Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

pymle: A Python Package for Maximum Likelihood Estimation and Simulation of Stochastic Differential Equations

J.L. Kirkby
Georgia Institute of Technology

D.H. Nguyen
University of Alabama

D. Nguyen
Marist College

N. Nguyen
University of Rhode Island

Abstract

This paper introduces the object-oriented Python package **pymle**, which provides core functionality for maximum likelihood estimation and simulation of univariate stochastic differential equations. The package supports maximum likelihood estimation using Euler, Elerian, Ozaki, Shoji-Ozaki, Hermite polynomial, and Kessler density approximations, as well as a recently proposed continuous-time Markov chain approximation scheme. Exact maximum likelihood estimation is also provided when available. The framework supports estimation and simulation for 21 stochastic differential equations models at the time of writing, and its object oriented design facilitates easy extensions to new models and approximation methods.

Keywords: MLE, diffusion, SDE, Python, Maximum Likelihood estimation, CTMC, continuous-time Markov Chain.

1. Introduction

Continuous-time diffusion processes, defined by stochastic differential equations, are used extensively in modern financial theory to model the dynamics of asset prices, interest rates, and foreign exchange rates, among numerous other applications, see Karatzas and Shreve (2014), Glasserman (2013). Notable examples include the Vasicek process Vasicek (1977) and Cox–Ingersoll–Ross (CIR) diffusion Cox, Ingersoll Jr, and Ross (2005) which are widely used to price zero coupon bound and to describe the evolution of interest rates. The celebrated

Black-Scholes framework Black and Scholes (1973) assumes that the underlying asset dynamics follows a geometric Brownian motion (GBM), although more complex stochastic models have surfaced over the last few decades.

When pricing option values or drawing statistical inferences based on diffusions, one needs to know the parameters in the drift and diffusion terms of the underlying process. Typically, one assumes that the process belongs to some parametric family and uses a discretized finite sample path of the process to estimate the unknown parameters. For a continuous-time diffusion, its transition density function plays a crucial role in understanding the dynamics of the process. Most importantly, it can be used to estimate the model's unknown parameters by means of maximum likelihood. Unfortunately, except for some special cases such as Geometric Brownian Motion or CIR processes, the closed-form of the transition density function is not available for most diffusions. As a result, it becomes virtually impossible to determine the *exact* maximum likelihood estimates for the unknown parameters.

To overcome the unavailability of the transition density, approximation methods are usually employed. Several econometric approaches have been proposed to estimate the unknown parameters. These econometric methods include the simulation approach (Gourieroux, Monfort, and Renault (1993); Gallant and Tauchen (1996)), (generalized) method of moments (Hansen and Scheinkman (1993); Kessler and Sørensen (1999)), (non)parametric density matching (Aït-Sahalia (1995); Ait-Sahalia (1996), empirical characteristic functions Cui, Kirkby, and Nguyen (2021a), and Bayesian methodologies (Eraker (2001); Jones (1997)). Aït-Sahalia (2002) makes a fruitful breakthrough in using Hermite polynomials to orthogonally approximate the transition density of a univariate time-homogeneous diffusion. This idea was later extended to time-inhomogenous diffusions in Egorov, Li, and Xu (2003), multivariate time-homogenous (Aït-Sahalia (2008)) and time-inhomogeneous diffusions (Choi (2013)), stochastic volatility (Aït-Sahalia and Kimmel (2007)) and affine multi-factor models (Aït-Sahalia and Kimmel (2010)).

For many years, R has been the de-facto scripting language for applied statisticians, boasting a tremendous collection of state-of-the-art as well as cutting-edge statistical packages. On the contrary, while Python has become the forerunner for machine learning applications and research, its statistical stack is, to phrase it politely, inadequate.[1] Python is often described as a having "batteries included", which has no-doubt contributed to its rise in popularity in recent years, though it seems that its statistical batteries were never fully charged. A notable library is **statsmodels** Seabold and Perktold (2010), which provides a broad set of functionality for classical statistics problems such regression and time-series analysis. Frameworks such as **pandas** McKinney *et al.* (2011) and **scikit-learn** Pedregosa, Varoquaux, Gramfort, Michel, Thirion, Grisel, Blondel, Prettenhofer, Weiss, Dubourg *et al.* (2011) have provided essential ingredients for data science in Python, but there is much more ground to cover before python can be taken seriously by statisticians.

As scripting languages go, the object-orientation of Python positions it well as a prototyping language for pre-production systems, and a recent trend is to even employ Python applications directly into production. As machine learning and statistical applications forge ahead with increasing momentum, the benefits of strengthening Python's statistical offering are hard to overstate.

Motivated by practical applications in fields such as finance and economics, in this paper

[1] Disclaimer: the authors use and enjoy both languages, among others.

we introduce the **pymle** package which aims to humbly address a key shortcoming of the existing Python stack, which is simulation and inference for stochastic differential equations (SDE). Several powerful packages exist in R, see those listed in Table 1, but relatively little is available to the Python statistician (see for example Rydin Gorjão, Witthaut, and Lind (2023) for a very recent work, **jumpdiff**, for jump diffusion inference using non-parametric NadarayaWatson estimators). While much existing SDE functionality exists in R, the **pymle** packages harnesses the strengths of a fully object-oriented design. As such, it is quite easily extensible, providing the necessary interfaces to access simulation and inference for any SDE, with a trivial amount of additional work required to add new models. We take a "replaceable parts" perspective, allowing maximal customization to the users of the library, while providing useful built-in functionality.

1.1. Existing Packages

While Python packages are quite sparse in the area of simulation and inference for SDEs, there are many well known frameworks written in R. To a large extent, the objective of **pymle** is to provide similar functionality to Python users. Before introducing the **pymle** package in detail, it is worth mentioning several high quality R packages including the **yuima** package of Brouste, Fukasawa, Hino, Iacus, Kamatani, Koike, Masuda, Nomura, Ogihara, Shimuzu *et al.* (2014) and **sde** of Iacus (2007), which can be utilized for simulation and inference for SDEs. These two packages in particular provide the inspiration for what is offered in **pymle**. The **yuima** package is designed for simulation and inference of stochastic differential equations encompassing both one-dimensional and multidimensional diffusions. Within the **yuima** package stochastic differential equations can take on highly abstract forms, driven by a (multidimensional) Brownian motion process or even fractional Brownian motion with a general Hurst parameter. They can also include jumps specified as Lévy noise. Under this framework, the **yuima** package offers various functions for conducting simulations and statistical analyses. We also note that **yuima** depends on some other interesting libraries, such as **zoo** (Zeileis and Grothendieck 2005). For further applications of **yuima**, please refer to Iacus and Yoshida (2018).
On the other hand, the **sde** package developed by Iacus (2007) is an R package primarily focused on the simulation and inference of one-dimensional diffusion processes driven by the Brownian (Wiener) process. The **pymle** package is inspired especially by the **sde** package and the book Iacus (2009), and offers much of the same functionality to the Python user. As a result, in essence, **pymle** shares many similarities with the **sde** package. However, it is worth emphasizing that, while the two frameworks cover many of the same estimation procedures, the design of the packages is different in many important ways. In particular, **pymle** is a fully objected-oriented library that utilizes design principals/patterns that make it very easy to extend by the user with minimal coding effort. It embraces a replaceable parts approach that is well-suited for R&D purposes, as described in what follows. We also note that **pymle** supports the recently proposed continuous time Markov chain (CTMC) scheme of Kirkby, Nguyen, Nguyen, and Nguyen (2022). One distinct feature of the CTMC approximation is that it introduces no time-discretization error during parameter estimation, and is thus well-suited for typical econometric situations with infrequently sampled data.

Package	Language	Reference	Note
yuima	R	Brouste *et al.* (2014)	Simulation & inference for SDE
sde	R	Iacus (2007)	Simulation & inference for 1D SDE
Sim.DiffProc	R	Guidoum and Boukhetala (2020)	Parallel Monte Carlo and estimation
ctmcd	R	Pfeuffer (2017)	CTMC estimation
spate	R	Sigrist, Künsch, and Stahel (2015)	SPDE and spatio-temporal simulation
jumpdiff	Python	Rydin Gorjão *et al.* (2023)	Jump-diffusion inference

Table 1: Existing statistical packages for SDE simulation and inference.

1.2. Organization of paper

The rest of this paper is organized as follows: Section 2 reviews some fundamental facts of stochastic differential equations. Several examples are provided for the later use. Section 3 provides some numerical methods which are used to approximate the transition density of diffusion. Section 4 gives an in-depth description and usage of the **pymle** package. Numerous examples are provided demonstration. Section 5 concludes the paper.

2. Stochastic differential equations

Consider the stochastic diffusion process

$$dS_t = \mu(S_t, t; \theta)dt + \sigma(S_t, t; \theta)dW_t, \quad t \geq 0, \tag{1}$$

where $(W_t)_{t\geq 0}$ is the standard Brownian motion, $\mu(S_t, t) \equiv \mu(S_t, t; \theta) : \mathbb{R} \times \mathbb{R}_+ \times \mathbb{R}^d \to \mathbb{R}$ and $\sigma(S_t, t) \equiv \sigma(S_t, t; \theta) : \mathbb{R} \times \mathbb{R}_+ \times \mathbb{R}^d \to \mathbb{R}_+$ are the drift and diffusion terms, respectively. Note that it is well-known that a stochastic differential equation of type (1) is Markovian, see Jazwinski (2007). The unknown parameters $\theta = (\theta_1, \ldots, \theta_d)$ appear in both the drift term as well as the diffusion term. In order to simulate or perform statistical inferences for the diffusion given in (1), one must estimate the parameter θ from available information, which normally is given in terms of a finite data sample. It is assumed that the unknown parameter vector $\theta = (\theta_1, \theta_2, \ldots, \theta_d)$ belongs to a compact set $\Theta \subset \mathbb{R}^d$. Given $\Delta > 0$, let $p(s'|s, \Delta) \equiv p(s'|s, \Delta; \theta)$ denote the transition density function of S_t. That is

$$\mathbb{P}(S_{t+\Delta} \in ds'|S_t = s) = p(s'|s, \Delta)ds'. \tag{2}$$

Suppose that $S_0, S_1, \ldots, S_N$ is a sequence of $(N + 1)$ historical observations of S_t sampled at non-stochastic times $t_0 < t_1 < \ldots < t_N$. The joint likelihood of this sample is given by

$$p_0(S_0|\theta) \prod_{n=1}^{N-1} p(S_{n+1}|S_n, t_{n+1} - t_n), \tag{3}$$

where $p_0(S_0|\theta)$ is the density of the initial state. In this paper, we choose $\Delta = t_{i+1} - t_i$. For notational simplicity, we will suppress the dependence on the parameter vector θ in what follows. We assume that S_0 is given. The equation (1) can be written in the integral form

$$S_T = S_0 + \int_0^T \mu(S_t, t)dt + \int_0^T \sigma(S_t, t)dW_t, \quad T \geq 0,$$

in which the first integral on the right hand side is of Riemann-Stieltjes type and the second integral is an Ito integral, see Kloeden and Platen (1992). In general, for the simulation and

