Chapter 7

mocat

This thesis is accompanied by an open source python Van Rossum and Drake (2009) package mocat that represents a general purpose toolbox for fully customisable Monte Carlo sampling including plug and play implementations for all of the sampling algorithms presented in this thesis (Chapter 2, Chapter 3, Chapter 5, Chapter 6).

Just like bmm, Section 4.6, mocat is hosted on PyPI (2021) and can be installed easily from the command line with pip.

```
pip install mocat
```

Source code can be found at github.com/SamDuffield/mocat.

7.1 JAX

mocat is written in python but is entirely dependent on the package JAX, Bradbury et al. (2018). JAX code is compiled using XLA Sabne (2020), a computational backend in C++, as such JAX provides lightning fast scientific calculations via a convenient python frontend. JAX documentation can be found at jax.readthedocs.io, but note the caveat that JAX is still being regularly updated and some numpy function are yet to be implemented or may not be fully optimised.

JAX adopts the familiar interface of numpy Harris et al. (2020) but goes much further. Indeed many numpy manipulations can be replaced with JAX counterparts

```python
from jax import numpy as jnp
```

for an easy speedup.

Carefully written JAX code is **differentiable**

```
from jax import grad

grad(jnp.sin)(2 * jnp.pi)
    DeviceArray(1., dtype=float32)
```

or perhaps more usefully

```
from jax import value_and_grad

value_and_grad(jnp.sin)(2 * jnp.pi)
    (DeviceArray(0., dtype=float32), DeviceArray(1., dtype=float32))
```

Additionally, JAX is equipped with a `jax.jit` function for accelerated just-in-time compilation in a similar fashion to numba Lam et al. (2015) as well as efficient parallel dispatch on modern computer architectures (GPUs and TPUs).

In order to retain differentiability and allow XLA acceleration, `if` statements as well as `for` and `while` loops have to be written with care.

`if` statements

For very simple `if` statements that require no additional function calls dependent on the value of the boolean `condition` we can replace

```
if condition:
    x = 5
else:
    x = 10
```

with

```
x = jnp.where(condition, 5, 10)
```

For more complex `if` statements where we only want to execute one of two functions we can use `jax.lax.cond`

```python
from jax.lax import cond

x = cond(condition,
         lambda y: y,
         lambda y: y * 2,
         operand=5)
```

`for` loops

Basic `for` loops that can be written in `python` using *list comprehension*

```python
out_list = [func(x) for x in range(10)]
```

can be vectorised (for a suitable `func`) using `jax.vmap`

```python
out_array = vmap(func)(jnp.arange(10))
```

Typically the vectorised `vmap` call is significantly faster than the native `python` list comprehension.

For Markovian `for` loops where the function iterates depending on its previous value, we can use the extremely flexible `jax.lax.scan`. From the `JAX` documentation we have

```python
def scan(f, init, xs):
  carry = init
  ys = []
  for x in xs:
    carry, y = f(carry, x)
    ys.append(y)
  return carry, np.stack(ys)
```

which becomes succinctly

```
from jax.lax import scan

carry, stacked_ys = scan(f, init, xs)
```

Note the flexibility in the output of `scan` - the first element `carry` represents the first element of the output of the final call to `f` whereas the second element `stacked_ys` represents the second element of the output from every call to `f` stacked into a single array. This flexibility as well as the speed of `XLA` compilation makes `jax.lax.scan` extremely useful for iterative algorithms such as Markov chain Monte Carlo Section 2.4.

`JAX` has a similar implementation for `while` loops in `jax.lax.while_loop`, however only the output from the final iteration call is returned. Since `jax.lax.scan` has the added flexibility of being able to return stacked output, it is generally favoured for `while` loops that can be reformulated as `for` loops. For `while` loops where we desire stacked output, `mocat` provides a solution with `mocat.utils.while_loop_stacked`.

Bonus comments

In order to retain differentiability (`jax.grad` and `jax.value_and_grad`), all arrays (intermediate or otherwise) within calls of `vmap`, `scan` etc must be of constant size - this can be an issue for algorithms such as rejection sampling Section 2.2.

`JAX` favours pure functions and as such does not immediately support the in-place updates that are common in `numpy` (e.g. `x[0] = 4`). This can be circumvented using `jax.ops.index_update` however it is typically preferred to use pure functions defined over full arrays or calls to `jnp.where`.

Finally, `JAX` behaves somewhat atypically in its treatment of pseudo-random seeds. In particular, every call to one of `JAX`'s functions that calls a pseudo-random number generator must be accompanied with a two element array representing a random key.

```
from jax import random

random_key = random.PRNGKey(0)
random_key
    DeviceArray([0, 0], dtype=uint32)
```

Using the same key will generate the same random numbers

```
random.normal(random_key)
    DeviceArray(-0.20584235, dtype=float32)

random.normal(random_key)
    DeviceArray(-0.20584235, dtype=float32)
```

for this reason we have to split the random key before calling a pseudo-random number generator

```
random_key, sub_key_1, sub_key_2 = random.split(random_key, 3)
random.normal(sub_key_1)
    DeviceArray(0.5781487, dtype=float32)

random.normal(sub_key_2)
    DeviceArray(0.85355157, dtype=float32)
```

Of course `JAX` has many more useful features and many more *gotchas* - all thoroughly described in the documentation jax.readthedocs.io.

7.2 Monte Carlo Sampling

`mocat` makes it easy to run the most common Monte Carlo algorithms for offline Bayesian inference as well as providing a framework to implement exciting, new sampling algorithms.

