StochRare

Mar 01, 2020

Contents:

CHAPTER 1

Installation

1.1 Dependencies

- Python 3
- numpy
- scipy
- matplotlib
- Optional: jupyter

stochrare only requires a running Python 3 (we recommend python 3.7) install and standard scientific packages. Tutorial notebooks require Jupyter.

1.2 Basic Install

First clone the repository:

```
git clone https://framagit.org/cherbert/stochrare
```

Then go to the directory and run the install script:

```
cd stochrare
python setup.py install
```

CHAPTER 2

Tutorials

To illustrate the functionnalities and use cases of stochrare, we provide Jupyter notebooks, which can either be viewed on-line here or run interactively.

2.1 1D Diffusion

This tutorial illustrates some basic features of the `stochrare` package with simple 1D diffusion processes.

First, let us import standard packages: `numpy` and `matplotlib`.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

2.1.1 Monte-Carlo simulation of 1D diffusions

One of the very common tasks `stochrare` can be used for is sampling trajectories of a stochastic process, or estimating directly some statistical properties of the process. We have tried to design the package so that such tasks can be easily fulfilled with an intuitive interface. The generic class for 1D diffusion processes is `stochrare.dynamics.diffusion1d.DiffusionProcess1D`. As we shall see later in this tutorial, any 1D diffusion process can be represented as members of this class, simply by providing the drift and diffusion functions to the constructor. But the package also ships with predefined standard processes, like the Wiener process. Let us import this process:

[2]:
```
from stochrare.dynamics.diffusion1d import Wiener1D
```

Sample paths

First, we seed the random number generator with a fixed value, so that the results are always the same when we run the whole notebook. Of course, if you run a given cell multiple times, you will get different realizations of the stochastic processes.

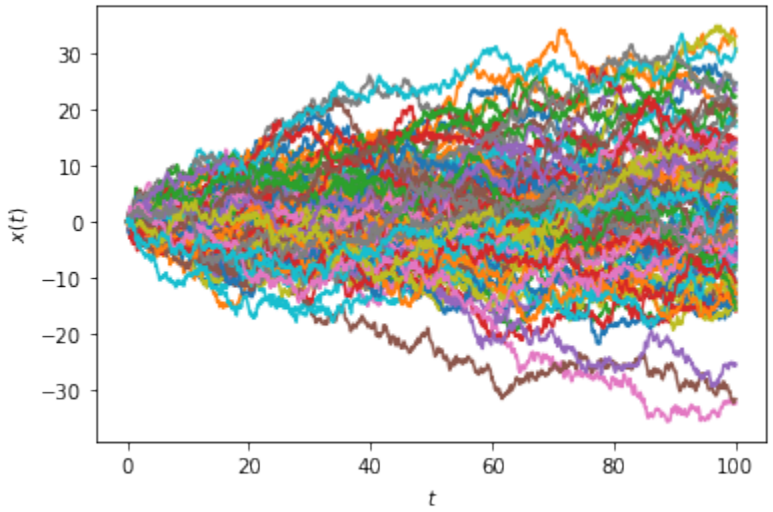
[3]:
```
np.random.seed(seed=100)
```

The basic tool to generate sample paths is the `trajectory` method. It integrates numerically stochastic differential equations of the form:

$$dX_t = F(X_t, t) + \sigma(X_t, t) dW_t.$$

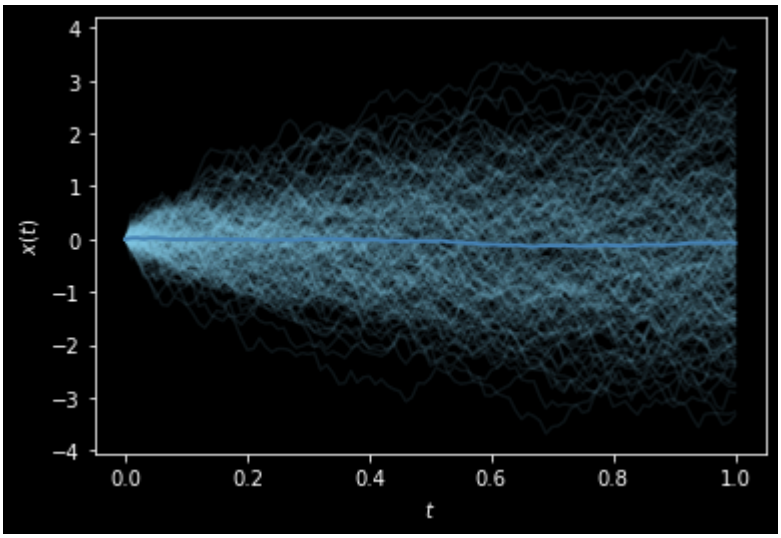
Let us start by plotting sample paths of the Wiener process W (also known as *Brownian motion*) with a one-liner:

[4]:
```
Wiener1D().trajectoryplot(*[Wiener1D().trajectory(0.,0.,T=100) for _ in range(100)]);
```



Or, in a different style:

[5]:
```
from stochrare.io.plot import trajectory_plot1d
def ensemble_plot1d(*args, **kwargs):
    fig, ax = trajectory_plot1d(*((t, x, {'color': 'skyblue', 'alpha': 0.1}) for t, x␣
↪in args), **kwargs)
    ax.plot(*np.array(args).mean(axis=0), color='steelblue', lw=2)
    return fig, ax
with plt.style.context(('dark_background')):
    ensemble_plot1d(*[Wiener1D().trajectory(0., 0., T=1, dt=0.01) for _ in␣
↪range(200)]);
```



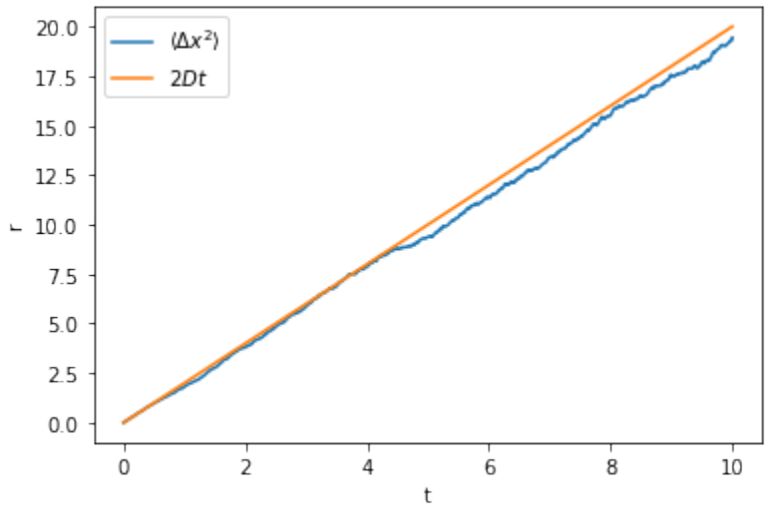
We have used two convenience methods that we provide for making quick plots: `trajectory_plot1d`, which is a function from the `stochrare.io.plot` submodule, and `trajectoryplot`, a method of `DiffusionProcess1D` objects which just provides an interface to `trajectory_plot1d` for quick access, and which can be overriden by subclasses to systematically include specific details in trajectory plots.

Observables and PDFs

We now estimate some statistical properties of stochastic processes.

Let us first check that for the Brownian motion the mean square displacement increases linearly with time:

[6]:
```
ensemble = np.array([Wiener1D().trajectory(0., 0., T=10, dt=0.01) for _ in␣
↪range(1000)])
time = np.average(ensemble[:, 0, :], axis=0)
ax = plt.axes(xlabel='t', ylabel='r')
ax.plot(time, np.average(ensemble[:, 1, :]**2, axis=0), label=r'$\langle \Delta x^2␣
↪\rangle$')
ax.plot(time, 2*time, label=r'$2Dt$')
ax.legend()
plt.show()
```



Now we estimate the probability for the stochastic process to take value x at time t, knowing the initial condition $X_{t_0} = x_0$, i.e. the transition probability $p(x, t|x_0, t_0)$. For this, the `empirical_vector` method should be used: it simulates on the fly an ensemble of sample paths and returns the histogram of the values at each desired time.

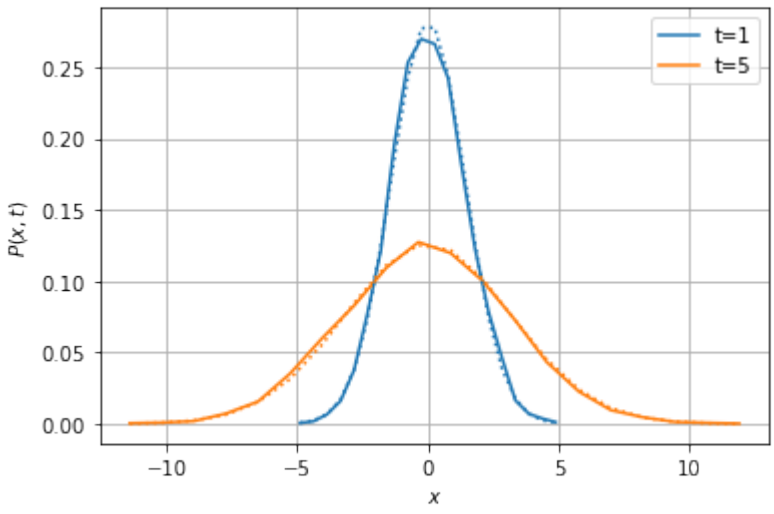
Again, we work with the Wiener process, for which an analytical solution is known:

$$p(x, t|0, 0) = \frac{1}{\sqrt{4\pi D t}} e^{-x^2/4Dt}.$$

$$This solution is hard-coded in the "Wiener1D" class, in the "_fpthsol" method.$$

Both solutions are represented using the `pdf_plot1d` function, another tool for quick plots dedicated to probability distributions. In the figure below, the Monte-Carlo estimate is the solid line and the theoretical result is the dotted line.

[7]:
```
from stochrare.io.plot import pdf_plot1d
pdf = list(Wiener1D().empirical_vector(0, 0, 10000, 1, 5, bins=20))
fig, ax, lines = pdf_plot1d(*((0.5*(xx[1:]+xx[:-1]), pp, {'label': f't={t}'}) for t,␣
↪pp, xx in pdf));
pdf_plot1d(*((0.5*(xx[1:]+xx[:-1]), Wiener1D()._fpthsol(0.5*(xx[1:]+xx[:-1]), t),
              {'ls': 'dotted', 'color': l.get_color()}) for l, (t, _, xx) in␣
↪zip(lines, pdf)),
           fig=fig, ax=ax);
```



Numerical convergence

The sample paths computed with the `trajectory` method are discrete approximations of the sample paths of the stochastic differential equation, using the Euler-Maruyama method, which consists in computing a sequence of random numbers X_n defined by:

$where : math : `t_n`isthesampletime, : math : `\Delta t = t_{n+1} - t_n`thetimestepofthemethodand : math : `\Delta W_n = W_{t_{n+1}} - W_{t_n}`a$

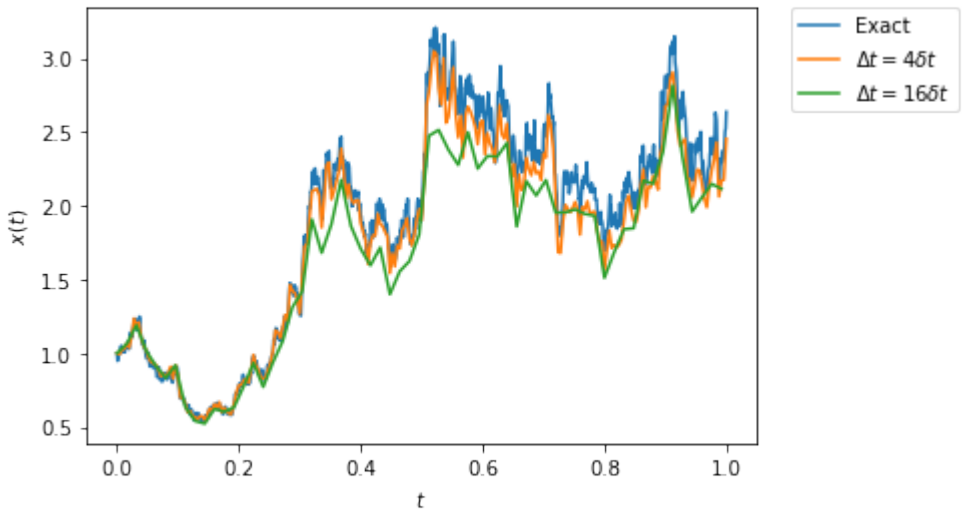
Let us now illustrate the numerical convergence of the Euler-Maruyama method. For this, we precompute the Brownian path with respect to which we integrate the SDE, and we vary the time step used for the Euler-Maruyama method.

