particles Documentation

Release alpha

Nicolas Chopin

Dec 15, 2023

CONTENTS

1 Overview **3**
- 1.1 Features . . . 3
- 1.2 Notebook tutorials . . . 3

2 Installation **57**
- 2.1 Python 2 vs Python 3 . . . 57
- 2.2 Requirements . . . 57
- 2.3 Local installation, organisation of the package . . . 58
- 2.4 Installation: recommended method . . . 58
- 2.5 Installation: alternative method . . . 58

3 General structure **59**
- 3.1 Folders . . . 59
- 3.2 API reference . . . 59

Python Module Index **101**

Index **103**

This package was developed to complement the book:

An introduction to Sequential Monte Carlo

by Nicolas Chopin and Omiros Papaspiliopoulos.

The scripts used to perform the numerical experiments discussed in the book may be found in folder `book`.

The documentation refers sometimes to the book for some theoretical details, but otherwise is meant to be self-sufficient.

To get an overview of what **particles** can do, we strongly recommend that you have a look at the **notebook tutorials** first (see *Overview* section).

To get help on a specific module, class, function, etc., see the indices below, or use the `help` command in python:

```
from particles import resampling as rs
help(rs)  # help on module resampling
help(rs.multinomial)  # help on a specific function in that module
```

CHAPTER

ONE

OVERVIEW

1.1 Features

Here is a brief list of the features of particles:

- state-space models may be defined as python objects, in a basic form of probabilistic programming.
- Bootstrap filter, guided filter, auxiliary particle filter.
- exact filtering/smoothing algorithms: Kalman (linear Gaussian models), and forward-backward (finite hidden Markov models).
- Several resampling schemes are implemented.
- Sequential quasi-Monte Carlo (and related tools: Hilbert ordering, RQMC sampling).
- Smoothing: on-line and off-line, O(N^2) and O(N) versions of standard algorithms (FFBS, two-filter).
- SMC samplers: IBIS (data-tempering) and SMC tempering. Static models may be defined as Python objects.
- Bayesian inference for state-space models: several PMCMC (particle MCMC algorithms are implemented), such as PMMH and Particle Gibbs. Also SMC^2.
- Genealogy-based variance estimators (Chan & Lai, 2013; Lee & Whiteley, 2018; Olsson & Douc, 2019).
- A Pima indian example is included.

1.2 Notebook tutorials

The tutorials below walk you through most of the functionality of the package.

Note: These tutorials are also available as ipython/jupyter notebooks: Once you have installed the package, go to subfolder `docs/source/notebooks` and open them with `jupyter` or `ipython` to interact with the recorded sessions.

1.2.1 Basic tutorial

This basic tutorial gives a brief overview of some of functionality of the **particles** package. Details are deferred to more advanced tutorials.

First steps: defining a state-space model

We start by importing some standard libraries, plus some modules from the package.

[1]:
```
%matplotlib inline
import warnings; warnings.simplefilter('ignore')  # hide warnings

# standard libraries
from matplotlib import pyplot as plt
import numpy as np
import seaborn as sb

# modules from particles
import particles  # core module
from particles import distributions as dists  # where probability distributions are␣
↪defined
from particles import state_space_models as ssm  # where state-space models are defined
from particles.collectors import Moments
```

Let's define our first state-space model **class**. We consider a basic stochastic volalitility model, that is:

$$
\begin{aligned}
X_0 &\sim N\left(\mu, \frac{\sigma^2}{1-\rho^2}\right), \\
X_t | X_{t-1} = x_{t-1} &\sim N\left(\mu + \rho(x_{t-1} - \mu), \sigma^2\right), && t \geq 1, \\
Y_t | X_t = x_t &\sim N\left(0, e^{x_t}\right), && t \geq 0.
\end{aligned}
$$

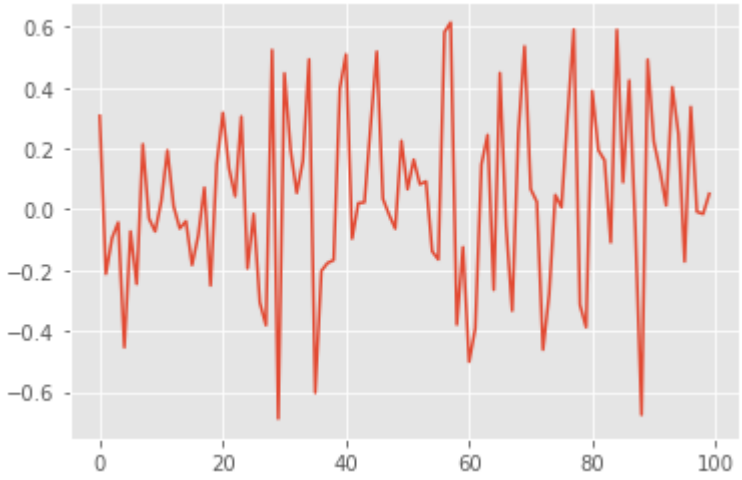
Note that this model depends on fixed parameter $\theta = (\mu, \rho, \sigma)$.

In case you are not familiar with the notations above: a state-space model is a model for a joint process (X_t, Y_t), where (X_t) is an unobserved Markov process (the *state* of the system), and Y_t is some noisy measurement of X_t (hence it is observed). For instance, in stochastic volatility, Y_t is typically the log-return of some asset, and X_t its (unobserved) volatility.

The code below is hopefully transparent.

[2]:
```
class StochVol(ssm.StateSpaceModel):
    def PX0(self):  # Distribution of X_0
        return dists.Normal(loc=self.mu, scale=self.sigma / np.sqrt(1. - self.rho**2))
    def PX(self, t, xp):  # Distribution of X_t given X_{t-1}=xp (p=past)
        return dists.Normal(loc=self.mu + self.rho * (xp - self.mu), scale=self.sigma)
    def PY(self, t, xp, x):  # Distribution of Y_t given X_t=x (and possibly X_{t-1}=xp)
        return dists.Normal(loc=0., scale=np.exp(x))

my_model = StochVol(mu=-1., rho=.9, sigma=.1)  # actual model
true_states, data = my_model.simulate(100)  # we simulate from the model 100 data points

plt.style.use('ggplot')
plt.figure()
plt.plot(data)
```

[2]:
```
[<matplotlib.lines.Line2D at 0x7f602e517ca0>]
```



Methods `PX0`, `PX` and `PY` return objects that represent probability distributions (defined in module `distributions`. Parameters μ, ρ and σ are defined as **attributes** of a class instance: i.e. `self.mu` and so on. (`self` is the generic name for an instance of a class in Python.)

If your are not very familiar with OOP (object oriented programming) and related concepts (classes, instances, etc.), here is a simple way to understand them in our context:

- The **class** `StochVol` represents the parametric class of stochastic volatility models.
- The object `my_model` (a **class instance** of `StochVol`) defines a particular model, where parameters μ, ρ and σ are fixed to certain values.

In particular, we can inspect the attributes of `my_model` that store the parameter values.

[3]:
```
my_model.mu, my_model.rho, my_model.sigma
```

[3]:
```
(-1.0, 0.9, 0.1)
```

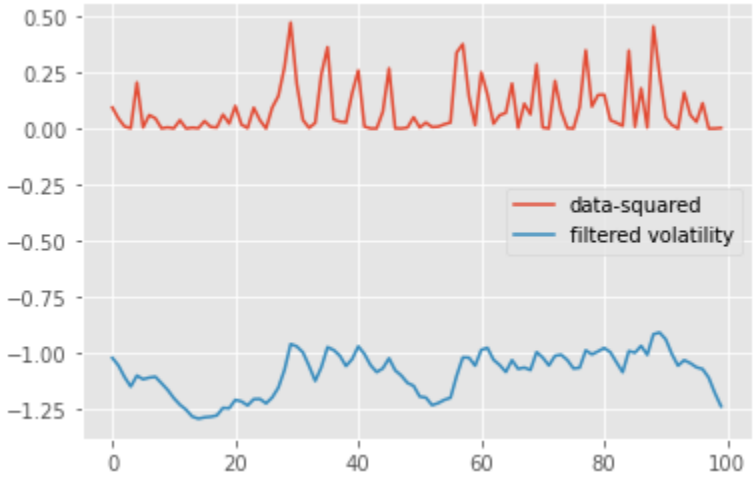
Class `StochVol` is a sub-class of `StateSpaceModel`. (You can see that from the first line of its definition.) For instance, it inherits a method called `simulate` that generates states and datapoints from the considered model.

Particle filtering

There are several particle algorithms that one may associate to a given state-space model. Here we consider the simplest option: the **bootstrap filter**. (See next tutorial for how to implement a guided or auxiliary filter.) The code below runs such a bootstrap filter for $N = 100$ particles, using stratified resampling.

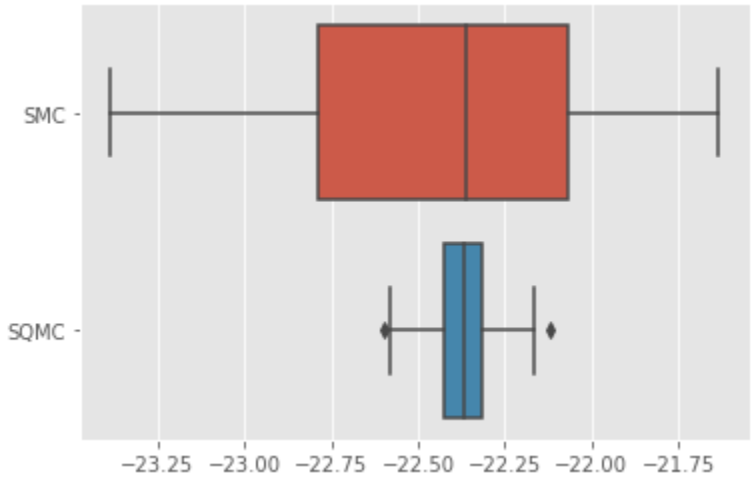
[4]:
```
fk_model = ssm.Bootstrap(ssm=my_model, data=data)  # we use the Bootstrap filter
pf = particles.SMC(fk=fk_model, N=100, resampling='stratified',
                   collect=[Moments()], store_history=True)  # the algorithm
pf.run()  # actual computation

# plot
plt.figure()
plt.plot([yt**2 for yt in data], label='data-squared')
plt.plot([m['mean'] for m in pf.summaries.moments], label='filtered volatility')
plt.legend()
```

[4]:
```
<matplotlib.legend.Legend at 0x7f602dd6ec40>
```



Recall that a particle filter is a Monte Carlo algorithm: each execution returns a random, slightly different result. Thus, it is useful to run a particle filter multiple times to assess how stable are the results. Say, for instance, that we would like to compare the variability of the log-likelihood estimate provided by a particle filter, when either a standard Monte Carlo algorithm is used, or its QMC variant, called SQMC. The following command runs 30 times each of these two algorithms.

[5]:
```
results = particles.multiSMC(fk=fk_model, N=100, nruns=30, qmc={'SMC':False, 'SQMC':True}
↪)
plt.figure()
sb.boxplot(x=[r['output'].logLt for r in results], y=[r['qmc'] for r in results]);
```



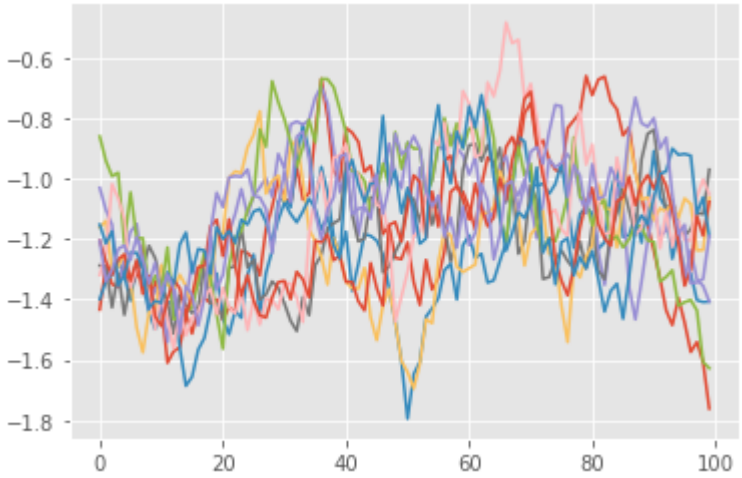
As expected, the variance of SQMC estimates is quite lower.

Command `multiSMC` makes it possible to run several particle filters, with varying options. Parallel execution is also possible, as explained in next tutorial.

Smoothing

So far, we have only considered filtering; let's try smoothing, that is, approximating the distribution of the whole trajectory $X_{0:T}$, given data $Y_{0:T} = y_{0:T}$, for some fixed time horizon $T = 100$. In particular, we are going to sample smoothing trajectories from the output of the first particle filter we ran a few steps above.

[6]:
```
smooth_trajectories = pf.hist.backward_sampling(10)
plt.figure()
plt.plot(smooth_trajectories);
```



Here, we used the standard version of the FFBS (forward filtering, backward sampling) algorithm, which generates smoothing trajectories from the particle **history** (which we generated when running pf above). Other smoothing algorithms are available, see the next tutorial.

(Bayesian) Parameter estimation

Finally, we consider how to estimate the parameter $\theta = (\alpha, \rho, \sigma)$ from a given data-set. First, we set up a prior as follows.

[7]:
```
prior_dict = {'mu':dists.Normal(),
              'sigma': dists.Gamma(a=1., b=1.),
              'rho':dists.Beta(9., 1.)}
my_prior = dists.StructDist(prior_dict)
```

Again, the code above should be fairly readable: the prior for μ is $N(0, 1)$, the one for σ is a Gamma(1,1) and so on. (As before, probability distributions are represented by objects defined in the `distributions` module.)

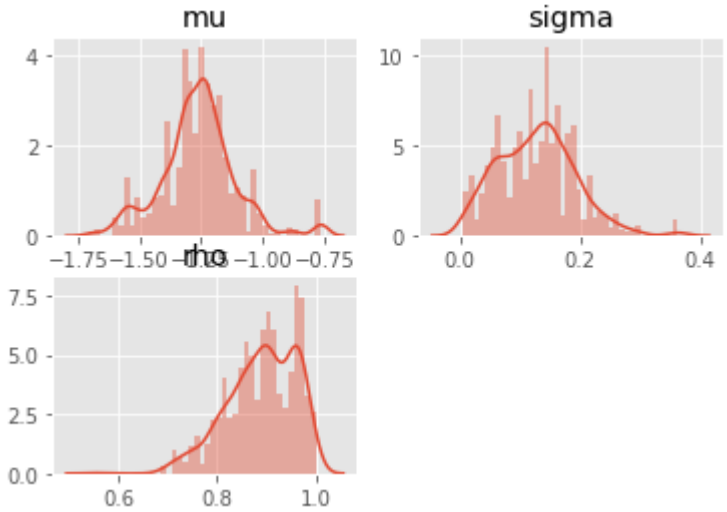
This may not be a sensible prior distribution; for instance it restricts ρ to $[0, 1]$. However, this will suffice for our purposes. A popular way to simulate from the posterior distribution of the parameters of a state-space model is PMMH, a particular instance of the PMMC framework. Basically, this is a MCMC algorithm that runs at each iteration a particle filter so as to evaluate the likelihood.

[8]:
```
%%capture
from particles import mcmc  # where the MCMC algorithms (PMMH, Particle Gibbs, etc) live
pmmh = mcmc.PMMH(ssm_cls=StochVol, prior=my_prior, data=data, Nx=50, niter = 1000)
pmmh.run()  # Warning: takes a few seconds
```

Again, the code above is hopefully readable: PMMH is run for 1000 iterations (`niter`), the particle filter run at each iteration have $N_x = 50$ particles, and so on. One point worth mentioning is that we pass as an argument the *class* `StochVol`. Again, remember that this class indeed represents the considered parametric class (as opposed to `my_model`, which was a stochastic volatility model, for certain fixed parameter values).

OK, we have waited long enough, let's plot the results.

[9]:
```
# plot the marginals
burnin = 100  # discard the 100 first iterations
for i, param in enumerate(prior_dict.keys()):
    plt.subplot(2, 2, i+1)
    sb.distplot(pmmh.chain.theta[param][burnin:], 40)
    plt.title(param)
```



These results might not be very reliable, given that we used a fairly small number of MCMC iterations. If you are familiar with MCMC, you may wonder how the Metropolis proposal was set: by default, PMMH uses an adaptive Gaussian random walk proposal, such that the covariance matrix of the random step is iteratively adapted to the running simulation.

The library also implements SMC^2, and Particle Gibbs, but that will be covered in the next tutorial.

The end

That's all, folks! This very basic tutorial is over. If you crave for more, head to the other tutorials:

- Advanced tutorial for state-space models: covers the same topics as above (filtering, smoothing, parameter estimation for state-space models) but with more details.
- Tutorial on Bayesian estimation: covers PMCMC and SMC^2 algorithms that may be used to estimate the parameters of a state-space model.
- Tutorial on SMC samplers: a tutorial on how to run a SMC sampler (such as IBIS or tempering SMC) for a given static target distribution.
- How to define manually Feynman-Kac models: an advanced tutorial on how to define your own Feynman-Kac models.

1.2.2 Advanced Tutorial (geared toward state-space models)

This tutorial covers more or less the same topics as the basic tutorial (filtering and smoothing of state-space models), but in greater detail.

Defining state-space models

We consider a state-space model of the form:

$$
\begin{aligned}
X_0 &\sim N(0,1) \\
X_t &= f(X_{t-1}) + U_t, \quad U_t \sim N(0, \sigma_X^2) \\
Y_t &= X_t + V_t, \quad V_t \sim N(0, \sigma_Y^2)
\end{aligned}
$$

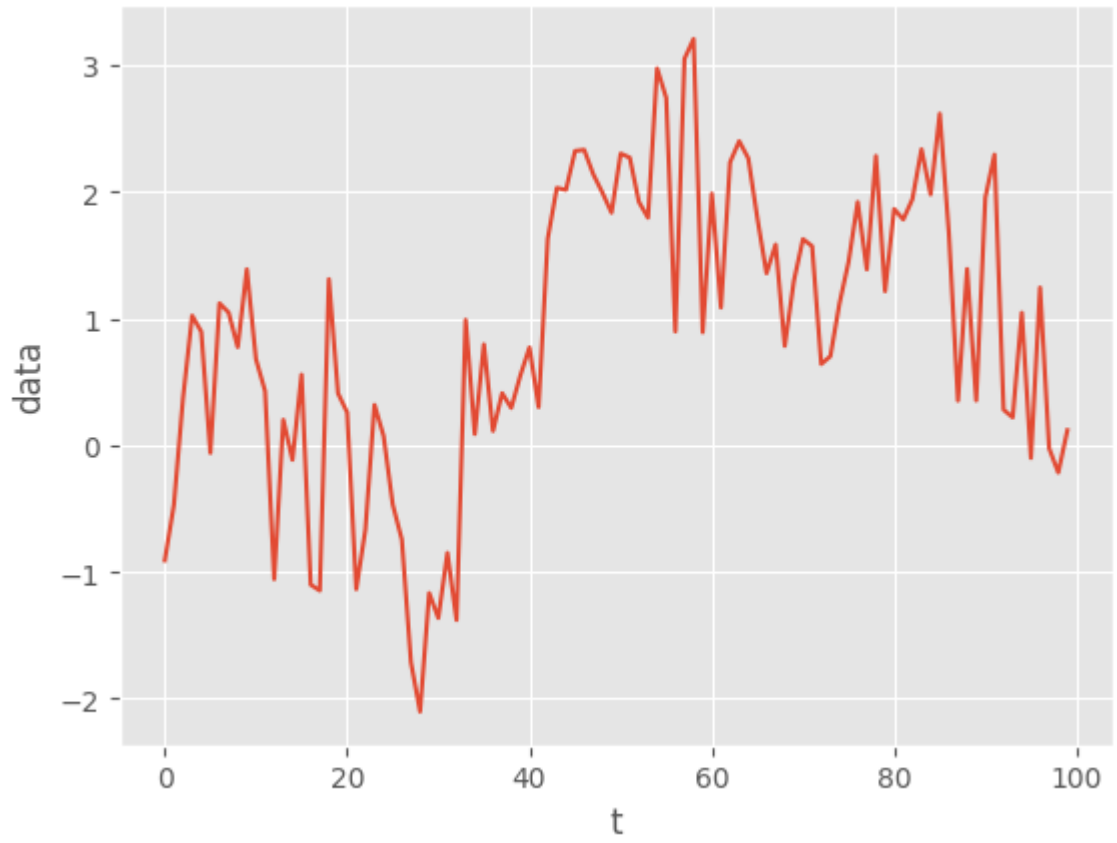
where function f is defined as follows: $f(x) = \tau_0 - \tau_1 * \exp(\tau_2 * x)$. This model comes from Population Ecology; there X_t stands for the logarithm of the population size of a given species. This model may be defined as follows.

[1]:
```
# the usual imports
from matplotlib import pyplot as plt
import seaborn as sb
import numpy as np

# imports from the package
import particles
from particles import state_space_models as ssm
from particles import distributions as dists


class ThetaLogistic(ssm.StateSpaceModel):
    """ Theta-Logistic state-space model (used in Ecology).
    """
    default_params = {'tau0':.15, 'tau1':.12, 'tau2':.1, 'sigmaX': 0.47, 'sigmaY': 0.39}

    def PX0(self):  # Distribution of X_0
        return dists.Normal()

    def f(self, x):
        return (x + self.tau0 - self.tau1 * np.exp(self.tau2 * x))

    def PX(self, t, xp):  #  Distribution of X_t given X_{t-1} = xp (p=past)
        return dists.Normal(loc=self.f(xp), scale=self.sigmaX)

    def PY(self, t, xp, x):  # Distribution of Y_t given X_t=x, and X_{t-1}=xp
        return dists.Normal(loc=x, scale=self.sigmaY)
```

This is most similar to what we did in the previous tutorial (for stochastic volatility models): methods `PX0`, `PX` and `PY` return objects defined in module `distributions`. (See the documentation of that module for a list of available distributions).

The only novelty is that we defined (as a class attribute) the dictionary `default_parameters`, which provides default values for each parameter. When it is defined, each parameter that is not set explicitly when instantiating (calling) `ThetaLogistic` is replaced by its default value:

[2]:
```
my_ssm = ThetaLogistic()  # use default values for all parameters
x, y = my_ssm.simulate(100)

plt.style.use('ggplot')
plt.plot(y)
plt.xlabel('t')
plt.ylabel('data');
```



"Bogus Parameters" (parameters that do not appear in `PX0`, `PX` and `PY`) are simply ignored:

[3]:
```
just_for_fun = ThetaLogistic(tau2=0.3, bogus=92.)  # ok
```

This behaviour may look surprising, but it will allow us to define prior distributions that involve hyper-parameters.

Automatic definition of `FeynmanKac` objects

We have seen in the previous tutorial how to run a bootstrap filter: we first define some `Bootstrap` object, and then passes it to SMC.

[4]:
```
fk_boot = ssm.Bootstrap(ssm=my_ssm, data=y)
my_alg = particles.SMC(fk=fk_boot, N=100)
my_alg.run()
```

In fact, `ssm.Bootstrap` is a subclass of `FeynmanKac`, the base class for objects that represent "Feynman-Kac models" (covered in Chapters 5 and 10 of the book). To make things simple, a Feynman-Kac model is a "recipe" for our SMC

algorithms; in particular, it tells us:

1. how to sample each particle X_t^n at time t, given their ancestors X_{t-1}^n;
2. how to reweight each particle X_t^n at time t.

The bootstrap filter is a particular "recipe", where:

1. we sample the particles X_t^n according to the state transition of the model; in our case a $N(f(x_{t-1}), \sigma_X^2)$ distribution.
2. we reweight the particles according to the likelihood of the model; here the density of $N(x_t, \sigma_Y^2)$ at point y_t.

The class `ssm.Bootstrap` defines this recipe automatically from the supplied state-space model and data.

The bootstrap filter is not the only available "recipe". We may want to run a *guided* filter, where the particles are simulated according to user-chosen proposal kernels. Such proposal kernels may be defined by adding methods `proposal` and `proposal0` to our `StateSpaceModel` class:

[5]:
```
class ThetaLogistic_with_prop(ThetaLogistic):
    def proposal0(self, data):
        return self.PX0()
    def proposal(self, t, xp, data):
        prec_prior = 1. / self.sigmaX**2
        prec_lik = 1. / self.sigmaY**2
        var = 1. / (prec_prior + prec_lik)
        mu = var * (prec_prior * self.f(xp) + prec_lik * data[t])
        return dists.Normal(loc=mu, scale=np.sqrt(var))

my_better_ssm = ThetaLogistic_with_prop()
```

In this particular case, we implemented the "optimal" proposal, that is, the distribution of X_t given X_{t-1} and Y_t. (Check this is indeed this case, this is a simple exercise!). (For simplicity, the proposal at time 0 is simply the distribution of X_0, so this one is not optimal.)

Now we may define our guided Feynman-Kac model:

[6]:
```
fk_guided = ssm.GuidedPF(ssm=my_better_ssm, data=y)
```

An APF (auxiliarly particle filter) may be implemented in the same way: for this, we must also define method `logeta`, which computes the auxiliary function used in the resampling step; see the documentation and the end of Chapter 10 of the book.

Running a particle filter

Here is the signature of class `SMC`:

[7]:
```
alg = particles.SMC(fk=fk_guided, N=100, qmc=False, resampling='systematic', ESSrmin=0.5,
                    store_history=False, verbose=False, collect=None)
```

Apart from `fk` (which expects a `FeynmanKac` object), all the other arguments are optional. Here is what they do:

- `N`: the number of particles
- `qmc`: whether to use the QMC (quasi-Monte Carlo) version
- `resampling`: which resampling scheme to use (possible choices: `'multinomial'`, `'residual'`, `'stratified'`, `'systematic'` and `'ssp'`)

- `ESSrmin`: the particle filter resamples at each iteration such that ESS / N is below this threshold; set it to `1.` (resp. `0.`) to resample every time (resp. to never resample)
- `verbose`: whether to print progress information

The remaining arguments (`store_history` and `collect`) will be explained in the following sections.

Once we have a created a SMC object, we may run it, either step by step, or in one go. For instance:

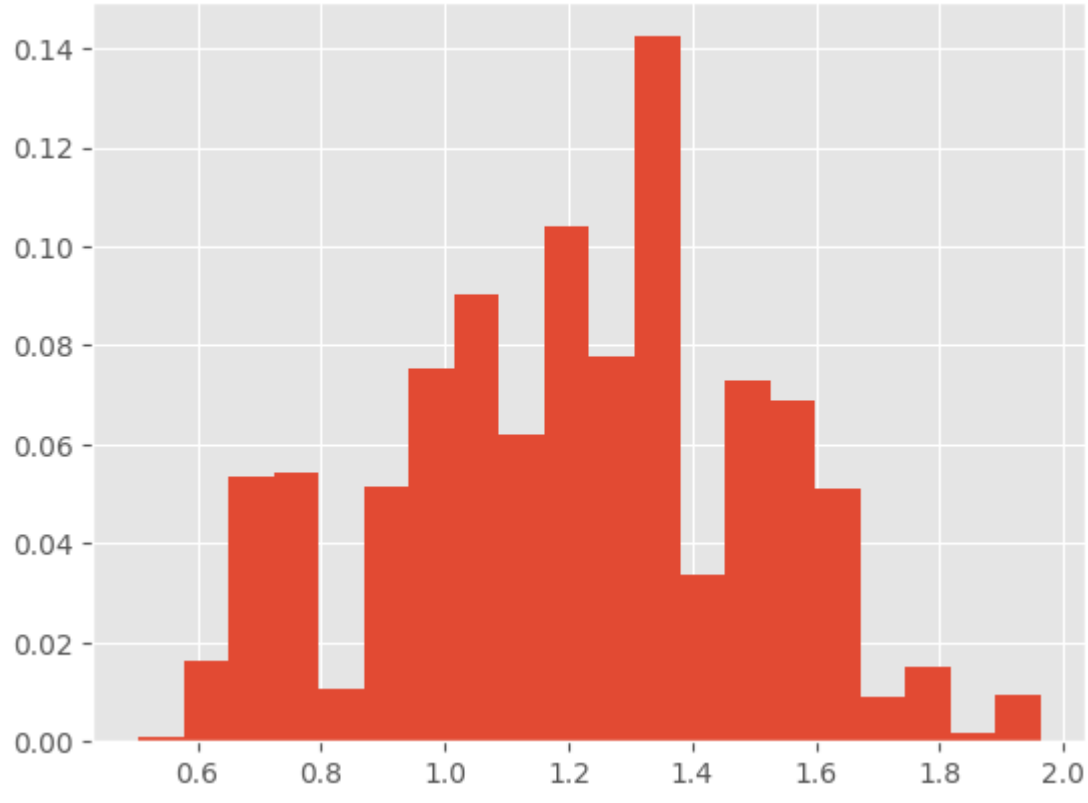
[8]:
```
next(alg) # processes data-point y_0
next(alg)  # processes data-point y_1
for _ in range(8):
    next(alg)  # processes data-points y_3 to y_9
# alg.run()  # would process all the remaining data-points
```

At any time, object `alg` has the following attributes:

- `alg.t`: index of next iteration
- `alg.X`: the N current particles X_t^n; typically a (N,) or (N,d) numpy array
- `alg.W`: the N normalised weights W_t^n (a (N,) numpy array)
- `alg.Xp`: the N particles at the previous iteration, X_{t-1}^n
- `alg.A`: the N ancestor variables: A[3] = 12 means that the parent of X_t^3 was X_{t-1}^{12}.
- `alg.summaries`: various summaries collected at each iteration.

Let's do for instance a weighted histogram of the particles.

