Diego Barba Follow

May 3 · 7 min read ★ · Listen

Stochastic Processes Simulation — The Ornstein Uhlenbeck Process

Part 2 of the Stochastic Processes Simulation series. Simulating the Ornstein-Uhlenbeck process, the canonical stationary process, in Python.

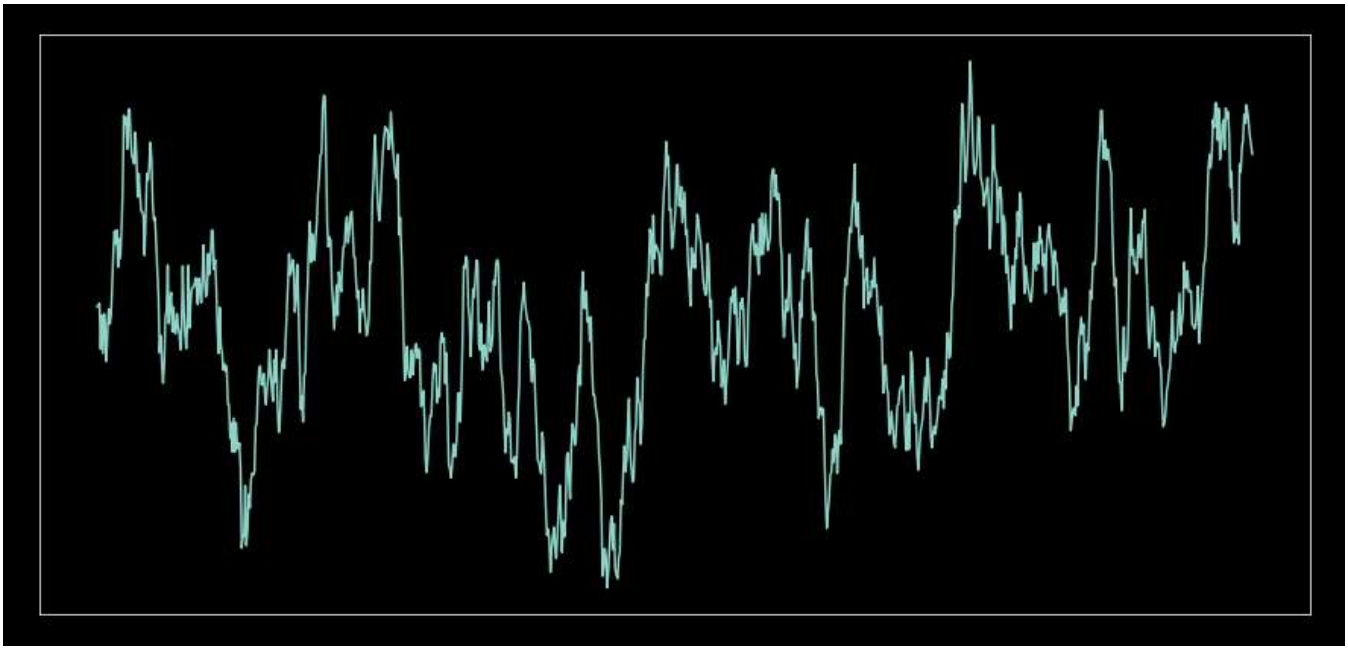

Image by author.

The quest for stationarity. Doesn't matter if we want to perform statistical inference or fit a machine learning model, we always seem to be searching for stationary features, stationary explanatory variables and well, stationary everything.

In this story, we will discuss the **Ornstein-Uhlenbeck process**. The canonical process among the stationary stochastic processes. **This process is not just stationary but also normally distributed**. If only time-series features behaved like this.

In the upcoming sections, we will simulate the Ornstein-Uhlenbeck process, learn how to estimate its parameters from data, and lastly, simulate multiple correlated processes. The idea is that by the end of this story you can take with you a complete neat mini-library for Ornstein-Uhlenbeck simulations.

Throughout the story, we will use tools and concepts introduced in the first story of the series about Brownian motion. If you already know the basics about Brownian motion just check out the code, as we will use the Brownian motions simulation methods outlined there. Otherwise, give it a thorough read first and then come back.