We do this for a stochastic process which can be analytically solved:

$andcomparethenumericalapproximationtotheanalyticalsolution, : math : `X_t = X_0 e^{3t/2+W_t}`.Thisstochasticprocessiseasily$

[8]:
```
from stochrare.dynamics.diffusion1d import DiffusionProcess1D
model = DiffusionProcess1D(lambda x, t: 2*x, lambda x, t: x)
```

[9]:
```
dt_brownian = 0.001
brownian_path = Wiener1D(D=0.5, deterministic=True).trajectory(0., 0., T=1, dt=dt_
↪brownian)
model.trajectoryplot((brownian_path[0], np.exp(1.5*brownian_path[0]+brownian_
↪path[1])),
                     model.trajectory(1., 0., T=1, dt=4*dt_brownian, brownian_
↪path=brownian_path),
                     model.trajectory(1., 0., T=1, dt=16*dt_brownian, brownian_
↪path=brownian_path),
                    labels=('Exact', r'$\Delta t= 4 \delta t$', r'$\Delta t = 16␣
↪\delta t$'));
```



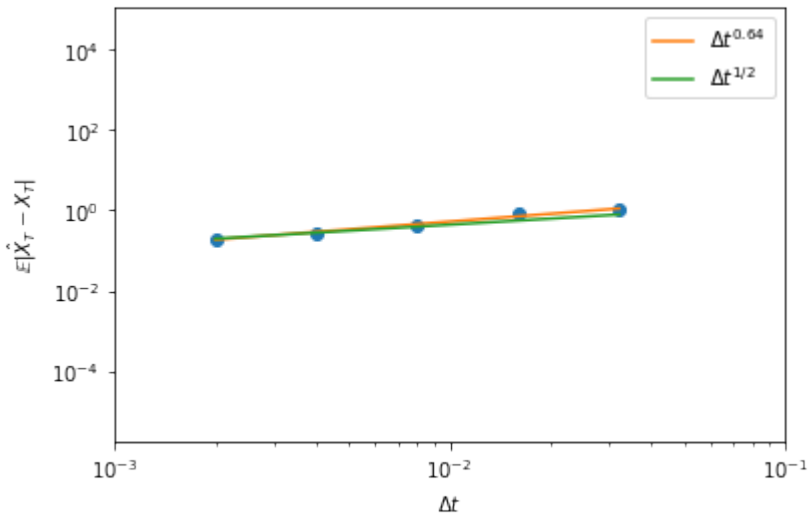
Slightly more precisely, let us try to illustrate that the Euler-Maruyama has strong order of convergence $1/2$:

$Below, weplottheerroronthefinalpointasafunctionofthetimestepusedintheEuler - Maruyamamethod, andwecomparetoal$

[10]:
```
import scipy.stats

ensemble_size = 1000
dt_brownian = 0.002
dtarr = np.array((1, 2, 4, 8, 16))*dt_brownian
model = DiffusionProcess1D(lambda x, t : 2*x, lambda x, t: x)
trajs = np.array([[np.abs(model.trajectory(1., 0., T=1, dt=dt,
                                           brownian_path=brownian_path)[1][-1]-np.
↪exp(1.5*brownian_path[0][-1]+brownian_path[1][-1]))
                   for dt in dtarr]
                  for brownian_path in (Wiener1D(D=0.5).trajectory(0., 0., T=1, dt=dt_
↪brownian) for _ in range(ensemble_size))])
error = trajs.mean(axis=0)

ax = plt.axes(xlim=(0.001, 0.1), xlabel=r'$\Delta t$', ylabel=r'$\mathbb{E}|\hat{X}_T-
↪X_T|$', xscale='log', yscale='log')
ax.scatter(dtarr, error)
slope, intercept, _, _, _ = scipy.stats.linregress(np.log10(dtarr), np.log10(error))
ax.plot(dtarr, 10**intercept*dtarr**slope, label=r'$\Delta t^{'+format(slope, '.2f')+
↪'}$', color='C1')
ax.plot(dtarr, error[0]*(dtarr/dtarr[0])**0.5, label=r'$\Delta t^{1/2}$', color='C2')
ax.legend();
```



2.1.2 Numerical solution of the Fokker-Planck equation

Above we have considered stochastic processes fron the standpoint of stochastic differential equations. We now turn to an alternative point of view, the probability distribution. Markov processes are fully determined by the transition probabilities $p(x', t'|x, t)$, which satisfy the Fokker-Planck equations:

$$Some properties of stochastic processes are much more conveniently addressed in the Fokker - Planck framework : rather than$$

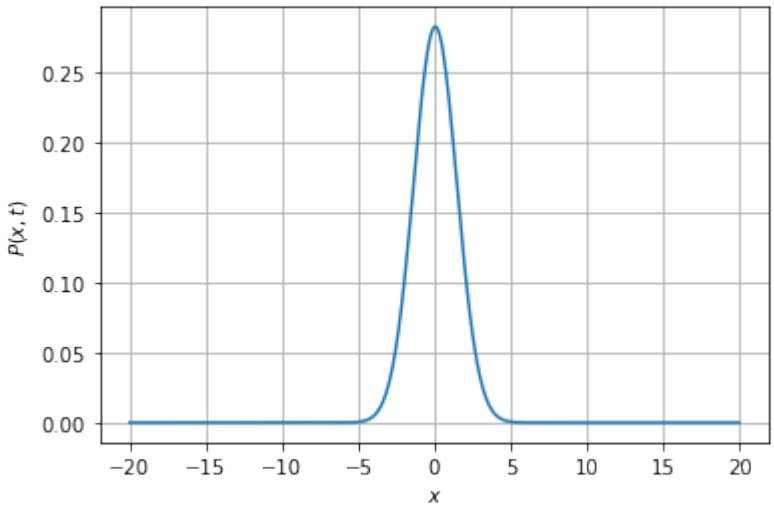
The package includes a basic finite-difference solver for the 1D Fokker-Planck equation. We illustrate its use below.

Let us first import the submodule:

[11]:
```
import stochrare.fokkerplanck as fp
```

All we need to do is create an object representing the Fokker-Planck equation (an instance of the `FokkerPlanck1D` class) and then use its `fpintegrate` method:

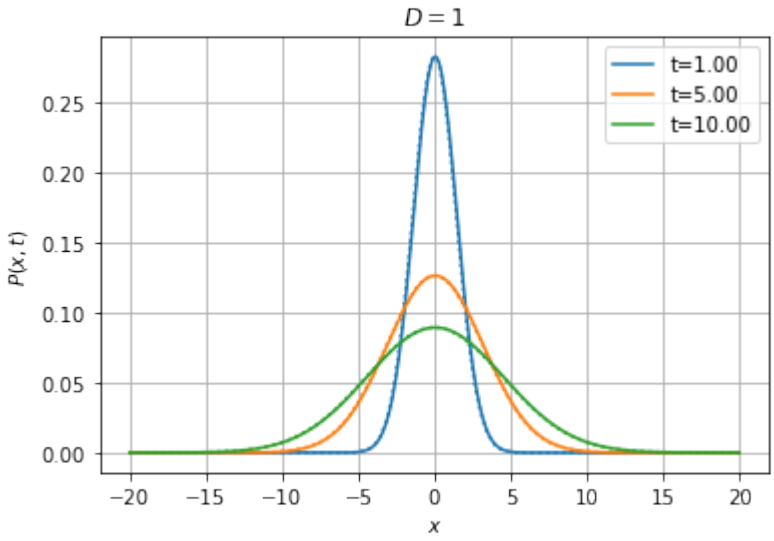
[12]:
```
fpe = fp.FokkerPlanck1D(lambda x, t: 0, 1)
t, X, P = fpe.fpintegrate(0, 1, dt=0.001, npts=400, bounds=(-20., 20.), P0='dirac',␣
↪bc=('absorbing', 'absorbing'))
pdf_plot1d((np.array(X), np.array(P)), legend=False);
```



Note that we have provided as arguments to `fpintegrate` the domain on which to solve the equation (`bounds`), the space resolution (`npts`), the time step (`dt`), as well as the boundary conditions (`bc`; here they are both absorbing boundary conditions) and an initial distribution (`P0`).

`FokkerPlanck1D` objects offer another method to compute easily the probability distribution at different times: `fpintegrate_generator` is a generator yielding the pdf at times given as arguments (see API documentation for more information). It relies on `fpintegrate` under the hood. For convenience, `ConstantDiffusionProcess1D` objects offer a `pdfplot` method which wraps the `FokkerPlanck1D.fpintegrate_generator` method. As an example, the solution of the heat equation at different times can be obtained as a one-liner:

[13]:
```
Wiener1D().pdfplot(1, 5, 10, dt=0.001, npts=400, bounds=(-20.0, 20.0),
                   t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), th=True);
```

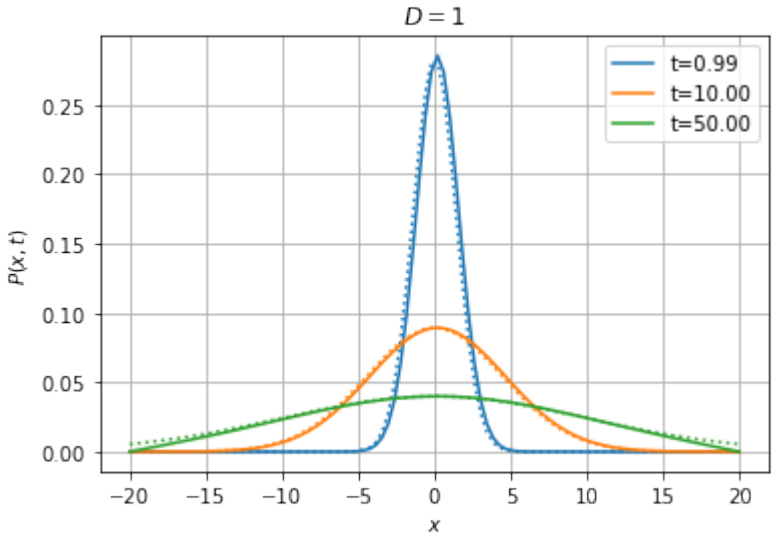


The theoretical solution is shown as a dotted line with the same color.

Effect of Boundary Conditions

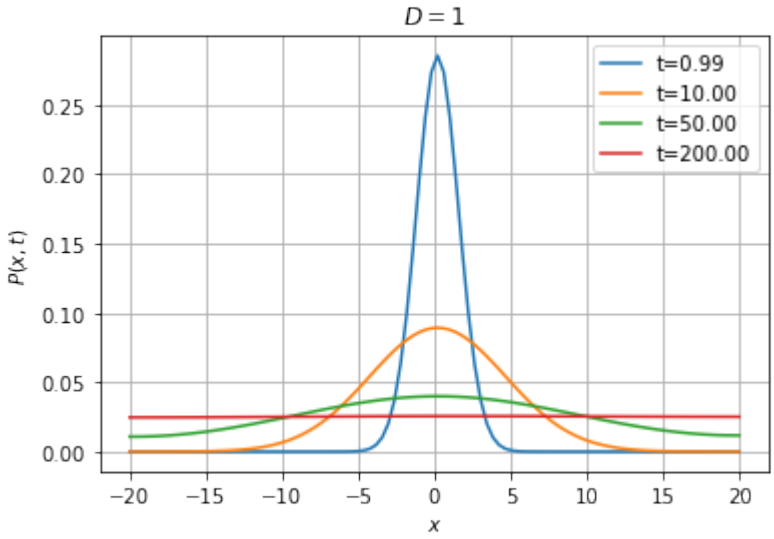
Above we have compared the solution of the Fokker-Planck equation with absorbing boundary conditions with the theoretical solution on an infinite domain. Discrepancies should start appearing when we reach the boundaries.

[14]:
```
Wiener1D().pdfplot(1, 10, 50, dt=0.01, npts=100, bounds=(-20.0, 20.0),
                   t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), th=True);
```



Below, we solve the equation with reflecting boundary conditions on both sides. In that case, we expect the system to reach a stationary state where the probability distribution is uniform on the interval.

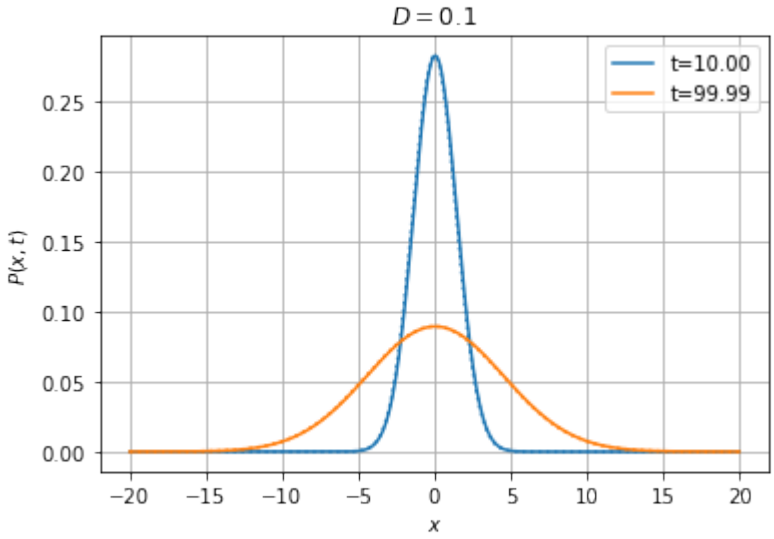
[15]:
```
Wiener1D().pdfplot(1, 10, 50, 200, dt=0.01, npts=100, bounds=(-20.0, 20.0),
                   t0=0.0, P0='dirac', bc=('reflecting', 'reflecting'));
```



Comments on numerical aspects

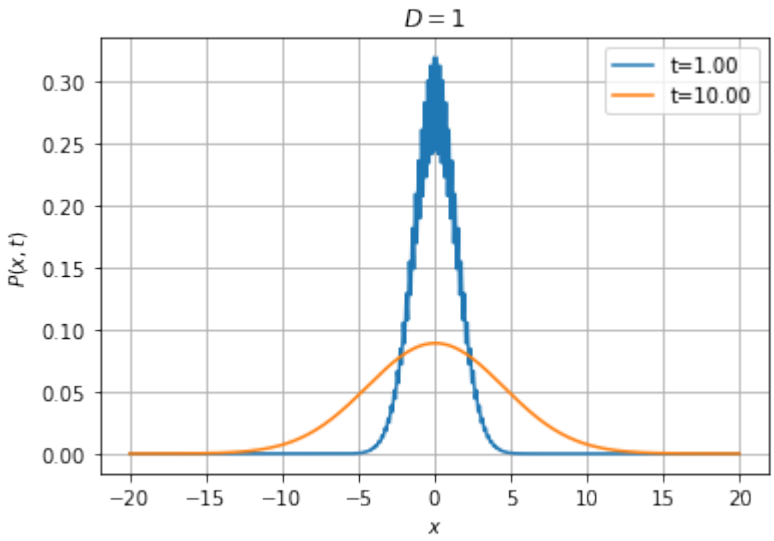
With the default Euler time-stepping scheme, the time step dt should be much smaller than dx^2/D, with dx the spatial resolution and D the diffusivity. For instance, if we decrease the diffusivity by a factor 10, we can afford to multiply the time step by the same factor (of course the probability distribution will spread 10 times slower).