[9]:
```
plt.hist(alg.X, 20, weights=alg.W);
```

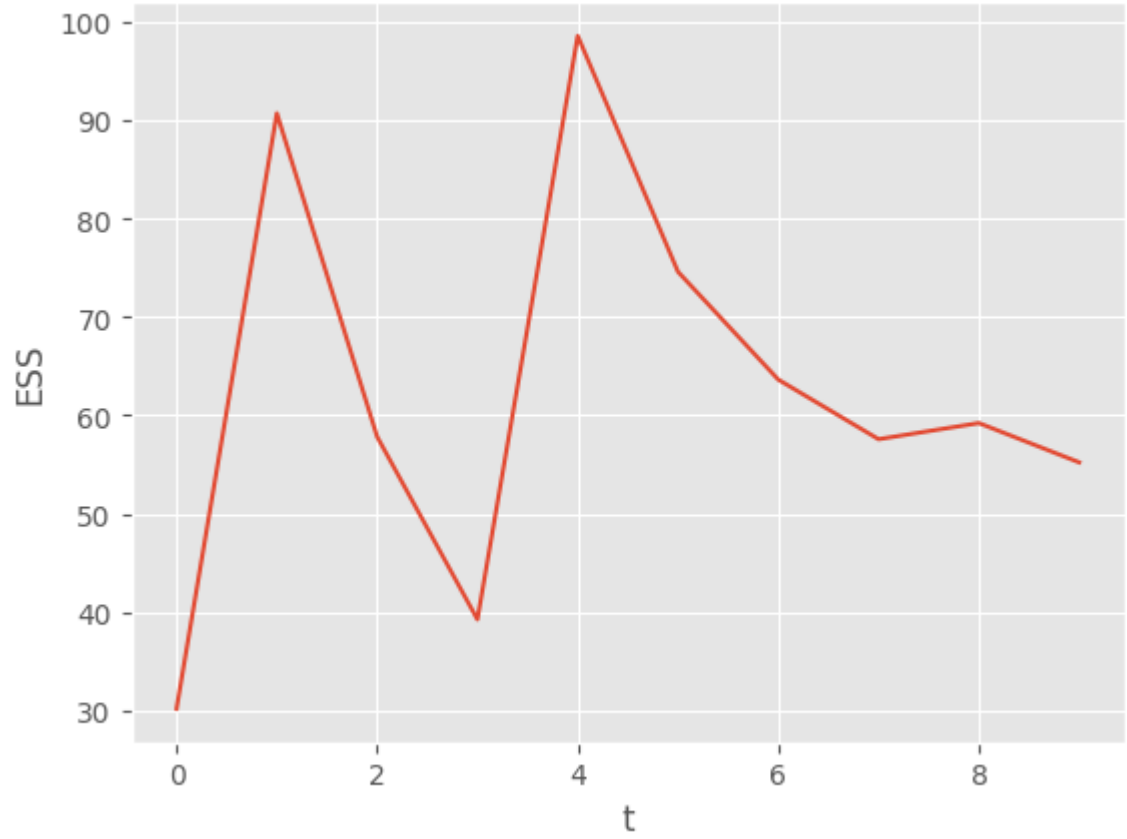


Object alg.summaries contains various lists of quantities collected at each iteration, such as:

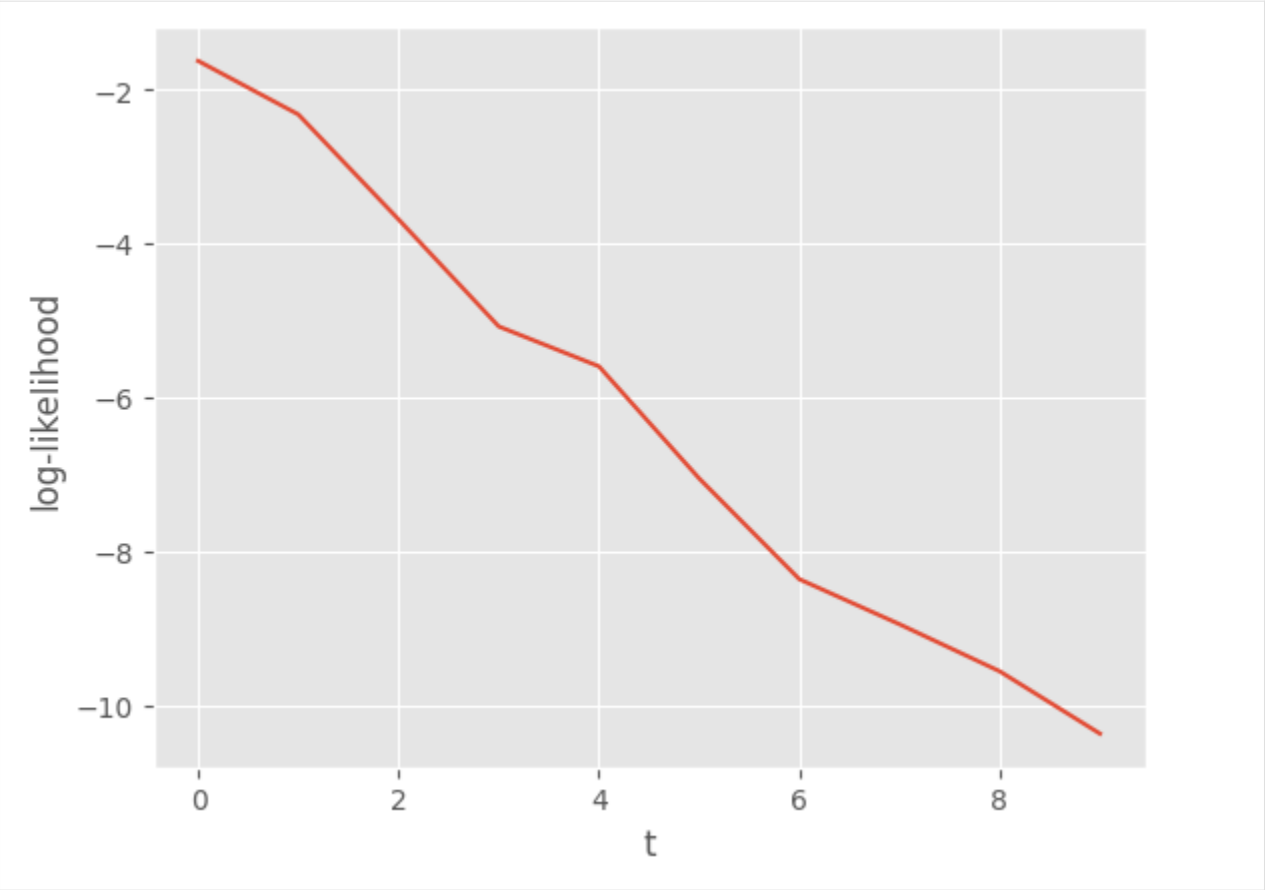
```
* `alg.summaries.ESSs`: the ESS (effective sample size) at each iteration
* `alg.summaries.rs_flags`: whether or not resampling was triggered at each step
* `alg.summaries.logLts`: estimates of the log-likelihood of the data $y_{0:t}$
```

All this and more is explained in the documentation of the `collectors` module. Let's plot the ESS and the log-likelihood:

[10]:
```
plt.plot(alg.summaries.ESSs)
plt.xlabel('t')
plt.ylabel('ESS');
```



[11]:
```
plt.plot(alg.summaries.logLts)
plt.xlabel('t')
plt.ylabel('log-likelihood');
```



Running many particle filters in one go

Function multiSMC accepts the same arguments as `SMC` plus the following extra arguments:

- `nruns`: number of runs
- `nprocs`: if >0, number of CPU cores to use; if <=0, number of cores *not to* use; i.e. `nprocs=0` means use all cores
- `out_func`: a function that is applied to each resulting particle filter (see below).

To explain how exactly `multiSMC` works, let's try to compare the bootstrap and guided filters for the theta-logistic model we defined at the beginning of this tutorial:

[12]:
```
outf = lambda pf: pf.logLt
results = particles.multiSMC(fk={'boot':fk_boot, 'guid':fk_guided},
                             nruns=20, nprocs=1, out_func=outf)
```

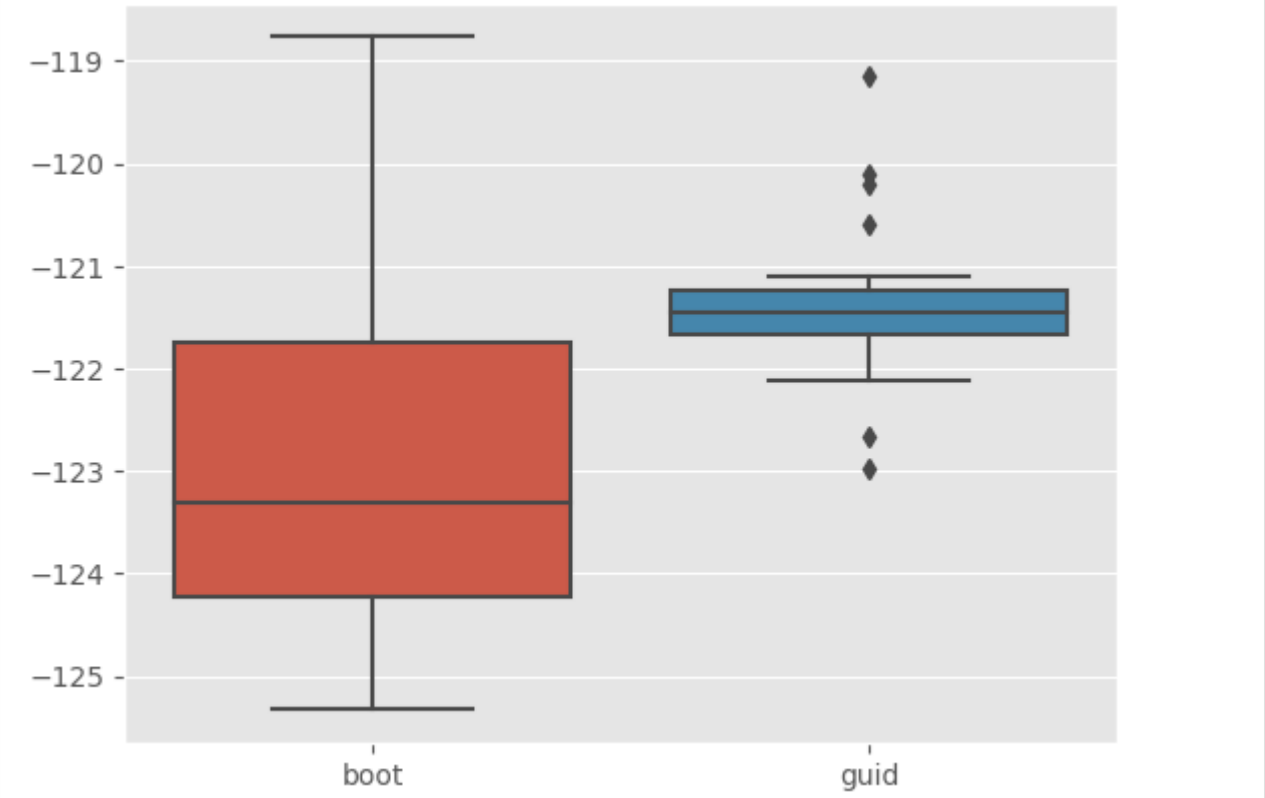
The command above runs **40** particle algorithms (on a single core): 20 bootstrap filters, and 20 guided filters. The output, `results`, is a list of 40 dictionaries; each dictionary contains the following (key, value) pairs:

- `'model'`: either `'boot'` or `'guid'` (according to whether a bootstrap or guided filter has been run)
- `'run'`: a run indicator (between 0 and 19)
- `'output'`: the result of `outf(pf)` where pf is the SMC object that was run. (If `outf` is set to None, then the SMC object is returned.)

The rationale for function `outf` is that SMC objects may take a lot of memory in certain cases (especially if you set `store_history=True`, see section on smoothing below), so we may want to save only some results of interest rather than the complete object itself. Here the output is simply the estimate of the log-likelihood of the (complete) data computed by each particle filter. Let's check if the guided filter provides lower-variance estimates, relative to the bootstrap filter.

[13]:
```
sb.boxplot(x=[r['fk'] for r in results],  y=[r['output'] for r in results])
```

[13]:
```
<Axes: >
```



This is indeed the case. To understand this line of code, you must be a bit familiar with list comprehensions.

More generally, function `multiSMC` may be used to run multiple SMC algorithms, while varying any possible arguments; for more details, see the documentation of `multiSMC` and of the module `particles.utils`.

Collectors, on-line smoothing

We have said that `alg.summaries` (where `alg` is a SMC object) contains **lists** that contains quantities computed each iteration (such as the ESS, the log-likelihood estimates). It is possible to compute extra such quantities such as:

- moments: at each time t, a dictionary with keys 'mean', and 'var', which stores the component-wise weighted means and variances.
- on-line smoothing estimates (naive, and $O(N^2)$, see module `collectors` for more details)

by providing a list of `Collector` objects to parameter `collect`. For instance, to collect moments:

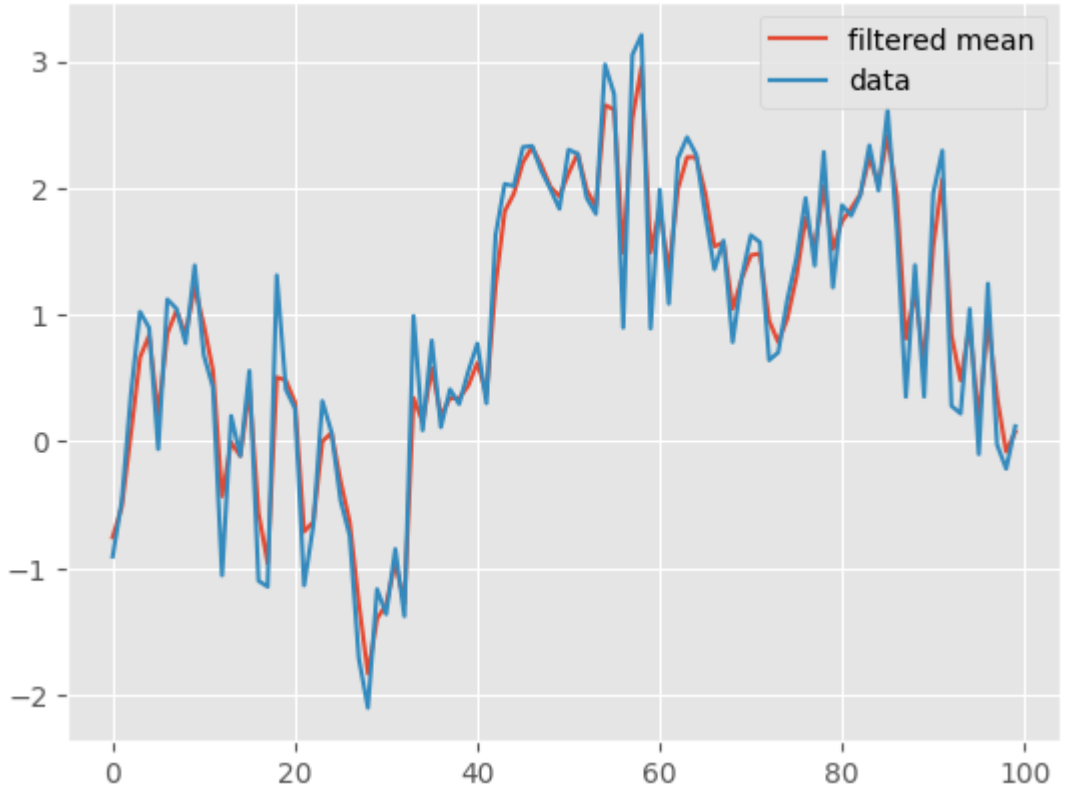
[14]:
```
from particles.collectors import Moments
```

(continues on next page)

(continued from previous page)

```
alg_with_mom = particles.SMC(fk=fk_guided, N=100, collect=[Moments()])
alg_with_mom.run()

plt.plot([m['mean'] for m in alg_with_mom.summaries.moments],
         label='filtered mean')
plt.plot(y, label='data')
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7ff95979a050>
```



Off-line smoothing

Off-line smoothing is the task of approximating, at some final time T (i.e. when we have stopped acquiring data), the distribution of all the states, $X_{0:T}$, given the full data, $Y_{0:T}$.

To run a particular off-line smoothing algorithm, one must first run a particle filter, and save its **history**:

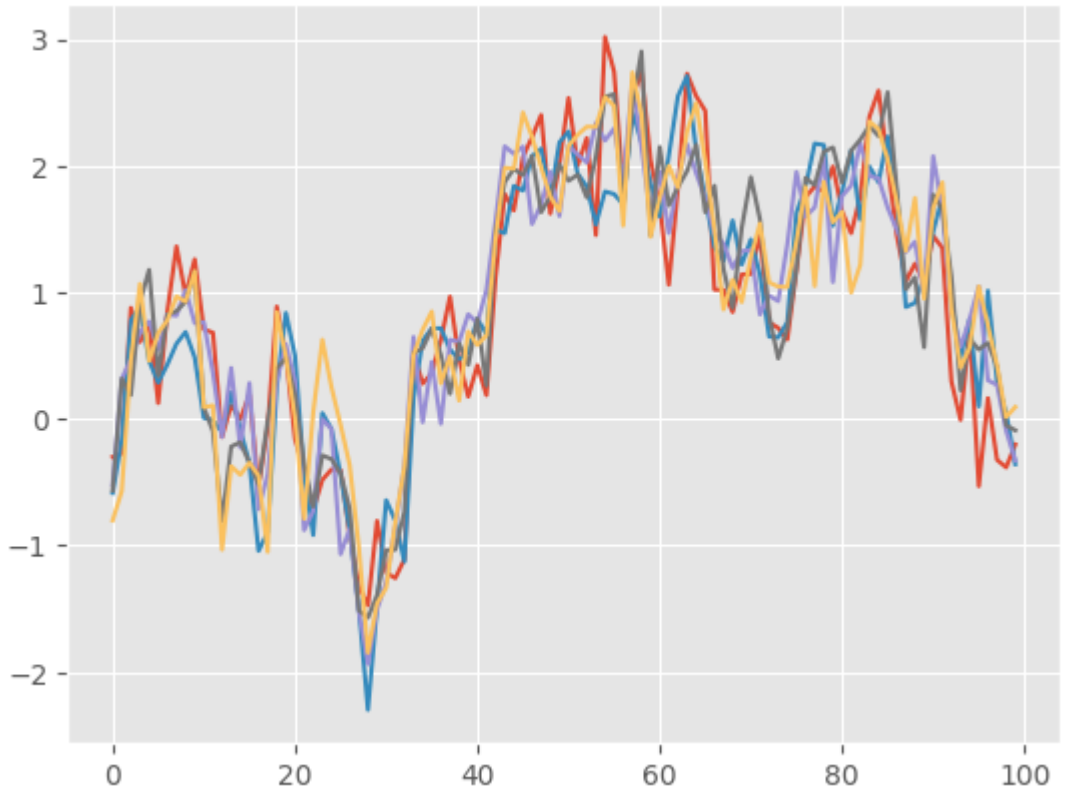
[15]:
```
alg = particles.SMC(fk=fk_guided, N=100, store_history=True)
alg.run()
```

Now `alg` has a `hist` attribute, which is a `ParticleHistory` object. Basically, `alg.hist` recorded, at each time t:

- the N particles X_t^n
- their weights W_t^n
- the N ancestor variables

Smoothing algorithms are implemented as methods of class `ParticleHistory`. For instance, the FFBS (forward filtering backward sampling) algorithm, which samples complete smoothing trajectories, may be called as follows:

[18]:
```
trajectories = alg.hist.backward_sampling_ON2(5)
plt.plot(trajectories);
```



The output of `backward_sampling_ON2` is a list of 100 arrays: `trajectories[t][m]` is the t-component of trajectory m. (If you want to turn it into a numpy array, simply do: `np.array(trajectories)`.)

Remark: `particles` implement several FFBS algorithms; `backward_sampling_ON2` implements the most basic one, which has complexity $\mathcal{O}(N^2)$ if you want to generate N trajectories (where N is the initial number of particles). There are faster variants, the recommended one, based on Dau & Chopin (2023), is `backward_sampling_mcmc`, which relies on MCMC. This is fast, and has a deterministic running time, $\mathcal{O}(N)$, contrary to some other rejection-based schemes (which are also implemented). To learn more about this, see again Dau & Chopin (2023).

Remark: Two-filter smoothing is also available. The difficulty with two-filter smoothing is that it requires to design an "information filter", that is a particle filter that computes recursively (backwards) the likelihood of the model. Since this is not trivial for the model considered here, we refer to Section 12.5 of the book and the documentation of package `smoothing`.

1.2.3 Bayesian inference for state-space models

Defining a prior distribution

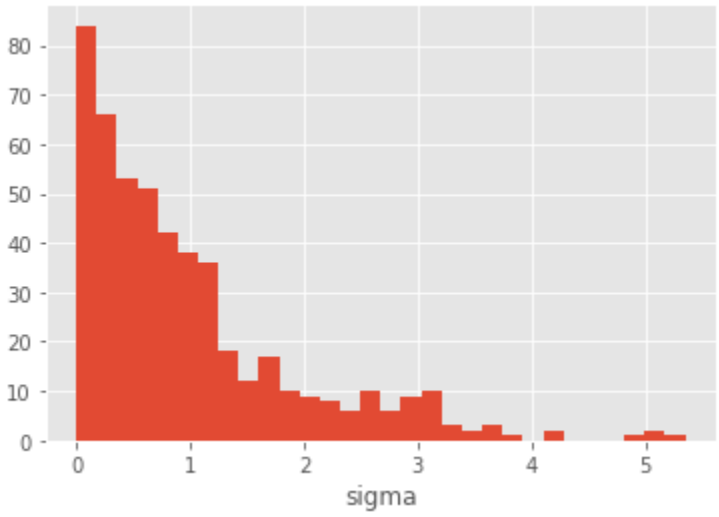
We have already seen that module `particles.distributions` defines various `ProbDist` objects; i.e. objects that represent probability distribution. Such objects have methods to simulate random variates, compute the log-density, and so on.

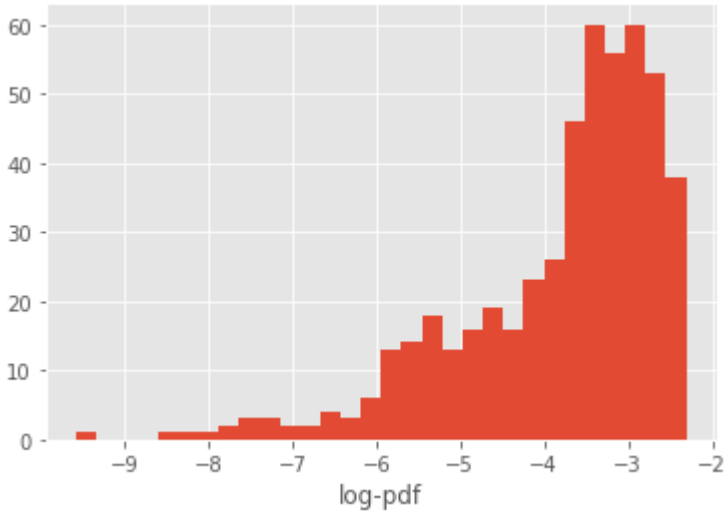
This module defines in particular a class called `StructDist`, whose methods take as inputs and outputs structured arrays. This is what we are going to use to define prior distributions. Here is a simple example:

[1]:
```
import warnings; warnings.simplefilter('ignore')  # hide warnings

from matplotlib import pyplot as plt
import numpy as np

from particles import distributions as dists

prior_dict = {'mu': dists.Normal(scale=2.),
              'rho': dists.Uniform(a=-1., b=1.),
              'sigma':dists.Gamma()}
my_prior = dists.StructDist(prior_dict)
```

Object `my_prior` represents a distribution for $\theta = (\mu, \rho, \sigma)$ where $\mu \sim N(0, 2^2)$, $\rho \sim \mathcal{U}([-1, 1])$, $\sigma \sim \text{Gamma}(1, 1)$, independently. We may now sample from this distribution, or compute its pdf, and so on. For each of the operations, the inputs and outputs must be structured arrays, with named variables `'rho'` and `'sigma'`.

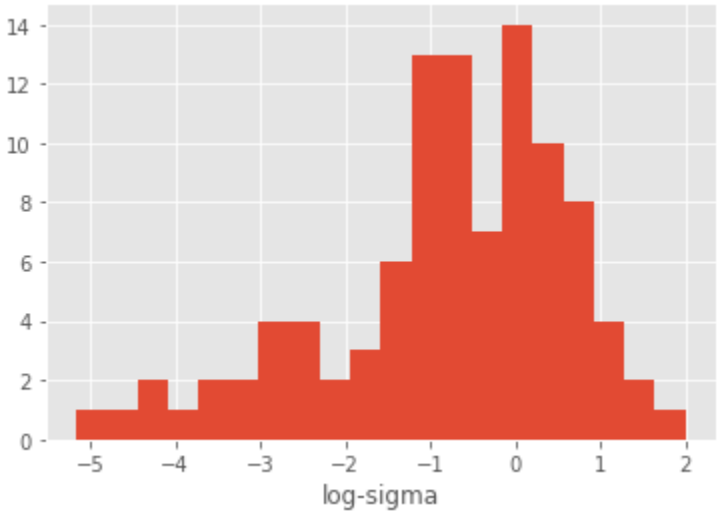
[2]:
```
theta = my_prior.rvs(size=500)  # sample 500 theta-parameters

plt.style.use('ggplot')
plt.hist(theta['sigma'], 30);
plt.xlabel('sigma')

plt.figure()
z = my_prior.logpdf(theta)
plt.hist(z, 30)
plt.xlabel('log-pdf');
```





We may want to transform `sigma` into its logarithm, so that the support of the distribution is not constrained to $\mathbb{R}^+$:

[3]:
```
another_prior_dict = {'rho': dists.Uniform(a=-1., b=1.),
                      'log_sigma':dists.LogD(dists.Gamma())}
another_prior = dists.StructDist(another_prior_dict)
another_theta = another_prior.rvs(size=100)

plt.hist(another_theta['log_sigma'], 20)
plt.xlabel('log-sigma');
```



Now, `another_theta` contains two variables, `rho` and `log_sigma`, and the latter variable is distributed according to $Y = \log(X)$, with $X \sim \text{Gamma}(1, 1)$. (The documentation of module `distributions` has more details on transformed distributions.)

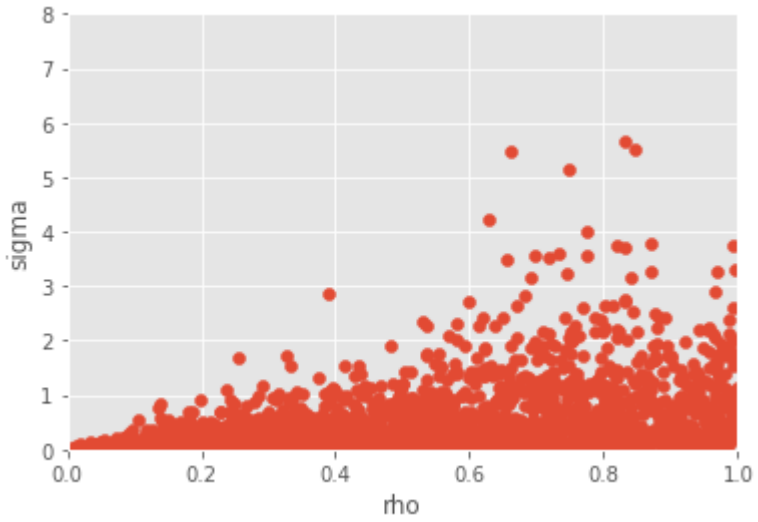
We may also want to introduce dependencies between ρ and σ. Consider this:

[4]:
```
from collections import OrderedDict
```

(continues on next page)

(continued from previous page)

```
dep_prior_dict = OrderedDict()
dep_prior_dict['rho'] = dists.Uniform(a=0., b=1.)
dep_prior_dict['sigma'] = dists.Cond( lambda theta: dists.Gamma(b=1./theta['rho']))
dep_prior = dists.StructDist(dep_prior_dict)
dep_theta = dep_prior.rvs(size=2000)

plt.scatter(dep_theta['rho'], dep_theta['sigma'])
plt.axis([0., 1., 0., 8.])
plt.xlabel('rho')
plt.ylabel('sigma');
```



The lines above encodes a **chain rule** decomposition: first we specify the marginal distribution of ρ, then we specify the distribution of σ given ρ. A standard dictionary in Python is unordered: there is no way to make sure that the keys appear in a certain order. Thus we use instead an `OrderedDict`, and define first the distribution of ρ, then the distribution of σ given ρ; `Cond` is a particular `ProbDist` class that defines a conditional distribution, based on a function that takes an argument `theta`, and returns a `ProbDist` object.

All the example above involve univariate distributions; however, the components of `StructDist` also accept multivariate distributions.