Model	Dynamics	Constraint	Reference
BM	$dS_t = \mu dt + \sigma dW_t$	$\sigma > 0$	Karatzas and Shreve (2014)
GBM	$dS_t = \mu S_t dt + \sigma S_t dW_t$	$\sigma > 0$	Karatzas and Shreve (2014)
IGBM	$dS_t = \kappa(\mu - S_t)dt + \sigma S_t dW_t$	$\sigma > 0$	Abadie and Chamorro (2008); Zhao (2009)
Peral-Verhulst	$dS_t = \kappa(\mu - S_t)S_t dt + \sigma S_t dW_t$	$\sigma > 0$	Dixit, Pindyck, and Pindyck (1994)
Linear SDE 1	$dS_t = (a + bS_t)dt + (c + dS_t)dW_t$	$a, c, d \neq 0$	Kloeden and Platen (1992)
Linear SDE 2	$dS_t = (a + bS_t)dt + cS_t dW_t$	$c \neq 0$	Kloeden and Platen (1992)
Logistic	$dS_t = S_t(1 - aS_t)dt + bS_t dW_t$	$b > 0$	Kloeden and Platen (1992)
3/2	$dS_t = S_t(\kappa(\mu - S_t)dt + \sigma S_t\sqrt{S_t}dW_t$	$\sigma > 0$	Grasselli (2017); Kirkby and Nguyen (2020)
CEV	$dS_t = \kappa(\mu - S_t)dt + \sigma S_t^{\gamma} dW_t$	$\gamma, \sigma > 0$	Cox (1996)
CIR	$dS_t = \kappa(\mu - S_t)dt + \sigma\sqrt{S_t}dW_t$	$2\kappa\mu \geq \sigma^2$	Cox *et al.* (2005)
CKLS	$dS_t = (\theta_1 + \theta_2 S_t)dt + \theta_3 S_t^{\theta_4} dW_t$	$\theta_3 > 0$	Chan, Karolyi, Longstaff, and Sanders (1992)
Feller's square root	$dS_t = S_t(\theta_1 - (\theta_3^3 - \theta_1\theta_2)S_t)dt + \theta_3 S_t^{3/2} dW_t$	$\theta_3 > 0$	Ahn and Gao (1999)
Hyperbolic	$dS_t = -\frac{\kappa S_t}{\sqrt{1+S_t^2}}dt + \sigma dW_t$	$\kappa, \sigma > 0$	Eberlein and Keller (1995)
Hyperbolic 2	$dS_t = \frac{\sigma^2}{2}\left(\beta - \gamma\frac{S_t}{\sqrt{\delta^2 + (S_t - \mu)^2}}\right)dt + \sigma dW_t$	$\sigma, \lvert\delta\rvert > 0$	Iacus (2009)
Jacobi	$dS_t = -\theta(S_t - 1/2)dt + \sqrt{\theta S_t(1 - S_t)}dW_t$	$\theta > 0$	Iacus (2009)
Modified CIR	$dS_t = -\theta_1 S_t dt + \theta_2\sqrt{1 + S_t^2}dW_t$	$\theta_1 + \theta_2^2 > 0$	Iacus (2009)
OU	$dS_t = \kappa(\mu - S_t)dt + \sigma dW_t$	$\sigma > 0$	Uhlenbeck and Ornstein (1930)
Radial OU	$dS_t = (\theta S_t^{-1} - S_t)dt + \sigma dW_t$	$\sigma > 0$	Iacus (2009)
Pearson	$dS_t = -\theta(S_t - \mu)dt + \sqrt{2\theta(aS_t^2 + bS_t + c)}dW_t$	$\theta > 0$	Forman and Sørensen (2008)
Nonlinear mean reversion	$dS_t = (\alpha_{-1}S_t^{-1} + \alpha_0 + \alpha_1 S_t + \alpha_2 S_t^2)dt + \sqrt{\beta_0 + \beta_1 S_t + \beta_2 S_t^{\beta_3}}dW_t$	See ref.	Ait-Sahalia (1996)
Nonlinear SDE	$dS_t = (\alpha_{-1}S_t^{-1} + \alpha_0 + \alpha_1 S_t + \alpha_2 S_t^2)dt + (\beta_0 + \beta_1 S_t + \beta_2 S_t^{\beta_3})dW_t$	See ref.	Ait-Sahalia (1996)

Table 2: Models provided by the **pymle** package.

inference procedures to make sense, we will assume at least that

$$\mathbb{P}\left\{\int_0^T \sup_{|x|\leq B}(|\mu(x,t)| + \sigma^2(x,t))dt < \infty\right\} = 1, \quad \forall T, B \in [0, \infty),$$

while additional growth conditions on the coefficients are required to ensure the existence of a unique (strong) solution to (1) (see Karatzas and Shreve (2014)). We also note that while the majority of models typically encountered are time-homogeneous, the simulation and inference procedures are designed to support the case of time-dependent coefficients. For more details, please refer to Section 4.8.

Table 2 lists some of the models provided in this package, and the addition of new models requires minimal coding, while automatically inheriting all of the core functionality for simulation and inference. We note that in Table 2, we are aware that some models nest other models as special cases. However, due to their popularity and practical needs, we list them there for the user's convenience, and moreover due to specialized constraints and initial conditions that one may place on the model during the fitting process.

2.1. Example: Geometric Brownian motion

For the purpose of illustration, let's consider the Geometric Brownian Motion (GBM) whose dynamics is given by

$$dS_t = S_t\mu dt + S_t\sigma dW_t.$$

This model is widely used in economics and finance to model the dynamics of a risky asset, see Black and Scholes (1973). For any $t' > t \geq 0$, $\Delta := t' - t$, and $\theta := (\mu, \sigma)$, the log-normal

Model	Reference	Dynamics
GBM	Brown (1828)	$dS_t = S_t\mu dt + S_t\sigma dW_t$
Vasicek	Vasicek (1977)	$dS_t = \kappa(\mu - S_t)dt + \sigma dW_t$
CIR	Cox *et al.* (2005)	$dS_t = \kappa(\mu - S_t)dt + \sigma\sqrt{S_t}dW_t$
CEV	Cox (1996)	$dS_t = \mu S_t dt + \sigma S_t^{\gamma} dW_t$

Table 3: Some special cases of the CKLS model.

transition density of S_t is given in closed form by

$$p(s'|s,\Delta) = \frac{1}{s'\sigma_\Delta\sqrt{2\pi}}\exp\left(-\frac{(\ln(s') - \mu_\Delta(s))^2}{2\sigma_\Delta^2}\right),$$

where $\mu_\Delta(s) := \ln(s) + \left(\mu - \frac{1}{2}\sigma^2\right)\Delta$, and $\sigma_\Delta := \sigma\sqrt{\Delta}$.

2.2. Example: Chan-Karolyi-Longstaff-Sanders (CKLS)

Another interesting example is the Chan-Karolyi-Longstaff-Sanders (CKLS) family of models (see Chan *et al.* (1992)), which is a four-parameter extension of the constant elasticity of variance model (CEV) Cox (1996) given by

$$dS_t = (\theta_1 + \theta_2 S_t)dt + \theta_3 S_t^{\theta_4} dW_t.$$

This model does not admit an explicit transition density, except in the case where $\theta_1 = 0$ (Hsu, Lin, and Lee 2008) or $\theta_4 = 1/2$. We assume that $\theta_3 > 0$, and the process is positive as long as $\theta_1, \theta_2 > 0$ and $\theta_4 > 1/2$. It is noticed that the CKLS model nests many interest rate as well as short-rate models including GBM, Vasicek model Vasicek (1977), Cox-Ingersoll-Ross (CIR) Cox *et al.* (2005), constant elasticity of variance model (CEV) Cox (1996) as special cases. We summarize this nesting in Table 3.

Special case: Cox-Ingersoll-Ross

When $\theta_4 = 1/2$, the CKLS model reduces to the Cox-Ingersoll-Ross (CIR) model. The dynamics of S_t under CIR is given by

$$dS_t = \kappa(\mu - S_t)dt + \sigma\sqrt{S_t}dW_t.$$

It can be shown that $S_t \geq 0$ almost surely, and the CIR model is widely used to model short term interest rates (Cox *et al.* (2005)) or equity volatilities (Heston (1993)), both of which exhibit mean-reversion and tend to be positive[2]. The true transition density function is given by

$$p(s'|s,\Delta) = \frac{e^{\kappa\Delta}}{2c(\Delta)}\left(\frac{s'e^{\kappa\Delta}}{s}\right)^{(d-2)/4}\exp\left(-\frac{s + s'e^{\kappa\Delta}}{2c(\Delta)}\right) I_{d/2-1}\left(\frac{\sqrt{ss'e^{-\kappa\Delta}}}{c(\Delta)}\right),$$

where

$$c(\Delta) = \frac{\sigma^2}{4\kappa}(e^{\kappa\Delta} - 1), \quad d = \frac{4\kappa\mu}{\sigma^2},$$

[2]In some rare cases, interest rates have actually gone negative, requiring a model such as OU to capture this phenomenon, shown in Table 2.

and

$$I_\gamma(x) = \sum_{i=0}^{\infty} \frac{(x/2)^{2i+\gamma}}{i!\Gamma(i+\gamma+1)},$$

is the modified Bessel function of the first kind of order γ. Numerical evaluation of $p(s'|s;\theta)$ is delicate, and is best implemented using the exponentially damped Bessel function. The use of the exponentially dampened modified Bessel function is a well-established numerical practice to avoid overflow and underflow issues associated with the modified Bessel; see for example Carley (2013).

Simulation Scheme	Reference
Exact	Glasserman (2013)
Euler	Kloeden and Platen (1992)
Milstein	Mil'shtein (1979)
2nd order Milstein	Mil'shtein (1979)
CTMC	Kirkby *et al.* (2022)

Table 4: Simulation schemes provided in the **pymle** package for numerical simulation of SDE. Other simulation schemes can be easily added by extending the '`Stepper`' class; see Section 4.10 for details on adding new simulation schemes.

2.3. Simulation schemes

Here we briefly review the simulation schemes supported, which are listed in Table 4 for reference. When available, the package supports "Exact" simulation, which requires that the exact transition density be known for the model to simulate a draw from its dynamics without discretization error. The other methods listed in Table 4 are all in the family of time-discretization methods, as opposed to, for example, spatial discretization Cui, Kirkby, and Nguyen (2021b); Meier, Li, and Zhang (2023). Here we recall that a scheme is *strongly convergent* with order γ if $\mathbb{E}\left(|S_T - \widetilde{S}_T|\right) \leq K_T\Delta^\gamma$, and *weakly convergent* with order γ if there exists a constant K_T^g such that for all functions g in some class,

$$\left|\mathbb{E}[g(S_T)] - \mathbb{E}[g(\widetilde{S}_T)]\right| \leq K_T^g\Delta^\gamma,$$

for $\Delta \to 0$. Typically, the function g allowed must satisfy some smoothness and polynomial growth conditions, see Kloeden and Platen (1992). Note that in the above expectations, we have used the notation $\widetilde{S}_t$ to distinguish the time-discretization it from the true process, S_t, and that the discretization is dependent on the (uniform) step-size Δ. See Higham (2001) for an algorithmic introduction to numerical simulation of stochastic differential equations or Glasserman (2013) for an excellent financial treatment of SDE simulation.

Euler scheme

The simplest, and likely most widely used, is the Euler scheme (also known as Euler-Maruyama), which discretizes (1) according to

$$\widetilde{S}_{t+\Delta} = \widetilde{S}_t + \mu(\widetilde{S}_t, t)\Delta + \sigma(\widetilde{S}_t, t)(W_{t+\Delta} - W_t),$$

where $W_{t+\Delta} - W_t \sim \mathcal{N}(0, \Delta)$. The Euler scheme has order $\gamma = 1/2$ of *strong convergence*, and order $\gamma = 1$ of *weak convergence*, see Kloeden and Platen (1992).

Milstein scheme

By utilizing Ito's Lemma, Milstein Mil'shtein (1979) obtains a scheme of order $\gamma = 1$ for both weak and strong convergence. The approximation is given by

$$\widetilde{S}_{t+\Delta} = \underbrace{\widetilde{S}_t + \mu(\widetilde{S}_t, t)\Delta + \sigma(\widetilde{S}_t, t)(W_{t+\Delta} - W_t)}_{\text{Euler}} + \frac{1}{2}\underbrace{\sigma(\widetilde{S}_t, t)\sigma_s(\widetilde{S}_t, t)\left((W_{t+\Delta} - W_t)^2 - \Delta\right)}_{\text{Correction}}, \quad (4)$$

which corresponds to Euler's scheme plus a correction, where σ_s is the derivative of σ with respect to the first (spatial) argument. Note that for models with a constant diffusion term, $\sigma_s \equiv 0$, such as Brownian Motion, the Euler and Milstein schemes coincide.

Milstein second scheme

The second Milstein scheme, Mil'shtein (1979), improves the weak convergence order of the first scheme to $\gamma = 2$, and is given by

$$\begin{aligned}\widetilde{S}_{t+\Delta} = \widetilde{S}_t &+ \left(\mu - \frac{1}{2}\sigma\sigma_s\right)\Delta + \sigma Z\sqrt{\Delta} + \frac{1}{2}\sigma\sigma_s\Delta Z^2 \\ &+ \Delta^{\frac{3}{2}}\left(\frac{1}{2}\mu\sigma_s + \frac{1}{2}\mu_s\sigma + \frac{1}{4}\sigma^2\sigma_{ss}\right)Z + \Delta^2\left(\frac{1}{2}\mu\mu_s + \frac{1}{4}\mu_{ss}\sigma^2\right),\end{aligned}$$

where we have suppressed the arguments of $\mu(S_t, t), \sigma(S_t, t)$ for simplicity, and denoted $Z \sim \mathcal{N}(0, 1)$.