[16]:
```
Wiener1D(D=0.1).pdfplot(10, 100, dt=0.01, npts=400, bounds=(-20.0, 20.0),
                        t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), th=True);
```

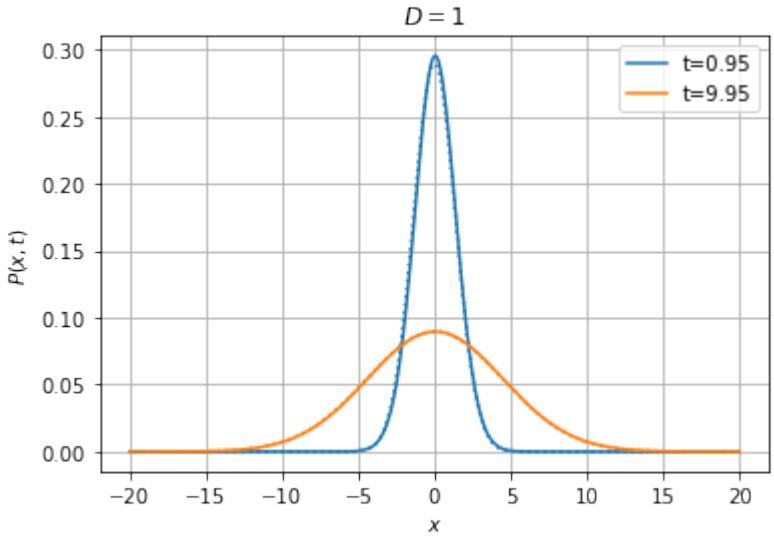


With the *implicit* and *Crank-Nicolson* schemes, we can afford using much larger timesteps than with the explicit method, as illustrated below:

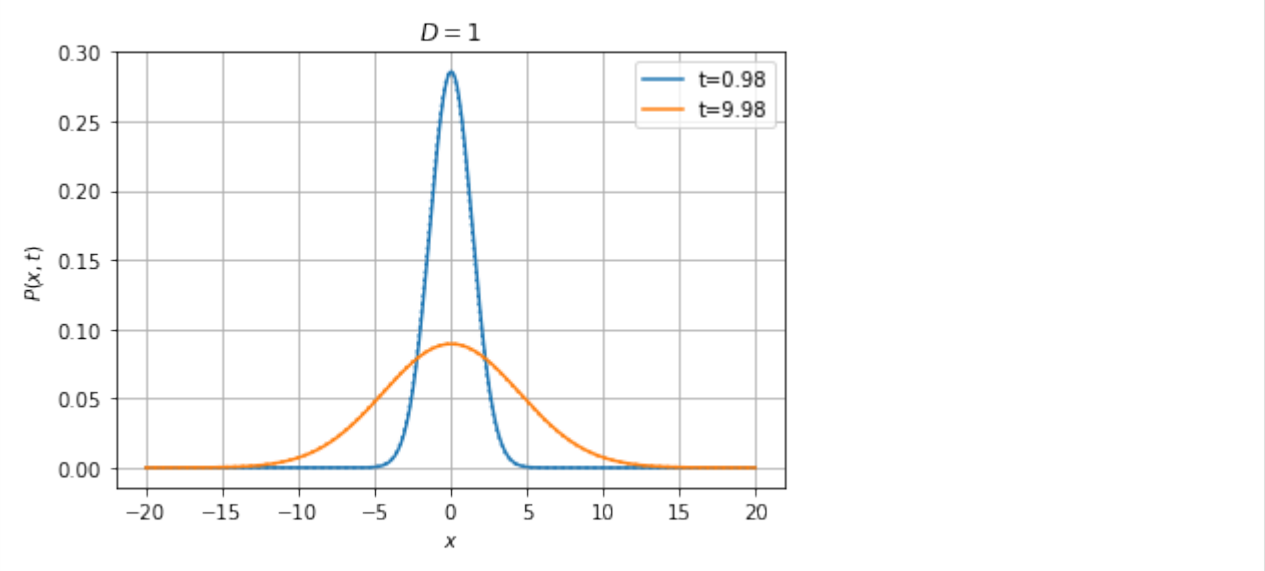
[17]:
```
Wiener1D(D=1).pdfplot(1, 10, dt=0.005, npts=400, bounds=(-20.0, 20.0),
                      t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), method=
↪'explicit');
```



[18]:
```
Wiener1D(D=1).pdfplot(1, 10, dt=0.05, npts=400, bounds=(-20.0, 20.0),
                      t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), method=
↪'implicit', th=True);
```



[19]:
```
Wiener1D(D=1).pdfplot(1, 10, dt=0.025, npts=400, bounds=(-20.0, 20.0),
                      t0=0.0, P0='dirac', bc=('absorbing', 'absorbing'), method='cn',␣
↪th=True);
```



2.2 More simulations

In the previous tutorial we only simulated simple 1D diffusion processes. Here, we show more examples of simulations with different dynamics, illustrating in particular the generic class `stochrare.dynamics.diffusion.DiffusionProcess`.

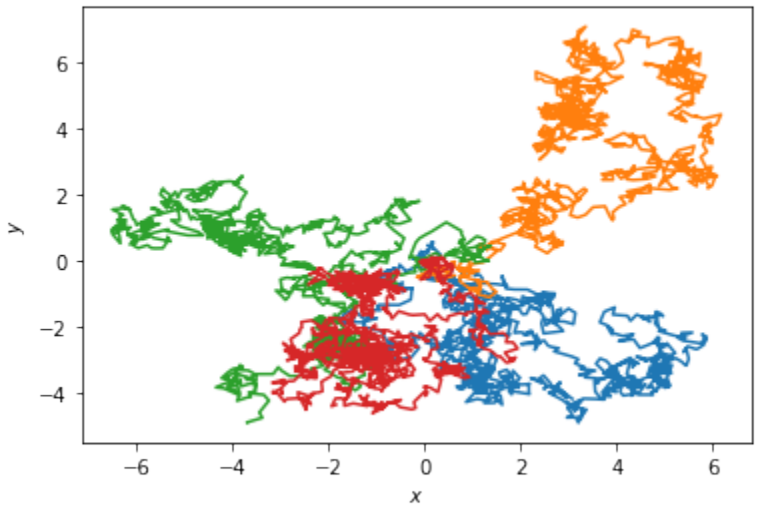
2.2.1 2D Diffusions: gradient dynamics

Let us start with basic diffusion processes in 2D: the Wiener process and the Ornstein-Uhlenbeck process.

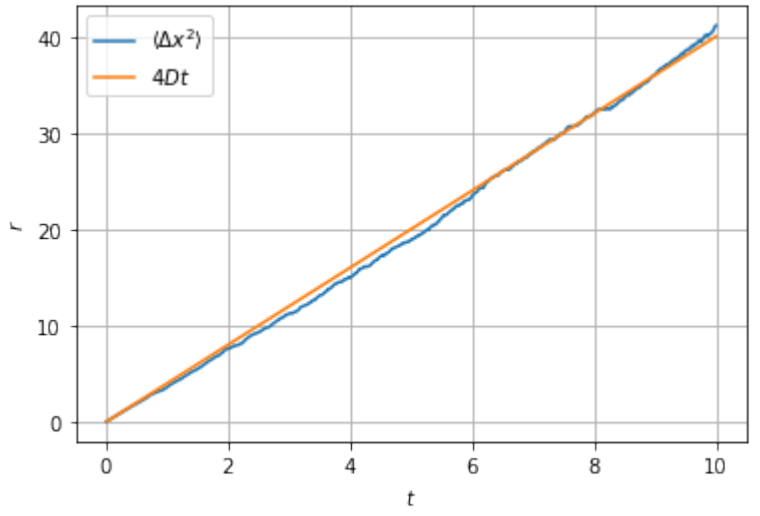
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from stochrare.dynamics.diffusion import Wiener, OrnsteinUhlenbeck
```

[2]:
```
np.random.seed(seed=100)
```

[3]:
```
ax = plt.axes(xlabel=r'$x$', ylabel=r'$y$')
for _ in range(4):
    t, x = Wiener(2).trajectory(np.array([0., 0.]), 0., dt=0.01)
    ax.plot(x[:, 0], x[:, 1])
```



[4]:
```
time, dist2 = zip(*[(t,r) for t,r in Wiener(2).sample_mean(np.array([0.,0.]), 0.,␣
↪1000, 1000, dt=0.01,
                                                          observable=lambda x, t:␣
↪x[0]**2+x[1]**2)])
ax = plt.axes(xlabel=r'$t$', ylabel=r'$r$')
ax.grid()
ax.plot(time, dist2, label=r'$\langle \Delta x^2 \rangle$')
ax.plot(time, 4*np.array(time), label=r'$4Dt$')
ax.legend();
```

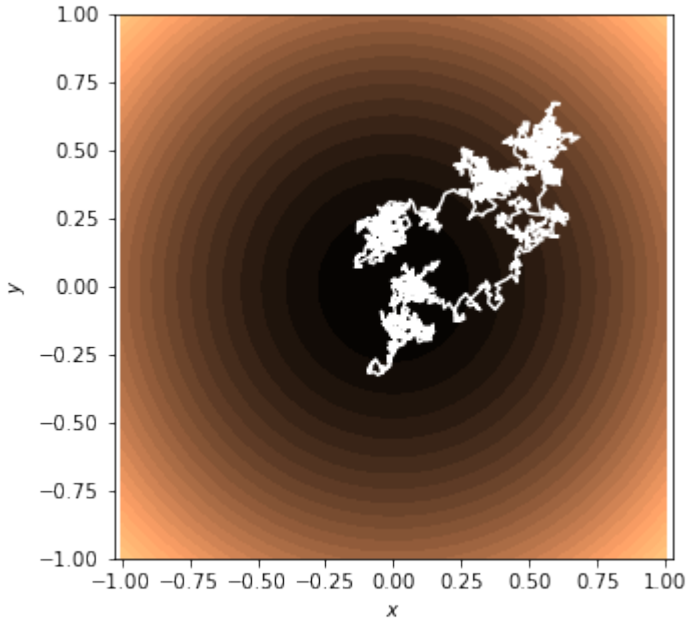


[5]:
```
model = OrnsteinUhlenbeck(0,1, 0.1, 2)
xvec = np.linspace(-1., 1.)
yvec = np.linspace(-1., 1.)
potential = np.array([model.potential(np.array([x, y])) for x in xvec for y in yvec]).
↪reshape(50, 50)
fig = plt.figure(figsize=(5, 5))
ax = plt.axes(xlabel=r'$x$', ylabel=r'$y$')
ax.axis('equal')
ax.contourf(xvec, yvec, potential, 30, cmap='copper')
```

(continues on next page)

(continued from previous page)

```
t, x = model.trajectory(np.array([0., 0.]), 0., T=2, dt=0.001)
ax.plot(x[:, 0], x[:, 1], color='white');
```



2.2.2 Langevin equation

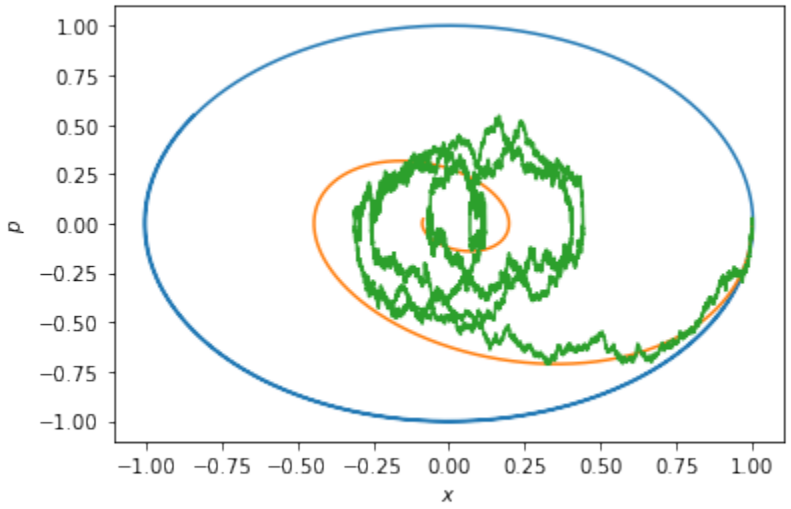
Now we simulate the Langevin dynamics:

$$\dot{x} = p/m, \quad \dot{p} = -V'(x) - \gamma p + \eta(t),$$
$$with a harmonic potential : math : `V(x) = x^2/2` and : math : `\mathbb{E}[\eta(t)\eta(t')] = D\delta(t-t')`.$$

[6]:
```
from stochrare.dynamics.diffusion import DiffusionProcess
gamma = 0
def langevin(gamma, D):
    return DiffusionProcess(lambda X, t: np.array([X[1], -X[0]-gamma*X[1]]),
                            lambda X, t: np.array([[0., 0.], [0., D]]))
```

Without friction and noise ($\gamma = D = 0$), the system is Hamiltonian and $H = x^2 + p^2$ is conserved. Adding some friction $\gamma > 0$, the system relaxes towards equilibrium in a spiraling motion, because of inertia. With noise, we inject energy randomly in the system, and the stationary distribution spreads over a region of phase space centered on the origin.

[7]:
```
xvec = np.linspace(-1., 1.)
pvec = np.linspace(-1., 1.)
ax = plt.axes(xlabel=r'$x$', ylabel=r'$p$')
t, x = langevin(0, 0).trajectory(np.array([1., 0.]), 0., T=10, dt=0.001)
ax.plot(x[:,0], x[:,1]);
t, x = langevin(0.5, 0).trajectory(np.array([1., 0.]), 0., T=10, dt=0.001)
ax.plot(x[:,0], x[:,1]);
t, x = langevin(0.5, 0.2).trajectory(np.array([1., 0.]), 0., T=20, dt=0.001)
ax.plot(x[:,0], x[:,1]);
```



[ ]:

2.3 First Passage Times: the Kramers problem

This tutorial illustrates the use of the `stochrare` package for first-passage time computations in the context of the Kramers problem [1] (diffusion in a double well potential). Here, we simply compute numerically the rate of transition between the two attractors.

2.3.1 References

- [1] Kramers, Physica, 7, 284-304 (1940)
- [2] Gardiner, Handbook of Stochastic Methods, Springer, chap. 9 and §5.5.
- [3] Risken, The Fokker-Planck equation, Springer, §5.10
- [4] Caroli, Caroli and Roulet, J. Stat. Phys., 21, 415-437 (1979)
- [5] Caroli, Caroli and Roulet, J. Stat. Phys., 26, 83-111 (1981)
- [6] Hanggi, Talkner, Borkovec, Rev. Mod. Phys., 62, 251-342 (1990)

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import stochrare as sr
```