`mocat.cdict`

A fundamental object within `mocat` is that of a `cdict`. A `cdict` conveniently stores multiple named attributes including `DeviceArrays`.

```
import mocat

random_key = random.PRNGKey(0)
random_key, sub_key_1, sub_key_2 = random.split(random_key, 3)

sample = mocat.cdict(x=-5+0.1*random.normal(sub_key_1, shape=(4, 2)),
                     momenta=random.normal(sub_key_2, shape=(4, 2)),
                     name='Gaussian sample')

sample.name
    'Gaussian sample'

sample.x
    DeviceArray([[-5.1619368, -4.871614 ],
                 [-5.1136875, -5.048856 ],
                 [-5.0195227, -4.8458843],
                 [-5.037027 , -5.017855 ]], dtype=float32)

sample.momenta
    DeviceArray([[-0.38537452, -1.4707391 ],
                 [ 0.5467919 ,  2.095505  ],
                 [ 1.1165614 ,  0.16117463],
                 [-0.5371375 , -0.89213836]], dtype=float32)
```

We can display the elements of the cdict

```
sample.keys()
    dict_keys(['x', 'momenta', 'name'])
```

and even index all of the `DeviceArrays` stored within the `cdict`

```
first_sample = sample[0]

first_sample.name
    'Gaussian sample'

first_sample.x
    DeviceArray([-5.1619368, -4.871614 ], dtype=float32)

sample.momenta
    DeviceArray([-0.38537452, -1.4707391 ], dtype=float32)
```

We can also save and load `cdicts` easily

```
sample.save(path)
same_sample = mocat.load_cdict(path)
```

`mocat.Scenario`

In order to apply Monte Carlo methods, we need a distribution to sample from! For classical Monte Carlo methods such as Markov chain Monte Carlo and importance sampling we require access to evaluations of the target distribution's density function. In `mocat`, this is achieved via the so called *potential* function $U(x)$

$$U(x) = -\log \pi(x) \iff \pi(x) = \exp(-U(x)),$$

where $\pi(x)$ is the probability density function of the target distribution. A strength of Monte Carlo methods is that $\pi(x)$ (and subsequently $U(x)$) only need be defined up to normalisation constant.

In `mocat`, a target distribution's potential is stored in a `class` that inherits `mocat.Scenario`. This can be done either directly

```python
class Funnel(mocat.Scenario):
    dim = 5
    def potential(self,
                  x: jnp.ndarray,
                  random_key: jnp.ndarray = None) -> float:
        return 0.5 * (x[-1] ** 2 / 9 + (x[:-1] ** 2 / jnp.exp(x[-1]) +
        ↪  x[-1]).sum())
```

or by defining a Bayesian prior and likelihood function

```python
class BayesFunnel(mocat.Scenario):
    dim = 5
    prior_sd = 10.

    def prior_potential(self,
                        x: jnp.ndarray,
                        random_key: jnp.ndarray = None) -> float:
        return 0.5 * jnp.square(x / self.prior_sd).sum()

    def likelihood_potential(self,
                             x: jnp.ndarray,
                             random_key: jnp.ndarray = None) -> float:
        return 0.5 * (x[-1] ** 2 / 9 + (x[:-1] ** 2 / jnp.exp(x[-1]) +
        ↪  x[-1]).sum())

    def prior_sample(self,
                     random_key: jnp.ndarray) -> jnp.ndarray:
        return self.prior_sd * random.normal(random_key,
        ↪  shape=(self.dim,))
```

Note that the `Scenario` class has a compulsory `dim` attribute that is used by subsequent sampling algorithms. Additionally note that the `BayesFunnel` has a `prior_sd` parameter that is accessed within `prior_potential` via `self.prior_sd` - this technique could also be used alongside a `data` attribute accessed within `likelihood_potential`.

The `prior_sample` method is required for initiating some sampling algorithms, such as the approaches that use tempering in Chapter 5 and Chapter 6.

The additional `random_key` argument in the potential methods is there to permit the option of stochastic mini-batching in the likelihood calls - in the majority of cases it can be left as an unused argument.

Initiating an instance of the scenario

```
funnel_scen = BayesFunnel()
```

will initiate a `potential` method in the case of Bayesian prior and likelihood scenario. Additionally it will automatically initiate the first derivative of the potential `grad_potential` and `potential_and_grad` as well as the equivalent methods for `prior_potential` and `likelihood_potential`.

Markov Chain Monte Carlo

We can now develop our sampling algorithms. Recall that Markov chain Monte Carlo (MCMC), Section 2.4 collects samples by iterating a π-invariant kernel

$$x^{(i)} \sim K(\cdot \mid x^{(i-1)}), \qquad i = 1, \dots, N,$$

where K most commonly consists of a proposal followed by an accept-reject step. `mocat` has a built-in `MCMCSampler` class that can be inherited to build bespoke MCMC sampling algorithms.

`mocat` has built-in the very general gradient based sampler `mocat.Underdamped` of Horowitz (1991) which as described in Section 6.2 incorporates the popular MALA and HMC sampling algorithms. `mocat` also has the basic random-walk Metropolis Hastings algorithm `mocat.RandomWalk` whose code we describe here

```python
class RandomWalk(MCMCSampler):
    name = 'Random Walk'
    correction = Metropolis

    def __init__(self,
                 stepsize: float = None):
        super().__init__()
        self.parameters.stepsize = stepsize
        self.tuning.target = 0.234

    def startup(self,
                scenario: Scenario,
                n: int,
                initial_state: cdict,
                initial_extra: cdict,
                **kwargs) -> Tuple[cdict, cdict]:
        initial_state, initial_extra = super().startup(scenario, n,
                                                       initial_state, initial_extra, **kwargs)
        initial_extra.random_key, scen_key = random.split(initial_extra.random_key)
        initial_state.potential = scenario.potential(initial_state.value, scen_key)
        return initial_state, initial_extra

    def proposal(self,
                 scenario: Scenario,
                 reject_state: cdict,
                 reject_extra: cdict) -> Tuple[cdict, cdict]:
        proposed_state = reject_state.copy()

        d = scenario.dim
        x = reject_state.value

        stepsize = reject_extra.parameters.stepsize

        reject_extra.random_key, subkey, scen_key = random.split(reject_extra.random_key, 3)

        proposed_state.value = x + jnp.sqrt(stepsize) * random.normal(subkey, (d,))
        proposed_state.potential = scenario.potential(proposed_state.value, scen_key)

        return proposed_state, reject_extra

    def acceptance_probability(self,
                               scenario: Scenario,
                               reject_state: cdict, reject_extra: cdict,
                               proposed_state: cdict, proposed_extra: cdict) -> float:
        return jnp.minimum(1., jnp.exp(- proposed_state.potential + reject_state.potential))
```

Firstly, note that the `MCMCSampler` has a `correction` attribute. This represents a `mocat.Correction` object and determines the nature of the accept reject step. The three built-in corrections are

- `Uncorrected` - always accept proposal.