[5]:
```
reg_prior_dict = OrderedDict()
reg_prior_dict['sigma2'] = dists.InvGamma(a=2., b=3.)
reg_prior_dict['beta'] = dists.MvNormal(cov=np.eye(20))
reg_prior = dists.StructDist(reg_prior_dict)
reg_theta = reg_prior.rvs(size=200)
```

Bayesian inference for state-space models

We return to the simplified stochastic volatility introduced in the basic tutorial:

$$
\begin{aligned}
X_0 &\sim N\left(\mu, \frac{\sigma^2}{1-\rho^2}\right) \\
X_t | X_{t-1} = x_{t-1} &\sim N\left(\mu + \rho(x_{t-1} - \mu), \sigma^2\right) \\
Y_t | X_t = x_t &\sim N\left(0, e^{x_t}\right)
\end{aligned}
$$

which we implemented as follows (this time with default values for the parameters):

[6]:
```
from particles import state_space_models as ssm

class StochVol(ssm.StateSpaceModel):
    default_parameters = {'mu':-1., 'rho':0.95, 'sigma': 0.2}
    def PX0(self):  # Distribution of X_0
        return dists.Normal(loc=self.mu, scale=self.sigma / np.sqrt(1. - self.rho**2))
    def PX(self, t, xp):  # Distribution of X_t given X_{t-1}=xp (p=past)
        return dists.Normal(loc=self.mu + self.rho * (xp - self.mu), scale=self.sigma)
    def PY(self, t, xp, x):  # Distribution of Y_t given X_t=x (and possibly X_{t-1}=xp)
        return dists.Normal(loc=0., scale=np.exp(x))
```

We mentioned in the basic tutorial that `StochVol` represents the parameteric class of univariate stochastic volatility model. Indeed, `StochVol` will be the object we pass to Bayesian inference algorithms (such as PMMH or SMC^2) in order to perform inference with respect to that class of models.

PMMH (Particle marginal Metropolis-Hastings)

Let's try first PMMH. This is a Metropolis-Hastings algorithm that samples from the posterior of parameter θ (given the data). However, since the corresponding likelihood is intractable, each iteration of PMMH runs a particle filter that approximates it.

[9]:
```
from particles import mcmc
from particles import datasets as dts  # real datasets available in the package

# real data
T = 50
data = dts.GBP_vs_USD_9798().data[:T]

my_pmmh = mcmc.PMMH(ssm_cls=StochVol, prior=my_prior, data=data, Nx=200,
                    niter=1000)
my_pmmh.run();  # may take several seconds...
```

The arguments we set when instantiating class `PMMH` requires little explanation; just in case:

- `Nx` is the number of particles (for the particle filter run at each iteration);
- `niter` is the number of MCMC iterations.

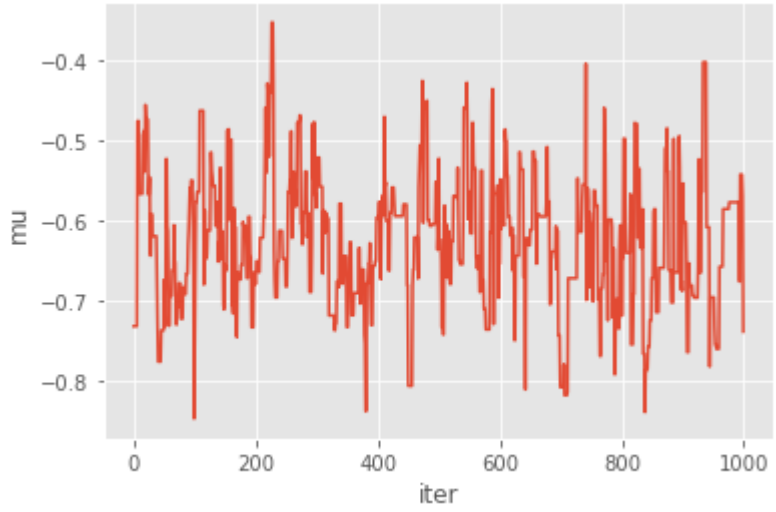
Upon completion, object `my_pmmh.chain` is a `ThetaParticles` object, with the following attributes:

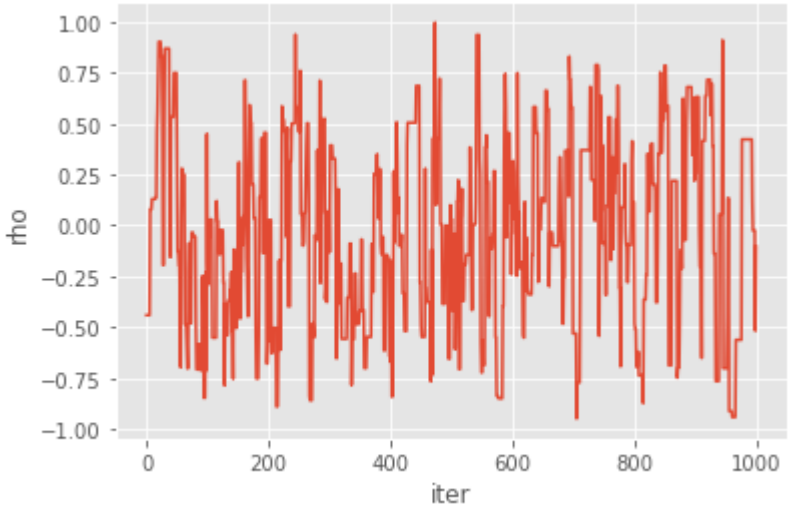
- `my_pmmh.chain.theta` is a structured array of size 10 (the number of iterations) with keys `'mu'`, `'rho'` and `'sigma'`;

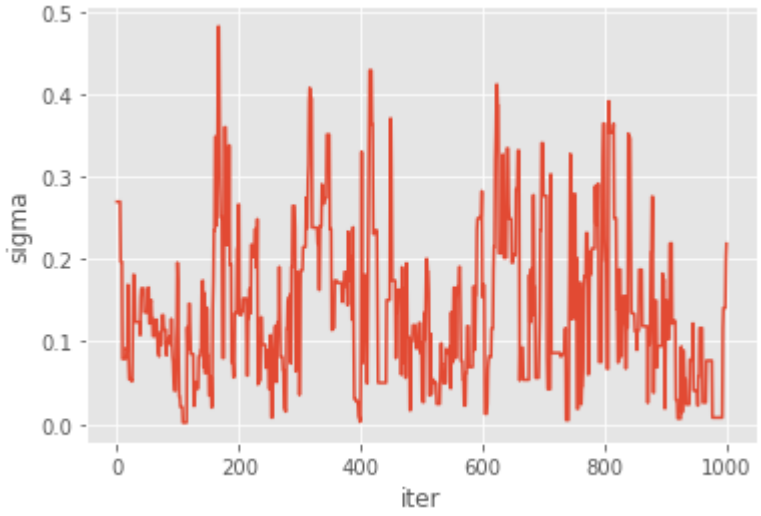
- `my_pmmh.chain.lpost` is an array of length 10, contanining the (estimated) log-posterior density for each simulated θ.

Let's plot the mcmc traces.

[10]:
```
for p in prior_dict.keys():  # loop over mu, theta, rho
    plt.figure()
    plt.plot(my_pmmh.chain.theta[p])
    plt.xlabel('iter')
    plt.ylabel(p)
```







You might wonder what type of Metropolis sampler is really implemented here:

- the starting point of the chain is sampled from the prior; you may instead set it to a specific value using option `starting_point` (when instantiating `PMMH`);
- the proposal is an **adaptative** Gaussian random walk: this means that the covariance matrix of the random step is calibrated on the fly on past simulations (using vanishing adaptation). This may be disabled by setting option `adaptive=False`;
- a bootstrap filter is run to approximate the log-likelihood; you may use a different filter (e.g. a guided filter) by passing a `FeynmanKac` class to option `fk_cls`;
- you may also want to pass various parameters to each call to `SMC` through (dict) argument `smc_options`; e.g. `smc_options={'qmc': True}` will make each particle filter a SQMC algorithm.

Thus, by and large, quite a lot of flexibility is hidden behind this default behaviour.

Particle Gibbs

PMMH is just a particular instance of the general family of PMCMC samplers; that is MCMC samplers that run some particle filter at each iteration. Another instance is Particle Gibbs (PG), where one simulate alternatively: 1. from the distribution of θ given the states and the data; 2. renew the state trajectory through a CSMC (conditional SMC step).

Since Step 1 is model- (and user-)dependent, you need to define it for the model you are considering. This is done by sub-classing `ParticleGibbs` and defining method `update_theta` as follows:

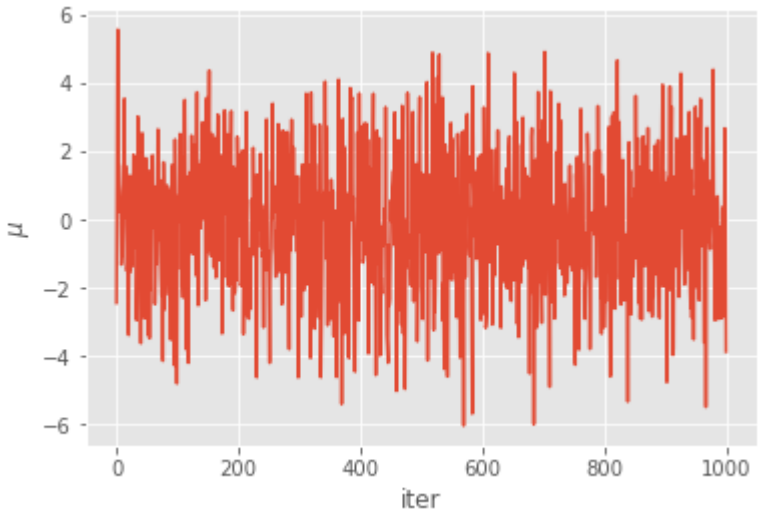
[11]:
```
class PGStochVol(mcmc.ParticleGibbs):
    def update_theta(self, theta, x):
        new_theta = theta.copy()
        sigma, rho = 0.2, 0.95  # fixed values
        xlag = np.array(x[1:] + [0.,])
        dx = (x - rho * xlag) / (1. - rho)
        s = sigma / (1. - rho)**2
        new_theta['mu'] = self.prior.laws['mu'].posterior(dx, sigma=s).rvs()
        return new_theta
```

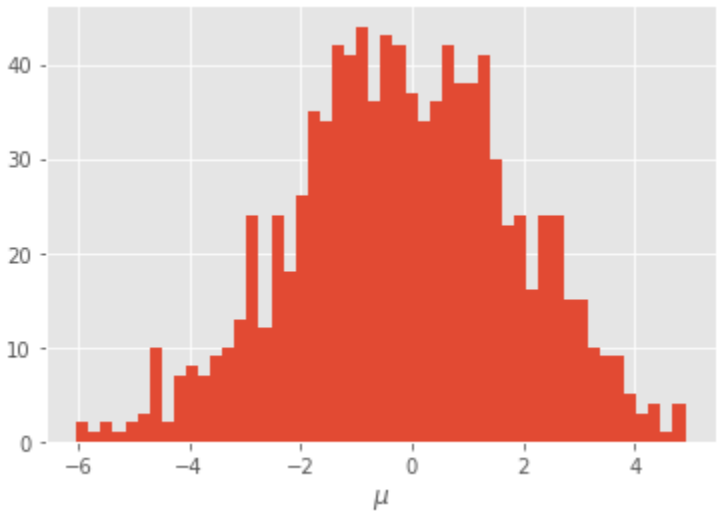
For simplicity ρ and σ are kept constant; only μ is updated. This means we are actually sampling from the posterior of μ given the data, while these other parameters are kept constant. Let's run our PG algorithm:

[12]:
```
pg = PGStochVol(ssm_cls=StochVol, data=data, prior=my_prior, Nx=200, niter=1000)
pg.run()  # may take several seconds...
```

Now let's plot the results:

[16]:
```
plt.plot(pg.chain.theta['mu'])
plt.xlabel('iter')
plt.ylabel(r'$\mu$')

plt.figure()
plt.hist(pg.chain.theta['mu'][20:], 50)
plt.xlabel(r'$\mu$');
```





SMC^2

Finally, we consider SMC^2, a SMC algorithm that makes it possible to approximate:

- all the partial posteriors (of θ given $y_{0:t}$, for $t = 0, 1, ..., T$) rather than only the final posterior;
- the marginal likelihoods of the data.

SMC^2 is a two-level SMC sampler:

1. it simulates many θ-values from the prior, and update their weights recursively, according to the likelihood of each new datapoint;
2. however, since these likelihood factors are intractable, for each θ, a particle filter is run to approximate it; hence a number N_x of x-particles are generated for, and attached to, each θ.

The class `SMC2` is defined inside module `smc_samplers`. It is run in the same way as the other SMC algorithms.

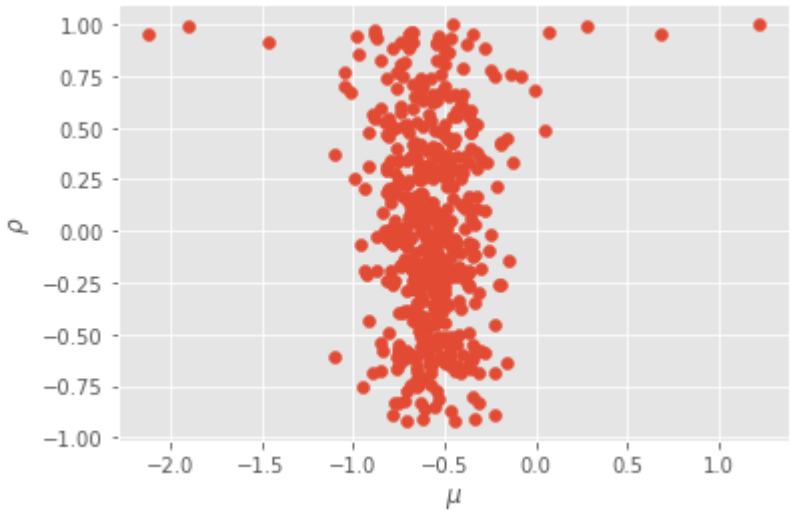
[17]:
```
import particles
from particles import smc_samplers as ssp

fk_smc2 = ssp.SMC2(ssm_cls=StochVol, data=data, prior=my_prior,init_Nx=50,
                   ar_to_increase_Nx=0.1)
alg_smc2 = particles.SMC(fk=fk_smc2, N=500)
alg_smc2.run()
```

Again, a few choices are made for you by default:

- A bootstrap filter is run for each $\theta-$particle; this may be changed by setting option `fk_class` while instantiating `SMC2`; e.g. `fk_class=ssm.GuidedPF` will run instead a guided filter.
- Option `init_Nx` determines the **initial** number of $x-$particles; the algorithm automatically increases N_x each time the acceptance rate drops below 10

- The particle filters (in the $x-$dimension) are run with the default options of class `SMC`; e.g. resampling is set to `systematic` and so on; other options may be set by using option `smc_options`.

[19]:
```
plt.scatter(alg_smc2.X.theta['mu'], alg_smc2.X.theta['rho'])
plt.xlabel(r'$\mu$')
plt.ylabel(r'$\rho$');
```



1.2.4 SMC samplers

This tutorial gives a basic introduction to SMC samplers, and explains how to run the SMC samplers already implemented in `particles`. For a more advanced tutorial on how to design new SMC samplers, see the next tutorial. For more background on SMC samplers, check either Chapter 17 of the book or Dau & Chopin (2022) for waste-free SMC. Arxiv version is here.

SMC samplers: what for?

A SMC sampler is a SMC algorithm that samples from a sequence of probability distributions π_t, $t = 0, \ldots, T$ (and compute their normalising constants). Sometimes one is genuinely interested in each π_t; more often one is interested only in the final distribution π_T. In the latter case, the sequence is purely instrumental.

Examples of SMC sequences are:

1. $\pi_t(\theta) = p(\theta|y_{0:t})$, the Bayesian posterior distribution of parameter θ given data $y_{0:t}$, for a certain model.
2. A tempering sequence, $\pi_t(\theta) \propto \nu(\theta)L(\theta)^{\gamma_t}$,where the γ_t's form an increasing sequence of exponents: $0 = \gamma_0 < \ldots < \gamma_T = 1$. You can think of ν being the prior, L the likelihood function, and π_T the posterior. However, more generally, tempering is a way to interpolate between any two distributions, ν and π, with $\pi(\theta) \propto \nu(\theta)L(\theta)$.

We discuss first how to specify a sequence of the first type.

Defining a Bayesian model

To define a particular Bayesian model, you must subclass `StaticModel`, and define method `logpyt`, which evaluates the log-likelihood of datapoint Y_t given parameter θ and past datapoints $Y_{0:t-1}$. Here is a simple example:

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt
import seaborn as sb
import numpy as np
from scipy import stats

import particles
from particles import smc_samplers as ssp
from particles import distributions as dists

class ToyModel(ssp.StaticModel):
    def logpyt(self, theta, t):  # density of Y_t given theta and Y_{0:t-1}
        return stats.norm.logpdf(self.data[t], loc=theta['mu'],
                                 scale = theta['sigma'])
```

In words, we are considering a model where the observations are $Y_t \sim N(\mu, \sigma^2)$ (independently). The parameter is $\theta = (\mu, \sigma)$. Note the fields notation; more about this later.

Class `ToyModel` implicitly defines the likelihood of the considered model for any sample size (since the likelihood at time t is $p^\theta(y_{0:t}) = \prod_{s=0}^{t} p^\theta(y_s|y_{0:s-1})$, and method `logpyt` defines each factor in this product; note that y_s does not depend on the past values in our particular example). We now define the data and the prior:

[2]:
```
T = 30
my_data = stats.norm.rvs(loc=3.14, size=T)  # simulated data
my_prior = dists.StructDist({'mu': dists.Normal(scale=10.),
                             'sigma': dists.Gamma()})
```

For more details about to define prior distributions, see the documentation of module `distributions`, or the previous *tutorial on Bayesian estimation of state-space models*. Now that we have everything, let's specify our static model:

[3]:
```
my_static_model = ToyModel(data=my_data, prior=my_prior)
```

This time, object `my_static_model` entirely defines the posterior.

[4]:
```
thetas = my_prior.rvs(size=5)
my_static_model.logpost(thetas, t=2)
# if t is omitted, gives the full posterior
```

[4]:
```
array([-2.37897769e+02, -8.61877866e+01, -4.94969347e+02, -2.32511323e+05,
       -2.63076061e+02])
```

The input of `logpost` and output of `myprior.rvs()` are structured arrays, that is, arrays with fields:

[5]:
```
thetas['mu'][0]
```

[5]:
```
-4.241375242422195
```

Typically, you won't need to call `logpost` yourself, this will be done by the SMC sampler for you.

IBIS

IBIS (iterated batch importance sampling) is the standard name for a SMC sampler that tracks a sequence of partial posterior distributions; i.e. π_t is $p(\theta|y_{0:t})$, for $t = 0, 1, \ldots$.

Module `smc_samplers` defines `IBIS` as a subclass of `FeynmanKac`.

[6]:
```
my_ibis = ssp.IBIS(my_static_model, len_chain=50)
my_alg = particles.SMC(fk=my_ibis, N=20,
                       store_history=True, verbose=True)
my_alg.run()
```

```
t=0, ESS=30.38
t=1, Metropolis acc. rate (over 49 steps): 0.179, ESS=320.47
t=2, Metropolis acc. rate (over 49 steps): 0.265, ESS=739.71
t=3, ESS=495.11
t=4, Metropolis acc. rate (over 49 steps): 0.239, ESS=699.01
t=5, ESS=362.60
t=6, Metropolis acc. rate (over 49 steps): 0.357, ESS=855.81
t=7, ESS=563.24
t=8, ESS=429.52
t=9, Metropolis acc. rate (over 49 steps): 0.348, ESS=944.20
t=10, ESS=821.65
t=11, ESS=672.69
t=12, ESS=600.56
t=13, ESS=512.21
t=14, ESS=812.56
t=15, ESS=738.51
t=16, ESS=675.90
t=17, ESS=618.24
t=18, ESS=558.24
t=19, ESS=482.73
t=20, Metropolis acc. rate (over 49 steps): 0.349, ESS=972.66
```

(continues on next page)

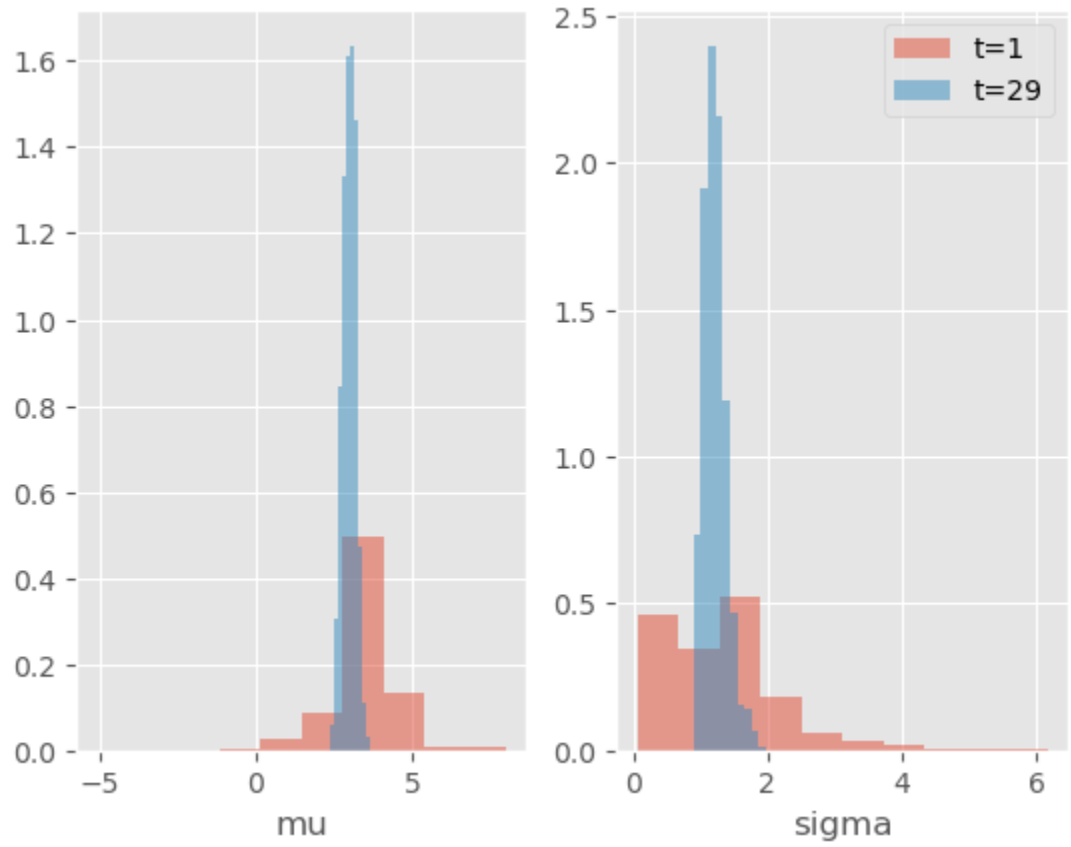
(continued from previous page)

```
t=21, ESS=925.96
t=22, ESS=889.08
t=23, ESS=970.02
t=24, ESS=950.08
t=25, ESS=732.06
t=26, ESS=714.42
t=27, ESS=453.15
t=28, Metropolis acc. rate (over 49 steps): 0.316, ESS=978.24
t=29, ESS=951.31
```

Note: we use option `verbose=True` in `SMC` in order to print some information on the intermediate distributions.

Note: Since we set `store_history` to `True`, the particles and their weights have been saved at every time (in attribute `hist`, see previous tutorials on smoothing). Let's plot the posterior distributions of μ and σ at various times.

[7]:
```
plt.style.use('ggplot')
for i, p in enumerate(['mu', 'sigma']):
    plt.subplot(1, 2, i + 1)
    for t in [1, 29]:
        plt.hist(my_alg.hist.X[t].theta[p], weights=my_alg.hist.wgts[t].W, label="t=%i"
↪% t,
                 alpha=0.5, density=True)
    plt.xlabel(p)
plt.legend();
```



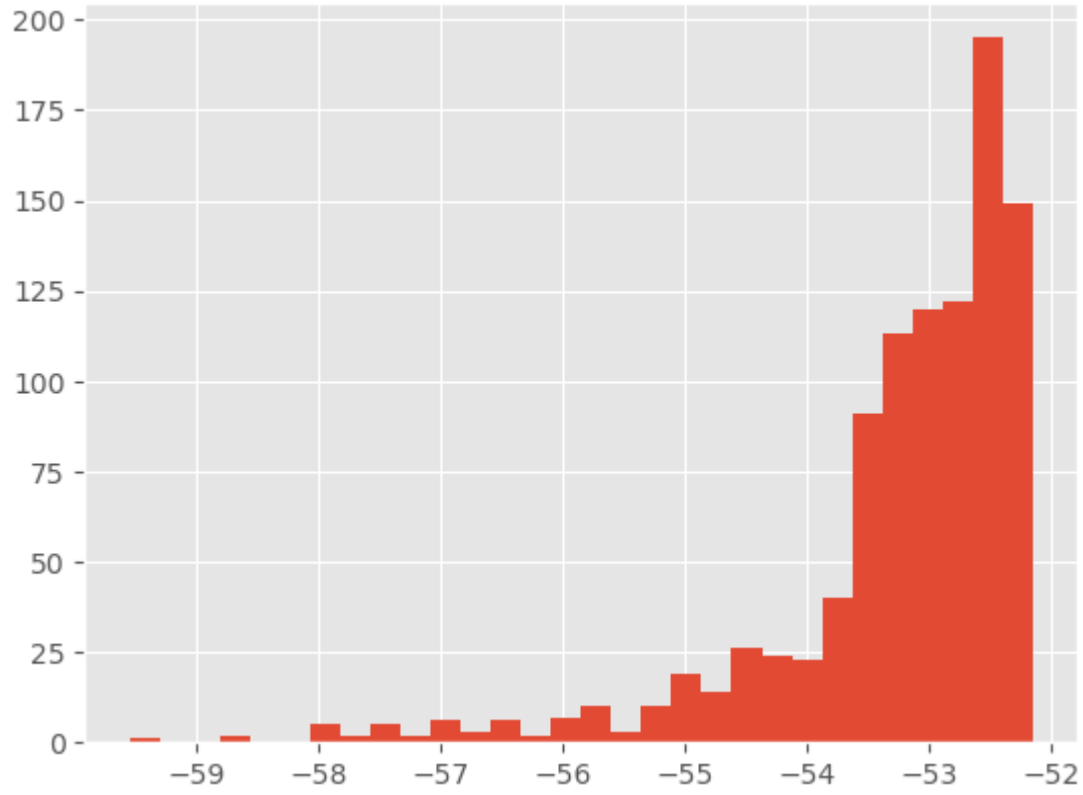
As expected, the posterior distribution concentrates progressively around the true values.

As always, once the algorithm is run, `my_smc.X` contains the final particles. However, object `my_smc.X` is no longer a simple numpy array. It is a `ThetaParticles` object, with attributes:

- `theta`: a structured array (an array with fields); i.e. `my_smc.X.theta['mu']` is a (N,) array that contains the the $\mu-$component of the N particles;
- `lpost`: a 1D numpy array that contains the target (posterior) log-density of each of the particles;
- `shared`: a dictionary that contains "meta-data" on the particles; for instance `shared['acc_rates']` is a list of the acceptance rates of the successive Metropolis steps.

[8]:
```
print(["%2.f%%" % (100 * np.mean(r)) for r in my_alg.X.shared['acc_rates']])
plt.hist(my_alg.X.lpost, 30);
```

```
['18%', '26%', '24%', '36%', '35%', '35%', '32%']
```



You do not need to know much more about class `ThetaParticles` in practice (if you're curious, however, see the next tutorial on SMC samplers or the documentation of module `smc_samplers`).

Waste-free versus standard SMC samplers

The library now implements by default waste-free SMC (Dau & Chopin, 2020), a variant of SMC samplers that keeps all the intermediate Markov steps (rather than "wasting" them). In practice, this means that, in the piece of code above:

- at each time t, $N = 20$ particles are resampled, and used as a starting points of the MCMC chains;
- the MCMC chains are run for 49 iterations, hence the chain length is 50 (parameter `len_chain=50`)
- and since we keep all the intermediate steps, we get 50*20 = 1000 particles at each iteration. In particular, we do O(1000) operations at each step. (At time 0, we also generate 1000 particles.)

Thus, the number of particles is actually `N * len_chain`; given this number of particles, the performance typically does not depend too much on `N` and `len_chain`, provided the latter is "big enough" (relative to the mixing of the MCMC kernels).

See Dau & Chopin (2020) for more details on waste-free SMC. If you wish to run a standard SMC sampler instead, you may set `wastefree=False`, like this:

[9]:
```
my_ibis = ssp.IBIS(my_static_model, wastefree=False, len_chain=11)
my_alg = particles.SMC(fk=my_ibis, N=100, store_history=True)
my_alg.run()
```

This runs a standard SMC sampler which tracks $N = 100$ particles; these particles are resampled from time to time, and then moved through 10 MCMC steps. (As explained in Dau & Chopin, 2020, you typically get a better performance vs CPU time trade-off with wastefree SMC.)

Single-run variance estimates

An advantage of waste-free SMC is that it gives you the possibility to estimate the (asymptotic) variance of a given estimate from a single run. Here is a quick example, but check also the documentation of `smc_samplers.var_wf` and the collectors `smc_samplers.Var_logLt`, `smc_samplers.Var_phi` for more details.

[10]:
```
phi = lambda x : x.theta['mu']  # scalar function
est = np.average(phi(my_alg.X), weights=my_alg.W)
var_est = ssp.var_wf(my_alg, phi)
print(f'estimate of E(mu): {est}')
print(f'asymptotic variance of the above estimate: {var_est}')
print(f'95% confidence interval for E[mu] {est} +-{1.96 * np.sqrt(var_est / 1000)}')
```