Let us define the `DoubleWell` class, corresponding to a simple bistable system, by subclassing the generic 1D diffusion class `ConstantDiffusionProcess1D`:

[2]:
```
class DoubleWell(sr.dynamics.diffusion1d.ConstantDiffusionProcess1D):
    """
    Double well potential model.
    """
    default_dt = 0.01
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, Damp, **kwargs):
        sr.dynamics.diffusion1d.ConstantDiffusionProcess1D.__init__(self, lambda x, t:
↪ -x*(x**2-1), Damp, **kwargs)

    def potential(self, X, t):
        """
        Return the value of the potential at the input points.
        """
        Y = X**2
        return Y*(Y-2.0)/4.0
```

The dynamics is given by the stochastic differential equation:

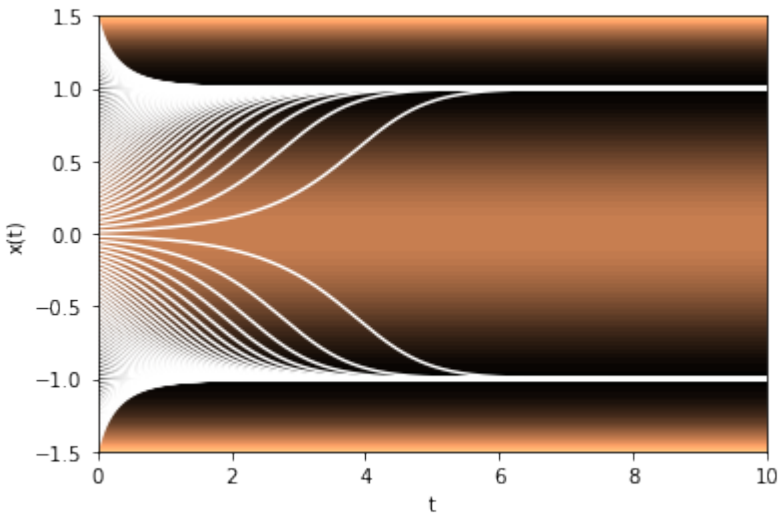
$$dX_t = -V'(X_t)dt + \sqrt{2D}dW_t,$$
$$with : math : `V(x) = x^4/4 - x^2/2`.$$

2.3.2 Qualitative understanding

Deterministic dynamics

Let us first look at the dynamics for the deterministic system: we plot below the phase portrait of the system, superimposed upon the potential.

[3]:
```
ax = plt.axes(xlabel='t', ylabel='x(t)')
X = np.linspace(-1.5, 1.5, num=100)
ax.contourf(np.linspace(0, 10), X, np.tile(DoubleWell(0).potential(X, 0), (50, 1)).T,␣
↪50, cmap='copper')
for x0 in X:
    ax.plot(*DoubleWell(0).trajectory(x0, 0, T=10), color='white')
```

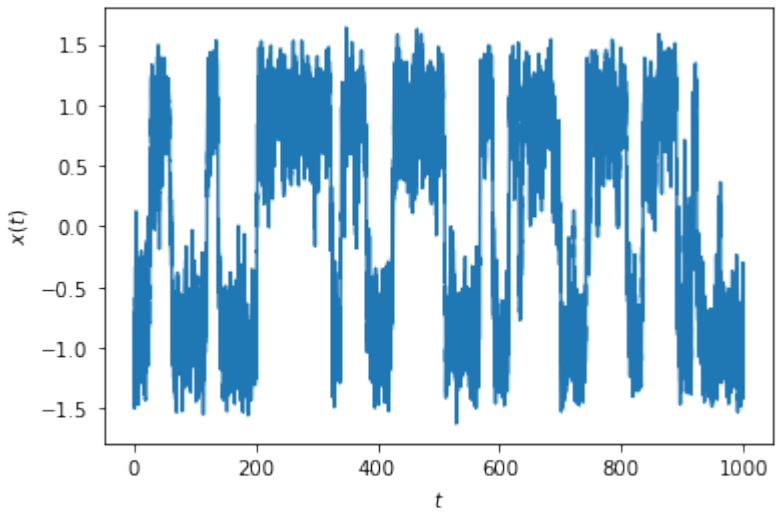


$x = -1$ and $x = 1$ are stable fixed points with basins of attraction $]-\infty, 0[$ and $]0, +\infty[$, respectively, separated by an unstable fixed point at $x = 0$. Typical relaxation time is of order one but diverges as x_0 goes to 0.

Noise-induced transitions between the two attractors

When the diffusion coefficient does not vanish, one should expect transitions between the two attractors. Let us illustrate this below with a moderate value of the noise.

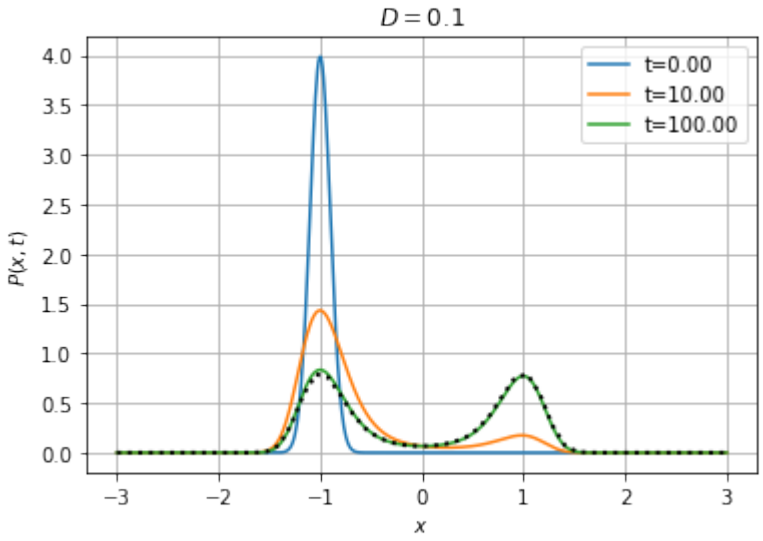
[4]:
```
DoubleWell.trajectoryplot(DoubleWell(0.1, deterministic=True).trajectory(-1, 0,␣
↪T=1000, dt=0.01));
```



The stationary distribution of the system should therefore be bimodal. It is simply given by $e^{-V(x)/D}$, up to a normalization factor. Let us compute this stationary distribution numerically by solving the Fokker-Planck equation with absorbing boundary conditions at a sufficient distance.

[5]:
```
_, ax = DoubleWell(0.1).pdfplot(0.0, 10.0, 100.0, dt=0.0005, npts=600, bounds=(-3.0,␣
↪3.0),
                                P0=sr.fokkerplanck.FokkerPlanck1D.gaussian1d(-1, 0.1,␣
↪np.linspace(-3, 3, 600)));
x = np.linspace(-3, 3)
V = DoubleWell(0.1).potential(x, 0)
ax.plot(x, np.exp(-V/0.1)/np.trapz(np.exp(-V/0.1), x=x), color='black', ls='dotted',␣
↪lw=2);
```

```
/Users/corentin/codes/stochrare/stochrare/fokkerplanck.py:151: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  if P0 == 'gauss':
/Users/corentin/codes/stochrare/stochrare/fokkerplanck.py:153: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  if P0 == 'dirac':
/Users/corentin/codes/stochrare/stochrare/fokkerplanck.py:157: FutureWarning:␣
↪elementwise comparison failed; returning scalar instead, but in the future will␣
↪perform elementwise comparison
  if P0 == 'uniform':
```



Particles initially concentrated in the left well *tunnel* through the potential barrier, and ultimately the relative probability of the two attractors is the same (because the quasi-potential is symmetric).

To study the statistical properties of the transitions between the two attractors, we introduce the *first-passage time*:

$$\tau_M = \inf\{$$

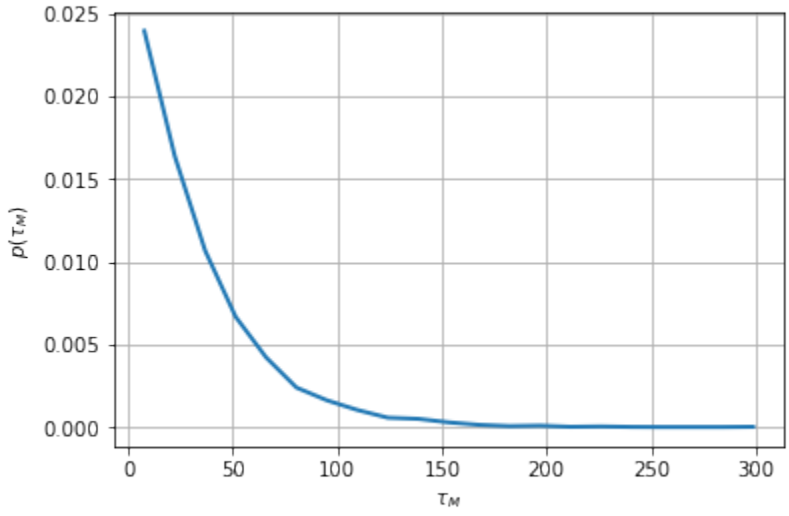
$Itisarandomvariableentirelydeterminedbythestochasticprocess : math : `X`.In "stochrare", itcanberepresentedbythe "Firs$

[6]:
```
tau = sr.firstpassage.FirstPassageProcess(DoubleWell(0.1))
```

Methods of the `FirstPassageProcess` class allow for sampling the random variable, estimating its mean, moments and PDF, using direct simulation or using the Fokker-Planck equation.

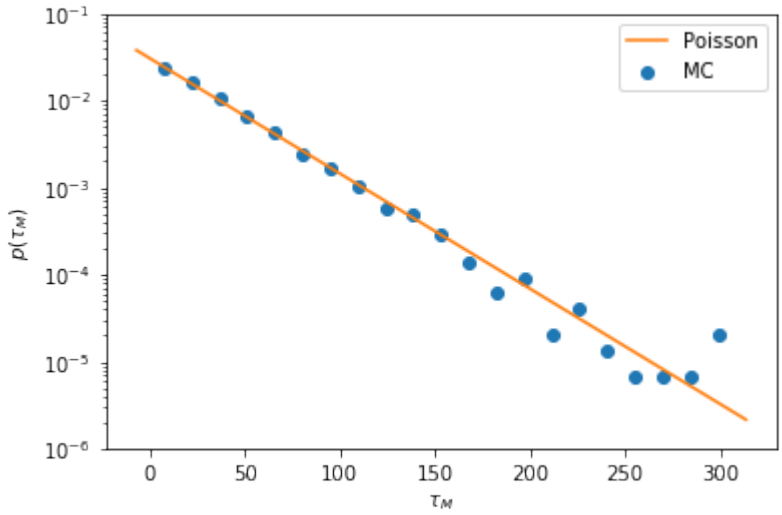
[7]:
```
tau_samples = tau.escapetime_sample(-1, 0, 0, ntraj=10000)
```

[8]:
```
tau.escapetime_pdfplot(tau.escapetime_pdf(tau_samples))
```



When the noise is small enough, it is expected that the transition times are Poisson distributed. Their statistics is entirely determined by the transition rate (the parameter of the Poisson distribution), which is the inverse of the average time between two transitions.

[9]:
```
ax = plt.axes(xlabel=r'$\tau_M$', ylabel=r'$p(\tau_M)$', yscale='log', ylim=(1e-6, 0.
↪1))
ax.scatter(*tau.escapetime_pdf(tau_samples), label='MC');
t = np.linspace(*ax.get_xlim())
ax.plot(t, np.exp(-t/np.mean(tau_samples))/np.mean(tau_samples), color='C1', label=
↪'Poisson');
ax.legend();
```



2.3.3 Computing the transition rate

Let us now compute the transition rate, or equivalently, the mean first-passage time. For 1D homogeneous processes, a theoretical result can be obtained analytically:

$$\mathbb{E}[\tau_M] = \frac{1}{D}\int_{-1}^{M} dx e^{V(x)/D} \int_{-\infty}^{x} e^{-V(y/D} dy.$$

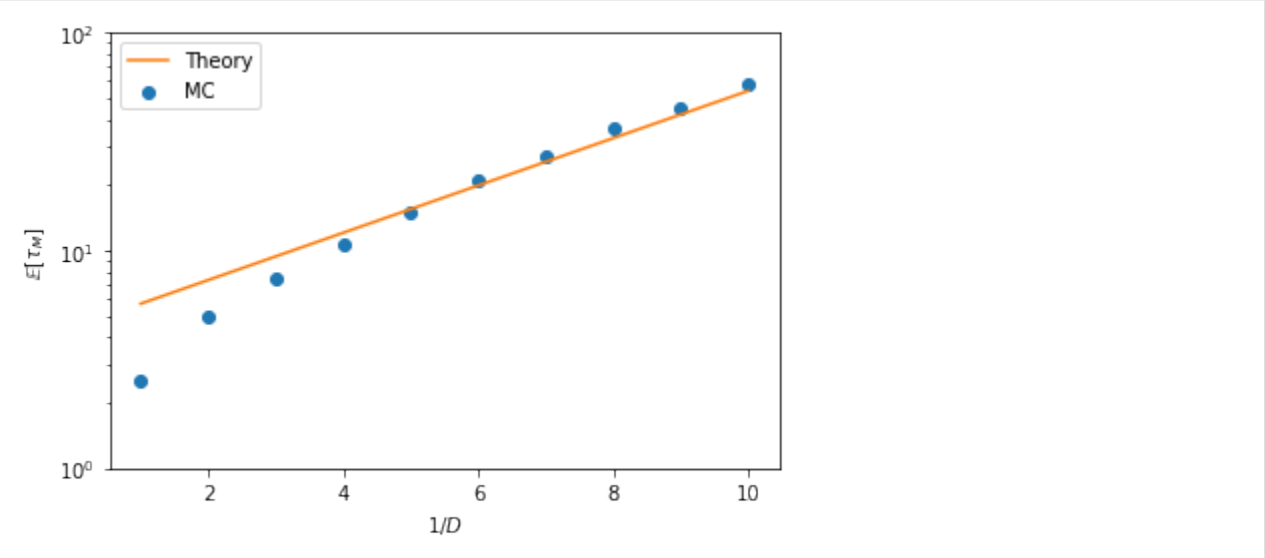
$In the small noise limit : math : `D \to 0`, a saddle - point approximation yields the Eyring - Kramers formula :$

$$\mathbb{E}[\tau_M] \approx \frac{2\pi}{\sqrt{|V''(0)V''(-1)|}} e^{\Delta V/D},$$

$for : math : `M > 0` and : math : `\Delta V = V(0) - V(-1) = 1/4` here.$

[10]:
```
mfpt_mc = np.array([sr.firstpassage.FirstPassageProcess(DoubleWell(D)).escapetime_
↪avg(-1, 0, 0.5, ntraj=1000)
                    for D in 1./np.arange(1, 11)])
```

[13]:
```
ax = plt.axes(xlabel=r'$1/D$', ylabel=r'$\mathbb{E}[\tau_M]$', yscale='log', ylim=(1,␣
↪100))
ax.scatter(np.arange(1, 11), mfpt_mc, label='MC');
ax.plot(np.linspace(1, 10), np.sqrt(2)*np.pi*np.exp(0.25*np.linspace(1, 10)), label=
↪'Theory', color='C1')
ax.legend();
```



The code also allows for computing the mean first passage time using the Fokker-Planck equation or its adjoint. We will soon update this notebook to show this method.