- `Metropolis` - accept proposal with probability determined by the `MCMCSampler`'s `acceptance_probability` method, otherwise duplicate previous sample.
- `RMMetropolis` - as above but additionally adapt the `stepsize` parameter with a Robbins-Monro schedule Andrieu and Thoms (2008) according to the `MCMCSampler`'s `tuning` attribute.

Of course `mocat` permits fully customisable inheritance from the `mocat.Correction` class in order to create alternative MCMC algorithms based on the same proposal.

There are therefore four key methods to define when inheriting `MCMCSampler`.

- `__init__` is the method that is called when an instance of the sampler is initiated. The `MCMCSampler` already creates a `parameters` `cdict`, the sampler's `__init__` is where the sampling algorithm parameters and their defaults are defined - in the case of `RandomWalk`, only the `stepsize` parameter. It is also the opportunity to set sampler defaults in the `tuning` `cdict`

```
rw_sampler = mocat.RandomWalk(stepsize=0.1)
rw_sampler.tuning
    mocat.cdict({'parameter': 'stepsize', 'target': 0.234,
     ↪  'metric': 'alpha', 'monotonicity': 'decreasing'})
```

- `startup` is called when the sampler is first exposed to the `Scenario`. The purpose of `startup` is to setup the `cdicts` - `initial_state` and `initial_extra`. When `jax.lax.scan` is called the sampler will adjust iterated `state` and `extra` `cdicts` were anything in `extra` will be discarded at the end (e.g. random keys) and anything in `state` will be stacked and returned. The `startup` ensures all attributes are initiated correctly to be consistent through `jax.lax.scan` - this includes, for example, initiating gradient evaluations. By default `initial_state` will only be initiated with an initial `value` and `initial_extra` with a `random_key` and iteration counter `iter`.
- `proposal` represents the function that modifies the `state` and `extra` at each iteration. If in addition it is desired to make iterative modifications that are always accepted, these can be applied in the `always` method that is applied before `proposal` (not required for `RandomWalk`).
- `acceptance_probability` determines the probability of accepting a proposal for when the `correction` is set to `Metropolis` (or an inheritance thereof).

We are then ready to sample!

```
mcmc_sample = mocat.run(funnel_scen, rw_sampler, n=100000,
 ↪  random_key=random.PRNGKey(0))
mcmc_sample.keys()
    dict_keys(['value', 'alpha', 'potential', 'time', 'summary'])
```

where the output is a `cdict` where the samples are stored in `value`, runtime in `time`, a summary of the run parameters in `summary` and any other attributes initiated by the `MCMCSampler` or its `Correction` in `startup`.

Alternatively we could have sampled with stepsize adaptation

```
rw_sampler = RandomWalk(stepsize=0.1)

mcmc_sample = mocat.run(funnel_scen, rw_sampler, n=100000,
 ↪  random_key=random.PRNGKey(0), correction=mocat.RMMetropolis)
mcmc_sample.keys()
    dict_keys(['value', 'stepsize', 'alpha', 'potential', 'time',
     ↪  'summary'])
```

Transport Sampling

`mocat` also provides a framework for Monte Carlo sampling under an alternative paradigm - that of iteratively updating a full particle approximation of fixed size, as opposed to gradually building a particle approximation of increasing size as in MCMC. In `mocat` samplers following this paradigm (such as sequential Monte Carlo with likelihood tempering) inherit the `TransportSampler` class.

A vanilla `TransportSampler` has four key methods to be implemented

- `__init__` as in `MCMCSampler`, initiate any sampler `parameters` and their default values however `TransportSampler` does not pre-initiate a `tuning` attribute.
- `startup` as in `MCMCSampler` except that `initial_state` now contains a full particle approximation and thus attributes (including `value` which by default is initiated with `n` calls to `scenario.prior_sample`) having a leading dimension of length `n`. `initial_extra` attributes remain singular in length.

- `update` modifies the particle approximation at each iteration.
- `termination_criterion` tells `mocat.run` when to stop.

`mocat` has built-in implementations of Stein Variational Gradient Descent (SVGD) Liu and Wang (2016) (`mocat.SVGD` with some basic kernels defined in `mocat.kernels`) as well as a customisable framework for sequential Monte Carlo with likelihood tempering Section 6.1 including

- `TemperedSMCSampler` a general class with `forward_proposal` and `log_weight` methods to be defined. Either requires a `temperature_schedule` at initiation or the method `next_temperature_adaptive` to be defined.
- `MetropolisedSMCSampler` - specifically for π_t-invariant transition kernels as in Chapter 6. Takes on initiation an `mcmc_sampler` argument which is an `MCMCSampler` object defining the nature the transition kernel. Supports a `temperature_schedule` or effective sample size based adaptive tempering.
- `RMMetropolisedSMCSampler` - as above but additionally adapts the stepsize of the `mcmc_sampler` using a Robbins-Monro schedule according to `mcmc_sampler.tuning` as in Algorithm 21.

We can again sample using `mocat.run`. For SVGD

```
svgd_sampler = mocat.SVGD(stepsize=0.1,
 ↪  kernel=mocat.kernels.Gaussian(), max_iter=1000)

svgd_sample = mocat.run(funnel_scen, svgd_sampler, n=100,
 ↪  random_key=random.PRNGKey(0))

svgd_sample.keys()
    dict_keys(['value', 'potential', 'grad_potential', 'time',
     ↪  'summary'])

svgd_sample.value.shape
    (1001, 100, 5)
```

or for sequential Monte Carlo with likelihood tempering

```
mcmc_sampler=mocat.Underdamped(leapfrog_steps=10, stepsize=0.1,
 ↪  friction=jnp.inf)

smc_sampler =
 ↪  mocat.RMMetropolisedSMCSampler(mcmc_sampler=mcmc_sampler)

smc_sample = mocat.run(funnel_scen, smc_sampler, n=10000,
 ↪  random_key=random.PRNGKey(0))

smc_sample.keys()
    dict_keys(['value', 'log_weight', 'ess', 'prior_potential',
     ↪  'grad_prior_potential', 'likelihood_potential',
     ↪  'grad_likelihood_potential', 'potential', 'grad_potential',
     ↪  'temperature', 'log_norm_constant', 'alpha', 'momenta',
     ↪  'stepsize', 'time', 'summary'])
```

Sample Metrics

`mocat` contains a collection of functions to analyse sample quality.