```
estimate of E(mu): 2.9956255516306944
asymptotic variance of the above estimate: 0.03999515708204
95% confidence interval for E[mu] 2.9956255516306944 +-0.012395377987232371
```

The confidence interval accounts for the *Monte Carlo error* (not the statistical error, which you would measure through the posterior variance). Classically, you must divide the asymptotic variance by the number of particles to get an estimate of the actual variance.

Regarding the MCMC steps

The default MCMC kernel used to move the particles is a Gaussian random walk Metropolis kernel, whose covariance matrix is calibrated automatically to γ times of the empirical covariance matrix of the particle sample, where $\gamma = 2.38/\sqrt{d}$ (standard choice in the literature).

It is possible to specify a different value for γ, or more generally other types of MCMC moves; for instance the following uses Metropolis kernels based on independent Gaussian proposals:

[11]:
```
mcmc = ssp.ArrayIndependentMetropolis(scale=1.1)
# Independent Gaussian proposal, with mean and variance determined by
# the particle sample (variance inflated by factor scale=1.1)
alt_move = ssp.MCMCSequenceWF(mcmc=mcmc)
# This object represents a particular way to apply several MCMC steps
# in a row. WF = WasteFree
alt_ibis = ssp.IBIS(my_static_model, move=alt_move)
alt_alg = particles.SMC(fk=alt_ibis, N=100,ESSrmin=0.2,
```

(continues on next page)

(continued from previous page)

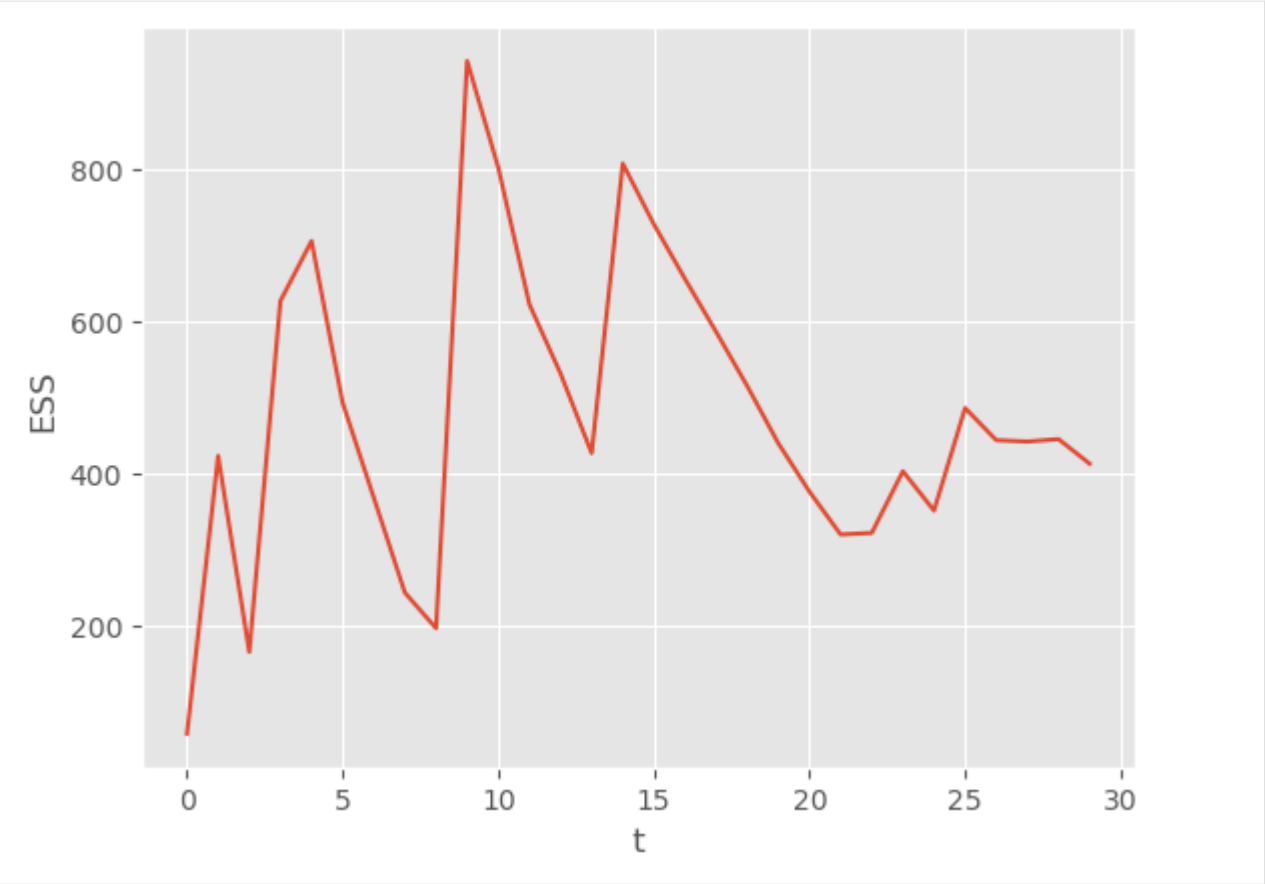
```
                          verbose=True)
alt_alg.run()
```

```
t=0, ESS=58.32
t=1, Metropolis acc. rate (over 9 steps): 0.383, ESS=423.19
t=2, ESS=165.63
t=3, Metropolis acc. rate (over 9 steps): 0.526, ESS=626.73
t=4, ESS=705.49
t=5, ESS=493.02
t=6, ESS=368.40
t=7, ESS=243.54
t=8, ESS=196.84
t=9, Metropolis acc. rate (over 9 steps): 0.748, ESS=942.05
t=10, ESS=801.91
t=11, ESS=622.29
t=12, ESS=531.48
t=13, ESS=426.85
t=14, ESS=807.19
t=15, ESS=727.41
t=16, ESS=655.57
t=17, ESS=586.12
t=18, ESS=514.90
t=19, ESS=439.62
t=20, ESS=376.15
t=21, ESS=319.90
t=22, ESS=322.03
t=23, ESS=403.06
t=24, ESS=351.35
t=25, ESS=486.05
t=26, ESS=443.87
t=27, ESS=442.08
t=28, ESS=445.23
t=29, ESS=412.93
```

In the future, the package may also implement other type of MCMC kernels such as MALA. It is also possible to define your own MCMC kernels, as explained in the next tutorial.

For now, note the following practical detail: the algorithm resamples whenever the ESS gets below a certain threshold $\alpha * N$; the default value $\alpha = 0.5$, but here we changed it (to $\alpha = 0.2$) by setting `ESSrmin=0.2`.

[12]:
```
plt.plot(alt_alg.summaries.ESSs)
plt.xlabel('t')
plt.ylabel('ESS');
```



As expected, the algorithm waits until the ESS is below 200 to trigger a resample-move step.

SMC tempering

SMC tempering is a SMC sampler that samples iteratively from the following sequence of distributions:

$$\pi_t(\theta) \propto \pi(\theta) L(\theta)_t^{\gamma} \tag{1.1}$$

with $0 = \gamma_0 < \ldots < \gamma_T = 1$. In words, this sequence is a **geometric bridge**, which interpolates between the prior and the posterior.

SMC tempering implemented in the same was as IBIS: as a sub-class of `FeynmanKac`, whose `__init__` function takes as argument a `StaticModel` object.

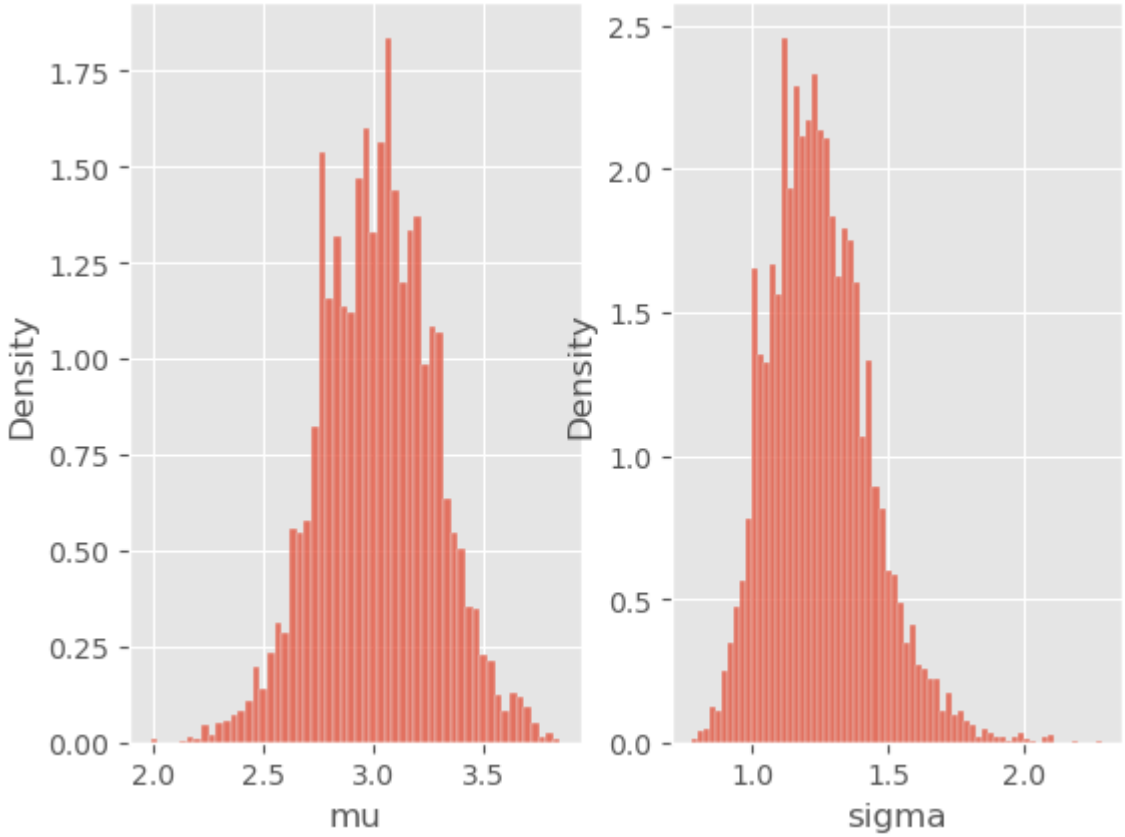
[13]:
```
fk_tempering = ssp.AdaptiveTempering(my_static_model)
my_temp_alg = particles.SMC(fk=fk_tempering, N=1000, ESSrmin=1.,
                            verbose=True)
my_temp_alg.run()
```

```
t=0, ESS=5000.00, tempering exponent=0.000798
t=1, Metropolis acc. rate (over 9 steps): 0.270, ESS=5000.00, tempering exponent=0.0111
t=2, Metropolis acc. rate (over 9 steps): 0.245, ESS=5000.00, tempering exponent=0.062
t=3, Metropolis acc. rate (over 9 steps): 0.253, ESS=5000.00, tempering exponent=0.227
t=4, Metropolis acc. rate (over 9 steps): 0.291, ESS=5000.00, tempering exponent=0.788
t=5, Metropolis acc. rate (over 9 steps): 0.333, ESS=9573.62, tempering exponent=1
```

Note: Recall that SMC resamples every time the ESS drops below value N times option ESSrmin; here we set it to to 1, since we want to resample at every time. This makes sense: Adaptive SMC chooses adaptively the successive values of γ_t so that the ESS equals a certain value ($N/2$ by default).

We have not saved the intermediate results this time (option store_history was not set) since they are not particularly interesting. Let's look at the final results:

[14]:
```
for i, p in enumerate(['mu', 'sigma']):
    plt.subplot(1, 2, i + 1)
    sb.histplot(my_temp_alg.X.theta[p], stat='density')
    plt.xlabel(p)
```



This looks reasonable! You can see from the output that the algorithm automatically chooses the tempering exponents $\gamma_1, \gamma_2, \ldots$. In fact, at iteration t, the next value for γ is set that the ESS drops at most to $N/2$. You can change this particular threshold by passing argument ESSrmin to TemperingSMC. (Warning: do not mistake this with the ESSrmin argument of class SMC):

[15]:
```
lazy_tempering = ssp.AdaptiveTempering(my_static_model, ESSrmin = 0.1)
lazy_alg = particles.SMC(fk=lazy_tempering, N=1000, verbose=True)
lazy_alg.run()
```

```
t=0, ESS=1000.00, tempering exponent=0.037
t=1, Metropolis acc. rate (over 9 steps): 0.246, ESS=1000.00, tempering exponent=0.79
t=2, Metropolis acc. rate (over 9 steps): 0.335, ESS=9537.70, tempering exponent=1
```

The algorithm progresses faster this time, but the ESS drops more between each step. Another optional argument for Class TemperingSMC is options_mh, which works exactly as for IBIS, see above. That is, by default, the particles are moved according to a certain (adaptative) number of random walk steps, with a variance calibrated to the particle

variance.

Again, all the indications above should be sufficient if you simply want to run a SMC sampler with random-walk Metropolis steps. If you are interested in more exotic SMC samplers (based on different MCMC kernels, or such that the state-space is not $\mathbb{R}^d$), see the next tutorial on SMC samplers.

1.2.5 Manual definition of Feynman-Kac models

It is not particularly difficult to define manually your own `FeynmanKac` classes. Consider the following problem: we would like to approximate the probability that $X_t \in [a, b]$ for all $0 \leq t < T$, where (X_t) is a random walk: $X_0 \sim N(0, 1)$, and

$$X_t | X_{t-1} = x_{t-1} \sim N(x_{t-1}, 1).$$

This probability, at time t, equals L_t, the normalising constant of the following Feynman-Kac sequence of distributions:

$$\mathbb{Q}_t(dx_{0:t}) = \frac{1}{L_t} M_0(dx_0) \prod_{s=1}^{t} M_s(x_{s-1}, dx_s) \prod_{s=0}^{t} G_s(x_{s-1}, x_s) \tag{1.2}$$

where:

- $M_0(dx_0)$ is the $N(0, 1)$ distribution;
- $M_s(x_{s-1}, dx_s)$ is the $N(x_{s-1}, 1)$ distribution;
- $G_s(x_{s-1}, x_s) = \mathbb{1}_{[0,\epsilon]}(x_s)$

Let's define the corresponding `FeymanKac` object:

[1]:
```
from matplotlib import pyplot as plt
import seaborn as sb
import numpy as np
from scipy import stats

import particles

class GaussianProb(particles.FeynmanKac):
    def __init__(self, a=0., b=1., T=10):
        self.a, self.b, self.T = a, b, T

    def M0(self, N):
        return stats.norm.rvs(size=N)

    def M(self, t, xp):
        return stats.norm.rvs(loc=xp, size=xp.shape)

    def logG(self, t, xp, x):
        return np.where((x < self.b) & (x > self.a), 0., -np.inf)
```

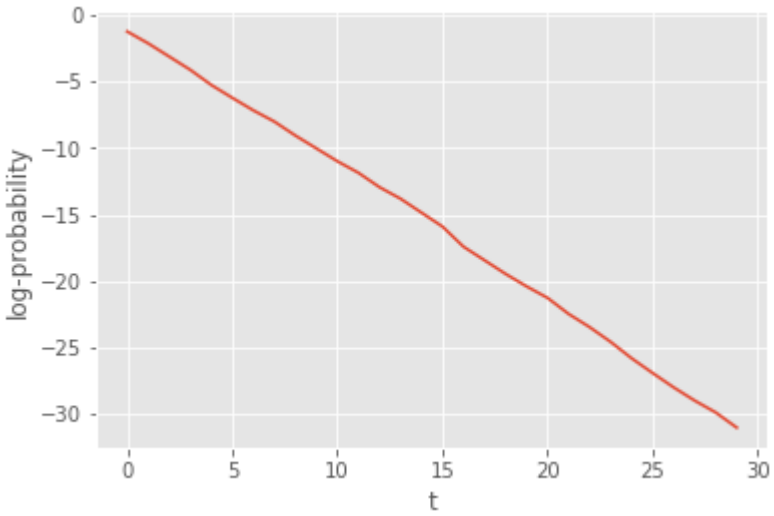
The class above defines:

- the initial distribution, $M_0(dx_0)$ and the kernels $M_t(x_{t-1}, dx_t)$, through methods `M0(self, N)` and `M(self, t, xp)`. In fact, these methods simulate N random variables from the corresponding distributions.
- Function `logG(self, t, xp, x)` returns the log of function $G_t(x_{t-1}, x_t)$.

Methods `M0` and `M` also define implicitly how the N particles should be represented internally: as a (N,) numpy array. Indeed, at time 0, method `M0` generates a (N,) numpy array, and at times $t \geq 1$, method `M` takes as an input (`xp`) and returns as an output arrays of shape (N,). We could use another type of object to represent our N particles; for instance, the `smc_samplers` module defines a `ThetaParticles` class for storing N particles representing N parameter values (and associated information).

Now let's run the corresponding SMC algorithm:

[2]:
```
fk_gp = GaussianProb(a=0., b=1., T=30)
alg = particles.SMC(fk=fk_gp, N=100)
alg.run()

plt.style.use('ggplot')
plt.plot(alg.summaries.logLts)
plt.xlabel('t')
plt.ylabel(r'log-probability');
```



That was not so hard. However our implementation suffers from several limitations:

1. The SMC sampler we ran may be quite inefficient when interval $[a, b]$ is small; in that case many particles should get a zero weight at each iteration.
2. We cannot currently run the SQMC algorithm (the quasi Monte Carlo version of SMC); to do so, we need to specify the Markov kernels M_t in a different way: not as simulators, but as deterministic functions that take as inputs uniform variates (see below).

Let's address the second point:

[3]:
```
class GaussianProb(particles.FeynmanKac):
    du = 1  # dimension of uniform variates

    def __init__(self, a=0., b=1., T=10):
        self.a, self.b, self.T = a, b, T

    def M0(self, N):
        return stats.norm.rvs(size=N)

    def M(self, t, xp):
```

(continues on next page)

(continued from previous page)

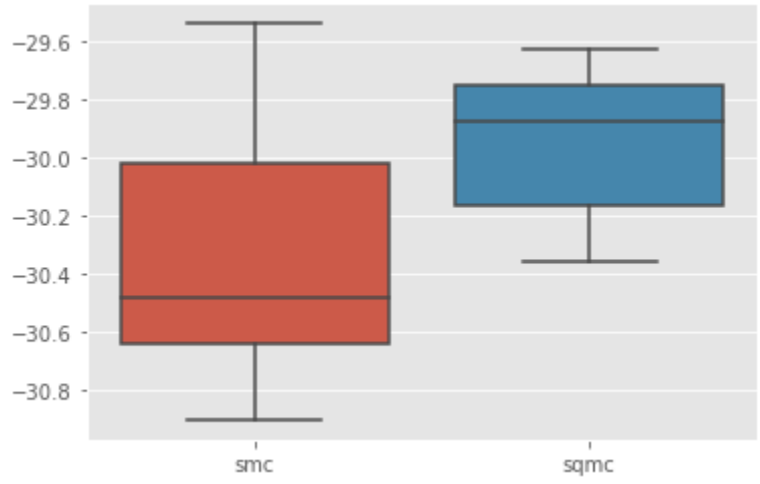
```
        return stats.norm.rvs(loc=xp, size=xp.shape)

    def Gamma0(self, u):
        return stats.norm.ppf(u)

    def Gamma(self, t, xp, u):
        return stats.norm.ppf(u, loc=xp)

    def logG(self, t, xp, x):
        return np.where((x < self.b) & (x > self.a), 0., -np.inf)

fk_gp = GaussianProb(a=0., b=1., T=30)
```

We have added:

- methods `Gamma0` and `Gamma`, which define the deterministic functions Γ_0 and Γ we mentioned above. Mathematically, for $U \sim \mathcal{U}([0,1]^{d_u})$, then $\Gamma_0(U)$ is distributed according to $M_0(dx_0)$, and $\Gamma_t(x_{t-1}, U)$ is distributed according to $M_t(x_{t-1}, dx_t)$.
- class attribute `du`, i.e. d_u, the dimension of the u-argument of functions Γ_0 and Γ_t.

We are now able to run both the SMC and the SQMC algorithms that corresponds to the Feyman-Kac model of interest; let's compare their respective performance. (Recall that function `multiSMC` runs several algorithms multiple times, possibly with varying parameters; here we vary parameter `qmc`, which determines whether we run SMC or SMQC.)

[4]:
```
results = particles.multiSMC(fk=fk_gp, qmc={'smc': False, 'sqmc': True}, N=100, nruns=10)

sb.boxplot(x=[r['qmc'] for r in results], y=[r['output'].logLt for r in results]);
```



We do get some variance reduction, but not so much. Let's see if we can do better by addressing point 1 above.

The considered problem has the structure of a state-space model, where process (X_t) is a random walk, $Y_t = \mathbb{1}_{[a,b]}(X_t)$, and $y_t = 1$ for all t's. This remark leads us to define alternative Feynman-Kac models, that would correspond to *guided* and *auxiliary* formalisms of that state-space model. In particular, for the guided filter, the optimal proposal distribution, i.e. the distribution of $X_t | X_{t-1}, Y_t$, is simply a Gaussian distribution truncated to interval $[a, b]$; let's implement the corresponding Feynman-Kac class.

[5]:
```
def logprobint(a, b, x):
    """ returns log probability that X_t\in[a,b] conditional on X_{t-1}=x
    """
    return np.log(stats.norm.cdf(b - x) - stats.norm.cdf(a - x))

class Guided_GP(GaussianProb):

    def Gamma(self, t, xp, u):
        au = stats.norm.cdf(self.a - xp)
        bu = stats.norm.cdf(self.b - xp)
        return xp + stats.norm.ppf(au + u * (bu - au))

    def Gamma0(self, u):
        return self.Gamma(0, 0., u)

    def M(self, t, xp):
        return self.Gamma(t, xp, stats.uniform.rvs(size=xp.shape))

    def M0(self, N):
        return self.Gamma0(stats.uniform.rvs(size=N))

    def logG(self, t, xp, x):
        if t == 0:
            return np.full(x.shape, logprobint(self.a, self.b, 0.))
        else:
            return logprobint(self.a, self.b, xp)

fk_guided = Guided_GP(a=0., b=1., T=30)
```

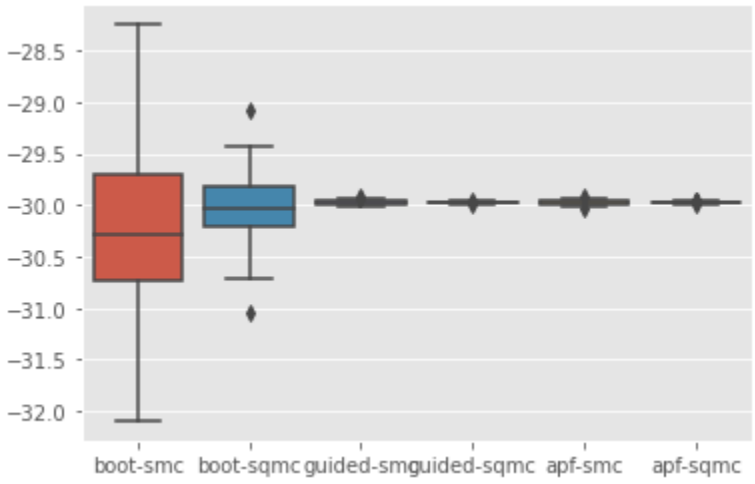
In this particular case, it is a bit more convenient to define methods `Gamma0` and `Gamma` first, and then define methods `M0` and `M`.

To derive the APF version, we must define the auxiliary functions (functions η_t in Chapter 10 of the book) that modify the resampling probabilities; in practice, we define the log of these functions, as follows:

[6]:
```
class APF_GP(Guided_GP):
    def logeta(self, t, x):
        return logprobint(self.a, self.b, x)

fk_apf = APF_GP(a=0., b=1., T=30)
```

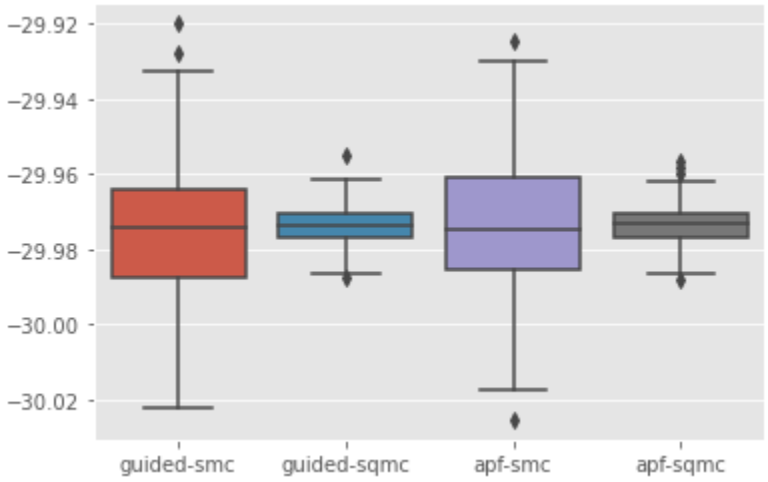
Ok, now everything is set! We can do a 3x2 comparison of SMC versus SQMC, for the 3 considered Feynman-Kac models.

[7]:
```
results = particles.multiSMC(fk={'boot':fk_gp, 'guided':fk_guided, 'apf': fk_apf},
                             N=100, qmc={'smc': False, 'sqmc': True}, nruns=200)

sb.boxplot(x=['%s-%s'%(r['fk'], r['qmc'])  for r in results], y=[r['output'].logLt for r␣
↪in results]);
```



Let's discard the bootstrap algorithms to better visualise the results for the other algorithms:

[8]:
```
res_noboot = [r for r in results if r['fk']!='boot']
sb.boxplot(x=['%s-%s'%(r['fk'], r['qmc'])  for r in res_noboot], y=[r['output'].logLt␣
↪for r in res_noboot]);
```



Voilà!

1.2.6 Defining "complicated" state-space models

Some users reported difficulties with defining "complicated" state-space models, that is models where the variables X_t and Y_t may:

- be multivariate;
- have non-standard distributions;
- have "named" components (such as "S" /"I" / "R" for the number of susceptible / infected / recovered cased in a SIR model), and the user would like the specify the model accordingly.
- be missing at certain times.

This tutorial explains how to deal with such issues.

Before we start

Note that if you are only interested in implementing the bootstrap filter associated to your model, and if the following tasks are easy to implement:

- writing a simulator that samples (N times) from the distribution of X_t given X_{t-1};
- writing a function that computes the log of density $f(y_t|x_t)$ of data-point y_t given $X_t = x_t$, for an array of N particles X_t^n;

then one option is to implement manually the corresponding Feynman-Kac object that describes that bootstrap filter. See the previous tutorial.

Multivariate state-space models (with conditional independent distributions)

The bearings-only model is a famous (toy) tracking model. The tracked object (e.g. a ship) moves according to a 2D motion model, where the speed evolves according to a random walk. State (X_t) is 4-dimensional, the first two components give the position, the next two give the velocity, and one has:

$$X_t = \begin{pmatrix} I_2 & I_2 \\ 0_2 & I_2 \end{pmatrix} X_{t-1} + \begin{pmatrix} 0_2 & 0_2 \\ 0_2 & U_t \end{pmatrix}, \qquad U_t \sim N_2\left(0_2, \sigma_X^2 I_2\right).$$

Note in particular that the first two components (the position in Cartesian coordinates) are **deterministic** functions of X_{t-1}.

Furthermore, ones observes some radar measurement, that gives the **direction** (angle) of that object, up to some noise:

$$Y_t = \operatorname{atan}\left(\frac{X_t[1]}{X_t[2]}\right) + V_t, \qquad V_t \sim N(0, \sigma_Y^2).$$

Here how you may define such a model; pay particular attention to method `PX`.

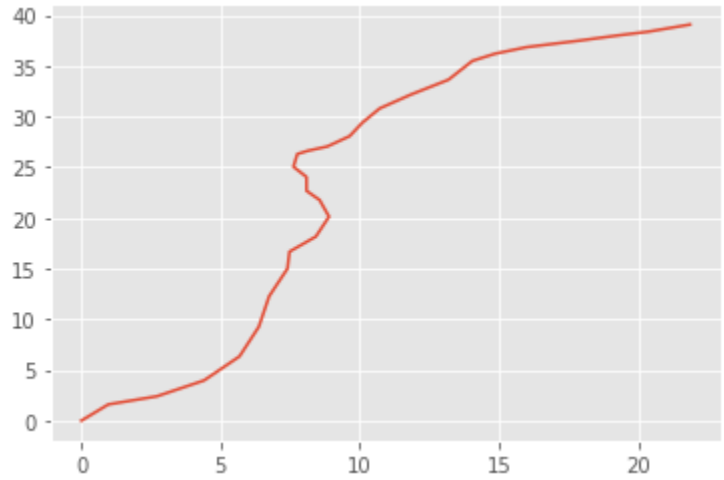
[8]:
```
from matplotlib import pyplot as plt

import particles
from particles import distributions as dists
from particles import state_space_models as ssms

class BearingsOnly(ssms.StateSpaceModel):
    """ Bearings-only tracking SSM.