One could also show how the prefactor of the first-passage time depends on the threshold M (above we chose $M = 0.5$), for a fixed noise amplitude D. For D small enough, there is a sharp transition around $M = 0$, and the first-passage time depends very little on M away from this boundary layer (and it is given by the above Eyring-Kramers formula). In general, the expression of the first-passage time is always given by the integral formula before the saddle-point approximation.

Finally, these transitions can also be characterized using the *instanton formalism*. We shall also illustrate this in a future version of the notebook.

2.4 Return times with rare event algorithms

This tutorial demonstrates some of the features of the `stochrare` package for rare event simulation.

As an example, we show how to compute *return times* using the block maximum method (a standard method applicable to any time series) and using a particular rare event algorithm, the *Adaptive Multilevel Splitting* algorithm.

Let us first import the modules:

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import stochrare as sr
np.random.seed(seed=100)
```

As an illustration, we shall work with a very simple stochastic process, the *Ornstein-Uhlenbeck* process in 1D:

[2]:
```
oup = sr.dynamics.diffusion1d.OrnsteinUhlenbeck1D(0, 1, 0.5)
```

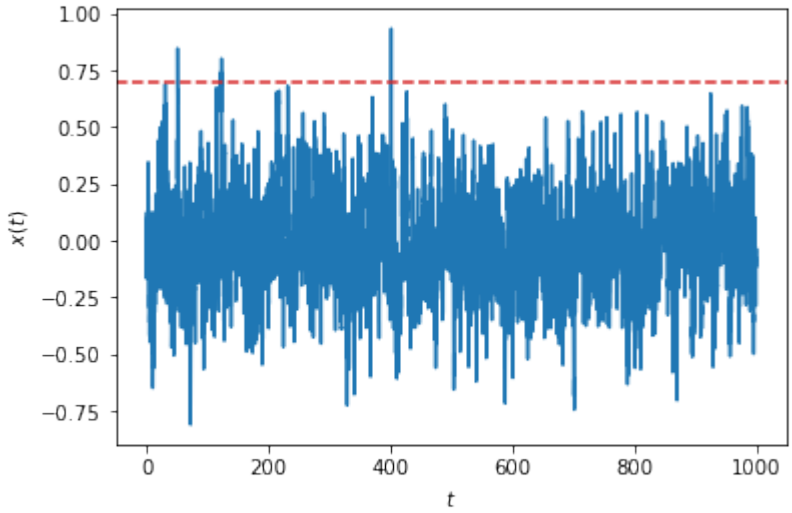
We simulate a realization $x(t)$ of this process with many samples:

[3]:
```
%%time
reftraj = oup.trajectory(0., 0., T=10000)
```

```
CPU times: user 1.31 s, sys: 25 ms, total: 1.34 s
Wall time: 1.32 s
```

We are interested in rare events corresponding to extreme values of the process. For instance, we represent below occurrences when the signal reaches a certain threshold a.

[4]:
```
_, ax = sr.io.plot.trajectory_plot1d((reftraj[0][:10000], reftraj[1][:10000]));
ax.axhline(y=0.7, color='C3', ls='dashed');
```



Such events may be characterized by their probability of occurrence per unit time. Conversely, one may consider the typical time associated with the event, for instance by looking at the average time between two successive independent events. When the events are rare enough (i.e. the threshold a is sufficiently large), the two quantities are inverse of each other. The event follow Poisson statistics, and the parameter of the Poisson distribution is the inverse of the average time between two successive independent events. This time is called the *return time* of the event. It is a very useful metric to quantify rare events. In this notebook, we show how to compute the return time $r(a)$ as a function of the amplitude a of the event using different methods.

This tutorial is inspired from the paper:

Computing return times or return periods with rare event algorithms, T. Lestang, F. Ragone, C.-E. Bréhier, C. Herbert and F. Bouchet, J. Stat. Mech. (2018).

2.4.1 Return times with the block maximum method

The idea of the block maximum method is to divide the trajectory into M blocks of duration ΔT (so that the total length of the trajectory is $T_d = M\Delta T$) much larger than the correlation time of the timeseries τ_c (to make sure that the events are independent). On each block, the maximum value is computed:

$$a_m = \max\{x(t) \| (m-1)\Delta T \leq t \leq m\Delta T\}.$$

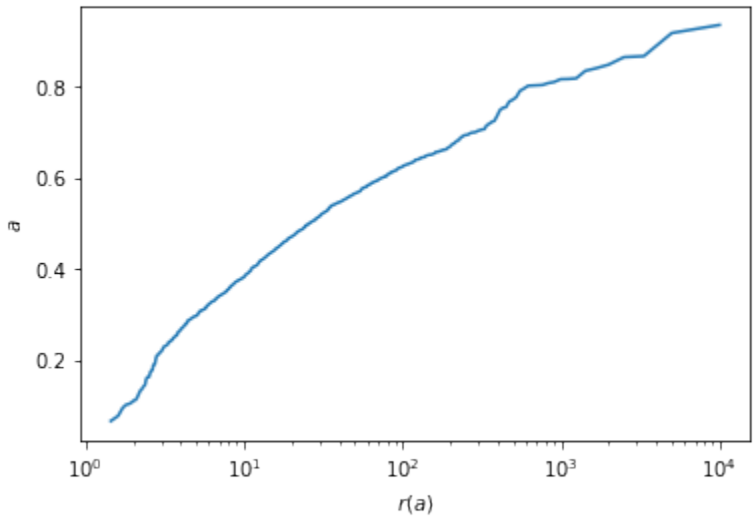
To the largest of these values, we associate a return time T_d, to the second largest value we associate a return time $T_d/2$, and so on.

This procedure is implemented in the `stochrare.timeseries.blockmaximum` routine. In addition to the time series itself, the block size ΔT should be given to the routine. It should be chosen such that $\tau_c \ll \Delta T \ll r(a)$.

[5]:
```
%%time
aref, rref = zip(*sr.timeseries.blockmaximum(reftraj[1], 1000, mode='returnitime',␣
↪time=reftraj[0], modified=True))
```

```
CPU times: user 12 ms, sys: 5.31 ms, total: 17.3 ms
Wall time: 13.4 ms
```

[6]:
```
ax = plt.axes(xlabel=r'$r(a)$', ylabel=r'$a$', xscale='log')
ax.plot(rref, aref);
```



2.4.2 Return times with the *Adaptive Multilevel Splitting* algorithm

As explained in Lestang et al. (2019), the method can be extended to non-equiprobable blocks like trajectories simulated by rare event algorithms.

As an example, we use the *Trajectory Adaptive Multilevel Splitting algorithm*, which is defined by a score function (below it is just the identity) and a fixed length for trajectories (equal to 5 below). The idea of the algorithm is to evolve an ensemble of trajectories through selection-mutation steps. At each iteration, the poorest performers of the ensemble (measured by the score function) are *killed* and replaced by a copy of a *surviving* trajectory resampled from the point where it reached the maximum score function level obtained by the killed trajectory.

We first define the `TAMS` object, which requires a dynamics, a score function and the duration for trajectories:

[7]:
```
tams = sr.rare.ams.TAMS(oup, (lambda t, x: x), 5.)
```

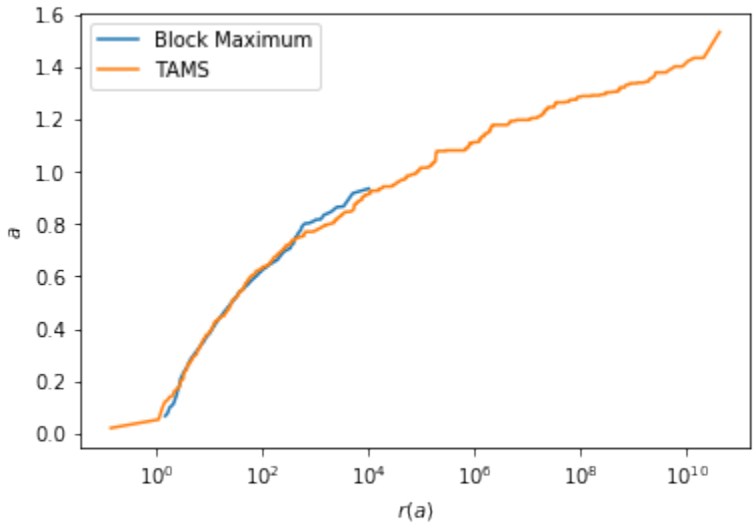
Then, we run the algorithm. Here we use directly the `returntimes` method, which samples trajectories by running the algorithm and then computes the corresponding return times. The method takes as arguments the number of member of the initial ensemble (here 100), and the number of iterations for the algorithm (here 600).

[8]:
```
%%time
aams, rams = tams.returntimes(100, 600)
```