Let us analyse the `mcmc_sample` we generated from the `BayesFunnel` distribution using `RandomWalk`. The first thing we might check is some univariate marginals

```
mocat.hist_1d_samples(mcmc_sample, dim=0)
mocat.hist_1d_samples(mcmc_sample, dim=-1)
```

or perhaps the bivariate marginals without and with burn-in

```
mocat.plot_2d_samples(mcmc_sample, dim1=0, dim2=-1)
mocat.plot_2d_samples(mcmc_sample[1000:], dim1=0, dim2=-1)
```

Given that we modified the stepsize adaptively we should check our acceptance rates were appropriate

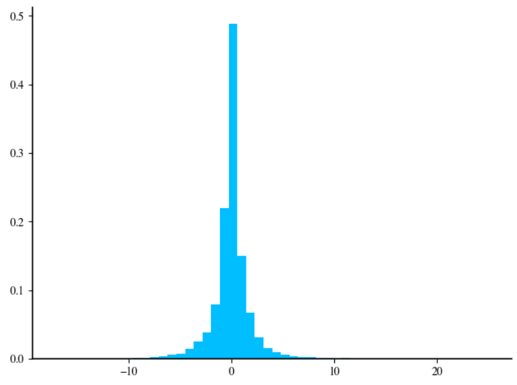
Fig. 7.1 `dim=0`

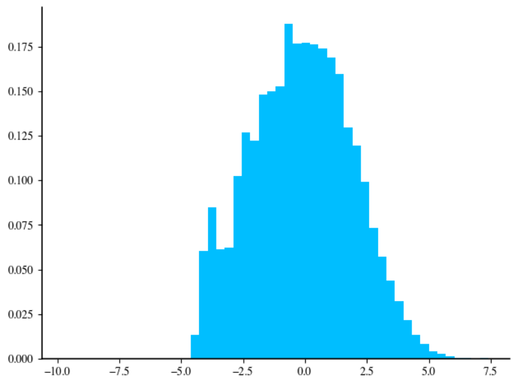
Fig. 7.2 `dim=-1`

Univariate marginals for MCMC samples from `BayesFunnel`.

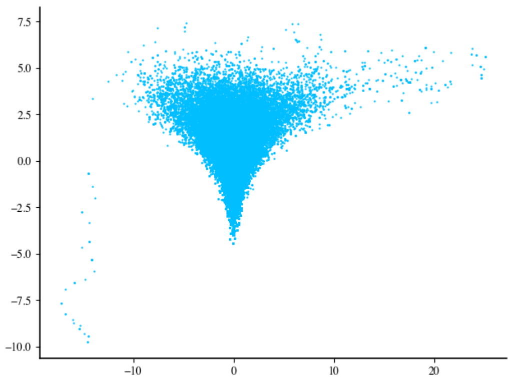
Fig. 7.3 No burn-in

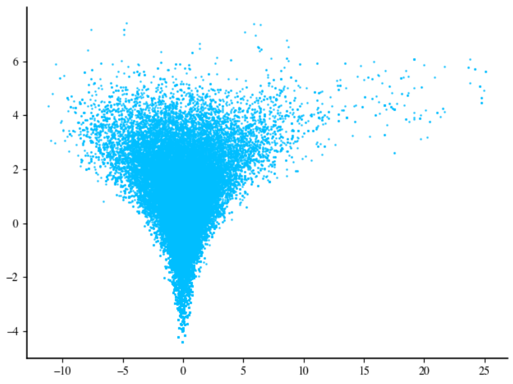
Fig. 7.4 Burn-in of 1000

Bivariate marginals for MCMC samples from `BayesFunnel`.

```
rw_sampler.tuning.target
    0.234

mcmc_sample.alpha.mean()
    DeviceArray(0.23056579, dtype=float32)
```

Lovely stuff. We can even visualise the stepsize adaptation (simply using `matplotlib` Hunter (2007))

```
import matplotlib.pyplot as plt

plt.plot(mcmc_sample.stepsize)
```

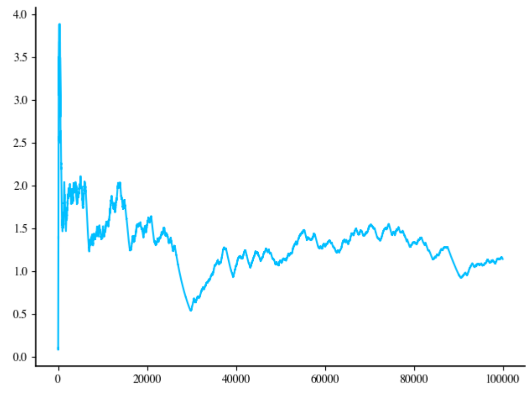
Fig. 7.5 `RandomWalk` stepsize adaptation on `BayesFunnel`

It wouldn't be Markov chain Monte Carlo without some trace plots

```
plt.plot(mcmc_sample.potential[1000:])
plt.plot(mcmc_sample.value[1000:, 0])
```

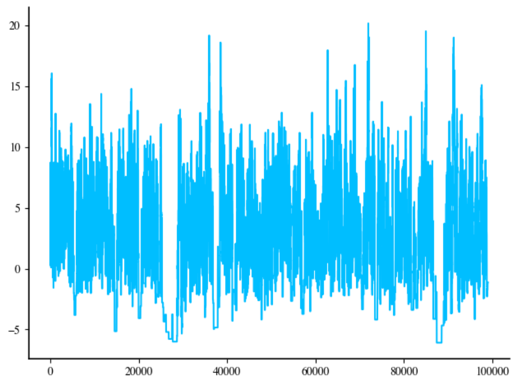
Fig. 7.6 `potential`

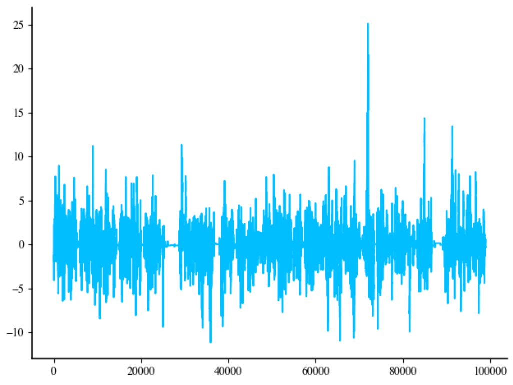
Fig. 7.7 `dim=0`

Trace plots for MCMC samples (with burn-in of 1000) from `BayesFunnel`.