Table of contents

The equation of the process

The Ornstein-Uhlenbeck (OU for short) process satisfies the stochastic differential equation (SDE):

$$dX_t = \alpha(\gamma - X_t)dt + \beta dW_t$$

where *W_t* is a Brownian motion, α and β are positive constants.

The deterministic part (the drift of the process) which is the time differential term is what causes the mean reversion. When X_t is larger than γ (the asymptotic mean), the

β controls the random shocks of the process. Note that for very high values of β, the process is, asymptotically, a rescaled Brownian motion as the drift no longer has a significant effect.

There are two main reasons why this process is used as the canonical process for stationary processes:

1. The stochastic differential equation is integrable in closed-form (and we love that).
2. The process path is normally distributed. So if Brownian motion is at the center of stochastic processes because its increments are normally distributed, then the OU process is at the center of mean-reverting processes because its path is normally distributed.

The solution to the SDE is:

$$X_t = X_0 e^{-\alpha t} + \gamma(1 - e^{-\alpha t}) + \beta e^{-\alpha t} \int_0^t e^{\alpha s} dW_s$$

its asymptotic mean and variance are γ and $\beta^2 / 2\alpha$ respectively. Hence, the distribution of the process is $N(\gamma, \beta^2 / 2\alpha)$.

Simulating the process

To simulate the process we need to convert the solution of the SDE to a discrete vectorial equation, each unit time of the process is an index of a vector.

We let t = (0, 1, 2, …, *T*-1), where T is the sample size. We build the arrays for the exponentials and then approximate the integral. To approximate the integral we use the cumulative sum. Note that this approximation for the integral is not suitable for very small sample sizes, so letting the sample size be greater than about 100 is a good idea.

Lets code this.

First we will define the parameters in a dataclass. Passing around many parameters often gets messy, it's cleaner to pass them in an object of their own:

```python
class OUParams:
    alpha: float  # mean reversion parameter
    gamma: float  # asymptotic mean
    beta: float  # Brownian motion scale (standard deviation)
```


To generate the OU process simulation we will use the code to generate Brownian motions from the first story of the series. Save the code as “brownian_motion.py”. If you don’t save it in the folder where you run the following code, you will have to change the import statement.

```python
from typing import Optional

import numpy as np

import brownian_motion


def get_OU_process(
    T: int,
    OU_params: OUParams,
    X_0: Optional[float] = None,
    random_state: Optional[int] = None,
) -> np.ndarray:
    """
    - T is the sample size.
    - Ou_params is an instance of OUParams dataclass.
    - X_0 the initial value for the process, if None, then X_0 is taken
        to be gamma (the asymptotic mean).
    Returns a 1D array.
    """
    t = np.arange(T, dtype=np.float128) # float to avoid np.exp overflow
    exp_alpha_t = np.exp(-OU_params.alpha * t)
    dW = brownian_motion.get_dW(T, random_state)
    integral_W = _get_integal_W(t, dW, OU_params)
    _X_0 = _select_X_0(X_0, OU_params)
    return (
        _X_0 * exp_alpha_t
        + OU_params.gamma * (1 - exp_alpha_t)
        + OU_params.beta * exp_alpha_t * integral_W
```

```
    if X_0_in is not None:
        return X_0_in
    return OU_params.gamma


def _get_integal_W(
    t: np.ndarray, dW: np.ndarray, OU_params: OUParams
) -> np.ndarray:
    """Integral with respect to Brownian Motion (W), ∫...dW."""
    exp_alpha_s = np.exp(OU_params.alpha * t)
    integral_W = np.cumsum(exp_alpha_s * dW)
    return np.insert(integral_W, 0, 0)[:-1]
```

Lets simulate an OU process:

```
OU_params = OUParams(alpha=0.07, gamma=0.0, beta=0.001)
OU_proc = get_OU_process(1_000, OU_params)

#-----------------------------------------------------
# plot
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(15, 7))

title = "Ornstein-Uhlenbeck process, "
title += r"$\alpha=0.07$, $\gamma = 0$, $\beta = 0.001$"
plt.plot(OU_proc)
plt.gca().set_title(title, fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
```