```
CPU times: user 726 ms, sys: 6.14 ms, total: 732 ms
Wall time: 732 ms
```

Let us compare the solution to the one obtained with the Block Maximum method:

[9]:
```
ax = plt.axes(xlabel=r'$r(a)$', ylabel=r'$a$', xscale='log')
ax.plot(rref, aref, label='Block Maximum');
ax.plot(rams, aams, label='TAMS');
ax.legend();
```



In this simple example, for a similar computational cost, the AMS algorithm allows to estimate return times 7 orders of magnitude larger than the Block Maximum method. This depends on the realization, and to properly characterize the performance of the AMS algorithm one would need to study the statistics over an ensemble of realizations.

2.4.3 Applications of the rare event algorithms

For more information about application of the algorithm shown here to sample efficiently rare events, we refer the reader to the following articles:

- F. Cérou, A. Guyader, T. Lelièvre and D. Pommier, *J. Chem. Phys.* 134, 054108 (2011).
- J. Rolland, F. Bouchet and E. Simonnet, *J. Stat. Phys.* 162, 277–311 (2015).
- F. Ragone, J. Wouters and F. Bouchet, *Proc. Nat. Acad. Sci.* 115, 24-29 (2018).
- T. Lestang, F. Ragone, C.-E. Bréhier, C. Herbert and F. Bouchet, *J. Stat. Mech.* (2018).
- J. Rolland, *Phys. Rev. E* 97, 023109 (2018).
- F. Bouchet, J. Rolland and E. Simonnet, *Phys. Rev. Lett.* 122, 074502 (2019).

In addition, a recent review of the AMS algorithm can be found here:

- F. Cérou, A. Guyader, and M. Rousset, *Chaos* 29, 043108 (2019).

CHAPTER 3

API Reference

stochrare is organized in several modules serving different purposes:

stochrare.dynamics	Sample stochastic processes
stochrare.fokkerplanck	Numerical solvers for the Fokker-Planck equations
stochrare.firstpassage	First-passage processes
stochrare.rare	Rare event algorithms
stochrare.io	Input/Output
stochrare.utils	Utilities

3.1 stochrare.dynamics

3.1.1 Sample stochastic processes

This is the core module of stochrare. It contains submodules for simulating trajectories corresponding to different stochastic dynamics.

For now, only diffusion processes are available.

diffusion	Simulating diffusion processes in arbitrary dimensions
diffusion1d	Simulating 1D diffusion processes

stochrare.dynamics.diffusion

Simulating diffusion processes in arbitrary dimensions

This module defines the *DiffusionProcess* class, representing generic diffusion processes with arbitrary drift and diffusion coefficients, in arbitrary dimension.

This class can be subclassed for specific diffusion processes for which methods can be specialized, both to simplify

the code (e.g. directly enter analytical formulae when they are available) and for performance. As an exemple of this mechanism, we also provide in this module the *ConstantDiffusionProcess* class, for which the diffusion term is constant and proportional to the identity matrix, the *OrnsteinUhlenbeck* class representing the particular case of the Ornstein-Uhlenbeck process, and the *Wiener* class corresponding to Brownian motion. These classes form a hierarchy deriving from the base class, *DiffusionProcess*.

class stochrare.dynamics.diffusion.**DiffusionProcess**(*vecfield*, *sigma*, ***kwargs*)

Bases: object

Generic class for diffusion processes in arbitrary dimensions.

It corresponds to the family of SDEs $dx_t = F(x_t, t)dt + \sigma(x_t, t)dW_t$, where F is a time-dependent N-dimensional vector field and W the M-dimensional Wiener process. The diffusion matrix sigma has size NxM.

Parameters

- **vecfield** (*function with two arguments*) – The vector field $F(x, t)$.
- **sigma** (*function with two arguments*) – The diffusion coefficient $\sigma(x, t)$.

update(*xn*, *tn*, ***kwargs*)

Return the next sample for the time-discretized process.

Parameters

- **xn** (*ndarray*) – A n-dimensional vector (in $\mathbb{R}^n$).
- **tn** (*float*) – The current time.

Keyword Arguments

- **dt** (*float*) – The time step.
- **dw** (*ndarray*) – The brownian increment if precomputed. By default, it is generated on the fly from a Gaussian distribution with variance dt.

Returns **x** – The position at time tn+dt.

Return type ndarray

Notes

This method uses the Euler-Maruyama method[1][2]: $x_{n+1} = x_n + F(x_n, t_n)\Delta t + \sigma(x_n, t_n)\Delta W_n$, for a fixed time step Δt, where ΔW_n is a random vector distributed according to the standard normal distribution[1][2].

It is the straightforward generalization to SDEs of the Euler method for ODEs.

The Euler-Maruyama method has strong order 0.5 and weak order 1.

References

trajectory(*x0*, *t0*, ***kwargs*)

Integrate the SDE with given initial condition.

Parameters

- **x0** (*ndarray*) – The initial position (in $\mathbb{R}^n$).
- **t0** (*float*) – The initial time.

Keyword Arguments

[1] G. Maruyama, "Continuous Markov processes and stochastic equations", Rend. Circ. Mat. Palermo 4, 48-90 (1955).
[2] P. E. Kloeden and E. Platen, "Numerical solution of stochastic differential equations", Springer (1992).

- **dt** (*float*) – The time step, forwarded to the *update()* routine (default 0.1, unless overridden by a subclass).
- **T** (*float*) – The time duration of the trajectory (default 10).
- **finite** (*bool*) – Filter finite values before returning trajectory (default False).

Returns **t, x** – Time-discrete sample path for the stochastic process with initial conditions (t0, x0). The array t contains the time discretization and x the value of the sample path at these instants.

Return type ndarray, ndarray

trajectory_generator(*x0*, *t0*, *nsteps*, ***kwargs*)

Integrate the SDE with given initial condition, generator version.

Parameters

- **x0** (*ndarray*) – The initial position (in $\mathbb{R}^n$).
- **t0** (*float*) – The initial time.
- **nsteps** (*int*) – The number of samples to generate.

Keyword Arguments

- **dt** (*float*) – The time step, forwarded to the *update()* routine (default 0.1, unless overridden by a subclass).
- **observable** (*function with two arguments*) – Time-dependent observable $O(x, t)$ to compute (default $O(x, t) = x$)

Yields **t, y** (*ndarray, ndarray*) – Time-discrete sample path (or observable) for the stochastic process with initial conditions (t0, x0). The array t contains the time discretization and y=O(x, t) the value of the observable (it may be the stochastic process itself) at these instants.

sample_mean(*x0*, *t0*, *nsteps*, *nsamples*, ***kwargs*)

Compute the sample mean of a time dependent observable, conditioned on initial conditions.

Parameters

- **x0** (*ndarray*) – The initial position (in $\mathbb{R}^n$).
- **t0** (*float*) – The initial time.
- **nsteps** (*int*) – The number of samples in each sample path.
- **nsamples** (*int*) – The number of sample paths in the ensemble.

Keyword Arguments

- **dt** (*float*) – The time step, forwarded to the *update()* routine (default 0.1, unless overridden by a subclass).
- **observable** (*function with two arguments*) – Time-dependent observable $O(x, t)$ to compute (default $O(x, t) = x$)

Yields **t, y** (*ndarray, ndarray*) – Time-discrete ensemble mean for the observable, conditioned on the initial conditions (t0, x0). The array t contains the time discretization and $y = \mathbb{E}[O(x, t)]$ the value of the sample mean of the observable (it may be the stochastic process itself) at these instants.

class stochrare.dynamics.diffusion.**ConstantDiffusionProcess**(*vecfield*, *Damp*, *dim*, ***kwargs*)

Bases: *stochrare.dynamics.diffusion.DiffusionProcess*

Diffusion processes, in arbitrary dimensions, with constant diffusion coefficient.

It corresponds to the family of SDEs $dx_t = F(x_t, t)dt + \sigma dW_t$, where F is a time-dependent N-dimensional vector field and W the N-dimensional Wiener process. The diffusion coefficient σ is independent of the stochastic process (additive noise) and time, and we further assume that it is proportional to the identity matrix: all the components of the noise are independent.

Parameters

- **vecfield** (*function with two arguments*) – The vector field $F(x, t)$.
- **Damp** (*float*) – The amplitude of the noise.
- **dim** (*int*) – The dimension of the system.

Notes

The diffusion coefficient is given by $\sigma = \sqrt{2\text{Damp}}$. This convention leads to simpler expressions, for instance for the Fokker-Planck equations.

update (*xn*, *tn*, ***kwargs*)

Return the next sample for the time-discretized process.

Parameters

- **xn** (*ndarray*) – A n-dimensional vector (in $\mathbb{R}^n$).
- **tn** (*float*) – The current time.

Keyword Arguments

- **dt** (*float*) – The time step.
- **dw** (*ndarray*) – The brownian increment if precomputed. By default, it is generated on the fly from a Gaussian distribution with variance dt.

Returns **x** – The position at time tn+dt.

Return type ndarray

See also:

DiffusionProcess.update() for details about the Euler-Maruyama method.

Notes

This is the same as the *DiffusionProcess.update()* method from the parent class *DiffusionProcess*, except that a matrix product is no longer necessary.

class stochrare.dynamics.diffusion.**OrnsteinUhlenbeck** (*mu*, *theta*, *D*, *dim*, ***kwargs*)

Bases: *stochrare.dynamics.diffusion.ConstantDiffusionProcess*

The Ornstein-Uhlenbeck process, in arbitrary dimensions.

It corresponds to the SDE $dx_t = \theta(\mu - x_t)dt + \sqrt{2D}dW_t$, where $\theta > 0$ and $\mu \in \mathbb{R}^n$ are arbitrary coefficients and $D > 0$ is the amplitude of the noise.

Parameters

- **mu** (*ndarray*) – The expectation value.
- **theta** (*float*) – The inverse of the relaxation time.
- **D** (*float*) – The amplitude of the noise.
- **dim** (*int*) – The dimension of the system.

Notes

The Ornstein-Uhlenbeck process has been used to model many systems. It was initially introduced to describe the motion of a massive Brownian particle with friction[3] . It may also be seen as a diffusion process in a harmonic potential.

Because many of its properties can be computed analytically, it provides a useful toy model for developing new methods.

References

potential (*x*)

Compute the potential from which the force derives.

Parameters **x** (*ndarray*) – The point where we want to compute the potential

Returns **V** – The potential from which the force derives, at the given point.

Return type float

Notes

Not all diffusion processes derive from a potential, but the Ornstein Uhlenbeck does. It is a gradient system, with a quadratic potential: $dx_t = -\nabla V(x_t)dt + \sqrt{2D}dW_t$, with $V(x) = \theta(\mu - x)^2/2$.

class stochrare.dynamics.diffusion.**Wiener**(*dim*, *D=1*, ***kwargs*)

Bases: *stochrare.dynamics.diffusion.OrnsteinUhlenbeck*

The Wiener process, in arbitrary dimensions.

Parameters

- **dim** (*int*) – The dimension of the system.
- **D** (*float, optional*) – The amplitude of the noise (default is 1).

Notes

The Wiener process is a central object in the theory or stochastic processes, both from a mathematical point of view and for its applications in different scientific fields. We refer to classical textbooks for more information about the Wiener process and Brownian motion.

classmethod potential (*x*)

Compute the potential from which the force derives.

Parameters **x** (*ndarray*) – The point where we want to compute the potential.

Returns **V** – The potential from which the force derives, at the given point.

Return type float

Notes

The Wiener Process is a trivial gradient system, with vanishing potential. It is useless (and potentially source of errors) to call the general potential routine, so we just return zero directly.

[3] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of Brownian Motion". Phys. Rev. 36, 823–841 (1930).

Classes

ConstantDiffusionProcess(vecfield, Damp, ...)	Diffusion processes, in arbitrary dimensions, with constant diffusion coefficient.
DiffusionProcess(vecfield, sigma, **kwargs)	Generic class for diffusion processes in arbitrary dimensions.
OrnsteinUhlenbeck(mu, theta, D, dim, **kwargs)	The Ornstein-Uhlenbeck process, in arbitrary dimensions.
Wiener(dim[, D])	The Wiener process, in arbitrary dimensions.

stochrare.dynamics.diffusion1d

Simulating 1D diffusion processes

This module defines the *DiffusionProcess1D* class, representing diffusion processes with arbitrary drift and diffusion coefficients in 1D.

This class can be subclassed for specific diffusion processes for which methods can be specialized, both to simplify the code (e.g. directly enter analytical formulae when they are available) and for performance. As an exemple of this mechanism, we also provide in this module the *ConstantDiffusionProcess1D* class, for which the diffusion term is constant (additive noise), the *OrnsteinUhlenbeck1D* class representing the particular case of the Ornstein-Uhlenbeck process, and the *Wiener1D* class corresponding to Brownian motion. These classes form a hierarchy deriving from the base class, *DiffusionProcess1D*.

class stochrare.dynamics.diffusion1d.**DiffusionProcess1D**(*vecfield*, *sigma*, ***kwargs*)

Bases: object

Generic class for 1D diffusion processes.

It corresponds to the family of 1D SDEs $dx_t = F(x_t, t)dt + \sigma(x_t, t)dW_t$, where F is a time-dependent vector field and W the Wiener process.

Parameters

- **vecfield** (*function with two arguments*) – The vector field $F(x, t)$.
- **sigma** (*function with two arguments*) – The diffusion coefficient $\sigma(x, t)$.

potential(*X*, *t*)

Compute the potential from which the force derives.

Parameters **X** (*ndarray*) – The points where we want to compute the potential.

Returns **V** – The potential from which the force derives, at the given points.

Return type ndarray

Notes

We integrate the vector field to obtain the value of the underlying potential at the input points. Caveat: This works only for 1D dynamics.

update(*xn*, *tn*, ***kwargs*)

Return the next sample for the time-discretized process.

Parameters

- **xn** (*float*) – The current position.

- **tn** (*float*) – The current time.

Keyword Arguments

- **dt** (*float*) – The time step (default 0.1 if not overriden by a subclass).
- **dw** (*float*) – The brownian increment if precomputed. By default, it is generated on the fly from a Gaussian distribution with variance dt.

Returns **x** – The position at time tn+dt.

Return type float

Notes

This method uses the Euler-Maruyama method[1][2]: $x_{n+1} = x_n + F(x_n, t_n)\Delta t + \sigma(x_n, t_n)\Delta W_n$.

It is the straightforward generalization to SDEs of the Euler method for ODEs.

The Euler-Maruyama method has strong order 0.5 and weak order 1.

References

trajectory (*x0*, *t0*, ***kwargs*)
Integrate the SDE with given initial condition.

Parameters

- **x0** (*float*) – The initial position.
- **t0** (*float*) – The initial time.

Keyword Arguments

- **dt** (*float*) – The time step, forwarded to the *update()* routine (default 0.1, unless overridden by a subclass).
- **T** (*float*) – The time duration of the trajectory (default 10).
- **brownian_path** (*(ndarray, ndarray)*) – A precomputed Brownian path with respect to which we integrate the SDE. If not provided (default behavior), one will be computed one the fly.
- **deltat** (*float*) – The time step for the Brownian path, when generated on the fly (default: dt).