and autocorrelations

```
mocat.autocorrelation_plot(mcmc_sample.potential[1000:])
mocat.autocorrelation_plot(mcmc_sample.value[1000:, 0])
```

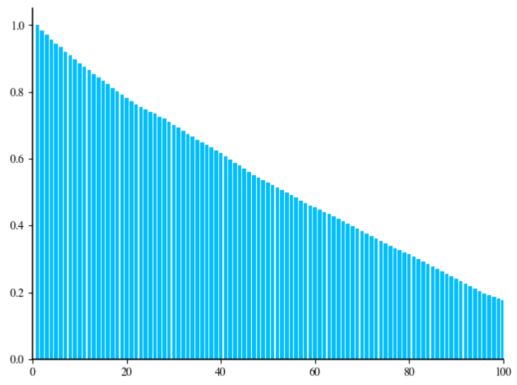

Fig. 7.8 `potential`

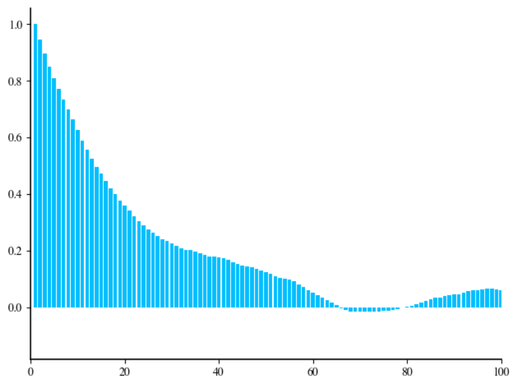

Fig. 7.9 `dim=0`

Autocorrelation plots for MCMC samples (with burn-in of 1000) from `BayesFunnel`.

Indeed we can also calculate the `ess_autocorrelation` (or `integrated_autocorrelation_time`)

```
mocat.ess_autocorrelation(mcmc_sample.potential)
    DeviceArray(8026.1, dtype=float32)

vmap(mocat.ess_autocorrelation)(mcmc_sample.value.T)
    DeviceArray([2079.1, 1665.7, 767.7, 647.3, 33951.8],
     ↪  dtype=float32)
```

In addition, we can analyse the quality of any sample (irrespective of its generating mechanism) using a kernelised Stein discrepancy (KSD) Liu et al. (2016) (which we mini-batch to avoid having to calculate a 100000x100000 gram matrix).

```
mcmc_sample.grad_potential =
 ↪  vmap(funnel_scen.grad_potential)(mcmc_sample.value)

mocat.ksd(mcmc_sample, kernel=mocat.kernels.Gaussian(),
 ↪  ensemble_batchsize=100, random_key=random.PRNGKey(0))
    DeviceArray(0.9774283, dtype=float32)
```

where we had to compute the gradient of the potential (required for KSDs) as it wasn't computed during sampling - this of course wouldn't have been the case if we had used `mocat.Underdamped`.

The kernelised Stein discrepancy is applicable to any sample including `TemperedSMCSamplers` as it also takes a *log_weight* argument. Remember that the output of a `TransportSampler` is a particle approximation at each iteration and so metrics should only be called on the final particle approximation

```
mocat.ksd(smc_sample[-1], kernel=mocat.kernels.Gaussian(),
 ↪  ensemble_batchsize=100, random_key=random.PRNGKey(0),
 ↪  log_weight=smc_sample.log_weight[-1])
```

Naturally the output of a `MetropolisedSMCSampler` also contains the attributes generated during sampling `ess`, `temperature`, `alpha`, `stepsize` and `log_norm_constant` that can be analysed to assess sample quality in addition to the marginal plots and KSD.

7.3 ABC

But what about problems where we cannot compute `likelihood_potential` but can can compute `likelihood_sample`? We can do exactly that using `mocat`'s submodule `abc`

```
from mocat import abc
```

We can define a target distribution with an intractable likelihood density by inheriting the `abc.ABCScenario` class

```python
class GaussianABC(abc.ABCScenario):
    dim = 3
    prior_std = 5.
    likelihood_matrix = jnp.array([[1., -0.5, 2.],
                                   [-0.4, -0.1, 0.]])
    likelihood_std = 1.
    data = jnp.array([3., -1.])

    def prior_sample(self,
                     random_key: jnp.ndarray) -> jnp.ndarray:
        return self.prior_std * random.normal(random_key, (self.dim,))

    def prior_potential(self,
                        x: jnp.ndarray,
                        random_key: jnp.ndarray = None) -> float:
        return 0.5 * jnp.square(x / self.prior_std).sum()

    def likelihood_sample(self,
                          x: jnp.ndarray,
                          random_key: jnp.ndarray) -> jnp.ndarray:
        return self.likelihood_matrix @ x
               + self.likelihood_std * random.normal(random_key,
                ↪  shape=(self.likelihood_matrix.shape[0],))
```

If data is summarised, as is common in approximate Bayesian computation (ABC), this is defined implicitly in the `data` attribute and `likelihood_sample` method, i.e. `data` represents the summarised data and `likelihood_sample` directly simulates summarised synthetic data.

`abc.ABCScenario` additionally hosts a `distance_function` method that defaults to the Euclidean distance

```python
def distance_function(self,
                      simulated_data: jnp.ndarray) -> float:
    return jnp.sqrt(jnp.square(simulated_data - self.data).sum())
```

mocat.abc has built-in implementations of the ABC algorithms described Section 2.5. Indeed abc.VanillaABC inherits abc.ImportanceABC and is jointly an importance and rejection sampler. The threshold parameter of abc.VanillaABC can be defined explicitly

```
vanilla_abc = abc.VanillaABC(threshold=3.)
```

or post-hoc via an acceptance rate

```
vanilla_abc = abc.VanillaABC(acceptance_rate=0.1)
```

and as usual we can run using mocat.run

```
gaussian_abc_scen = GaussianABC()

vanilla_abc_sample = mocat.run(gaussian_abc_scen, vanilla_abc,
 ↪  n=10000, random_key=random.PRNGKey(0))

(vanilla_abc_sample.log_weight == 0).mean()
    DeviceArray(0.1, dtype=float32)
```

where vanilla_abc_sample.log_weight represents an array of accept (0.) and reject (-inf) values.

