divergences after burnin, this indicates that the results might not be reliable. Reparametrization can often help, but you can also try to increase *target_accept* to something like 0.9 or 0.95.
- *energy*: The energy at the point in phase-space where the sample was accepted. This can be used to identify posteriors with problematically long tails. See below for an example.
- *energy_change*: The difference in energy between the start and the end of the trajectory. For a perfect integrator this would always be zero.

- *max_energy_change*: The maximum difference in energy along the whole trajectory.
- *depth*: The depth of the tree that was used to generate this sample
- *tree_size*: The number of leafs of the sampling tree, when the sample was accepted. This is usually a bit less than *2 ** depth*. If the tree size is large, the sampler is using a lot of leapfrog steps to find the next sample. This can for example happen if there are strong correlations in the posterior, if the posterior has long tails, if there are regions of high curvature ("funnels"), or if the variance estimates in the mass matrix are inaccurate. Reparametrisation of the model or estimating the posterior variances from past samples might help.
- *tune*: This is *True*, if step size adaptation was turned on when this sample was generated.
- *step_size*: The step size used for this sample.
- *step_size_bar*: The current best known step-size. After the tuning samples, the step size is set to this value. This should converge during tuning.
- *model_logp*: The model log-likelihood for this sample.

References

[1]

__init__(*logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]], model_ndim: int, scaling: Optional[numpy.ndarray] = None, is_cov: bool = False, potential=None, target_accept: float = 0.8, Emax: float = 1000, adapt_step_size: bool = True, step_scale: float = 0.25, gamma: float = 0.05, k: float = 0.75, t0: int = 10, step_rand: Optional[Callable[[float], float]] = None, path_length: float = 2.0, max_treedepth: int = 10, early_max_treedepth: int = 8*)

Set up the No-U-Turn sampler.

Parameters

logp_dlogp_func [Python callable] Python callable that returns the log-probability and derivative of the log-probability, respectively.

model_ndim [int] Total number of parameters. Dimensionality of the output of `logp_dlogp_func`.

scaling [1 or 2-dimensional array-like] Scaling for momentum distribution. 1 dimensional arrays are interpreted as a matrix diagonal.

is_cov [bool] Treat scaling as a covariance matrix/vector if True, else treat it as a precision matrix/vector

potential [littlemcmc.quadpotential.Potential, optional] An object that represents the Hamiltonian with methods `velocity`, `energy`, and `random` methods. Only one of `scaling` or `potential` may be non-None.

target_accept [float] Adapt the step size such that the average acceptance probability across the trajectories are close to target_accept. Higher values for target_accept lead to smaller step sizes. Setting this to higher values like 0.9 or 0.99 can help with sampling from difficult posteriors. Valid values are between 0 and 1 (exclusive).

Emax [float] The maximum allowable change in the value of the Hamiltonian. Any trajectories that result in changes in the value of the Hamiltonian larger than `Emax` will be declared divergent.

adapt_step_size [bool, default=True] If True, performs dual averaging step size adaptation. If False, `k`, `t0`, `gamma` and `target_accept` are ignored.

step_scale [float] Size of steps to take, automatically scaled down by 1 / (`size` ** 0.25).

gamma [float, default .05]

k [float, default .75] Parameter for dual averaging for step size adaptation. Values between 0.5 and 1 (exclusive) are admissible. Higher values correspond to slower adaptation.

t0 [int, default 10] Parameter for dual averaging. Higher values slow initial adaptation.

step_rand [Python callable] Callback for step size adaptation. Called on the step size at each iteration immediately before performing dual-averaging step size adaptation.

path_length [float, default=2] total length to travel

max_treedepth [int, default=10] The maximum tree depth. Trajectories are stoped when this depth is reached.

early_max_treedepth [int, default=8] The maximum tree depth during the first 200 tuning samples.