3. Maximum likelihood estimation of diffusions

To estimate the unknown parameter θ, we assume that a discrete sample of S_t is observed: $S_0, S_1, S_2, \ldots, S_N$, with observations taken at a uniform frequency $\Delta > 0$. By the Markovian property of S_t and from (3), the sample log-likelihood function is given by

$$L_N(\theta, \Delta) := p_0(S_0|\theta) + \sum_{n=1}^{N-1} \ln p(S_{n+1}|S_n, \Delta). \quad (5)$$

Note that the log likelihood function $L_N(\theta, \Delta)$ depends on the first observation S_0. However, this can be ignored as it is dominated by the sum of the other terms as $N \to \infty$. For more discussions along this line, please see Section 3.1 in Aït-Sahalia (2002). The maximum likelihood estimator (MLE) of θ is defined to be the maximizer of the following optimization problem:

$$\widehat{\theta}_N := \text{argmax}_{\theta\in\Theta} L_N(\theta, \Delta). \quad (6)$$

Equivalently, $\widehat{\theta}_N$ is obtained by minimizing the negative log-likelihood function, and we will refer to this as the *exact MLE*. We note that in (5), the loglikelihood function $L_N(\theta, \Delta)$ crucially depends on the transition density function $p(S_{n+1}|S_n, \Delta)$. Unfortunately, except for some special cases such as Geometric Brownian Motion or CIR processes, the closed-form of the transition density function is not available for most of diffusions. As a result, it virtually impossible to carry out the exact MLE estimate for θ. Hence, numerical approximations of the transition density function are often employed for the purposes of simulation as well as inference. More specifically, one will approximate the true transition density $p(S_{n+1}|S_n, \Delta)$

Scheme	Reference	Notes
Exact	Glasserman (2013)	Available when exact transition density is known
Euler	Kloeden and Platen (1992)	Based on Euler expansion of SDE
Elerian	Elerian (1998)	Based on Milstein expansion of SDE
Hermite polynomial expansion	Aït-Sahalia (2002)	Based on Hermite polynomials
Kessler	Kessler and Sørensen (1999)	Based on higher-order Ito-Taylor expansion
Ozaki	Ozaki (1985, 1992, 1993)	Based on normal approximation of the pdf
Shoji-Ozaki	Shoji and Ozaki (1997)	Based on normal approximation of the pdf
CTMC	Kirkby *et al.* (2022)	Based on CTMC approximation of diffusion generator

Table 5: Maximum-Likelihood estimation procedures supported.

function by $\tilde{p}(S_{n+1}|S_n, \Delta)$. Hence the sample log-likelihood function $L_N(\theta, \Delta)$ is approximated by

$$\tilde{L}_N(\theta, \Delta) = \sum_{n=1}^{N-1} \ln \tilde{p}(S_{n+1}|S_n, \Delta). \tag{7}$$

Finally, $\widehat{\theta}_N$ is approximated using $\tilde{L}_N$. Normally, numerical procedures such as Newton's method are used to optimize $\tilde{L}_N$. A quick summary of maximum likelihood estimation schemes supported by **pymle** package is given in Table 5. In the following subsections, we briefly describe some popular approximation schemes which can be used to numerically approximate the transition density function.

3.1. Euler approximation

Recall that the Euler scheme approximates the SDE (1) by

$$\widetilde{S}_{t+\Delta} - \widetilde{S}_t = \mu(\widetilde{S}_t, t)\Delta + \sigma(\widetilde{S}_t, t)(W_{t+\Delta} - W_t). \tag{8}$$

It follows that $\widetilde{S}_{t+\Delta} - \widetilde{S}_t$ is normally distributed, which yields the approximate density

$$p^{Euler}(s'|s, \Delta) := \frac{1}{\sqrt{2\pi\Delta\sigma^2(s,t)}} \exp\left(-\frac{(s' - s - \mu(s,t)\Delta)^2}{2\Delta\sigma^2(s,t)}\right).$$

Euler's approximation only works well for small Δ, but is a reasonable first approximation which requires minimal computational (or coding) effort to apply the update in (8). The cost of this simplicity is slower convergence, and improved approximations are available which account for derivatives of the SDE coefficients.

3.2. Elerian approximation

The Elerian Elerian (1998) approximation of the transition density is based on a Milstein approximation of the SDE dynamics, recall (4). In particular,

$$p^{Elerian}(s'|s, \Delta) := \frac{z^{-\frac{1}{2}} \cosh(\sqrt{Cz})}{|A|\sqrt{2\pi}} e^{-\frac{C+z}{2}}, \tag{9}$$

where

$$A(s, \Delta) := \frac{\sigma\sigma_s\Delta}{2}, \qquad B(s, \Delta) := -\frac{\sigma}{2\sigma_s} + s + \mu\Delta - A(s, \Delta),$$
$$z(s', s, \Delta) := \frac{s' - B(s, \Delta)}{A(s, \Delta)}, \quad C(s, \Delta) := \frac{1}{\sigma_s^2\Delta},$$

which is valid when $\sigma_s \neq 0$, and $z > 0$. While this approach works reasonably well in some cases, we have noticed some severe numerical instabilities with the method in practice if not carefully implemented, due to overflow issues with $\cosh(x) := (e^x + e^{-x})/2$. For this reason, we use the equivalent form

$$p^{Elerian}(s'|s,\Delta) = \frac{z^{-\frac{1}{2}}\left(e^{\sqrt{Cz}-\frac{C+z}{2}} + e^{-\sqrt{Cz}-\frac{C+z}{2}}\right)}{2|A|\sqrt{2\pi}}, \tag{10}$$

which is much more stable. In Section 4.6, we encounter a real data example for which the Elerian approximation in (9) causes numerical overflow and a convergence failure during maximum likelihood optimization, whereas the stable version in (10) produces estimates consistent with alternative methods.

3.3. Ozaki scheme

In case the diffusion term of S_t is a constant, that is $\sigma(s,\theta) = \sigma$, Ozaki (1985, 1992, 1993), show that the transition density of S_t can be approximated by a normal density. Specifically, $\widetilde{S}_{t+\Delta}|\widetilde{S}_t = s \sim N(E(s), V(s))$, where

$$E(s) = s + \frac{\mu(s;\theta)}{\mu_s(s;\theta)}\left(e^{\mu_s(s;\theta)\Delta} - 1\right), \quad V(s) = \sigma^2\frac{e^{2K(s)\Delta} - 1}{2K(s)},$$

where we further define

$$K(s) = \frac{1}{\Delta}\ln\left(1 + \frac{\mu(s;\theta)}{s\mu_s(s;\theta)}\left(e^{\mu_s(s;\theta)} - 1\right)\right).$$

In the above equations, $\mu_s(s;\theta)$ denotes the $\frac{\partial}{\partial s}\mu(s;\theta)$.

3.4. Shoji-Ozaki scheme

In case the dynamics of the diffusion S_t admits the following form

$$dS_t = \mu(S_t, t;\theta)dt + \sigma dW_t,$$

with constant σ, another scheme is proposed in Shoji (1995), Shoji (1998) and Shoji and Ozaki (1997). Here we follow Section 2.12.2 of Iacus (2007). By defining

$$M(s,t,\theta) = \frac{\sigma^2}{2}\mu_{ss}(s,t;\theta) + \mu_t(s,t;\theta),$$

$$L(s,t,\theta) = \mu_s(s,t;\theta),$$

the transition density function can be approximated by

$$\widetilde{S}_{t+\Delta}|\widetilde{S}_t = s \sim N(A(s,t,\theta)s, B^2(s,t,\theta)),$$

with

$$A(s,t,\theta) = 1 + \frac{\mu(s,t,\theta)}{sL(s,t,\theta)}\left(e^{L(s,t,\theta)\Delta} - 1\right) + \frac{M(s,t,\theta)}{sL^2(s,t,\theta)}\left(e^{L(s,t,\theta)\Delta} - 1 - L(s,t,\theta)\Delta\right),$$

and

$$B(s,t,\theta) = \sigma\sqrt{\frac{e^{2L(s,t,\theta)\Delta} - 1}{2L(s,t,\theta)}}.$$

Remark: We note that the original Ozaki and Shoji-Ozaki schemes are designed for diffusions with constant diffusion coefficients. In case the diffusion coefficient is not constant, that is,

$$dS_t = \mu(S_t, t; \theta)dt + \sigma(S_t; \theta)dW_t, \quad t \geq 0,$$

one can convert it to a unit-coefficient diffusion using the Lamperti transform. Specifically, let

$$y = \int_0^s \frac{1}{\sigma(u;\theta)}du,$$

under which the dynamics of Y_t is given by

$$dY_t = \tilde{\mu}(Y_t, t; \theta)dt + dW_t, \quad Y_0 = y_0,$$

where the drift function of Y_t has the form

$$\tilde{\mu}(y,t;\theta) = \frac{\mu(s,t;\theta)}{\sigma(s;\theta)} - \frac{1}{2}\frac{\partial\sigma(s;\theta)}{\partial s}.$$

This can be used to extend the scope of applications of both the Ozaki and Shoji-Ozaki schemes to a broader class of diffusions, assuming the Lamperti transform can be computed.

3.5. Kessler approximation

The Kessler approximation Kessler (1997); Kessler and Sørensen (1999) is the conditional Gaussian approximation

$$p^{Kessler}(s'|s,\Delta) := \frac{1}{\sqrt{2\pi V(s,t)}}\exp\left(-\frac{(s' - E(s,t))^2}{2\sqrt{V(s,t)}}\right),$$

where the mean and variance are given respectively by

$$E(s,t) := s + \mu\Delta + \left(\mu\mu_s + \frac{1}{2}\sigma^2\mu_{ss}\right)\frac{\Delta^2}{2},$$
$$V(s,t) := s^2 + (2\mu s + \sigma^2)\Delta + (A(s,t) + B(s,t))\frac{\Delta^2}{2} - E^2(s,t),$$

and where

$$A(s,t) := 2\mu(\mu_s s + \mu + \sigma\sigma_s)$$
$$B(s,t) := \sigma^2(\mu_{ss}s + 2\mu_s + \sigma_s^2 + \sigma\sigma_{ss}).$$

This approximation is based on a higher-order Ito-Taylor expansion.

3.6. Hermite polynomials expansion

Perhaps the most widely used scheme to date is the Hermite polynomial expansion (HPE) of Aït-Sahalia (2002). This approach is based on transforming the variable S_t to an equivalent,

yet more amendable variable Y_t using the Lamperti transformation, see Florens (1998). The key idea is that after applying the transformation, the new variable Y_t has a transitional density much "closer" to that of a normal distribution. More specifically, let

$$y = \int_0^s \frac{1}{\sigma(u;\theta)} du,$$

under which the dynamics of Y_t is given by

$$dY_t = \tilde{\mu}(Y_t;\theta)dt + dW_t, \quad Y_0 = y_0,$$

where the drift function of Y_t has the form

$$\tilde{\mu}(y;\theta) = \frac{\mu(s;\theta)}{\sigma(s;\theta)} - \frac{1}{2}\frac{\partial\sigma(s;\theta)}{\partial s}.$$

The transition density of $Z = (Y - Y_k)/\sqrt{\Delta}$ is given by

$$p = \phi(z)\sum_{j=0}^{\infty} c_j(\Delta, Y_k)H_j(\theta), \tag{11}$$

in which $\phi(\cdot)$ denotes the standard normal probability density function. For $j = 0, 1, 2, \ldots$, the function $H_j(\cdot)$ is the probabilists' Hermite polynomial of order j defined by the generating rule

$$H_j(z) = (-1)^j e^{z^2/2}\frac{d^j}{dz^j}\left(e^{-z^2/2}\right).$$

Moreover, the coefficient c_j is given by

$$c_j(\Delta, Y_k) = \frac{1}{j!}\mathbb{E}\left[H_j\left(\frac{Y - Y_k}{\sqrt{\Delta}}\right)|Y_k\right],$$

which can be approximated, see Aït-Sahalia (2002) for more details. Aït-Sahalia (2002) notes that for practical application only finite terms are needed to approximate the infinite sum in (11). For example, expansions of the coefficients $c_0, c_1, \ldots, c_6$ give the accuracy to $o(\Delta^3)$ while the expansions of the coefficients $c_0, c_1, \ldots, c_{10}$ give the accuracy to $o(\Delta^5)$. Note that once the transition density of Z can be approximated, the transition density of S_t can be approximated using the reverse transformation, see Aït-Sahalia (2002). For an accessible exposition, please see the recent paper Hurn, Lindsay, and Xu (2021).

3.7. Markov chain approximation

In this section, we review the idea of using a continuous-time Markov chain to perform MLE for the continuous diffusion in (1), recently proposed in Kirkby *et al.* (2022). The key idea is to discretize the state space of the diffusion (1) into a finite discere grid of spatial points while preserving the continuous-time dimension of the diffusion process. Unlike typical time discretization approaches, such as pseudo-likelihood approximations with Shoji-Ozaki or Kessler's method, the CTMC approximation introduces no time-discretization error during parameter estimation, and is thus well-suited for typical econometric situations with infrequently sampled data.