mocat.abc also contains a fully customisable ABCMCMCSampler class and an implementation of RandomWalkABC which by default runs with mocat.Metropolis correction but can also adaptively determine both the stepsize and threshold parameters using the RMMetropolis-DiagStepsize correction as described in Vihola and Franks (2020)

```
abc_mcmc_sampler = abc.RandomWalkABC()

abc_mcmc_sample = mocat.run(gaussian_abc_scen, abc_mcmc_sampler,
 ↪   n=10000, random_key=random.PRNGKey(0),
 ↪   correction=abc.RMMetropolisDiagStepsize())

abc_mcmc_sampler.tuning
    mocat.cdict({'parameter': 'threshold', 'target': 0.1, 'metric':
     ↪   'alpha', 'monotonicity': 1})

abc_mcmc_sample.alpha.mean()
    DeviceArray(0.08237284, dtype=float32)

plt.plot(abc_mcmc_sample.stepsize)
plt.plot(abc_mcmc_sample.threshold)
```

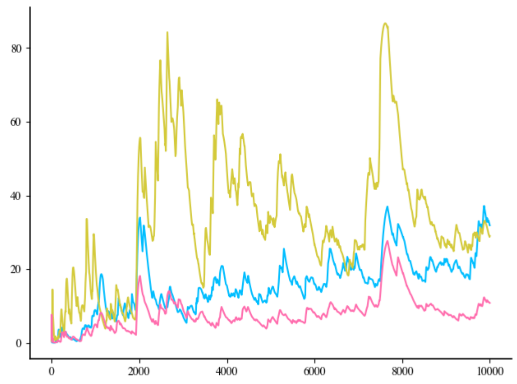

Fig. 7.10 Stepsize adaptation to match sample covariance in each dimension.

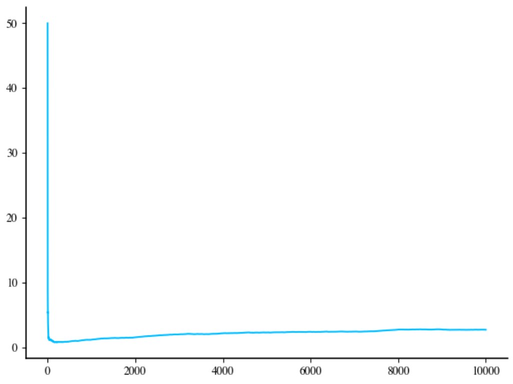

Fig. 7.11 Threshold adaptation

Adaptive ABC-MCMC on `GaussianABC`.

Similarly, `mocat.abc` implements ABC-SMC via a customisable `ABCSMCSampler` and `MetropolisedABCSMCSampler`. By default, `MetropolisedABCSMCSampler` uses an `abc.RandomWalkABC` proposal, an adaptive effective sample size based threshold schedule and sets the `MetropolisedABCSMCSampler.adapt_mcmc_params` to modify the stepsize (diagonal pre-conditioner) to the diagonal sample covariance scaled by $d/2.38^2$ as in Del Moral et al. (2012).

```
abc_smc_sampler = abc.MetropolisedABCSMCSampler()
abc_smc_sample = mocat.run(gaussian_abc_scen, abc_smc_sampler, n=1000,
↪   random_key=random.PRNGKey(0))

abc_smc_sample.keys()
    dict_keys(['value', 'log_weight', 'ess', 'prior_potential',
    ↪   'simulated_data', 'distance', 'threshold', 'alpha', 'time',
    ↪   'summary'])

plt.plot(abc_smc_sample.ess)
plt.plot(abc_smc_sample.alpha.mean(1))
plt.plot(abc_smc_sample.threshold)
plt.plot(abc_smc_sample.distance.mean(1))
```

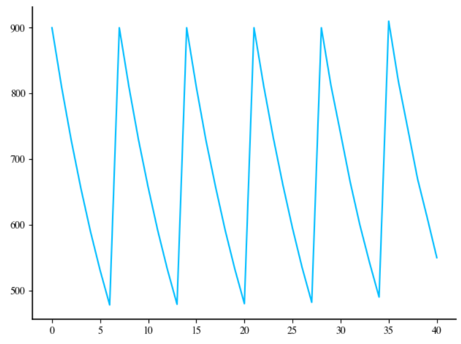
Fig. 7.12 Effective sample size

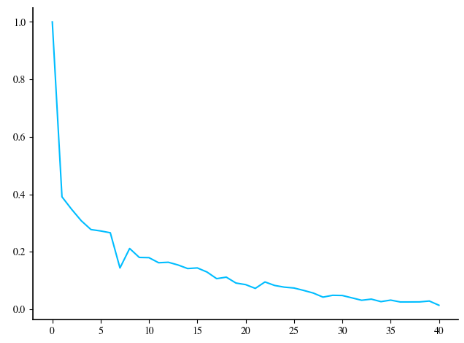
Fig. 7.13 Average Metropolis acceptance rate

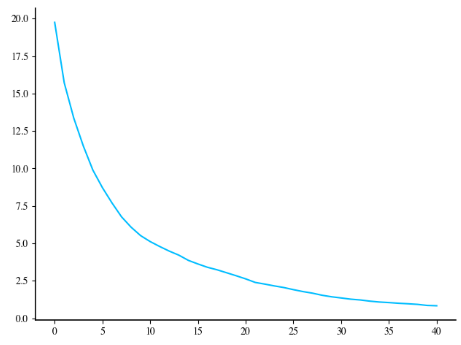
Fig. 7.14 Adaptive threshold parameter

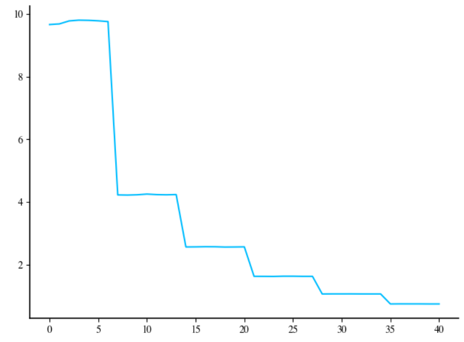
Fig. 7.15 Average distance between simulated and true data

Adaptive ABC-SMC on `GaussianABC`.