Notes

The step size adaptation stops when *self.tune* is set to False. This is usually achieved by setting the *tune* parameter if *pm.sample* to the desired number of tuning steps.

Methods

__init__(logp_dlogp_func, ... [, potential])	Set up the No-U-Turn sampler.
`reset`(start)	Reset quadpotential and begin retuning.
`reset_tuning`(start)	Reset quadpotential and step size adaptation, and begin retuning.
`stop_tuning`()	Stop tuning.
`warnings`()	Generate warnings from NUTS sampler.

Attributes

`default_blocked`
`generates_stats`
`name`
`stats_dtypes`

4.3 Quadpotentials (a.k.a. Mass Matrices)

quad_potential(C, is_cov)	Compute a QuadPotential object from a scaling matrix.
QuadPotentialDiag(v[, dtype])	Quad potential using a diagonal covariance matrix.
QuadPotentialFull(cov[, dtype])	Basic QuadPotential object for Hamiltonian calculations.
QuadPotentialFullInv(A[, dtype])	QuadPotential object for Hamiltonian calculations using inverse of covariance matrix.
QuadPotentialDiagAdapt(n, initial_mean[, ...])	Adapt a diagonal mass matrix from the sample variances.

Continued on next page

Table 7 – continued from previous page

QuadPotentialFullAdapt(n, initial_mean[, ...])	Adapt a dense mass matrix using the sample covariances.

4.3.1 littlemcmc.quad_potential

littlemcmc.**quad_potential**(*C*, *is_cov*)

Compute a QuadPotential object from a scaling matrix.

Parameters

C [arraylike, 0 <= ndim <= 2] scaling matrix for the potential vector treated as diagonal matrix.

is_cov [Boolean] whether C is provided as a covariance matrix or hessian

Returns

q [Quadpotential]

4.3.2 littlemcmc.QuadPotentialDiag

class littlemcmc.**QuadPotentialDiag**(*v*, *dtype=None*)

Quad potential using a diagonal covariance matrix.

__init__(*v*, *dtype=None*)

Use a vector to represent a diagonal matrix for a covariance matrix.

Parameters

v [vector, 0 <= ndim <= 1] Diagonal of covariance matrix for the potential vector

Methods

__init__(v[, dtype])	Use a vector to represent a diagonal matrix for a covariance matrix.
energy(x[, velocity])	Compute kinetic energy at a position in parameter space.
raise_ok([vmap])	Check if the mass matrix is ok, and raise ValueError if not.
random()	Draw random value from QuadPotential.
reset()	Reset quadpotential adaptation routine.
update(sample, grad, tune)	Inform the potential about a new sample during tuning.
velocity(x[, out])	Compute the current velocity at a position in parameter space.
velocity_energy(x, v_out)	Compute velocity and return kinetic energy at a position in parameter space.

4.3.3 littlemcmc.QuadPotentialFull

class littlemcmc.**QuadPotentialFull**(*cov*, *dtype=None*)

Basic QuadPotential object for Hamiltonian calculations.

__init__(*cov*, *dtype=None*)

Compute the lower cholesky decomposition of the potential.

Parameters

A [matrix, ndim = 2] scaling matrix for the potential vector

Methods

`__init__`(cov[, dtype])	Compute the lower cholesky decomposition of the potential.
`energy`(x[, velocity])	Compute kinetic energy at a position in parameter space.
`raise_ok`([vmap])	Check if the mass matrix is ok, and raise ValueError if not.
`random`()	Draw random value from QuadPotential.
`reset`()	Reset quadpotential adaptation routine.
`update`(sample, grad, tune)	Inform the potential about a new sample during tuning.
`velocity`(x[, out])	Compute the current velocity at a position in parameter space.
`velocity_energy`(x, v_out)	Compute velocity and return kinetic energy at a position in parameter space.

4.3.4 littlemcmc.QuadPotentialFullInv

class littlemcmc.**QuadPotentialFullInv**(*A*, *dtype=None*)

QuadPotential object for Hamiltonian calculations using inverse of covariance matrix.