mean_reverting_procs_3.py hosted with ❤ by GitHub view raw

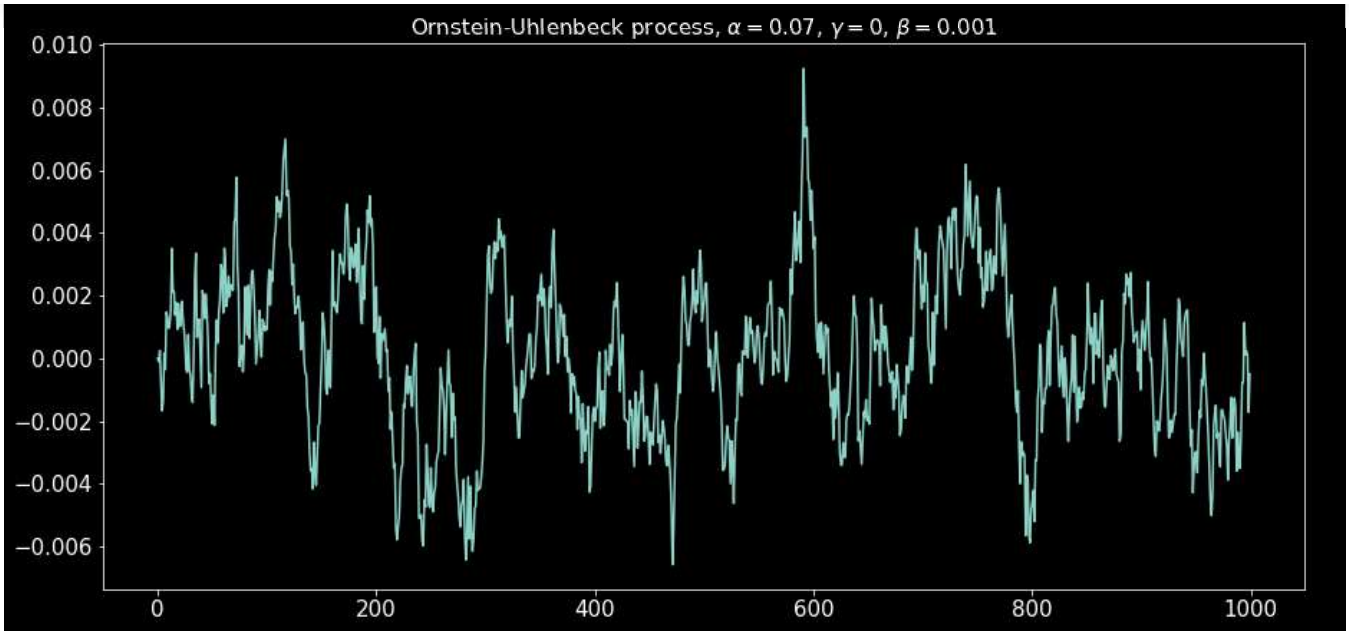

Image by author.

Estimating parameters from data

To estimate the OU parameters from a given process we use ordinary least squares (OLS) regression.

We approximate the stochastic differential equation by a discrete equation (the time series way), known as the Euler-Maruyama method (the random difference ΔX_t is taken as a forward difference):

$$\Delta X_t = \alpha\gamma\Delta t - \alpha X_t \Delta t + \beta\epsilon_t$$

furthermore, if we make $\Delta t = 1$, then:

$$\Delta X_t = \alpha\gamma - \alpha X_t + \beta\varepsilon_t$$

where ε is i.i.d standard normal.

Now it'a easy to see the regression specification, i.e. an equation of the form $y = a + b X + \varepsilon$.

The following function performs the regression and estimates the OU parameters:

```
    Estimate OU params from OLS regression.
    - X_t is a 1D array.
    Returns instance of OUParams.
    """
    y = np.diff(X_t)
    X = X_t[:-1].reshape(-1, 1)
    reg = LinearRegression(fit_intercept=True)
    reg.fit(X, y)
    # regression coeficient and constant
    alpha = -reg.coef_[0]
    gamma = reg.intercept_ / alpha
    # residuals and their standard deviation
    y_hat = reg.predict(X)
    beta = np.std(y - y_hat)
    return OUParams(alpha, gamma, beta)
```

mean_reverting_procs_4.py hosted with ❤ by GitHub view raw

Let's test it with a simulated process to see if we can recover the OU params.

```
# generate process with random_state to reproduce results
OU_params = OUParams(alpha=0.07, gamma=0.0, beta=0.001)
OU_proc = get_OU_process(100_000, OU_params, random_state=7)

OU_params_hat = estimate_OU_params(OU_proc)
```

mean_reverting_procs_5.py hosted with ❤ by GitHub view raw

We get:

alpha=0.06845699877932063, gamma=-3.9905935875610906e-05, beta=0.0009278569354622049 7,

the estimation is not perfect, but is good enough. To recover the true parameters we should estimate them in many samples, the averages will converge closer to the real parameters.