mocat also includes implementation of the tempered ensemble Kalman inversion (EKI) introduced in Chapter 5. Ensemble Kalman inversion represents a TransportSampler and only requires the Scenario to have prior_sample and likelihood_sample implemented. A general implementation is stored in mocat.TemperedEKI and EKI with adaptive temperature schedule using the pseudo-weights Equation (5.17) is found in mocat.AdaptiveTemperedEKI. The stopping criterion can be adjusted by modifying the max_temperature attribute and/or the termination_criterion method.

```
eki_sample = mocat.run(gaussian_abc_scen, mocat.AdaptiveTemperedEKI(),
 ↪  n=1000, random_key=random.PRNGKey(0))
```

7.4 State-space Models

Sequential Bayesian inference in state-space models Section 2.6 is also supported via mocat's ssm submodule.

```
from mocat import ssm
```

A fully general state-space model can be stored by inheriting the ssm.StateSpaceModel. For example consider the univariate non-linear benchmark model Gordon et al. (1993)

```python
class NonLinear1DBenchmark(ssm.StateSpaceModel):
    name = '1D Non-linear Benchmark'
    dim = 1
    dim_obs = 1

    def __init__(self,
                 initial_sd: float = jnp.sqrt(2.),
                 transition_sd: float = jnp.sqrt(10.),
                 likelihood_sd: float = 1.,
                 name: str = None):
        self.initial_sd = initial_sd
        self.transition_sd = transition_sd
        self.likelihood_sd = likelihood_sd
        super().__init__(name=name)

    def initial_potential(self,
                          x: jnp.ndarray,
                          t: float) -> float:
        return 0.5 * jnp.square(x / self.initial_sd).sum()

    def initial_sample(self,
                       t: float,
                       random_key: jnp.ndarray) -> jnp.ndarray:
        return random.normal(random_key, (1,)) * self.initial_sd

    def transition_potential(self,
                             x_previous: jnp.ndarray,
                             t_previous: float,
                             x_new: jnp.ndarray,
                             t_new: float) -> float:
        transition_mean = 0.5 * x_previous+ 25 * x_previous / (1 + x_previous ** 2) + 8 *
        ↪  jnp.cos(1.2 * t_previous)
        return 0.5 * jnp.square((x_new - transition_mean) / self.transition_sd).sum()

    def transition_sample(self,
                          x_previous: jnp.ndarray,
                          t_previous: float,
                          t_new: float,
                          random_key: jnp.ndarray) -> jnp.ndarray:
        transition_mean = 0.5 * x_previous+ 25 * x_previous / (1 + x_previous ** 2) + 8 *
        ↪  jnp.cos(1.2 * t_previous)
        return transition_mean + random.normal(random_key, (1,)) * self.transition_sd

    def likelihood_potential(self,
                             x: jnp.ndarray,
                             y: jnp.ndarray,
                             t: float) -> float:
        lik_mean = x ** 2 / 20
        return 0.5 * jnp.square((y - lik_mean)/self.likelihood_sd).sum()

    def likelihood_sample(self,
                          x: jnp.ndarray,
                          t: float,
                          random_key: jnp.ndarray) -> jnp.ndarray:
        lik_mean = x ** 2 / 20
        return lik_mean + random.normal(random_key, (1,)) * self.likelihood_sd
```

Observe the additional compulsory attribute `dim_obs` describing the dimension of a single observation. The state-space model comes down to the implementation of the following six methods

- `initial_potential` evaluates the potential corresponding to the first latent variable $p(x_0)$.
- `initial_sample` generates a single random sample from $p(x_0)$.
- `transition_potential` evaluates the potential corresponding to the transition density $p_t(x_t \mid x_{t-1})$ which may vary with t.
- `transition_sample` generates a single random sample from $p_t(x_t \mid x_{t-1})$.
- `likelihood_potential` evaluates the potential corresponding to the transition density $p_t(y_t \mid x_t)$ which may vary with t.
- `likelihood_sample` generates a synthetic observation from $p_t(y_t \mid x_t)$. Not necessary for the most basic inference in state-space models (i.e. `ssm.BootstrapFilter`).

Synthetic values of both the underlying trajectory and observations can then be generated

```
benchmark_ssm = NonLinear1DBenchmark()

simulated_values = benchmark_ssm.simulate(t_all=jnp.arange(10),
 ↪   random_key=random.PRNGKey(0))

simulated_values.keys()
    dict_keys(['x', 'y', 't', 'name'])
```

Linear Gaussian

As discussed in Equation (2.34), a convenient class of state-space models occurs when all distributions $p(x_0)$, $p_t(x_t \mid x_{t-1})$ and $p_t(y_t \mid x_t)$ are linear and Gaussian. `mocat` provides a customisable `ssm.LinearGaussian` class and `ssm.TimeHomogenousLinearGaussian` which reduces computational cost by assuming all of the matrices in the transition and likelihood are time homogeneous, i.e.

$$p_t(x_t \mid x_{t-1}) = p(x_t \mid x_{t-1}), \qquad p_t(y_t \mid x_t) = p(y_t \mid x_t).$$

```
lg_ssm = ssm.TimeHomogenousLinearGaussian(
              initial_mean=jnp.zeros(2),
              initial_covariance=jnp.eye(2),
              transition_matrix=jnp.eye(2),
              transition_covariance=0.2 * jnp.eye(2),
              likelihood_matrix=jnp.array([[0.5, 0.5]]),
              likelihood_covariance=0.3*jnp.eye(1))
```

Specifically for `ssm.LinearGaussian` models, we can run exact marginal filtering - Algorithm 10

```
t = jnp.arange(10)
y = random.normal(random.PRNGKey(0), shape=(10, 1))

filter_means, filter_covs =
 ↪  ssm.run_kalman_filter_for_marginals(lg_ssm, y, t)
```

and exact marginal smoothing Algorithm 11

```
smoother_means, smoother_covs =
 ↪  ssm.run_kalman_smoother_for_marginals(lg_ssm, y, t,
 ↪  filter_output=(filter_means, filter_covs))
```

Particle Methods

For more general state-space models such as the `NonLinear1DBenchmark` model defined above, we cannot do exact inference. Instead we can adopt the Monte Carlo approaches described in Section 2.6 and Chapter 3.