Given a parametric diffusion family characterized by (1), we will construct a continuous-time Markov chain $\{S_t^m\}_{t\geq 0}$, taking values in some discrete state-space $\mathbb{S}_m := \{s_1, s_2, \ldots, s_m\}$, whose dynamics well resemble those of S_t. For the Markov chain S_t^m, its transitional dynamics are described by the *rate matrix* $\mathbf{Q} = \mathbf{Q}(\theta) = [q_{ij}(\theta)]_{m\times m} \in \mathbb{R}^{m\times m}$, whose elements $q_{ij} = q_{ij}(\theta)$ satisfy the q-property: (i) $q_{ii} \leq 0$, $q_{ij} \geq 0$ for $i \neq j$, and (ii) $\sum_j q_{ij} = 0, \forall i = 1, 2, \ldots, m$. In terms of q_{ij}'s, the transitional probability of the CTMC S_t^m is given by:

$$\mathbb{P}(S_{t+\Delta}^m = s_j | S_t^m = s_i) = \delta_{ij} + q_{ij}\Delta + o(\Delta^2), \tag{12}$$

where in the above expression δ_{ij} denotes the Kronecker delta, and we note that $\mathbb{P}(S_{t+\Delta}^m = s_j | S_t^m = s_i, S_{t'}^m, 0 \leq t' \leq t) = \mathbb{P}(S_{t+\Delta}^m = s_j | S_t^m = s_i)$ due to the Markov property. The generator of the approximate process is given by $\mathbf{Q} = (q_{ij}(\theta))_{m\times m}$ where

$$q_{ij}(\theta) = \begin{cases} \dfrac{\mu^-(s_i;\theta)}{k_{i-1}} + \dfrac{\sigma^2(s_i;\theta) - (k_{i-1}\mu^-(s_i;\theta) + k_i\mu^+(s_i;\theta))}{k_{i-1}(k_{i-1}+k_i)}, & \text{if } j = i-1, \\ \dfrac{\mu^+(s_i;\theta)}{k_i} + \dfrac{\sigma^2(s_i;\theta) - (k_{i-1}\mu^-(s_i;\theta) + k_i\mu^+(s_i;\theta))}{k_i(k_{i-1}+k_i)}, & \text{if } j = i+1, \\ -q_{i,i-1} - q_{i,i+1}, & \text{if } j = i, \\ 0, & \text{if } j \neq i-1, i, i+1. \end{cases} \tag{13}$$

Here μ^+ (respectively, μ^-) denotes the positive (respectively, negative) part of μ and $\mathbf{k} := \{k_1, k_2, \ldots, k_{m-1}\}$ is assumed to be chosen such that

$$0 < \max_{1\leq i\leq m-1}\{k_i\} \leq \min_{\theta\in\Theta}\min_{1\leq i\leq m}\left\{\frac{\sigma^2(s_i;\theta)}{|\mu(s_i;\theta)|}\right\}.$$

It can be shown that the generator $\mathbf{Q}$ is diagonalizable and has exactly m real distinct eigenvalues, see Kirkby *et al.* (2022). This is a crucial property which facilitates computing the probability transition matrix $\mathbf{T}(\Delta)$ defined below. Next, let $\Delta > 0$ and assume that we observe $\boldsymbol{S}^m := (S_1^m, S_2^m, \ldots, S_N^m) = (S_\Delta^m, S_{2\Delta}^m, \ldots, S_{N\Delta}^m)$. Define the $m \times m$ probability transition matrix

$$\mathbf{T}(\Delta) = \exp(\mathbf{Q}\Delta) = \sum_{i=0}^{\infty}\frac{(\mathbf{Q}\Delta)^i}{i!}. \tag{14}$$

Note that since our $\mathbf{Q} = \mathbf{Q}(\theta)$ is a function of θ so is $\mathbf{T}(\Delta)$, and $\mathbf{T}(\Delta)_{ij}$ is the transition probability from the state s_i to state s_j. The likelihood of the sample is given by

$$P(\boldsymbol{S}^m | S_1^m, \mathbf{Q}) = \prod_{n=1}^{N-1}\mathbf{T}(\Delta)_{S_{n\Delta}^m, S_{(n+1)\Delta}^m}. \tag{15}$$

Here $\mathbf{T}(\Delta)_{S_{i\Delta}^m, S_{(i+1)\Delta}^m}$ corresponds to $\mathbf{T}(\Delta)_{j,k}$, with $j = \mathcal{I}(S_{i\Delta}^m)$ and $k = \mathcal{I}(S_{(i+1)\Delta}^m)$, where we define the index mapping

$$\mathcal{I} : \mathbb{S}_m \to \{1, \ldots, m\},$$

which maps $\mathcal{I}(S_j^m) \to j$, the corresponding state index. As in Kalbfleisch and Lawless (1985); McGibbon and Pande (2015), let $\mathbf{C}(\Delta) \in \mathbb{N}^{m\times m}$ be the matrix such that

$$\mathbf{C}(\Delta)_{i,j} = \sum_{n=1}^{N-1}\mathbf{1}_{\{S_{n\Delta}^m = s_i\}} \cdot \mathbf{1}_{\{S_{(n+1)\Delta}^m = s_j\}}, \tag{16}$$

which counts the number of times in the sample that a transition from state s_i to s_j occurs. We can then see from (15) that

$$P(\boldsymbol{S}^m|\mathbf{Q}, S_1^m) = \prod_{1\leq i,j\leq m} \mathbf{T}(\Delta)_{i,j}^{C(\Delta)_{i,j}}.$$

The log likelihood function is

$$\begin{aligned} L_{N,m}(\theta, \Delta) &= \ln P(\boldsymbol{S}^m|\mathbf{Q}(\theta), S_1^m) \\ &= \sum_{i,j} \mathbf{C}(\Delta)_{i,j} \ln \mathbf{T}(\Delta)_{i,j} \\ &= \sum_{i,j} \left(\mathbf{C}(\Delta) \circ \ln \exp(\Delta \mathbf{Q}(\theta))\right)_{i,j}. \end{aligned} \tag{17}$$

Here $\circ$ denotes the Hadamard matrix product and $\ln(\mathbf{A})$ is the element-wise logarithm. The maximum likelihood estimator (MLE) is

$$\widehat{\theta}_{N,m} = \arg\max_{\theta\in\Theta} L_{N,m}(\theta, \Delta), \tag{18}$$

which can be solved numerically using Newton's method. Additionally, it can be proved that $\widehat{\theta}_{N,m} \to \widehat{\theta}_N$ as $m \to \infty$, see Kirkby *et al.* (2022).

4. Package description

We now describe the **pymle** package, which is available under the MIT licence, and can be downloaded (and cloned) from `https://github.com/jkirkby3/pymle`. To install **pymle** into your Python environment, please use the following command in your terminal to install the stable version:

```
pip install pymle-diffusion --upgrade
```

The latest version of the code may be downloaded from github:

```
pip install git+https://github.com/jkirkby3/pymle.git
```

Table 6 provides an overview of the core components of the **pymle** package. At the highest level, the code is organized into four main components as described in Table 6: core, models, fit, and sim(ulation).

Folder	Notes
core	Contains the core classes used by fit and simulation components
models	Contains the models included by the pacakage
fit	Contains the classes/functions for fitting the models
sim	Contains the classes/functions for simulating the models

Table 6: An overview of **pymle** package organization.

4.1. Design overview

The **pymle** package relies heavily on an object-oriented design, with the core classes listed in Table 7. The column "Parent" displays the parent from which this class inherits, where '`ABC`' is used to denote an Abstract Base Class, intended to provide a basic interface. We take a "replaceable parts" approach to the design, providing built-in functionality with the ability to swap out components for maximal customization. Each of the components in Table 7 is completely replaceable/customizable to the user's preference, but has built-in functionality allowing the code to be useful out-of-the-box.

Class	Parent	Folder	Notes
'`Model1D`'	'`ABC`'	core	Base class inherited by all models
'`TransitionDensity`'	'`ABC`'	core	Base class inherited by all transition densities
'`Minimizer`'	'`ABC`'	fit	Base class for minimization
'`ScipyMinimizer`'	'`Minimizer`'	fit	Minization wrapper around **scipy** minimization
'`Estimator`'	'`ABC`'	fit	Base class inherited by all estimators
'`LikelihoodEstimator`'	'`Estimator`'	fit	Base class for liklihood-based estimators
'`AnalyticalMLE`'	'`LikelihoodEstimator`'	fit	Concrete MLE estimator
'`Stepper`'	'`ABC`'	sim	Interface for time stepping scheme
'`Simulator1D`'	–	sim	Simulate sample path from some '`Stepper`'

Table 7: List of key classes from the **pymle** package.

The most basic building blocks of **pymle** are the '`Model`' class and the '`TransitionDensity`' class, both located in the core directory. In the following subsections, we will briefly summarize the two classes to highlight the object-oriented and customizable nature of the library. We then provide several examples to demonstrate the usage of the package, and how the components interact.

4.2. Model class

We first describe the main class in the **pymle** package, which is the abstract '`Model1D`' class, for which we display the main methods.

```
class Model1D(ABC):
    def __init__(self,
                 has_exact_density: bool = False,
                 default_sim_method: str = "Milstein"):
        self._has_exact_density = has_exact_density
        self._params: Optional[np.ndarray] = None
        self._positive = False
        self._default_sim_method = default_sim_method

    @abstractmethod
    def drift(self,
              x: Union[float, np.ndarray],
              t: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        raise NotImplementedError
```

```
@abstractmethod
def diffusion(self,
              x: Union[float, np.ndarray],
              t: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
    raise NotImplementedError

@property
def params(self) -> np.ndarray:
    return self._params

@params.setter
def params(self, vals: np.ndarray):
    self._positive = self._set_is_positive(params=vals)
    self._params = vals
```

The class is quite simple, requiring the user to override just two methods which define the dynamics of the model (the drift and diffusion components). Each of these components takes two arguments:

- `x`: float or array, the value of the process at which to evaluate the drift/diffusion term.
- `t`: float or array, the time at which to evaluate the drift/diffusion term. This parameter only matters for time-inhomogeneous diffusions.

The model also provides a "setter" and "getter" method for accessing the model parameters, which are required by the fitting framework of **pymle**. The pythonic way of doing this is through the use of the property decorator, `@property`. Also note that all methods that must be implemented by any child of '`Model1D`' are decorated with `@abstractmethod`, and similarly for all other abstract classes in **pymle**.

We node that two parameters are provided in the constructor of each '`Model1D`'. The first is `has_exact_density`, which the child class should set to true if an exact density is implemented for that class (for example, Brownian motion has an exact density). In particular, when an exact transition density is known for the model, the user has the ability to override the `exact_density` method with the known functional form. The second parameter is `default_sim_method`, which is the default method for simulating the process, and is set to 'Milstein' by default. This can be overriden by simulator, but it allows each child class to set a good default (or perhaps an exact simulation method if available). There are several additional methods in the '`Model1D`' class which expose the derivatives of the drift and diffusion terms. Each method has a default implementation using finite difference schemes, which can be overriden by the user as discussed in Section 4.9.

The **pymle** package provides dozens of pre-defined models out-of-the-box, see Table 2. However, adding your own models is quite simple by extending '`Model1D`', and is discussed briefly in Section 4.7.

4.3. Transition density class

Another core component is the '`TransitionDensity`' base class, which defines the interface

of a transition density (approximation). This is used by the primary estimation producedure of **pymle**, which is Maximum Likelihood Estimation.

```
class TransitionDensity(ABC):
    def __init__(self, model: Model1D):
        self._model = model

    @property
    def model(self) -> Model1D:
        return self._model

    @abstractmethod
    def __call__(self,
                 x0: Union[float, np.ndarray],
                 xt: Union[float, np.ndarray],
                 t0: Union[float, np.ndarray],
                 dt: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        raise NotImplementedError
```

Each child of '`TransitionDensity`' must implement the `__call__` method, which returns the transition density evaluated at these arguments:

- `x0`: float or array, the current value of the process.
- `xt`: float or array, the value to transition to (must be same dimension as x0).
- `t0`: float, the time at which to evaluate the coefficients. This parameter is irrelevant (and not used) for time-homogenous models.
- `dt`: float or array, the time step between x0 and xt. If the time step is constant, then a float may be supplied.

Note that the constructor of '`TransitionDensity`' takes an instance of a '`Model1D`', which binds the transition density to that model, allowing it to access parameters and methods of the model (such as its drift and diffusion terms). This is a common pattern in pymle, where classes are given access to a generic instance of '`Model1D`', allowing them to access the methods as needed without knowing the particular type of model.

Example: Euler Approximation

A simple example of a '`TransitionDensity`' implementation is the '`EulerDensity`', which implements an Euler approximation to the transition density from the diffusion and drift terms of a model:

```
class EulerDensity(TransitionDensity):
    def __init__(self, model: Model1D):
        super().__init__(model=model)

    def __call__(self,
```

```
                x0: Union[float, np.ndarray],
                xt: Union[float, np.ndarray],
                t0: Union[float, np.ndarray],
                dt: float) -> Union[float, np.ndarray]:

        sig2t = (self._model.diffusion(x0, t0) ** 2) * 2 * dt
        mut = x0 + self._model.drift(x0, t0) * dt
        return np.exp(-(xt - mut) ** 2 / sig2t) / np.sqrt(np.pi * sig2t)
```

Recall how the '`EulerDensity`', like each '`TransitionDensity`', is defined in terms of a generic '`Model1D`'. This means that any newly added model automatically has an implementation of an Euler transition density, and similarly for the other simulated density approximations. The user simply defines a new model, and inherits the simulation and inference functionality for free. Also, Ait-Sahalia transition density is supplied to all models listed in Table 2, with the exception of the Hyperbolic and Hyperbolic 2 models. When users introduce a new model not listed in Table 2, they must provide the Ait-Sahalia transition density for their new model if they wish to use this functionality.