__init__(*A*, *dtype=None*)

Compute the lower cholesky decomposition of the potential.

Parameters

A [matrix, ndim = 2] Inverse of covariance matrix for the potential vector

Methods

`__init__`(A[, dtype])	Compute the lower cholesky decomposition of the potential.
`energy`(x[, velocity])	Compute kinetic energy at a position in parameter space.
`raise_ok`([vmap])	Check if the mass matrix is ok, and raise ValueError if not.
`random`()	Draw random value from QuadPotential.
`reset`()	Reset quadpotential adaptation routine.
`update`(sample, grad, tune)	Inform the potential about a new sample during tuning.
`velocity`(x[, out])	Compute the current velocity at a position in parameter space.
`velocity_energy`(x, v_out)	Compute velocity and return kinetic energy at a position in parameter space.

4.3.5 littlemcmc.QuadPotentialDiagAdapt

class littlemcmc.**QuadPotentialDiagAdapt**(*n*, *initial_mean*, *initial_diag=None*, *initial_weight=0*, *adaptation_window=101*, *adaptation_window_multiplier=1*, *dtype=None*)

Adapt a diagonal mass matrix from the sample variances.

__init__(*n*, *initial_mean*, *initial_diag=None*, *initial_weight=0*, *adaptation_window=101*, *adaptation_window_multiplier=1*, *dtype=None*)

Set up a diagonal mass matrix.

Methods

__init__(n, initial_mean[, initial_diag, ...])	Set up a diagonal mass matrix.
energy(x[, velocity])	Compute kinetic energy at a position in parameter space.
raise_ok(vmap)	Check if the mass matrix is ok, and raise ValueError if not.
random()	Draw random value from QuadPotential.
reset()	Reset quadpotential adaptation routine.
update(sample, grad, tune)	Inform the potential about a new sample during tuning.
velocity(x[, out])	Compute the current velocity at a position in parameter space.
velocity_energy(x, v_out)	Compute velocity and return kinetic energy at a position in parameter space.

4.3.6 littlemcmc.QuadPotentialFullAdapt

class littlemcmc.**QuadPotentialFullAdapt**(*n*, *initial_mean*, *initial_cov=None*, *initial_weight=0*, *adaptation_window=101*, *adaptation_window_multiplier=2*, *update_window=1*, *dtype=None*)

Adapt a dense mass matrix using the sample covariances.

__init__(*n*, *initial_mean*, *initial_cov=None*, *initial_weight=0*, *adaptation_window=101*, *adaptation_window_multiplier=2*, *update_window=1*, *dtype=None*)

Compute the lower cholesky decomposition of the potential.

Parameters

A [matrix, ndim = 2] scaling matrix for the potential vector

Methods

__init__(n, initial_mean[, initial_cov, ...])	Compute the lower cholesky decomposition of the potential.
energy(x[, velocity])	Compute kinetic energy at a position in parameter space.
raise_ok(vmap)	Check if the mass matrix is ok, and raise ValueError if not.
random()	Draw random value from QuadPotential.

Continued on next page

Table 12 – continued from previous page

reset()	Reset quadpotential adaptation routine.
update(sample, grad, tune)	Inform the potential about a new sample during tuning.
velocity(x[, out])	Compute the current velocity at a position in parameter space.
velocity_energy(x, v_out)	Compute velocity and return kinetic energy at a position in parameter space.

4.4 Dual Averaging Step Size Adaptation

step_sizes.DualAverageAdaptation(...)	Dual averaging step size adaptation.

4.4.1 littlemcmc.step_sizes.DualAverageAdaptation

class littlemcmc.step_sizes.**DualAverageAdaptation**(*initial_step*, *target*, *gamma*, *k*, *t0*)

Dual averaging step size adaptation.