Underlying all of these particle methods is the concept of a particle filter Algorithm 12. A particle filter is defined by its a sequential proposal distribution $q(x_t \mid x_{t-1}, y_t)$ that is permitted to incorporate the new observation y_t. `mocat.ssm` provides a customisable `ParticleFilter` class to be inherited as well as a built-in implementation of the most basic `BootstrapFilter` described here

```python
class BootstrapFilter(ParticleFilter):
    name = 'Bootstrap Filter'

    def proposal_potential(self,
                           ssm_scenario: StateSpaceModel,
                           x_previous: jnp.ndarray,
                           t_previous: float,
                           x_new: jnp.ndarray,
                           y_new: jnp.ndarray,
                           t_new: float) -> Union[float, jnp.ndarray]:
        return ssm_scenario.transition_potential(x_previous,
         ↪  t_previous, x_new, t_new)

    def proposal_sample(self,
                        ssm_scenario: StateSpaceModel,
                        x_previous: jnp.ndarray,
                        t_previous: float,
                        y_new: jnp.ndarray,
                        t_new: float,
                        random_key: jnp.ndarray) -> jnp.ndarray:
        return ssm_scenario.transition_sample(x_previous, t_previous,
         ↪  t_new, random_key)

    def intermediate_log_weight(self,
                                ssm_scenario: StateSpaceModel,
                                x_previous: jnp.ndarray,
                                t_previous: float,
                                x_new: jnp.ndarray,
                                y_new: jnp.ndarray,
                                t_new: float) -> Union[float,
                                 ↪  jnp.ndarray]:
        return -ssm_scenario.likelihood_potential(x_new, y_new, t_new)
```

where `intermediate_log_weight` refers to the function $h(x_{t-1}, x_t, y_t)$ that updates the logarithm of the (unnormalised) importance weights

$$\log w_t = \log w_{t-1} + h(x_{t-1}, x_t, y_t) + k \iff w_t \propto w_{t-1} e^{h(x_{t-1}, x_t, y_t)}.$$

`mocat.ssm` has built-in more efficient, informed particle filtering for the specific `NonLinearGaussian` class of state-space models Section 5.1, Godsill et al. (2004) where transitions and likelihoods are time-homogenous and take the form

$$p(x_t \mid x_{t-1}) = \mathbf{N}(x_t \mid f(x_{t-1}), \mathrm{R}),$$
$$p(y_t \mid x_t) = \mathbf{N}(y_t \mid \mathrm{H}x_t, \mathrm{Q}).$$

In this case the (locally) optimal proposal Equation (2.39) is tractable and an implementation is provided in `ssm.OptimalNonLinearGaussianParticleFilter`. Additionally these models are amenable to ensemble Kalman filtering Algorithm 19 and an implementation is stored in `ssm.EnsembleKalmanFilter`.

Now for a given `ParticleFilter` we can tackle inference.

Online

For online particle filtering (for marginals $p(x_T \mid y_{0:T})$ we can initiate a particle approximation at the first observation

```
y, t = simulated_values.y, simulated_values.t

pf = ssm.BootstrapFilter()

random_key, key0 = random.split(random.PRNGKey(0))
filter_marginal = ssm.initiate_particles(benchmark_ssm, pf, n=1000,
 ↪  random_key=key0, y=y[0], t=t[0])

filter_marginal.keys()
    dict_keys(['value', 'log_weight', 't', 'y', 'ess'])
```

and then update for new observations

```
random_key, key1 = random.split(random_key)

filter_marginal = ssm.propagate_particle_filter(benchmark_ssm, pf,
 ↪  filter_marginal, y_new=y[1], t_new=t[1], random_key=key1)
```

which by default will append the proposal to the trajectory - the most recent values can be extracted with `filter_marginal[-1]`.

Additionally, we can generate an online particle approximation to the full joint smoothing distribution $p(x_{0:T} \mid y_{0:T})$ using the techniques from Chapter 3

```
smoothing_joint = ssm.initiate_particles(benchmark_ssm, pf, n=1000,
 ↪  random_key=random.PRNGKey(0), y=y[0], t=t[0])

smoothing_joint = ssm.propagate_particle_smoother(benchmark_ssm, pf,
 ↪  filter_marginal, y_new=y[1], t_new=t[1], random_key=key1, lag=5)
```

which will execute the online smoother with backward simulation Algorithm 17, setting `backward_sim=False` in `propagate_particle_smoother` will run the online smoother with particle filter block proposal Algorithm 16.

Offline

In the case we have all of the observations at once, we can execute forward filtering-backward simulation Algorithm 13 to generate a particle approximation to the smoothing distribution $p(x_{0:T} \mid y_{0:T})$. We can do this either via an explicit two-step procedure

```
random_key, key0, key1 = random.split(random.PRNGKey(0), 3)

filtering_marginals =
 ↪  ssm.run_particle_filter_for_marginals(benchmark_ssm, pf, y, t,
 ↪  n=1000, random_key=key0)

smoothing_joint = ssm.backward_simulation(benchmark_ssm,
 ↪  filtering_marginals, key1)
```

or all at once

```
smoothing_joint =
 ↪  ssm.forward_filtering_backward_simulation(benchmark_ssm, pf, y, t,
 ↪  n_samps=100, random_key=random.PRNGKey(0))

smoothing_joint.keys()
    dict_keys(['value', 't', 'y', 'ess', 'num_transition_evals',
     ↪  'time', 'pf_time', 'bsi_time'])

plt.plot(smoothing_joint.t, smoothing_joint.value[:, :, 0],
 ↪  c='orange')
plt.plot(simulated_values.t, simulated_values.x, c='red')
```

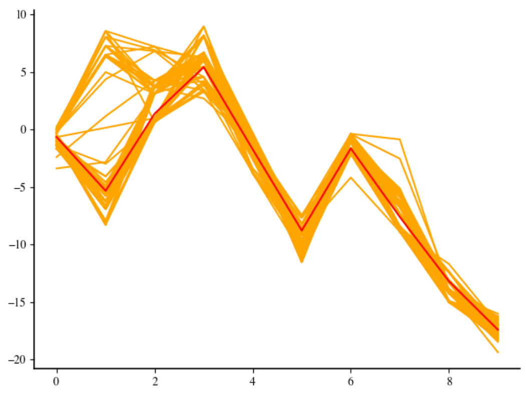

Fig. 7.16 FFBSi applied to `NonLinear1DBenchmark`