    """
    default_params = {'sigmaX': 2.e-4,
                      'sigmaY': 1e-3,
                      'x0': np.array([3e-3, -3e-3, 1., 1.])
                     }

    def PX0(self):
        return dists.IndepProd(dists.Dirac(loc=self.x0[0]),
                               dists.Dirac(loc=self.x0[1]),
                               dists.Normal(loc=self.x0[2], scale=self.sigmaX),
                               dists.Normal(loc=self.x0[3], scale=self.sigmaX),
                               )

    def PX(self, t, xp):
        return dists.IndepProd(dists.Dirac(loc=xp[:, 0] + xp[:, 2]),
```

(continues on next page)

(continued from previous page)

```
                                dists.Dirac(loc=xp[:, 1] + xp[:, 3]),
                                dists.Normal(loc=xp[:, 2], scale=self.sigmaX),
                                dists.Normal(loc=xp[:, 3], scale=self.sigmaX),
                                )

    def PY(self, t, xp, x):
        angle = np.arctan(x[:, 0] / x[:, 1])
        angle[x[:, 1] < 0.] += np.pi
        return dists.Normal(loc=angle, scale=self.sigmaY)

bear = BearingsOnly(sigmaX=0.5)
x, y = bear.simulate(30)
xarr = np.array(x).squeeze()
plt.plot(xarr[:, 0], xarr[:, 1])
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f63da280880>]
```



The following points are noteworthy:

- Since the components of X_t are **independent** (conditional on X_{t-1}) we specify the distribution of X_t as a **product** of independent distributions, through `dists.IndepProd`. This particular object takes as input an arbitrary number (2 or more) of univariate distributions, and combines then to define a joint distribution.
- Since the state-space is $\mathbb{R}^4$, the particles will be stored in a numpy array of shape (N, 4): e.g. first component will be in `x[:, 0]`. (Recall that python uses zero-based indexing.)
- The probability distributions implemented in `particles` "operate on arrays": whenever a parameter of that distribution varies across particles, one should specify that parameter as an array. For instance, when we want to specify that $X_t[3]|X_{t-1} \sim N(X_{t-1}[3], \sigma_X^2)$ we must define a Gaussian distribution, where the mean (parameter `loc`) is set to `xp[:, 2]`, the array that contains the N possible values of component $X_{t-1}[3]$.

Non-independent joint distribution, named components

Of course, the big limitation of `dists.IndepProd` is that it does not let you specify joint distributions where components are *not* independent. For this, you may use "structured distributions" (`dists.StructDist`). A nice extra is that it makes it possible to name components.

Structured distributions were designed initially to specify prior distributions for parameters, but nothing prevents you from using them in the definition of a state-space model. Recall that structured distributions are intimately linked to structured arrays; e.g. when you simulate from a structured distribution, you get a structured array with the same keys. Have a quick look here if you are not familiar already with structured arrays and structured distributions.

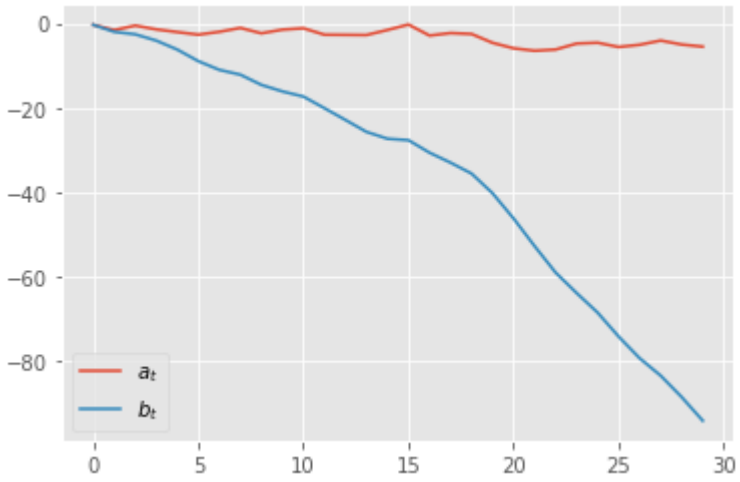
Here is a silly example: Suppose that you have a state-space model where $X_t = (A_t, B_t)$, and

$$A_t \sim N(A_{t-1}, 1)$$
$$B_t = B_{t-1} + A_t$$

In words, B_t is the cumulative sum of process A_t. We may specify a state-space model with such a process as follows:

[2]:
```
def abdist(xp):  # xp means X_{t-1}
    d = {'a': dists.Normal(loc=xp['a']),
         'b': dists.Cond(lambda x: dists.Dirac(xp['b'] + x['a']))}
    return dists.StructDist(d)

class SillyModel(ssms.StateSpaceModel):
    def PX0(self):
        return abdist({'a': 0., 'b': 0.})
    def PX(self, t, xp):
        return abdist(xp)
    def PY(self, t, xp, x):
        return dists.Normal(loc=x['a'], scale=0.3)  # whatever

silly = SillyModel()
x, y = silly.simulate(30)

plt.style.use('ggplot')
plt.plot([xt['a'] for xt in x], label=r'$a_t$')
plt.plot([xt['b'] for xt in x], label=r'$b_t$')
plt.legend()
```

[2]:
```
<matplotlib.legend.Legend at 0x7f63da991b20>
```



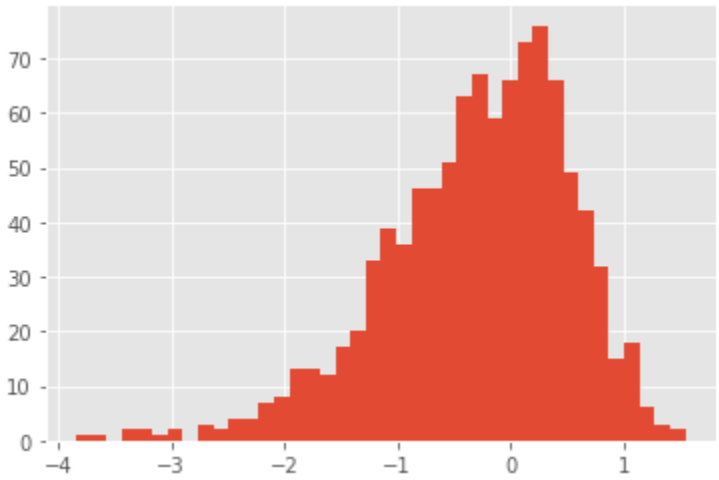
Let's unpack things:

- `dists.StructDist` takes as input a dict-like object; keys are the names of the components, and values are their distributions.
- Component `'a'` has a certain distribution (here, a Gaussian centred on A_{t-1}).
- Component `'b'` is assigned a **conditional** distribution: the `dists.Cond` object takes as input a **function**, which for a given input `x`, returns the conditional distribution for that value of `x`. Note the role played `x`: it is a structured array such that `x[f]` returns N value for any component `f` (here `'a'`) whose distribution has already been defined.
- More generally, you may implement any kind of chain rule decomposition with `StructDist`; e.g; you could introduce a component `c` whose distribution would depend on `a` and `b`, and so on. One important point: these components should defined in order; i.e. if the distribution of `'c'` depends on `'a'` and `'b'`, then `'a'` and `'b'` must be defined previously.
- Technical point: since Python 3.6, dictionaries preserve order (they remember you defined them with keys ordered in a certain way). If you use an older version of Python, you might need to use ordered dictionary, as explained in the documentation of `StructDist`.

Arbitrary distributions

With `dists.IndepProd` and `dists.StrucDist` you can create many complicated multivariate distributions out of simple univariate distributions. Many standard univariate distributions are defined in module distributions. Note you can also define many more by using transformations, e.g:

[3]:
```
# law of Y=log(X), X~Gamma(2, 2)
dist_log_gam = dists.LogD(dists.Gamma(a=2., b=2.))
x = dist_log_gam.rvs(size=1000)
plt.hist(x, 40);
```



Finally, it is always possible to create your **own** (univariate or multivariate) probability distributions, by sub-classing the base class ProbDist; see the documentation of the module for more information.

Missing data

We are going to consider two slightly different scenarios for missing data.

Missing at fixed times

Suppose we know in advance that the observation Y_t will be missing at certain times t; e.g. every Sunday for daily data. In that case, we may use the `dists.FlatNormal` distribution as follows.

[4]:
```
class ToyModelWithMissingData(ssms.StateSpaceModel):
    default_params = {'sigmaX': 1., 'sigmaY': 0.2}
    def PX0(self):
        return dists.Normal(scale=self.sigmaX)
    def PX(self, t, xp):
        return dists.Normal(loc=xp, scale=self.sigmaX)
    def PY(self, t, xp, x):
        if t % 7 == 0:  # Sunday
            return dists.FlatNormal(loc=x)
        else:  # Other days
            return dists.Normal(loc=x, scale=self.sigmaY)
```

`dists.FlatNormal` behaves like a Normal distribution with infinite variance: its log-density is flat (=zero). The model above is simply going to ignore the value of `data[t]` for any t that is a multiple of seven.

Missing at random times

In case you don't want to specify in advance at which times the data may be missing, you may use instead `dists.MixMissing`. This distribution represents a mixture distribution, where:

- with probability `pmiss`, the outcome is 'missing' (represented by value `NaN`).
- with probability `1. - pmiss`, the outcome follows a certain base distribution.

[5]:
```
class ToyModelWithMissingDataRandomTimes(ssms.StateSpaceModel):
    default_params = {'sigmaX': 1., 'sigmaY': 0.2}
    def PX0(self):
        return dists.Normal(scale=self.sigmaX)
    def PX(self, t, xp):
        return dists.Normal(loc=xp, scale=self.sigmaX)
    def PY(self, t, xp, x):
        return dists.MixMissing(pmiss=0.20,
                                base_dist=dists.Normal(loc=x, scale=self.sigmaY))
```

Now, if you simulate from this model, you get a `Nan` with probability 20% at any given time step.

[9]:
```
toymod = ToyModelWithMissingDataRandomTimes()
x, y = toymod.simulate(10)
print(np.array(y))
```

```
[[ 1.15719099]
 [ 1.12199113]
 [ 2.47971689]
 [ 2.74431727]
 [ 0.84142741]
 [ 1.10468696]
 [        nan]
 [-0.70324617]
 [        nan]
 [ 0.19505344]]
```

Then, when a particle filter is run, each Nan in the data is treated as a missing observation, and is treated as such.

[11]:
```
fk = ssms.Bootstrap(ssm=toymod, data=y)
alg = particles.SMC(fk=fk, N=100, verbose=True)
alg.run()
```

```
t=0: resample:False, ESS (end of iter)=13.63
t=1: resample:True, ESS (end of iter)=26.37
t=2: resample:True, ESS (end of iter)=14.31
t=3: resample:True, ESS (end of iter)=19.71
t=4: resample:True, ESS (end of iter)=5.47
t=5: resample:True, ESS (end of iter)=22.68
t=6: resample:True, ESS (end of iter)=100.00
t=7: resample:False, ESS (end of iter)=8.63
t=8: resample:True, ESS (end of iter)=100.00
t=9: resample:False, ESS (end of iter)=21.27
```

Note in particular how the ESS stays equals to 100 (the maximum value, since $N = 100$) at times where a Nan was observed.

Technical point: the normalising constant estimate takes into account the probability that you have a missing value; e.g. if your data consist of 10 Nans, the estimate equals p^{10}, where p is the missing probability.

Multivariate observations with missing data

The examples above assume that the observations Y_t are univariate, but you can combine `FlatNormal` or `MixMissing` with `dists.IndepProd` to specify a model where components of Y_t may be missing. For instance:

[16]:
```
class MultivariateModelWithMissingData(ssms.StateSpaceModel):
    def PX0(self):
        return dists.IndepProd(dists.Normal(),
                               dists.Normal())
    def PX(self, t, xp):
        return dists.IndepProd(dists.Normal(loc=xp[:, 0]),
                               dists.Normal(loc=xp[:, 1]))
    def PY(self, t, xp, x):
        return dists.IndepProd(dists.Normal(loc=x[:, 0]),
                               dists.MixMissing(pmiss=0.20,
                                base_dist=dists.Normal(loc=x[:,0])))
```

In this model, only the second component of Y_t may be missing, with probability 20%:

[27]:
```
ssm = MultivariateModelWithMissingData()
x, y = ssm.simulate(10)
print(np.squeeze(y))
```

```
[[-1.65844712         nan]
 [-0.47070688 -1.0353565 ]
 [ 1.99688692  1.46915379]
 [-0.54656891  0.68420457]
 [ 1.06249224  1.62856419]
 [ 0.93699561 -1.67133867]
 [-1.21527128 -0.65343266]
 [-4.56505231 -3.10377106]
 [-6.88535672         nan]
 [-7.42773445 -5.51978829]]
```

Questions?

This tutorial was written to answer some recurring questions from users; if you feel you are still unable to implement your model, feel free to get in touch with the author of the package.

1.2.7 Variance estimators

Outline

Consider a particle estimate computed at time t of a SMC algorithm:

$$\mathbb{Q}_t^N(\varphi) = \sum_{n=1}^N W_t^n \varphi(X_t^n).$$

A basic way to evaluate its variability is to run the algorithm many times, and report the empirical variance. Which is of course expensive. (Note however that function `multiSMC` in the `core` module makes it possible to run SMC algorithms in parallel on multiple-core machines).

Several recent papers (Chan and Lai, 2013; Lee and Whiteley, 2018; Olsson and Douc, 2019) have proposed estimators of the variance of a particle estimate that may be computed from a **single run**. These estimators require to track the genealogy of the particle system. In fact, the expression of these estimators is a variation of this:

$$\sum_{n=1}^{N}\left[\sum_{m:B_t^m=n} W_t^m\left\{\varphi(X_t^m)-\mathbb{Q}_t^N(\varphi)\right\}\right]^2 \tag{1.3}$$

where B_t^n is the index of the ancestor of X_t^n at time 0 (the so-called **eve variables** in Lee and Whiteley, 2018). Note that this quantity **equals zero** as soon as all the particles have the same ancestor at time 0. More generally, we expect these estimators to suffer from **path degeneracy** (i.e. number of distinct ancestors drops quickly.)

This notebook:

- explains how to compute such estimators, using module `variance_estimators`;
- showcases their performance in a toy example.

Computing the variance estimators

We start with the usual imports.

[1]:
```
import itertools as it

from matplotlib import pyplot as plt
import numpy as np
from numpy import random
import pandas as pd
from scipy import stats
import seaborn as sb

import particles
from particles import state_space_models as ssms
from particles import kalman  # Linear Gaussian state space models are defined here
from particles import collectors as col  # standard collectors

plt.style.use('ggplot')
```

We consider a basic univariate linear Gaussian model:

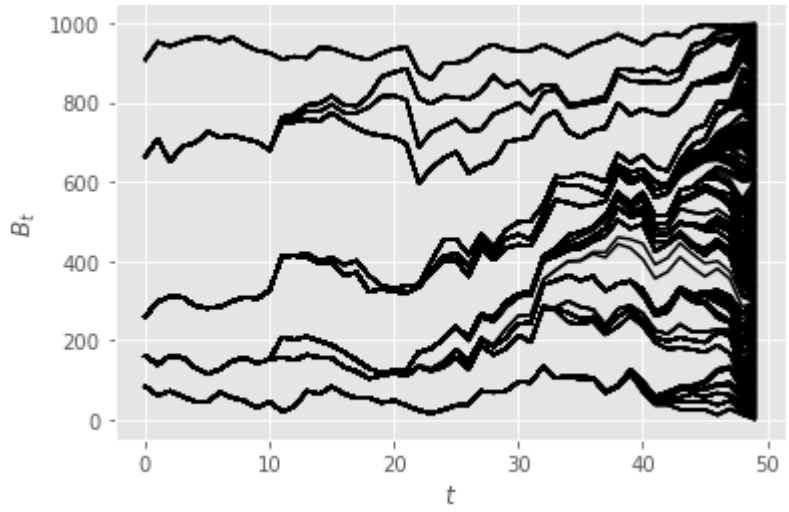
$$X_t = \rho X_{t-1} + \sigma_X U_t$$
$$Y_t = X_t + \sigma_Y V_t$$

with $\rho = 0.9$, $\sigma_X = 1$, $\sigma_Y = 0.2$.

We simulate data ($T = 50$) from the model.

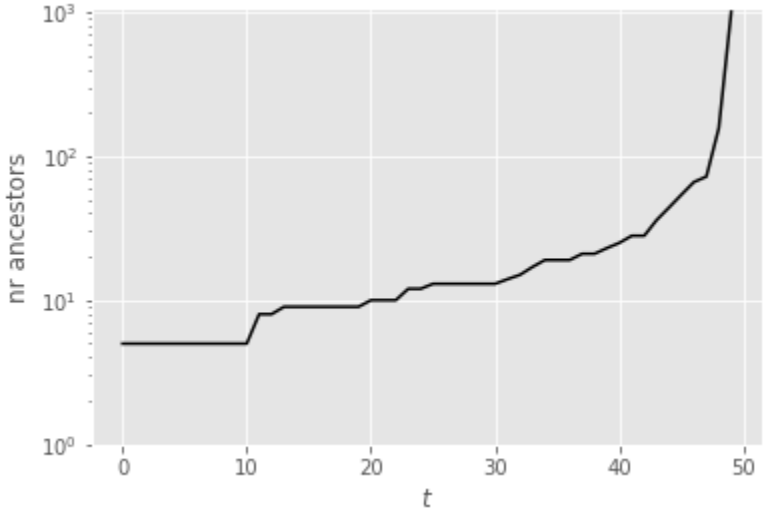
Let's run a single bootstrap filter, and have a look at the genealogical tree.

[2]:
```
T = 50
ssm = kalman.LinearGauss(rho=0.9, sigmaX=1., sigmaY=0.2)
true_x, data = ssm.simulate(T)
fk = ssms.Bootstrap(ssm=ssm, data=data)
N = 1000
alg = particles.SMC(fk=fk, N=N, store_history=True) # store_history: keeps the complete␣
↪history
alg.run()

B = alg.hist.compute_trajectories()

plt.figure()
for n in range(N):
    plt.plot(B[:, n], 'k')
plt.xlabel(r'$t$')
plt.ylabel(r'$B_t$');
```



The plot above represents the index of the ancestors of the final particles X_T^n, at times $t = 0, ..., T$. Alternatively, we can plot the number of distinct ancestors (of particles X_T^n) at time t; see below. Clearly, although this is a toy example, and the number of time steps is small, we already observe some strong path degeneracy (although the number of ancestors at time 0 does not collapse to one).

[3]:
```
plt.figure()
plt.plot([np.unique(B[t, :]).shape[0] for t in range(T)], 'k')
plt.xlabel(r'$t$')
plt.ylabel('nr ancestors')
plt.ylim(bottom=1)
plt.yscale('log')
```



The block below shows how one may compute various variance estimators. These estimators are implemented as collectors (see module `collectors`); i.e. as objects that compute and collect, at each time t, a certain variance estimator, and save the result in an an attribute of `smc.summaries`, where `smc` is the considered `SMC` instance (the algorithm you are running).

More precisely, module `variance_estimators` includes the following collectors:

- `Var(phi=None)`: computes the variance estimator defined above, for function `phi` (default = identity function).
- `Var_logLt()`: computes the variance estimator for the estimates of the log-normalising constants, $\log L_t$ (Lee & Whiteley, 2018).
- `Lag_based_Var(phi=None)`: computes the lag-based estimator of Olsson and Douc (2019) for function `phi`. You must then specify a `maximum lag`, through the extra command `store_history=max_lag`. See module `smoothing` for more details on storing (part of) the history of a SMC algorithm.

[4]:
```
from particles import variance_estimators as var

def phi(x):
    return x

max_lag = 11
nruns = 10_000
runs = particles.multiSMC(fk=fk, N=N, resampling='multinomial', # see below
                          collect=[col.Moments(), var.Var_logLt(), var.Var(phi=phi), var.
↪Lag_based_var(phi=phi)],
                          store_history=max_lag, nruns=nruns, nprocs=0)

# We could do var.Var() and var.Lag_based_var(), since identity function is the default␣
↪test function anyway
```

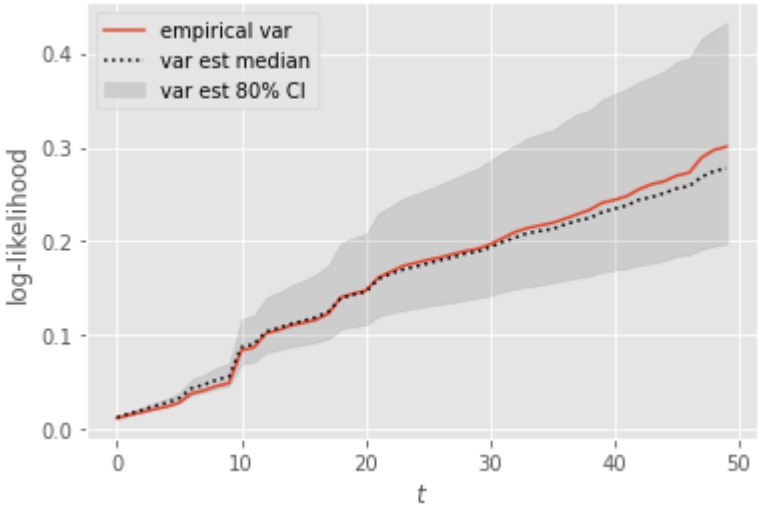
Note: In principle, these variance estimators are valid only when multinomial resampling is used, which is why we chose that particular resampling scheme.

Numerical results

We now use the results to assess the variability of the standard (not lag-based) variance estimator. To do so, we compare the empirical variance (over the 10^4 runs) with the empirical distribution of the estimates (summarised by the median, the $10\%-$ and the $90\%-$ quantile).

We start with the normalising constants (i.e. the marginal likelihood of the data until time t).

[13]:
```
def plot_ci(time_range, arr):
    "Plot (empirical) confidence intervals."
    plt.plot(time_range, np.percentile(arr, 50, axis=0), 'k:',
             label='var est median')
    plt.fill_between(time_range, np.percentile(arr, 10, axis=0), np.percentile(arr, 90,␣
↪axis=0),
                     alpha=0.1, color='black', label='var est 80% CI')

plt.plot([np.var([r['output'].summaries.logLts[t] for r in runs]) for t in range(T)],
         label='empirical var')
ll_var_ests = np.array([r['output'].summaries.var_logLt for r in runs])
plot_ci(np.arange(T), ll_var_ests)
plt.xlabel(r'$t$')
plt.ylabel('log-likelihood')
plt.legend(loc='upper left');
```



Not too bad, right? Note however that the error (in estimating the variance) seems to grow over time, and tends to be negative (under-estimation of the variance).

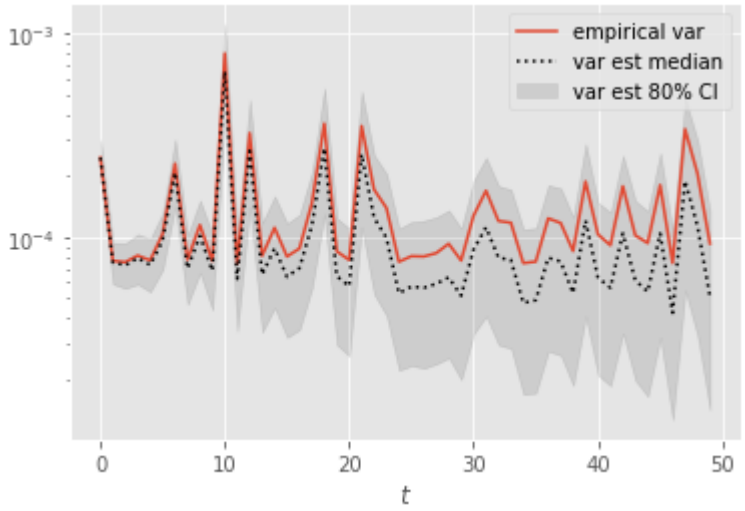
Let's now look at the performance of the variance estimators for function $\varphi(x) = x$; i.e. the variance of the particle estimates of the filtering expectations.

[6]:
```
plt.figure()
ests =  np.array([[r['output'].summaries.moments[t]['mean'] for r in runs]
                  for t in range(T)]).T
plt.plot(np.var(ests, axis=0), label='empirical var')
var_ests = np.array([r['output'].summaries.var for r in runs])
plot_ci(np.arange(T), var_ests)
plt.legend()
```

(continues on next page)

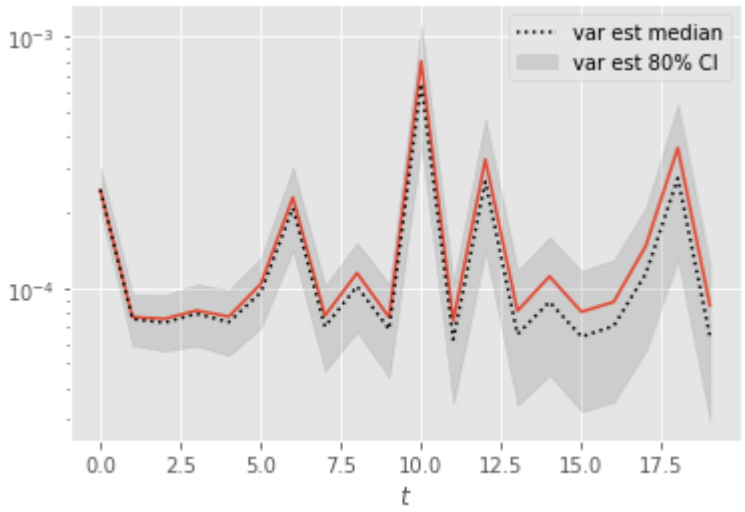
(continued from previous page)

```
plt.xlabel(r'$t$')
plt.yscale('log')
```



Again, error seems to grow with time. Let's zoom on the first time steps:

[7]:
```
t0 = 20
plt.figure()
plt.plot(np.var(ests[:, :t0], axis=0))
plot_ci(np.arange(t0), var_ests[:, :t0])
plt.legend()
plt.xlabel(r'$t$')
plt.yscale('log')
```



At the first time steps, the variance estimates seem more reliable, presumably because path degeneracy remains limited at times ≤ 20.

Performance over repeated runs

Variance estimates may get more robust if averaged over repeated runs. To see if this is true, we bootstrap the sample of estimates to assess the performance over 10 runs.

[8]:
```
# Bootstrap
nsample = 10
nboot = 10_000
boot_emp_var = np.zeros((nboot, T))
boot_var_ests = np.zeros((nboot, T))
for n in range(nboot):
    idx = random.choice(nruns, size=nsample, replace=False)
    boot_emp_var[n, :] = np.var(ests[idx, :], axis=0)
    boot_var_ests[n, :] = np.mean(var_ests[idx, :], axis=0)
```

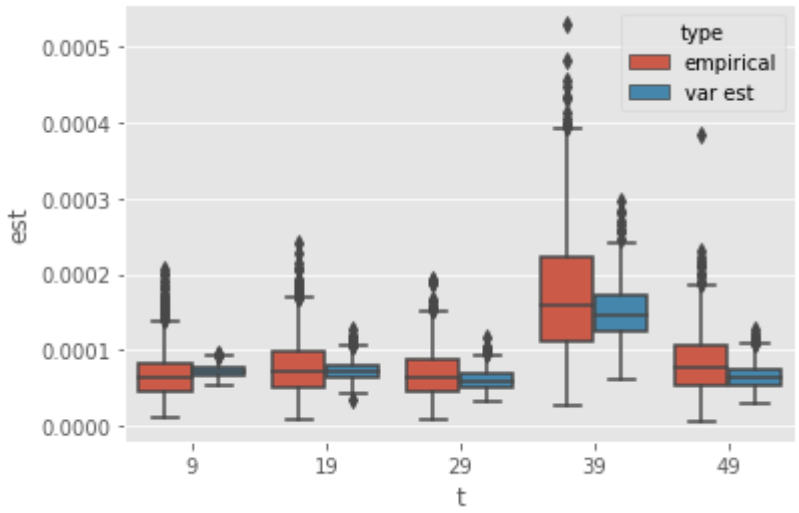
The following plot compares the performance of:

- the empirical variance (of the considered estimate) over 10 runs;
- the empirical mean of the variance estimator over 10 runs.

Through averaging, the estimator becomes far more reliable, and seems to show comparable or better performance than the empirical variance. However, the level of improvement seems to decrease over time (again because of path degeneracy).

[14]:
```
import itertools
import pandas

d1 = [{'t': t, 'type': 'empirical', 'est': boot_emp_var[n, t]}
      for n, t in it.product(range(N), range(T))]
d2 = [{'t': t, 'type': 'var est', 'est': boot_var_ests[n, t]}
      for n, t in it.product(range(N), range(T))]
df = pandas.DataFrame(d1 + d2)
dft = df[df['t'] % 10 == 9 ]
sb.boxplot(x='t', y='est', hue='type', data=dft)
```

[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7226657700>
```



Thus, averaging does seem to help in making these estimators more robust (even at times where there is already strong

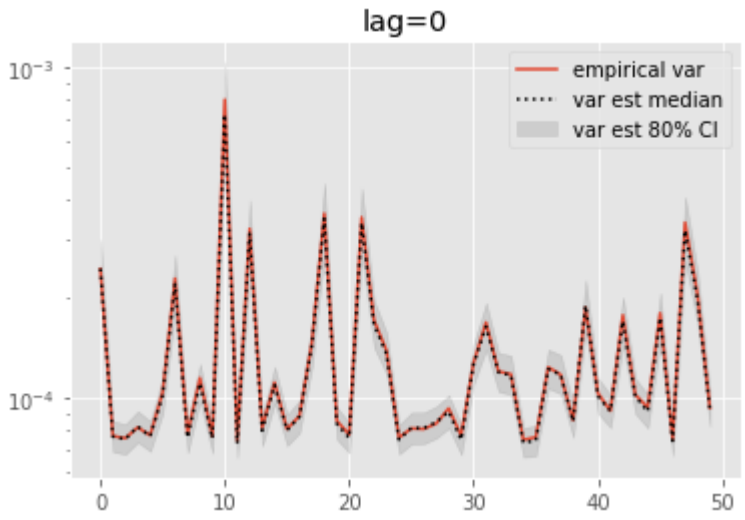
path degeneracy).