Correlated processes

As we saw in the first part of the series, Itô diffusive processes' random terms can be correlated. So naturally we want to have a way to create correlated OU processes. This will be useful in many models

There are two main functionalities that we may want here:

1. All processes have the same OU parameters and they are correlated.
2. Every process has different OU parameters but processes are correlated.

We achieve this by letting the "OU_params" argument in the following code take two different types. If it is an instance of "OUParams" then it's case 1; if it is a tuple of instances of "OUParams" then it's case 2.

```python
from typing import Optional, Union

import numpy as np


def get_corr_OU_procs(
    T: int,
    OU_params: Union[OUParams, tuple[OUParams, ...]],
    n_procs: Optional[int] = None,
    rho: Optional[float] = None,
    random_state: Optional[int] = None,
) -> np.ndarray:
    """
    Simulate correlated OU processes, correlation (rho) can be 0 or None.
    - T is the sample size of the processes.
    - OU_params can be a an instance of OUParams, in that case
        all processes have the same parameters. It can also be a tuple,
        in that case each process will have the parameters in the tuple,
        each column in the resulting 2D array corresponds to the tuple index.
    - n_procs is ignored if OU_params is tuple, else, corresponds to the number
        of processes desired. If OU_params is not tuple and n_procs is None, will
        raise ValueError.
    - rho is the correlation coefficient.
    - random_state to reproduce results.

    """
    _n_procs = _get_n_procs(OU_params, n_procs)
    corr_dWs = brownian_motion.get_corr_dW_matrix(
        T, _n_procs, rho, random_state
    )
    is_OU_params_tpl = _is_OU_params_tuple(OU_params)
```

```
    return np.asarray(OU_procs).T


def _is_OU_params_tuple(
    OU_params: Union[OUParams, tuple[OUParams, ...]]
) -> bool:
    """
    Check is OU_params is a tuple of params,
    return bool.
    """
    return isinstance(OU_params, tuple)


def _get_n_procs(
    OU_params: Union[OUParams, tuple[OUParams, ...]], n_procs: Optional[int]
) -> int:
    """
    Define the number of processes, if Ou_params is a tuple the
    number of processes is the lenght of the tuple. If it is not a tuple
    then it is the "n_procs" supplied as argument,
    if it is None will raise ValueError.
    """
    if _is_OU_params_tuple(OU_params):
        return len(OU_params)  # type: ignore
    elif n_procs is None:
        raise ValueError("If OU_params is not tuple, n_procs cannot be None.")
    return n_procs


def _get_OU_params_i(
    OU_params: Union[OUParams, tuple[OUParams, ...]],
    i: int,
    is_OU_params_tpl: bool,
) -> OUParams:
    """
    Returns the ith value of the OU_params tuple if it is a tuple,
    otherwise returns OUParams.
    """
    if is_OU_params_tpl:
        return OU_params[i]  # type: ignore
    return OU_params  # type: ignore
```

```
    Simulates the OU process with an external dW.
    X_0 is taken as the asymptotic mean gamma for simplicity.
    """
    t = np.arange(T, dtype=np.float128)  # float to avoid np.exp overflow
    exp_alpha_t = np.exp(-OU_params.alpha * t)
    integral_W = _get_integal_W(t, dW, OU_params)
    return (
        OU_params.gamma * exp_alpha_t
        + OU_params.gamma * (1 - exp_alpha_t)
        + OU_params.beta * exp_alpha_t * integral_W
```

```
# case 1

T = 1_000
OU_params = OUParams(alpha=0.07, gamma=0.0, beta=0.001)
n_proc = 5
rho = 0.9
OU_procs = get_corr_OU_procs(T, OU_params, n_proc, rho)

#-----------------------------------------------------
# plot
import matplotlib.pyplot as plt
import seaborn as sns

fig = plt.figure(figsize=(15, 5))

title = "Correlated Ornstein-Uhlenbeck processes, single params"
plt.subplot(1, 2, 1)
plt.plot(OU_procs)
plt.gca().set_title(title, fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)

title = "Correlation matrix (increments) heatmap"
plt.subplot(1, 2, 2)
sns.heatmap(np.corrcoef(np.diff(OU_procs, axis=0), rowvar=False), cmap="mako")
plt.gca().set_title(title, fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
```

