SdePy Package Documentation

Release 1.2.1-dev0

MC

Aug 23, 2021

Contents

I Getting Started **1**

1 SdePy **3**
- 1.1 Start here . . . 3
- 1.2 License . . . 3

2 Quick Guide **5**
- 2.1 Install and import . . . 5
- 2.2 How to state an SDE . . . 5
- 2.3 How to integrate an SDE . . . 5
- 2.4 How to handle the integration output . . . 9
- 2.5 Example - Stochastic Runge-Kutta . . . 12
- 2.6 Example - Fokker-Planck Equation . . . 13
- 2.7 Example - Basket Lookback Option . . . 14

II Release Notes **17**

3 SdePy 1.2.0 **21**

4 SdePy 1.1.0 **23**

III API Documentation **25**

5 Overview **27**

6 Infrastructure **29**
- 6.1 sdepy.process . . . 29
- 6.2 sdepy.piecewise . . . 37
- 6.3 sdepy.montecarlo . . . 37

7 Stochasticity Sources **43**
- 7.1 sdepy.source . . . 44
- 7.2 sdepy.wiener_source . . . 45
- 7.3 sdepy.poisson_source . . . 46
- 7.4 sdepy.cpoisson_source . . . 47
- 7.5 sdepy.odd_wiener_source . . . 49
- 7.6 sdepy.even_poisson_source . . . 49
- 7.7 sdepy.even_cpoisson_source . . . 50
- 7.8 sdepy.true_source . . . 50
- 7.9 sdepy.true_wiener_source . . . 52
- 7.10 sdepy.true_poisson_source . . . 53

7.11 sdepy.true_cpoisson_source . . . 54
7.12 sdepy.norm_rv . . . 55
7.13 sdepy.uniform_rv . . . 55
7.14 sdepy.exp_rv . . . 55
7.15 sdepy.double_exp_rv . . . 55
7.16 sdepy.rvmap . . . 56

8 SDE Integration Framework **57**
8.1 sdepy.paths_generator . . . 57
8.2 sdepy.integrator . . . 62
8.3 sdepy.SDE . . . 64
8.4 sdepy.SDEs . . . 71
8.5 sdepy.integrate . . . 73

9 Stochastic Processes **75**
9.1 sdepy.wiener_process . . . 76
9.2 sdepy.lognorm_process . . . 76
9.3 sdepy.ornstein_uhlenbeck_process . . . 77
9.4 sdepy.hull_white_process . . . 78
9.5 sdepy.hull_white_1factor_process . . . 79
9.6 sdepy.cox_ingersoll_ross_process . . . 80
9.7 sdepy.full_heston_process . . . 80
9.8 sdepy.heston_process . . . 82
9.9 sdepy.jumpdiff_process . . . 82
9.10 sdepy.merton_jumpdiff_process . . . 83
9.11 sdepy.kou_jumpdiff_process . . . 84
9.12 sdepy.wiener_SDE . . . 84
9.13 sdepy.lognorm_SDE . . . 85
9.14 sdepy.ornstein_uhlenbeck_SDE . . . 85
9.15 sdepy.hull_white_SDE . . . 85
9.16 sdepy.cox_ingersoll_ross_SDE . . . 86
9.17 sdepy.full_heston_SDE . . . 86
9.18 sdepy.heston_SDE . . . 86
9.19 sdepy.jumpdiff_SDE . . . 87
9.20 sdepy.merton_jumpdiff_SDE . . . 87
9.21 sdepy.kou_jumpdiff_SDE . . . 88

10 Analytical Results **89**
10.1 sdepy.wiener_mean . . . 91
10.2 sdepy.wiener_var . . . 92
10.3 sdepy.wiener_std . . . 92
10.4 sdepy.wiener_pdf . . . 93
10.5 sdepy.wiener_cdf . . . 93
10.6 sdepy.wiener_chf . . . 93
10.7 sdepy.lognorm_mean . . . 94
10.8 sdepy.lognorm_var . . . 94
10.9 sdepy.lognorm_std . . . 94
10.10 sdepy.lognorm_pdf . . . 95
10.11 sdepy.lognorm_cdf . . . 95
10.12 sdepy.lognorm_log_chf . . . 95
10.13 sdepy.oruh_mean . . . 96
10.14 sdepy.oruh_var . . . 96
10.15 sdepy.oruh_std . . . 97
10.16 sdepy.oruh_pdf . . . 97
10.17 sdepy.oruh_cdf . . . 97
10.18 sdepy.hw2f_mean . . . 98
10.19 sdepy.hw2f_var . . . 98
10.20 sdepy.hw2f_std . . . 98
10.21 sdepy.hw2f_pdf . . . 99

10.22 sdepy.hw2f_cdf . . . 99
10.23 sdepy.cir_mean . . . 99
10.24 sdepy.cir_var . . . 100
10.25 sdepy.cir_std . . . 100
10.26 sdepy.cir_pdf . . . 101
10.27 sdepy.heston_log_mean . . . 101
10.28 sdepy.heston_log_var . . . 101
10.29 sdepy.heston_log_std . . . 102
10.30 sdepy.heston_log_pdf . . . 102
10.31 sdepy.heston_log_chf . . . 103
10.32 sdepy.mjd_log_pdf . . . 103
10.33 sdepy.mjd_log_chf . . . 103
10.34 sdepy.kou_mean . . . 104
10.35 sdepy.kou_log_pdf . . . 104
10.36 sdepy.kou_log_chf . . . 105
10.37 sdepy.bsd1d2 . . . 105
10.38 sdepy.bscall . . . 105
10.39 sdepy.bscall_delta . . . 106
10.40 sdepy.bsput . . . 107
10.41 sdepy.bsput_delta . . . 107

11 Shortcuts **109**
11.1 sdepy.kfunc . . . 110
11.2 sdepy.iskfunc . . . 111

IV Testing **113**

12 sdepy.test **117**

Bibliography **119**

Python Module Index **121**

Index **123**

Part I

Getting Started

CHAPTER 1

SdePy

The SdePy package provides tools to state and numerically integrate Ito Stochastic Differential Equations (SDEs), including equations with time-dependent parameters, time-dependent correlations, and stochastic jumps, and to compute with, and extract statistics from, their realized paths.

Several preset processes are provided, including lognormal, Ornstein-Uhlenbeck, Hull-White n-factor, Heston, and jump-diffusion processes.

Computations are fully vectorized across paths, via NumPy and SciPy, making live sessions with 100000 paths reasonably fluent on single cpu hardware.

This package came out of practical need, so expect a flexible tool that gets real-life things done. On the other hand, not every part of it is clean and polished, so expect rough edges, and the occasional bug (please report!).

Developers are committed to the stability of the public API, here again out of practical need to safeguard dependencies.

1.1 Start here

- Installation: `pip install sdepy`
- Quick Guide (as notebook)
- Documentation (as pdf)
- Source
- License
- Bug Reports

1.2 License

CHAPTER 2

Quick Guide

2.1 Install and import

Install using `pip install sdepy`, or copy the package source code in a directory in your Python path.

Import as

```
>>> import sdepy
>>> import numpy as np
>>> import matplotlib.pyplot as plt  # optional, if plots are needed
>>> plt.rcParams['figure.figsize'] = (11., 5.5)
>>> plt.rcParams['lines.linewidth'] = 1.
```

2.2 How to state an SDE

Here follows a bare-bone definition of a Stochastic Differential Equation (SDE), in this case a Ornstein-Uhlenbeck process:

```
>>> @sdepy.integrate
... def my_process(t, x, theta=1., k=1., sigma=1.):
...     return {'dt': k*(theta - x), 'dw': sigma}
```

This represents the SDE `dX = k*(theta - X)*dt + sigma*dW(t)`, where `theta`, `k` and `sigma` are parameters and `dW(t)` are Wiener process increments. A further `'dn'` or `'dj'` entry in the returned dictionary would allow for Poisson or compound Poisson jumps.

A number of preset processes are provided, including lognormal processes, Hull-White n-factor processes, Heston processes, and jump-diffusion processes.

2.3 How to integrate an SDE

Now `my_process` is a class, a subclass of the cooperating `sdepy.SDE` and `sdepy.integrator` classes:

```
>>> issubclass(my_process, sdepy.integrator), issubclass(my_process, sdepy.SDE)
(True, True)
```

It is to be instantiated with a number of parameters, including the SDE parameters `theta`, `k` and `sigma`; its instances are callable, given a timeline they will integrate and return the process along it. Decorating `my_process` with `sdepy.kunfc` allows for more concise handling of parameters:

```
>>> myp = sdepy.kfunc(my_process)
```

It is best explained by examples, involving `my_process`, `myp` and

```
>>> coarse_timeline = (0., 0.25, 0.5, 0.75, 1.0)
>>> timeline = np.linspace(0., 1., 500)
```

1. **Scalar process** in 100000 paths, with default parameters, computed at 5 time points (`coarse_timeline`), using 100 steps in between:

   ```
   >>> x = my_process(x0=1, paths=100*1000,
   ...                steps=100)(coarse_timeline)
   >>> x.shape
   (5, 100000)
   ```

2. The same scalar process computed on a **fine-grained timeline** (`timeline`) and 1000 paths, using **one integration step for each point in the timeline** (no `steps` parameter):

   ```
   >>> x = my_process(x0=1, paths=1000,
   ...                steps=100)(timeline)
   >>> x.shape
   (500, 1000)
   ```

   A plot of a few paths may be used to **inspect the integration result**:

   ```
   >>> gr = plt.plot(timeline, x[:, :30])
   >>> plt.show()
   ```

2. **Vector process** with three components and **correlated Wiener increments** (same other parameters as above):

   ```
   >>> corr = ((1, .2, -.3), (.2, 1, .1), (-.3, .1, 1))
   >>> x = my_process(x0=1, vshape=3, corr=corr,
   ...                paths=1000)(timeline)
   >>> x.shape
   (500, 3, 1000)
   ```

3. Vector process as above, with 10000 paths and **time-dependent parameters and correlations**:

   ```
   >>> sigma = lambda t: 0.1 + t
   >>> theta = lambda t: 2-t
   >>> k = lambda t: 2/(t+1)
   >>> c02 = lambda t: -0.1*np.cos(3*t)
   >>> c12 = lambda t: 0.1*np.sign(0.5-t)
   >>> corr = lambda t: ((     1,    -.2, c02(t)),
   ...                   (   -.2,      1, c12(t)),
   ...                   (c02(t), c12(t),      1))
   >>> x = my_process(x0=1, vshape=3, corr=corr,
   ...                theta=theta, k=k, sigma=sigma, paths=10*1000)(timeline)
   >>> x.shape
   (500, 3, 10000)
   ```

   This plot illustrates the correlations among the components of `x` increments, as a function of time and as compared to `corr(t)`:

   ```
   >>> dx = np.diff(x, axis=0)
   >>> for i in range(3):
   ...     for j in range(i + 1, 3):
   ```

(continues on next page)

(continued from previous page)

```
...         gr = plt.plot(
...             timeline, corr(timeline)[i][j] + 0*timeline,
...             timeline[1:], [np.cov(z)[i, j]/(z[i].std()*z[j].std())
...                            for z in dx]
...             )
>>> plt.show()
```

4. A 1000 paths scalar process with **path-dependent initial conditions and parameters**, integrated **backwards** (`i0=-1`):

```
>>> x0, sigma = np.zeros(1000), np.zeros(1000)
>>> x0[::2], x0[1::2] = 0., 2.
>>> sigma[::2], sigma[1::2] = 0.5, 0.1
>>> x = my_process(x0=x0, sigma=sigma, paths=1000,
...                theta=1, k=-2,
...                i0=-1)(timeline)
>>> x.shape
(500, 1000)
```

When integrating backwards, the inital conditions are applied at the final point in the given timeline:

```
>>> assert (x[-1, :] == x0).all()
>>> gr = plt.plot(timeline, x[:, :30])
>>> gr = plt.plot(timeline, np.full_like(timeline, 1), 'k--')
>>> plt.show()
```

Note the negative value of `k`, with mean reversion towards `theta=1` occurring backwards in time.

5. A scalar process computed on a **10 x 15 grid of parameters** `sigma` and `k` (note that the shape of the initial conditions and of each parameter should be broadcastable to the values of the process across paths, i.e. to shape `vshape + (paths,)`):

```
>>> sigma = np.linspace(0., 1., 10).reshape(10, 1, 1)
>>> k = np.linspace(1., 2., 15).reshape(1, 15, 1)
>>> x = my_process(x0=1, theta=2, k=k, sigma=sigma, vshape=(10, 15),
...                paths=10*1000)(coarse_timeline)
>>> x.shape
(5, 10, 15, 10000)
```

A plot of the final average process values against `k` illustrates a faster reversion to `theta=2` as `k` increases, as well as the independence of the process mean from `sigma`:

```
>>> for i in range(10):
...     gr = plt.plot(k[0, :, 0], x[-1, i, :, :].mean(axis=-1))
>>> lb = plt.xlabel('k'), plt.ylabel('x(t=2).mean()')
>>> plt.show()
```

In the example above, set `steps>=100` to go from inaccurate and fast, to meaningful and slow.

6. **Interactive modification** of process and integration parameters using the `sdepy.kfunc` decorator `myp = sdepy.kfunc(my_process)`.

   The `sdepy.kfunc` decorated version of `my_process` is a subclass of `sdepy.integrator` and `sdepy.SDE`, as `my_process` is, and fully replicates its functionality and interface:

```
>>> issubclass(myp, sdepy.integrator), issubclass(myp, sdepy.SDE)
(True, True)
```

In addition, and in contrast to `my_process`, `myp` instances accept either an integration timeline, or a modified value of some integration or SDE parameters, or both, as illustrated below:

```
>>> p = myp(x0=1, sigma=1, paths=1000)

>>> x = p(timeline)
>>> x1, x2 = p(timeline, sigma=0.5), p(timeline, sigma=1.5)
>>> q = p(paths=100, vshape=(3,), k=2)
>>> y = q(timeline, sigma=0.5)
```

`x` is the result of integrating `p` along `timeline` (no difference here from a `my_process` instance); `x1, x2` are obtained by integration along `timeline` by setting `sigma` to the given values, and keeping other parameters as stated when `p` was instantiated; `q` is another `myp` instance with updated default values for `paths`, `vshape` and `k`, and all else set as in `p`; and finally, `y` was obtained by integrating `q` along `timeline`, with its own parameters, save for `sigma` that was modified to `0.5`.

Moreover, for `sdepy.kfunc` classes, instantiation and computation may happen contextually:

```
>>> x = myp(timeline, x0=1, sigma=1, paths=1000)
```

is equivalent to:

```
>>> x = my_process(x0=1, sigma=1, paths=1000)(timeline)
```

`sdepy.kfunc`-decorated classes allow to **state some central values of parameters for a given problem**, and to explore the effects of variatons in some of them via a concise interface, that **keeps the modified parameters in focus** and all the rest in the background.

To **inspect the parameters stored** in a `sdepy.kfunc` instance, use the read-only `params` attribute:

```
>>> q.params
{
   'paths': 100,
   'vshape': (3,),
   'x0': array(1),
   'sigma': array(1),
   'k': array(2),
   ...,
}
```

To test if an object is a kfunc, use `sdepy.iskfunc()`:

```
>>> sdepy.iskfunc(myp), sdepy.iskfunc(p), sdepy.iskfunc(my_process)
(True, True, False)
```

The examples that follow illustrate, among other things, the use of `myp` as a `sdepy.kfunc` class.

7. Processes generated using **integration results as stochasticity sources** (mind using consistent `vshape` and `paths`, and synchronizing timelines):

```
>>> my_dw = sdepy.integrate(lambda t, x: {'dw': 1})(vshape=1,␣
↪paths=1000)(timeline)
>>> p = myp(dw=my_dw, vshape=3, paths=1000,
...         x0=1, sigma=((1,), (2,), (3,)))
>>> x = p(timeline)
>>> x.shape
(500, 3, 1000)
```

Now, `x1, x2, x3 = = x[:, 0], x[:, 1], x[:, 2]` have different `sigma`, but share the same `dw` increments, as can be seen plotting a path:

```
>>> k = 0  # path to be plotted
>>> gr = plt.plot(timeline, x[:, :, k])
>>> plt.show()
```

If more integrations steps are needed between points in the output timeline, use `steps` to keep the integration timeline consistent with the one of `my_dw`:

```
>>> x = p(coarse_timeline, steps=timeline)
>>> x.shape
(5, 3, 1000)
```

8. Using **stochasticity sources with memory** (mind using consistent `vshape` and `paths`):

```
>>> my_dw = sdepy.true_wiener_source(paths=1000)
>>> p = myp(x0=1, theta=1, k=1, sigma=1, dw=my_dw, paths=1000)
```

`my_dw`, as a `sdepy.true_wiener_source` instance has memory of, and generates new Wiener process increments consistent with, its formerly realized values. As a consequence, processes defined invoking `p` share the same underlying Wiener process increments:

```
>>> t1 = np.linspace(0., 1.,  30)
>>> t2 = np.linspace(0., 1., 100)
>>> t3 = t = np.linspace(0., 1., 300)
>>> x1, x2, x3 = p(t1), p(t2), p(t3)
>>> y1, y2, y3 = p(t, theta=1.5), p(t, theta=1.75), p(t, theta=2)
```

`x1, x2, x3` illustrate SDE integration convergence as time steps become smaller, and `y1, y2, y3` illustrate how `theta` affects paths, all else being equal:

```
>>> i = 0 # path to be plotted
>>> gr = plt.plot(t, x1(t)[:, i], t, x2(t)[:, i], t, x3(t)[:, i])
>>> plt.show()
>>> gr = plt.plot(t, y1[:, i], t, y2[:, i], t, y3[:, i])
>>> plt.show()
```

2.4 How to handle the integration output

SDE integrators return instances of `sdepy.process`, a subclass of `np.ndarray` with a timeline stored in the `t` attribute (note the shape of `x`, repeatedly used in the examples below):

```
>>> coarse_timeline = (0., 0.25, 0.5, 0.75, 1.0)
>>> timeline = np.linspace(0., 1., 101)
>>> x = my_process(x0=1, vshape=3, paths=1000)(timeline)
>>> x.shape
(101, 3, 1000)
```

`x` is a `sdepy.process` instance:

```
>>> type(x)
<class 'sdepy.infrastructure.process'>
```

and is based on the given timeline:

```
>>> np.isclose(timeline, x.t).all()
True
```

Whenever possible, a process will store references, not copies, of timeline and values. In fact:

```
>>> timeline is x.t
True
```

The first axis is reserved for the timeline, the last for paths, and axes in the middle match the shape of process values:

```
>>> x.shape == x.t.shape + x.vshape + (x.paths,)
True
```

Calling processes interpolates in time:

```
>>> y = x(coarse_timeline)
>>> y.shape
(5, 3, 1000)
```

The result is always an array, not a process:

```
>>> type(y)
<class 'numpy.ndarray'>
```

Indexing works as usual, and returns NumPy arrays:

```
>>> type(x[0])
<class 'numpy.ndarray'>
```

All array methods are unchanged (no overriding), and return NumPy arrays as well:

```
>>> type(x.mean(axis=0))
<class 'numpy.ndarray'>
```

You can slice processes along time, values and paths with special indexing.