- **finite** (*bool*) – Filter finite values before returning trajectory (default False).

Returns **t, x** – Time-discrete sample path for the stochastic process with initial conditions (t0, x0). The array t contains the time discretization and x the value of the sample path at these instants.

Return type ndarray, ndarray

trajectory_conditional (*x0*, *t0*, *pred*, ***kwargs*)
Compute sample path satisfying arbitrary condition.

Parameters

- **x0** (*float*) – The initial position.

[1] G. Maruyama, Continuous Markov processes and stochastic equations, Rend. Circ. Mat. Palermo 4, 48-90 (1955).
[2] P. E. Kloeden and E. Platen, Numerical Solution of Stochastic Differential Equations, Springer (1992).

- **t0** (*float*) – The initial time.
- **pred** (*function with two arguments*) – The predicate to select trajectories.

Keyword Arguments

- **dt** (*float*) – The time step, forwarded to the *update()* routine (default 0.1, unless overridden by a subclass).
- **T** (*float*) – The time duration of the trajectory (default 10).
- **finite** (*bool*) – Filter finite values before returning trajectory (default False).

Returns **t, x** – Time-discrete sample path for the stochastic process with initial conditions (t0, x0). The array t contains the time discretization and x the value of the sample path at these instants.

Return type ndarray, ndarray

blowuptime (*x0*, *t0*, ***kwargs*)
Compute the last time with finite values, for one realization.

Parameters

- **x0** (*float*) – The initial position.
- **t0** (*float*) – The initial time.

Returns

- *The last time with finite values for a realization with initial conditions (t0, x0).*
- *This is a random variable.*

empirical_vector (*x0*, *t0*, *nsamples*, **args*, ***kwargs*)
Empirical vector at given times.

Parameters

- **x0** (*float*) – Initial position.
- **t0** (*float*) – Initial time.
- **nsamples** (*int*) – The size of the ensemble.
- ***args** (*variable length argument list*) – The times at which we want to estimate the empirical vector.

Keyword Arguments ****kwargs** – Keyword arguments forwarded to *trajectory()* and to `numpy.histogram()`.

Yields **t, pdf, bins** (*float, ndarray, ndarray*) – The time and histogram of the stochastic process at that time.

Notes

This method computes the empirical vector, or in other words, the relative frequency of the stochastic process at different times, conditioned on the initial condition. At each time, the empirical vector is a random vector. It is an estimator of the transition probability $p(x, t|x_0, t_0)$.

classmethod trajectoryplot (**args*, ***kwargs*)
Plot 1D trajectories.

Parameters

- ***args** (*variable length argument list*) –

- **trajs** (*tuple (t, x)*) –

Keyword Arguments

- **fig** (*matplotlig.figure.Figure*) – Figure object to use for the plot. Create one if not provided.
- **ax** (*matplotlig.axes.Axes*) – Axes object to use for the plot. Create one if not provided.
- ****kwargs** – Other keyword arguments forwarded to matplotlib.pyplot.axes.

Returns **fig, ax** – The figure.

Return type matplotlib.figure.Figure, matplotlib.axes.Axes

Notes

This is just an interface to the function *stochrare.io.plot.trajectory_plot1d()*. However, it may be overwritten in subclasses to systematically include elements to the plot which are specific to the stochastic process.

class stochrare.dynamics.diffusion1d.**ConstantDiffusionProcess1D**(*vecfield*, *Damp*, ***kwargs*)

Bases: *stochrare.dynamics.diffusion1d.DiffusionProcess1D*

Diffusion processes in 1D with constant diffusion coefficient (additive noise).

It corresponds to the family of SDEs $dx_t = F(x_t, t)dt + \sigma dW_t$, where F is a time-dependent vector field and W the Wiener process. The diffusion coefficient σ is independent of space and time.

Parameters

- **vecfield** (*function with two arguments*) – The vector field $F(x, t)$.
- **Damp** (*float*) – The amplitude of the noise.

Notes

The diffusion coefficient is given by $\sigma = \sqrt{2\text{Damp}}$. This convention leads to simpler expressions, for instance for the Fokker-Planck equations.

update(*xn*, *tn*, ***kwargs*)

Return the next sample for the time-discretized process.

Parameters

- **xn** (*float*) – The current position.
- **tn** (*float*) – The current time.

Keyword Arguments

- **dt** (*float*) – The time step (default 0.1 if not overriden by a subclass).
- **dw** (*float*) – The brownian increment if precomputed. By default, it is generated on the fly from a Gaussian distribution with variance dt.

Returns **x** – The position at time tn+dt.

Return type float

Notes

This method uses the Euler-Maruyama method[3][4]: $x_{n+1} = x_n + F(x_n, t_n)\Delta t + \sqrt{2D}\Delta W_n$.

References

traj_cond_gen (*x0*, *t0*, *tau*, *M*, ***kwargs*)
Generate trajectories conditioned on the first-passage time tau at value M.

Parameters

- **x0** (*float*) – Initial position.
- **t0** (*float*) – Initial time.
- **tau** (*float*) – The value of the first passage time required.
- **M** (*float*) – The threshold for the first passage time.

Keyword Arguments

- **dt** (*float*) – The integration timestep (default is self.default_dt).
- **ttol** (*float*) – The first-passage time tolerance (default is 1% of trajectory duration).
- **num** (*int*) – The number of trajectories generated (default is 10).
- **interp** (*bool*) – Interpolate to generate unifomly sampled trajectories.
- **npts** (*int*) – The number of points for interpolated trajectories (default (tau-t0)/dt).

Yields **t, x** (*ndarray, ndarray*) – Trajectories satisfying the condition on the first passage time.

pdfplot (**args*, ***kwargs*)
Plot the pdf P(x,t) at various times.

Parameters **args** (*variable length argument list*) – The times at which to plot the PDF.

Keyword Arguments

- **t0** (*float*) – Initial time.
- **potential** (*bool*) – Plot potential on top of PDF.
- **th** (*bool*) – Plot theoretical solution, if it exists, on top of PDF.

instanton (*x0*, *p0*, **args*, ***kwargs*)
Numerical integration of the equations of motion for instantons. x0 and p0 are the initial conditions. Return the instanton trajectory (t,x).

action (**args*)
Compute the action for all the trajectories given as arguments

class stochrare.dynamics.diffusion1d.**OrnsteinUhlenbeck1D** (*mu*, *theta*, *D*, ***kwargs*)
Bases: *stochrare.dynamics.diffusion1d.ConstantDiffusionProcess1D*

The 1D Ornstein-Uhlenbeck process.

It corresponds to the SDE $dx_t = \theta(\mu - x_t)dt + \sqrt{2D}dW_t$, where $\theta > 0$ and μ are arbitrary coefficients and $D > 0$ is the amplitude of the noise.

Parameters

[3] G. Maruyama, Continuous Markov processes and stochastic equations, Rend. Circ. Mat. Palermo 4, 48-90 (1955).
[4] P. E. Kloeden and E. Platen, Numerical Solution of Stochastic Differential Equations, Springer (1992).

- **mu** (*float*) – The expectation value.
- **theta** (*float*) – The inverse of the relaxation time.
- **D** (*float*) – The amplitude of the noise.

Notes

The Ornstein-Uhlenbeck process has been used to model many systems. It was initially introduced to describe the motion of a massive Brownian particle with friction[5] . It may also be seen as a diffusion process in a harmonic potential.

Because many of its properties can be computed analytically, it provides a useful toy model for developing new methods.

References

update(*xn*, *tn*, ***kwargs*)

Return the next sample for the time-discretized process, using the Gillespie method.

Parameters

- **xn** (*float*) – The current position.
- **tn** (*float*) – The current time.

Keyword Arguments

- **dt** (*float*) – The time step (default 0.1 if not overriden by a subclass).
- **dw** (*float*) – The brownian increment if precomputed. By default, it is generated on the fly from a standard Gaussian distribution.
- **method** (*str*) – The numerical method for integration: 'gillespie' (default) or 'euler'.

Returns **x** – The position at time tn+dt.

Return type float

Notes

For the Ornstein-Uhlenbeck process, there is an exact method, the Gillespie algorithm[6]. This method is selected by default. If necessary, the Euler-Maruyama method can still be chosen using the `method` keyword argument.

References

class stochrare.dynamics.diffusion1d.**Wiener1D**(*D=1*, ***kwargs*)

Bases: *stochrare.dynamics.diffusion1d.OrnsteinUhlenbeck1D*

The 1D Wiener process.

Parameters **D** (*float, optional*) – The amplitude of the noise (default is 1).

[5] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of Brownian Motion". Phys. Rev. 36, 823–841 (1930).
[6] D. T. Gillespie, Exact numerical simulation of the Ornstein-Uhlenbeck process and its integral, Phys. Rev. E 54, 2084 (1996).

Notes

The Wiener process is a central object in the theory or stochastic processes, both from a mathematical point of view and for its applications in different scientific fields. We refer to classical textbooks for more information about the Wiener process and Brownian motion.

classmethod potential(*X*)

Compute the potential from which the force derives.

Parameters **X** (*ndarray*) – The points where we want to compute the potential.

Returns **V** – The potential from which the force derives, at the given points.

Return type float

Notes

The Wiener Process is a trivial gradient system, with vanishing potential. It is useless (and potentially source of errors) to call the general potential routine, so we just return zero directly.

Classes

ConstantDiffusionProcess1D(vecfield, Damp, ...)	Diffusion processes in 1D with constant diffusion coefficient (additive noise).
DiffusionProcess1D(vecfield, sigma, **kwargs)	Generic class for 1D diffusion processes.
DrivenOrnsteinUhlenbeck1D(mu, theta, D, A, ...)	The 1D Ornstein-Uhlenbeck model driven by a periodic forcing:
OrnsteinUhlenbeck1D(mu, theta, D, **kwargs)	The 1D Ornstein-Uhlenbeck process.
Wiener1D([D])	The 1D Wiener process.

3.2 stochrare.fokkerplanck

3.2.1 Numerical solvers for the Fokker-Planck equations

This module contains numerical solvers for the Fokker-Planck equations associated to diffusion processes.

For now, it only contains a basic finite difference solver for the 1D case.

class stochrare.fokkerplanck.**FokkerPlanck1D**(*drift*, *diffusion*)

Bases: object

Solver for the 1D Fokker-Planck equation.

$\partial_t P(x,t) = -\partial_x a(x,t) P(x,t) + D\partial^2_{xx} P(x,t)$

Parameters

- **drift** (*function with two variables*) – The drift coefficient $a(x,t)$.
- **diffusion** (*float*) – The constant diffusion coefficient D.

Notes

This is just the legacy code which was migrated from the stochrare.dynamics.DiffusionProcess1D class. It should be rewritten with a better structure. In particular, it only

works with a constant diffusion for now.

classmethod gaussian1d(*mean*, *std*, *X*)
Return a 1D Gaussian pdf.

Parameters

- **mean** (*float*) –
- **std** (*float*) –
- **X** (*ndarray*) – The sample points.

Returns **pdf** – The Gaussian pdf at the sample points.

Return type ndarray

fpintegrate(*t0*, *T*, ***kwargs*)
Numerical integration of the associated Fokker-Planck equation, or its adjoint.

Parameters

- **t0** (*float*) – Initial time.
- **T** (*float*) – Integration time.

Keyword Arguments

- **bounds** (*float 2-tuple*) – Domain where we should solve the equation (default (-10.0,10.0))
- **npts** (*ints*) – Number of discretization points in the domain (i.e. spatial resolution). Default: 100.
- **dt** (*float*) – Timestep (default choice suitable for the heat equation with forward scheme)
- **bc** (*stochrare.edpy.BoundaryCondition object or tuple*) – Boundary conditions (either a BoundaryCondition object or a tuple sent to _fpbc)
- **method** (*str*) – Numerical scheme: explicit ('euler', default), implicit, or crank-nicolson
- **adjoint** (*bool*) – Integrate the adjoint FP rather than the forward FP (default False).
- **P0** (*str*) – Initial condition: 'gauss' (default), 'dirac' or 'uniform'.

Returns **t, X, P** – Final time, sample points and solution of the Fokker-Planck equation at the sample points.

Return type float, ndarray, ndarray

fpintegrate_generator(**args*, ***kwargs*)
Numerical integration of the associated Fokker-Planck equation, generator version.

Parameters ***args** (*variable length argument list*) – Times at which to yield the pdf.

Yields **t, X, P** (*float, ndarray, ndarray*) – Time, sample points and solution of the Fokker-Planck equation at the sample points.

Classes

FokkerPlanck1D(drift, diffusion)	Solver for the 1D Fokker-Planck equation.

3.3 stochrare.firstpassage

3.3.1 First-passage processes

This module defines a class corresponding to the random variable defined as the first-passage time in a given set for a given stochastic process.

class stochrare.firstpassage.**FirstPassageProcess**(*model*)

Bases: object

Represents a first-passage time random variable associated to a stochastic process and a given set.

Parameters

- **model** (*stochrare.dynamics.DiffusionProcess1D*) – The stochastic process to which the first-passage time is associated
- **CAUTION** (*methods only tested with ConstantDiffusionProcess1D class, not DiffusionProcess1D!*) –

firstpassagetime(*x0*, *t0*, *A*, ***kwargs*)

Computes the first passage time, defined by $ au_A = inf{t>t0 | x(t)>A}$, for one realization

escapetime_sample(*x0*, *t0*, *A*, ***kwargs*)

Computes realizations of the first passage time, defined by $ au_A = inf{t>t0 | x(t)>A}$, using direct Monte-Carlo simulations. This method can be overwritten by subclasses to call compiled code for better performance.

escapetime_avg(*x0*, *t0*, *A*, ***kwargs*)

Compute the average escape time for given initial condition (x0,t0) and threshold A

classmethod escapetime_pdf(*samples*, ***kwargs*)

Compute the probability distribution function of the first-passage time based on the input samples

classmethod escapetime_pdfplot(**args*, ***kwargs*)

Plot previously computed pdf of first passage time

classmethod traj_fpt(*M*, **args*)

Compute the first passage time for each trajectory given as argument

firstpassagetime_cdf(*x0*, *A*, **args*, ***kwargs*)

Computes the CDF of the first passage time, Prob_{x0,t0}[au_A<t] by solving the Fokker-Planck equation

firstpassagetime_moments(*x0*, *A*, **args*, ***kwargs*)

Computes the moments of the first passage time, <tau_A^n>_{x0,t0}, by solving the Fokker-Planck equation

firstpassagetime_avg(*x0*, **args*, ***kwargs*)

Compute the mean first passage time by one of the following methods: solving the FP equation, its adjoint, or using the theoretical solution.

x0 is the initial condition (at t0), and 'args' contains the list of threshold values for which to compute the first passage time.

The theoretical formula is valid only for an homogeneous process; for the computation, we 'freeze' the potential at t=t0.

Classes

FirstPassageProcess(model)	Represents a first-passage time random variable associated to a stochastic process and a given set.

3.4 stochrare.rare

3.4.1 Rare event algorithms

This module contains numerical algorithms designed specifically to sample rare events.

For now, only algorithms of the *Adaptive Multilevel Splitting* family are implemented.

ams	Rare event algorithms of the *Adaptive Multilevel Splitting* family

stochrare.rare.ams