__init__(*initial_step*, *target*, *gamma*, *k*, *t0*)

Class for dual averaging step size adaptation.

Parameters

initial_step

target

gamma [float, default .05]

k [float, default .75] Parameter for dual averaging for step size adaptation. Values between 0.5 and 1 (exclusive) are admissible. Higher values correspond to slower adaptation.

t0 [int, default 10] Parameter for dual averaging. Higher values slow initial adaptation.

Methods

__init__(initial_step, target, gamma, k, t0)	Class for dual averaging step size adaptation.
current(tune)	Get current step size.
reset()	Reset step size adaptation routine.
stats()	Get step size adaptation statistics.
update(accept_stat, tune)	Update step size.
warnings()	Generate warnings from dual averaging step size adaptation.

4.5 Integrators

integration.CpuLeapfrogIntegrator(potential, ...)	Leapfrog integrator using the CPU.

4.5.1 littlemcmc.integration.CpuLeapfrogIntegrator

class littlemcmc.integration.**CpuLeapfrogIntegrator**(*potential: littlemcmc.quadpotential.QuadPotential, logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]]*)

Leapfrog integrator using the CPU.

__init__(*potential: littlemcmc.quadpotential.QuadPotential, logp_dlogp_func: Callable[[numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray]]*) → None

Instantiate a CPU leapfrog integrator.

Parameters

potential

logp_dlogp_func

Methods

__init__(potential, logp_dlogp_func, ...)	Instantiate a CPU leapfrog integrator.
compute_state(q, p)	Compute Hamiltonian functions using a position and momentum.
step(epsilon, state[, out])	Leapfrog integrator step.

CHAPTER 5

About LittleMCMC

LittleMCMC is a lightweight, performant implementation of Hamiltonian Monte Carlo (HMC) and the No-U-Turn Sampler (NUTS) in Python. This document aims to explain and contextualize the motivation and purpose of LittleMCMC. For an introduction to the user-facing API, refer to the quickstart tutorial.

5.1 Motivation and Purpose

Bayesian inference and probabilistic computation is complicated and has many moving parts[1]_. As a result, many probabilistic programming frameworks (or any library that automates Bayesian inference) are monolithic libraries that handle everything from model specification (including automatic differentiation of the joint log probability), to inference (usually via Markov chain Monte Carlo or variational inference), to diagnosis and visualization of the inference results[2]_. PyMC3 and Stan are two excellent examples of such monolithic frameworks.

However, such monoliths require users to buy in to entire frameworks or ecosystems. For example, a user that has specified their model in one framework but who now wishes to migrate to another library (to take advantage of certain better-supported features, say) must now reimplement their models from scratch in the new framework.

LittleMCMC remedies this exact use case: by isolating PyMC's HMC/NUTS code in a standalone library, users with their own log probability function and its derivative may use PyMC's battle-tested HMC/NUTS samplers.

5.2 LittleMCMC in Context

LittleMCMC stands on the shoulders of giants (that is, giant open source projects). Most obviously, LittleMCMC builds from (or, more accurately, is a spin-off project from) the PyMC project (both PyMC3 and PyMC4).

In terms of prior art, LittleMCMC is similar to several other open-source libraries, such as NUTS by Morgan Fouesneau or Sampyl by Mat Leonard. However, these libraries do not offer the same functionality as LittleMCMC: for example, they do not allow for easy changes of the mass matrix (instead assuming that an identity mass matrix), or they do not raise sampler errors or track sampler statistics such as divergences, energy, etc.

By offering step methods, integrators, quadpotentials and the sampling loop in separate Python modules, LittleMCMC offers not just a battle-tested sampler, but also an extensible one: users may configure the samplers as they wish.

Bibliography

[1] Hoffman, Matthew D., & Gelman, Andrew. (2011). The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo.

Python Module Index

l

littlemcmc, 4

Index

Symbols

C

D

H

L

N

Q

S