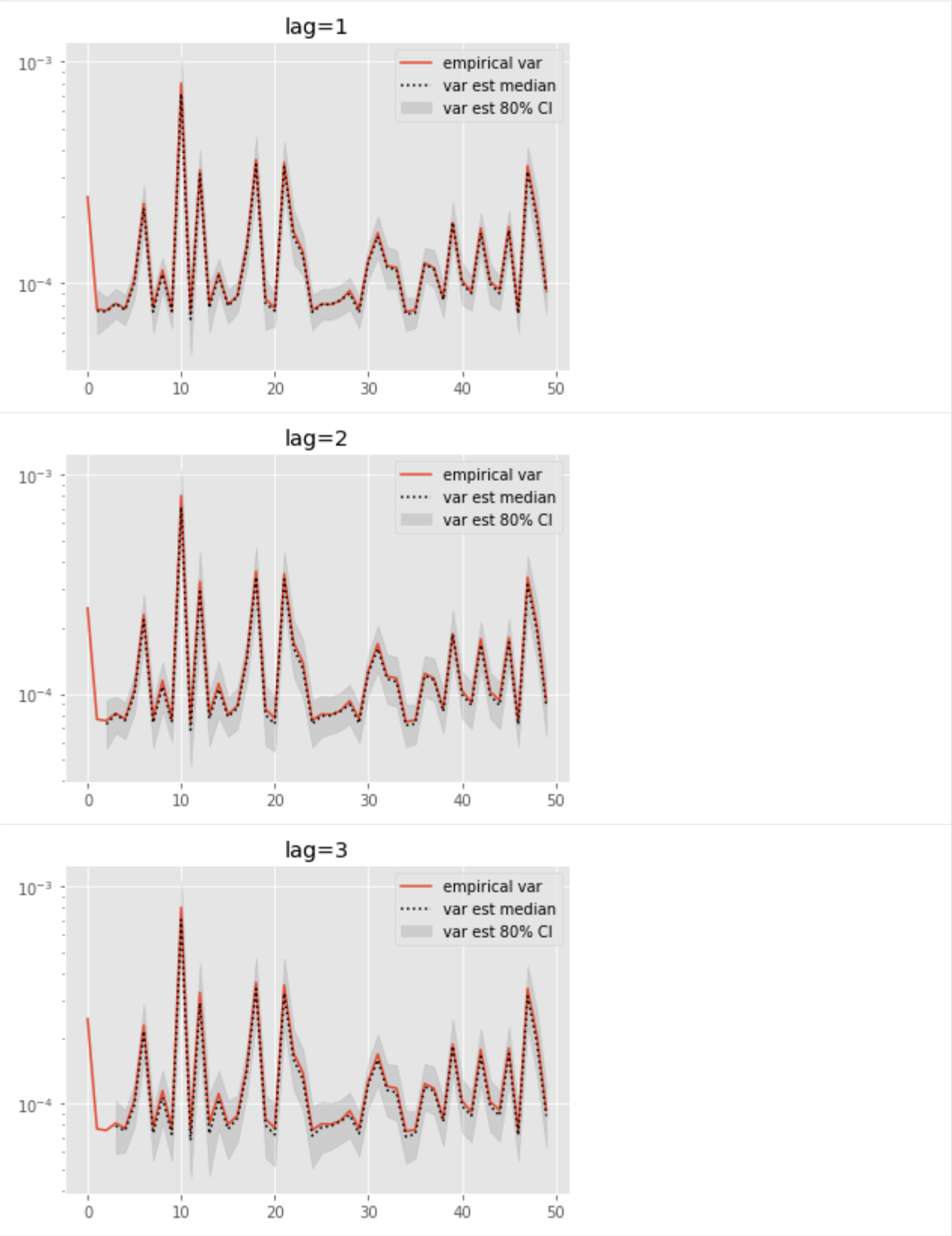
Lag-based estimators

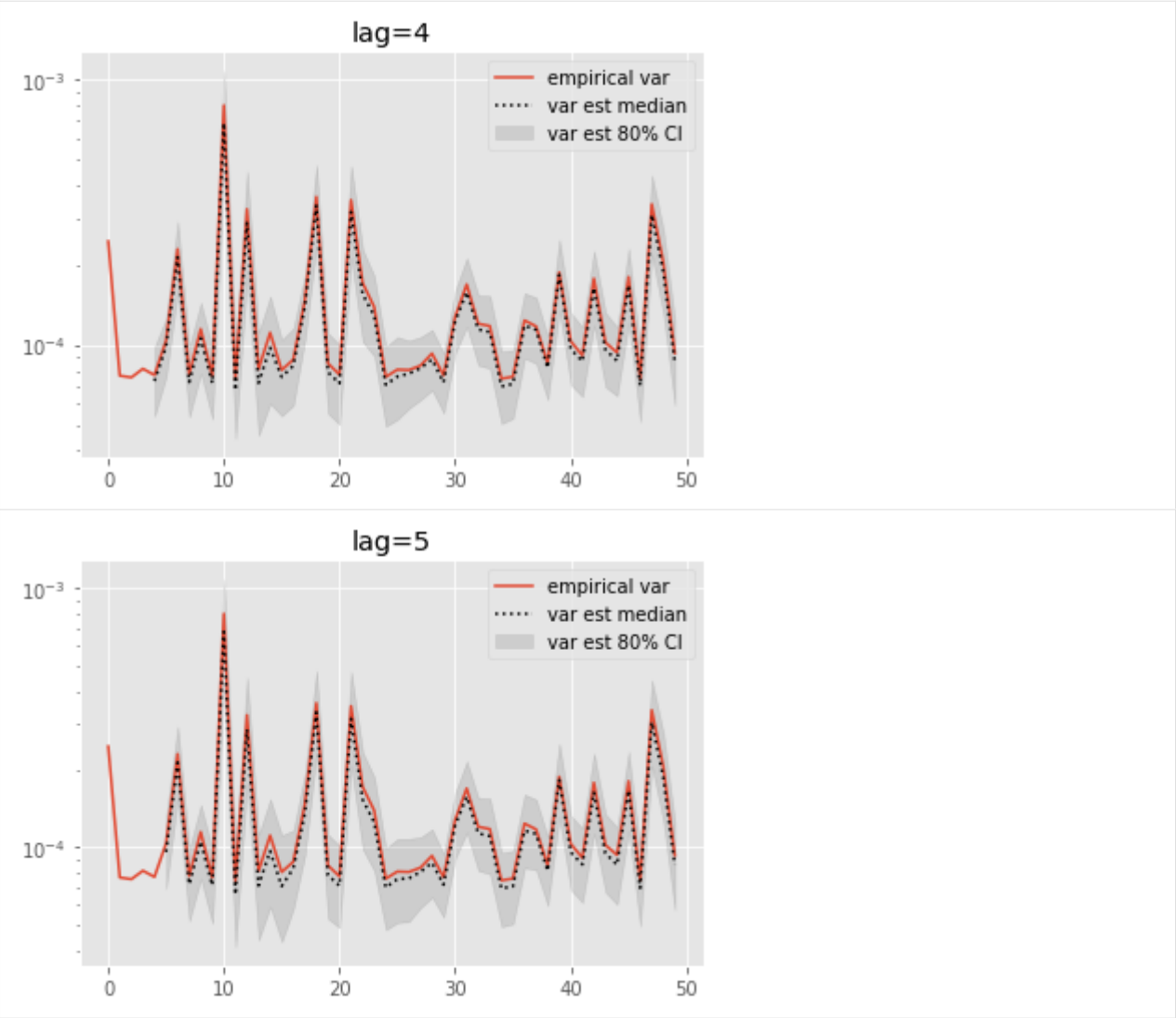
Olsson and Douc (2019) proposed a variant of the above estimators based on a lag approximation. This introduces a bias, but this also should alleviate the path degeneracy.

We now look at the performance of lag-based variance estimators.

[10]:
```
def get_list(l, k):
    if k < len(l):
        return l[k]
    else:
        return 0.
for lag in range(0, 6):
    plt.figure()
    plt.title('lag=%i' % lag)
    plt.plot(np.var(ests, axis=0), label='empirical var')
    times = list(range(lag, T))
    lag_var_ests = np.array([[get_list(est, lag)
                              for est in r['output'].summaries.lag_based_var[lag:]]
                             for r in runs])
    plot_ci(times, lag_var_ests)
    plt.legend()
    plt.yscale('log')
```




lag=1
empirical var
var est median
var est 80% CI
10^{-3}
10^{-4}
0
10
20
30
40
50
lag=2
empirical var
var est median
var est 80% CI
10^{-3}
10^{-4}
0
10
20
30
40
50
lag=3
empirical var
var est median
var est 80% CI
10^{-3}
10^{-4}
0
10
20
30
40
50



These results are a tad surprising: little bias is observed even for lag=0. Increasing the lag seems only to increase the error.

Bottom line

Based on these experiments (which are of course a bit limited), the following recommendations seem reasonable:

- Averaging over a small number of runs seems to make variance estimators more robust.
- Even with averaging, the performance gain (relative to the empirical variance) may be modest as soon as the number of distinct eve variables is too small.
- When the number of distinct ever variables drop to one, these variance estimators do not work any more.
- In that case, it may be worth switching to the lag-based variant of Olsson and Douc (2019).
- However, choosing the value of the lag seems non-trivial, and deserves further investigation.

References

- Chan, H.P. and Lai, T.L. (2013). A general theory of particle filters in hidden Markov models and some applications. Ann. Statist. 41, pp. 2877–2904.
- Lee, A and Whiteley, N (2018). Variance estimation in the particle filter. Biometrika 3, pp. 609-625.
- Olsson, J. and Douc, R. (2019). Numerically stable online estimation of variance in particle filters. Bernoulli 25.2, pp. 1504-1535.

CHAPTER

TWO

INSTALLATION

2.1 Python 2 vs Python 3

Short version: **particles** only supports Python 3.

Long version: initially, **particles** supported both Python 2 and Python 3. However, Python 2 has been deprecated, and supporting it has become troublesome. The last version to support it has been turned into a release; you may find in the releases section of the github page (first release, April 15, 2020).

2.2 Requirements

The short version: install The Anaconda distribution to get all the scientific computing libraries you may ever need in one go.

The long version: **particles** requires the following libraries:

- NumPy
- SciPy
- numba
- joblib
- scikit-learn (only for module *binary_smc*)

In addition, it is **strongly recommended** to install the following libraries (optional `[extra]` dependencies):

- Matplotlib
- seaborn
- pandas
- statsmodels

Most of the scripts require the first two libraries to plot the results; a few of them also requires `pandas` or `statsmodels`.

Again, the easiest way to install all these libraries in one go is to simply install the Anaconda distribution. Manual installation is of course also possible; e.g. on Ubuntu/Debian:

```
sudo apt install python3-numpy python3-scipy python3-numba python3-joblib python3-sklearn
```

However, in the maintainer's experience, conda is usually less hassle, runs faster (because it install a more efficient version of low-level libraries such as BLAS) and provides more recent versions.

2.3 Local installation, organisation of the package

You are strongly recommended to install the package *locally* (in a folder in your home directory). The installation instructions in the following sections explains how to do it.

The reason why it is better to do a local install is that you will have easier access to the scripts in the package. In particular:

- Folder `book` contains all the scripts that were used to perform the experiments and generate the plots in the book. Each sub-folder corresponds to a different chapter.
- Folder `papers` contains scripts that reproduce some of the experiments of relevant papers. Each sub-folder corresponds to a different paper, and contains a `README.md` which gives the reference of the paper.

The rest of the package is organised in a completely standard manner (modules are in folder `particles`, documentation is in `docs`, etc.).

2.4 Installation: recommended method

The package is available on Github and may be installed using git:

```
cd some_folder_of_your_choosing
git clone https://github.com/nchopin/particles.git
cd particles
pip install -e .[extra]
```

The option `-e` installs the package in editable mode (any modification you make in this folder will be taken into account when you import the package). Replace `-e` by `--user` if you want to avoid that. Also, you may need to replace the last line by:

```
pip install -e '.[extra]'
```

on certain shells (e.g. zsh).

2.5 Installation: alternative method

The package is also available on PyPI (the Python package index), so you may install it by running pip. On a Linux machine:

```
pip install --user particles[extra]
```

Option `--user` lets you install the package in your home directory, rather than globally for all users.

The main drawback of this method is that it installs only the modules of the package, and not the scripts found in folders book and papers.

CHAPTER

THREE

GENERAL STRUCTURE

3.1 Folders

The package contains the following noteworthy folders:

- `particles/`: contains the modules of the package (in case you want to read the source, or edit it in editable mode, see *Installation*).
- `book/`: scripts to generate the plots in the book; each sub-folder corresponds to a different chapter (e.g. `smoothing`).
- `papers/`: scripts to reproduce numerical experiments from a few relevant papers (i.e. papers which describe algorithms that are now implemented in `particles`). Each sub-folder corresponds to a different paper, and contains a `README.md` file that describe briefly the experiments and give the full reference of the paper. At the moment, these papers are:
  - waste-free SMC (Dau & Chopin, 2022);
  - nested sampling SMC (Salomone et al, 2018);
  - SMC for large binary spaces (Schäfer & Chopin, 2014);
  - on backward smoothing algorithms (Dau & Chopin, 2023).
- `docs/`: documentation (managed by `sphinx`). The jupyter notebooks are in `docs/source/notebooks`.

3.2 API reference

particles	Sequential Monte Carlo in python.

3.2.1 particles

Sequential Monte Carlo in python.

Modules

particles.binary_smc	SMC samplers for binary spaces.
particles.collectors	Objects that collect summaries at each iteration of a SMC algorithm.
particles.core	Core module.
particles.datasets	Where datasets live.
particles.distributions	Probability distributions as Python objects.
particles.hilbert	Hilbert curve and its inverse, in any dimension.
particles.hmm	Baum-Welch filter/smoother for hidden Markov models.
particles.kalman	Basic implementation of the Kalman filter (and smoother).
particles.mcmc	MCMC (Markov chain Monte Carlo) and related algorithms.
particles.nested	Nested sampling (vanilla and SMC).
particles.resampling	Resampling and related numerical algorithms.
particles.rqmc	Randomised quasi-Monte Carlo sequences.
particles.smc_samplers	Classical and waste-free SMC samplers.
particles.smoothing	Off-line particle smoothing algorithms.
particles.state_space_models	State-space models as Python objects.
particles.utils	Non-numerical utilities (notably for parallel computation).
particles.variance_estimators	Single-run variance estimators.
particles.variance_mcmc	MCMC variance estimators.

particles.binary_smc

SMC samplers for binary spaces.

Overview

This module implements SMC tempering samplers for target distributions defined with respect to a binary space, {0, 1}^d. This is based on Schäfer & Chopin (2014). Note however the version here also implements the waste-free version of these SMC samplers, see Dang & Chopin (2020).

This module builds on the *smc_samplers* module. The general idea is that the N particles are represented by a (N, d) boolean numpy array, and the different components of the SMC sampler (e.g. the MCMC steps) operate on such arrays.

More precisely, this module implements:

- `NestedLogistic`: the proposal distribution used in Schäfer and Chopin (2014), which amounts to fit a logistic regression to each component i, based on the (i-1) previous components. This is a sub-class of `distributions.DiscreteDist`.
- `BinaryMetropolis`: Independent Metropolis step based on a NestedLogistic proposal. This is a sub-class of `smc_samplers.ArrayMetropolis`.
- Various sub-classes of `smc_samplers.StaticModel` that implements Bayesian variable selection.

See also the scripts in papers/binarySMC for numerical experiments.

References

- Dau, H. D., & Chopin, N. (2022). Waste-free sequential Monte Carlo. Journal of the Royal Statistical Society Series B: Statistical Methodology, 84(1), 114-148.
- Schäfer, C., & Chopin, N. (2013). Sequential Monte Carlo on large binary sampling spaces. Statistics and Computing, 23, 163-184.

Functions

all_binary_words(p)	
chol_and_friends(gamma, xtx, xty, vm2)	
corr_bin(pi, pj, pij)	
jitted_chol_and_fr(gamma, xtx, xty, vm2)	
log_no_warn(x)	log without the warning about x <= 0.

Classes

BIC([data, lamb])	Likelihood is exp{ - lambda * BIC(gamma)}
BayesianVS([data, prior, nu, lamb, iv2, jitted])	Marginal likelihood for the following hierarchical model: Y = X beta + noise noise ~ N(0, sigma^2) sigma^2 ~ IG(nu / 2, lambda*nu / 2) beta | sigma^2 ~ N(0, v2 sigma^2 I_p)
BayesianVS_gprior([data, prior, nu, lamb, ...])	Same model as parent class, except: beta | sigma^2 ~ N(0, g sigma^2 (X'X)^-1)
Bernoulli(p)	
BinaryMetropolis()	
NestedLogistic(coeffs, edgy)	Nested logistic proposal distribution.
VariableSelection([data])	Meta-class for variable selection.

particles.collectors

Objects that collect summaries at each iteration of a SMC algorithm.

Overview

This module implements "summary collectors", that is, objects that collect at every time t certain summaries of the particle system. Important applications are **fixed-lag smoothing** and **on-line smoothing**. However, the idea is a bit more general that that. Here is a simple example:

```
import particles
from particles import collectors as col

# ...
# define some_fk_model
# ...
alg = particles.SMC(fk=some_fk_model, N=100,
                    collect=[col.Moments(), col.Online_smooth_naive()])

alg.run()
print(alg.summaries.moments)  # list of moments
print(alg.summaries.naive_online_smooth)  # list of smoothing estimates
```

Once the algorithm is run, the object `alg.summaries` contains the computed summaries, stored in lists of length T (one component for each iteration t). Note that:

- argument `collect` expects a **list** of Collector objects;
- the name of the collector classes are capitalised, e.g. `Moments`;
- by default, the name of the corresponding summaries are not, e.g. `pf.summaries.moments`.

Default summaries

By default, the following summaries are collected (even if argument `collect` is not used):

- `ESSs`: ESS (effective sample size) at each iteration;
- `rs_flags`: whether resampling was triggered or not at each time t;
- `logLts`: log-likelihood estimates.

For instance:

```
print(alg.summaries.ESSs) # sequence of ESSs
```

You may turn off summary collection entirely:

```
alg = particles.SMC(fk=some_fk_model, N=100, collect='off')
```

This might be useful in very specific cases when you need to keep a large number of SMC objects in memory (as in SMC^2). In that case, even the default summaries might take too much space.

Computing moments

To compute moments (functions of the current particle sample):

```
def f(W, X):  # expected signature for the moment function
    return np.average(X, weights=W)  # for instance

alg = particles.SMC(fk=some_fk_model, N=100,
                        collect=[Moments(mom_func=f)])
```

Without an argument, i.e. `Moments()`, the collector computes the default moments defined by the `FeynmanKac` object; for instance, for a `FeynmanKac` object derived from a state-space model, the default moments at time t consist of a dictionary, with keys `'mean'` and `'var'`, containing the particle estimates (at time t) of the filtering mean and variance.

It is possible to define different defaults for the moments. To do so, override method `default_moments` of the considered FeynmanKac class:

```
from particles import state_space_models as ssms
class Bootstrap_with_better_moments(ssms.Bootstrap):
    def default_moments(W, X):
        return np.average(X**2, weights=W)
#  ...
#  define state-space model my_ssm
#  ...
my_fk_model = Bootstrap_with_better_moments(ssm=my_ssm, data=data)
alg = particles.SMC(fk=my_fk_model, N=100, moments=True)
```

In that case, `my_fk_model.summaries.moments` is a list of weighed averages of the squares of the components of the particles.

Fixed-lag smoothing

Fixed-lag smoothing means smoothing of the previous h states; that is, computing (at every time t) expectations of

$$\mathbb{E}[\phi_t(X_{t-h:t})|Y_{0:t} = y_{0:t}]$$

for a fixed integer h (at times t <= h; if t<h, replace h by t).

This requires keeping track of the h previous states for each particle; this is achieved by using a rolling window history, by setting option `store_history` to an int equals to h+1 (the length of the trajectories):

```
alg = particles.SMC(fk=some_fk_model, N=100,
                        collect=[col.Fixed_lag_smooth(phi=phi)],
                        store_history=3)  # h = 2
```

See module *smoothing* for more details on rolling window and other types of particle history. Function phi takes as an input the N particles, and returns a numpy.array:

```
def phi(X):
    return np.exp(X - 2.)
```

If no argument is provided, test function $\varphi(x) = x$ is used.

Note however that X is a deque of length at most h; it behaves like a list, except that its length is always at most h + 1. Of course this function could simply return its arguments `W` and `X`; in that case you simply record the fixed-lag trajectories (and their weights) at every time t.

On-line smoothing

On-line smoothing is the task of approximating, at every time t, expectations of the form:

$$\mathbb{E}[\phi_t(X_{0:t})|Y_{0:t} = y_{0:t}]$$

On-line smoothing is covered in Sections 11.1 and 11.3 in the book. Note that on-line smoothing is typically restricted to *additive* functions ϕ, see below.

The following collectors implement online-smoothing algorithms:

- `Online_smooth_naive`: basic forward smoothing (carry forward full trajectories); cost is O(N) but performance may be poor for large t.
- `Online_smooth_ON2`: O(N^2) on-line smoothing. Expensive (cost is O(N^2), so big increase of CPU time), but better performance.
- `Paris`: on-line smoothing using Paris algorithm. (Warning: current implementation is very slow, work in progress).

These algorithms compute the smoothing expectation of a certain additive function, that is a function of the form:

$$\phi_t(x_{0:t}) = \psi_0(x_0) + \psi_1(x_0, x_1) + ... + \psi_t(x_{t-1}, x_t)$$

The elementary function ψ_t is specified by defining method `add_func` in considered state-space model. Here is an example:

```
class BootstrapWithAddFunc(ssms.Bootstrap):
    def add_func(self, t, xp, x):  # xp means x_{t-1} (p=past)
        if t == 0:
            return x**2
        else:
            return (xp - x)**2
```

The reason why additive functions are specified in this way is that additive functions often depend on fixed parameters of the state-space model (which are available in the closure of the `StateSpaceModel` object, but not outside).

The two first algorithms do not have any parameter, the third one (Paris) have one (default: 2). To use them simultaneously:

```
alg = particles.SMC(fk=some_fk_model, N=100,
                        collect=[col.Online_smooth_naive(),
                                 col.Online_smooth_ON2(),
                                 col.Paris(Nparis=5)])
```

Variance estimators

The variance estimators of Chan & Lai (2013), Lee & Whiteley (2018), etc., are implemented as collectors in module *variance_estimators*; see the documentation of that module for more details.

User-defined collectors

You may implement your own collectors as follows:

```
import collectors

class Toy(collectors.Collector):
    # optional, default: toy (same name without capital)
    summary_name = 'toy'

    # signature of the __init__ function (optional, default: {})
    signature = {phi=None}

    # fetch the quantity to collect at time t
    def fetch(self, smc):  # smc is the particles.SMC instance
        return np.mean(self.phi(smc.X))
```

Once this is done, you may use this new collector exactly as the other ones:

```
alg = particles.SMC(N=30, fk=some_fk_model, collect=[col.Moments(), Toy(phi=f)])
```

Then `pf.summaries.toy` will be a list of the summaries collected at each time by the `fetch` method.

Classes

`Collector`(**kwargs)	Base class for collectors.
`ESSs`(**kwargs)	
`Fixed_lag_smooth`(**kwargs)	Compute some function of fixed-lag trajectories.
`LogLts`(**kwargs)	
`Moments`(**kwargs)	Collects empirical moments (e.g.
`OnlineSmootherMixin`()	Mix-in for on-line smoothing algorithms.
`Online_smooth_ON2`(**kwargs)	
`Online_smooth_naive`(**kwargs)	
`Paris`(**kwargs)	Hybrid version of the Paris algorithm.
`Rs_flags`(**kwargs)	
`Summaries`(cols)	Class to store and update summaries.

particles.core

Core module.

Overview

This module defines the following core objects:

- `FeynmanKac`: the base class for Feynman-Kac models;
- `SMC`: the base class for SMC algorithms;
- `multiSMC`: a function to run a SMC algorithm several times, in parallel and/or with varying options.

You don't need to import this module: these objects are automatically imported when you import the package itself:

```
import particles
help(particles.SMC)  # should work
```

Each of these three objects have extensive docstrings (click on the links above if you are reading the HTML version of this file). However, here is a brief summary for the first two.

The FeynmanKac abstract class

A Feynman-Kac model is basically a mathematical model for the operations that we want to perform when running a particle filter. In particular:

- The distribution *M_0(dx_0)* says how we want to simulate the particles at time 0.
- the Markov kernel *M_t(x_{t-1}, dx_t)* says how we want to simulate particle X_t at time t, given an ancestor X_{t-1}.
- the weighting function *G_t(x_{t-1}, x_t)* says how we want to reweight at time t a particle X_t and its ancestor is X_{t-1}.

For more details on Feynman-Kac models and their properties, see Chapter 5 of the book.

To define a Feynman-Kac model in particles, one should, in principle:

(a) sub-class `FeynmanKac` (define a class that inherits from it) and define certain methods such as `M0`, `M`, `G`, see the documentation of `FeynmanKac` for more details;

(b) instantiate (define an object that belongs to) that sub-class.

In many cases however, you do not need to do this manually:

- module *state_space_models* defines classes that automatically generate the bootstrap, guided or auxiliary Feynman-Kac model associated to a given state-space model; see the documentation of that module.
- Similarly, module *smc_samplers* defines classes that automatically generates `FeynmanKac` objects for SMC tempering, IBIS and so on. Again, check the documentation of that module.

That said, it is not terribly complicated to define manually a Feynman-Kac model, and there may be cases where this might be useful. There is even a basic example in the tutorials if you are interested.

SMC class

SMC is the class that define SMC samplers. To get you started:

```
import particles
... # define a FeynmanKac object in some way, see above
pf = particles.SMC(fk=my_fk_model, N=100)
pf.run()
```

The code above simply runs a particle filter with N=100 particles for the chosen Feynman-Kac model. When this is done, object pf contains several attributes, such as:

- X: the current set of particles (at the final time);
- W: their weights;
- cpu_time: as the name suggests;
- and so on.

SMC objects are iterators, making it possible to run the algorithm step by step: replace the last line above by:

```
next(step) # do iteration 0
next(step) # do iteration 1
pf.run() # do iterations 2, ... until completion (dataset is exhausted)
```

All options, minus model, are optional. Perhaps the most important ones are:

- qmc: if set to True, runs SQMC (the quasi-Monte Carlo version of SMC)
- resampling: the chosen resampling scheme; see *resampling* module.
- **store_history: whether we should store the particles at all iterations;**
  useful in particular for smoothing, see *smoothing* module.

See the documentation of SMC for more details.

Functions

multiSMC([nruns, nprocs, out_func, collect])	Run SMC algorithms in parallel, for different combinations of parameters.

Classes

FeynmanKac(T)	Abstract base class for Feynman-Kac models.
SMC([fk, N, qmc, resampling, ESSrmin, ...])	Metaclass for SMC algorithms.

particles.datasets

Where datasets live.

This module gives access to several useful datasets. A dataset is represented as a class that inherits from base class `Dataset`. When instantiating such a class, you get an object with attributes:

- `raw_data`: data in the original file;
- `data` : data obtained after a pre-processing step was applied to the raw data.

The pre-processing step is performed by method `preprocess` of the class. For instance, for a regression dataset, the pre-processing steps normalises the predictors and adds an intercept. The pre-processing step of base class `Dataset` does nothing (`raw_data` and `data` point to the same object).

Here a quick example:

```
from particles import datasets as dts

dataset = dts.Pima()
help(dataset)  # basic info on dataset
help(dataset.preprocess)  # info on how data was pre-processed
data = dataset.data  # typically a numpy array
```

And here is a table of the available datasets; see the documentation of each sub-class for more details on the preprocessing step.

Dataset	parent class	typical use/model
`Boston`	`RegressionDataset`	regression
`Eeg`	`BinaryRegDataset`	binary regression
`GBP_vs_USD_9798`	`LogReturnsDataset`	stochastic volatility
`Liver`	`BinaryRegDataset`	binary regression
`Nutria`	`Dataset`	population ecology
`Pima`	`BinaryRegDataset`	binary regression
`Sonar`	`BinaryRegDataset`	binary regression
`Neuro`	`Dataset`	neuroscience ssm

See also utility function `prepare_predictors`, which prepares (rescales, adds an intercept) predictors/features for a regression or classification task.

Functions

`get_path`(file_name)	
`prepare_predictors`(predictors[, ...])	Rescale predictors and (optionally) add an intercept.

Classes

BinaryRegDataset(**kwargs)	Binary regression (classification) dataset.
Boston(**kwargs)	Boston house-price data of Harrison et al (1978).
Concrete(**kwargs)	Concrete compressive strength data of Yeh (1998).
Dataset(**kwargs)	Base class for datasets.
Eeg(**kwargs)	EEG dataset from UCI repository.
GBP_vs_USD_9798(**kwargs)	GBP vs USD daily rates in 1997-98.
Liver(**kwargs)	Indian liver patient dataset (ILPD).
LogReturnsDataset(**kwargs)	Log returns dataset.
Neuro(**kwargs)	Neuroscience experiment data from Temereanca et al (2008).
Nutria(**kwargs)	Nutria dataset.
Pima(**kwargs)	Pima Indians Diabetes.
RegressionDataset(**kwargs)	Regression dataset.
Sonar(**kwargs)	Sonar dataset from UCI repository.

particles.distributions

Probability distributions as Python objects.

Overview

This module lets users define probability distributions as Python objects.

The probability distributions defined in this module may be used:

- to define state-space models (see module *state_space_models*);
- to define a prior distribution, in order to perform parameter estimation (see modules *smc_samplers* and *mcmc*).

Univariate distributions

The module defines the following classes of univariate continuous distributions:

class (with signature)	comments
Beta(a=1., b=1.)	
Dirac(loc=0.)	Dirac mass at point *loc*
FlatNormal(loc=0.)	Normalp with inf variance (missing data)
Gamma(a=1., b=1.)	scale = 1/b
InvGamma(a=1., b=1.)	Distribution of 1/X for X~Gamma(a,b)
Laplace(loc=0., scale=1.)	
Logistic(loc=0., scale=1.)	
LogNormal(mu=0., sigma=1.)	Dist of Y=e^X, X ~ N(, ^2)
Normal(loc=0., scale=1.)	N(loc,scale^2) distribution
Student(loc=0., scale=1., df=3)	
TruncNormal(mu=0, sigma=1., a=0., b=1.)	N(mu, sigma^2) truncated to intervalp [a,b]
Uniform(a=0., b=1.)	uniform over intervalp [a,b]

and the following classes of univariate discrete distributions:

class (with signature)	comments
Binomial(n=1, p=0.5)	
Categorical(p=None)	returns i with prob p[i]
DiscreteUniform(lo=0, hi=2)	uniform over a, . . . , b-1
Geometric(p=0.5)	
Poisson(rate=1.)	Poisson with expectation `rate`

Note that allp the parameters of these distributions have default values, e.g.:

```
some_norm = Normal(loc=2.4)  # N(2.4, 1)
some_gam = Gamma()  # Gamma(1, 1)
```

Mixture distributions (new in version 0.4)

A (univariate) mixture distribution may be specified as follows:

```
mix = Mixture([0.5, 0.5], Normal(loc=-1), Normal(loc=1.))
```

The first argument is the vector of probabilities, the next arguments are the k component distributions.