Rare event algorithms of the *Adaptive Multilevel Splitting* family

There are two kinds of variants of the AMS algorithms. On the one hand, there are "scientific" variants, corresponding to different formulations of the algorithm (e.g. AMS vs TAMS). On the other hand, there are "technical" variants, corresponding to different implementations: for instance, keeping all the trajectories in memory or storing them on disk (necessary for applications to complex systems)

class stochrare.rare.ams.**AMS**(*model*, *scorefun*, *initcond=<function AMS.<lambda>>*)

Bases: object

Original version of the *Adaptive Multilevel Splitting* Algorithm.

Parameters

- **model** (*stochrare.dynamics.DiffusionProcess1D object (or a subclass of it)*) – The dynamical model; so far we are restricted to SDEs of the form $dX_t = F(X_t, t) + \sqrt{2D}dW_t$. We only use the stochrare.dynamics.DiffusionProcess1D.update() method of the object.
- **scorefun** (*function with two arguments*) – The score function $\xi(t, x)$.
- **initcond** (*function with no arguments, optional*) – Function to generate initial conditions. It can be for instance a constant: lambda: x0, t0 or generate random initial conditions: lambda: np.random.random(), t0

Notes

The algorithm evolves an ensemble of trajectories in an interactive manner, using selection and mutation steps[1][2][3][4]. The algorithm requires two sets A and B, and a *reactive coordinate* or *score function* ξ, measuring the distance between the two. In fact, we require that ξ vanishes over the boundary of A, and takes unit value over the boundary of B.

- Initialization:

The ensemble is initialized by running N trajectories until they reach set A or set B.

- Selection

Then at each iteration, the maximum value of the score function over each member of the ensemble is computed. The q trajectories with lowest score function are *killed*.

- Mutation

For each trajectory killed, we pick a random trajectory among the survivors. We clone that trajectory until it reaches the level of the killed trajectory for the first time, then we restart it from that point until it reaches set A or B.

The algorithm is iterated until all trajectories reach set B.

References

getcrossingtime(*level*, *times*, *traj*)

Return the time and position at which the trajectory reaches a given threshold.

Parameters

- **level** (*float*) – The threshold.
- **times** (*numpy.ndarray*) – Sampling times for the trajectory.
- **traj** (*numpy.ndarray*) – Position of the system at the sampling times.

Returns **t, x** – The time and position at the crossing point.

Return type float, float

getlevel(*times*, *traj*)

Return the maximum reached by the score function over the trajectory.

Parameters

- **times** (*numpy.ndarray*) – Sampling times for the trajectory.
- **traj** (*numpy.ndarray*) – Position of the system at the sampling times.

Returns **max** – The maximum of the score function over the trajectory.

Return type float

[1] F. Cerou and A. Guyader, Stoch. Anal. Appl. 25, 417 (2007)

[2] F. Cerou, A. Guyader, T. Lelievre and D. Pommier J. Chem. Phys. 134, 054108 (2011)

[3] J. Rolland, F. Bouchet and E. Simonnet, J. Stat. Phys. 162, 277 (2016)

[4] C.-E. Brehier, M. Gazeau, L. Goudenege, T. Lelievre and M. Rousset, Ann. Appl. Probab. 26, 3559 (2016)

resample (*time*, *pos*, *told*, *xold*, ***kwargs*)
Resample a killed trajectory after a given time.

Parameters

- **time** (*float*) – The time from which to resample.
- **pos** (*float*) – The position from which to resample.
- **told** (*numpy.ndarray*) – The sample times from the killed trajectory.
- **xold** (*numpy.ndarray*) – The killed trajectory.

Keyword Arguments ****kwargs** – Keyword arguments, forwarded to *simul_trajectory()*.

Returns **tnew, xnew** – The resampled trajectory.

Return type numpy.ndarray, numpy.ndarray

simul_trajectory (*x0*, *t0*, ***kwargs*)
Simulate a trajectory until it reaches either set A (score <= 0) or set B (score >= 1).

Parameters

- **x0** (*float*) – Initial position.
- **t0** (*float*) – Initial time.

Keyword Arguments **dt** (*float*) – The time step.

Returns **t, x** – The simulated trajectory.

Return type numpy.ndarray, numpy.ndarray

initialize_ensemble (*ntraj*, ***kwargs*)
Generate the initial ensemble.

Parameters **ntraj** (*int*) – Number of trajectories in the ensemble.

Keyword Arguments ****kwargs** – Keyword arguments forwarded to *simul_trajectory()*.

static selectionstep (*levels*, *npart=1*)
Selection step of the AMS algorithm.

Parameters

- **levels** (*numpy.ndarray*) – The list of levels reached by the ensemble members.
- **npart** (*int, optional*) – The number of levels to select. npart=1 corresponds to the last particle method. Note that one level can correspond to several trajectories in the ensemble.

Returns **killed, survivors** – The indices of the killed and surviving ensemble members.

Return type numpy.ndarray, numpy.ndarray

Notes

Return the trajectories in the ensemble which performed worse, i.e. the trajectories for which the maximum value of the score function over the trajectory is minimum.

mutationstep (*killed_pool*, *survivor_pool*, ***kwargs*)
Mutation step for the AMS algorithm.

Parameters

- **killed_pool** (*array_like*) – The indices of the ensemble members to kill.
- **survivor_pool** (*array_like*) – The indices of the ensemble members to keep.

Keyword Arguments ****kwargs** – Keyword arguments forwarded to *resample()*.

Notes

This is the only method which modifies the state of the ensemble (the *AMS* object).

run_iter(*ntraj*, *niter*, ***kwargs*)
Generate trajectories with the AMS algorithm.

Parameters

- **ntraj** (*int*) – The number of trajectories in the initial ensemble.
- **niter** (*int*) – The number of iterations of the algorithm.
- **Arguments** (*Keywords*) –
- **----------** –
- ****kwargs** – Keyword arguments passed to the "trajectory" method of the dynamics object.

Yields trajectory, weight (*numpy.ndarray, float*) – The generator yields (trajectory, weight) pairs which allows to compute easily the probability associated to each sampled trajectory.

Notes

This method yields first the killed trajectories as the algorithm is iterated, then the trajectories in the final ensemble.

run_resamp(*ntraj*, *niter*, ***kwargs*)
Generate trajectories with the AMS algorithm.

Parameters

- **ntraj** (*int*) – The number of trajectories in the initial ensemble.
- **niter** (*int*) – The number of iterations of the algorithm.
- **Arguments** (*Keywords*) –
- **----------** –
- ****kwargs** – Keyword arguments passed to the "trajectory" method of the dynamics object.

Yields trajectory, weight (*numpy.ndarray, float*) – The generator yields (trajectory, weight) pairs which allows to compute easily the probability associated to each sampled trajectory.

Notes

This method yields first the trajectories in the initial ensemble, then the resampled trajectories as the algorithm is iterated.

run_level(*ntraj*, *target_lev*, ***kwargs*)
Generate trajectories with the AMS algorithm.

Parameters

- **ntraj** (*int*) – The number of trajectories in the initial ensemble.
- **target_lev** (*float*) – The target level.
- **Arguments** (*Keywords*) –
- **------------------** –
- ****kwargs** – Keyword arguments passed to the "trajectory" method of the dynamics object.

Yields trajectory, weight (*numpy.ndarray, float*) – The generator yields (trajectory, weight) pairs which allows to compute easily the probability associated to each sampled trajectory.

Notes

This method yields first the killed trajectories as the algorithm is iterated, then the trajectories in the final ensemble.

class stochrare.rare.ams.**TAMS**(*model*, *scorefun*, *duration*, ***kwargs*)

Bases: *stochrare.rare.ams.AMS*

Implement the TAMS algorithm[5].

Parameters

- **dynamics** (*stochrare.dynamics.StochModel object (or a subclass of it)*) – The dynamical model; so far we are restricted to SDEs of the form $dX_t = F(X_t, t) + \sqrt{2D}dW_t$ We only use the trajectory method of the dynamics object.
- **score** (*function with two arguments.*) – The score function $\xi(t, x)$.
- **duration** (*float*) – The fixed duration for each trajectory.

Notes

This implementation keeps all the information in memory: this should not be suitable for complex dynamics. Similarly, the algorithm is not parallelized, even if the dynamics itself may be.

References

resample(*time*, *pos*, *told*, *xold*, ***kwargs*)

Resample a killed trajectory after a given time.

Parameters

- **time** (*float*) – The time from which to resample.
- **pos** (*float*) – The position from which to resample.
- **told** (*numpy.ndarray*) – The sample times from the killed trajectory.
- **xold** (*numpy.ndarray*) – The killed trajectory.

Keyword Arguments ****kwargs** – Keyword arguments, forwarded to *simul_trajectory()*.

[5] T. Lestang, F. Ragone, C.-E. Brehier, C. Herbert and F. Bouchet, J. Stat. Mech. (2018)

Returns **tnew, xnew** – The resampled trajectory.

Return type numpy.ndarray, numpy.ndarray

simul_trajectory(*x0*, *t0*, ***kwargs*)

Simulate a trajectory with given initial conditions for a fixed duration.

Parameters

- **x0** (*float*) – Initial position.
- **t0** (*float*) – Initial time.

Keyword Arguments

- **T** (*float*) – The duration.
- **dt** (*float*) – The time step.

Returns **t, x** – The simulated trajectory.

Return type numpy.ndarray, numpy.ndarray

average(*ntraj*, *niter*, *observable*, ***kwargs*)

Estimate the average of an observable using AMS sampling.

Parameters

- **ntraj** (*int*) – The number of initial trajectories in the ensemble.
- **niter** (*int*) – The number of iterations of the AMS algorithm.
- **observable** (*function with two arguments*) – A function of the form O(t, x), where t and x are numpy arrays. It should itself return a numpy array. For instance, it could be a time-independent function of the type `lambda t, x: x**2` or a functional of the trajectory such as `lambda t, x: np.array([np.max(x**2)])` Note that in the latter case it is crucial to convert the scalar to an array.

Keyword Arguments

- **method** (*function*) – The method used to sample the trajectories with the AMS algorithm. It can be one of `run_iter()` (default) or `run_resamp()`.
- **condition** (*function*) – A predicate for conditional averaging. It should be of the form `pred((t,x))` in (True, False).

Returns **obs** – The expectation value of the observable.

Return type numpy.ndarray

returntimes(*ntraj*, *niter*, ***kwargs*)

Estimate the return time of an observable using AMS sampling.

Parameters

- **ntraj** (*int*) – The number of initial trajectories in the ensemble.
- **niter** (*int*) – The number of iterations of the AMS algorithm.

Keyword Arguments

- **method** (*function*) – The method used to sample the trajectories with the AMS algorithm. It can be one of `run_iter()` (default) or `run_resamp()`.
- **observable** (*function*) – The time-dependent observable O(t, x). The default is the score function.

Returns a, r(a) – The amplitude and associated return time using the generalized block-maximum method.

Return type numpy.ndarray, numpy.ndarray

Classes

AMS(model, scorefun[, initcond])	Original version of the *Adaptive Multilevel Splitting* Algorithm.
TAMS(model, scorefun, duration, **kwargs)	Implement the TAMS algorithm[5].

3.5 stochrare.io

3.5.1 Input/Output

Stochpy generates two kinds of outputs: - data (results of computations that we wish to store on disk for future use) - plots (results fo computations that we wish to represent in a graphic manner)

Hence the `io` module is organized into two submodules, *stochrare.io.data* and *stochrare.io.plot*.

data	
plot	Plotting routines

stochrare.io.data

Classes

`Database`(path)	A simple generic database class I use to cache the results from computations on simple stochastic systems to avoid making the same computations over and over again.
`FirstPassageData`(model[, path])	Specializing the above class for our specific use case: computing and storing first passage time realizations for stochastic models.
`FirstPassageFP`([path])	Specializing the above class for storing results from numerical integration of the adjoint FP equation.
`TrajectoryData`(model[, path])	Specialized database for storing realizations of reacting trajectories, indexed by their first-passage time.

stochrare.io.plot

Plotting routines

This module contains several functions for making quick plots.

`stochrare.io.plot.`**`trajectory_plot1d`**(**args*, ***kwargs*)

Plot 1D trajectories.

Parameters **`*args`** (*variable length argument list*) – trajs: tuple (t, x) or (t, x, kwargs_dict)

Keyword Arguments

- **fig** (*matplotlig.figure.Figure*) – Figure object to use for the plot. Create one if not provided.
- **ax** (*matplotlig.axes.Axes*) – Axes object to use for the plot. Create one if not provided.
- ****kwargs** – Other keyword arguments forwarded to matplotlib.pyplot.axes.

Returns **fig, ax** – The figure.

Return type matplotlib.figure.Figure, matplotlib.axes.Axes

stochrare.io.plot.**pdf_plot1d**(**args*, *legend=True*, ***kwargs*)

Plot 1D PDFs.

Parameters ***args** (*variable length argument list*) – PDFs: tuple (X, P) or (X, P, kwargs_dict)

Keyword Arguments

- **potential** (*ndarray 2-tuple*) – X, V where V is the value of the potential at the sample points X. Default (None, None).
- **fig** (*matplotlig.figure.Figure*) – Figure object to use for the plot. Create one if not provided.
- **ax** (*matplotlig.axes.Axes*) – Axes object to use for the plot. Create one if not provided.
- **legend** (*bool*) – Add legend (default True).
- ****kwargs** – Other keyword arguments forwarded to matplotlib.pyplot.axes.

Returns **fig, ax** – The figure.

Return type matplotlib.figure.Figure, matplotlib.axes.Axes

stochrare.io.plot.**returntime_plot**(**args*)

Make return time plot: amplitude a as a function of the return time r(a)

Parameters ***args** (*variable length argument list*) – Pairs of the form (a, r(a))

Returns **fig, ax** – The figure.

Return type matplotlib.figure.Figure, matplotlib.axes.Axes

Functions

ensemble_plot1d_box(*args, **kwargs)	Plot an ensemble of 1D trajectories in a 3D box.
pdf_plot1d(*args[, legend])	Plot 1D PDFs.
returntime_plot(*args)	Make return time plot: amplitude a as a function of the return time r(a)
trajectory_plot1d(*args, **kwargs)	Plot 1D trajectories.

3.6 stochrare.utils

3.6.1 Utilities

This module contains various tools which are not really intended to be used by external code, but should be used by various other modules in the package. This includes decorators, but is not restricted to it.

stochrare.utils.**pseudorand**(*fun*)

Decorator for methods of random objects. If the object's *__deterministic__* attribute is set to True, the random number generator will be seeded with a fixed value (here, 100, chosen arbitrarily) before calling the method. Hence the method will behave in a deterministic way, only if the instance was initialized with the *__deterministic__* flag. This is essentially useful for testing.

The decorator will raise an error if the object does not have a *__deterministic__* attribute.

Functions

pseudorand(fun)	Decorator for methods of random objects.

Python Module Index

S

Index