Class	folder	Purpose
'`ExactDensity`'	core	Class representing exact transition density (when available)
'`EulerDensity`'	core	Class representing Euler transition density approximation
'`OzakiDensity`'	core	Class representing Ozaki transition density approximation
'`ShojiOzakiDensity`'	core	Class representing Shoji-Ozaki transition density approximation
'`ElerianDensity`'	core	Class representing Elerian transition density approximation
'`KesslerDensity`'	core	Class representing Kessler transition density approximation
'`AitSahaliaDensity`'	core	Class representing Ait-Sahalia transition density approximation

Table 8: Supported transition density classes

Table 8 lists seven classes that extend the '`TransitionDensity`' base class, and are provided out-of-the-box with the package. Each of these implements a particular method that is useful for MLE, with references provided previously in Table 5. Implementing your own MLE procedure, based on some alternative transition density approximation, is as simple as creating a new '`TransitionDensity`' child class. Each of the classes listed in Table 5 provides a template for doing so.

4.4. Maximum Likelihood Estimation

We now discuss the main classes used for maximum likelihood estimation in **pymle**. The base estimator class, '`Estimator`', is a very simple and generic class with just one method to override:

```
def estimate_params(self, params0: np.ndarray) -> EstimatedResult:
    raise NotImplementedError
```

This method takes as input an initial parameter guess, and returns an '`EstimatedResult`' object containing the estimated parameters, and some measures of fit such as Akaike information criterion (AIC), Bayesian information criterion (BIC) (Hastie, Tibshirani, Friedman, and Friedman 2009) and the final estimated likelihood. This is further extended by

'LikelihoodEstimator', shown below, which is a type of 'Estimator' that optimizes a (negative) log likelihood function.

```
class LikelihoodEstimator(Estimator):
    def __init__(self,
                 sample: np.ndarray,
                 param_bounds: List[Tuple],
                 dt: Union[float, np.ndarray],
                 model: Model1D,
                 minimizer: Minimizer = ScipyMinimizer(),
                 t0: Union[float, np.ndarray] = 0):
        super().__init__(sample=sample, param_bounds=param_bounds,
                         dt=dt, model=model, t0=t0)
        self._min_prob = 1e-30
        self._minimizer = minimizer

    def estimate_params(self, params0: np.ndarray) -> EstimatedResult:
        res = self._minimizer.minimize(function=self.log_likelihood_negative,
                                       bounds=self._param_bounds,
                                       guess=params0)
        params = res.params
        final_like = -res.value
        return EstimatedResult(params=params,
                               log_like=final_like,
                               sample_size=len(self._sample) - 1)

    @abstractmethod
    def log_likelihood_negative(self, params: np.ndarray) -> float:
        raise NotImplementedError
```

A 'LikelihoodEstimator' is constructed from a sample, some parameter bounds that will be enforced during optimization, a time step between observation points (or an array of time steps in the time non-homogeneous case), a generic model of type 'Model1D', a minimizer of some form (defaulting to a 'ScipyMinimizer', discussed below), and an initial time (or array of times in the time non-homogeneous case). The method `log_likelihood_negative` must be implemented by every 'LikelihoodEstimator', and upon doing so the estimator is complete.

For example, the main estimation class we will use is the 'AnalyticalMLE' which extends 'LikelihoodEstimator' in a very simple way by overriding the likelihood function as follows:

```
class AnalyticalMLE(LikelihoodEstimator):
    def __init__(self,
                 sample: np.ndarray,
                 param_bounds: List[Tuple],
                 dt: Union[float, np.ndarray],
                 density: TransitionDensity,
                 minimizer: Minimizer = ScipyMinimizer(),
```

```
                t0: Union[float, np.ndarray] = 0):
        super().__init__(sample=sample, param_bounds=param_bounds,
                         dt=dt, model=density.model,
                         minimizer=minimizer, t0=t0)
        self._density = density

    def log_likelihood_negative(self, params: np.ndarray) -> float:
        self._model.params = params
        return -np.sum(np.log(np.maximum(self._min_prob,
                                         self._density(x0=self._sample[:-1],
                                                       xt=self._sample[1:],
                                                       t0=self._t0,
                                                       dt=self._dt))))
```

In particular, '`AnalyticalMLE`' is given a '`TransitionDensity`' of some form (such as one of the classes listed in Table 8), which it uses to compute the negative log likelihood. We use this class when performing maximum likelihood estimation with each of the density approximation methods.

Remark: We note that by default, the '`AnalyticalMLE`' is constructed with a '`Minimizer`' of type '`ScipyMinimizer`', which simply wraps the numerical minimization routines provided by **scipy.optimize.minimize** to enable out-of-the-box maximum likelihood estimation. However, this form of "dependency injection" also allows the user to supply their own custom or preferred optimizer, which is a key way that we achieve the "replaceable parts" design. The only requirement is that it implements the '`Minimizer`' interface, which is easy to accomplish by writing a wrapper on top of the user's desired minimizer. In this way, the '`AnalyticalMLE`' class is just a marriage between a '`TransitionDensity`' and a '`Minimizer`', either or both of which can be customized by the user as desired.

Another example of the '`LikelihoodEstimator`' class is given by the child '`CTMCEstimator`', which implements the CTMC approximation approach discussed in 3.7. The next section will illustrate both estimation procedures.

4.5. Example usage

In this section, we provide an extended example to demonstrate the usage of the **pymle** package, and to tie together the various components from simulation to inference. The model of interest is the OU model (see Uhlenbeck and Ornstein (1930)) whose dynamics is given by

$$dS_t = \kappa(\mu - S_t)dt + \sigma dW_t.$$

Among its numerous applications, OU is commonly used to model the instantaneous short interest rate (Vasicek (1977)) in economics, as well as commodity prices Schwartz (1997). Like the CIR model, its applications are ubiquitous, see for example Zhang, Grzelak, and Oosterlee (2012); Brignone, Kyriakou, and Fusai (2021); Kirkby (2023). Its true transition density $p(s'|s;\theta)$ is Gaussian,

$$p(s'|s,\Delta) = \frac{1}{\sigma_\Delta\sqrt{2\pi}}\exp\left(-\frac{(s'-\mu_\Delta(s))^2}{2\sigma_\Delta^2}\right),$$

with mean $\mu_\Delta(s) := \mu + (s-\mu)e^{-\kappa\Delta}$, and variance $\sigma^2_\Delta := \frac{\sigma^2}{2\kappa}(1-e^{-2\kappa\Delta})$. For this model, there are three parameters $0 < \kappa, \mu \in (-\infty, +\infty), 0 < \sigma$ that are needed to estimate from a given sample. We choose this model as it demonstrates the use of all maximum likelihood schemes currently supported in the package: Euler, Elerian, Ozaki, Shoji-Ozaki, Hermite polynomial expansion, Kessler approximation, and continuous-time Markov chain approximation.

Path simulation

To illustrate, we will simulate a data set, where we control the parameters of the true population: $S_0 = 0.4$ (the initial value of process), $\kappa = 3$ (the rate of mean reversion), $\mu = 0.3$ (the long term level of process), $\sigma = 0.2$ (volatility). We start by importing the basic dependencies, and initializing the model with the desired parameters:

```
from pymle.models import OrnsteinUhlenbeck
import numpy as np

S0 = 0.4
kappa = 3
mu = 0.3
sigma = 0.2

model = OrnsteinUhlenbeck()
model.params = np.array([kappa, mu, sigma])
```

Next we simulate a sample path of S_t on $0 \leq t \leq T = 5$ with the time step size $\Delta = 1/250$:

```
from pymle.sim.Simulator1D import Simulator1D

T = 5
freq = 250
dt = 1. / freq
seed = 123
```

For now, we simply create a default instance of the simulator class, and return to the topic in more detail Section 4.10:

```
simulator = Simulator1D(S0, T * freq, dt, model).set_seed(seed)
sample = simulator.sim_path()
```

We can plot the sample path, as follows, which is displayed in Figure 1:

```
import matplotlib.pyplot as plt
plt.plot(sample)
plt.xlabel('t')
plt.ylabel(r' $S_t$')
plt.show()
```

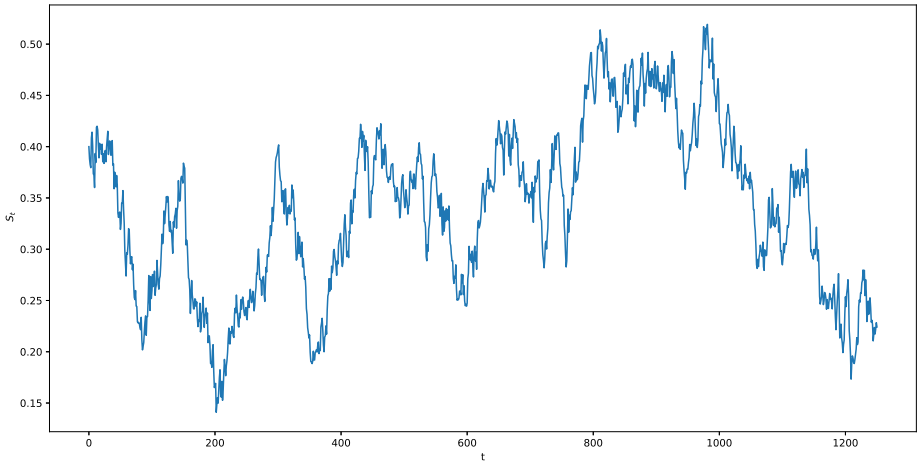

Figure 1: OU sample path

Maximum likelihood estimation

To fit using any of the estimation procedures, we must specify the parameter bounds we wish the optimizer to enforce during the fit, along with an initial guess for the parameters κ, μ, σ, respectively:

```
param_bounds = [(0.01, 10), (0, 4), (0.01, 1)]
guess = np.array([1, 0.1, 0.4])
```

Each tuple in `param_bounds` corresponds to a parameter, in order. We can then fit using Euler's approximation,

```
from pymle.fit.AnalyticalMLE import AnalyticalMLE
from pymle.core.TransitionDensity import EulerDensity

euler_est = AnalyticalMLE(sample, param_bounds, dt,
                          density=EulerDensity(model)).estimate_params(guess)
print(euler_est)
```

The previous lines construct an '`AnalyticalMLE`' estimator, using an '`EulerDensity`' approximation for the model, where '`EulerDensity`' is constructed with an instance of the model. The result of the maximum likelihood estimation is an object of type '`EstimatedResult`', and it contains the estimated parameters, as well as information about the goodness of fit, such as AIC and BIC. The output of `print(euler_est)` is:

```
params      | [2.91552452 0.32770741 0.19994576]
sample size | 1250
likelihood  | 3689.37631736423
AIC         | -7372.75263472846
BIC         | -7357.359938237571
```

Note that we have supplied the argument `dt` when constructing the estimator, indicating a uniform sample. Alternatively, in the case of a non-uniform sample, or when the model itself has time-dependent coefficients, we can supply a full array of time steps `dt`, as well as the initial starting time `t0` of the sample.

Fitting alternative approximations is equally simple, for example:

```
from pymle.core.TransitionDensity import *

ozaki_est = AnalyticalMLE(sample, param_bounds, dt,
                  density=OzakiDensity(model)).estimate_params(guess)

shoji_ozaki_est = AnalyticalMLE(sample, param_bounds, dt,
                  density=ShojiOzakiDensity(model)).estimate_params(guess)

kessler_est = AnalyticalMLE(sample, param_bounds, dt,
                  density=KesslerDensity(model)).estimate_params(guess)

AitSahalia_est = AnalyticalMLE(sample, param_bounds, dt,
                  density=AitSahaliaDensity(model)).estimate_params(guess)
```

Printing the final result for example, `print(AitSahalia_est)`, yields:

```
    params      | [2.93266185 0.32770741 0.20111964]
    sample size | 1250
    likelihood  | 3689.376317299021
    AIC         | -7372.752634598042
    BIC         | -7357.3599381071535
```

Note that the only difference for each estimation is that we supply a different density estimator class which fully captures the differences in MLE estimation procedures.

Remark: It is important to note that the exact numerical results presented in this work are dependent on the computer architecture of the machine used to run the experiments. All experiments in this work are conducted in Python 3.8 using a Windows machine with an Intel(R) Core(TM) i709750H CPU @2.60GHz. Due to the nature of the problem we are studying (the numerical optimization of functions that depend on numerical derivatives and potentially special functions), the reader should expect some small differences in results when reproducing the examples in this work.

CTMC-based estimation

The package also supports the recent CTMC approximation method of Kirkby *et al.* (2022), described in Section 3.7. The main classes for CTMC estimation are summarized in Table 9.

In this case, we construct a state space for the CTMC (e.g., with 350 states) based on the supplied sample (collected data, or simulated as in previous examples). We then bin the sample path so that the continuous points in the sample are assigned to the nearest bins:

Class	folder	Purpose
'StateSpace'	ctmc	Represents state space of CTMC
'Generator1D'	ctmc	Represents the transition generator matrix
'CTMCEstimator'	ctmc	Manages the estimation of CTMC parameters

Table 9: List of key CTMC estimation classes in the **pymle** package.