See also `MixMissing` for defining a mixture distributions, between one component that generates the labelp "missing", and another component:

```
mixmiss = MixMissing(pmiss=0.1, base_dist=Normal(loc=2.))
```

This particular distribution is usefulp to specify a state-space model where the observation may be missing with a certain probability.

Transformed distributions

To further enrich the list of available univariate distributions, the module lets you define **transformed distributions**, that is, the distribution of Y=f(X), for a certain function f, and a certain base distribution for X.

class name (and signature)	description
LinearD(base_dist, a=1., b=0.)	Y = a * X + b
LogD(base_dist)	Y = log(X)
LogitD(base_dist, a=0., b=1.)	Y = logit((X-a)/(b-a))

A quick example:

```
from particles import distributions as dists
d = dists.LogD(dists.Gamma(a=2., b=2.))  # law of Y=log(X), X~Gamma(2, 2)
```

Note: These transforms are often used to obtain random variables defined over the fullp real line. This is convenient in particular when implementing random walk Metropolis steps.

Multivariate distributions

The module implements one multivariate distribution class, for Gaussian distributions; see `MvNormal`.

Furthermore, the module provides two ways to construct multivariate distributions from a collection of univariate distributions:

- `IndepProd`: product of independent distributions; mainly used to define state-space models.
- `StructDist`: distributions for named variables; mainly used to specify prior distributions; see modules *smc_samplers* and *mcmc* (and the corresponding tutorials).

Under the hood

Probability distributions are represented as objects of classes that inherit from base class `ProbDist`, and implement the following methods:

- `logpdf(self, x)`: computes the log-pdf (probability density function) at point `x`;
- `rvs(self, size=None)`: simulates `size` random variates; (if set to None, number of samples is either one if allp parameters are scalar, or the same number as the common size of the parameters, see below);
- `ppf(self, u)`: computes the quantile function (or Rosenblatt transform for a multivariate distribution) at point `u`.

A quick example:

```
some_dist = dists.Normal(loc=2., scale=3.)
x = some_dist.rvs(size=30)  # a (30,) ndarray containing IID N(2, 3^2) variates
z = some_dist.logpdf(x)  # a (30,) ndarray containing the log-pdf at x
```

By default, the inputs and outputs of these methods are either scalars or Numpy arrays (with appropriate type and shape). In particular, passing a Numpy array to a distribution parameter makes it possible to define "array distributions". For instance:

```
some_dist = dists.Normal(loc=np.arange(1., 11.))
x = some_dist.rvs(size=10)
```

generates 10 Gaussian-distributed variates, with respective means 1., . . . , 10. This is how we manage to define "Markov kernels" in state-space models; e.g. when defining the distribution of X_t given X_{t-1} in a state-space model:

```
class StochVol(ssm.StateSpaceModel):
    def PX(self, t, xp, x):
        return stats.norm(loc=xp)
    ### ... see module state_space_models for more details
```

Then, in practice, in e.g. the bootstrap filter, when we generate particles X_t^n, we callp method `PX` and pass as an argument a numpy array of shape (N,) containing the N ancestors.

Note: ProbDist objects are roughly similar to the frozen distributions of package `scipy.stats`. However, they are not equivalent. Using such a frozen distribution when e.g. defining a state-space modelp will return an error.

Posterior distributions

A few classes also implement a `posterior` method, which returns the posterior distribution that corresponds to a prior set to `self`, a modelp which is conjugate for the considered class, and some data. Here is a quick example:

```
from particles import distributions as dists
prior = dists.InvGamma(a=.3, b=.3)
data = random.randn(20)  # 20 points generated from N(0,1)
post = prior.posterior(data)
# prior is conjugate wrt modelp X_1, ..., X_n ~ N(0, theta)
print("posterior is Gamma(%f, %f)" % (post.a, post.b))
```

Here is a list of distributions implementing posteriors:

Distribution	Corresponding model	comments
Normalp	N(theta, sigma^2),	sigma fixed (passed as extra argument)
TruncNormalp	same	
Gamma	N(0, 1/theta)	
InvGamma	N(0, theta)	
MvNormalp	N(theta, Sigma)	Sigma fixed (passed as extra argument)

Implementing your own distributions

If you would like to create your own univariate probability distribution, the easiest way to do so is to sub-class `ProbDist`, for a continuous distribution, or `DiscreteDist`, for a discrete distribution. This willp properly set class attributes `dim` (the dimension, set to one, for a univariate distribution), and `dtype`, so that they play nicely with `StructDist` and so on. You will also have to properly define methods `rvs`, `logpdf` and `ppf`. You may omit `ppf` if you do not plan to use SQMC (Sequentialp quasi Monte Carlo).

Functions

`IID`(law, k)	Joint distribution of k iid (independent and identically distributed) variables.

Classes

`Beta`([a, b])	Beta(a,b) distribution.
`Binomial`([n, p])	Binomial(n,p) distribution.
`Categorical`([p])	Categorical distribution.
`Cond`(law[, dim, dtype])	Conditionalp distributions.
`Dirac`([loc])	Dirac mass.
`DiscreteDist`()	Base class for discrete probability distributions.
`DiscreteUniform`([lo, hi])	Discrete uniform distribution.
`FlatNormal`([loc])	Normalp with infinite variance.
`Gamma`([a, b])	Gamma(a,b) distribution, scale=1/b.
`Geometric`([p])	Geometric(p) distribution.

continues on next page

Table 1 – continued from previous page

IndepProd(*dists)	Product of independent univariate distributions.
InvGamma([a, b])	Inverse Gamma(a,b) distribution.
Laplace([loc, scale])	Laplace(loc,scale) distribution.
LinearD(base_dist[, a, b])	Distribution of Y = a*X + b.
LocScaleDist([loc, scale])	Base class for location-scale distributions.
LogD(base_dist)	Distribution of Y = log(X).
LogNormal([mu, sigma])	Distribution of Y=e^X, with X ~ N(mu, sigma^2).
Logistic([loc, scale])	Logistic(loc, scale) distribution.
LogitD(base_dist[, a, b])	Distributions of Y=logit((X-a)/(b-a)).
MixMissing([pmiss, base_dist])	Mixture between a given distribution and 'missing'.
Mixture(pk, *components)	Mixture distributions.
MvNormal([loc, scale, cov])	Multivariate Normalp distribution.
NegativeBinomial([n, p])	Negative Binomialp distribution.
Normal([loc, scale])	N(loc, scale^2) distribution.
Poisson([rate])	Poisson(rate) distribution.
ProbDist()	Base class for probability distributions.
StructDist(laws)	A distribution such that inputs/outputs are structured arrays.
Student([df, loc, scale])	Student distribution.
TransformedDist(base_dist)	Base class for transformed distributions.
TruncNormal([mu, sigma, a, b])	Normal(mu, sigma^2) truncated to [a, b] interval.
Uniform([a, b])	Uniform([a,b]) distribution.
VaryingCovNormal([loc, cov])	Multivariate Normalp (varying covariance matrix).

particles.hilbert

Hilbert curve and its inverse, in any dimension.

Functions

Hilbert_to_int(coords)	
child_start_end(parent_start, parent_end, ...)	
gray_decode(n)	
gray_decode_travel(start, end, mask, g)	
gray_encode(bn)	
gray_encode_travel(start, end, mask, i)	
hilbert_array(xint)	Compute Hilbert indices.
hilbert_sort(x)	Hilbert sort: sort vectors according to their Hilbert index.
initial_start_end(nChunks, nD)	
int_to_Hilbert(i[, nD])	
invlogit(x)	
pack_coords(chunks, nD)	
pack_index(chunks, nD)	
transpose_bits(srcs, nDests)	
unpack_coords(coords)	
unpack_index(i, nD)	

particles.hmm

Baum-Welch filter/smoother for hidden Markov models.

Overview

A hidden Markov model (HMM) is a state-space model with a finite state-space, {1, ..., K}. The Baum-Welch algorithm allows to compute (exactly) the filtering and smoothing distributions of such a model; i.e. the probabilities that X_t=k given data Y_{0:t} (filtering) or the complete data Y_{0:T} (smoothing). In addition, one may sample trajectories from the complete smoothing distribution (the distribution of all the states, X_{0:T}, given all the data).

Hidden Markov models and the Baum-Welch algorithm are covered in Chapter 6 of the book.

Defining a hidden Markov model

Hidden Markov models are represented as `HMM` objects; `HMM` is a subclass of `StateSpaceModel` (from module *state_space_models*), which assigns:

- A categorical distribution to X_0 (parameter `init_dist`)
- A categorical distribution to X_t given X_{t-1} (parameter `trans_mat`)

The distributions of Y_t|X_t must be defined by sub-classing `HMM`. For instance, this module defines a `GaussianHMM` class as follows:

```
class GaussianHMM(HMM):
    default_params = {'mus': None, 'sigmas': None}
    default_params.update(HMM.default_params)

    def PY(self, t, xp, x):
        return dists.Normal(loc=self.mus[x], scale=self.sigmas[x])
```

One may now define a particular model in the usual way:

```
tm = np.array([[0.9 0.1], [0.2, 0.8]])
my_hmm = hmm.GaussianHMM(mus=np.array([0., 1.], sigmas=np.ones(2),
                         trans_mat=tm)
```

and e.g. sample data from this model:

```
true_states, y = my_hmm.simulate(100)
```

(This works because, again, `HMM` is a subclass of `StateSpaceModels`).

Warning: Since `HMM` is a subclass of `StateSpaceModel`, method `PY` has the same signature as in its parent class, but argument `xp` is not used. In other words, you cannot specify a HMM model where Y_t would depend on both X_t and X_{t-1} (unlike in the general case).

Running the Baum-Welch algorithm

Class `BaumWelch` is instantiated as follows:

```
bw = BaumWelch(hmm=my_hmm, data=y)
```

To actually run the algorithm, one must invoke the appropriate methods, e.g.:

```
bw.forward()  # computes the filtering probs
bw.backward()  # computes the marginal smoothing probs
bw.sample(N=30)  # generate 30 smoothing trajectories
```

If you invoke either `backward` or `sample` directly, the forward pass will be run first. The output of the forward and backward passes are attributes of object `bw`, which are lists of K-length numpy arrays. For instance, `self.filt` is a list of arrays containing the filtering probabilities; see the documentation of `BaumWelch` for more details.

Running the forward pass step by step

A `BaumWelch` object is an iterator; each iteration performs a single step of the forward pass. It is thus possible for the user to run the forward pass step by step:

```
next(bw)  # performs one step of the forward pass
```

This may be useful in a variety of scenarios, such as when data are acquired on the fly (in that case, modify attribute `self.data` directly), or when one wants to perform the smoothing pass at different times; in particular:

```
bw = BaumWelch(hmm=mh_hmm, data=y)
for t, _ in enumerate(y):
    bw.step()
    bw.backward()
    ## save the results in bw.smth somewhere
```

would compute all the intermediate smoothing distributions (for data Y_0, then $Y_{0:1}$, and so on). This is expensive, of course (cost is O(T^2)).

Classes

`BaumWelch`([hmm, data])	Baum-Welch filtering/smoothing algorithm.
`GaussianHMM`(**kwargs)	Gaussian HMM: $Y_t \| X_t = k \sim N(\mu_k, \sigma_k^2)$
`HMM`(**kwargs)	Base class for hidden Markov models.

particles.kalman

Basic implementation of the Kalman filter (and smoother).

Overview

The Kalman filter/smoother is a well-known algorithm for computing recursively the filtering/smoothing distributions of a linear Gaussian model, i.e. a model of the form:

$$
\begin{aligned}
X_0 &\sim N(\mu_0, C_0) \\
X_t &= FX_{t-1} + U_t, \quad U_t \sim N(0, C_X) \\
Y_t &= GX_t + V_t, \quad V_t \sim N(0, C_Y)
\end{aligned}
$$

Linear Gaussian models and the Kalman filter are covered in Chapter 7 of the book.

MVLinearGauss class and subclasses

To define a specific linear Gaussian model, we instantiate class `MVLinearGauss` (or one its subclass) as follows:

```
import numpy as np
from particles import kalman

ssm = kalman.MVLinearGauss(F=np.eye(2), G=np.ones((1, 2)), covX=np.eye(2),
                           covY=.3)
```

where the parameters have the same meaning as above. It is also possible to specify `mu0`and `cov0` (the mean and covariance of the initial state X_0). (See the documentation of the class for more details.)

Class `MVLinearGauss` is a sub-class of `StateSpaceModel` in module *state_space_models*, so it inherits methods from its parent such as:

```
true_states, data = ssm.simulate(30)
```

Class `MVLinearGauss` implements methods `proposal`, `proposal0` and `logeta`, which correspond respectively to the optimal proposal distributions and auxiliary function for a guided or auxiliary particle filter; see Chapter 11 and module *state_space_models* for more details. (That the optimal quantities are tractable is, of course, due to the fact that the model is linear and Gaussian.)

To define a univariate linear Gaussian model, you may want to use instead the more conveniently parametrised class `LinearGauss` (which is a sub-class of `MVLinearGauss`):

```
ssm = LinearGauss(rho=0.3, sigmaX=1., sigmaY=.2, sigma0=1.)
```

which corresponds to model:

$$X_0 \sim N(0, \sigma_0^2)$$
$$X_t | X_{t-1} = x_{t-1} \sim N(\rho * X_{t-1}, \sigma_X^2)$$
$$Y_t | X_t = x_t \sim N(x_t, \sigma_Y^2)$$

Another sub-class of `MVLinearGauss` defined in this module is `MVLinearGauss_Guarniero_etal`, which implements a particular class of linear Gaussian models often used as a benchmark (after Guarniero et al, 2016).

`Kalman` class

The Kalman filter is implemented as a class, `Kalman`, with methods `filter` and `smoother`. When instantiating the class, one passes as arguments the data, and an object that represents the considered model (i.e. an instance of MvLinearGauss, see above):

```
kf = kalman.Kalman(ssm=ssm, data=data)
kf.filter()
```

The second line implements the forward pass of a Kalman filter. The results are stored as lists of `MeanAndCov` objects, that is, named tuples with attributes 'mean' and 'cov' that represent a Gaussian distribution. For instance:

```
kf.filt[3].mean  # mean of the filtering distribution at time 3
kf.pred[7].cov  # cov matrix of the predictive distribution at time 7
```

The forward pass also computes the log-likelihood of the data:

```
kf.logpyt[5]  # log-density of Y_t | Y_{0:t-1} at time t=5
```

Smoothing works along the same lines:

```
kf.smoother()
```

then object kf contains a list called smooth, which represents the successive (marginal) smoothing distributions:

```
kf.smth[8].mean  # mean of the smoothing dist at time 8
```

It is possible to call method `smoother` directly (without calling `filter` first). In that case, the filtering step is automatically performed as a preliminary step.

Kalman objects as iterators

It is possible to perform the forward pass step by step; in fact a `Kalman` object is an iterator:

```
kf = kalman.Kalman(ssm=ssm, data=data)
next(kf)  # one step
next(kf)  # one step
```

If you run the smoother after k such steps, you will obtain the smoothing distribution based on the k first data-points. It is therefore possible to compute recursively the successive smoothing distributions, but (a) at a high CPU cost; and (b) at each time, you must save the results somewhere, as attribute `kf.smth` gets written over and over.

Functions to perform a single step

The module also defines low-level functions that perform a single step of the forward or backward step. Some of these function makes it possible to perform such steps *in parallel* (e.g. for N predictive means). The table below lists these functions. Some of the required inputs are `MeanAndCov` objects, which may be defined as follows:

```
my_predictive_dist = kalman.MeanAndCov(mean=np.ones(2), cov=np.eye(2))
```

Function (with signature)
predict_step(F, covX, filt)
filter_step(G, covY, pred, yt)
filter_step_asarray(G, covY, pred, yt)
smoother_step(F, filt, next_pred, next_smth)

Functions

`dotdot`(a, b, c)	
`dotdotinv`(a, b, c)	a * b * c^{-1}, where c is symmetric positive
`filter_step`(G, covY, pred, yt)	Filtering step of Kalman filter.
`filter_step_asarray`(G, covY, pred, yt)	Filtering step of Kalman filter: array version.
`predict_step`(F, covX, filt)	Predictive step of Kalman filter.
`smoother_step`(F, filt, next_pred, next_smth)	Smoothing step of Kalman filter/smoother.

Classes

`Kalman`([ssm, data])	Kalman filter/smoother.
`LinearGauss`(**kwargs)	A basic (univariate) linear Gaussian model.
`MVLinearGauss`([F, G, covX, covY, mu0, cov0])	Multivariate linear Gaussian model.
`MVLinearGauss_Guarniero_etal`([alpha, dx])	Special case of a MV Linear Gaussian ssm from Guarnierio et al. (2016).
`MeanAndCov`(mean, cov)	Create new instance of MeanAndCov(mean, cov)

particles.mcmc

MCMC (Markov chain Monte Carlo) and related algorithms.

Overview

This module contains various classes that implement MCMC samplers:

- `MCMC`: the base class for all MCMC samplers;
- `GenericRWHM`: base class for random-walk Hastings-Metropolis;
- `GenericGibbs`: base class for Gibbs samplers;
- `PMMH`, `ParticleGibbs`: base classes for the PMCMC (particle MCMC algorithms) with the same name.

For instance, here is how to run 200 iterations of an adaptive random-walk sampler:

```
# ...
# define some_static_model, some_prior
# ...
my_mcmc = BasicRWHM(model=some_static_model, prior=some_prior, niter=200,
                    adaptive=True)
my_mcmc.run()
```

Upon completion, object `my_mcmc` have an attribute called `chain`, which is a `ThetaParticles` object (see module *smc_samplers*). In particular, `my_mcmc.chain` has the following attributes:

- `theta`: a structured array that contains the 200 simulated parameters;
- `lpost`: an array that contains the log-posterior density at these 200 parameters.

See the dedicated notebook tutorial (on Bayesian inference for state-space models) for more examples and explanations.

Random walk Metropolis

Both `GenericRWHM` and `PMMH` rely on a random walk proposal; that is, given the current point theta, the proposed point is sampled from a Gaussian distribution, with mean theta, and some covariance matrix Sigma (or Sigma_t at iteration t the adaptive case, see below). Various parameters may be set to tune this type of proposal:

- to run a standard random walk Metropolis algorithm, set `adaptive=False`, and specify the covariance matrix Sigma through parameter `rw_cov`. Otherwise, Sigma is set by default to the identity matrix (which could a terrible choice in certain problems).
- to run an adaptive random walk Metropolis algorithm, where the matrix Sigma_t is progressively adapted to the past states of the chain, set `adaptive=True`. In that case, Sigma_t is set to a fraction of the running estimate of the covariance matrix of the *target* distribution, and `rw_cov` is used as a preliminary estimate for that target covariance. See parameter `scale` (in the documentation of `GenericRWHM`) for more details on the factor in front of the running estimate, and `VanishCovTracker` for how the running estimate is computed recursively.

By default, the adaptive version is used.

Beyond the bootstrap filter within PMMH

PMMH runs, at each iteration, a particle filter to approximate the likelihood of the considered state-space model. By default, a bootstrap filter is used (with default choices for the resampling scheme, and so on). It is possible to change the settings of this filtering algorithm, or run a different type of algorithm, as follows:

1. You can set parameter fk_cls to a FeynmanKac subclass such as e.g. ssms.GuidedPF if you wish to use a guided filter (rather than a bootstrap filter). See module *state_space_models* for more details on these FeynmanKac subclasses derived from a given state-space model.
2. You can use parameter smc_options (dict-like) to pass various parameters to class SMC when the algorithm is instantiated.
3. You can even use parameter smc_cls to specify a different class for the

   algorithm itself (a SMC subclass instead of SMC itself).
4. Finally, if you need something even more general, you can also subclass

   PMMH and redefine method alg_instance, which takes as argument theta (a dict-like object) and returns an algorithm (an instance of class SMC or one its subclasses).

Functions

msjd(theta)	Mean squared jumping distance.

Classes

BasicRWHM([niter, verbose, theta0, ...])	Basic random walk Hastings-Metropolis sampler.
CSMC([fk, N, ESSrmin, xstar])	Conditional SMC.
GenericGibbs([niter, verbose, theta0, ...])	Generic Gibbs sampler for a state-space model.
GenericRWHM([niter, verbose, theta0, ...])	Base class for random walk Hasting-Metropolis samplers.
MCMC([niter, verbose])	MCMC base class.
PMMH([niter, verbose, ssm_cls, smc_cls, ...])	Particle Marginal Metropolis Hastings.
ParticleGibbs([niter, verbose, ssm_cls, ...])	Particle Gibbs sampler (abstract class).
VanishCovTracker([alpha, dim, mu0, Sigma0])	Tracks the vanishing mean and covariance of a sequence of points.

particles.nested

Nested sampling (vanilla and SMC).

Warning: This module is less tested than the rest of the package.

Moreover, this documentation does not explain precisely how nested sampling works (and this topic is not covered in our book). Thus, refer to e.g. the original papers of Skilling or Chopin and Robert (2010, Biometrika). For nested sampling SMC, see the paper of Salomone et al (2018).

Vanilla nested sampling

This module contains classes that implement nested sampling:

- `NestedSampling`: base class;
- `Nested_RWmoves` : nested sampling algorithm based on random walk Metropolis steps.

To use the latter, you need to define first a static model, in the same way as in the *smc_samplers* module. For instance:

```
import particles
from particles import smc_samplers as ssp
from particles import distributions as dists

class ToyModel(ssp.smc_samplers):
    def logpyt(self, theta, t):  # log-likelihood of data-point at time t
        return stats.norm.logpdf(self.data[t], loc=theta)

my_prior = dists.Normal()  # theta ~ N(0, 1)
y = random.normal(size=1)  # y | theta ~ N(theta, 1)
toy_model = ToyModel(data=y, prior=my_prior)
```

Then, the algorithm may be set and run as follows:

```
from particles import nested

algo = nested.Nested_RWmoves(model=toy_model, N=1000, nsteps=3, eps=1e-8)
algo.run()
print('estimate of log-evidence: %f' % algo.lZhats[-1])
```

This will run a nested sampling algorithm which propagates N=1000 particles; each time a point is deleted, it is replaced by another point obtained as follows: another point is selected at random, and then moved through `nsteps` steps of a Gaussian random walk Metropolis kernel (which leaves invariant the prior constrained to the current likelihood contour). To make these steps reasonably efficient, the covariance matrix of the random walk proposal is dynamically adapted to the current sample of points. The algorithm is stopped when the different between the two most recent estimates of the log-evidence is below `eps`.

To implement your own algorithm, you must sub-class `NestedSampling` like this:

```
from particles import nested

class MyNestedSampler(nested.NestedSampling):
    def mutate(self, n, m):
        # implement a MCMC step that replace point X[n] with the point
        # obtained by starting at X[m] and doing a certain number of steps
        return value
```

Nested sampling Sequential Monte Carlo

Salomone et al (2018) proposed a SMC sampler inspired by nested sampling. The target distribution at time t is the prior constrained to the likelihood being larger than constant l_t. These constants may be chosen adaptively: in this implementation, the next l_t is set to the `ESSrmin` upper-quantile of the likelihood of the current points (where `ESSrmin` is specified by the user). In other words, l_t is chosen so that the ESS equals this value. (`ESSrmin` corresponds to 1 - rho in Salomone et al's notations.)

This module implements this SMC sampler as `NestedSamplingSMC`, a sub-class of `smc_samplers.FKSMCsampler`, which may be used the same way as other SMC samplers defined in module *smc_samplers*:

```
fk = nested.NestedSamplingSMC(model=toy_model, wastefree=True, ESSrmin=0.3)
alg = particles.SMC(fk=fk, N=1_000)
alg.run()
```

Upon completion, the dictionary `alg.X.shared` will contain the successive estimates of the log-evidence (log of marginal likelihood, in practice the final one is the one you want to use), and the successive values of l_t.

Note that a waste-free version of NS-SMC is run by default, but the original paper of Salomone et al (which predates NS-SMC) only considers a standard version. (For more details on waste-free SMC vs standard SMC, see module *smc_samplers* and the corresponding jupyter notebook.)

Reference

Salomone, South L., Drovandi C. and Kroese D. (2018). Unbiased and Consistent Nested Sampling via Sequential Monte Carlo, arxiv 1805.03924.

Functions

`unif_minus_one`(N, m)	Sample uniformly from 0, ..., N-1, minus m.
`xxT`(x)	

Classes

`MeanCovTracker`(x)	Tracks mean and cov of a set of points.
`NestedParticles`([theta, lprior, llik])	
`NestedSampling`([model, N, eps])	Base class for nested sampling algorithms.
`NestedSamplingSMC`([model, wastefree, ...])	Feynman-Kac class for the nested sampling SMC algorithm.
`Nested_RWmoves`([model, N, eps, nsteps, scale])	Nested sampling with (adaptive) random walk Metropolis moves.

particles.resampling

Resampling and related numerical algorithms.

Overview

This module implements resampling schemes, plus some basic numerical functions related to weights and weighted data (ESS, weighted mean, etc). The recommended import is:

```
from particles import resampling as rs
```

Resampling is covered in Chapter 9 of the book.

Resampling schemes

All the resampling schemes are implemented as functions with the following signature:

```
A = rs.scheme(W, M=None)
```

where:

- `W` is a vector of N normalised weights (i.e. positive and summing to one).
- `M` (int) is the number of resampled indices that must be generated; (optional, set to N if not provided).
- `A` is a ndarray containing the M resampled indices (i.e. ints in the range 0, . . . , N-1).

Here the list of currently implemented resampling schemes:

- `multinomial`
- `residual`
- `stratified`
- `systematic`
- `ssp`
- `killing`

If you don't know much about resampling, it's best to use the default scheme (systematic). See Chapter 9 of the book for a discussion.

Alternative ways to sample from a multinomial distribution

Function `multinomial` samples efficiently M times from the multinomial distribution that produces output n with probability `W[n]`. It does so using an algorithm with complexity O(M+N), as explained in Section 9.4 of the book. However, this function is not really suited:

1. if you don't want to get ordered samples, but truly IID ones;
1. if you want to draw only **once** from that distribution;
2. If you do not know in advance how many draws you need.

The three functions below cover these scenarios:

- `multinomial_iid`

- multinomial_once
- MultinomialQueue

Weights objects

Objects of class SMC, which represent the output of a particle filter, have an attribute called wgts, which is an instance of class Weights. The attributes of that object are:

- lw: the N un-normalised log-weights
- W: the N normalised weights (sum equals one)
- ESS: the effective sample size (1/sum(W^2))

For instance:

```
pf = particles.SMC(fk=some_fk_model, N=100)
pf.run()
print(pf.wgts.ESS)  # The ESS of the final weights
```

The rest of this section should be of interest only to advanced users (who wish for instance to subclass SMC in order to define new particle algorithms). Basically, class Weights is used to automate and abstract away the computation of the normalised weights and their ESS. Here is a quick example:

```
from numpy import random

wgts = rs.Weights(lw=random.randn(10))  # we provide log-weights
print(wgts.W)  # the normalised weights have been computed automatically
print(wgts.ESS)  # and so the ESS
incr_lw = 3. * random.randn(10)  # incremental weights
new_wgts = wgts.add(incr_lw)
print(new_wgts.ESS)  # the ESS of the new weights
```

Warning: Weights objects should be considered as immutable: in particular method add returns a new Weights object. Trying to modify directly (in place) a Weights object may introduce hairy bugs. Basically, SMC and the methods of ParticleHistory do not *copy* such objects, so if you modify them later, then you also modify the version that has been stored at a previous iteration.

Other functions related to resampling

The following basic functions are called by some resampling schemes, but they might be useful in other contexts.

- inverse_cdf
- uniform_spacings

Other functions of interest

In *particles*, importance weights and similar quantities are always computed and stored on the log-scale, to avoid numerical overflow. This module also contains a few basic functions to deal with log-weights:

- essl
- exp_and_normalise
- log_mean_exp
- log_sum_exp
- wmean_and_var
- wmean_and_var_str_array
- wquantiles

Functions

essl(lw)	ESS (Effective sample size) computed from log-weights.
exp_and_normalise(lw)	Exponentiate, then normalise (so that sum equals one).
idiotic(W, M)	Idiotic resampling.
inverse_cdf(su, W)	Inverse CDF algorithm for a finite distribution.
killing(W, M)	Killing resampling.
log_mean_exp(v[, W])	Returns log of (weighted) mean of exp(v).
log_sum_exp(v)	Log of the sum of the exp of the arguments.
log_sum_exp_ab(a, b)	log_sum_exp for two scalars.
multinomial(W, M)	Multinomial resampling.
multinomial_iid(W[, M])	Multinomial resampling (IID draws).
multinomial_once(W)	Sample once from a Multinomial distribution.
resampling(scheme, W[, M])	
resampling_scheme(func)	Decorator for resampling schemes.
residual(W, M)	Residual resampling.
ssp(W, M)	SSP resampling.
stratified(W, M)	Stratified resampling.
systematic(W, M)	Systematic resampling.
uniform_spacings(N)	Generate ordered uniform variates in O(N) time.
wmean_and_cov(W, x)	Weighted mean and covariance matrix.
wmean_and_var(W, x)	Component-wise weighted mean and variance.
wmean_and_var_str_array(W, x)	Weighted mean and variance of each component of a structured array.
wquantiles(W, x[, alphas])	Quantiles for weighted data.
wquantiles_str_array(W, x[, alphas])	quantiles for weighted data stored in a structured array.