- Time indexing:

  ```
  >>> y = x['t', ::2]
  >>> y.shape
  (51, 3, 1000)
  ```

- Values indexing:

  ```
  >>> y = x['v', 0]
  >>> y.shape
  (101, 1000)
  ```

- Paths indexing:

  ```
  >>> y = x['p', :10]
  >>> y.shape
  (101, 3, 10)
  ```

The output of a special indexing operation is a process:

```
>>> isinstance(y, sdepy.process)
True
```

Smart indexing is allowed. To select paths that cross `x=0` at some point and for some component, use:

```
>>> i_negative = x.min(axis=(0, 1)) < 0
>>> y = x['p', i_negative]
>>> y.shape == (101, 3, i_negative.sum())
True
```

You can do algebra with processes that either share the same timeline, or are constant (a process with a one-point timeline is assumed to be constant), and either have the same number of paths, or are deterministic (with one path):

```
>>> x_const = x['t', 0]  # a constant process
>>> x_one_path = x['p', 0]  # a process with one path
```

(continues on next page)

(continued from previous page)

```
>>> y = np.exp(x) - x_const
>>> z = np.maximum(x, x_one_path)

>>> isinstance(y, sdepy.process), isinstance(z, sdepy.process)
(True, True)
```

When integrating SDEs, the SDE parameters and/or stochasticity sources accept processes as valid values (mind using deterministic processes, or synchronizing the number of paths, and make sure that the shape of values do broadcast together). To use a realization of `my_process` as the volatility of a 3-component lognormal process, do as follows:

```
>>> stochastic_vol = my_process(x0=1, paths=10*1000)(timeline)
>>> stochastic_vol_x = sdepy.lognorm_process(x0=1, vshape=3, paths=10*1000,
...     mu=0, sigma=stochastic_vol)(timeline)
```

Processes have specialized methods, and may be analyzed, and their statistics cumulated across multiple runs, using the `sdepy.montecarlo` class. Some examples follow:

1. Cumulative probability distribution function at t=0.5 of the process values of `x` across paths:

   ```
   >>> cdf = x.cdf(0.5, x=np.linspace(-2, 2, 100))  # an array
   ```

2. Characteristic function at t=0.5 of the same distribution:

   ```
   >>> chf = x.chf(0.5, u=np.linspace(-2, 2, 100))  # an array
   ```

3. Standard deviation across paths:

   ```
   >>> std = x.pstd()  # a one-path process
   >>> std.shape
   (101, 3, 1)
   ```

4. Maximum value reached along the timeline:

   ```
   >>> xmax = x.tmax()  # a constant process
   >>> xmax.shape
   (1, 3, 1000)
   ```

5. A linearly interpolated, or Gaussian kernel estimate (default) of the probability distribution function (pdf) and its cumulated values (cdf) across paths, at a given time point, may be obtained using the `montecarlo` class:

   ```
   >>> y = x(1)[0]  # 0-th component of x at time t=1
   >>> a = sdepy.montecarlo(y, bins=30)
   >>> ygrid = np.linspace(y.min(), y.max(), 200)
   >>> gr = plt.plot(ygrid, a.pdf(ygrid), ygrid, a.cdf(ygrid))
   >>> gr = plt.plot(ygrid, a.pdf(ygrid, method='interp', kind='nearest'))
   >>> plt.show()  # doctest: +SKIP
   ```

6. A `sdepy.montecarlo` instance can be used to cumulate the results of multiple simulations, across multiple components of process values:

   ```
   >>> p = my_process(x0=1, vshape=3, paths=10*1000)
   >>> a = sdepy.montecarlo(bins=100)  # empty montecarlo instance
   >>> for _ in range(10):
   ...     x = p(timeline)  # run simulation
   ...     a.update(x(1))  # cumulate x values at t=1
   >>> a.paths
   100000
   >>> gr = plt.plot(ygrid, a[0].pdf(ygrid), ygrid, a[0].cdf(ygrid))
   ```

(continues on next page)

(continued from previous page)

```
>>> gr = plt.plot(ygrid, a[0].pdf(ygrid, method='interp', kind='nearest'))
>>> plt.show()
```

2.5 Example - Stochastic Runge-Kutta

Minimal implementation of a basic stochastic Runge-Kutta integration scheme, as a subclass of `sdepy.integrator` (the `A` and `dZ` methods below are the standardized way in which equations are exposed to integrators):

```
>>> from numpy import sqrt
>>> class my_integrator(sdepy.integrator):
...     def next(self):
...         t, new_t = self.itervars['sw']
...         x, new_x = self.itervars['xw']
...         dt = new_t - t
...         A, dZ = self.A(t, x), self.dZ(t, dt)
...         a, b, dw = A['dt'], A['dw'], dZ['dw']
...         b1 = self.A(t, x + a*dt + b*sqrt(dt))['dw']
...         new_x[...] = x + a*dt + b*dw + (b1 - b)/2 * (dw**2 - dt)/sqrt(dt)
```

SDE of a lognormal process, as a subclass of `sdepy.SDE`, and classes that integrate it with the default integration method (`euler`) and via `my_integrator` (`rk`):

```
>>> class my_SDE(sdepy.SDE):
...     def sde(self, t, x):
...         return {'dt': 0, 'dw': x}

>>> class euler(my_SDE, sdepy.integrator):
...     pass

>>> class rk(my_SDE, my_integrator):
...     pass
```

Comparison of integration errors, as the integration from `t=0` to `t=1` is carried out with an increasing number of steps, against the integration result of `sdepy.lognorm_process`, which returns an exact result irrespective of the number and size of the integration steps (this happens since, by implementation, it integrates the linear SDE for `log(x)`):

```
>>> args = dict(dw=sdepy.true_wiener_source(paths=100),
...             paths=100, x0=10)
>>> timeline = (0, 1)
>>> steps = np.array((2, 3, 5, 10, 20, 30, 50, 100,
...                   200, 300, 500, 1000, 2000, 3000))

>>> # exact integration results at t=1
>>> exact = sdepy.lognorm_process(mu=0, sigma=1, **args)(timeline)[-1].mean()

>>> # errors of approximate integration results at t=1
>>> errors = np.abs(np.array([
...     [euler(**args, steps=s)(timeline)[-1].mean()/exact - 1,
...      rk(**args, steps=s)(timeline)[-1].mean()/exact - 1]
...     for s in steps]))

>>> # plots
>>> ax = plt.axes(label=0); ax.set_xscale('log'); ax.set_yscale('log')
>>> gr = ax.plot(steps, errors)
>>> plt.show()
>>> print(f'euler error: {errors[-1,0]:.2e}\n'
```

(continues on next page)

(continued from previous page)

```
...       f'   rk error: {errors[-1,1]:.2e}')
euler error: 1.70e-03
   rk error: 8.80e-06
```

2.6 Example - Fokker-Planck Equation

Monte Carlo integration of partial differential equations, illustrated in the simplest example of the heat equation `diff(u, t) - k*diff(u, x, 2) == 0`, for the function `u(x, t)`, i.e. the Fokker-Planck equation for the SDE `dX(t) = sqrt(2*k)*dW(t)`. Initial conditions at `t=t0`, two examples:

1. `u(x, t0) = 1` for `lb < x < hb` and `0` otherwise,
2. `u(x, t0) = sin(x)`.

Setup:

```
>>> from numpy import exp, sin
>>> from scipy.special import erf
>>> from scipy.integrate import quad

>>> k = .5
>>> x0, x1 = 0, 10;
>>> t0, t1 = 0, 1
>>> lb, hb = 4, 6
```

Exact green function and solutions for initial conditions 1. and 2., to be checked against results:

```
>>> def green_exact(y, s, x, t):
...     return exp(-(x - y)**2/(4*k*(t - s)))/sqrt(4*np.pi*k*(t - s))

>>> def u1_exact(x, t):
...     return (erf((x - lb)/2/sqrt(k*(t - t0))) - erf((x - hb)/2/sqrt(k*(t -␣
↪t0))))/2

>>> def u2_exact(x, t):
...     return exp(-k*(t - t0))*sin(x)
```

Realization of the needed stochastic process, by backward integration from a grid of final values of `x` at `t=t1`, using the preset `wiener_process` class (the `steps` keyword is added as a reminder of the setup needed for less-than-trivial equations, it does not actually make a difference here):

```
>>> xgrid = np.linspace(x0, x1, 51)
>>> tgrid = np.linspace(t0, t1, 5)
>>> xp = sdepy.wiener_process(
...     paths=10000, steps=100,
...     sigma=sqrt(2*k),
...     vshape=xgrid.shape, x0=xgrid[..., np.newaxis],
...     i0=-1,
...     )(timeline=tgrid)
```

Computation of the green function and of the solutions `u(x, t1)` via Monte Carlo integration (note the liberal use of `scipy.integrate.quad` below, enabled by the smoothness of the Gaussian kernel estimate `a[i, j].pdf`):

```
>>> a = sdepy.montecarlo(xp, bins=100)

>>> def green(y, i, j):
...     """green function from (y=y, s=tgrid[i]) to (x=xgrid[j], t=t1)"""
...     return a[i, j].pdf(y)
```

(continues on next page)

(continued from previous page)

```
>>> u1, u2 = np.empty(51), np.empty(51)
>>> for j in range(51):
...     u1[j] = quad(lambda y: green(y, 0, j), lb, hb)[0]
...     u2[j] = quad(lambda y: sin(y)*green(y, 0, j), -np.inf, np.inf)[0]
```

Comparison against exact values:

```
>>> y = np.linspace(x0, x1, 500)
>>> for i, j in ((1, 20), (2, 30), (3, 40)):
...     gr = plt.plot(y, green(y, i, j),
...                   y, green_exact(y, tgrid[i], xgrid[j], t1), ':')
>>> plt.show()

>>> gr = plt.plot(xgrid, u1, y, u1_exact(y, t1), ':')
>>> gr = plt.plot(xgrid, u2, y, u2_exact(y, t1), ':')
>>> plt.show()

>>> print(f'u1 error: {np.abs(u1 - u1_exact(xgrid, t1)).mean():.2e}\n'
...       f'u2 error: {np.abs(u2 - u2_exact(xgrid, t1)).mean():.2e}'
... )
u1 error: 2.49e-03
u2 error: 5.51e-03
```

2.7 Example - Basket Lookback Option

Take a basket of 4 financial securities, with risk-neutral probabilities following lognormal processes in the Black-Scholes framework. Correlations, dividend yields and term structure of volatility (will be linearly interpolated) are given below:

```
>>> corr = [
...     [1,    0.50, 0.37, 0.35],
...     [0.50,    1, 0.47, 0.46],
...     [0.37, 0.47,    1, 0.19],
...     [0.35, 0.46,  0.19,   1]]

>>> dividend_yield = sdepy.process(c=(0.20, 4.40, 0., 4.80))/100
>>> riskfree = 0  # to keep it simple

>>> vol_timepoints = (0.1, 0.2, 0.5, 1, 2, 3)
>>> vol = np.array([
...     [0.40, 0.38, 0.30, 0.28, 0.27, 0.27],
...     [0.31, 0.29, 0.22, 0.16, 0.18, 0.21],
...     [0.24, 0.22, 0.19, 0.19, 0.21, 0.22],
...     [0.35, 0.31, 0.21, 0.18, 0.19, 0.19]])
>>> sigma = sdepy.process(t=vol_timepoints, v=vol.T)
>>> sigma.shape
(6, 4, 1)
```

The prices of the securities at the end of each quarter for the next 2 years, simulated across 50000 independent paths and their antithetics (`sdepy.odd_wiener_source` is used), are:

```
>>> maturity = 2
>>> timeline = np.linspace(0, maturity, 4*maturity + 1)
>>> p = sdepy.lognorm_process(
...     x0=100, corr=corr, dw=sdepy.odd_wiener_source,
...     mu=(riskfree - dividend_yield),
...     sigma=sigma,
```

(continues on next page)

(continued from previous page)

```
...     vshape=4, paths=100*1000, steps=maturity*250)
>>> x = p(timeline)
>>> x.shape
(9, 4, 100000)
```

A call option knocks in if any of the securities reaches a price below 80 at any quarter (starting from 100), and pays the lookback maximum attained by the basket (equally weighted), minus 105, if positive. Its price is:

```
>>> x_worst = x.min(axis=1)
>>> x_basket = x.mean(axis=1)
>>> down_and_in_paths = (x_worst.min(axis=0) < 80)
>>> lookback_x_basket = x_basket.max(axis=0)
>>> payoff = np.maximum(0, lookback_x_basket - 105)
>>> payoff[np.logical_not(down_and_in_paths)] = 0
>>> a = sdepy.montecarlo(payoff, use='even')
>>> print(a)
4.997 +/- 0.027
```

Part II

Release Notes

This section reports notable changes and additions in SdePy releases.

Not all releases are covered. For further information, and a complete list of releases, please refer to the SdePy repository on GitHub.

CHAPTER 3

SdePy 1.2.0

New in this release

SdePy was upgraded to the current NumPy pseudo-random numbers generation framework, based on the PCG-64 algorithm. The following additions were made:

- SDEs and stochasticity source classes now accept upon instantiation a new optional parameter `rng`, to locally set the random number generator used by each instance. Such `rng` is expected to expose the interface used by `numpy.random.Generator` and `numpy.random.RandomState` instances.
- In case no `rng` parameter is given, random number generation is delegated to a global `numpy.random.default_rng()` object, created upon import: absent any user intervention, unmodified pre-existing code will rely on PCG-64 random number generation after upgrading to current NumPy and SdePy releases.
- To access the stably reproducible NumPy legacy random generation, it is recommended to instantiate sources and SDEs with the `rng=numpy.random.RandomState(SEED)` parameter. As a result, each object is served the random numbers stream formerly obtained from the global NumPy random state after a `numpy.random.seed(SEED)` call.
- Compatibility with legacy NumPy versions was maintained, if now deprecated: in such case, SdePy silently falls back on the legacy global NumPy random state.

The present SdePy version safeguards functional compatibility with code written for previous versions, note however that, if such code is run with current NumPy and SdePy versions, it will **break pseudo-random numbers reproducibility**:

- `numpy.random.seed` calls no longer affect SdePy objects.
- To reproduce the default output of former SdePy versions, `numpy.random.seed(SEED)` statements may be replaced with the following assignment:

```
sdepy.infrastructure.default_rng = numpy.random.RandomState(SEED)
```

  Any other use of such global variable should rather be avoided, in favor of the `rng` keyword.

Improvements

- The testing suite was updated, and its interface `sdepy.test()` extended, for use beyond NumPy legacy random generation.
- GitHub Actions were updated accordingly, to perform CI tests using both legacy and current NumPy random generation.

Changes

Python 3.5 is no longer supported.

CHAPTER 4

SdePy 1.1.0

New in this release

- The `sdepy.process` class acquired new methods `vmin`, `vmax`, `vmean`, `vvar`, `vstd` to perform summary operations across values, for each time point and path.
- A piecewise constant process constructor was added as the `piecewise` function; it replaces, and improves upon, the private `_piecewise_constant_process` (undocumented, not part of the API), now deprecated.

Bug-fixes

- An incompatibility issue of the `sdepy.process` class with NumPy versions >= 1.16.0 was solved.
- A bug in the `out` parameter of process methods `pmin`, `pmax`, `tmin`, `tmax` was fixed. No backward incompatible changes were made against the stated API, note however that code relying on the `out` parameter of the mentioned process methods might need fixing as well.

Part III

API Documentation

CHAPTER 5

Overview

This package provides tools to state and numerically integrate Ito Stochastic Differential Equations (SDEs), including equations with time-dependent parameters, time-dependent correlations, and stochastic jumps, and to compute with, and extract statistics from, their realized paths.

Package contents:

1. A set of tools to ease computations with stochastic processes, as obtained from numerical integration of the corresponding SDE, is provided via the `process` and `montecarlo` classes (see *Infrastructure*):
    - The `process` class, a subclass of `numpy.ndarray` representing a sequence of values in time, realized in one or several paths. Algebraic manipulations and ufunc computations are supported for instances that share the same timeline, or are constant, and comply with numpy broadcasting rules. Interpolation along the timeline is supported via callability of `process` instances. Process-specific functionalities, such as averaging and indexing along time or across paths, are delegated to process-specific methods, attributes and properties (no overriding of `numpy.ndarray` operations).
    - The `montecarlo` class, as an aid to cumulate the results of several Monte Carlo simulations of a given stochastic variable, and to extract summary estimates for its probability distribution function and statistics.
2. Numerical realizations of the differentials commonly found as stochasticity sources in SDEs, are provided via the `source` class and its subclasses, with or without memory of formerly invoked realizations (see *Stochasticity Sources*).
3. A general framework for stochastic step by step simulations, and for numerical SDE integration, is provided via the `paths_generator` class, and its cooperating subclasses `integrator`, `SDE` and `SDEs` (see *SDE Integration Framework*). The full API allows for extensive customization of preprocessing, post-processing, stochasticity sources instantiation and handling, integration algorithms etc. The `integrate` decorator provides a simple and concise interface to handle standard use cases, via Euler-Maruyama integration.
4. Several preset stochastic processes are provided, including lognormal, Ornstein-Uhlenbeck, Hull-White n-factor, Heston, and jump-diffusion processes (see *Stochastic Processes*). Each process consists of a process generator class, a subclass of `integrator` and `SDE`, named with a `_process` suffix, and a definition of the underlying SDE, a subclass of `SDE` or `SDEs`, named with a `_SDE` suffix.
5. Several analytical results relating to the preset stochastic processes are made available, as a general reference and for testing purposes (see *Analytical Results*). They are limited to the case of constant process parameters, and with some further limitations on the parameters' domains. Function arguments are consistent with those of the corresponding processes. Suffixes `_pdf`, `_cdf` and `_chf` stand respectively for

probability distribution function, cumulative probability distribution function, and characteristic function. Black-Scholes formulae for the valuation of call and put options have been included (with prefix `bs`).

6. As an aid to interactive and notebook sessions, shortcuts are provided for stochasticity sources and preset processes (see *Shortcuts*). Shortcuts have been wrapped as "kfuncs", objects with managed keyword arguments that simplify interactive workflow when frequent parameters tuning operations are needed (see `kfunc` decorator documentation). Analytical results are wrapped as kfuncs as well.

For all sources and processes, values can take any shape, scalar or multidimensional. Correlated multivariate stochasticity sources are supported. Poisson jumps are supported, and may be compounded with any random variable supported by scipy.stats. Time-varying process parameters (correlations, intensity of Poisson processes, volatilities etc.) are allowed whenever applicable. `process` instances act as valid stochasticity source realizations (as does any callable object complying with a `source` protocol), and may be passed as a source specification when computing the realization of a given process.

Computations are fully vectorized across paths, providing an efficient infrastructure for simulating a large number of process realizations. Less so, for large number of time steps: integrating 100 time steps across one million paths takes seconds, one million time steps across 100 paths takes minutes.

CHAPTER 6

Infrastructure

process([t, x, v, c, dtype])	Array representation of a process (a subclass of numpy.ndarray).
piecewise([t, x, v, dtype, mode])	Return a process that interpolates to a piecewise constant function.
montecarlo([sample, axis, bins, range, use, ...])	Summary statistics of Monte Carlo simulations.

6.1 sdepy.process

class sdepy.**process**(*t=0.*, *, *x=None*, *v=None*, *c=None*, *dtype=None*)

Array representation of a process (a subclass of numpy.ndarray).

If `p` is a process instance, `p[i, ..., k]` is the value that the `k`-th path of the represented process takes at time `p.t[i]`. The first and last indexes of `p` are reserved for the timeline and paths respectively. A process should contain no less than 1 time point and 1 path. Zero or more middle indexes refer to the values that the process takes at each given time and path.

If `p` has `N` time points, `paths` is its number of paths and `vshape` is the shape of its values at any given time point and path, then `p.shape` is `(N,) + vshape + (paths,)`. `N, vshape, paths` are inferred at instantiation from the shape of `t` and `x, v` or `c` parameters.

Parameters

t [array-like] Timeline of the process, as a one dimensional array with shape `(N,)`, in increasing order. Defaults to 0.

x [array-like, optional] Values of the process along the timeline and across paths. Should broadcast to `(N,) + vshape + (paths,)`. The shapes of `t` and of the firs index of `x` must match. One and only one of `x`, `v`, `c` must be provided upon process creation, as a keyword argument.

v [array-like, optional] Values of a deterministic process along the timeline. Should broadcast to `(N,) + vshape`. The shapes of `t` and of the firs index of `v` must match.

c [array-like, optional] Value of a constant, single-path process, with shape `vshape`. Each time point of the resulting process contains a copy of `c`.

dtype [data-type, optional] Data-type of the values of the process. `x`, `v` or `c` will be converted to `dtype` if need be.

Notes

A reference and not a copy of `t, x, v, c` is stored if possible.

A process is a subclass of numpy.ndarray, where its values as an array are the process values along the timeline and across paths. All numpy.ndarray methods, attributes and properties are guaranteed to act upon such values, as would those of the parent class. Such no overriding commitment is intended to safeguard predictablity of array operations on process instances; process-specific functionalities are delegated to process-specific methods, attributes and properties.

A process with a single time point is assumed to be constant.

Processes have the `__array_priority__` attribute set to 1.0 by default. Ufuncs acting on a process, or on a process and an array, or on different processes sharing the same timeline, or on different processes one of which is constant, return a process with the timeline of the original process(es) passed as a reference. Ufuncs calls on different processes fail if non constant processes do not share the same timeline (interpolation should be handled explicitly), or in case broadcasting rules would result in mixing time, values and/or paths axes.

Let p be a process instance. Standard numpy indexing acts on the process values and returns numpy.ndarray instances: in fact, `p[i]` is equivalent to `p.x[i]`, i.e. the same as `p.view(numpy.ndarray)[i]`. Process-specific indexing is addressed via the following syntax, where `i` can be an integer, a multi-index or smart indexing reference consistent with the process shape:

- `p['t', i]` : timeline indexing, roughly equivalent to `process(t=p.t[i], x=p.x[i, ..., :])`
- `p['v', i]` : values indexing, roughly equivalent to `process(t=p.t, x=p.x[:, i, :])`
- `p['p', i]` : paths indexing, roughly equivalent to `process(t=p.t, x=p.x[:, ..., i])`

Attributes

x Process values, viewed as a numpy.ndarray.

paths Number of paths of the process (coincides with the size of the last dimension of the process).

vshape Shape of the values of the process.

tx Timeline of the process, reshaped to be broadcastable to the process values and paths across time.

dt Process timeline increments, as returned by numpy.diff.

dtx Process timeline increments, as returned by numpy.diff, reshaped to be broadcastable to the process values.

t [array] Stores the timeline of the process.

interp_kind [str] Stores the default interpolation kind, passed upon interpolation (`interp` and `__call__` methods) to `scipy.interpolate.interp1d` unless a specific kind is provided. Defaults to the class attribute of the same name, initialized to `'linear'`. Note that ufuncs and methods, when returning new processes, do *not* preserve the `interp_kind` attribute, which falls back on the class default and should be set explicitly again if needed.

Methods

interp(*[, kind])	Interpolation in time of the process values.
__call__(s[, ds, kind])	Interpolation in time of process values or increments.

Continued on next page

Table 2 – continued from previous page

`__getitem__`(key)	See documentation of the process class.
rebase(t, *[, kind])	Change the process timeline to t, using interpolation.
shapeas(vshape_or_process)	Reshape process values according to the given target shape.
pcopy(**args)	Copy timeline and values of the process (`args` are passed to `numpy.ndarray.copy`).
xcopy(**args)	Copy values of the process, share timeline (`args` are passed to `numpy.ndarray.copy`).
tcopy(**args)	Copy timeline of the process, share values.
pmin([out])	One path process exposing for each time point the minimum process value attained across paths.
pmax([out])	One path process exposing for each time point the maximum process value attained across paths.
psum([dtype, out])	One path process exposing for each time point the sum of process values across paths.
pmean([dtype, out])	One path process exposing for each time point the mean of process values across paths.
pvar([dtype, out, ddof])	One path process exposing for each time point the variance of process values across paths.
pstd([dtype, out, ddof])	One path process exposing for each time point the standard deviation of process values across paths.
vmin([out])	Process exposing for each time point and path the minimum of process values.
vmax([out])	Process exposing for each time point and path the maximum of process values.
vsum([dtype, out])	Process exposing for each time point and path the sum of process values.
vmean([dtype, out])	Process exposing for each time point and path the mean of process values.
vvar([dtype, out, ddof])	Process exposing for each time point and path the variance of process values.
vstd([dtype, out, ddof])	Process exposing for each time point and path the standard deviation of process values.
tmin([out])	Constant process exposing for each path the minimum process value attained along time.
tmax([out])	Constant process exposing for each path the maximum process value attained along time.
tsum([dtype, out])	Constant process exposing for each path the sum of process values along time.
tmean([dtype, out])	Constant process exposing for each path the mean of process values along time.
tvar([dtype, out, ddof])	Constant process exposing for each path the variance of process values along time.
tstd([dtype, out, ddof])	Constant process exposing for each path the standard deviation of process values along time.
tdiff([dt_exp, fwd])	Process increments along the timeline, optionally weighted by time increments.
tder()	Forward looking derivative of the given process, linearly interpolated between time points.
tint()	Integral of the given process, linearly interpolated between time points.
chf([t, u])	Characteristic function of the probability distribution of process values.

Continued on next page

Table 2 – continued from previous page

cdf([t, x])	Cumulative probability distribution function of process values.

6.1.1 sdepy.process.interp

process.**interp**(*, *kind=None*)

Interpolation in time of the process values.

Returns a callable `f`, as returned by `scipy.interpolate.interp1d`, such that `f(s)` approximates the value of the process at time point `s`. `f` refers to the process timeline and values, without storing copies. `s` may be of any shape.

Parameters

kind [string, optional] An interpolation kind as accepted by `scipy.interpolate.interp1d`. If None, defaults to the `interp_kind` attribute.

Returns

f [callable] `f`, as returned by scipy.interpolate.interp1d, such that `f(s)` approximates the value of the process at time point s. `f` refers to the process timeline and values, without storing copies.

`s` may be of any shape: if `p` is a process instance, `p.interp()(s).shape == s.shape + p.vshape + (p.paths,)`.

In case `p` has a single time point, interpolation is not handled via `scipy.interpolate.interp1d`; the process is assumed to be constant in time, and `f` is a function object behaving accordingly.

See also:

process.__call__

Notes

The process is extrapolated as constant outside the timeline boundaries.

If `p` is a process instance, `p.interp(s)` is an array, not a process. If an interpolated process is needed, it should be explicitly created using `q = process(s, x=p(s))`, or its shorthand `q = p.rebase(s)`.

6.1.2 sdepy.process.__call__

process.**__call__**(*s*, *ds=None*, *, *kind=None*)

Interpolation in time of process values or increments.

If `p` is a process instance and `f = p.interp(kind)`:

- `p(s)` returns `f(s)`,
- `p(s, ds)` returns `f(s + ds) - f(s)`.

See also:

process.interp

6.1.3 sdepy.process.rebase

`process.`**`rebase`**(*t*, *, *kind=None*)

Change the process timeline to t, using interpolation.

A new process is returned with timeline `t` and values set to the calling process values, interpolated at `t` using `process.interp` with the given interpolation kind.

If `t` is a scalar, a constant process is returned.

6.1.4 sdepy.process.shapeas

`process.`**`shapeas`**(*vshape_or_process*)

Reshape process values according to the given target shape.

Returns a process pointing to the same data as the calling process, adding new 1-dimensional axes, or removing existing 1-dimensional axes to the left of the first dimension of process values, as needed to make the returned process broadcastable to a process with values of the given shape.

To achieve broadcastability the unaffected dimensions, including the shape of the timeline and the number of paths, have to be compatible.

Raises

ValueError [if requested to remove a non 1-dimensional axis]

6.1.5 sdepy.process.pcopy

`process.`**`pcopy`**(***args*)

Copy timeline and values of the process (`args` are passed to `numpy.ndarray.copy`).

6.1.6 sdepy.process.xcopy

`process.`**`xcopy`**(***args*)

Copy values of the process, share timeline (`args` are passed to `numpy.ndarray.copy`).

6.1.7 sdepy.process.tcopy

`process.`**`tcopy`**(***args*)

Copy timeline of the process, share values. (`args` are passed to `numpy.ndarray.copy`).

6.1.8 sdepy.process.pmin

`process.`**`pmin`**(*out=None*)

One path process exposing for each time point the minimum process value attained across paths.

6.1.9 sdepy.process.pmax

`process.`**`pmax`**(*out=None*)

One path process exposing for each time point the maximum process value attained across paths.

6.1.10 sdepy.process.psum

`process.`**`psum`**(*dtype=None*, *out=None*)

One path process exposing for each time point the sum of process values across paths.

6.1.11 sdepy.process.pmean

process.**pmean**(*dtype=None*, *out=None*)
One path process exposing for each time point the mean of process values across paths.

6.1.12 sdepy.process.pvar

process.**pvar**(*dtype=None*, *out=None*, *ddof=0*)
One path process exposing for each time point the variance of process values across paths.

6.1.13 sdepy.process.pstd

process.**pstd**(*dtype=None*, *out=None*, *ddof=0*)
One path process exposing for each time point the standard deviation of process values across paths.

6.1.14 sdepy.process.vmin

process.**vmin**(*out=None*)
Process exposing for each time point and path the minimum of process values.

6.1.15 sdepy.process.vmax

process.**vmax**(*out=None*)
Process exposing for each time point and path the maximum of process values.

6.1.16 sdepy.process.vsum

process.**vsum**(*dtype=None*, *out=None*)
Process exposing for each time point and path the sum of process values.

6.1.17 sdepy.process.vmean

process.**vmean**(*dtype=None*, *out=None*)
Process exposing for each time point and path the mean of process values.

6.1.18 sdepy.process.vvar

process.**vvar**(*dtype=None*, *out=None*, *ddof=0*)
Process exposing for each time point and path the variance of process values.

6.1.19 sdepy.process.vstd

process.**vstd**(*dtype=None*, *out=None*, *ddof=0*)
Process exposing for each time point and path the standard deviation of process values.

6.1.20 sdepy.process.tmin

process.**tmin**(*out=None*)
Constant process exposing for each path the minimum process value attained along time.

6.1.21 sdepy.process.tmax

process.**tmax**(*out=None*)
Constant process exposing for each path the maximum process value attained along time.

6.1.22 sdepy.process.tsum

process.**tsum**(*dtype=None*, *out=None*)
Constant process exposing for each path the sum of process values along time.

6.1.23 sdepy.process.tmean

process.**tmean**(*dtype=None*, *out=None*)
Constant process exposing for each path the mean of process values along time.

6.1.24 sdepy.process.tvar

process.**tvar**(*dtype=None*, *out=None*, *ddof=0*)
Constant process exposing for each path the variance of process values along time.

6.1.25 sdepy.process.tstd

process.**tstd**(*dtype=None*, *out=None*, *ddof=0*)
Constant process exposing for each path the standard deviation of process values along time.

6.1.26 sdepy.process.tdiff

process.**tdiff**(*dt_exp=0*, *fwd=True*)
Process increments along the timeline, optionally weighted by time increments.

Parameters

dt_exp [int or float, optional] Exponent applied to time increment weights. If 0, returns process increments. If 1, approximates a time derivative. If 0.5, approximates realized volatility.

fwd [bool, optional] If True, the differences are forward-looking

Returns

q [process] If `p` is a process shaped `(N,) + p.vshape + (p.paths,)`, with timeline `t`, `p.tdiff(dt_exp, fwd)` returns a process `q`, shaped `(N-1,) + p.vshape + (p.paths,)` with values

```
q[i] = (p[i+1] - p[i])/(t[i+1] - t[i])**dt_exp
```

If `fwd` evaluates to `True`, `q[i]` is assigned to time point `t[i]` (`q` stores at `t[i]` the increments of `p` looking forwards) or to `t[i+1]` otherwise (increments looking backwards).

See also:

tder

tint

Notes

if `p` is a process instance realizing a solution of the SDE `dp(t) = sigma(t)*dw(t)` across several paths, then

```
p.tdiff(dt_exp=0.5).pstd()
```

is a 1-path process that estimates `sigma(t)`.

6.1.27 sdepy.process.tder

`process.`**`tder`**`()`

Forward looking derivative of the given process, linearly interpolated between time points.

Shorthand for `p.tdiff(dt_exp=1)`.

See also:

tdiff

tint

Notes

`p.tder().tint()` equals, within rounding errors, `p['t', :-1] - p['t', 0]`

6.1.28 sdepy.process.tint

`process.`**`tint`**`()`

Integral of the given process, linearly interpolated between time points.

See also:

tdiff

tder

Notes

`p.tin().tder()` equals, within rounding errors, `p['t', :-1]`

6.1.29 sdepy.process.chf

`process.`**`chf`**`(`*t=None*, *u=None*`)`

Characteristic function of the probability distribution of process values.

`p.chf(t, u)` estimates the characteristic function of interpolated process values `p(t)` at time(s) `t`. `p.chf(u)` is a shorthand for `p.chf(p.t, u)` (no interpolation).

Parameters

t [array-like, optional] Time points at which to compute the characteristic function. If omitted or `None`, the entire process timeline is used.

u [array-like, mandatory] Values at which to evaluate the characteristic function.

Returns

array Returns an array, with shape `t.shape + u.shape + vshape`, where `vshape` is the shape of values of the calling process `p`, containing the average across paths of `exp(1j*u*p(t))`.

6.1.30 sdepy.process.cdf

`process.`**`cdf`**(*t=None*, *x=None*)

Cumulative probability distribution function of process values.

`p.cdf(t, x)` estimates the cumulative probability distribution function of interpolated process values `p(t)` at time(s) `t`. `p.cdf(x)` is a shorthand for `p.cdf(p.t, x)` (no interpolation).

Parameters

t [array-like, optional] Time points along the process timeline. If omitted or `None`, the entire process timeline is used.

x [array-like, mandatory] Values at which to evaluate the cumulative probability distribution function.

Returns

array Returns an array, with shape `t.shape + x.shape + vshape`, where `vshape` is the shape of the values of the calling process `p`, containing the average across paths of `1 if p(t) <= x else 0`.

6.2 sdepy.piecewise

`sdepy.`**`piecewise`**(*t=0.0*, ***, *x=None*, *v=None*, *dtype=None*, *mode='mid'*)

Return a process that interpolates to a piecewise constant function.

Parameters

t [array-like] Reference timeline (see below).

x [array-like, optional] Values of the process along the timeline and across paths. One and only one of `x`, `v`, must be provided, as a keyword argument.

v [array-like, optional] Values of a deterministic (one path) process along the timeline.

dtype [data-type, optional] Data-type of the values of the process.

mode [string, optional] Specifies how the piecewise constant segments relate to the reference timeline: 'mid', 'forward', 'backward' set `t[i]` to be the midpoint, start or end point respectively, of the constant segment with value `x[i]` or `v[i]`.

See also:

process

Notes

Parameters `t`, `x`, `v`, `dtype` conform to the `process` instantiation interface and shape requirements.

The returned process `p` behaves as advertised upon interpolation with default interpolation kind (set to `'nearest'` via the `interp_kind` attribute), and may be used as a time dependent piecewise constant parameter in SDE integration. However, its timeline `p.t` and values `p.x` are not guaranteed to coincide with the given `t` or `x`, and should not be relied upon.

6.3 sdepy.montecarlo

`class` `sdepy.`**`montecarlo`**(*sample=None*, *axis=-1*, *bins=100*, *range=None*, *use='all'*, *dtype=None*, *ctype=<class 'numpy.int64'>*)

Summary statistics of Monte Carlo simulations.

Compute, store and cumulate results of Monte Carlo simulations across multiple runs. Cumulated results include mean, standard deviation, standard error, skewness, kurtosis, and 1d-histograms of the distribution of outcomes. Probability distribution function estimates are provided, based on the cumulated histograms.

Parameters

sample [array-like, optional] Initial data set to be summarized. If `None`, an empty instance is provided, initialized with the given parameters.

axis [integer, optional] Axis of the given `sample` enumerating single data points (paths, or different realizations of a simulated process or event). Defaults to the last axis of the sample.

use [{'all', 'even', 'odd'}, optional] If `'all'` (default), the data set is processed as is. If `'even'` or `'odd'`, the sample `x` is assumed to consist of antithetic values along the specified axis, assumed of even size `2*N`, where `x[0], x[1], ...` is antithetic respectively to `x[N], x[N+1], ...`. Summary operations are then applied to a sample of size `N` consisting of the half-sum (`'even'`) or half-difference (`'odd'`) of antithetic values.

bins [array-like, or int, or str, optional] Bins used to evaluate the counts' cumulated distribution are computed, against the first data set encountered, according to the `bins` parameter:

- If `int` or `str`, it dictates the number of bins or their determination method, as passed to `numpy.histogram` when processing the first sample.
- If array-like, overrides `range`, setting explicit bins' boundaries, so that `bins[i][j]` is the lower bound of the `j`-th bin used for the distribution of the `i`-th component of data points.
- If `None`, no distribution data will be computed.

Defaults to `100`.

range [(float, float) or None, optional] Bins range specification, as passed to `numpy.histogram`.

dtype [data-type, optional] Data type used for cumulating moments. If `None`, the data-type of the first sample is used, if of float kind, or `float` otherwise.

ctype [data-type, optional] Data type used for cumulating histogram counts. Defaults to `numpy.int64`.

Notes

The shape of cumulated statistics is set as the shape of the data points of the first data set processed (shape of the first `sample` after summarizing along the paths axis). When cumulating subsequent samples, broadcasting rules apply.

Indexing can be used to access single values or slices of the stored data. Given a montecarlo instance `a`, `a[i]` is a new instance referencing statistics of the `i`-th component of data summarized in `a` (no copying).

The first data set encountered fixes the histogram bins. Points of subsequent data sets that fall outside the bins, while properly taken into account in summary statistics (mean, standard error etc.), are ignored when building cumulated histograms and probability distribution functions. Their number is accounted for in the `outpaths` property and `outerr` method.

Histograms and distributions, and the related `outpaths` and `outerr`, must be invoked on single-valued `montecarlo` instances. For multiple valued simulations, use indexing to select the value to be addressed (e.g. `a[i].histogram()`).

Attributes

`paths` Number of cumulated sample data points (`0` for an empty instance).

vshape Shape of cumulated sample data points.

shape Shape of cumulated sample data set, rearranged with averaging axis as last axis.

outpaths Data points fallen outside of the bins' boundaries.

m Shortcut for the `mean` method.

s Shortcut for the `std` method.

e Shortcut for the `stderr` method.

stats Dictionary of cumulated statistics.

h Shortcut for the `histogram` method.

dh Shortcut for the `density_histogram` method.

Methods

update(sample[, axis])	Add the given sample to the montecarlo simulation.
mean()	Mean of cumulated sample data points.
var()	Variance of cumulated sample data points.
std()	Standard deviation of cumulated sample data points.
skew()	Skewness of cumulated sample data points.
kurtosis()	Kurtosis of cumulated sample data points.
stderr()	Standard error of the mean of cumulated sample data points.
histogram()	Distribution of the cumulated sample data, as a counts histogram.
density_histogram()	Distribution of the cumulated sample data, as a normalized counts histogram.
pdf(x[, method, bandwidth, kind])	Normalized sample probability density function, evaluated at `x`.
cdf(x[, method, bandwidth, kind])	Cumulative sample probability density function, evaluated at `x`.
outerr()	Fraction of cumulated data points fallen outside of the bins' boundaries.

6.3.1 sdepy.montecarlo.update

montecarlo.**update**(*sample*, *axis=-1*)

Add the given sample to the montecarlo simulation.

Combines the given sample data with summary statistics obtained (if any) from former samples to which the `montecarlo` instance was exposed at instantiation and at previous calls to this method. Updates cumulated statistics and histograms accordingly.

Parameters

sample [array-like] Data set to be summarized.

axis [integer, optional] Axis of the given `sample` enumerating single data points (paths, or different realizations of a simulated process or event). Defaults to the last axis of the sample.

6.3.2 sdepy.montecarlo.mean

montecarlo.**mean**()
Mean of cumulated sample data points.

6.3.3 sdepy.montecarlo.var

montecarlo.**var**()
Variance of cumulated sample data points.

6.3.4 sdepy.montecarlo.std

montecarlo.**std**()
Standard deviation of cumulated sample data points.

6.3.5 sdepy.montecarlo.skew

montecarlo.**skew**()
Skewness of cumulated sample data points.

6.3.6 sdepy.montecarlo.kurtosis

montecarlo.**kurtosis**()
Kurtosis of cumulated sample data points.

6.3.7 sdepy.montecarlo.stderr

montecarlo.**stderr**()
Standard error of the mean of cumulated sample data points.

`a.stderr()` equals `a.std()/sqrt(a.paths - 1)`.

6.3.8 sdepy.montecarlo.histogram

montecarlo.**histogram**()
Distribution of the cumulated sample data, as a counts histogram.

Returns a `(counts, bins)` tuple of arrays representing the one-dimensional histogram data of the distribution of cumulated samples (as returned by `numpy.histogram`).

6.3.9 sdepy.montecarlo.density_histogram

montecarlo.**density_histogram**()
Distribution of the cumulated sample data, as a normalized counts histogram.

Returns a `(counts, bins)` tuple of arrays representing the one-dimensional density histogram data of the distribution of cumulated samples (as returned by `numpy.histogram`, the sum of the counts times the bins' widths is 1).

May systematically over-estimate the probability distribution within the bins' boundaries if part of the cumulated samples data (accounted for in the `outpaths` property and `outerr` method) fall outside.

6.3.10 sdepy.montecarlo.pdf

`montecarlo.`**`pdf`**(*x*, *method='gaussian_kde'*, *bandwidth=1.0*, *kind='linear'*)

Normalized sample probability density function, evaluated at `x`.

Parameters

x [array-like] Values at which to evaluate the pdf.

method [{'gaussian_kde', 'interp'}] Specifies the method used to estimate the pdf value. One of: 'gaussian_kde' (default), smooth Gaussian kernel density estimate of the probability density function; 'interp', interpolation of density histogram values, of the given `kind`.

bandwidth [float] The bandwidth of Gaussian kernels is set to `bandwidth` times each bin width.

kind [str] Interpolation kind for the 'interp' method, passed to `scipy.interpolate.intep1d`.

Returns

array An estimate of the sample probability density function of the cumulated sample data, at the given 'x' values, according to the stated method.

Notes

For the 'gaussian_kde' method, kernels are computed at bins midpoints, weighted according to the density histogram counts, using in each bin a bandwidth set to `bandwidth` times the bin width. The resulting pdf:

- Has support on the real line.
- Integrates exactly to 1.
- May not closely track the density histogram counts.

For the 'interp' method, the pdf evaluates to the density histogram counts at each bin midpoint, and to 0 at the bins boundaries and outside. The resulting pdf:

- Has support within the bins boundaries.
- Is intended to track the density histogram counts.
- Integrates close to, but not exactly equal to, 1.

May systematically overestimate the probability distribution within the bins' boundaries if part of the cumulated samples data (accounted for in the `outpaths` property and `outerr` method) fall above or below the bins boundaries.

6.3.11 sdepy.montecarlo.cdf

`montecarlo.`**`cdf`**(*x*, *method='gaussian_kde'*, *bandwidth=1.0*, *kind='linear'*)

Cumulative sample probability density function, evaluated at `x`.

See `pdf` method documentation.

Notes

For the 'gaussian_kde' method, the integral of the Gaussian kernels is expressed in terms of `scipy.special.erf`, and coincides with the integral of the pdf computed with the same method.