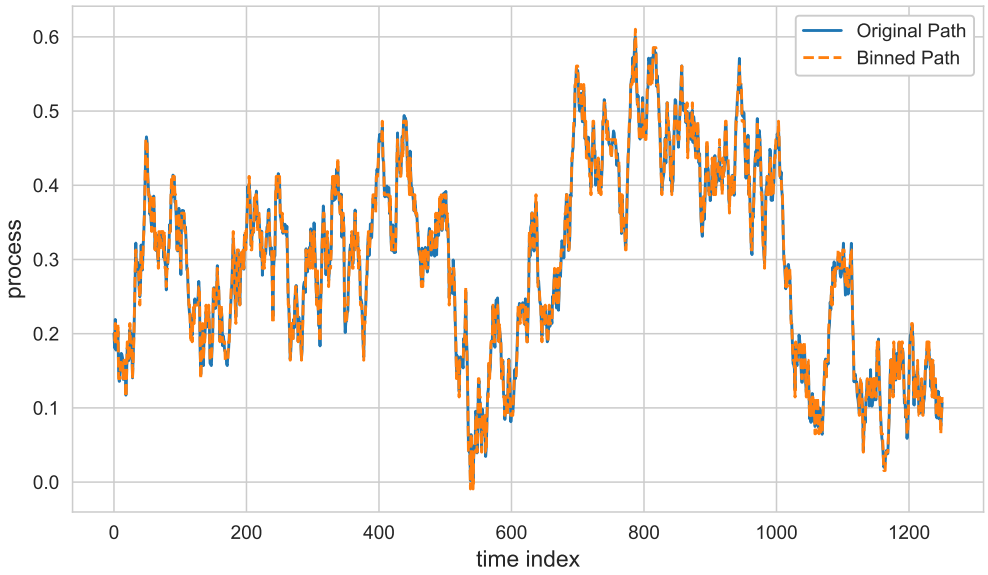

Figure 2: Simulated path of OU process with $S_0 = 0.2, \kappa = 4.0, \theta = 0.2, \sigma = 0.4$, with $T = 5$, $\Delta = 1/250$, along with the Binned Path (with 30 CTMC states).

```
from pymle.ctmc.StateSpace import StateSpace
state_space = StateSpace.from_sample(sample, is_positive=True, N_states=350)
binned_path, state_index = state_space.bin_path(sample)
```

Figure 2 displays the path of a simulated OU process, along with the binned sampled using 30 states for illustration. The CTMC generator is then constructed as in (13) using

```
generator = Generator1D(model)
generator.states = state_space.states
```

Finally, we construct a 'CTMCEstimator' based on the binned sample, and estimate in the usual fashion:

```
ctmc_est = CTMCEstimator(binned_path, state_index, dt, generator,
                         param_bounds).estimate_params(guess)
print(ctmc_est)
```

The print statement shows the CTMC estimation result:

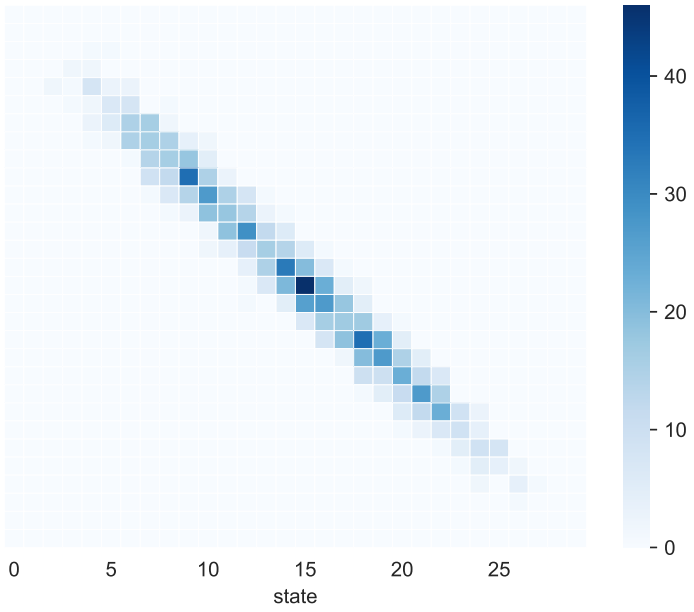

Figure 3: Transition counts matrix from simulated process with $S_0 = 0.2, \kappa = 4.0, \theta = 0.2, \sigma = 0.4$, with $T = 5$, $\Delta = 1/250$, and 30 CTMC states.

```
params      | [4.3003113  0.24962709 0.40240265]
sample size | 1250
likelihood  | 2825.764354953858
AIC         | -5645.528709907716
BIC         | -5630.136013416827
```

Figure 3 shows the transition counts matrix $\mathbf{C}(\Delta)_{i,j}$ corresponding to (16), where we use 30 CTMC states for illustration. The code to generate the plot is as follows:

```
def plot_CTMC_Counts_C_Matrix(estimator: CTMCEstimator):
    import seaborn as sns
    C = pd.DataFrame(estimator.transition_counts)
    ax = sns.heatmap(C, cmap=sns.color_palette("Blues", as_cmap=True),
                     linewidths=.5, yticklabels=False, xticklabels=5)
    plt.xlabel('state')
    plt.show()
```

Comparison

A comparison of results for each of the estimators is given in Table 10. In this table, the true parameters $\kappa = 3, \mu = 0.3, \sigma = 0.2$ are estimated by $\widehat{\kappa}, \widehat{\mu}, \widehat{\sigma}$. The sample size, the value of the likelihood function, AIC, and BIC corresponding to $(\widehat{\kappa}, \widehat{\mu}, \widehat{\sigma})$ are also reported. For this example, the alternative methods produce very similar parameter estimates. In general,

the Hermite polynomial method of Aït-Sahalia (2002) is considered to be state-of-the-art, and is often used when available. Given that we know the true parameters in this controlled experiment, we also include a column "RMSE" which provides the Root Mean Squared Error of the estimated parameters compared with the true model parameters. This example shows the promise of the CTMC method, which is often quite accurate, albeit slow computationally.

Method	$\widehat{\kappa}$	$\widehat{\mu}$	$\widehat{\sigma}$	Likelihood	AIC	BIC	RMSE
Exact MLE	2.93268	0.32771	0.20112	3689.37632	-7372.75263	-7357.35994	0.04204
Euler	2.91552	0.32771	0.19995	3689.37632	-7372.75263	-7357.35994	0.05133
Ozaki	2.85870	0.32690	0.19994	3689.21485	-7372.4297	-7357.03701	0.08305
Shoji-Ozaki	2.93295	0.32771	0.20112	3689.37632	-7372.75263	-7357.35994	0.04189
Kessler	2.93136	0.32764	0.19994	3689.37556	-7372.75111	-7357.35842	0.04272
Ait-Sahalia	2.93266	0.32771	0.20112	3689.37632	-7372.75263	-7357.35994	0.04205
CTMC	2.94352	0.32772	0.20144	3687.46903	-7368.93807	-7353.54537	0.03633

Table 10: OU estimation results, sample size of 1250. Exact values $\kappa = 3, \mu = 0.3, \sigma = 0.2$. Values in table are rounded to the nearest fifth decimal.

4.6. Real data example

In this section, we provide an example to demonstrate how to fit an SDE based on historical interest rate data. The data consists of daily observations U.S. / Euro Foreign Exchange Rate [DEXUSEU], retrieved from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DEXUSEU, May 24, 2021. A plot of the data is given in Figure 4, which shows the cost (in USD) at which one Euro may be purchased.

The model we choose to fit is CIR, which captures the non-negativity and mean-reverting tendency of FX rates:

$$dS_t = \kappa(\mu - S_t)dt + \sigma\sqrt{S_t}dW_t.$$

We illustrate the use of five Maximum Likelihood estimators as shown in Table 11. to estimate the unknown parameters: κ, μ, σ. The Python code is given below:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from pymle.models.CIR import CIR
from pymle.core.TransitionDensity import *
from pymle.fit.AnalyticalMLE import AnalyticalMLE
import seaborn as sns
import matplotlib.dates as mdates
import datetime

sns.set_style('whitegrid')
```

As a first step, we read the data into a **pandas** '`DataFrame`' using the built-in data loader for this example:

```
from pymle.data.loader import load_FX_USD_EUR
df = load_FX_USD_EUR()
print(df.head())
```

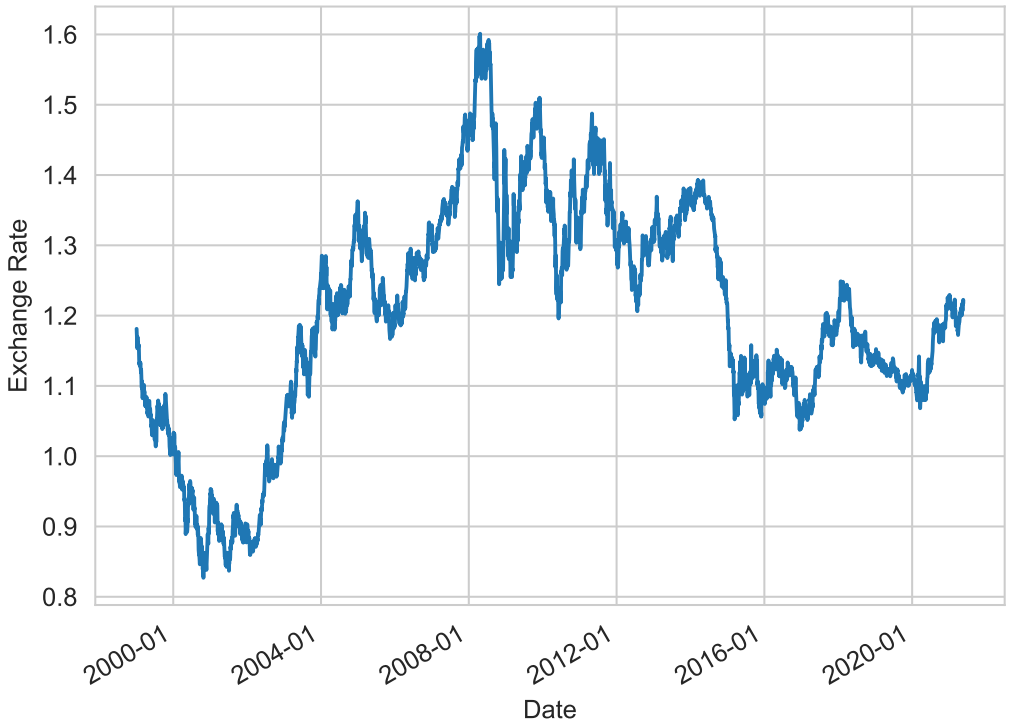

Figure 4: FX rates - EUR/USD from 1999 to 2021.

which generates the output (from the first 5 rows):

```
       Date    Rate
0  1/4/1999  1.1812
1  1/5/1999  1.1760
2  1/6/1999  1.1636
3  1/7/1999  1.1672
4  1/8/1999  1.1554
```

To generate Figure 4 with properly formatted dates we set '`skip=20`' to change to sample time series at a different frequency, we then run the following:

```
skip = 20
dt = skip / 252.
sample = df['Rate'].values[:-1:skip]

df['Date'] = [datetime.datetime.strptime(d, "%m/%d/%Y").date()
                for d in df['Date']]
```

```
fig, ax = plt.subplots()
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y-%m'))
ax.plot(df['Date'].values, df['Rate'].values)
plt.xlabel('Date')
plt.ylabel('Exchange Rate')
fig.autofmt_xdate()

plt.show()
```

We then initialize the CIR model as follows, and supply an initial guess for κ, μ, σ and parameter bounds for the fit:

```
model = CIR()
guess = np.asarray([.24, 1.0, 0.1])
param_bounds = [(0.01, 5), (0.01, 2), (0.01, 0.9)]
```

Fitting the estimators proceeds in the same manner as the previous example. The estimated results from running the above code is given in Table 11 for three of the supported estimators for illustration.

Remark: As discussed in Section 3.2, a more numerically stable version of the Elerian approximation is implemented in **pymle**. For this CIR example, the original form of the Elerian density approximation in (9) produces a numerical overflow and convergence failure during the maximum likelihood optimization, whereas the version in (10) converges in line with the alternative estimators. For experimentation purposes, we have exposed a parameter `use_stable_form` (defaulting to True) in the '`ElerianDensity`' class that the user can set to False in order to experiment with this observed instability.

Method	$\widehat{\kappa}$	$\widehat{\mu}$	$\widehat{\sigma}$	Likelihood	AIC	BIC
Exact MLE	0.25206	1.20534	0.10720	561.23301	-1116.46602	-1105.56165
Euler	0.24456	1.20544	0.10616	561.16121	-1116.32242	-1105.41805
Kessler	0.24663	1.20489	0.10592	561.13577	-1116.27154	-1105.36717
Elerian	0.25167	1.2053	0.10613	561.26081	-1116.52162	-1105.61725
Ait-Sahalia	0.25159	1.2053	0.10720	561.22655	-1116.45310	-1105.54873

Table 11: CIR parameter estimation results for EUR/USD example, sample size of 280.

4.7. Adding your own model

The **pymle** package provides its users great flexibility in adding new models with minimal coding. Figure 5 gives an overview of all steps the user needs to do in order to add a new model into the **pymle** package.

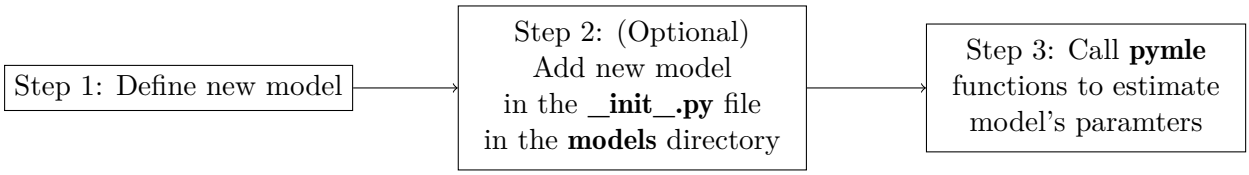

Figure 5: Adding a new model

In the following, we will provide an example with all necessary details. More specifically, we consider the problem of estimating the parameters of Inhomogeneous Geometric Brownian Motion (IGBM), see Abadie and Chamorro (2008); Zhao (2009). The dynamics of IGBM process is given by

$$dS_t = \kappa(\mu - S_t)dt + \sigma S_t dW_t$$

It is noted that the exact transition density of S_t is not available in an easy-to-use form. Hence, numerical approximations are employed in order to estimate the unknown parameters κ, μ, σ. In the below, we details all the steps the user needs to do in order to add his/her new model into the package.

Step 1

The model must be defined to extend the '`Model1D`' class. That is, the dynamics of the model must be specified by overriding the '`Model1D`' methods. If desired to extend the package directly, you can add this model to the **models** directory (otherwise, skip step 2). Name and save your new model in its own module, for example, **IGBM.py**.

```
from typing import Union
import numpy as np
from pymle.core.Model import Model1D

class IGBM(Model1D):
    """
    Model for inhomogeneous geometric Brownian motion
    Parameters: [kappa, mu, sigma]

    dX(t) = mu(X,t)*dt + sigma(X,t)*dW_t

    where:
        mu(X,t)    = kappa(mu-X(t))
        sigma(X,t) = sigma X(t)
    """
    def drift(self,
              x: Union[float, np.ndarray],
              t: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        return self._params[0] * (self._params[1] - x)

    def diffusion(self,
```

```
                x: Union[float, np.ndarray],
                t: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        return self._params[2] * x
```

We note that the two primary functions to override for full compatability are `drift()` and `diffusion()`. Other methods can be overridden as desired, for example, to provide analytical expressions for the derivatives of the `drift()` and `diffusion()` terms. These are not necessary, as numerical derivatives are implemented by default, but it can help to speed up the fitting process.

Step 2

If you desire to extend the **pymle** package directly: import your newly defined model in the **__init__.py** file in the **models** directory:

```
from pymle.models.IGBM import IGBM.
```

Step 3

After following steps 1-2, you are done! The model is now fully compatible with the fitting and simulation framework, and the final step is simply to use your model in a same manner as the previous examples.

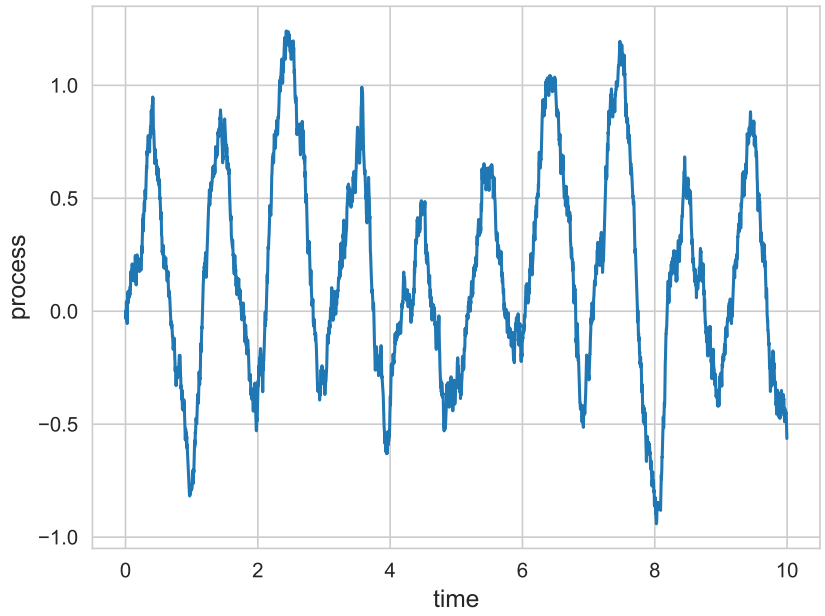

Figure 6: Simulated path of time-dependent process in (19) with $S_0 = 0.0, \kappa = 2, \gamma = 2, \sigma = 0.5$, and $T = 10$, $\Delta = 1/365$.

4.8. Time-Dependent Example

We next illustrate the application of **pymle** for estimating the parameters of a diffusion process with time-dependent (non-homogenous) coefficients. For illustration, we posit the following three-parameter process with a periodic level to which the process mean-reverts:

$$dS_t = \kappa \cdot (\gamma \cdot \sin(t \cdot 2\pi) - S_t)dt + \sigma dW_t, \tag{19}$$

where κ controls the strength of mean-reversion, γ the peak and trough level, and σ the volatility.

The code for defining this process is quite simple, requiring only that we implement the definition of the drift and diffusion terms of the process. We therefore define a new class '`BrownianMotion_SinLevel`' which extends '`Model1D`' as follows:

```
from pymle.core.Model import Model1D
from typing import Union
import numpy as np

class BrownianMotion_SinLevel(Model1D):
    def drift(self, x: Union[float, np.ndarray], t: float):
        return self._params[0] * (self._params[1] * np.sin(t * 2 * np.pi) - x)

    def diffusion(self, x: Union[float, np.ndarray], t: float):
        return self._params[2] * (x > -np.inf)
```

Note how little code is required here to define a new model that is fully compatible with the inference and simulation framework in **pymle**. For example, we can simulate a sample from this process over a 10 year period, sampled daily:

```
from pymle.sim.Simulator1D import Simulator1D

S0 = 0.0
kappa = 2
gamma = 2
sigma = 0.5

model = BrownianMotion_SinLevel()
model.params = np.array([kappa, gamma, sigma])

T = 10
freq = 365
dt = 1. / freq
seed = 123

simulator = Simulator1D(S0=S0, M=T * freq, dt=dt, model=model).set_seed(seed=seed)
sample = simulator.sim_path()
```

To plot the simulated path, which results in Figure 6, we use:

```
import seaborn as sns
sns.set_style('whitegrid')
```

```
ts = np.linspace(start=0, stop=T - dt, num=T * freq)

plt.plot(ts, sample[:-1])
plt.xlabel('time', fontsize=12)
plt.ylabel('process', fontsize=12)
plt.show()
```

From Figure 6, the periodicity of the time-varying level is clearly observed, and we can estimate the parameters of the process as follows:

```
param_bounds = [(0.5, 7), (0.01, 5), (0.001, 2)]
guess = np.array([3, 1.5, 0.7])

euler_est = AnalyticalMLE(sample=sample, param_bounds=param_bounds, dt=dt,
                          t0=ts,
                          density=EulerDensity(model)).estimate_params(guess)

print(f'\nEuler MLE: {euler_est} \n')
```

The only real difference in the previous step from the case of time-homogeneous coefficients is that we now pass the time vector to the '`AnalyticalMLE`' constructor. Otherwise, the procedure is identical. The final line above prints the following Euler estimate:

```
params      | [1.85962458 2.0123528  0.50838796]
sample size | 3650
likelihood  | 8057.450002072974
AIC         | -16108.900004145948
BIC         | -16090.292556806218
```

The estimated coefficients are quite close to the true parameters, $[2, 2, 0.5]$. The Kessler density estimate, for example, is produced analogously, and provides:

```
params      | [1.86482026 2.01202043 0.50829539]
sample size | 3650
likelihood  | 8057.455150822385
AIC         | -16108.91030164477
BIC         | -16090.30285430504
```

4.9. Numerical Derivatives

We note that each extension of the '`Model1D`' class automatically inherits methods for computing the numerical derivatives of the diffusion and drift coefficients. For example, `drift_x` provides the first derivative of $\mu(S, t)$ with respect to S. This is implemented by default as a central difference with step size controlled by `h_x`, which defaults to $1e-05$. The user can override this value easily, for example `model.h_x = 0.001`. The other numerical derivatives are also defined with default step sizes, each of which can be controlled by the user in a similar

fashion. In cases where the derivatives are known analytically, which is true for many common models (such as Brownian motion), greater accuracy and efficiency can be achieved by overriding the derivative methods with the closed-form expressions. However, by providing a default implementation for each, the user can quickly add a model with few lines of additional code.

4.10. Simulation schemes

The **pymle** package supports each of the simulation schemes listed in Table 4. These are implemented as children of the '`Stepper`' class, which requires the user to simply override the `next()` method of this base class:

```
class Stepper(ABC):
    def __init__(self, model: Model1D):
        self._model = model

    @property
    def model(self) -> Model1D:
        return self._model

    @abstractmethod
    def next(self,
             t: float,
             dt: float,
             x: Union[float, np.ndarray],
             dZ: Union[float, np.ndarray]) -> Union[float, np.ndarray]:
        raise NotImplementedError
```

The `next()` method, whose purpose is to evolve the current state forward one step over a time increment dt, requires four parameters as follows:

- `t`: float, the current time of the process during the simulation.
- `dt`: float, the increment between now and the next state transition.
- `x`: float or array, the current state.
- `dZ`: float or array, independent normal random variates, $N(0, 1)$, used to evolve the current state forward.

As a simple example, the '`EulerStepper`' implements the Euler simulation scheme as follows:

```
class EulerStepper(Stepper):
    def __init__(self, model: Model1D):
        super().__init__(model=model)

    def next(self,
             t: float,
```

```
            dt: float,
            x: Union[float, np.ndarray],
            dZ: Union[float, np.ndarray]) -> Union[float, np.ndarray]:

        xp = x + self._model.drift(x, t) * dt \
            + self._model.diffusion(x, t) * np.sqrt(dt) * dZ