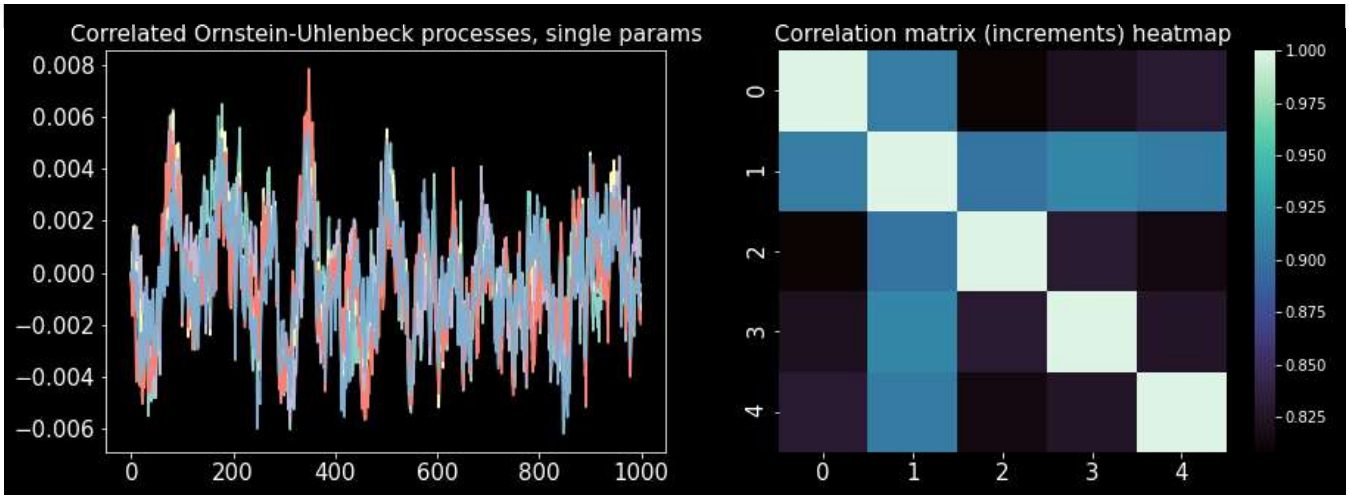

Image by author.

```python
# case 2

T = 1_000
OU_params = (
    OUParams(alpha=0.07, gamma=0.0, beta=0.005),
    OUParams(alpha=0.05, gamma=0.0, beta=0.003),
    OUParams(alpha=0.06, gamma=0.0, beta=0.002),
    OUParams(alpha=0.09, gamma=0.0, beta=0.002),
    OUParams(alpha=0.08, gamma=0.0, beta=0.001),
)
rho = 0.9
OU_procs = get_corr_OU_procs(T, OU_params, n_proc, rho)

#-----------------------------------------------------
# plot
import matplotlib.pyplot as plt
import seaborn as sns

fig = plt.figure(figsize=(15, 5))

title = "Correlated Ornstein-Uhlenbeck processes, multi params"
plt.subplot(1, 2, 1)
plt.plot(OU_procs)
plt.gca().set_title(title, fontsize=15)
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)

title = "Correlation matrix (increments) heatmap"
```