Classes

MultinomialQueue(W[, M])	On-the-fly generator for the multinomial distribution.
Weights([lw])	A class to store N log-weights, and automatically compute normalised weights and their ESS.

particles.rqmc

Randomised quasi-Monte Carlo sequences.

Functions

halton(N, d)	
latin(N, d)	
safe_generate(N, d, engine_cls)	
sobol(N, d)	

particles.smc_samplers

Classical and waste-free SMC samplers.

Overview

This module implements SMC samplers, that is, SMC algorithms that may sample from a sequence of arbitrary probability distributions (and approximate their normalising constants). Applications include sequential and non-sequential Bayesian inference, rare-event simulation, etc. For more background on (standard) SMC samplers, see Chapter 17 (and references therein). For the waste-free variant, see Dau & Chopin (2022).

The following type of sequences of distributions are implemented:

- SMC tempering: target distribution at time t has a density of the form mu(theta) L(theta)^{gamma_t}, with gamma_t increasing from 0 to 1.
- IBIS: target distribution at time t is the posterior of parameter theta given data Y_{0:t}, for a given model.
- SMC^2: same as IBIS, but a for state-space model. For each theta-particle, a local particle filter is run to approximate the likelihood up to time t; see Chapter 18 in the book.
- Nested sampling: target at time t is prior constrained to likelihood being above threshold l_t, with an increasing sequence of l_t's (inspired by Salomone et al, 2018).

SMC samplers for binary distributions (and variable selection) are implemented elsewhere, in module *binary_smc*.

Before reading the documentation below, you might want to have a look at the following notebook tutorial, which may be a more friendly introduction.

Target distribution(s)

If you want to use a SMC sampler to perform Bayesian inference, you may specify your model by sub-classing `StaticModel`, and defining method `logpyt` (the log density of data Y_t, given previous datapoints and parameter values) as follows:

```
class ToyModel(StaticModel):
    def logpyt(self, theta, t):  # density of Y_t given parameter theta
        return -0.5 * (theta['mu'] - self.data[t])**2 / theta['sigma2']
```

In this example, theta is a structured array, with fields named after the different parameters of the model. For the sake of consistency, the prior should be a `distributions.StructDist` object (see module `distributions` for more details), whose inputs and outputs are structured arrays with the same fields:

```
from particles import distributions as dists

prior = dists.StructDist(mu=dists.Normal(scale=10.),
                         sigma2=dists.Gamma())
```

Then you can instantiate the class as follows:

```
data = np.random.randn(20)  # simulated data
my_toy_model = ToyModel(prior=prior, data=data)
```

This object may be passed as an argument to the `FeynmanKac` classes that define SMC samplers, see below.

Under the hood, class `StaticModel` defines methods `loglik` and `logpost` which computes respectively the log-likelihood and the log posterior density of the model at a certain time.

What if I don't want to do Bayesian inference

This is work in progress, but if you just want to sample from some target distribution, using SMC tempering, you may define your target as follows:

```
class ToyBridge(TemperingBridge):
    def logtarget(self, theta):
        return -0.5 * np.sum(theta**2, axis=1)
```

and then define:

```
base_dist = dists.MvNormal(scale=10., cov=np.eye(10))
toy_bridge = ToyBridge(base_dist=base_dist)
```

Note that, this time, we went for standard, bi-dimensional numpy arrays for argument theta. This is fine because we use a prior object that also uses standard numpy arrays.

FeynmanKac objects

SMC samplers are represented as `FeynmanKac` classes. For instance, to perform SMC tempering with respect to the bridge defined in the previous section, you may do:

```
fk_tpr = AdaptiveTempering(model=toy_bridge, len_chain=100)
alg = SMC(fk=fk_tpr, N=200)
alg.run()
```

This piece of code will run a tempering SMC algorithm such that:

- the successive exponents are chosen adaptively, so that the ESS between two successive steps is cN, with c=1/2 (use parameter ESSrmin to change the value of c).
- the waste-free version is implemented; that is, the actual number of particles is 100 * 200, but only 200 particles are resampled at each time, and then moved through 100 MCMC steps (parameter len_chain) (set parameter wastefree=False to run a standard SMC sampler). See Dau & Chopin (2022) for more details (or the notebook on SMC samplers).
- if you use the waste-free version, you may also compute a single-run estimate of the variance of a particular estimate; see `var_wf`, `Var_logLt` and `Var_phi` for more details.
- the default MCMC strategy is random walk Metropolis, with a covariance proposal set to a fraction of the empirical covariance of the current particle sample. See next section for how to use a different MCMC kernel.

To run IBIS instead, you may do:

```
fk_ibis = IBIS(model=toy_model, len_chain=100)
alg = SMC(fk=fk_ibis, N=200)
```

Again see the notebook tutorials for more details and examples.

Under the hood

ThetaParticles

In a SMC sampler, a particle sample is represented as a `ThetaParticles` object `X`, which contains several attributes such as, e.g.:

- `X.theta`: a structured array of length N, representing the N particles (or alternatively a numpy array of shape (N,d))
- `X.lpost`: a numpy float array of length N, which stores the log-target density of the N particles.
- `X.shared`: a dictionary that contains meta-information on the N particles; for instance it may be used to record the successive acceptance rates of the Metropolis steps.

Details may vary in a given algorithm; the common idea is that attribute `shared` is the only one which not behave like an array of length N. The main point of `ThetaParticles` is to implement fancy indexing, which is convenient for e.g. resampling:

```
from particles import resampling as rs

A = rs.resampling('multinomial', W)  # an array of N ints
Xp = X[A]  # fancy indexing
```

MCMC schemes

A MCMC scheme (e.g. random walk Metropolis) is represented as an `ArrayMCMC` object, which has two methods:

- `self.calibrate(W, x)`: calibrate (tune the hyper-parameters of) the MCMC kernel to the weighted sample (W, x).
- `self.step(x)`: apply a single step to the `ThetaParticles` object `x`, in place.

Furthermore, the different ways one may repeat a given MCMC kernel is represented by a `MCMCSequence` object, which you may pass as an argument when instantiating the `FeynmanKac` object that represents the algorithm you want to run:

```
move = MCMCSequenceWF(mcmc=ArrayRandomWalk(), len_chain=100)
fk_tpr = AdaptiveTempering(model=toy_bridge, len_chain=100, move=move)
# run a waste-free SMC sampler where the particles are moved through 99
# iterations of a random walk Metropolis kernel
```

Such objects may either keep all intermediate states (as in waste-free SMC, see sub-class `MCMCSequenceWF`) or only the states of the last iteration (as in standard SMC, see sub-class `AdaptiveMCMCSequence`).

The bottom line is: if you wish to implement a different MCMC scheme to move the particles, you should sub-class `ArrayMCMC`. If you wish to implement a new strategy to repeat several MCMC steps, you should sub-cass MCMCSequence (or one of its sub-classes).

References

Dau, H.D. and Chopin, N. Waste-free Sequential Monte Carlo, J. R. Stat. Soc. Ser. B. Stat. Methodol. 84, 1 (2022), 114–148, arxiv:2011.02328, doi:10.1111/rssb.12475

Functions

`all_distinct`(l, idx)	Returns the list [l[i] for i in idx] When needed, objects l[i] are replaced by a copy, to make sure that the elements of the list are all distinct.
`gen_concatenate`(*xs)	
`next_annealing_epn`(epn, alpha, lw)	Find next annealing exponent by solving ESS(exp(lw)) = alpha * N.
`rec_to_dict`(arr)	Turns record array *arr* into a dict
`var_wf`(smc, phi)	Single-run variance estimate for waste-free SMC.
`view_2d_array`(theta)	Returns a view to record array theta which behaves like a (N,d) float array.

Classes

AdaptiveMCMCSequence([mcmc, len_chain, ...])	MCMC sequence for a standard SMC sampler (keep only final states).
AdaptiveTempering([model, wastefree, ...])	Feynman-Kac class for adaptive tempering SMC.
ArrayIndependentMetropolis([scale])	Independent Metropolis (Gaussian proposal).
ArrayMCMC()	Base class for a (single) MCMC step applied to an array.
ArrayMetropolis()	Base class for Metropolis steps (whatever the proposal).
ArrayRandomWalk()	Gaussian random walk Metropolis.
FKSMCsampler([model, wastefree, len_chain, move])	Base FeynmanKac class for SMC samplers.
FancyList(data)	A list that implements fancy indexing, and forces elements to be distinct.
IBIS([model, wastefree, len_chain, move])	
ImportanceSampler([model, proposal])	Importance sampler.
MCMCSequence([mcmc, len_chain])	Base class for a (fixed length or adaptive) sequence of MCMC steps.
MCMCSequenceWF([mcmc, len_chain])	MCMC sequence for a waste-free SMC sampler (keep all intermediate states).
SMC2([ssm_cls, prior, data, smc_options, ...])	Feynman-Kac subclass for the SMC^2 algorithm.
StaticModel([data, prior])	Base class for static models.
Tempering([model, wastefree, len_chain, ...])	Feynman-Kac class for tempering SMC (fixed exponents).
TemperingBridge([base_dist])	**param data** data
ThetaParticles([shared])	Base class for particle systems for SMC samplers.
Var_logLt(**kwargs)	Collects single-run estimates of the variance of log L_t (normalising cst) from a waste-free sampler (Dau & Chopin, 2022).
Var_phi(**kwargs)	Collects single-run estimates of the variance of a given estimate (for a certain function phi) from a waste-free sampler (Dau & Chopin, 2022).

particles.smoothing

Off-line particle smoothing algorithms.

Overview

This module implements:

1. particle history classes, which store the full or partial history of a SMC algorithm.
2. off-line smoothing algorithms as methods of these classes.

For on-line smoothing, see instead the *collectors* module.

History classes

A `SMC` object has a `hist` attribute, which is used to record at *certain* times t:

- the N current particles X_t^n;
- their weights;
- (optionally, see below), the ancestor variables A_t^n.

The frequency at which history is recorded depends on option `store_history` of class `SMC`. Possible options are:

- `True`: records full history (at every time t);
- `False`: no history (attribute `hist` set to `None`);
- callable `f`: history is recorded at time t if `f(t)` returns True
- int k: records a rolling window history of length k (may be used to perform fixed-lag smoothing)

This module implements different classes that correspond to the different cases:

- `ParticleHistory`: full history (based on lists)
- `PartialParticleHistory`: partial history (based on dictionaries)
- `RollingParticleHistory`: rolling window history (based on deques)

All these classes provide a similar interface. If `smc` is a `SMC` object, then:

- `smc.hist.X[t]` returns the N particles at time t
- `smc.hist.wgts[t]` returns the N weights at time t (see `resampling.weights`)
- `smc.hist.A[t]` returns the N ancestor variables at time t

Partial History

Here are some examples on how one may record history only at certain times:

```
# store every other 10 iterations
smc = SMC(fk=fk, N=100, store_history=lambda t: (t % 10) == 0)

# store at certain times given by a list
times = [10, 30, 84]
smc = SMC(fk=fk, N=100, store_history=lambda t: t in times)
```

Once the algorithm is run, `smc.hist.X` and `smc.hist.wgts` are dictionaries, the keys of which are the times where history was recorded. The ancestor variables are not recorded in that case:

```
smc.run()
smc.hist.X[10]  # the N particles at time 10
smc.hist.A[10]  # raises an error
```

Full history, off-line smoothing algorithms

For a given state-space model, off-line smoothing amounts to approximate the distribution of the complete trajectory $X_{0:T}$, given data $y_{0:T}$, at some fixed time horizon T. The corresponding algorithms take as an input the complete history of a particle filter, run until time T (forward pass). Say:

```
# forward pass
fk = ssm.Bootstrap(ssm=my_ssm, data=y)
pf = particles.SMC(fk=fk, N=100, store_history=True)
pf.run()
```

Then, `pf.hist` is an instance of class `ParticleHistory`. It implements two types of approaches:

- two-filter smoothing: uses two particle filters (one forward, one backward) to estimate marginal expectations; see `two_filter_smoothing`.
- FFBS (forward filtering backward sampling): uses one particle filter, then generates trajectories from its history, using different methods (exact, rejection, MCMC, QMC). See `backward_sampling_mcmc`, `backward_sampling_ON2`, `backward_sampling_reject`, and `backward_sampling_qmc`. Recommended method is `backward_sampling_mcmc`, see discussion in Dang & Chopin (2022).

For more details, see the documentation of `ParticleHistory`, the ipython notebook on smoothing, Chapter 12 of the book, and Dang & Chopin (2022).

Warning: the complete history of a particle filter may take a lot of memory.

Rolling history, Fixed-lag smoothing

To obtain a rolling window (fixed-length) history:

```
smc = SMC(fk=fk, N=100, store_history=10)
smc.run()
```

In that case, fields `smc.hist.X`, `smc.hist.wgts` and `smc.hist.A` are deques of max length 10. Using negative indices:

```
smc.hist.X[-1]  # the particles at final time T
smc.hist.X[-2]  # the particles at time T - 1
# ...
smc.hist.X[-10] # the N particles at time T - 9
smc.hist.X[-11] # raises an error
```

Note that this type of history makes it possible to perform fixed-lag smoothing as follows:

```
B = smc.hist.compute_trajectories()
# B[t, n] is index of ancestor of X_T^n at time t
phi = lambda x: x  # any test function
est = np.average(phi(smc.hist.X[-10][B[-10, :]]), weights=smc.W)
# est is an estimate of E[ phi(X_{T-9}) | Y_{0:T}]
```

Note: recall that it is possible to run `SMC` algorithms step by step, since they are iterators. Hence it is possible to do fixed-lag smoothing step-by-step as well.

Functions

generate_hist_obj(option, smc)	
smoothing_worker([method, N, fk, fk_info, ...])	Generic worker for off-line smoothing algorithms.

Classes

PartialParticleHistory(func)	Partial history.
ParticleHistory(fk, qmc)	Particle history.
RollingParticleHistory(length)	Rolling window history.

particles.state_space_models

State-space models as Python objects.

Overview

This module defines:

1. the StateSpaceModel class, which lets you define a state-space model as a Python object;
2. FeynmanKac classes that automatically define the Bootstrap, guided or auxiliary Feynman-Kac models associated to a given state-space model;
3. several standard state-space models (stochastic volatility, bearings-only tracking, and so on).

The recommended import is:

```
from particles import state_space_models as ssms
```

For more details on state-space models and their properties, see Chapters 2 and 4 of the book.

Defining a state-space model

Consider the following (simplified) stochastic volatility model:

$$\begin{aligned} Y_t|X_t = x_t &\sim N(0, e^{x_t}) \\ X_t|X_{t-1} = x_{t-1} &\sim N(0, \rho x_{t-1}) \\ X_0 &\sim N(0, \sigma^2/(1-\rho^2)) \end{aligned}$$

To define this particular model, we sub-class StateSpaceModel as follows:

```
import numpy as np
from particles import distributions as dists

class SimplifiedStochVol(ssms.StateSpaceModel):
    default_params = {'sigma': 1., 'rho': 0.8}  # optional
    def PY(self, t, xp, x):  # dist of Y_t at time t, given X_t and X_{t-1}
```

(continues on next page)

(continued from previous page)

```
        return dists.Normal(scale=np.exp(x))
    def PX(self, t, xp):  # dist of X_t at time t, given X_{t-1}
        return dists.Normal(loc=self.mu + self.rho * (xp - self.mu),
                            scale=self.sigma)
    def PX0(self):  # dist of X_0
        return dists.Normal(scale=self.sigma / np.sqrt(1. - self.rho**2))
```

Then we define a particular object (model) by instantiating this class:

```
my_stoch_vol_model = SimplifiedStochVol(sigma=0.3, rho=0.9)
```

Hopefully, the code above is fairly transparent, but here are some noteworthy details:

- probability distributions are defined through `ProbDist` objects, which are defined in module `distributions`. Most basic probability distributions are defined there; see module `distributions` for more details.
- The class above actually defines a **parametric** class of models; in particular, `self.sigma` and `self.rho` are **attributes** of this class that are set when we define object `my_stoch_vol_model`. Default values for these parameters may be defined in a dictionary called `default_params`. When this dictionary is defined, any undefined parameter will be replaced by its default value:

  ```
  default_stoch_vol_model = SimplifiedStochVol()  # sigma=1., rho=0.8
  ```

- There is no need to define a `__init__()` method, as it is already defined by the parent class. (This parent `__init__()` simply takes care of the default parameters, and may be overridden if needed.)

Now that our state-space model is properly defined, what can we do with it? First, we may simulate states and data from it:

```
x, y = my_stoch_vol_model.simulate(20)
```

This generates two lists of length 20: a list of states, X_0, ..., X_{19} and a list of observations (data-points), Y_0, ..., Y_{19}.

Associated Feynman-Kac models

Now that our state-space model is defined, we obtain the associated Bootstrap Feynman-Kac model as follows:

```
my_fk_model = ssms.Bootstrap(ssm=my_stoch_vol_model, data=y)
```

That's it! You are now able to run a bootstrap filter for this model:

```
my_alg = particles.SMC(fk=my_fk_model, N=200)
my_alg.run()
```

In case you are not clear about what are Feynman-Kac models, and how one may associate a Feynman-Kac model to a given state-space model, see Chapter 5 of the book.

To generate a guided Feynman-Kac model, we must provide proposal kernels (that is, Markov kernels that define how we simulate particles X_t at time t, given an ancestor X_{t-1}):

```
class StochVol_with_prop(StochVol):
    def proposal0(self, data):
        return dists.Normal(scale = self.sigma)
```

(continues on next page)

(continued from previous page)

```
    def proposal(t, xp, data):  # a silly proposal
        return dists.Normal(loc=rho * xp + data[t], scale=self.sigma)

my_second_ssm = StochVol_with_prop(sigma=0.3)
my_better_fk_model = ssms.GuidedPF(ssm=my_second_ssm, data=y)
# then run a SMC as above
```

Voilà! You have now implemented a guided filter.

Of course, the proposal distribution above does not make much sense; we use it to illustrate how proposals may be defined. Note in particular that it depends on `data`, an object that represents the complete dataset. Hence the proposal kernel at time `t` may depend on y_t but also y_{t-1}, or any other datapoint.

For auxiliary particle filters (APF), one must in addition specify auxiliary functions, that is the (log of) functions η_t that modify the resampling probabilities (see Section 10.3.3 in the book):

```
class StochVol_with_prop_and_aux_func(StochVol_with_prop):
    def logeta(self, t, x, data):
        "Log of auxiliary function eta_t at time t"
        return -(x-data[t])**2

my_third_ssm = StochVol_with_prop_and_aux_func()
apf_fk_model = ssms.AuxiliaryPF(ssm=my_third_ssm, data=y)
```

Again, this particular choice does not make much sense, and is just given to show how to define an auxiliary function.

Already implemented state-space models

This module implements a few basic state-space models that are often used as numerical examples:

Class	Comments
`StochVol`	Basic, univariate stochastic volatility model
`StochVolLeverage`	Univariate stochastic volatility model with leverage
`MVStochVol`	Multivariate stochastic volatility model
`BearingsOnly`	Bearings-only tracking
`Gordon_etal`	Popular toy model often used as a benchmark
`DiscreteCox`	A discrete Cox model (Y_t|X_t is Poisson)
`ThetaLogistic`	Theta-logistic model from Population Ecology

Note: Linear Gaussian state-space models are implemented in module *kalman*; similarly hidden Markov models (state-space models with a finite state-space) are implemented in module *hmm*.

Classes

APFMixin()	
AuxiliaryBootstrap([ssm, data])	Base class for auxiliary bootstrap particle filters
AuxiliaryPF([ssm, data])	Auxiliary particle filter for a given state-space model.
BearingsOnly(**kwargs)	Bearings-only tracking SSM.
Bootstrap([ssm, data])	Bootstrap Feynman-Kac formalism of a given state-space model.
DiscreteCox(**kwargs)	A discrete Cox model.
Gordon_etal(**kwargs)	Popular toy example that appeared initially in Gordon et al (1993).
GuidedPF([ssm, data])	Guided filter for a given state-space model.
MVStochVol(**kwargs)	Multivariate stochastic volatility model.
StateSpaceModel(**kwargs)	Base class for state-space models.
StochVol(**kwargs)	Univariate stochastic volatility model.
StochVolLeverage(**kwargs)	Univariate stochastic volatility model with leverage effect.
ThetaLogistic(**kwargs)	Theta-Logistic state-space model (used in Ecology).

particles.utils

Non-numerical utilities (notably for parallel computation).

Overview

This module gathers several non-numerical utilities. The only one of direct interest to the user is the `multiplexer` function, which we now describe briefly.

Say we have some function `f`, which takes only keyword arguments:

```
def f(x=0, y=0, z=0):
    return x + y + z**2
```

We wish to evaluate f repetitively for a range of x, y and/or z values. To do so, we may use function multiplexer as follows:

```
results = multiplexer(f=f, x=3, y=[2, 4, 6], z=[3, 5])
```

which returns a list of 3*2 dictionaries of the form:

```
[ {'x':3, 'y':2, 'z':3, 'out':14},  # 14=f(3, 2, 3)
  {'x':3, 'y':2, 'z':5, 'out':30},
  {'x':3, 'y':4, 'z':3, 'out':16},
   ... ]
```

In other words, `multiplexer` computes the **Cartesian product** of the inputs.

For each argument, you may use a dictionary instead of a list:

```
results = multiplexer(f=f, z={'good': 3, 'bad': 5})
```

In that case, the values of the dictionaries are used in the same way as above, but the output reports the corresponding keys, i.e.:

```
[ {'z': 'good', 'out': 12},  # f(0, 0, 3)
  {'z': 'bad', 'out': 28}    # f(0, 0, 5)
]
```

This is useful when f takes as arguments complex objects that you would like to replace by more legible labels; e.g. option ` model` of class `SMC`.

`multiplexer` also accepts three extra keyword arguments (whose name may not therefore be used as keyword arguments for function f):

- `nprocs` (default=1): if >0, number of CPU cores to use in parallel; if <=0, number of cores *not* to use; in particular, `nprocs=0` means all CPU cores must be used.
- `nruns` (default=1): evaluate f *nruns* time for each combination of arguments; an entry `run` (ranging from 0 to nruns-1) is added to the output dictionaries.
- `seeding` (default: True if `nruns`>1, False otherwise): if True, seeds the pseudo-random generator before each call of function `f` with a different seed; see below.

Warning: Parallel processing relies on library joblib, which generates identical workers, up to the state of the Numpy random generator. If your function involves random numbers: (a) set option `seeding` to True (otherwise, you will get identical results from all your workers); (b) make sure the function f does not rely on scipy frozen distributions, as these distributions also freeze the states. For instance, do not use any frozen distribution when defining your own Feynman-Kac object.

See also:

`multiSMC`

Functions

`add_to_dict`(d, obj[, key])	
`cartesian_args`(args, listargs, dictargs)	Compute a list of inputs and outputs for a function with kw arguments.
`cartesian_lists`(d)	turns a dict of lists into a list of dicts that represents the cartesian product of the initial lists
`distinct_seeds`(k)	generates distinct seeds for random number generation.
`distribute_work`(f, inputs[, outputs, ...])	For each input i (a dict) in list **inputs**, evaluate f(**i) using multiprocessing if nprocs>1
`multiplexer`([f, nruns, nprocs, seeding, ...])	Evaluate a function for different parameters, optionally in parallel.
`timer`(method)	
`worker`(qin, qout, f)	Worker for muliprocessing.

Classes

seeder(func)

particles.variance_estimators

Single-run variance estimators.

Overview

As discussed in Section 19.3 of the book, several recent papers (Chan & Lai, 2013; Lee & Whiteley, 2018; Olsson & Douc, 2019) have proposed variance estimates that may be computed from a **single** run of the algorithm. These estimates rely on genealogy tracking; more precisely they require to track eve variables; i.e. the index of the ancestor at time 0 (or some other time, in Olsson and Douc, 2019) of each particle. See function `var_estimate` for the exact expression of this type of estimate.

Variance estimators (Chan & Lai, 2013; Lee & Whiteley, 2018)

These estimates may be *collected* (see module *collectors*) as follows:

```
import particles
from particles import variance_estimators as var  # this module

# Define a FeynmanKac object, etc.
#  ...
phi = lambda x: x**2  # for instance
my_alg = particles.SMC(fk=my_fk_model, N=100,
                       collect=[var.Var(phi=phi), var.Var_logLt()])
```

The first collector will compute at each time t an estimate of the variance of $\sum_{n=1}^N W_t^n \varphi(X_t^n)$ (which is itself a particle estimate of expectation $\mathbb{Q}_t(\varphi)$). If argument `phi` is not provided, the function $\varphi(x) = x$ will be used.

The second collector will compute an estimate of the variance of the log normalising constant, i.e. $\log L_t$.

Note: The estimators found in Chan & Lai (2013) and Lee & Whiteley (2018) differ only by a factor $(N/(N-1))^t$; the collectors above implement the former version, without the factor.

Lag-based variance estimators (Olsson and Douc, 2019)

The above estimators suffer from the well known problem of **particle degeneracy**; as soon as the number of distinct ancestors falls to one, these variance estimates equal zero. Olsson and Douc (2019) proposed a variant based on a fixed-lag approximation. To compute it, you need to activate the tracking of a rolling-window history, as for fixed-lag smoothing (see below):

```
my_alg = particles.SMC(fk=my_fk_model, N=100,
                       collect=[var.Lag_based_var(phi=phi)],
                       store_history=10)
```

which is going to compute the same type of estimates, but using as eve variables (called Enoch variables in Olsson and Douc) the index of the ancestor of each particle X_t^n as time $t - l$, where l is the lag. This collector actually computes and stores simultaneously the estimates that correspond to lags 0, 1, ..., k (where k is the size of the rolling window history). This makes it easier to assess the impact of the lag on the estimates. Thus:

```
print(my_alg.lag_based_var[-1])  # prints a list of length 10
```

Numerical experiments

See here for a jupyter notebook that illustrates these variance estimates in a simple example.

References

- Chan, H.P. and Lai, T.L. (2013). A general theory of particle filters in hidden Markov models and some applications. Ann. Statist. 41, pp. 2877–2904.
- Lee, A and Whiteley, N (2018). Variance estimation in the particle filter. Biometrika 3, pp. 609-625.
- Olsson, J. and Douc, R. (2019). Numerically stable online estimation of variance in particle filters. Bernoulli 25.2, pp. 1504-1535.

Functions

var_estimate(W, phi_x, B)	Variance estimate based on genealogy tracking.

Classes

Lag_based_var(**kwargs)	Computes and collects Olsson and Douc (2019) variance estimates, which are based on a fixed-lag approximation.
Var(**kwargs)	Computes and collects variance estimates for a given test function phi.
VarColMixin()	
Var_logLt(**kwargs)	Computes and collects estimates of the variance of the log normalising constant estimator.

particles.variance_mcmc

MCMC variance estimators.

Author: Hai-Dang Dau

Various estimators of the asymptotic variance of a MCMC kernel, based on M chains of length P:

- initial sequence estimator of Geyer
- Tukey-Hanning

This may be used to estimate the (asymptotic) variance of estimates generated by waste-free SMC.

Functions

MCMC_Tukey_Hanning(X[, method, bias, ...])	MCMC Variance estimator using spectral variance method with Tukey_Hanning window.
MCMC_init_seq(X[, method, bias])	initial sequence estimator, see Practical MCMC (Geyer 1992) Let c_0, c_1, .
MCMC_variance(X, method)	**param X** a (P, M) numpy array which contains M MCMC chains of lengths P
MCMC_variance_naive(X)	
MCMC_variance_weighted(X, W, method)	Like MCMC_variance, but each column of X has a weight W that sums to 1.
autocovariance(X, order[, mu, bias])	
autocovariance_fft_multiple(X[, mu, bias])	**param X** numpy array of shape (P,M), which corresponds typically to M MCMC runs of length P each.
autocovariance_fft_single(x[, mu, bias])	**param x** numpy array of shape (n,)
default_collector(ls)	

Classes

AutoCovarianceCalculator(X[, method, bias])	An artificial device to efficiently calculate the auto-covariances based on (possibly) multiple runs of an MCMC method.

PYTHON MODULE INDEX

p

particles, 59
particles.binary_smc, 60
particles.collectors, 61
particles.core, 66
particles.datasets, 68
particles.hilbert, 73
particles.hmm, 74
particles.kalman, 76
particles.mcmc, 79
particles.nested, 80
particles.resampling, 83
particles.rqmc, 86
particles.smc_samplers, 86
particles.smoothing, 90
particles.state_space_models, 93
particles.utils, 96
particles.variance_estimators, 98
particles.variance_mcmc, 99

INDEX

M

module
    particles, 59
    particles.binary_smc, 60
    particles.collectors, 61
    particles.core, 66
    particles.datasets, 68
    particles.distributions, 69
    particles.hilbert, 73
    particles.hmm, 74
    particles.kalman, 76
    particles.mcmc, 79
    particles.nested, 80
    particles.resampling, 83
    particles.rqmc, 86
    particles.smc_samplers, 86
    particles.smoothing, 90
    particles.state_space_models, 93
    particles.utils, 96
    particles.variance_estimators, 98
    particles.variance_mcmc, 99

P

particles
    module, 59
particles.binary_smc
    module, 60
particles.collectors
    module, 61
particles.core
    module, 66
particles.datasets
    module, 68
particles.distributions
    module, 69
particles.hilbert
    module, 73
particles.hmm
    module, 74
particles.kalman
    module, 76
particles.mcmc
    module, 79
particles.nested
    module, 80
particles.resampling
    module, 83
particles.rqmc
    module, 86
particles.smc_samplers
    module, 86
particles.smoothing
    module, 90
particles.state_space_models
    module, 93
particles.utils
    module, 96
particles.variance_estimators
    module, 98
particles.variance_mcmc
    module, 99