        return np.maximum(0., xp) if self._model.is_positive else xp
```

Paths are then simulated using the '`Simulator1D`' class, to which one provides either a '`Stepper`' or the name of one of the built in '`Stepper`' classes (e.g., "Euler"). This provides several common simulation schemes, but allows the user to customize their own simulation approach as desired. An example usage was previously provided in Section 4.5.

4.11. Next steps

While **pymle** offers a wide range of support for common SDE applications, there are several areas of interest for extending its functionality. Adding support for sticky diffusions Meier, Li, and Zhang (2021); Meier *et al.* (2023), diffusions with non-smooth/discontinuous coefficients Lejay and Martinez (2006); Zhang and Li (2022); Ding and Cui (2022), multivariate diffusions Brouste *et al.* (2014), subordinated diffusions Guo and Li (2019), and diffusion bridges Bladt, Finch, and Sørensen (2016); Chau, Kirkby, Nguyen, Nguyen, Nguyen, and Nguyen (2023) are all promising directions for further development as the library continues to evolve, and we welcome contributions and collaborations in the development to come. The additional support of jumps as well as fractional Brownian motion, as currently offered in the R package **yuima**, are also of interest for growth of the library.

5. Summary

We presented the **pymle** package for SDE simulation and inference in Python. The **pymle** package relies on an object-oriented design, to facilitate its extensibility. We take a "replaceable parts" approach to the design, providing built-in functionality with the ability to swap out components for maximal customization. The addition of new models, estimation methods, minimization routines, and simulation schemes is very simple with minimal additional code required. For example, simulation and parameter estimation interact with generic '`Model1D`' objects, so once the minimal interface is added for a new model, it automatically inherits fitting and simulation functionality out-of-the-box. This makes **pymle** ideal as a testing ground for new approaches in statistical inference of SDE, as well as a ready-to-use package to support typical estimation requirements.

References

Abadie LM, Chamorro JM (2008). "Valuing flexibility: the case of an integrated gasification combined cycle power plant." *Energy Economics*, **30**(4), 1850–1881.

Ahn DH, Gao B (1999). "A parametric nonlinear model of term structure dynamics." *The Review of Financial Studies*, **12**(4), 721–762.

Aït-Sahalia Y (1995). "Nonparametric pricing of interest rate derivative securities." *Technical report*, National Bureau of Economic Research.

Ait-Sahalia Y (1996). "Testing continuous-time models of the spot interest rate." *The Review of Financial Studies*, **9**(2), 385–426.

Aït-Sahalia Y (2002). "Maximum likelihood estimation of discretely sampled diffusions: a closed-form approximation approach." *Econometrica*, **70**(1), 223–262.

Aït-Sahalia Y (2008). "Closed-form likelihood expansions for multivariate diffusions." *Annals of Statistics*, **36**(2), 906–937.

Aït-Sahalia Y, Kimmel R (2007). "Maximum likelihood estimation of stochastic volatility models." *Journal of Financial Economics*, **83**(2), 413–452.

Aït-Sahalia Y, Kimmel RL (2010). "Estimating affine multifactor term structure models using closed-form likelihood expansions." *Journal of Financial Economics*, **98**(1), 113–144.

Black F, Scholes M (1973). "The pricing of options and corporate liabilities." *Journal of Political Economy*, **81**(3), 637–654.

Bladt M, Finch S, Sørensen M (2016). "Simulation of multivariate diffusion bridges." *Journal of the Royal Statistical Society B: Statistical Methodology*, pp. 343–369.

Brignone R, Kyriakou I, Fusai G (2021). "Moment-matching approximations for stochastic sums in non-Gaussian Ornstein–Uhlenbeck models." *Insurance: Mathematics and Economics*, **96**, 232–247.

Brouste A, Fukasawa M, Hino H, Iacus S, Kamatani K, Koike Y, Masuda H, Nomura R, Ogihara T, Shimuzu Y, *et al.* (2014). "The yuima project: A computational framework for simulation and inference of stochastic differential equations." *Journal of Statistical Software*, **57**, 1–51.

Brown R (1828). "XXVII. A brief account of microscopical observations made in the months of June, July and August 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies." *The Philosophical Magazine*, **4**(21), 161–173.

Carley M (2013). "Numerical solution of the modified Bessel equation." *IMA Journal of Numerical Analysis*, **33**(3), 1048–1062.

Chan KC, Karolyi GA, Longstaff FA, Sanders AB (1992). "An empirical comparison of alternative models of the short-term interest rate." *The Journal of Finance*, **47**(3), 1209–1227.

Chau H, Kirkby J, Nguyen D, Nguyen D, Nguyen N, Nguyen T (2023). "An efficient method to simulate diffusion bridges." *Working Paper, Submitted*.

Choi S (2013). "Closed-form likelihood expansions for multivariate time-inhomogeneous diffusions." *Journal of Econometrics*, **174**(2), 45–65.

Cox JC (1996). "The constant elasticity of variance option pricing model." *Journal of Portfolio Management*, p. 15.

Cox JC, Ingersoll Jr JE, Ross SA (2005). "A theory of the term structure of interest rates." In *Theory of Valuation*, pp. 129–164. World Scientific.

Cui Z, Kirkby JL, Nguyen D (2021a). "A data-driven framework for consistent financial valuation and risk measurement." *European Journal of Operational Research*, **289**(1), 381–398.

Cui Z, Kirkby JL, Nguyen D (2021b). "Efficient simulation of generalized SABR and stochastic local volatility models based on Markov chain approximations." *European Journal of Operational Research*, **290**(3), 1046–1062.

Ding K, Cui Z (2022). "A General Framework to Simulate Diffusions with Discontinuous Coefficients and Local Times." *ACM Transactions on Modeling and Computer Simulation*, **32**(4), 1–29.

Dixit AK, Pindyck RS, Pindyck R (1994). "Investment under uncertainty princeton univ." *Press, Princeton, New Jersey*.

Eberlein E, Keller U (1995). "Hyperbolic distributions in finance." *Bernoulli*, **1**(3), 281–299.

Egorov AV, Li H, Xu Y (2003). "Maximum likelihood estimation of time-inhomogeneous diffusions." *Journal of Econometrics*, **114**(1), 107–139.

Elerian O (1998). "A note on the existence of a closed form conditional transition density for the Milstein scheme." *Economics Discussion Paper*, p. W18.

Eraker B (2001). "MCMC analysis of diffusion models with application to finance." *Journal of Business & Economic Statistics*, **19**(2), 177–191.

Florens D (1998). "Estimation of the diffusion coefficient from crossings." *Statistical Inference for Stochastic Processes*, **1**(2), 175–195.

Forman JL, Sørensen M (2008). "The Pearson diffusions: A class of statistically tractable diffusion processes." *Scandinavian Journal of Statistics*, **35**(3), 438–465.

Gallant AR, Tauchen G (1996). "Which moments to match?" *Econometric theory*, pp. 657–681.

Glasserman P (2013). *Monte Carlo Methods in Financial Engineering*, volume 53. Springer-Verlag.

Gourieroux C, Monfort A, Renault E (1993). "Indirect inference." *Journal of Applied Econometrics*, **8**(S1), S85–S118.

Grasselli M (2017). "The 4/2 stochastic volatility model: A unified approach for the Heston and the 3/2 model." *Mathematical Finance*, **27**(4), 1013–1034.

Guidoum AC, Boukhetala K (2020). "Performing Parallel Monte Carlo and Moment Equations Methods for Itô and Stratonovich Stochastic Differential Systems: R Package Sim. DiffProc." *Journal of Statistical Software*, **96**(1), 1–82.

Guo W, Li L (2019). "Parametric inference for discretely observed subordinate diffusions." *Statistical Inference for Stochastic Processes*, **22**, 77–110.

Hansen LP, Scheinkman JA (1993). "Back to the future: Generating moment implications for continuous-time Markov processes."

Hastie T, Tibshirani R, Friedman JH, Friedman JH (2009). *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer.

Heston SL (1993). "A closed-form solution for options with stochastic volatility with applications to bond and currency options." *The Review of Financial Studies*, **6**(2), 327–343.

Higham DJ (2001). "An algorithmic introduction to numerical simulation of stochastic differential equations." *SIAM Review*, **43**(3), 525–546.

Hsu YL, Lin T, Lee C (2008). "Constant elasticity of variance (CEV) option pricing model: Integration and detailed derivation." *Mathematics and Computers in Simulation*, **79**(1), 60–71.

Hurn S, Lindsay K, Xu L (2021). "A Comparative Study of Likelihood Approximations for Univariate Diffusions." *Journal of Financial Econometrics*, pp. 1–28.

Iacus SM (2007). "sde: Simulation and inference for stochastic differential equations."

Iacus SM (2009). *Simulation and inference for stochastic differential equations: with R examples*. Springer Science & Business Media.

Iacus SM, Yoshida N (2018). "Simulation and inference for stochastic processes with YUIMA." *A comprehensive R framework for SDEs and other stochastic processes. Use R*.

Jazwinski AH (2007). *Stochastic processes and filtering theory*. Courier Corporation.

Jones CS (1997). "Bayesian analysis of the short-term interest rate." *Technical report*, Working paper, The Wharton School, University of Pennsylvania.

Kalbfleisch J, Lawless JF (1985). "The analysis of panel data under a Markov assumption." *Journal of the American Statistical Association*, **80**(392), 863–871.

Karatzas I, Shreve S (2014). *Brownian motion and stochastic calculus*, volume 113. springer.

Kessler M (1997). "Estimation of an ergodic diffusion from discrete observations." *Scandinavian Journal of Statistics*, **24**(2), 211–229.

Kessler M, Sørensen M (1999). "Estimating equations based on eigenfunctions for a discretely observed diffusion process." *Bernoulli*, **5**(2), 299–314.

Kirkby JL (2023). "Hybrid equity swap, cap, and floor pricing under stochastic interest by Markov chain approximation." *European Journal of Operational Research*, **305**(2), 961–978.

Kirkby JL, Nguyen D (2020). "Efficient Asian option pricing under regime switching jump diffusions and stochastic volatility models." *Annals of Finance*, **16**(3), 307–351.

Kirkby JL, Nguyen DH, Nguyen D, Nguyen NN (2022). "Maximum likelihood estimation of diffusions by continuous time Markov chain." *Computational Statistics & Data Analysis*, **168**, 107408.

Kloeden PE, Platen E (1992). "Stochastic differential equations." In *Numerical Solution of Stochastic Differential Equations*, pp. 103–160. Springer.

Lejay A, Martinez M (2006). "A scheme for simulating one-dimensional diffusion processes with discontinuous coefficients."

McGibbon RT, Pande VS (2015). "Efficient maximum likelihood parameterization of continuous-time Markov processes." *The Journal of Chemical Physics*, **143**(3), 034109.

McKinney W, *et al.* (2011). "pandas: a foundational Python library for data analysis and statistics." *Python for High Performance and Scientific Computing*, **14**(9), 1–9.

Meier C, Li L, Zhang G (2021). "Markov chain approximation of one-dimensional sticky diffusions." *Advances in Applied Probability*, **53**(2), 335–369.

Meier C, Li L, Zhang G (2023). "Simulation of multidimensional diffusions with sticky boundaries via Markov chain approximation." *European Journal of Operational Research*, **305**(3), 1292–1308.

Mil'shtein G (1979). "A method of second-order accuracy integration of stochastic differential equations." *Theory of Probability & Its Applications*, **23**(2), 396–401.

Ozaki T (1985). "2 Non-linear time series models and dynamical systems." *Handbook of Statistics*, **5**, 25–83.

Ozaki T (1992). "A bridge between nonlinear time series models and nonlinear stochastic dynamical systems: a local linearization approach." *Statistica Sinica*, pp. 113–135.

Ozaki T (1993). "A local linearization approach to nonlinear filtering." *International Journal of Control*, **57**(1), 75–96.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, *et al.* (2011). "Scikit-learn: Machine learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

Pfeuffer M (2017). "ctmcd: An R Package for Estimating the Parameters of a Continuous-Time Markov Chain from Discrete-Time Data." *R Journal*, **9**(2).

Rydin Gorjão L, Witthaut D, Lind PG (2023). "jumpdiff: A Python Library for Statistical Inference of Jump-Diffusion Processes in Observational or Experimental Data Sets." *Journal of Statistical Software*, **105**(1), 1–22. doi:10.18637/jss.v105.i04. URL https://www.jstatsoft.org/index.php/jss/article/view/v105i04.

Schwartz ES (1997). "The stochastic behavior of commodity prices: Implications for valuation and hedging." *The Journal of Finance*, **52**(3), 923–973.

Seabold S, Perktold J (2010). "Statsmodels: Econometric and statistical modeling with python." In *Proceedings of the 9th Python in Science Conference*, volume 57, p. 61. Austin, TX.

Shoji I (1998). "A comparative study of maximum likelihood estimators for nonlinear dynamical system models." *International Journal of Control*, **71**(3), 391–404.

Shoji I, Ozaki T (1997). "Comparative study of estimation methods for continuous time stochastic processes." *Journal of Time Series Analysis*, **18**(5), 485–506.

Shoji L (1995). *Estimation and inference for continuous time stochastic models*. Ph.D. thesis, PhD thesis, Institute of Statistical Mathematics, Tokyo.

Sigrist F, Künsch HR, Stahel WA (2015). "spate: An R Package for Spatio-Temporal Modeling with a Stochastic Advection-Diffusion Process." *Journal of Statistical Software*, **63**(14), 1–23. doi:10.18637/jss.v063.i14. URL https://www.jstatsoft.org/index.php/jss/article/view/v063i14.

Uhlenbeck GE, Ornstein LS (1930). "On the theory of the Brownian motion." *Physical Review*, **36**(5), 823.

Vasicek O (1977). "An equilibrium characterization of the term structure." *Journal of Financial Economics*, **5**(2), 177–188.

Zeileis A, Grothendieck G (2005). "zoo: S3 infrastructure for regular and irregular time series." *arXiv preprint math/0505527*.

Zhang B, Grzelak LA, Oosterlee CW (2012). "Efficient pricing of commodity options with early-exercise under the Ornstein–Uhlenbeck process." *Applied Numerical Mathematics*, **62**(2), 91–111.

Zhang G, Li L (2022). "Analysis of Markov chain approximation for diffusion models with nonsmooth coefficients." *SIAM Journal on Financial Mathematics*, **13**(3), 1144–1190.

Zhao B (2009). "Inhomogeneous geometric Brownian motion." *Available at SSRN 1429449*.

Affiliation:

J. L. Kirkby
School of Industrial and Systems Engineering
Georgia Institute of Technology
Atlanta, GA 30318, USA
Email: jkirkby3@gatech.edu

D.H. Nguyen
Department of Mathematics
University of Alabama
Tuscaloosa, AL 35487, USA
Email: dangnh.maths@gmail.com

D. Nguyen
Department of Mathematics
Marist College
Poughkeepsie, NY 12601, USA
Email: nducduy@gmail.com

N. Nguyen
Department of Mathematics and Applied Mathematical Sciences
University of Rhode Island
45 Upper College Rd, Kingston, RI 02881, USA
Email: nhu.nguyen@uri.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd