
jumpdiff

Release 0.4.2

Feb 10, 2023

Contents

1	Installation	3
2	Jump-diffusion processes	5
2.1	The theory	5
2.2	Integrating a jump-diffusion process	5
2.3	Using <code>jumpdiff</code> to retrieve the parameters	6
2.3.1	Moments and Kramers–Moyal coefficients	6
2.3.2	Retrieving the jump-related terms	7
2.4	Distinguishing pure diffusions from jump-diffusions	8
3	Table of Content	11
3.1	Installation	11
3.2	Jump-diffusion processes	11
3.2.1	The theory	11
3.2.2	Integrating a jump-diffusion process	12
3.2.3	Using <code>jumpdiff</code> to retrieve the parameters	12
3.3	Distinguishing pure diffusions from jump-diffusions	14
3.4	Functions	15
3.4.1	Jump-diffusion timeseries generator	15
3.4.2	Moments	16
3.4.3	Parameters	18
3.4.4	Q-ratio	19
3.4.5	Formulae	19
3.4.6	Helping functions	20
3.5	License	21
3.6	Contact	21
4	Literature	23
5	Funding	25
	Python Module Index	27
	Index	29

`jumpdiff` is a `python` library with non-parametric Nadaraya—Watson estimators to extract the parameters of jump-diffusion processes. With `jumpdiff` one can extract the parameters of a jump-diffusion process from one-dimensional timeseries, employing both a kernel-density estimation method combined with a set on second-order corrections for a precise retrieval of the parameters for short timeseries.

CHAPTER 1

Installation

To install `jumpdiff` simply use

```
pip install jumpdiff
```

Then on your favourite editor just use

```
import jumpdiff as jd
```

The library depends on `numpy`, `scipy`, and `sympy`.

Jump-diffusion processes

We will show here how to: (1) generate trajectories of jump-diffusion processes; (2) retrieve the parameters from a single trajectory of a jump-diffusion process. Naturally, if we already had some data – maybe from a real-world recording of a stochastic process – we would simply look at estimating the parameters for this process.

2.1 The theory

Jump-diffusion processes¹, as the name suggest, are a mixed type of stochastic processes with a diffusive and a jump term. One form of these processes which is mathematically traceable is given by the [Stochastic Differential Equation](#)

$$dX(t) = a(x, t) dt + b(x, t) dW(t) + \xi dJ(t),$$

which has four main elements: a drift term $a(x, t)$, a diffusion term $b(x, t)$, linked with a Wiener process $W(t)$, a jump amplitude term $\xi(x, t)$, which is given by a Gaussian distribution $\mathcal{N}(0, \sigma_\xi^2)$ coupled with a jump rate λ , which is the rate of the Poissonian jumps $J(t)$. You can find a good review on this topic in Ref. 2.

2.2 Integrating a jump-diffusion process

Let us use the functions in `jumpdiff` to generate a jump-diffusion process, and subsequently retrieve the parameters. This is a good way to understand the usage of the integrator and the non-parametric retrieval of the parameters.

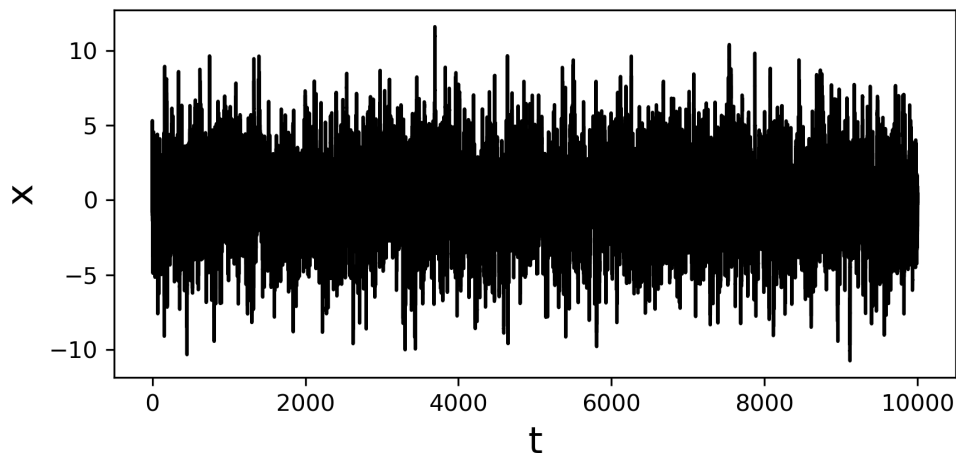
First we need to load our library. We will call it `jd`

```
1 import jumpdiff as jd
```

Let us thus define a jump-diffusion process and use `jd_process` to integrate it. Do notice here that we need the drift $a(x, t)$ and diffusion $b(x, t)$ as functions.

```
2 # integration time and time sampling
3 t_final = 10000
4 delta_t = 0.001
5
6 # A drift function
7 def a(x):
8     return -0.5*x
9
10 # and a (constant) diffusion term
11 def b(x):
12     return 0.75
13
14 # Now define a jump amplitude and rate
15 xi = 2.5
16 lamb = 1.75
17
18 # and simply call the integration function
19 X = jd.jd_process(t_final, delta_t, a=a, b=b, xi=xi, lamb=lamb)
```

This will generate a jump diffusion process X of length `int(10000/0.001)` with the given parameters.



2.3 Using jumpdiff to retrieve the parameters

2.3.1 Moments and Kramers–Moyal coefficients

Take the timeseries X and use the function `moments` to retrieve the conditional moments of the process. For now let us focus on the shortest time lag, so we can best approximate the Kramers–Moyal coefficients. For this case we can simply employ

```
20 edges, moments = jd.moments(timeseries = X)
```

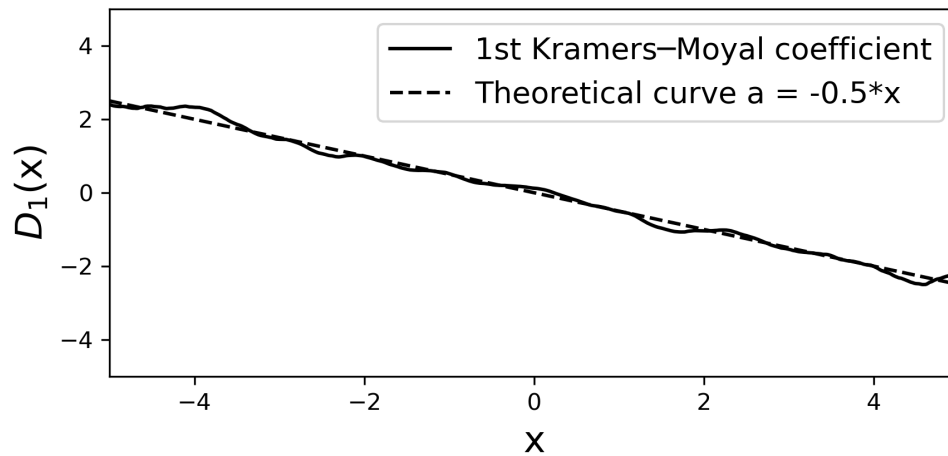
In the array `edges` are the limits of our space, and in our array `moments` are recorded all 6 powers/order of our conditional moments. Let us take a look at these before we proceed, to get acquainted with them.

We can plot the first moment with any conventional plotter, so let's use here `plotly` from `matplotlib`. To visualise the first moment, simply use

```

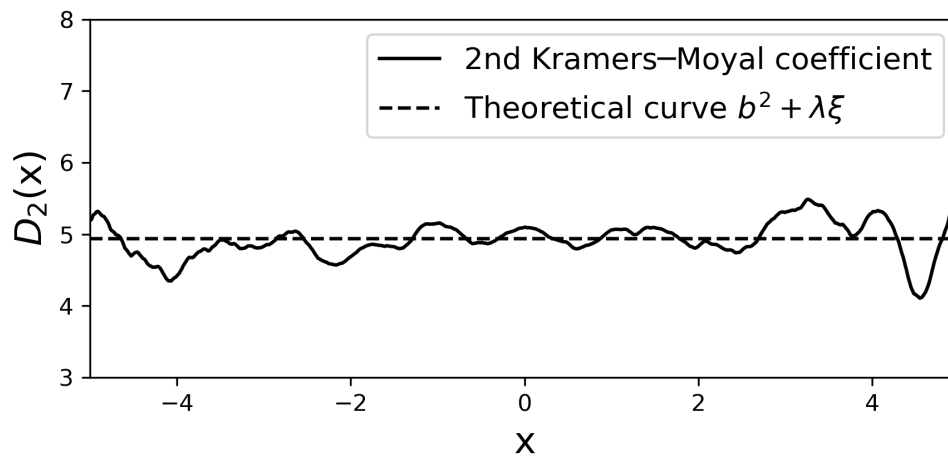
21 import matplotlib.pyplot as plt
22 plt.plot(edges, moments[1]/delta_t)

```



The first moment here (i.e., the first Kramers–Moyal coefficient) is given solely by the drift term that we have selected $-0.5 \cdot x$. In the plot we have also included the theoretical curve, which we know from having selected the value of $a(x)$ in line 8.

Similarly, we can extract the second moment (i.e., the second Kramers–Moyal coefficient) is a mixture of both the contributions of the diffusive term $b(x)$ and the jump terms ξ and λ .



You have this stored in `moments[2]`.

2.3.2 Retrieving the jump-related terms

Naturally one of the most pertinent questions when addressing jump-diffusion processes is the possibility of recovering these same parameters from data. For the given jump-diffusion process we can use the `jump_amplitude` and `jump_rate` functions to non-parametrically estimate the jump amplitude ξ and λ terms.

After having the `moments` in hand, all we need is

```
23 # first estimate the jump amplitude
24 xi_est = jd.jump_amplitude(moments = moments)
25
26 # and now estimated the jump rate
27 lamb_est = jd.jump_rate(moments = moments)
```

which resulted in our case in (xi_est) $\xi = 2.43 \pm 0.17$ and (lamb_est) $\lambda = 1.744 * \text{delta_t}$ (don't forget to divide lamb_est by delta_t)! We can compare these with our chose values in lines 15–16.

2.4 Distinguishing pure diffusions from jump-diffusions

One important question when we have some time series – possibly from real-world data – is to be able to discern if this timeseries is a pure diffusion process (a continuous stochastic process) or a jump-diffusion process (a discontinuous stochastic process). For this, jumpdiff has an easy to use function, called `q_ratio`. The idea behind distinguishing continuous and discontinuous processes is simple: diffusion processes *diffuse* over time, thus they take time to occupy space; jump-diffusion processes can jump, and thus statistically, they occupy all space very fast.

To analyse this let us design a simple example – with some numerically generated data – that shows the use of `q_ratio` and how to read it.

Let us generate two trajectories, using `jd_process`, denoted `d_timeseries` and `j_timeseries`, for *diffusion* timeseries and *jumpy* timeseries. Naturally the first must not include a jump term. To keep it simple, we will use the same parameters for both, expect for the jumps:

```
1 import jumpdiff as jd
2
3 # integration time and time sampling
4 t_final = 10000
5 delta_t = 0.01
6
7 # Drift function
8 def a(x):
9     return -0.5*x
10
11 # Diffusion function
12 def b(x):
13     return 0.75
14
15 # generate 2 trajectories
16 d_timeseries = jd.jd_process(t_final, delta_t, a=a, b=b, xi=0, lamb=0)
17 j_timeseries = jd.jd_process(t_final, delta_t, a=a, b=b, xi=2.5, lamb=1.75)
```

Note how `xi` and `lamb` are different for each process To now examine the rate of diffusion of the processes, we need to generate a time arrow, which we denote `lag`. This needs to be a integer list >0 .

```
18 import numpy as np
19 lag = np.logspace(0, 3, 25, dtype=int)
```

Lastly we just need to can the `q_ratio` for our two timeseries

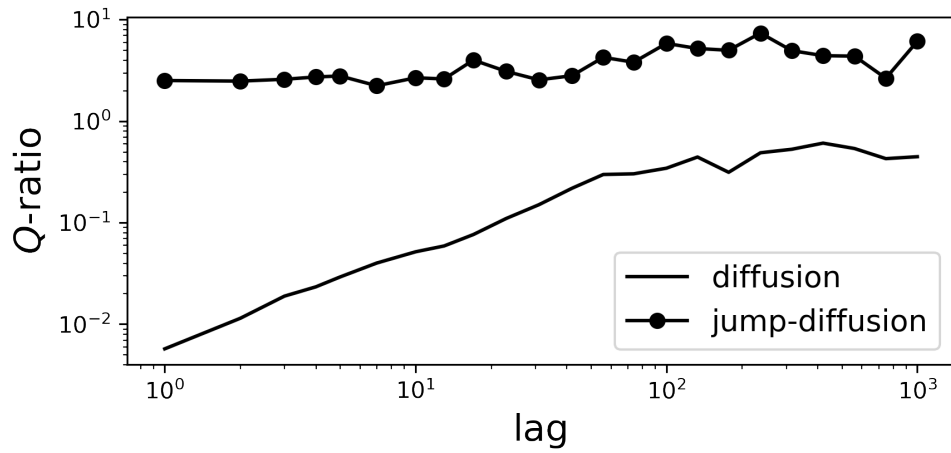
```
20 d_lag, d_Q = jd.q_ratio(lag, d_timeseries)
21 j_lag, j_Q = jd.q_ratio(lag, j_timeseries)
```

And with the help of matplotlib's `plotly`, we can visualise the results in a double logarithmic scale

```

22 import matplotlib.pyplot as plt
23
24 plt.loglog(d_lag, d_Q, '-', label='diffusion')
25 plt.loglog(j_lag, j_Q, 'o-', label='jump-diffusion')

```



As we can see, the diffusion process *grows* with our time arrow `lag`, where the jump-diffusion is constant (does not depend on `lag`). Jump processes will show a constant relation with code:`lag`, where diffusion processes a linear relation.

3.1 Installation

To install `jumpdiff` simply use

```
pip install jumpdiff
```

Then on your favourite editor just use

```
import jumpdiff as jd
```

The library depends on `numpy`, `scipy`, and `sympy`.

3.2 Jump-diffusion processes

We will show here how to: (1) generate trajectories of jump-diffusion processes; (2) retrieve the parameters from a single trajectory of a jump-diffusion process. Naturally, if we already had some data – maybe from a real-world recording of a stochastic process – we would simply look at estimating the parameters for this process.

3.2.1 The theory

Jump-diffusion processes¹, as the name suggest, are a mixed type of stochastic processes with a diffusive and a jump term. One form of these processes which is mathematically traceable is given by the [Stochastic Differential Equation](#)

$$dX(t) = a(x, t) dt + b(x, t) dW(t) + \xi dJ(t),$$

which has four main elements: a drift term $a(x, t)$, a diffusion term $b(x, t)$, linked with a Wiener process $W(t)$, a jump amplitude term $\xi(x, t)$, which is given by a Gaussian distribution $\mathcal{N}(0, \sigma_\xi^2)$ coupled with a jump rate λ , which is the rate of the Poissonian jumps $J(t)$. You can find a good review on this topic in Ref. 2.

3.2.2 Integrating a jump-diffusion process

Let us use the functions in `jumpdiff` to generate a jump-diffusion process, and subsequently retrieve the parameters. This is a good way to understand the usage of the integrator and the non-parametric retrieval of the parameters.

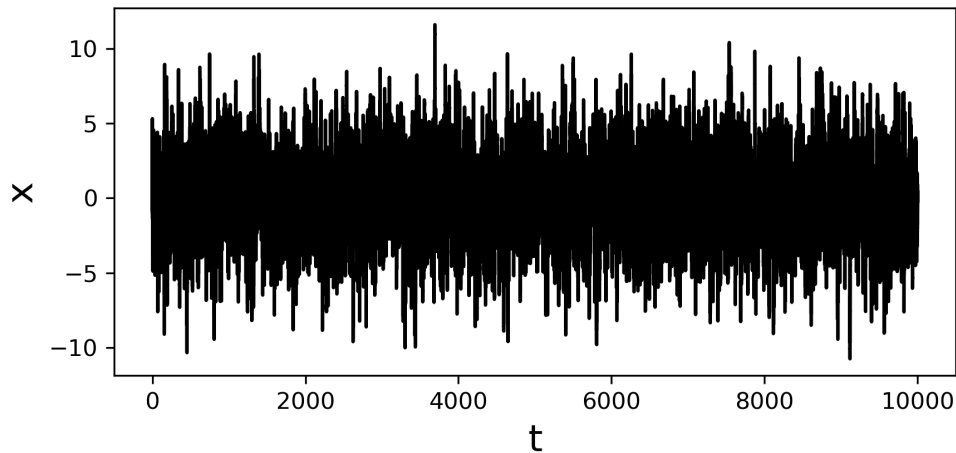
First we need to load our library. We will call it `jd`

```
1 import jumpdiff as jd
```

Let us thus define a jump-diffusion process and use `jd_process` to integrate it. Do notice here that we need the drift $a(x, t)$ and diffusion $b(x, t)$ as functions.

```
2 # integration time and time sampling
3 t_final = 10000
4 delta_t = 0.001
5
6 # A drift function
7 def a(x):
8     return -0.5*x
9
10 # and a (constant) diffusion term
11 def b(x):
12     return 0.75
13
14 # Now define a jump amplitude and rate
15 xi = 2.5
16 lamb = 1.75
17
18 # and simply call the integration function
19 X = jd.jd_process(t_final, delta_t, a=a, b=b, xi=xi, lamb=lamb)
```

This will generate a jump diffusion process X of length `int(10000/0.001)` with the given parameters.



3.2.3 Using `jumpdiff` to retrieve the parameters

Moments and Kramers–Moyal coefficients

Take the timeseries X and use the function `moments` to retrieve the conditional moments of the process. For now let us focus on the shortest time lag, so we can best approximate the Kramers–Moyal coefficients. For this case we can

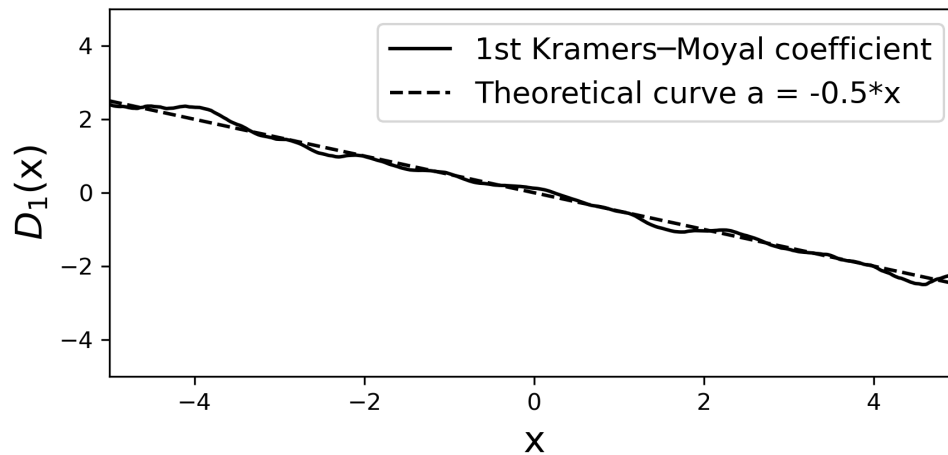
simply employ

```
20 edges, moments = jd.moments(timeseries = X)
```

In the array `edges` are the limits of our space, and in our array `moments` are recorded all 6 powers/order of our conditional moments. Let us take a look at these before we proceed, to get acquainted with them.

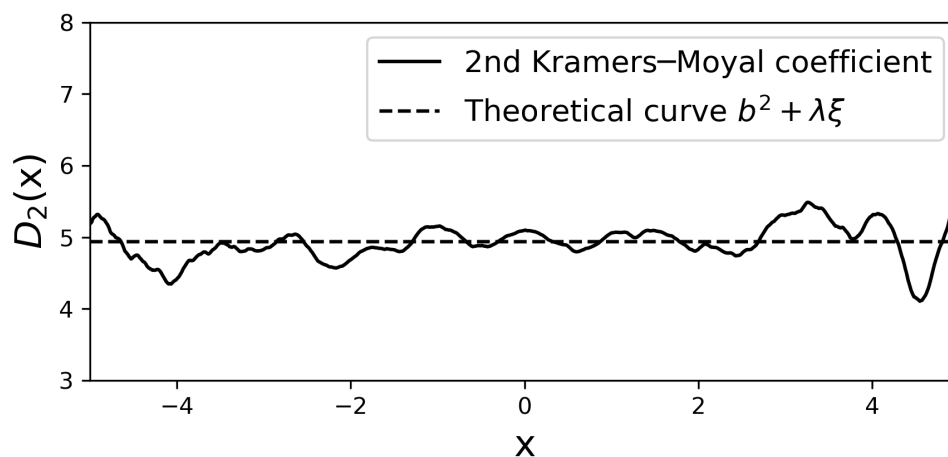
We can plot the first moment with any conventional plotter, so lets use here `plotly` from `matplotlib`. To visualise the first moment, simply use

```
21 import matplotlib.pyplot as plt
22 plt.plot(edges, moments[1]/delta_t)
```



The first moment here (i.e., the first Kramers—Moyal coefficient) is given solely by the drift term that we have selected $-0.5 \cdot x$. In the plot we have also included the theoretical curve, which we know from having selected the value of $a(x)$ in line 8.

Similarly, we can extract the second moment (i.e., the second Kramers—Moyal coefficient) is a mixture of both the contributions of the diffusive term $b(x)$ and the jump terms ξ and λ .



You have this stored in `moments[2]`.

Retrieving the jump-related terms

Naturally one of the most pertinent questions when addressing jump-diffusion processes is the possibility of recovering these same parameters from data. For the given jump-diffusion process we can use the `jump_amplitude` and `jump_rate` functions to non-parametrically estimate the jump amplitude ξ and λ terms.

After having the `moments` in hand, all we need is

```
23 # first estimate the jump amplitude
24 xi_est = jd.jump_amplitude(moments = moments)
25
26 # and now estimated the jump rate
27 lamb_est = jd.jump_rate(moments = moments)
```

which resulted in our case in `(xi_est)` $\xi = 2.43 \pm 0.17$ and `(lamb_est)` $\lambda = 1.744 * \text{delta_t}$ (don't forget to divide `lamb_est` by `delta_t`)! We can compare these with our chose values in lines 15–16.

3.3 Distinguishing pure diffusions from jump-diffusions

One important question when we have some time series – possibly from real-world data – is to be able to discern if this timeseries is a pure diffusion process (a continuous stochastic process) or a jump-diffusion process (a discontinuous stochastic process). For this, `jumpdiff` has an easy to use function, called `q_ratio`. The idea behind distinguishing continuous and discontinuous processes is simple: diffusion processes *diffuse* over time, thus they take time to occupy space; jump-diffusion processes can jump, and thus statistically, they occupy all space very fast.

To analyse this let us design a simple example – with some numerically generated data – that shows the use of `q_ratio` and how to read it.

Let us generate two trajectories, using `jd_process`, denoted `d_timeseries` and `j_timeseries`, for *diffusion* timeseries and *jumpy* timeseries. Naturally the first must not include a jump term. To keep it simple, we will use the same parameters for both, expect for the jumps:

```
1 import jumpdiff as jd
2
3 # integration time and time sampling
4 t_final = 10000
5 delta_t = 0.01
6
7 # Drift function
8 def a(x):
9     return -0.5*x
10
11 # Diffusion function
12 def b(x):
13     return 0.75
14
15 # generate 2 trajectories
16 d_timeseries = jd.jd_process(t_final, delta_t, a=a, b=b, xi=0, lamb=0)
17 j_timeseries = jd.jd_process(t_final, delta_t, a=a, b=b, xi=2.5, lamb=1.75)
```

Note how `xi` and `lamb` are different for each process To now examine the rate of diffusion of the processes, we need to generate a time arrow, which we denote `lag`. This needs to be a integer list >0 .

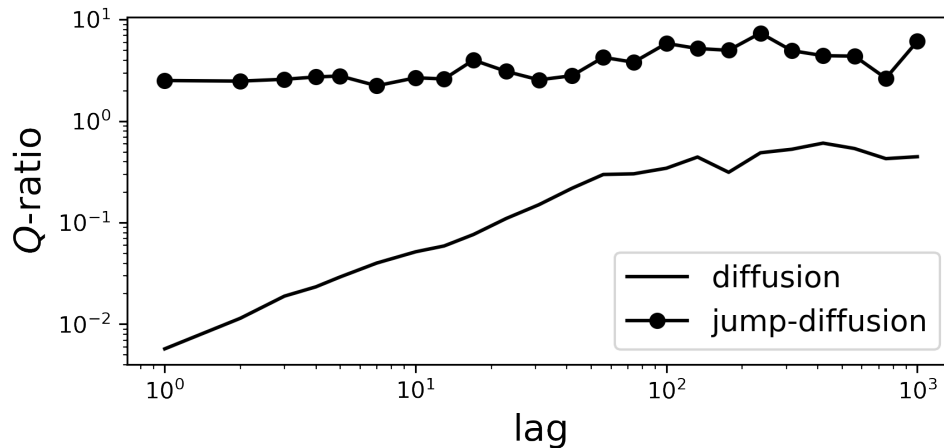
```
18 import numpy as np
19 lag = np.logspace(0, 3, 25, dtype=int)
```

Lastly we just need to can the `q_ratio` for our two timeseries

```
20 d_lag, d_Q = jd.q_ratio(lag, d_timeseries)
21 j_lag, j_Q = jd.q_ratio(lag, j_timeseries)
```

And with the help of matplotlib's `plotly`, we can visualise the results in a double logarithmic scale

```
22 import matplotlib.pyplot as plt
23
24 plt.loglog(d_lag, d_Q, '-', label='diffusion')
25 plt.loglog(j_lag, j_Q, 'o-', label='jump-diffusion')
```



As we can see, the diffusion process *grows* with our time arrow `lag`, where the jump-diffusion is constant (does not depend on `lag`). Jump processes will show a constant relation with code: `lag`, where diffusion processes a linear relation.

3.4 Functions

Documentation for all the functions in `jumpdiff`.

3.4.1 Jump-diffusion timeseries generator

`jumpdiff.jd_process.jd_process` (*time*: float, *delta_t*: float, *a*: callable, *b*: callable, *xi*: float, *lamb*: float, *init*: float = None, *solver*: str = 'Euler', *b_prime*: callable = None) → `numpy.ndarray`

Integrates a jump-diffusion process with drift $a(x)$, diffusion $b(x)$, jump amplitude ξ (ξ), and jump rate λ (λ).

$$dX(t) = a(x, t) dt + b(x, t) dW(t) + \xi dJ(t),$$

with J Poisson with jump rate λ . This integrator has both an Euler–Maruyama and a Milstein method of integration. For Milstein one has to introduce the derivative of the diffusion term b , denoted `b_prime`.

Parameters

- **time** (*float* > 0) – Total integration time. Positive float or int.
- **delta_t** (*float* > 0) – Time sampling, the smaller the better.

- **a** (*callable*) – The drift function. Can be a function of a `lambda`. For an Ornstein–Uhlenbeck process with drift $-2x$, `a` takes the form

```
a = lambda x: -2x.
```

- **b** (*callable*) – The diffusion function. Can be a function of a `lambda`. For an Ornstein–Uhlenbeck process with diffusion 1, `a` takes the form

```
b = lambda x: 1.
```

- **xi** (*float* > 0) – Variance of the jump amplitude, which will be turned into a normal distribution like $\mathcal{N}(0, \text{xi})$.
- **lamb** (*float* > 0) – Jump rate of the Poissonian jumps. This is implemented as the numpy function `np.random.poisson(lam = lamb * delta_t)`.
- **init** (*float* (default `None`)) – Initial conditions. If `None` given, generates a random value from a normal distribution $\sim \mathcal{N}(0, \text{delta_t})$.
- **solver** (*'Euler' or 'Milstein' (default 'Euler')*) – The regular Euler–Maruyama solver ‘Euler’ is the default, with an order of `delta_t`. To employ a state-dependent diffusion, i.e., `b(x)` as a function of `x`, the Milstein scheme has an order of `delta_t`. You must introduce as well the derivative of `b(x)`, i.e., `b'(x)`, as the argument `b_prime`.

Returns `X` – Timeseries of size `int(time/delta_t)`

Return type `np.array`

3.4.2 Moments

`jumpdiff.moments.moments` (*timeseries: numpy.ndarray, bw: float = None, bins: numpy.ndarray = None, power: int = 6, lag: list = [1], correction: bool = True, norm: bool = False, kernel: callable = None, tol: float = 1e-10, conv_method: str = 'auto', verbose: bool = False*) \rightarrow `numpy.ndarray`

Estimates the moments of the Kramers–Moyal expansion from a timeseries using a Nadaraya–Watson kernel estimator method. These later can be turned into the drift and diffusion coefficients after normalisation.

Parameters

- **timeseries** (*np.ndarray*) – A 1-dimensional timeseries.
- **bw** (*float*) – Desired bandwidth of the kernel. A value of 1 occupies the full space of the bin space. Recommended are values $0.005 < \text{bw} < 0.4$.
- **bins** (*np.ndarray* (default `None`)) – The number of bins for each dimension, defaults to `np.array([5000])`. This is the underlying space for the Kramers–Moyal conditional moments.
- **power** (*int* (default 6)) – Upper limit of the the Kramers–Moyal conditional moments to calculate. It will generate all Kramers–Moyal conditional moments up to power.
- **lag** (*list* (default 1)) – Calculates the Kramers–Moyal conditional moments at each indicated lag, i.e., for `timeseries[:,lag[]]`. Defaults to 1, the shortest timestep in the data.
- **corrections** (*bool* (default `True`)) – Implements the second-order corrections of the Kramers–Moyal conditional moments directly
- **norm** (*bool* (default `False`)) – Sets the normalisation. `False` returns the Kramers–Moyal conditional moments, and `True` returns the Kramers–Moyal coefficients.

- **kernel** (callable (default `None`)) – Kernel used to convolute with the Kramers–Moyal conditional moments. To select example an Epanechnikov kernel use

```
kernel = kernels.epanechnikov
```

If `None` the Epanechnikov kernel will be used.

- **tol** (float (default `1e-10`)) – Round to zero absolute values smaller than `tol`, after convolutions.
- **conv_method** (str (default `auto`)) – A string indicating which method to use to calculate the convolution. docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve.
- **verbose** (bool (default `False`)) – If `True` will report on the bandwidth used.

Returns

- **edges** (*np.ndarray*) – The bin edges with shape $(D, \text{bins.shape})$ of the calculated moments.
- **moments** (*np.ndarray*) – The calculated moments from the Kramers–Moyal expansion of the timeseries at each lag. To extract the selected orders of the moments, use `moments[i, :, j]`, with *i* the order according to powers, *j* the lag (if any given).

`jumpdiff.moments.corrections` (*m: numpy.ndarray, power: int*)

The moments function will by default apply the corrections. You can turn the corrections off in that fuction by setting `corrections = False`.

Second-order corrections of the Kramers–Moyal coefficients (conditional moments), given by

$$\begin{aligned}
 F_1 &= M_1, \\
 F_2 &= \frac{1}{2}(M_2 - M_1^2), \\
 F_3 &= \frac{1}{6}(M_3 - 3M_1M_2 + 3M_1^3), \\
 F_4 &= \frac{1}{24}(M_4 - 4M_1M_3 + 18M_1^2M_2 - 3M_2^2 - 15M_1^4), \\
 F_5 &= \frac{1}{120}(M_5 - 5M_1M_4 + 30M_1^2M_3 - 150M_1^3M_2 + 45M_1M_2^2 - 10M_2M_3 + 105M_1^5), \\
 F_6 &= \frac{1}{720}(M_6 - 6M_1M_5 + 45M_1^2M_4 - 300M_1^3M_3 + 1575M_1^4M_2 - 675M_1^2M_2^2 \\
 &\quad + 180M_1M_2M_3 + 45M_2^3 - 15M_2M_4 - 10M_3^2 - 945M_1^6),
 \end{aligned}$$

with the prefactor the normalisation, i.e., the normalised results are the Kramers–Moyal coefficients. If `norm` is `False`, this results in the Kramers–Moyal conditional moments.

Parameters

- **(moments)** (*m*) – The calculated conditional moments from the Kramers–Moyal expansion of the at each lag. To extract the selected orders of the moments use `moments[i, :, j]`, with *i* the order according to powers, *j* the lag.
- **power** (*int*) – Upper limit of the Kramers–Moyal conditional moments to calculate. It will generate all Kramers–Moyal conditional moments up to power.

Returns F – The corrections of the calculated Kramers–Moyal conditional moments from the Kramers–Moyal expansion of the timeseries at each lag. To extract the selected orders of the moments, use `F[i, :, j]`, with *i* the order according to powers, *j* the lag (if any introduced).

Return type `np.ndarray`

3.4.3 Parameters

`jumpdiff.parameters.jump_amplitude` (*moments: numpy.ndarray, tol: float = 1e-10, full: bool = False, verbose: bool = False*) \rightarrow `numpy.ndarray`

Retrieves the jump amplitude ξ (ξ) via

$$\lambda(x, t) = \frac{M_4(x, t)}{3\sigma_\xi^4}.$$

Take notice that the different normalisation of the `moments` leads to a different results.

Parameters

- **moments** (`np.ndarray`) – Moments extracted with the function `moments`. Needs moments up to order 6.
- **tol** (float (default `1e-10`)) – Toleration for the division of the moments.
- **full** (bool (default `False`)) – If `True` returns also the (biased) weighed standard deviation of the averaging process.
- **verbose** (bool (default `True`)) – Prints the result.

Returns `xi_est` – Estimator of the jump amplitude ξ (ξ).

Return type `np.ndarray`

References

Anvari, M., Tabar, M. R. R., Peinke, J., Lehnertz, K., ‘Disentangling the stochastic behavior of complex time series.’ Scientific Reports, 6, 35435, 2016. doi: 10.1038/srep35435.

Lehnertz, K., Zabawa, L., and Tabar, M. R. R., ‘Characterizing abrupt transitions in stochastic dynamics.’ New Journal of Physics, 20(11):113043, 2018. doi: 10.1088/1367-2630/aaf0d7.

`jumpdiff.parameters.jump_rate` (*moments: numpy.ndarray, xi_est: numpy.ndarray = None, tol: float = 1e-10, full: bool = False, verbose: bool = False*) \rightarrow `numpy.ndarray`

Retrieves the jump rate λ (λ) via

$$\sigma_\xi^2 = \frac{M_6(x, t)}{5M_4(x, t)}.$$

Take notice that the different normalisation of the `moments` leads to a different results.

Parameters

- **moments** (`np.ndarray`) – moments extracted with the function ‘`moments`’. Needs moments of order 6.
- **tol** (float (default `1e-10`)) – Toleration for the division of the moments.
- **full** (bool (default `False`)) – If `True` returns also the (biased) weighed standard deviation of the averaging process.
- **verbose** (bool (default `True`)) – Prints the result.

Returns `xi_est` – Estimator on the jump rate λ (λ)

Return type `np.ndarray`

References

Anvari, M., Tabar, M. R. R., Peinke, J., Lehnertz, K., ‘Disentangling the stochastic behavior of complex time series.’ Scientific Reports, 6, 35435, 2016. doi: 10.1038/srep35435.

Lehnertz, K., Zabawa, L., and Tabar, M. R. R., ‘Characterizing abrupt transitions in stochastic dynamics.’ New Journal of Physics, 20(11):113043, 2018. doi: 10.1088/1367-2630/aaf0d7.

3.4.4 Q-ratio

`jumpdiff.q_ratio.q_ratio` (*lag: numpy.ndarray, timeseries: numpy.ndarray, loc: int = None, correction: bool = False*) \rightarrow `numpy.ndarray`

`q_ratio` method to distinguish pure diffusion from jump-diffusion timeseries, Given by the relation of the 4th and 6th Kramers–Moyal coefficient with increasing lag

$$Q(x, \tau) = \frac{D_6(x, \tau)}{5D_4(x, \tau)} = \begin{cases} b(x)^2\tau, & \text{diffusive} \\ \sigma_\xi^2(x), & \text{jumpy} \end{cases}$$

Parameters

- **lag** (*np.ndarray of ints*) – An array with the time-lag to extract the Kramers–Moyal coefficient for different lags.
- **timeseries** (*np.ndarray*) – A 1-dimensional timeseries.
- **loc** (float (default `None`)) – Use a particular point in space to calculate the ratio. If `None` given, the maximum of the probability density function is taken.
- **corrections** (bool (default `False`)) – Select whether to use corrective terms.

Returns

- **lag** (*np.ndarray of ints*) – Same as input, but only lag > 0 and as ints.
- **ratio** (*np.ndarray of len(lag)*) – Ratio of the sixth-order over forth-order Kramers–Moyal coefficient.

References

Anvari, M., Tabar, M. R. R., Peinke, J., Lehnertz, K., ‘Disentangling the stochastic behavior of complex time series.’ Scientific Reports, 6, 35435, 2016. doi: 10.1038/srep35435.

Lehnertz, K., Zabawa, L., and Tabar, M. R. R., ‘Characterizing abrupt transitions in stochastic dynamics.’ New Journal of Physics, 20(11):113043, 2018. doi: 10.1088/1367-2630/aaf0d7.

3.4.5 Formulae

`jumpdiff.formulae.m_formula` (*power, tau=True*)

Generate the formula for the conditional moments with second-order corrections based on the relation with the ordinary Bell polynomials

$$M_n(x', \tau) \sim (n!) \tau D_n(x') + \frac{(n!) \tau^2}{2} \sum_{m=1}^{n-1} D_m(x') D_{n-m}(x')$$

Parameters **power** (*int*) – Desired order of the formula.

Returns **term** – Expression up to given power.

Return type sympy.symbols

`jumpdiff.formulae.f_formula(power)`

Generate the formula for the conditional moments with second-order corrections based on the relation with the ordinary Bell polynomials

$$D_n(x) = \frac{1}{\tau(n!)} \left[\hat{B}_{n,1}(M_1(x, \tau), M_2(x, \tau), \dots, M_n(x, \tau)) - \frac{\tau}{2} \hat{B}_{n,2}(M_1(x, \tau), M_2(x, \tau), \dots, M_{n-1}(x, \tau)) \right].$$

Parameters **power** (*int*) – Desired order of the formula.

Returns **term** – Expression up to given power.

Return type sympy.symbols

`jumpdiff.formulae.f_formula_solver(power)`

Generate the reciprocal relation of the moments to the Kramers–Moyal coefficients by sequential iteration.

$$D_n(x) = \frac{1}{\tau(n!)} \left[\hat{B}_{n,1}(M_1(x, \tau), M_2(x, \tau), \dots, M_n(x, \tau)) - \frac{\tau}{2} \hat{B}_{n,2}(M_1(x, \tau), M_2(x, \tau), \dots, M_{n-1}(x, \tau)) \right].$$

Parameters **power** (*int*) – Desired order of the formula.

Returns **term** – Expression up to given power.

Return type sympy.symbols

3.4.6 Helping functions

Kernels function

`jumpdiff.kernels.kernel(kernel_func)`

Transforms a kernel function into a scaled kernel function (for a certain bandwidth bw).

Currently implemented kernels are: Epanechnikov, Gaussian, Uniform, Triangular, Quartic.

For a good overview of various kernels see [https://en.wikipedia.org/wiki/Kernel_\(statistics\)](https://en.wikipedia.org/wiki/Kernel_(statistics))

`jumpdiff.kernels.volume_unit_ball(dims: int) → float`

Returns the volume of a unit ball in dimensions dims.

`jumpdiff.kernels.epanechnikov(x: numpy.ndarray, dims: int) → numpy.ndarray`

The Epanechnikov kernel in dimensions dims.

`jumpdiff.kernels.gaussian(x: numpy.ndarray, dims: int) → numpy.ndarray`

Gaussian kernel in dimensions dims.

`jumpdiff.kernels.uniform(x: numpy.ndarray, dims: int) → numpy.ndarray`

Uniform, or rectangular kernel in dimensions dims

`jumpdiff.kernels.triangular(x: numpy.ndarray, dims: int) → numpy.ndarray`

Triangular kernel in dimensions dims

`jumpdiff.kernels.quartic(x: numpy.ndarray, dims: int) → numpy.ndarray`

Quartic, or biweight kernel in dimensions dims

3.5 License

MIT License

Copyright (c) 2019-2021 Leonardo Rydin Gorjão

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3.6 Contact

If you need help with something, find a bug, issue, or typo on the repository or in the code, you can contact me here: leonardo.rydin@gmail.com or open an issue on the GitHub repository.

¹ Tabar, M. R. R. *Analysis and Data-Based Reconstruction of Complex Nonlinear Dynamical Systems*. Springer, International Publishing (2019), Chapter [Stochastic Processes with Jumps and Non-vanishing Higher-Order Kramers—Moyal Coefficients*](#).

² Friedrich, R., Peinke, J., Sahimi, M., Tabar, M. R. R. *Approaching complexity by stochastic methods: From biological systems to turbulence*, [Physics Reports](#) 506, 87–162 (2011).

³ Rydin Gorjão, L., Meirinhos, F. *kramersmoyal: Kramers–Moyal coefficients for stochastic processes*. [Journal of Open Source Software](#), 4(44) (2019).

An extensive review on the subject can be found [here](#).

CHAPTER 5

Funding

Helmholtz Association Initiative *Energy System 2050 - A Contribution of the Research Field Energy* and the grant No. VH-NG-1025 and *STORM - Stochastics for Time-Space Risk Models* project of the Research Council of Norway (RCN) No. 274410.

j

`jumpdiff.formulae`, [19](#)
`jumpdiff.kernels`, [20](#)

C

`corrections()` (in module *jumpdiff.moments*), 17

E

`epanechnikov()` (in module *jumpdiff.kernels*), 20

F

`f_formula()` (in module *jumpdiff.formulae*), 20

`f_formula_solver()` (in module *jumpdiff.formulae*), 20

G

`gaussian()` (in module *jumpdiff.kernels*), 20

J

`jd_process()` (in module *jumpdiff.jd_process*), 15

`jump_amplitude()` (in module *jumpdiff.parameters*), 18

`jump_rate()` (in module *jumpdiff.parameters*), 18

`jumpdiff.formulae` (module), 19

`jumpdiff.jd_process` (module), 15

`jumpdiff.kernels` (module), 20

`jumpdiff.moments` (module), 16

`jumpdiff.parameters` (module), 18

`jumpdiff.q_ratio` (module), 19

K

`kernel()` (in module *jumpdiff.kernels*), 20

M

`m_formula()` (in module *jumpdiff.formulae*), 19

`moments()` (in module *jumpdiff.moments*), 16

Q

`q_ratio()` (in module *jumpdiff.q_ratio*), 19

`quartic()` (in module *jumpdiff.kernels*), 20

T

`triangular()` (in module *jumpdiff.kernels*), 20

U

`uniform()` (in module *jumpdiff.kernels*), 20

V

`volume_unit_ball()` (in module *jumpdiff.kernels*), 20