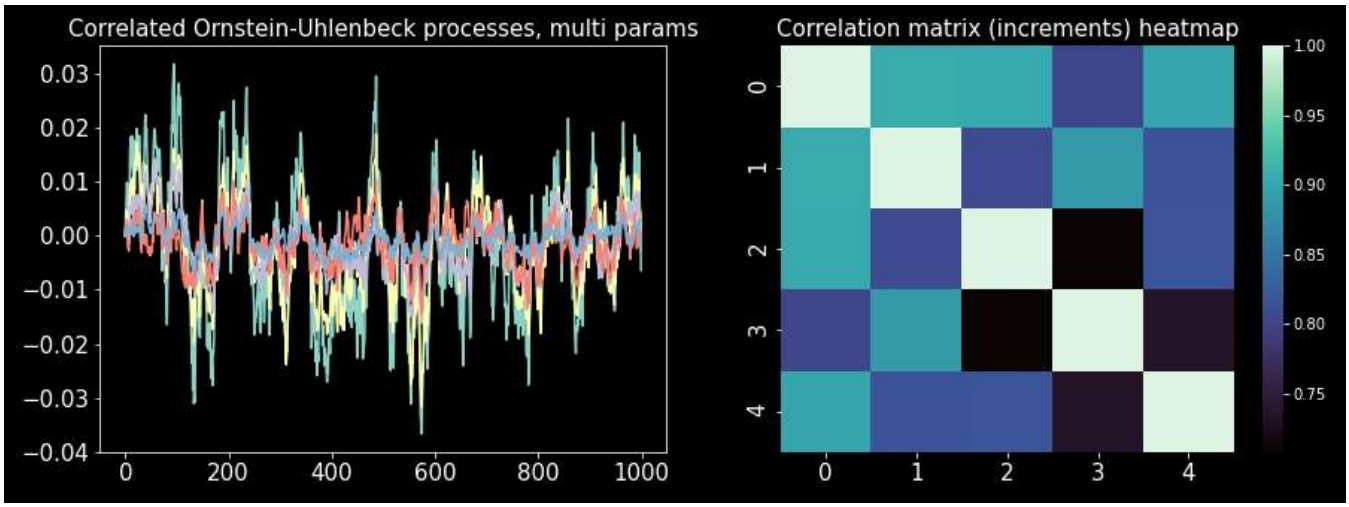

Image by author.

Code summary

As usual, I like to wrap up the story with a summary of all the code used throughout the sections and produce a single coherent mini-library that we can save for later use.

```
"""OU_proc.py"""

from dataclasses import dataclass
from typing import Optional, Union

import numpy as np

from sklearn.linear_model import LinearRegression

import brownian_motion

# -----------------------------------------------------------------
# OU process


@dataclass
class OUParams:
    alpha: float  # mean reversion parameter
    gamma: float  # asymptotic mean
    beta: float  # Brownian motion scale (standard deviation)

```

```python
    random_state: Optional[int] = None,
) -> np.ndarray:
    """
    - T is the sample size.
    - Ou_params is an instance of OUParams dataclass.
    - X_0 the initial value for the process, if None, then X_0 is taken
        to be gamma (the asymptotic mean).
    Returns a 1D array.
    """
    t = np.arange(T, dtype=np.float128)  # float to avoid np.exp overflow
    exp_alpha_t = np.exp(-OU_params.alpha * t)
    dW = brownian_motion.get_dW(T, random_state)
    integral_W = _get_integal_W(t, dW, OU_params)
    _X_0 = _select_X_0(X_0, OU_params)
    return (
        _X_0 * exp_alpha_t
        + OU_params.gamma * (1 - exp_alpha_t)
        + OU_params.beta * exp_alpha_t * integral_W
    )


def _select_X_0(X_0_in: Optional[float], OU_params: OUParams) -> float:
    """Returns X_0 input if not none, else gamma (the long term mean)."""
    if X_0_in is not None:
        return X_0_in
    return OU_params.gamma


def _get_integal_W(
    t: np.ndarray, dW: np.ndarray, OU_params: OUParams
) -> np.ndarray:
    """Integral with respect to Brownian Motion (W), ∫...dW."""
    exp_alpha_s = np.exp(OU_params.alpha * t)
    integral_W = np.cumsum(exp_alpha_s * dW)
    return np.insert(integral_W, 0, 0)[:-1]


# ------------------------------------------------------------------
# OU parameters from data with OLS.


def estimate_OU_params(X_t: np.ndarray) -> OUParams:
    """
```


```
        y = np.diff(X_t)
        X = X_t[:-1].reshape(-1, 1)
        reg = LinearRegression(fit_intercept=True)
        reg.fit(X, y)
        # regression coeficient and constant
        alpha = -reg.coef_[0]
        gamma = reg.intercept_ / alpha
        # residuals and their standard deviation
        y_hat = reg.predict(X)
        beta = np.std(y - y_hat)
        return OUParams(alpha, gamma, beta)


# ------------------------------------------------------------
# Correlated OU processes


def get_corr_OU_procs(
    T: int,
    OU_params: Union[OUParams, tuple[OUParams, ...]],
    n_procs: Optional[int] = None,
    rho: Optional[float] = None,
    random_state: Optional[int] = None,
) -> np.ndarray:
    """
    Simulate correlated OU processes, correlation (rho) can be 0 or None.
    - T is the sample size of the processes.
    - OU_params can be a an instance of OUParams, in that case
        all processes have the same parameters. It can also be a tuple,
        in that case each process will have the parameters in the tuple,
        each column in the resulting 2D array corresponds to the tuple index.
    - n_procs is ignored if OU_params is tuple, else, corresponds to the number
        of processes desired. If OU_params is not tuple and n_procs is None, will
        raise ValueError.
    - rho is the correlation coefficient.
    - random_state to reproduce results.

    """
    _n_procs = _get_n_procs(OU_params, n_procs)
    corr_dWs = brownian_motion.get_corr_dW_matrix(
        T, _n_procs, rho, random_state
    )
    is_OU_params_tpl = _is_OU_params_tuple(OU_params)
```

```
    return np.asarray(OU_procs).T


def _is_OU_params_tuple(
    OU_params: Union[OUParams, tuple[OUParams, ...]]
) -> bool:
    """
    Check is OU_params is a tuple of params,
    return bool.
    """
    return isinstance(OU_params, tuple)


def _get_n_procs(
    OU_params: Union[OUParams, tuple[OUParams, ...]], n_procs: Optional[int]
) -> int:
    """
    Define the number of processes, if Ou_params is a tuple the
    number of processes is the lenght of the tuple. If it is not a tuple
    then it is the "n_procs" supplied as argument,
    if it is None will raise ValueError.
    """
    if _is_OU_params_tuple(OU_params):
        return len(OU_params)  # type: ignore
    elif n_procs is None:
        raise ValueError("If OU_params is not tuple, n_procs cannot be None.")
    return n_procs


def _get_OU_params_i(
    OU_params: Union[OUParams, tuple[OUParams, ...]],
    i: int,
    is_OU_params_tpl: bool,
) -> OUParams:
    """
    Returns the ith value of the OU_params tuple if it is a tuple,
    otherwise returns OUParams.
    """
    if is_OU_params_tpl:
        return OU_params[i]  # type: ignore
    return OU_params  # type: ignore
```

```
170       X_0 is taken as the asymptotic mean gamma for simplicity.
171       """
172       t = np.arange(T, dtype=np.float128)  # float to avoid np.exp overflow
173       exp_alpha_t = np.exp(-OU_params.alpha * t)
174       integral_W = _get_integal_W(t, dW, OU_params)
175       return (
```

and estimate its parameters from data. This is extremely useful for Monte Carlo simulations, once we know the parameters of a process we want to study then we can generate many Monte Carlo trials to get some statistics and further analysis.

Stay tuned for the next story in the series.

References

[1] B. Øksendal, Stochastic Differential Equations (2013), Sixth Edition, Springer.

Stay connected for the next stories of the series. Follow me on Medium and subscribe to get the updates of the next stories as soon as they come out.

I hope this story was useful to you. If I missed anything, please let me know.

Liked the story? Become a Medium member through my referral link and get unlimited access to my stories and many others.

writers you read, and you get full access to every story...

medium.com

Navigate through the Stochastic Processes Simulation series

Previous story in the series:

Stochastic Processes Simulation — Brownian Motion, The Basics

Part 1 of the Stochastic Processes Simulation series. Simulate correlated Brownian motions in Python from scratch.

towardsdatascience.com

Next stories in the series:

Stochastic Processes Simulation — The Cox-Ingersoll-Ross Process

Part 3 of the Stochastic Processes Simulation series. Simulating the Cox-Ingersoll-Ross process in Python from scratch.

towardsdatascience.com

Stochastic Processes Simulation — Geometric Brownian Motion

Part 4 of the Stochastic Processes Simulation series. Simulating geometric Brownian motion in Python from scratch.

towardsdatascience.com

Stochastic Processes Simulation — Generalized Geometric Brownian Motion