For the 'interp' method, the cdf evaluates as follows:

- At bin endpoints, to the cumulated density histogram values weighed by the bins width.

- Below the bins boundaries, to 0.
- Above the bins boundaries, to 1.

It is close to, but not exactly equal to, the integral of the pdf computed with the same method.

6.3.12 sdepy.montecarlo.outerr

montecarlo.**outerr**()
Fraction of cumulated data points fallen outside of the bins' boundaries.

CHAPTER 7

Stochasticity Sources

source(*[, paths, vshape, dtype, rng])	Base class for stochasticity sources.
wiener_source(*[, paths, vshape, dtype, ...])	dw, a source of standard Wiener process (Brownian motion) increments.
poisson_source(*[, paths, vshape, dtype, ...])	dn, a source of Poisson process increments.
cpoisson_source(*[, paths, vshape, dtype, ...])	dj, a source of compound Poisson process increments (jumps).
odd_wiener_source(*[, paths, vshape, dtype, rng])	dw, a source of standard Wiener process (Brownian motion) increments with antithetic paths exposing opposite increments (averages exactly to 0 across paths).
even_poisson_source(*[, paths, vshape, ...])	dn, a source of Poisson process increments with antithetic paths exposing identical increments.
even_cpoisson_source(*[, paths, vshape, ...])	dj, a source of compound Poisson process increments (jumps) with antithetic paths exposing identical increments.
true_source(*[, paths, vshape, dtype, rng, ...])	Base class for stochasticity sources with memory.
true_wiener_source(*[, paths, vshape, ...])	dw, source of standard Wiener process (brownian motion) increments with memory.
true_poisson_source(*[, paths, vshape, ...])	dn, a source of Poisson process increments with memory.
true_cpoisson_source(*[, paths, vshape, ...])	dj, a source of compound Poisson process increments (jumps) with memory.
norm_rv([a, b])	Normal distribution with mean a and standard deviation b, possibly time-dependent.
uniform_rv([a, b])	Uniform distribution between a and b, possibly time-dependent.
exp_rv([a])	Exponential distribution with scale a, possibly time-dependent.
double_exp_rv([a, b, pa])	Double exponential distribution, with scale a with probability pa, and -b with probability (1 - pa), possibly time-dependent.
rvmap(f, y)	Map f to random variates of distribution y, possibly time-dependent.

7.1 sdepy.source

class sdepy.**source**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*)

Base class for stochasticity sources.

Parameters

paths [int] Number of paths (last dimension) of the source realizations.

vshape [tuple of int] Shape of source values.

dtype [data-type] Data type of source values. Defaults to `None`.

rng [numpy.random.Generator, or numpy.random.RandomState, or None] Random numbers generator used. If `None`, defaults to `sdepy.infrastructure.default_rng`, a global variabile initialized on import to `numpy.random.default_rng()`.

Returns

array Once instantiated as `dz`, `dz(t, dt)` returns a random realization of the stochasticity source increments from time `t` to time `t + dt`, with shape `(t + dt).shape + vshape + (paths,)`. For sources with memory (`true_source` class and subclasses), `dz(t)` returns the realized value at time `t` of the source process, according to initial conditions set at instantiation. The definition of source specific parameters, and computation of actual source realizations, are delegated to subclasses. Defaults to an array of `numpy.nan`.

Notes

Any callable object `dz(t, dt)`, with attributes `paths` and `vshape`, returning arrays broadcastable to shape `t_shape + vshape + (paths,)`, where `t_shape` is the shape of `t` and/or `dt`, complies with the `source` protocol. Such object may be passed to any of the process realization classes, to be used as a stochasticity source in integrating or computing the relevant SDE solution. `process` instances, in particular, may be used as stochasticity sources.

When calling `dz(t, dt)`, `t` and/or `dt` can take any shape.

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

__call__(t[, dt])	Realization of stochasticity source values or increments.

7.1.1 sdepy.source.__call__

source.**__call__**(*t*, *dt=None*)

Realization of stochasticity source values or increments.

7.2 sdepy.wiener_source

class sdepy.**wiener_source**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *corr=None*, *rho=None*)

dw, a source of standard Wiener process (Brownian motion) increments.

Parameters

paths [int] Number of paths (last dimension) of the source realizations.

vshape [tuple of int] Shape of source values.

dtype [data-type] Data type of source values. Defaults to `None`.

rng [numpy.random.Generator, or numpy.random.RandomState, or None] Random numbers generator used. If `None`, defaults to `sdepy.infrastructure.default_rng`, a global variabile initialized on import to `numpy.random.default_rng()`.

corr [array-like, or callable, or None] Correlation matrix of the standard Wiener process increments, possibly time-dependent, or `None` for no correlations, or for correlations specified by the `rho` parameter. If not `None`, overrides `rho`. If `corr` is a square matrix of shape `(M, M)`, or callable with `corr(t)` evaluating to such matrix, the last dimension of the source values must be of size `M` (`vshape[-1] == M`), and increments along the last axis of the source values will be correlated accordingly.

rho [array-like, or callable, or None] Correlations of the standard Wiener process increments, possibly time-dependent, or `None` for no correlations. If `rho` is scalar, or callable with `rho(t)` evaluating to a scalar, `M=2` is assumed, and `corr=((1, rho), (rho, 1))`. If `rho` is a vector of shape `(K,)`, or a callable with `rho(t)` evaluating to such vector, `M=2*K` is assumed, and the `M` source values along the last `vshape` dimension are correlated so that `rho[i]` correlates the `i`-th and `K+i`-th values, other correlations being zero (`corr = array((I, R), (R, I))` where `I = numpy.eye(K)` and `R = numpy.diag(rho)`).

Returns

array Once instantiated as `dw`, `dw(t, dt)` returns a random realization of standard Wiener process increments from time `t` to time `t + dt`, with shape `(t + dt).shape + vshape + (paths,)`. The increments are normal variates with mean 0, either independent with standard deviation `sqrt(dt)`, or correlated with covariance matrix `corr*dt`, or `corr(t + dt/2)*dt` (the latter approximates the integral of `corr(t)` from `t` to `t + dt`).

See also:

source

Notes

Realizations across different `t` and/or `dt` array elements, and/or across different paths, and/or along axes of the source values other than the last axis of `vshape`, are independent. `corr` should be a correlation matrix with unit diagonal elements and off-diagonal correlation coefficients, not a covariance matrix.

`corr` and `rho` values with a trailing one-dimensional paths axis are accepted, of shape `(M, M, 1)` or `(M/2, 1)` respectively. This last axis is ignored: this allows for deterministic `process` instances (single path processes) to be passed as valid `corr` or `rho` values. Path dependent `corr` and `rho` are not supported.

For time-dependent correlations, `dw(t, dt)` approximates the increments of a process `w(t)` obeying the SDE `dw(t) = D(t)*dz(t)`, where `z(t)` are standard uncorrelated Wiener processes, and `D(t)` is a time-dependent matrix such that `D(t) @ (D(t).T) == corr(t)`. Note that, given any two time points `s` and `t > s`, by the Ito isometry the expectation value of `(w(t)-w(s))[i] *`

`(w(t)-w(s))[j]`, i.e. the `i`, `j` element of the covariance matrix of increments of `w` from `s` to `t`, equals the integral of `corr(u)[i, j]` in `du` from `s` to `t`.

For time-independent correlations, as well as for correlations that depend linearly on `t`, the resulting `dw(t, dt)` is exact, as far as it can be within the accuracy of the pseudo-random normal variate generator of NumPy. Otherwise, mind using small enough `dt` intervals.

Attributes

corr [array, or callable] Stores the correlation matrix used computing increments. May expose either a reference to `corr`, if provided explicitly, or an appropriate object, in case `rho` was specified.

Methods

__call__(t, dt)	See wiener_source class documentation.

7.2.1 sdepy.wiener_source.__call__

`wiener_source.`**`__call__`**(*t*, *dt*)
See wiener_source class documentation.

7.3 sdepy.poisson_source

class `sdepy.`**`poisson_source`**(**, paths=1, vshape=(), dtype=<class 'int'>, rng=None, lam=1.0*)
dn, a source of Poisson process increments.

Parameters

paths [int] Number of paths (last dimension) of the source realizations.

vshape [tuple of int] Shape of source values.

dtype [data-type] Data type of source values. Defaults to `int`.

rng [numpy.random.Generator, or numpy.random.RandomState, or None] Random numbers generator used. If `None`, defaults to `sdepy.infrastructure.default_rng`, a global variabile initialized on import to `numpy.random.default_rng()`.

lam [array-like, or callable] Intensity of the Poisson process, possibly time-dependent. Should be an array of non-negative values, broadcastable to shape `vshape + (paths,)`, or a callable with `lam(t)` evaluating to such array.

Returns

array Once instantiated as `dn`, `dn(t, dt)` returns a random realization of Poisson process increments from time `t` to time `t + dt`, with shape `(t + dt).shape + vshape + (paths,)`. The increments are independent Poisson variates with mean `lam*dt`, or `lam(t + dt/2)*dt` (the latter approximates the integral of `lam(t)` from `t` to `t + dt`).

See also:

source

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

__call__(t, dt)	See poisson_source class documentation.

7.3.1 sdepy.poisson_source.__call__

poisson_source.**__call__**(*t*, *dt*)
See poisson_source class documentation.

7.4 sdepy.cpoisson_source

class sdepy.**cpoisson_source**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *dn=None*, *ptype=<class 'int'>*, *lam=1.0*, *y=None*)
dj, a source of compound Poisson process increments (jumps).

Parameters

paths [int] Number of paths (last dimension) of the source realizations.

vshape [tuple of int] Shape of source values.

dtype [data-type] Data type of source values. Defaults to `None`.

rng [numpy.random.Generator, or numpy.random.RandomState, or None] Random numbers generator used. If `None`, defaults to `sdepy.infrastructure.default_rng`, a global variabile initialized on import to `numpy.random.default_rng()`. Used to generate Poisson process increments (unless `dn` is explicilty given with its own `dn.rng`) and `y` random variates.

dn [source or source class, or None] Underlying source of Poisson process increments. If a class, it is used to instantiate the source; if a source, it is used as it is, overriding the given `ptype` and `lam` parameters. If `None`, it is instantiated as a `sdepy.poisson_source`.

ptype [data-type] Data type of Poisson process increments. Defaults to `int`.

lam [array-like, or callable] Intensity of the underlying Poisson process, possibly time-dependent. See `poisson_source` class documentation.

y [distribution, or callable, or None] Distribution of random variates to be compounded with the Poisson process increments, possibly time-dependent. May be any `scipy.stats` distribution instance, or any object exposing an `rvs` method to be invoked with signature `rvs(size=output_shape, random_state=rng)` to generate independent random variates with given shape and generator, or a callable with `y(t)` evaluating to such object. The following preset distributions may be specified, possibly with time-varying parameters:

- `y=norm_rv(a, b)` - normal distribution with mean `a` and standard deviation `b`.
- `y=uniform_rv(a, b)` - uniform distribution between `a` and `b`.
- `y=exp_rv(a)` - exponential distribution with scale `a`.
- `y=double_exp_rv(a, b, pa)` - double exponential distribution, with scale `a` with probability `pa`, and `-b` with probability `1 - pa`.

where `a, b, pa` are array-like with values in the appropriate domains, broadcastable to a shape `vshape + (paths,)`, or callables with `a(t), b(t), pa(t)` evaluating to such arrays. If `None`, defaults to `uniform_rv(a=0, b=1)`.

Returns

array Once instantiated as `dj`, `dj(t, dt)` returns a random realization of compound Poisson process increments from time `t` to time `t + dt`, with shape `(t + dt).shape + vshape + (paths,)`. The increments are independent compound Poisson variates, consisting of the sum of `N` independent `y` or `y(t + dt/2)` variates, where `N` is a Poisson variate with mean `lam*dt`, or `lam(t + dt/2)*dt` (approximates each variate being taken from `y` at the time of the corresponding Poisson process event).

See also:

poisson_source

source

norm_rv

uniform_rv

exp_rv

double_exp_rv

rvmap

Notes

Preset distributions `norm_rv, uniform_rv, exp_rv, double_exp_rv` behave as follows:

- If all parameters are array-like, return an object with an `rvs` method as described above, and with methods `mean, std, var, exp_mean` with signature `()`, returning the mean, standard deviation, variance and mean of the exponential of the random variate.
- If any parameter is callable, returns a callable `y` such that `y(t)` evaluates to the corresponding distribution with parameter values at time `t`.

To compound the Poisson process increments with a function `f(z)`, or time-dependent function `f(t, z)`, of a given random variate `z`, one can pass `y=rvmap(f, z)` to `compound_poisson`.

[ToDo: make a note on martingale correction using exp_mean]

Attributes

y [distribution, or callable] Stores the distribution used computing the Poisson process increments.

dn_value [array of int] After each realization, this attribute stores the underlying Poisson process increments.

y_value [list of array] After each realization, this attribute stores the underlying `y` random variates.

Methods

__call__(t, dt)	See cpoisson_source class documentation.

7.4.1 sdepy.cpoisson_source.__call__

cpoisson_source.**__call__**(*t*, *dt*)
See cpoisson_source class documentation.

7.5 sdepy.odd_wiener_source

class sdepy.**odd_wiener_source**(*, *paths=2*, *vshape=()*, *dtype=None*, *rng=None*, ***args*)
dw, a source of standard Wiener process (Brownian motion) increments with antithetic paths exposing opposite increments (averages exactly to 0 across paths).

Once instantiated as `dw` with `paths=2*K` paths, `x = dw(t, dt)` consists of leading `K` paths with independent increments, and trailing `K` paths consisting of a copy of the leading paths with sign reversed (`x[..., i] == -x[..., K + i]`).

See also:

wiener_source

Attributes

dtype

paths

rng

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

vshape

Methods

__call__	

7.6 sdepy.even_poisson_source

class sdepy.**even_poisson_source**(*, *paths=2*, *vshape=()*, *dtype=None*, *rng=None*, ***args*)
dn, a source of Poisson process increments with antithetic paths exposing identical increments.

Once instantiated as `dn` with `paths=2*K` paths, `x = dn(t, dt)` consists of leading `K` paths with independent increments, and trailing `K` paths consisting of a copy of the leading paths: (`x[..., i] == x[..., K + i]`). Intended to be used together with `odd_wiener_source` to generate antithetic paths in jump-diffusion processes.

See also:

source

poisson_source

Attributes

dtype

paths

rng

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

vshape

Methods

__call__	

7.7 sdepy.even_cpoisson_source

class sdepy.**even_cpoisson_source**(*, *paths=2*, *vshape=()*, *dtype=None*, *rng=None*, ***args*)

dj, a source of compound Poisson process increments (jumps) with antithetic paths exposing identical increments.

Once instantiated as `dj` with `paths=2*K` paths, `x = dj(t, dt)` consists of leading `K` paths with independent increments, and trailing `K` paths consisting of a copy of the leading paths: `x[..., i]` equals `x[..., K + i]`. Intended to be used together with `odd_wiener_source` to generate antithetic paths in jump-diffusion processes.

See also:

source

cpoisson_source

Attributes

dtype

paths

rng

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

vshape

Methods

__call__	

7.8 sdepy.true_source

class sdepy.**true_source**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *rtol='max'*, *t0=0.0*, *z0=0.0*)

Base class for stochasticity sources with memory.

Parameters

paths, vshape, dtype, rng See `source` class documentation.

rtol [float, or 'max'] relative tolerance used in assessing the coincidence of `t` with the time of a previously stored realization of the source. If set to `max`, the resolution of the `float` type is used.

t0, z0 [array-like] z0 is the initial value of the source at time t0.

Returns

array Once instantiated as `dz`, `dz(t)` returns the realized value at time `t` of the source process, such that `dz(t0) = z0`, with shape `(t + dt).shape + vshape + (paths,)`, as specified by subclasses. `dz(t, dt)` returns `dz(t + dt) - dz(t)`. New values of `dz(t)` should follow a probability distribution conditional on values realized in previous calls. Defaults to an array of `numpy.nan`.

See also:

source

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

`__getitem__`(index)	Reference to a sub-array or element of the source values sharing the same memory of past realizations.
new_inside(z1, z2, t1, t2, s)	Generate a new process increment, at a time s between those of formerly realized values.
new_outside(z, t, s)	Generate a new process increment, at a time s above or below those of formerly realized values.

7.8.1 sdepy.true_source.new_inside

`true_source.`**`new_inside`**(*z1*, *z2*, *t1*, *t2*, *s*)

Generate a new process increment, at a time s between those of formerly realized values.

Parameters

z1, z2 [array] Formerly realized values of the source at times `t1, t2` respectively.

t1, t2 [float] `t1, t2` are the times of former realizations closest to `s`, with `t1 < s < t2`.

Returns

array Value of the source at `s`, conditional on formerly realized value `z1` at `t1` and `z2` at `t2`. Should be defined by subclasses. Defaults to an array of `numpy.nan`.

7.8.2 sdepy.true_source.new_outside

`true_source.`**`new_outside`**(*z*, *t*, *s*)

Generate a new process increment, at a time s above or below those of formerly realized values.

Parameters

z [array] Formerly realized value of the source at time `t`.

t, s [float] `t` is the highest (lowest) time of former realizations, and s is above (below) `t`.

Returns

array Value of the source at `s`, conditional on formerly realized value `z` at `t`. Should be defined by subclasses. Defaults to an array of `numpy.nan`.

7.9 sdepy.true_wiener_source

class `sdepy.`**`true_wiener_source`**(**, paths=1, vshape=(), dtype=None, rng=None, corr=None, rho=None, rtol='max', t0=0.0, z0=0.0*)

dw, source of standard Wiener process (brownian motion) increments with memory.

Parameters

paths, vshape, dtype, corr, rho

See `wiener_source` class documentation.

rtol, t0, z0 See `true_source` class documentation.

Returns

array Once instantiated as `dw`, `dw(t)` returns `z0` plus a realization of the standard Wiener process increment from time `t0` to `t`, and `dw(t, dt)` returns `dw(t + dt) - dw(t)`. The returned values follow a probability distribution conditional on values realized in previous calls.

Notes

For time-independent correlations, as well as for correlations that depend linearly on `t`, the resulting `w(t)` is exact, as far as it can be within the accuracy of the pseudo-random normal variate generator of NumPy. Otherwise, mind running a first evaluation of `w(t)` on a sequence of consecutive closely spaced time points in the region of interest.

Given `t1 < s < t2`, the value of `w(s)` conditional on `w(t1)` and `w(t2)` is computed as follows.

Let `A` and `B` be respectively the time integral of `corr(t)` between `t1` and `s`, and between `s` and `t2`, such that:

- `A + B` is the expected covariance matrix of `w(t2) - w(t1)`,
- `A` is the expected covariance matrix of `w(s) - w(t1)`,
- `B` is the expected covariance matrix of `w(t2) - w(s)`.

Let `Z = B @ np.linalg.inv(A + B)`, and let `y` be a random normal variate, independent from `w(t1)` and `w(t2)`, with covariance matrix `Z @ A` (note that the latter is a symmetric matrix, as a consequence of the symmetry of `A` and `B`).

Then, the follwing expression provides for a `w(s)` with the needed correlations, and with `w(s) - w(t1)` independent from `w(t1)`, `w(t2) - w(s)` independent from `w(s)`:

```
w(s) = Z @ w(t1) + (1 - Z) @ w(t2) + y
```

This is easily proved by direct computation of the relevant correlation matrices, and by using the fact that the random variables at play are jointly normal, and hence lack of correlation entails independence.

Note that, when invoking `w(s)`, `A` is approximated as `corr((t1+s)/2)*(s-t1)`, and `B` is approximated as `corr(s+t2)/2)*(t2-s)`.

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

See source and true_source methods.	

7.10 sdepy.true_poisson_source

class sdepy.**true_poisson_source**(*, *paths=1*, *vshape=()*, *dtype=<class 'int'>*, *rng=None*, *lam=1.0*, *rtol='max'*, *t0=0.0*, *z0=0*)

dn, a source of Poisson process increments with memory.

Parameters

paths, vshape, dtype, lam See `poisson_source` class documentation.

rtol, t0, z0 See `true_source` class documentation.

Returns

array Once instantiated as `dn`, `dn(t)` returns `z0` plus a realization of Poisson process increments from time `t0` to `t`, and `dn(t, dt)` returns `dn(t + dt) - dn(t)`. The returned values follow a probability distribution conditional on the realized values in previous calls.

See also:

source

poisson_source

true_source

Notes

For time-dependent intensity `lam(t)` the result is approximate, mind running a first evaluation on a sequence of consecutive closely spaced time points in the region of interest.

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

See ‘‘source‘‘ and ‘‘true_source‘‘ methods.	

7.11 sdepy.true_cpoisson_source

class sdepy.**true_cpoisson_source**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *rtol='max'*, *t0=0.0*, *z0=0.0*, *dn=None*, *ptype=<class 'int'>*, *lam=1.0*, *y=None*)

dj, a source of compound Poisson process increments (jumps) with memory.

Parameters

paths, vshape, dtype, dn, ptype, lam, y See `cpoisson_source` class documentation.

rtol, t0, z0 See `true_source` class documentation.

dn [true_poission_source, or None] If provided, it is used as the underlying source of Poisson process increments with memory, overriding the given `ptype` and `lam`. If `None` (default), it is instantiated as a `sdepy.true_poission_source`.

Returns

array Once instantiated as `dj`, `dj(t)` returns `z0` plus a realization of compound Poisson process increments from time `t0` to `t`, and `dj(t, dt)` returns `dj(t + dt) - dj(t)`. The returned values follow a probability distribution conditional on the realized values in previous calls.

See also:

source

cpoisson_source

true_source

Notes

For time-dependent intensity `lam(t)` and compounding random variable `y(t)` the result is approximate, mind running a first evaluation on a sequence of consecutive closely spaced time points in the region of interest.

Attributes

rng Read-only access to the random number generator used by the stochasticity source.

size Returns the number of stored scalar values from previous evaluations, or 0 for sources without memory.

t Returns a copy of the time points at which source values have been stored from previous evaluations, as an array, or an empty array for sources without memory.

Methods

See ‘‘source‘‘ and ‘‘true_source‘‘ methods.	

7.12 sdepy.norm_rv

`sdepy.`**`norm_rv`**(*a=0*, *b=1*)

Normal distribution with mean a and standard deviation b, possibly time-dependent.

Wraps `scipy.stats.norm(loc=a, scale=b)`.

See also:

cpoisson_source

7.13 sdepy.uniform_rv

`sdepy.`**`uniform_rv`**(*a=0*, *b=1*)

Uniform distribution between a and b, possibly time-dependent.

Wraps `scipy.stats.uniform(loc=a, scale=b-a)`.

See also:

cpoisson_source

7.14 sdepy.exp_rv

`sdepy.`**`exp_rv`**(*a=1*)

Exponential distribution with scale a, possibly time-dependent.

Wraps `scipy.stats.expon(scale=a)`. The probability distribution function is:

- if `a > 0`, `pdf(x) = a*exp(-a*x)`, with support in `[0, inf)`
- if `a < 0`, `pdf(x) = -a*exp( a*x)`, with support in `(-inf, 0]`

See also:

cpoisson_source

7.15 sdepy.double_exp_rv

`sdepy.`**`double_exp_rv`**(*a=1*, *b=1*, *pa=0.5*)

Double exponential distribution, with scale a with probability pa, and -b with probability (1 - pa), possibly time-dependent.

Double exponential distribution, with probability distribution

- for `x` in `[0, inf)`, `pdf(x) = pa*exp(-a*x)*a`
- for `x` in `(-inf, 0)`, `pdf(x) = (1-pa)*exp(b*x)*b`

where `a` and `b` are positive and `pa` is in `[0, 1]`.

See also:

cpoisson_source

7.16 sdepy.rvmap

sdepy.**rvmap**(*f*, *y*)

Map f to random variates of distribution y, possibly time-dependent.

Parameters

f [callable] Callable with signature `f(y)`, or `f(t, y)` or `f(s, y)`, to be mapped to the random variates of `y` or `y(t)`

y [distribution, or callable] Distribution, possibly time-dependent, as accepted by `cpoisson_source`.

Returns

new_y [Distribution, or callable] An object with and `rvs(shape)` method, or a callable with `new_y(t)` evaluating to such object, as accepted by `cpoisson_source`. `new_y.rvs(shape)`, or `new_y(t).rvs(shape)`, returns `f(y.rvs(shape))`, or `f([t, ] y(t).rvs(shape)`.

See also:

cpoisson_source

norm_rv

uniform_rv

exp_rv

double_exp_rv

Notes

`new_y` does not provide any `mean, std, var, exp_mean` method.

To be recognized as time-dependent, `f` should have its first parameter named `t` or `s`.

CHAPTER 8

SDE Integration Framework

paths_generator(*[, paths, xshape, wshape, ...])	Step by step generation of stochastic simulations across multiple paths, intended for subclassing.
integrator(*[, paths, xshape, wshape, ...])	Step by step numerical integration of Ito Stochastic Differential Equations (SDEs), intended for subclassing.
SDE(*[, paths, vshape, dtype, rng, steps, ...])	Class representation of a user defined Stochastic Differential Equation (SDE), intended for subclassing.
SDEs(*[, paths, vshape, dtype, rng, steps, ...])	Class representation of a user defined system of Stochastic Differential Equations (SDEs), intended for subclassing.
integrate([sde, q, sources, log, addaxis])	Decorator for Ito Stochastic Differential Equation (SDE) integration.

8.1 sdepy.paths_generator

class sdepy.**paths_generator**(*, *paths=1*, *xshape=()*, *wshape=()*, *dtype=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*)

Step by step generation of stochastic simulations across multiple paths, intended for subclassing.

Given a number of requested paths, shapes and output timeline, encapsulates the low level tasks of memory allocation and step by step iteration along the timeline.

The definition of the target iteration steps (`pace` method), initialization (`begin` method), computation of next step (`next` method), storing results at points on the requested timeline (`store` method), cleaning up (`end` method) and evaluation of a final result to be returned (`exit` method), are delegated to subclasses.

Instances are callables with signature `(timeline)` that iterate subclass methods along the given timeline, using the configuration set out at instantiation.

Parameters

paths [int] Size of last axis of the allocated arrays (number of paths of the simulation).

xshape [int or tuple of int] Shape of values that will be stored at each point of the output timeline.

wshape [int or tuple of int] Shape of working space used for step by step iteration.

dtype [data-type] Data-type of the output and working space.

steps [iterable, or int, or None] Specification of the time points to be touched during the simulation (as defined by the `pace` method). Default behaviour is:

- if `None`, the simulated steps coincide with the timeline;
- if `int`, the simulated steps touch all timeline points, as well as `steps` equally spaced points between the minimum and maximum point in the timeline;
- if iterable, the simulated steps touch all timeline points, as well as all values in `steps` between the minimum and maximum points in the timeline.

i0 [int] Index along the timeline at which the simulation starts. The timeline is assumed to be in ascending order. The simulation is performed backwards from `timeline[i0]` to `timeline[0]`, and forwards from `timeline[i0]` to `timeline[-1]`.

info [dict, or None] Diagnostic information about the simulation is stored in this dictionary and is accessible as the `info` attribute. If `None`, a new empty `dict` is used.

getinfo [bool] If `True` (default), records basic information in the `info` attribute about the last performed simulation (if the simulation is both backwards and forwards, the information pertains to the forwards part only). Used by subclasses to enable/disable diagnostic info generation.

Returns

simulation results Once instantiated as `p`, `p(timeline)` runs the simulation along the given timeline, based on parameters of instantiation, returning results as determined by subclass methods. Defaults to `(tt, xx)` where `tt` is a reference to `timeline` and `xx` is an array of `numpy.nan` of the requested shape.

See also:

integrator

SDE

SDEs

Notes

All initialization parameters are stored as attributes of the same name, and may be accessed by subclasses.

During the simulation, a `itervars` attribute is present, pointing to a dictionary that contains the following items, to be used by subclass methods (double letters refer to values along the entire timeline, single letters refer to single time points):

- `steps_tt` : an array of all time points to be touched by the simulation. It consolidates the output timeline and the time points to be touched, as specified by `steps`.
- `tt`: the output timeline.
- `xx`: simulation output, an array of shape `tt.shape + xshape + (paths,)`. `xx[i]` is the simulated value at time `tt[i]`.
- `sw`: working space for time points, an array of shape `(depth,)`.
- `xw`: working space for paths generation, an array of objects of shape `(depth,)`, where each of `xw[k]` is an array of shape `wshape + (paths,)`.
- `reverse` : `True` if the simulation runs backwards, `False` otherwise. If `True`, `steps_tt` and `tt` are in descending order.
- `i` : such that `tt[i]` is the next point along the timeline that will be reached (when invoking `next`), or the point that was just reached (when invoking `store`).

Note that:

- `sw` and `xw` are rolled at each iteration: subclass methods should not rely on storing references to their elements across iterations.
- `xw[k][...] = value` broadcasts `value` into the allocated memory of `xw[k]` (this is usually what you want), `xw[k] = value` stores `value`, as an object, in `xw[k]` (avoid).
- `xx` and `xw[k]` are initialized to arrays filled with `numpy.nan`.

Attributes

depth [int] Number of time points to be stored in the working space. Defaults to 2.

Methods

__call__(timeline)	Run the simulation along the given timeline.
pace(timeline)	Target integration steps for the current integration.
begin()	Set initial conditions.
next()	Numerical simulation step.
store(i, k)	Store the current integration step into the integration output.
end()	End of iteration optional tasks.
exit(tt, xx)	Final tasks and construction of the output value(s).

8.1.1 sdepy.paths_generator.__call__

paths_generator.**__call__**(*timeline*)

Run the simulation along the given timeline.

Parameters

timeline [array-like] A one dimensional array of strictly increasing numbers, defining the timeline of the simulation.

Returns

Simulation results, as specified by subclass methods.

8.1.2 sdepy.paths_generator.pace

paths_generator.**pace**(*timeline*)

Target integration steps for the current integration.

Parameters

timeline [array] Requested simulation timeline, cast as an array of float data-type.

Returns

array Target time points to be touched during the simulation (typically, more thinly spaced than the output time points in `timeline`, based on the `steps` parameter), to be merged with `timeline`.

May be overridden by subclasses. For default behaviour,

see paths_generator class documentation.

8.1.3 sdepy.paths_generator.begin

paths_generator.**begin**()
Set initial conditions.

Given the time points `sw[0], ..., sw[depth - 2]`, should define and store in the working space the corresponding initial values `xw[0], ..., xw[depth - 2]`. Note that when `begin` gets called, `sw[depth - 1]` and `xw[depth - 1]` are undefined and will be respectively set, and computed, at the first iteration.

Notes

It is called once for each backwards and forwards simulation, after memory allocation and before starting the iteration along the time points in `steps_tt`.

Outline of expected code for `depth=2`:

```
# access itervars
iv = self.itervars
sw, xw = iv['sw'], iv['xw']

# this is the initial time, taken from the
# simulation timeline
t0 = sw[0]
assert t0 == iv['steps_tt'][0] == iv['tt'][0]

# store the initial condition
xw[0][...] = 1.
```

Must be provided by subclasses.

8.1.4 sdepy.paths_generator.next

paths_generator.**next**()
Numerical simulation step.

Given the points `sw[0], ..., sw[depth - 2]` and the corresponding values `xw[0], ..., xw[depth - 2]`, should:

1. Optionally modify the target next time point `sw[depth - 1]`, to a value between `sw[depth - 2]` and `sw[depth - 1]` (this allows for adaptive time steps, with the constraint of touching all point specified by the output timeline and the `steps` parameter).
2. Compute the corresponding value `xw[depth - 1]`

Notes

It is called once per iteration step.

Outline of expected code for `depth=2`:

```
# access itervars
iv = self.itervars
sw, xw = iv['sw'], iv['xw']

# get starting values, and time step to be taken
s0, x0 = sw[0], xw[0]
ds = sw[1] - sw[0]

# compute and store the next step
xw[1][...] = x0 + ds
```

Must be provided by subclasses.

8.1.5 sdepy.paths_generator.store

paths_generator.**store**(*i*, *k*)

Store the current integration step into the integration output.

Should take the k-th value in the working space `xw`, transform it if need be, and store it as the output `xx[i]` at the output time point `tt[i]`.

Parameters

i [int] Index of the output timeline point to set as output.

k [int] Index of the working space point to use as input.

Notes

It is called initially to store the initial values that belong to the output timeline, among those put into the working space by `begin`, and later during the iteration, each time the simulation touches a point on the output timeline.

Outline of expected code for `xshape == wshape` and an exponentiation transformation:

```
# access itervars
iv = self.itervars
sw, xw = iv['sw'], iv['xw']
xx = iv['xx']

# this is the current time, also found
# along the output timeline
s = sw[k]
assert s == iv['tt'][i]

# transform and store
np.exp(xw[k], out=xx[i])
```

Must be provided by subclasses.

8.1.6 sdepy.paths_generator.end

paths_generator.**end**()

End of iteration optional tasks.

It is called once for each backwards and forwards simulation, once the final point in the output timeline has been reached and the simulation ends.

After it is called, `itervars` are deleted.

May be provided by subclasses.

8.1.7 sdepy.paths_generator.exit

paths_generator.**exit**(*tt*, *xx*)

Final tasks and construction of the output value(s).

Parameters

tt [array] Output timeline. It is the timeline passed to the `__call__` method, cast as an array, with its original data-type (if the data-type is of integer kind, the simulation is carried out using floats).

xx [array] Output values along the timeline, as computed by `next` and stored by `store` methods.

Notes

It is called once, after backwards and/or forwards simulations have been completed, and its return value is returned.

Default implementation:

```
return tt, xx
```

May be provided by subclasses.

8.2 sdepy.integrator

class sdepy.**integrator**(*, *paths=1*, *xshape=()*, *wshape=()*, *dtype=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*)

Step by step numerical integration of Ito Stochastic Differential Equations (SDEs), intended for subclassing.

For usage, see the `SDE` class documentation.

This class encapsulates SDE integration methods, and cooperates with the `SDE` class, that should always have precedence in method resolution order. As long as the respective APIs are complied with, a new integrator stated as an `integrator` subclass will interoperate with existing SDEs (as described by `SDE` subclasses), and a new SDE will interoperate with existing integrators.

Parameters

paths, xshape, wshape, dtype, steps, i0, info, getinfo See `paths_generator` class documentation.

method [string] Integration method. Defaults to `'euler'`, for the Euler-Maruyama method (at present, this single method is supported). It is stored as an attribute of the same name.

Returns

process Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, returning the resulting process as determined by the cooperating `SDE` subclass and the chosen integration method. Defaults to a process of `numpy.nan` along the given timeline.

See also:

paths_generator

SDE

SDEs

Notes

The equation to be integrated is exposed to the integration algorithm in a standardized form, via methods `A` and `dZ` delegated to a cooperating `SDE` class. The latter should take care of equation parameters, initial conditions, expected paths and shapes, and should instantiate all necessary stochasticity sources.

The integration method is exposed as the `next` method to the `paths_generator` parent class.

If the `getinfo` attribute is set to `True`, at each integration step the following items are added to the `itervars` dictionary, made available to subclasses to track the integration progress:

- `last_t`: starting time point of the last integration step.
- `last_dt`: time increment of the last integration step.
- `last_x` : starting value of the process, at time `last_t`.
- `last_A`: dictionary of the last computed values of the SDE terms, at time `last_t`.
- `last_dZ`: dictionary of the last realized SDE stochasticity source values, cumulated in the interval from `last_t` to `last_t + last_dt`.
- `new_x` : computed value of the process, at time `last_t + last_dt`.

This becomes relevant in case the output timeline is coarse (e.g. just the initial and final time) but diagnostic information is needed about all integration steps performed (e.g., to track how often the process has changed sign, or to count the number of realized jumps).

Methods

A(t, x)	Value of the SDE terms at time t and process value x.
dZ(t, dt)	Value of the SDE differentials at time t, for time increment dt.
next()	Perform an integration step with the requested method.
euler_next()	Euler-Maruyama integration step.

8.2.1 sdepy.integrator.A

`integrator.`**`A`**(*t*, *x*)

Value of the SDE terms at time t and process value x.

Example of expected code for the SDE `dx = (1 - x)*dt + 2*dw(t)`:

```
return {
    'dt': (1 - x),
    'dw': 2
    }
```

The `SDE` class takes care of casting user-specified equations into this format.

8.2.2 sdepy.integrator.dZ

`integrator.`**`dZ`**(*t*, *dt*)

Value of the SDE differentials at time t, for time increment dt.

Example of expected code for the SDE `dx = (1 - x)*dt + 2*dw(t)`, where `x` has two components:

```
shape = (2, self.paths)
return {
    'dt': dt,
    'dw': wiener_source(vshape=2, paths=self.paths)(0, dt)
    }
```

The `SDE` class takes care of instantiating user-specified stochasticity sources and casting them into this format.

8.2.3 sdepy.integrator.next

`integrator.next()`
Perform an integration step with the requested method.

8.2.4 sdepy.integrator.euler_next

`integrator.euler_next()`
Euler-Maruyama integration step.

8.3 sdepy.SDE

class `sdepy.SDE`(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)
Class representation of a user defined Stochastic Differential Equation (SDE), intended for subclassing.

This class aims to provide an easy to use and flexible interface, allowing to specify user-defined SDEs and expose them in a standardized form to the cooperating `integrator` class (the latter should always follow in method resolution order). A minimal definition of an Ornstein-Uhlenbeck process is as follows:

```
>>> from sdepy import SDE, integrator
>>> class my_process(SDE, integrator):
...     def sde(self, t, x, theta=1., k=1., sigma=1.):
...         return {'dt': k*(theta - x), 'dw': sigma}
```

An SDE is stated as a dictionary, containing for each differential the value of the corresponding coefficient:

```
dx = f(t, x)*dt + g(t, x)*dw + h(t, x)*dj
```

translates to:

```
{'dt': f(t, x), 'dw': g(t, x), 'dj': h(t, x)}
```

Instances are callables with signature `(timeline)` that integrate the SDE along the given timeline, using the configuration set out in the instantiation parameters:

```
>>> P = my_process(x0=1, sigma=0.5, paths=100*1000, steps=100)
>>> x = P(timeline=(0., 0.5, 1.))
>>> x.shape
(3, 100000)
```

Subclasses can specify or customize: the equation and its parameters (`sde` method), initial conditions and preprocessing (`init` method and `log` attribute), shape of the values to be computed and stored (`shapes` method), stochastic differentials appearing in the equation (`sources` attribute) and their parameters and initialization (methods `source_dt`, `source_dw`, `source_dn`, `source_dj`, or any custom `source_{id}` method for a corresponding differential `'{id}'` declared in `sources` and used as a key in `sde` return values), optional non array-like parameters (`more` method), how to store results at points on the requested timeline (`let` method), and postprocessing (`result` method and `log` attribute).

Parameters

paths [int] Number of paths of the process.

vshape [int or tuple of int] Shape of the values of the process.

dtype [data-type, optional] Data-type of the process. Defaults to the numpy default.

rng [numpy.random.Generator, or numpy.random.RandomState, or None] Random numbers generator used to instantiate sources. If `None`, defaults to `sdepy.infrastructure.default_rng`, a global variabile initialized on import to `numpy.random.default_rng()`.

steps [iterable, or int, or None] Specification of the time points to be touched during integration (as accepted by a cooperating `integrator` class). Default behaviour is:

- if `None`, the simulated steps coincide with the timeline;
- if `int`, the simulated steps touch all timeline points, as well as `steps` equally spaced points between the minimum and maximum point in the timeline;
- if iterable, the simulated steps touch all timeline points, as well as all values in `steps` between the minimum and maximum points in the timeline.

i0 [int] Index along the timeline at which the integration starts. The timeline is assumed to be in ascending order. Initial conditions are set at `timeline[i0]`, the integration is performed backwards from `timeline[i0]` to `timeline[0]`, and forwards from `timeline[i0]` to `timeline[-1]`.

info [dict, optional] Diagnostic information about the integration is stored in this dictionary and is accessible as the `info` attribute. Defaults to a new empty `dict`.

getinfo [bool] If `True`, subclass methods `info_begin`, `info_next`, `info_store`, `info_end` are invoked during integration. Defaults to `True`.

method [str] Integration method, as accepted by the `integrator` cooperating class.

****args** [SDE-specific parameters] SDE parameters and initial conditions, as implied by the signature of `sde`, `init` and `more` methods, and stochasticity sources parameters, as implied by the signature of `source_{id}` methods. Each keyword should be used once (e.g. `corr`, a `source_dw` parameter, should not be used as the name of a SDE parameter) or, if repeated, must have consistent default values.

Returns

process Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process as defined by subclass methods. Defaults to a process of `numpy.nan` along the given timeline.

See also:

paths_generator

integrator

SDEs

Notes

Custom stochastic differentials used in the SDE should be recognized, and treated appropriately, by the chosen integration method. This may require customization of the `next` method of the `integrator` class.

All named initialization parameters (`paths`, `steps` etc.) are stored as attributes.

Notes on SDE-specific parameters:

- `init` parameters are converted to arrays via `np.asarray`.
- `sde` and source quantitative parameters may be array-like, or time dependent with signature `(t)`.
- both are converted to arrays via `np.asarray`, and for both, their constant value, or values at each time point, should be broadcastable to a shape `wshape + (paths,)`.
- `more` parameters undergo no further initialization, before being made available to the `shapes` and `more` methods.

If `getinfo` is `True`, the invoked info subclass methods may initialize and cumulate diagnostic information in items of the `info` dictionary, based on read-only access of the internal variables set during integration by `paths_generator` and `integrator` cooperating classes, as exposed in the `itervars` attribute.

Attributes

sources [set or dict] As a class attribute, holds the names of the differentials `'dz'` expected to appear in the equation. As an instance attribute, `sources['dz']` is an object, complying with the `source` protocol, that instantiates the differential `'dz'` used during integration. `sources['dz'](t, dt)` is computed at every step for each `'dz' in sources`, as required by the chosen integration method.

args [dict] Stores parameters passed as `**args` upon initialization of the SDE.

log [bool] If True, the natural logarithm of the initial values set by the `init` method is taken as the initial value of the integration, and the result of the integration is exponentiated back before serving it to the `result` method. The `sde` should expose the appropriate equation for integrating the logarithm of the intended process.

Methods

sde(t, x)	Stochastic Differential Equation (SDE) to be integrated.
shapes(vshape)	Shape of the values to be computed and stored upon integration of the SDE.
source_dt()	Setup a source of deterministic increments, to be used as 'dt' during integration.
source_dw([dw, corr, rho])	Setup a source of standard Wiener process (Brownian motion) increments, to be used as 'dw' during integration.
source_dn([dn, ptype, lam])	Setup a source of Poisson process increments, to be used as 'dn' during integration.
source_dj([dj, dn, ptype, lam, y])	Set up a source of compound Poisson process increments (jumps), to be used as 'dj' during integration.
more()	Further optional non array parameters, and initializations.
init(t, out_x[, x0])	Set initial conditions for SDE integration.
let(t, out_x, x)	Store the value of the integrated process at time point `t` belonging to the requested output timeline.
result(tt, xx)	Compute the integration output.
info_begin()	Optional diagnostic information logging function, called before the integration begins.
info_next()	Optional diagnostic information logging function, called after each integration step.
info_store()	Optional diagnostic information logging function, called after each invocation of the let method.
info_end()	Optional diagnostic information logging function, called after the integration has been completed.

8.3.1 sdepy.SDE.sde

SDE.**sde**(*t*, *x*)

Stochastic Differential Equation (SDE) to be integrated.

Parameters

t [float] Time point at which the SDE should be evaluated.

x [array] Values that the stochastic process takes at time `t`.

sde_args [zero or more arrays, as keyword arguments] SDE parameters, as implied by the `sde` method signature. Passed upon instantiation of the `SDE` class as possibly time-dependent array-like, these parameters are served to the `sde` method once evaluated at `t` and converted to arrays via `np.asarray`.

Returns

sde_terms [dict of arrays] Contains, for each differential stated in the `source` attribute, the value of the corresponding coefficient in the represented SDE.

Notes

`x` should be treated as read-only.

8.3.2 sdepy.SDE.shapes

`SDE.`**`shapes`**(*vshape*)

Shape of the values to be computed and stored upon integration of the SDE.

Parameters

vshape [int or tuple of int] Shape of the values of the integration result, as requested upon instantiation of `SDE`.

Returns

vshape [int or tuple of int] Confirms or overrides the given `vshape`.

xshape [int or tuple of int] Shape of the values stored during integration at the output time points. `out_x` array passed to the `let` method has shape `xshape + (paths,)`. Defaults to `vshape`.

wshape [int or tuple of int] Shape of the working space used during integration. `x` values passed to the `sde` and `let` methods have shape `wshape + (paths,)`. Defaults to `vshape`.

Notes

`xshape` and `wshape` are passed to the parent `paths_generator` class.

`hull_white_SDE` and `heston_SDE` classes illustrate use cases for different values of `vshape`, `xshape` and/or `wshape`.

8.3.3 sdepy.SDE.source_dt

`SDE.`**`source_dt`**()

Setup a source of deterministic increments, to be used as 'dt' during integration.

Returns

An object `z` complying with the `source` protocol,

such that `z(t, dt) == dt`.

8.3.4 sdepy.SDE.source_dw

SDE.**source_dw**(*dw=None*, *corr=None*, *rho=None*)

Setup a source of standard Wiener process (Brownian motion) increments, to be used as 'dw' during integration.

Parameters

dw [source, or source subclass, or None] If an object complying with the `source` protocol, it is returned (`corr` and `rho`, as well as `self.dtype` and `self.rng`, are ignored). If a source subclass, it is instantiated with the given parameters, and returned. If None, a new instance of `wiener_source` is returned, with the given parameters.

corr, rho [see `wiener_source` documentation]

Returns

An object complying with the `source` protocol,

instantiating the requested stochasticity source.

The shape and paths of source values are set to

`self.paths`, `self.wshape` respectively.

If instantiated, `dtype=self.dtype` and `rng=self.rng`

keywords are used upon instantiation.

See also:

wiener_source

8.3.5 sdepy.SDE.source_dn

SDE.**source_dn**(*dn=None*, *ptype=<class 'int'>*, *lam=1.0*)

Setup a source of Poisson process increments, to be used as 'dn' during integration.

Parameters

dn [source, or source subclass, or None] If an object complying with the `source` protocol, it is returned (`ptype` and `lam`, as well as `self.rng`, are ignored). If a source subclass, it is instantiated with the given parameters, and returned. If None, a new instance of `poisson_source` is returned, with the given parameters.

ptype, lam [see `poisson_source` documentation]

Returns

An object complying with the `source` protocol,

instantiating the requested stochasticity source.

The shape and paths of source values are set to

`self.paths`, `self.wshape` respectively.

If instantiated, `rng=self.rng` keyword is used

upon instantiation.

See also:

poisson_source

8.3.6 sdepy.SDE.source_dj

SDE.**source_dj**(*dj=None*, *dn=None*, *ptype=<class 'int'>*, *lam=1.0*, *y=None*)

Set up a source of compound Poisson process increments (jumps), to be used as 'dj' during integration.

Parameters

dj [source, or source subclass, or None] If an object complying with the `source` protocol, it is returned (`ptype`, `lam` and `y`, as well as `self.dtype` and `self.rng`, are ignored). If a source subclass, it is instantiated with the given parameters, and returned. If None, a new instance of `cpoisson_source` is returned, with the given parameters.

ptype, lam, y [see `cpoisson_source` documentation]

Returns

An object complying with the `source` protocol,

instantiating the requested stochasticity source.

The shape and paths of source values are set to

`self.paths`, `self.wshape` respectively.

If instantiated, `dtype=self.dtype` and `rng=self.rng`

keywords are used upon instantiation.

See also:

`cpoisson_source`

8.3.7 sdepy.SDE.more

SDE.**more**()

Further optional non array parameters, and initializations.

Parameters

more_args [zero or more keyword arguments] Further, possibly non array-like, SDE parameters, as implied by the `more` method signature. Passed upon instantiation of the `SDE` class, are served to the `more` method and made available to other methods as items in the `args` attribute.

Notes

The `factors` parameter of `hull_white_SDE` illustrates a use case for the `more` method.

8.3.8 sdepy.SDE.init

SDE.**init**(*t*, *out_x*, *x0=1.0*)

Set initial conditions for SDE integration.

Parameters

t [float] Time point at which initial conditions should be imposed.

out_x [array] Array, shaped `wshape + (paths,)`, where initial conditions are to be stored.

init_args [zero or more arrays, as keyword arguments] Initialization parameters, as implied by the `init` method signature. Passed upon instantiation of the `SDE` class as array-like, these parameters are served to the `init` method converted to arrays via `np.asarray`.

Notes

The default implementation has a single `x0` parameter, and sets `out_x[...] = x0`.

8.3.9 sdepy.SDE.let

`SDE.`**`let`**(*t*, *out_x*, *x*)

Store the value of the integrated process at time point `t` belonging to the requested output timeline.

Parameters

t [float] Time point to which the integration result `x` refers.

out_x [array] Array, shaped `xshape + (paths,)`, where the result `x` is to be stored.

x [array] Integration result at time `t`, shaped `wshape + (paths,)`

Notes

The default implementation sets `out_x[...] = x`.

In case `xshape != wshape`, this method should operate as needed in order to store in `out_x` a value broadcastable to its shape (e.g. it might store in `out_x` only some of the components of `x`).

`x` should be treated as read-only.

8.3.10 sdepy.SDE.result

`SDE.`**`result`**(*tt*, *xx*)

Compute the integration output.

Parameters

tt [array] Output integration timeline.

xx [array] Integration result, shaped `tt.shape + xshape + (paths,)`.

Returns

result Final result, returned to the user.

Notes

The default implementation returns `sdepy.process(t=tt, x=xx)`. In case `vshape != xshape`, this method should operate as needed in order to return a process with values shaped as `vshape` (e.g. it might return a function of the components of `xx`).

8.3.11 sdepy.SDE.info_begin

`SDE.`**`info_begin`**()

Optional diagnostic information logging function, called before the integration begins.

8.3.12 sdepy.SDE.info_next

SDE.**info_next**()
Optional diagnostic information logging function, called after each integration step.

8.3.13 sdepy.SDE.info_store

SDE.**info_store**()
Optional diagnostic information logging function, called after each invocation of the let method.

8.3.14 sdepy.SDE.info_end

SDE.**info_end**()
Optional diagnostic information logging function, called after the integration has been completed.

8.4 sdepy.SDEs

class sdepy.**SDEs**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)
Class representation of a user defined system of Stochastic Differential Equations (SDEs), intended for subclassing.

The parent SDE class represents a single SDE, scalar or multidimensional: by an appropriate choice of the vshape parameter, and composition of equation values, it suffices to describe any system of SDEs.

Its SDEs subclass is added for convenience of representation: it allows to state each equation separately and to retrieve separate processes as a result. The number of equations must be stated as the q attribute. The vshape parameter is taken as the common shape of values in each equation in the system.

A minimal definition of a lognormal process x with stochastic volatility y is as follows:

```
>>> from sdepy import SDEs, integrator
>>> class my_process(SDEs, integrator):
...     q = 2
...     def sde(self, t, x, y, mu=0., sigma=1., xi=1.):
...         return ({'dt': mu*x, 'dw': y*x},
...                 {'dt': 0,    'dw': xi*y})

>>> P = my_process(x0=(1., 2.), xi=0.5, vshape=5,
...                paths=100*1000, steps=100, )
>>> x, y = P(timeline=(0., 0.5, 1.))
>>> x.shape, y.shape
((3, 5, 100000), (3, 5, 100000))
```

See also:

SDE

integrator

paths_generator

Notes

By default, the stochasticity sources of each component equation are realized independently, even if represented in the sde output by the same key ('dw' in the example above).

The way stochasticity sources are instantiated and dispatched to each equation, and how correlations of the Wiener source are set via the corr parameter, depend on the value of the addaxis attribute:

- If `True`, source values have shape `vshape + (q,)`, and the `[kk, i]` component of source values is dispatched to the `kk` component of equation `i` (`kk` is a multiindex spanning shape `vshape`). If given, `corr` must be of shape `(q, q)` and correlates corresponding components across equations.
- If `addaxis` is `False` (default) and `N` is the size of the last axis of `vshape`, the values of the sources have shape `vshape[:-1] + (N*q,)`, and the `[kk, i*N + h]` component of the source values is dispatched to the `[kk, h]` component of equation `i` (`kk` is a multiindex spanning shape `vshape[:-1]`, and `h` is in `range(N)`). If given, `corr` must be of shape `(N*q, N*q)`, and correlates all last components of all equations to each other.

After instantiation, stochasticity sources and correlation matrices may be inspected as follows:

```
>>> P = my_process(vshape=(), rho=0.5)
>>> P.sources['dw'].vshape
(2,)
>>> P.sources['dw'].corr.shape
(2, 2)
>>> P.sources['dw'].corr[0, 1]
0.5
```

Attributes

q [int] Number of equations.

addaxis [bool] Affects the internal representation of the equations: if `True`, a last axis of size `q` is added to `vshape`, if `False`, components are stacked onto the last axis of `vshape`. Defaults to `False`. It is forced to `True` if the process components have scalar values.

Methods

pack(xs)	Packs the given arrays (one per equation) into a single array.
unpack(X)	Unpacks the given array into multiple arrays (one per equation).

8.4.1 sdepy.SDEs.pack

SDEs.**pack**(*xs*)

Packs the given arrays (one per equation) into a single array.

Parameters

xs [list of arrays] List of `self.q` arrays to be packed according to the `addaxis` attribute setting.

Returns

X [array] Array packing the given `xs` along its second-last dimension (the last dimension enumerates paths).

8.4.2 sdepy.SDEs.unpack

SDEs.**unpack**(*X*)

Unpacks the given array into multiple arrays (one per equation).

Parameters

X [array] Array with a last dimension enumerating paths, and a second last dimension to be unpacked according to the `addaxis` attribute setting.

Returns

x, y, ... [list of arrays] List of `self.q` arrays, unpacking the given `X`.

8.5 sdepy.integrate

`sdepy.`**`integrate`**(*sde=None*, *, *q=None*, *sources=None*, *log=False*, *addaxis=False*)

Decorator for Ito Stochastic Differential Equation (SDE) integration.

Decorates a function representing the SDE or SDEs into the corresponding `sdepy` integrator.

Parameters

sde [function] Function to be wrapped. Its signature and values should be as expected for the `sde` method of the `sdepy.SDE` or `sdepy.SDEs` classes.

q [int] Number of equations. If `None`, attempts a test evaluation of `sde` to find out. `q=0` indicates a single equation.

sources [set] Stochasticity sources used in the equation. If `None`, attempts a test evaluation of `sde` to find out.

log [bool] Sets the `log` attribute for the wrapping class.

addaxis [bool] Sets the `addaxis` attribute for the wrapping class.

Returns

A subclass of `sdepy.SDE` or `sdepy.SDEs` as appropriate,

and of `sdepy.integrator`, with the given `sde`

cast as its `sde` method.

Notes

To prevent a test evaluation of `sde`, explicitly provide the intended `q` and `sources` as keyword arguments to `integrate()`. The test evaluation is attempted as `sde()` and, upon failure, again as `sde(1., 1.)`.

Examples

```
>>> from sdepy import integrate
>>> @integrate
... def my_process(t, x, theta=1., k=1., sigma=1.):
...     return {'dt': k*(theta - x), 'dw': sigma}
```

```
>>> P = my_process(x0=1, sigma=0.5, paths=100*1000, steps=100)
>>> x = P(timeline=(0., 0.5, 1.))
>>> x.shape
(3, 100000)
```

CHAPTER 9

Stochastic Processes

wiener_process([paths, vshape, dtype, rng, ...])	Wiener process (Brownian motion) with drift.
lognorm_process([paths, vshape, dtype, rng, ...])	Lognormal process.
ornstein_uhlenbeck_process([paths, vshape, ...])	Ornstein-Uhlenbeck process (mean-reverting Brownian motion).
hull_white_process([paths, vshape, dtype, ...])	F-factors Hull-White process (sum of F correlated mean-reverting Brownian motions).
hull_white_1factor_process([paths, vshape, ...])	1-factor Hull-White process (F=1 Hull-White process with F-index collapsed to a scalar).
cox_ingersoll_ross_process([paths, vshape, ...])	Cox-Ingersoll-Ross mean reverting process.
full_heston_process([paths, vshape, dtype, ...])	Heston stochastic volatility process (returns both process and volatility).
heston_process([paths, vshape, dtype, rng, ...])	Heston stochastic volatility process (stores and returns process only).
jumpdiff_process([paths, vshape, dtype, ...])	Jump-diffusion process (lognormal process with compound Poisson logarithmic jumps).
merton_jumpdiff_process([paths, vshape, ...])	Merton jump-diffusion process (jump-diffusion process with normal jump size distribution).
kou_jumpdiff_process([paths, vshape, dtype, ...])	Double exponential (Kou) jump-diffusion process (jump-diffusion process with double exponential jump size distribution).
wiener_SDE(*[, paths, vshape, dtype, rng, ...])	SDE for a Wiener process (Brownian motion) with drift.
lognorm_SDE(*[, paths, vshape, dtype, rng, ...])	SDE for a lognormal process with drift.
ornstein_uhlenbeck_SDE(*[, paths, vshape, ...])	SDE for an Ornstein-Uhlenbeck process.
hull_white_SDE(*[, paths, vshape, dtype, ...])	SDE for an F-factors Hull White process.
cox_ingersoll_ross_SDE(*[, paths, vshape, ...])	SDE for a Cox-Ingersoll-Ross mean reverting process.
full_heston_SDE(*[, paths, vshape, dtype, ...])	SDE for a Heston stochastic volatility process.
heston_SDE(*[, paths, vshape, dtype, rng, ...])	SDE for a Heston stochastic volatility process.
jumpdiff_SDE(*[, paths, vshape, dtype, rng, ...])	SDE for a jump-diffusion process (lognormal process with compound Poisson logarithmic jumps).

Continued on next page

Table 1 – continued from previous page

merton_jumpdiff_SDE(*[, paths, vshape, ...])	SDE for a Merton jump-diffusion process.
kou_jumpdiff_SDE(*[, paths, vshape, dtype, ...])	SDE for a double exponential (Kou) jump-diffusion process.

9.1 sdepy.wiener_process

class sdepy.**wiener_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=0.*, *mu=0.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*)

Wiener process (Brownian motion) with drift.

Generates a process `x(t)` that solves the following SDE:

```
dx(t) = mu(t)*dt + sigma(t)*dw(t, dt)
```

where `dw(t, dt)` are standard Wiener process increments with correlation matrix specified by `corr(t)` or `rho(t)`. `x0`, SDE parameters and `dw(t, dt)` should broadcast to `vshape + (paths,)`.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

mu, sigma [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

Returns

x [process] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process.

See also:

SDE

SDE.source_dw

wiener_source

wiener_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.

9.2 sdepy.lognorm_process

class sdepy.**lognorm_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*)

Lognormal process.

Generates a process `x(t)` that solves the following SDE:

```
dx(t) = mu(t)*x(t)*dt + sigma(t)*x(t)*dw(t, dt)
```

where `dw(t, dt)` are standard Wiener process increments with correlation matrix specified by `corr(t)` or `rho(t)`. `x0`, SDE parameters and `dw(t, dt)` should broadcast to `vshape + (paths,)`. `x0` should be positive.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

mu, sigma [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

Returns

x [process] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process.

See also:

SDE

SDE.source_dw

wiener_source

wiener_SDE

Notes

`x(t)` is obtained via Euler-Maruyama numerical integration of the following equivalent SDE for `a(t) = log(x(t))`:

```
da(t) = (mu(t) - sigma(t)**2/2)*dt + sigma(t)*dw(t, dt)
```

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.3 sdepy.ornstein_uhlenbeck_process

class sdepy.**ornstein_uhlenbeck_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*)

Ornstein-Uhlenbeck process (mean-reverting Brownian motion).

Generates a process `x(t)` that solves the following SDE:

```
dx(t) = k(t)*(theta(t) - x(t))*dt + sigma(t)*dw(t, dt)
```

where `dw(t, dt)` are standard Wiener process increments with correlation matrix specified by `corr(t)` or `rho(t)`. `x0`, SDE parameters and `dw(t, dt)` should broadcast to `vshape + (paths,)`.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

theta, k, sigma [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

Returns

x [process] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process.

See also:

SDE

SDE.source_dw

wiener_source

ornstein_uhlenbeck_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.4 sdepy.hull_white_process

class sdepy.**hull_white_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *factors=1*, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*)

F-factors Hull-White process (sum of F correlated mean-reverting Brownian motions).

Generates a process x(t) that solves the following SDE:

```
x(t) = y_1(t) + ... + y_F(t)
dy_i(t) = k_i(t)*(theta_i(t) - y_i(t))*dt +
          + sigma_i(t)*dw_i(t, dt)
```

where `dw_i(t, dt)` are standard Wiener process increments with correlations `dw_i(t, dt)*dw_j(t, dt) = corr(t)[i, j]`. `x0`, SDE parameters and `dw(t, dt)` should broadcast to `vshape + (factors, paths)`.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

theta, k, sigma [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

See also:

SDE

SDE.source_dw

wiener_source

hull_white_SDE

ornstein_uhlenbeck_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.5 sdepy.hull_white_1factor_process

class sdepy.**hull_white_1factor_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*)

1-factor Hull-White process (F=1 Hull-White process with F-index collapsed to a scalar). See `hull_white_process` class documentation.

See also:

hull_white_process

ornstein_uhlenbeck_process

Notes

Class added for naming convenience. Differs from a `hull_white_process` with `factors=1` in that the last index of the process parameters has not been reserved to enumerate factors, and no `factors` parameter is present. Synonymous with `ornstein_uhlenbeck_process`.

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.6 sdepy.cox_ingersoll_ross_process

class sdepy.**cox_ingersoll_ross_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *dw=None*, *corr=None*, *rho=None*)

Cox-Ingersoll-Ross mean reverting process.

Generates a process `x(t)` that solves the following SDE:

```
dx(t) = k(t)*(theta(t) - x(t))*dt + xi(t)*sqrt(x(t))*dw(t, dt)
```

where `dw(t, dt)` are standard Wiener process increments with correlation matrix specified by `corr(t)` or `rho(t)`. `x0`, SDE parameters and `dw(t, dt)` should broadcast to `vshape + (paths,)`. `x0`, `theta, k` should be positive.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

theta, k, xi [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

Returns

x [process] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process.

See also:

SDE

SDE.source_dw

wiener_source

cox_ingersoll_ross_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.7 sdepy.full_heston_process

class sdepy.**full_heston_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *dw=None*, *corr=None*, *rho=None*)

Heston stochastic volatility process (returns both process and volatility).

Generates processes x(t) and an y(t) that solve the following SDEs:

```
dx(t) = mu(t)*x(t)*dt + sigma(t)*x(t)*sqrt(y(t))*dw_x(t, dt),
dy(t) = k(t)*(theta(t) - y(t))*dt + xi(t)*sqrt(y(t))*dw_y(t, dt)
```

where, if `N = vshape[-1]` is the size of the last dimension of `x(t)`, `y(t)`, and `dw(t, dt)` are standard Wiener process increments with shape `vshape + (2*N, paths)`:

```
dw(t)[..., i, :]*dw(t)[..., j, :] = corr(t)[..., i, j]*dt
dw_x(t) = dw(t)[..., :N, :],
dw_y(t) = dw(t)[..., N:, :],
```

`x0` and SDE parameters should broadcast to `vshape + (paths,)`. `dw(t, dt)` should broadcast to `vshape[:-1] + (2*vshape[-1], paths)`. `x0, y0, theta, k` should be positive.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0, y0 [array-like] Initial conditions for `x(t)` and `y(t)` processes respectively.

mu, sigma, theta, k, xi [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

Returns

x, y [processes] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting processes.

See also:

SDE

SDE.source_dw

wiener_source

full_heston_SDE

Notes

`x(t), y(t)` are obtained via Euler-Maruyama numerical integration of the above SDE for `y(t)` and of the following equivalent SDE for `a(t) = log(x(t))`, handling negative values of `y(t)` via the full truncation algorithm [1]:

```
da(t) = (mu(t) - y(t)*sigma(t)**2/2)*dt + sqrt(y(t))*dw_x(t)
```

References

[1]

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.8 sdepy.heston_process

class sdepy.**heston_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *dw=None*, *corr=None*, *rho=None*)

Heston stochastic volatility process (stores and returns process only).

Generates a process as in `full_heston_process` (see its documentation), storing and returning the `x(t)` component only.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0, mu, sigma, y0, theta, k, xi, dw, corr, rho See `full_heston_process` class documentation.

Returns

x [process] Once instantiated as `p`, `p(timeline)` performs the integration along the given timeline, based on parameters of instantiation, and returns the resulting process.

See also:

full_heston_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.9 sdepy.jumpdiff_process

class sdepy.**jumpdiff_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*, *dj=None*, *dn=None*, *ptype=int*, *lam=1.*, *y=None*)

Jump-diffusion process (lognormal process with compound Poisson logarithmic jumps).

Generates a process x(t) that solves the following SDE (see [1]):

```
dx(t) = mu(t)*x(t)*dt + sigma(t)*x(t)*dw(t, dt) + x(t)*dj(t, dt)
```

where `dw(t, dt)` are standard Wiener process increments with correlation matrix specified by `corr(t)` or `rho(t)`, and `dj(t, dt)` are increments of a Poisson process with intensity `lam(t)`, compounded with random variates distributed as `exp(y(t)) - 1`.

Parameters

paths, vshape, dtype, rng, steps, i0, info, getinfo, method See `SDE` class documentation.

x0 [array-like] Initial condition.

mu, sigma [array-like, or callable] SDE parameters.

dw, corr, rho Specification of stochasticity source of Wiener process increments. See `SDE.source_dw` documentation.

dj, dn, ptype, lam, y Specification of stochasticity source of compound Poisson process increments. See `SDE.source_dj` documentation.

See also:

SDE

SDE.source_dw

SDE.source_dj

wiener_source

cpoisson_source

jumpdiff_SDE

Notes

The drift of the mean value x_mean(t) of x(t) is mu(t) + nu(t), i.e. dx_mean(t)/dt = x_mean(t)*(mu(t) + nu(t)), where:

```
nu(t) = lam(t)*(y_exp_mean(t) - 1)
y_exp_mean(t) = average of exp(y(t))
```

`x(t)` is obtained via Euler-Maruyama numerical integration of the following equivalent SDE for `a(t) = log(x(t))`:

```
da(t) = (mu(t) - sigma(t)**2/2)*dt + x(t)*sigma(t)*dw(t, dt)
        + x(t)*dh(t, dt)
```

where `dh(t, dt)` are increments of a Poisson process with intensity `lam(t)` compounded with random variates distributed as `y(t)`.

References

[1]

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.10 sdepy.merton_jumpdiff_process

class sdepy.**merton_jumpdiff_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*, *dj=None*, *dn=None*, *ptype=int*, *lam=1.*, *a=0.*, *b=1.*)

Merton jump-diffusion process (jump-diffusion process with normal jump size distribution).

Same as `jumpdiff_process`, where the `y` parameter is specialized to `norm_rv(a, b)`, a normal variate with mean `a(t)` and standard deviation `b(t)`.

See also:

jumpdiff_process

norm_rv

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.11 sdepy.kou_jumpdiff_process

class sdepy.**kou_jumpdiff_process**(*paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, *x0=1.*, *mu=0.*, *sigma=1.*, *dw=None*, *corr=None*, *rho=None*, *dj=None*, *dn=None*, *ptype=int*, *lam=1.*, *a=0.5*, *b=0.5*, *pa=0.5*)

Double exponential (Kou) jump-diffusion process (jump-diffusion process with double exponential jump size distribution).

Same as `jumpdiff_process`, where the `y` parameter is specialized to `double_exp_rv(a, b, pa)`, a double exponential variate with scale `a(t)` with probability `pa(t)`, and `-b(t)` with probability `(1 - pa(t))`.

See also:

jumpdiff_process

double_exp_rv

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.12 sdepy.wiener_SDE

class sdepy.**wiener_SDE**(***, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a Wiener process (Brownian motion) with drift.

See also:

wiener_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.13 sdepy.lognorm_SDE

class sdepy.**lognorm_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a lognormal process with drift.

See also:

lognorm_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.14 sdepy.ornstein_uhlenbeck_SDE

class sdepy.**ornstein_uhlenbeck_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for an Ornstein-Uhlenbeck process.

See also:

ornstein_uhlenbeck_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.15 sdepy.hull_white_SDE

class sdepy.**hull_white_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for an F-factors Hull White process.

See also:

hull_white_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.16 sdepy.cox_ingersoll_ross_SDE

class sdepy.**cox_ingersoll_ross_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a Cox-Ingersoll-Ross mean reverting process.

See also:

cox_ingersoll_ross_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.17 sdepy.full_heston_SDE

class sdepy.**full_heston_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a Heston stochastic volatility process.

See also:

full_heston_process

heston_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.18 sdepy.heston_SDE

class sdepy.**heston_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a Heston stochastic volatility process.

See also:

heston_process

full_heston_process

full_heston_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.19 sdepy.jumpdiff_SDE

class sdepy.**jumpdiff_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a jump-diffusion process (lognormal process with compound Poisson logarithmic jumps).

See also:

jumpdiff_process

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.20 sdepy.merton_jumpdiff_SDE

class sdepy.**merton_jumpdiff_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a Merton jump-diffusion process.

See also:

merton_jumpdiff_process

jumpdiff_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

9.21 sdepy.kou_jumpdiff_SDE

class sdepy.**kou_jumpdiff_SDE**(*, *paths=1*, *vshape=()*, *dtype=None*, *rng=None*, *steps=None*, *i0=0*, *info=None*, *getinfo=True*, *method='euler'*, ***args*)

SDE for a double exponential (Kou) jump-diffusion process.

See also:

kou_jumpdiff_process

jumpdiff_SDE

Attributes

See SDE class documentation.

Methods

See SDE class documentation.	

CHAPTER 10

Analytical Results

wiener_mean(t, *[, x0, mu, sigma])	Mean of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.
wiener_var(t, *[, x0, mu, sigma])	Variance of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.
wiener_std(t, *[, x0, mu, sigma])	Standard deviation of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.
wiener_pdf(t, x, *[, x0, mu, sigma])	Probability distribution function of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at x.
wiener_cdf(t, x, *[, x0, mu, sigma])	Cumulative probability distribution function of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at x.
wiener_chf(t, u, *[, x0, mu, sigma])	Characteristic function of the probability distribution of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at u.
lognorm_mean(t, *[, x0, mu, sigma])	Mean of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.
lognorm_var(t, *[, x0, mu, sigma])	Variance of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.
lognorm_std(t, *[, x0, mu, sigma])	Standard deviation of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.
lognorm_pdf(t, x, *[, x0, mu, sigma])	Probability distribution function of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at x.

Continued on next page

Table 1 – continued from previous page

lognorm_cdf(t, x, *[, x0, mu, sigma])	Cumulative probability distribution function of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at x.
lognorm_log_chf(t, u, *[, x0, mu, sigma])	Characteristic function of the probability distribution of values at time t of the logarithm of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at u.
oruh_mean(t, *[, x0, theta, k, sigma])	Mean of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.
oruh_var(t, *[, x0, theta, k, sigma])	Variance of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.
oruh_std(t, *[, x0, theta, k, sigma])	Standard deviation of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.
oruh_pdf(t, x, *[, x0, theta, k, sigma])	Probability distribution function of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters, evaluated at x.
oruh_cdf(t, x, *[, x0, theta, k, sigma])	Cumulative probability distribution function of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters, evaluated at x.
hw2f_mean(t, *[, x0, theta, k, sigma, rho])	Mean of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.
hw2f_var(t, *[, x0, theta, k, sigma, rho])	Variance of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.
hw2f_std(t, *[, x0, theta, k, sigma, rho])	Standard deviation of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.
hw2f_pdf(t, x, *[, x0, theta, k, sigma, rho])	Probability distribution function of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters, evaluated at x.
hw2f_cdf(**args)	Cumulative probability distribution function of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters, evaluated at x.
cir_mean(t, *[, x0, theta, k, xi])	Mean of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.
cir_var(t, *[, x0, theta, k, xi])	Variance of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.
cir_std(t, *[, x0, theta, k, xi])	Standard deviation of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.

Continued on next page

Table 1 – continued from previous page

cir_pdf(t, x, *[, x0, theta, k, xi])	Probability distribution function of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters, evaluated at x.
heston_log_mean(t, *[, x0, mu, sigma, y0, ...])	Mean of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.
heston_log_var(**args)	Variance of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.
heston_log_std(t, *[, x0, mu, sigma, y0, ...])	Standard deviation of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.
heston_log_pdf(t, logx, *[, x0, mu, sigma, ...])	Probability distribution function of values at time t of the logarithm of a Heston process, (as per the full_heston_process class) with time-independent parameters, evaluated at logx.
heston_log_chf(t, u, *[, x0, mu, sigma, y0, ...])	Characteristic function of the probability distribution of values at time t of the logarithm of a Heston process (as per the full_heston_process class) , with time-independent parameters, evaluated at u.
mjd_log_pdf(t, logx, *[, x0, mu, sigma, ...])	Probability distribution function of values at time t of the logarithm of a Merton jump-diffusion process (as per the merton_jumpdiff_process class), with time-independent parameters, evaluated at logx.
mjd_log_chf(t, u, *[, x0, mu, sigma, lam, a, b])	Characteristic function of the probability distribution of values at time t of the logarithm of a Merton jump-diffusion process (as per the merton_jumpdiff_process class), with time-independent parameters, evaluated at u.
kou_mean(t, *[, x0, mu, sigma, lam, a, b, pa])	Mean of values at time t of a double exponential (Kou) jump-diffusion process (as per the kou_jumpdiff_process class) with time-independent parameters.
kou_log_pdf(t, logx, *[, x0, mu, sigma, ...])	Probability distribution function of values at time t of the logarithm of a double-exponential (Kou) jump-diffusion process (as per the kou_jumpdiff_process class), with time-independent parameters, evaluated at logx.
kou_log_chf(t, u, *[, x0, mu, sigma, lam, ...])	Characteristic function of the probability distribution of values at time t of the logarithm of a Kou jump-diffusion process, (as per the kou_jumpdiff_process class) with time-independent parameters, evaluated at u.
bsd1d2(k, t, *[, x0, r, q, sigma])	Black-Scholes d1 and d2 coefficients.
bscall(k, t, *[, x0, r, q, sigma])	Black-Scholes call option value.
bscall_delta(k, t, *[, x0, r, q, sigma])	Black-Scholes call option delta.
bsput(k, t, *[, x0, r, q, sigma])	Black-Scholes put option value.
bsput_delta(k, t, *[, x0, r, q, sigma])	Black-Scholes put option delta.

10.1 sdepy.wiener_mean

class sdepy.**wiener_mean**(*t*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Mean of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.2 sdepy.wiener_var

class sdepy.**wiener_var**(*t*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Variance of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.3 sdepy.wiener_std

class sdepy.**wiener_std**(*t*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Standard deviation of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.4 sdepy.wiener_pdf

class sdepy.**wiener_pdf**(*t*, *x*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Probability distribution function of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at x.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.5 sdepy.wiener_cdf

class sdepy.**wiener_cdf**(*t*, *x*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Cumulative probability distribution function of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at x.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.6 sdepy.wiener_chf

class sdepy.**wiener_chf**(*t*, *u*, *, *x0=0.*, *mu=0.*, *sigma=1.*)

Characteristic function of the probability distribution of values at time t of a Wiener process (as per the wiener_process class) with time-independent parameters, evaluated at u.

See also:

wiener_process

Attributes

params

Methods

__call__	

10.7 sdepy.lognorm_mean

class sdepy.**lognorm_mean**(*t*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Mean of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.8 sdepy.lognorm_var

class sdepy.**lognorm_var**(*t*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Variance of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.9 sdepy.lognorm_std

class sdepy.**lognorm_std**(*t*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Standard deviation of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.10 sdepy.lognorm_pdf

class sdepy.**lognorm_pdf**(*t*, *x*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Probability distribution function of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at x.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.11 sdepy.lognorm_cdf

class sdepy.**lognorm_cdf**(*t*, *x*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Cumulative probability distribution function of values at time t of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at x.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.12 sdepy.lognorm_log_chf

class sdepy.**lognorm_log_chf**(*t*, *u*, *, *x0=1.*, *mu=0.*, *sigma=1.*)

Characteristic function of the probability distribution of values at time t of the logarithm of a lognormal process (as per the lognorm_process class) with time-independent parameters, evaluated at u.

See also:

lognorm_process

Attributes

params

Methods

__call__	

10.13 sdepy.oruh_mean

class sdepy.**oruh_mean**(*t*, *, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*)

Mean of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.

See also:

ornstein_uhlenbeck_process

Attributes

params

Methods

__call__	

10.14 sdepy.oruh_var

class sdepy.**oruh_var**(*t*, *, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*)

Variance of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.

See also:

ornstein_uhlenbeck_process

Attributes

params

Methods

__call__	

10.15 sdepy.oruh_std

class sdepy.**oruh_std**(*t*, *, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*)

Standard deviation of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters.

See also:

ornstein_uhlenbeck_process

Attributes

params

Methods

__call__	

10.16 sdepy.oruh_pdf

class sdepy.**oruh_pdf**(*t*, *x*, *, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*)

Probability distribution function of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters, evaluated at x.

See also:

ornstein_uhlenbeck_process

Attributes

params

Methods

__call__	

10.17 sdepy.oruh_cdf

class sdepy.**oruh_cdf**(*t*, *x*, *, *x0=0.*, *theta=0.*, *k=1.*, *sigma=1.*)

Cumulative probability distribution function of values at time t of an Ornstein-Uhlenbeck process (as per the ornstein_uhlenbeck_process class) with time-independent parameters, evaluated at x.

See also:

ornstein_uhlenbeck_process

Attributes

params

Methods

__call__	

10.18 sdepy.hw2f_mean

class sdepy.**hw2f_mean**(*t*, *, *x0=(0., 0.)*, *theta=(0., 0.)*, *k=(1., 1.)*, *sigma=(1., 1.)*, *rho=0.*)

Mean of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.

See also:

hull_white_process

Attributes

params

Methods

__call__	

10.19 sdepy.hw2f_var

class sdepy.**hw2f_var**(*t*, *, *x0=(0., 0.)*, *theta=(0., 0.)*, *k=(1., 1.)*, *sigma=(1., 1.)*, *rho=0.*)

Variance of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.

See also:

hull_white_process

Attributes

params

Methods

__call__	

10.20 sdepy.hw2f_std

class sdepy.**hw2f_std**(*t*, *, *x0=(0., 0.)*, *theta=(0., 0.)*, *k=(1., 1.)*, *sigma=(1., 1.)*, *rho=0.*)

Standard deviation of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters.

See also:

hull_white_process

Attributes

params

Methods

__call__	

10.21 sdepy.hw2f_pdf

class sdepy.**hw2f_pdf**(*t*, *x*, *, *x0=(0., 0.)*, *theta=(0., 0.)*, *k=(1., 1.)*, *sigma=(1., 1.)*, *rho=0.*)

Probability distribution function of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters, evaluated at x.

See also:

hull_white_process

Attributes

params

Methods

__call__	

10.22 sdepy.hw2f_cdf

class sdepy.**hw2f_cdf**(**args*)

Cumulative probability distribution function of values at time t of a Hull-White 2-factors process (as per the hull_white_process class) with time-independent parameters, evaluated at x.

See also:

hull_white_process

Attributes

params

Methods

__call__	

10.23 sdepy.cir_mean

class sdepy.**cir_mean**(*t*, *, *x0=1.*, *theta=1.*, *k=1.*, *xi=1.*)

Mean of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.

See also:

cox_ingersoll_ross_process

Attributes

params

Methods

__call__	

10.24 sdepy.cir_var

class sdepy.**cir_var**(*t*, *, *x0=1.*, *theta=1.*, *k=1.*, *xi=1.*)

Variance of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.

See also:

cox_ingersoll_ross_process

Attributes

params

Methods

__call__	

10.25 sdepy.cir_std

class sdepy.**cir_std**(*t*, *, *x0=1.*, *theta=1.*, *k=1.*, *xi=1.*)

Standard deviation of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters.

See also:

cox_ingersoll_ross_process

Attributes

params

Methods

__call__	

10.26 sdepy.cir_pdf

class sdepy.**cir_pdf**(*t*, *x*, ***, *x0=1.*, *theta=1.*, *k=1.*, *xi=1.*)

Probability distribution function of values at time t of a Cox-Ingersoll-Ross process (as per the cox_ingersoll_ross_process class) with time-independent parameters, evaluated at x.

See also:

cox_ingersoll_ross_process

Attributes

params

Methods

__call__	

10.27 sdepy.heston_log_mean

class sdepy.**heston_log_mean**(*t*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *rho=0.*)

Mean of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.

See also:

full_heston_process

Attributes

params

Methods

__call__	

10.28 sdepy.heston_log_var

class sdepy.**heston_log_var**(***args*)

Variance of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.

See also:

full_heston_process

Attributes

params

Methods

__call__	

10.29 sdepy.heston_log_std

class sdepy.**heston_log_std**(*t*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *rho=0.*)

Standard deviation of the logarithm of values at time t of a Heston process (as per the full_heston_process class) with time-independent parameters.

See also:

full_heston_process

Attributes

params

Methods

__call__	

10.30 sdepy.heston_log_pdf

class sdepy.**heston_log_pdf**(*t*, *logx*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *rho=0.*)

Probability distribution function of values at time t of the logarithm of a Heston process, (as per the full_heston_process class) with time-independent parameters, evaluated at logx.

See also:

full_heston_process

Notes

Estimate by numerical integration, using `scipy.integrate.quad`, of the closed-form characteristic function `heston_log_chf`. Integration errors are not reported/checked. Either `t` or `logx` must be a scalar.

Attributes

params

Methods

__call__	

10.31 sdepy.heston_log_chf

class sdepy.**heston_log_chf**(*t*, *u*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *y0=1.*, *theta=1.*, *k=1.*, *xi=1.*, *rho=0.*)

Characteristic function of the probability distribution of values at time t of the logarithm of a Heston process (as per the full_heston_process class) , with time-independent parameters, evaluated at u.

See also:

full_heston_process

Attributes

params

Methods

__call__	

10.32 sdepy.mjd_log_pdf

class sdepy.**mjd_log_pdf**(*t*, *logx*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *lam=1.*, *a=0.0*, *b=1.*)

Probability distribution function of values at time t of the logarithm of a Merton jump-diffusion process (as per the merton_jumpdiff_process class), with time-independent parameters, evaluated at logx.

See also:

jumpdiff_process

merton_jumpdiff_process

mjd_log_chf

Notes

Estimate by numerical integration, using `scipy.integrate.quad`, of the closed-form characteristic function `mjd_log_chf`. Integration errors are not reported/checked. Either `t` or `logx` must be a scalar.

Attributes

params

Methods

__call__	

10.33 sdepy.mjd_log_chf

class sdepy.**mjd_log_chf**(*t*, *u*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *lam=1.*, *a=0.0*, *b=1.*)

Characteristic function of the probability distribution of values at time t of the logarithm of a Merton jump-diffusion process (as per the merton_jumpdiff_process class), with time-independent parameters, evaluated at u.

See also:

jumpdiff_process

merton_jumpdiff_process

Attributes

params

Methods

__call__	

10.34 sdepy.kou_mean

class sdepy.**kou_mean**(*t*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *lam=1.*, *a=0.5*, *b=0.5*, *pa=0.5*)

Mean of values at time t of a double exponential (Kou) jump-diffusion process (as per the kou_jumpdiff_process class) with time-independent parameters.

See also:

jumpdiff_process

kou_jumpdiff_process

Attributes

params

Methods

__call__	

10.35 sdepy.kou_log_pdf

class sdepy.**kou_log_pdf**(*t*, *logx*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *lam=1.*, *pa=0.5*, *a=0.5*, *b=0.5*)

Probability distribution function of values at time t of the logarithm of a double-exponential (Kou) jump-diffusion process (as per the kou_jumpdiff_process class), with time-independent parameters, evaluated at logx.

See also:

jumpdiff_process

kou_jumpdiff_process

kou_log_chf

Notes

Estimate by numerical integration, using `scipy.integrate.quad`, of the closed-form characteristic function `kou_log_chf`. Integration errors are not reported/checked. Either `t` or `logx` must be a scalar.

Attributes

params

Methods

__call__	

10.36 sdepy.kou_log_chf

class sdepy.**kou_log_chf**(*t*, *u*, ***, *x0=1.*, *mu=0.*, *sigma=1.*, *lam=1.*, *a=0.5*, *b=0.5*, *pa=0.5*)

Characteristic function of the probability distribution of values at time t of the logarithm of a Kou jump-diffusion process, (as per the kou_jumpdiff_process class) with time-independent parameters, evaluated at u.

See also:

jumpdiff_process

kou_jumpdiff_process

Attributes

params

Methods

__call__	

10.37 sdepy.bsd1d2

class sdepy.**bsd1d2**(*k*, *t*, ***, *x0=1.*, *r=0.*, *q=0.*, *sigma=1.*)

Black-Scholes d1 and d2 coefficients.

See also:

bscall

Attributes

params

Methods

__call__	

10.38 sdepy.bscall

class sdepy.**bscall**(*k*, *t*, ***, *x0=1.*, *r=0.*, *q=0.*, *sigma=1.*)

Black-Scholes call option value.

Parameters

k [array-like] Strike.

t [array-like] Time to maturity.

x0 [array-like] Initial value of underlying security.

r [array-like] Risk-free rate.

q [array-like] Dividend yield of underlying security.

sigma [array-like] Volatility of underlying security.

Returns

array Risk neutral valuation at time s=0 of an European call option paying `max(x(t) - k, 0)` at maturity, where the price `x(s)` of the underlying security follows a lognormal process with `x(0) = x0` and volatility `sigma`.

See also:

bsd1d2

bscall_delta

bsput

bsput_delta

Notes

`bscall(k, t, x0, r, q, sigma)` returns:

```
bscall_value = x0*exp(-q*t)*norm.cdf(d1) + k*exp(-r*t)*norm.cdf(d2)
```

where `cdf` is `scipy.stats.norm.cdf` and `d1, d2 = bsd1d2(k, t, x0, r, q, sigma)` are given as:

```
d1 = (log(x0/k) + (r - q + sigma**2/2)*t)/(sigma*sqrt(t))

d2 = d1 - sigma*sqrt(t)
```

Attributes

params

Methods

__call__	

10.39 sdepy.bscall_delta

class sdepy.**bscall_delta**(*k*, *t*, ***, *x0=1.*, *r=0.*, *q=0.*, *sigma=1.*)

Black-Scholes call option delta.

See also:

bscall

Attributes

params

Methods

__call__	

10.40 sdepy.bsput

class sdepy.**bsput**(*k*, *t*, ***, *x0=1.*, *r=0.*, *q=0.*, *sigma=1.*)

Black-Scholes put option value.

See also:

bscall

Attributes

params

Methods

__call__	

10.41 sdepy.bsput_delta

class sdepy.**bsput_delta**(*k*, *t*, ***, *x0=1.*, *r=0.*, *q=0.*, *sigma=1.*)

Black-Scholes put option delta.

See also:

bscall

Attributes

params

Methods

__call__	

CHAPTER 11

Shortcuts

Stochasticity sources and preset processes may be addressed using the following shortcuts:

Full name	Shortcut
wiener_source	dw
poisson_source	dn
cpoisson_source	dj
odd_wiener_source	odd_dw
even_poisson_source	even_dn
even_cpoisson_source	even_dj
true_wiener_source	true_dw
true_poisson_source	true_dn
true_cpoisson_source	true_dj
wiener_process	wiener
lognorm_process	lognorm
ornstein_uhlenbeck_process	oruh
hull_white_process	hwff
hull_white_1factor_process	hw1f
cox_ingersoll_ross_process	cir
full_heston_process	heston_xy
heston_process	heston
jumpdiff_process	jumpdiff
merton_jumpdiff_process	mjd
kou_jumpdiff_process	kou

Shortcuts have been wrapped as "kfuncs", objects with managed keyword arguments (see `kfunc` decorator documentation below).

Analytical results are named according to the shortcut of the corresponding process (e.g. `lognorm_mean`, `lognorm_cdf` etc. from the `lognorm` shortcut) and are wrapped as kfuncs as well.

kfunc([f, nvar])	Decorator to wrap classes or functions as objects with managed keyword arguments.
iskfunc(cls_or_object)	Tests if the given class or instance has been wrapped as a kfunc.

11.1 sdepy.kfunc

sdepy.**kfunc**(*f=None*, *, *nvar=None*)

Decorator to wrap classes or functions as objects with managed keyword arguments.

This decorator, intended as an aid to interactive and notebook sessions, wraps a callable, class or function, as a "kfunc" object that handles separately its parameters (keyword-only), whose values are stored in the object, and its variables (positional or keyword), always provided upon evaluation.

Syntax:

```
@kfunc
class my_class:
    def __init__(self, **kwparams):
        ...
    def __call__(self, *var, **kwvar):
        ...

@kfunc(nvar=k)
def my_function(*var, **kwargs):
    ...
```

After decoration, `my_class` is a kfunc with `kwparams` as parameters, and with `var` and `kwvar` as variables, and `my_function` is a kfunc with the first `k` of `var, kwargs` as variables, and the remaining `kwargs` as parameters. For usage, see examples below.

Examples

Wrap `wiener_source` into a kfunc, named `dw`:

```
>>> import numpy
>>> from sdepy import wiener_source, kfunc
>>> dw = kfunc(wiener_source)
```

Instantiate `dw` and evaluate it (this is business as usual):

```
>>> my_instance = dw(paths=100, dtype=numpy.float32)
>>> x = my_instance(t=0, dt=1)
>>> x.shape, x.dtype
((100,), dtype('float32'))
```

Inspect kfunc parameters stored in `my_instance`:

```
>>> my_instance.params  # doctest: +SKIP
{'paths': 100, 'vshape': (), 'dtype': <class 'numpy.float32'>,          'corr':␣
↪None, 'rho': None}
```

Evaluate `my_instance` changing some parameters (call the instance with one or more):

```
>>> x = my_instance(t=0, dt=1, paths=999)
>>> x.shape, x.dtype
((999,), dtype('float32'))
```

Parameters stored in `my_instance` are not affected:

```
>>> my_instance.paths == my_instance.params['paths'] == 100
True
```

Create a new instance, changing some parameters and keeping those already set (call the instance without passing any variables):

```
>>> new_instance = my_instance(vshape=2, rho=0.5)
>>> new_instance.params  # doctest: +SKIP
{'paths': 100, 'vshape': 2, 'dtype': <class 'numpy.float32'>,          'corr':␣
↪None, 'rho': 0.5}
```

Instantiate and evaluate at once (pass one or more variables to the class constructor):

```
>>> x = dw(0, 1, paths=100, dtype=numpy.float32)
>>> x.shape, x.dtype
((100,), dtype('float32'))
```

As long as variables are passed by name, order doesn't matter (omitted variables take default values, if any):

```
>>> x = dw(paths=100, dtype=numpy.float32, dt=1, t=0)
>>> x.shape, x.dtype
((100,), dtype('float32'))
```

Attributes

params [dictionary] Parameter values stored in the instance (read-only). For wrapped `SDE` subclasses, also includes default values of all SDE-specific parameters, as stored in the `args` attribute.

11.2 sdepy.iskfunc

`sdepy.`**`iskfunc`**(*cls_or_object*)

Tests if the given class or instance has been wrapped as a kfunc.

Part IV

Testing

Tests have been set up within the `numpy.testing` framework, and require the `pytest` package to be installed. To launch tests, invoke `sdepy.test()` or `sdepy.test('full')`.

The testing subpackage `sdepy.tests` was written in pursuit of the following goals:

- Maximize *case* coverage, by exposing the package functions and methods to a plurality of different input shapes, values, data types etc., and of different combinations thereof, as may be encountered in practice.
- Provide a quantitative validation of the algorithms, functions and processes covered in `sdepy`.
- Keep dependencies of the test code on the adopted testing framework to a bare minimum.

Most often, a number of testing cases is declared as a list or lists of classes and inputs, a general testing procedure is set up, and the latter is iteratively applied to the former. Unfortunately, all this resulted in a thinly documented (if at all), hard to read, and hard to maintain testing code base - sorry about that.

The quantitative validation of the package, via tests marked as `'slow'` and `'quant'`, is done in two steps:

- To validate a `sdepy` release, tests are run with 100_000 or more paths. Numerical integration results for the mean, standard deviation, probability distribution, and/or characteristic function are compared against their exact values computed analytically from the process parameters. Comparisons are then plotted and visually inspected, and the occasional larger than usual deviation is manually checked to be statistically acceptable, i.e. only so few standard deviations off the mark.
- Each time `sdepy.test('full')` is invoked, to keep testing times manageable and the testing procedure non invasive, tests are run with 100 paths, using NumPy legacy random generation with a fixed seed, the realized errors are then compared and checked against the expected errors, as distributed with the package and stored in the `./tests/cfr` directory. Note that such default tests rely on the stable reproducibility of expected errors, across platforms and versions of Python, NumPy and SciPy.

In order to run tests using current NumPy random generation, with a set number of paths (200_000 in the example below), and inspect graphs and realized errors as saved in the current directory, use the following statement (non backward compatible changes to the testing interface may occur in the future, without prior warning):

```
sdepy.test('full', rng=numpy.random.default_rng(),
           paths=200_000, plot=True, outdir='.')
```

For further details, see `sdepy.test` documentation:

sdepy.test	Invoke the sdepy testing suite (requires pytest>=3.8.1 to be installed).

CHAPTER 12

sdepy.test

sdepy.**test = <sdepy.tests.shared._pytest_tester object>**

Invoke the sdepy testing suite (requires pytest>=3.8.1 to be installed).

Parameters

label [string] Sets the scope of tests. May be one of `'full'` (run all tests), `'fast'` (avoid slow tests), `'slow'` (perform slow tests only), `'quant'` (perform quantitative tests only). Defaults to `'fast'`.

doctests [bool] If `True`, doctests are performed in addition to the tests specified by `label`.

warnings [string] If `'pass'` (default), allows warnings during tests. If `'fail'`, fails tests that issue warnings.

rng [string, or numpy.random.Generator, or numpy.random.RandomState,]

or callable, or None

Specifies the random number generator to be used across tests, and the test mode:

- If `'legacy'` (default), tests are run using legacy numpy random generation with a fixed seed, and fail if realized errors of quantitative tests are not consistent with expected errors for the same seed.
- If a `numpy.random.Generator`, or `numpy.random.RandomState` instance, or a callable with no parameters returning such object, this generator is used across tests; if `None`, the sdepy default random number generator is used. In both cases, realzied errors of quantitative tests are computed, but not checked; to make them available for inspection, use `plot` and `outidir` parameters.

paths [int] Reference number of paths used by quantitative tests (the actual number of paths, linked to the given `path`, may vary). If `rng` is set to `'legacy'`, only values 100 and 100_000 are allowed.

plot [bool] If `True`, execute test code that generates plots; needs matplotlib to be installed.

outdir [string, or None] If not `None`, name of a valid directory in which realized errors of quantitative tests, and plot images if generated, are saved.

verborse [int] A nonzero value increases verbosity of tests.

pytest_args [string, or iterable of strings, or None] If not `None`, specifies an additional argument, or a list of additional arguments, passed to `pytest.main()`.

Returns

`True` if all tests passed, `False` otherwise.

Notes

Non backward compatible changes to the present testing interface may occur in the future, without prior warning.

Bibliography

[1] Andersen L 2007, Efficient Simulation of the Heston Stochastic Volatility Model (available at: https://ssrn.com/abstract=946405 or http://dx.doi.org/10.2139/ssrn.946405)

[1] Tankov P Voltchkova E 2009, Jump-diffusion models: a practitioner's guide, Banque et Marches, No. 99, March-April 2009 (available at: http://www.proba.jussieu.fr/pageperso/tankov/tankov_voltchkova.pdf)

Python Module Index

S

sdepy, 27

Index

Symbols

__call__() (*sdepy.cpoisson_source method*), 49
__call__() (*sdepy.paths_generator method*), 59
__call__() (*sdepy.poisson_source method*), 47
__call__() (*sdepy.process method*), 32
__call__() (*sdepy.source method*), 44
__call__() (*sdepy.wiener_source method*), 46

A

A() (*sdepy.integrator method*), 63

B

begin() (*sdepy.paths_generator method*), 60
bscall (*class in sdepy*), 105
bscall_delta (*class in sdepy*), 106
bsd1d2 (*class in sdepy*), 105
bsput (*class in sdepy*), 107
bsput_delta (*class in sdepy*), 107

C

cdf() (*sdepy.montecarlo method*), 41
cdf() (*sdepy.process method*), 37
chf() (*sdepy.process method*), 36
cir_mean (*class in sdepy*), 99
cir_pdf (*class in sdepy*), 101
cir_std (*class in sdepy*), 100
cir_var (*class in sdepy*), 100
cox_ingersoll_ross_process (*class in sdepy*), 80
cox_ingersoll_ross_SDE (*class in sdepy*), 86
cpoisson_source (*class in sdepy*), 47

D

density_histogram() (*sdepy.montecarlo method*), 40
double_exp_rv() (*in module sdepy*), 55
dZ() (*sdepy.integrator method*), 63

E

end() (*sdepy.paths_generator method*), 61
euler_next() (*sdepy.integrator method*), 64
even_cpoisson_source (*class in sdepy*), 50
even_poisson_source (*class in sdepy*), 49
exit() (*sdepy.paths_generator method*), 61
exp_rv() (*in module sdepy*), 55

F

full_heston_process (*class in sdepy*), 80
full_heston_SDE (*class in sdepy*), 86

H

heston_log_chf (*class in sdepy*), 103
heston_log_mean (*class in sdepy*), 101
heston_log_pdf (*class in sdepy*), 102
heston_log_std (*class in sdepy*), 102
heston_log_var (*class in sdepy*), 101
heston_process (*class in sdepy*), 82
heston_SDE (*class in sdepy*), 86
histogram() (*sdepy.montecarlo method*), 40
hull_white_1factor_process (*class in sdepy*), 79
hull_white_process (*class in sdepy*), 78
hull_white_SDE (*class in sdepy*), 85
hw2f_cdf (*class in sdepy*), 99
hw2f_mean (*class in sdepy*), 98
hw2f_pdf (*class in sdepy*), 99
hw2f_std (*class in sdepy*), 98
hw2f_var (*class in sdepy*), 98

I

info_begin() (*sdepy.SDE method*), 70
info_end() (*sdepy.SDE method*), 71
info_next() (*sdepy.SDE method*), 71
info_store() (*sdepy.SDE method*), 71
init() (*sdepy.SDE method*), 69
integrate() (*in module sdepy*), 73
integrator (*class in sdepy*), 62
interp() (*sdepy.process method*), 32
iskfunc() (*in module sdepy*), 111

J

jumpdiff_process (*class in sdepy*), 82
jumpdiff_SDE (*class in sdepy*), 87

K

kfunc() (*in module sdepy*), 110

kou_jumpdiff_process (*class in sdepy*), 84
kou_jumpdiff_SDE (*class in sdepy*), 88
kou_log_chf (*class in sdepy*), 105
kou_log_pdf (*class in sdepy*), 104
kou_mean (*class in sdepy*), 104
kurtosis() (*sdepy.montecarlo method*), 40

L

let() (*sdepy.SDE method*), 70
lognorm_cdf (*class in sdepy*), 95
lognorm_log_chf (*class in sdepy*), 95
lognorm_mean (*class in sdepy*), 94
lognorm_pdf (*class in sdepy*), 95
lognorm_process (*class in sdepy*), 76
lognorm_SDE (*class in sdepy*), 85
lognorm_std (*class in sdepy*), 94
lognorm_var (*class in sdepy*), 94

M

mean() (*sdepy.montecarlo method*), 40
merton_jumpdiff_process (*class in sdepy*), 83
merton_jumpdiff_SDE (*class in sdepy*), 87
mjd_log_chf (*class in sdepy*), 103
mjd_log_pdf (*class in sdepy*), 103
montecarlo (*class in sdepy*), 37
more() (*sdepy.SDE method*), 69

N

new_inside() (*sdepy.true_source method*), 51
new_outside() (*sdepy.true_source method*), 52
next() (*sdepy.integrator method*), 64
next() (*sdepy.paths_generator method*), 60
norm_rv() (*in module sdepy*), 55

O

odd_wiener_source (*class in sdepy*), 49
ornstein_uhlenbeck_process (*class in sdepy*), 77
ornstein_uhlenbeck_SDE (*class in sdepy*), 85
oruh_cdf (*class in sdepy*), 97
oruh_mean (*class in sdepy*), 96
oruh_pdf (*class in sdepy*), 97
oruh_std (*class in sdepy*), 97
oruh_var (*class in sdepy*), 96
outerr() (*sdepy.montecarlo method*), 42

P

pace() (*sdepy.paths_generator method*), 59
pack() (*sdepy.SDEs method*), 72
paths_generator (*class in sdepy*), 57
pcopy() (*sdepy.process method*), 33
pdf() (*sdepy.montecarlo method*), 41
piecewise() (*in module sdepy*), 37
pmax() (*sdepy.process method*), 33
pmean() (*sdepy.process method*), 34
pmin() (*sdepy.process method*), 33
poisson_source (*class in sdepy*), 46
process (*class in sdepy*), 29
pstd() (*sdepy.process method*), 34
psum() (*sdepy.process method*), 33
pvar() (*sdepy.process method*), 34

R

rebase() (*sdepy.process method*), 33
result() (*sdepy.SDE method*), 70
rvmap() (*in module sdepy*), 56

S

SDE (*class in sdepy*), 64
sde() (*sdepy.SDE method*), 66
sdepy (*module*), 27
SDEs (*class in sdepy*), 71
shapeas() (*sdepy.process method*), 33
shapes() (*sdepy.SDE method*), 67
skew() (*sdepy.montecarlo method*), 40
source (*class in sdepy*), 44
source_dj() (*sdepy.SDE method*), 69
source_dn() (*sdepy.SDE method*), 68
source_dt() (*sdepy.SDE method*), 67
source_dw() (*sdepy.SDE method*), 68
std() (*sdepy.montecarlo method*), 40
stderr() (*sdepy.montecarlo method*), 40
store() (*sdepy.paths_generator method*), 61

T

tcopy() (*sdepy.process method*), 33
tder() (*sdepy.process method*), 36
tdiff() (*sdepy.process method*), 35
test (*in module sdepy*), 117
tint() (*sdepy.process method*), 36
tmax() (*sdepy.process method*), 35
tmean() (*sdepy.process method*), 35
tmin() (*sdepy.process method*), 34
true_cpoisson_source (*class in sdepy*), 54
true_poisson_source (*class in sdepy*), 53
true_source (*class in sdepy*), 50
true_wiener_source (*class in sdepy*), 52
tstd() (*sdepy.process method*), 35
tsum() (*sdepy.process method*), 35
tvar() (*sdepy.process method*), 35

U

uniform_rv() (*in module sdepy*), 55
unpack() (*sdepy.SDEs method*), 72
update() (*sdepy.montecarlo method*), 39

V

var() (*sdepy.montecarlo method*), 40
vmax() (*sdepy.process method*), 34
vmean() (*sdepy.process method*), 34
vmin() (*sdepy.process method*), 34
vstd() (*sdepy.process method*), 34
vsum() (*sdepy.process method*), 34
vvar() (*sdepy.process method*), 34

W

wiener_cdf (*class in sdepy*), 93
wiener_chf (*class in sdepy*), 93
wiener_mean (*class in sdepy*), 91
wiener_pdf (*class in sdepy*), 93
wiener_process (*class in sdepy*), 76
wiener_SDE (*class in sdepy*), 84
wiener_source (*class in sdepy*), 45
wiener_std (*class in sdepy*), 92
wiener_var (*class in sdepy*), 92

X

xcopy() (*sdepy.process method*), 33