PyProcess 0.2

A pure Python framework to simulate stochastic processes exactly

Cameron Davidson-Pilon

December 14, 2012

Abstract

We introduce a new framework, *PyProcess*, to simulate stochastic process *exactly*, or if unable to simulate exactly, will fall back on accurate approximation schemes. The advantages of using *PyProcess* over other libraries is its clean, simple *API for Humans*, its intuitive integration into the established Python scientific stack, and access to bleeding-edge algorithms. The purpose of this monograph is to present the implementation details, the trickier algorithms used to simulate process exactly, and examples uses of *PyProcess*.

Contents

1 Introduction

We introduce a new framework, *PyProcess*, to simulate stochastic process *exactly*, or if unable to simulate exactly, will fall back on accurate approximation schemes. The advantages of using *PyProcess* over other libraries:

- its clean, simple *API for Humans*,
- its intuitve integration into the established Python scientific stack,
- access to bleeding-edge algorithms.
- the ability to condition a large family of process on an end condition, that is, to *pin* a distribution.
- it is written in pure Python.

The *PyProcess* library [1] is the first Python library to attack the problem of stochastic process simulation. Given how useful stochastic process simulations are in finance, Bayesian estimators, particle physics, systems biology, queuing theory, and other disciplines, the authors of *PyProcess* felt a framework was needed for quick simulation and experimentation in the Python scientific stack. We immediately approach a trade-off: Python is known for being slower than other languages, like Java, MATLAB and C++. On the other hand, learning Python and employing libraries in Python is incredibly easy. Thus, this project does not target the extremely computationally intensive end user, but the more moderate user. The main users of *PyProcess* are likely the researchers who know little about programming, or little about the more complex languages like Java or C++, and want to run smaller-scale experiments. The API was developed with this user in mind. Another benefit of choosing Python for this library was that Python is available free to all users, while languages like MATLAB have expensive overheads for institutions.

1.1 Current Development

The *PyProcess* library code, version 0.2, is currently hosted on the open-source repository Github website, for other developers to fork their own version and submit code back to the original source. Currently, the library can simulate the following processes:

- Step process (characterized by finite activity in any finite interval)
    - renewal process (characterized by a positive inter-arrival distribution and jump size distribution)
        - marked poisson process
        - poisson process
        - compound poisson process
- Diffusion process (characterized by a continuous stochastic differential equation)
    - Brownian motion

[1] *framework* and *library* will be used interchangeably

- Geometric Brownian motion
- CEV process
- CIR process
- Square Bessel process
- Ornstein Uhlenbeck process
- Time-Integrated Ornstein Uhlenbeck process
- Periodic-drift process

- Jump diffusions (characterized by discontinuous infinite activity processes)
  - Gamma process
  - Variance-gamma process
  - Geometric gamma process
  - Inverse Gaussian process
  - Normal Inverse Gaussian process

The library uses some of the latest results to be able to simulate the above processes *exactly*, that is, the samples are distributed according to the intended measure. Further information about the implemented algorithms are available in section 3.

2 *APIs for Humans*

As we have the end-user in mind, we wanted to create an API for the library that was very simple to use. In *PyProcess* , all processes are objects , and to create a process we use the following:

```
import pyprocess as pp
poisson = pp.Poisson_process(rate = 2)
brownian_motion = pp.Wiener_process(mu = 0, sigma = 2)
cir = pp.CIR_process(lambda_0 = 2, lambda_1 = 0.5, nu = 1)
```

creates a Poisson process with rate parameter 2, a Wiener process with 0 drift and twice normal volatility, and CIR process. Unfortunately, for many processes the parameterization is ambiguous. For this reason, included in the documentation is a very clear detailing of the parameterization used. For example, the CIR process is defined:

$$dC_t = (\lambda_0 - \lambda_1 C_t)dt + nu\sqrt{C_t}dB_t$$

When, available, the most popular parameterization is used. For diffusion models, we chose to parameterize their stochastic differential equation form:

$$dX_t = \mu(t)dt + \sigma(t)dW_t$$

as to avoid confusion. This is most relevant in cases like Geometric Brownian motion, where the parameterization can be either via the stochastic differential equation or the closed form expression.

2.0.1 Adding start conditions

Often one wants to start the process at a certain position and time, and this is possible by supplying the keywords *startTime* and *startPosition* into the initialization. For example, to start a Brownian motion at position -1, and time -1, we call

```
bm = pp.Weiner_process(mu = 0, sigma = 1, startTime = -1,
    startPosition = -1)
```

Without the arguments, the process defaults to typically the most natural setting (often starting at time 0 at position 0 or 1).

2.0.2 Adding end conditions

Similar to adding start conditions, one can add end conditions, $X_T = x_T$, just as easily. The *PyProcess* library will only sample from processes that satisfy the constraint $X_T = x_T$. Details on how sampling conditional processes is performed is in section 4. For example, to create a Brownian bridge, one specifies the *endTime* and *endPosition*:

```
bm = pp.Weiner_process( mu = 0, sigma = 1, startTime = 0,
    startPosition = 0, endTime = 1, endPosition = 0)
```

2.1 Sampling processes

All process instances expose fundamental methods, including:

- `sample_path(times, N = 1)`

  Returns N sampled processes at each float in the array *times*.

- `sample_path(t, N = 1)`

  Returns N sampled processes from *startTime* to float t, and returns the jump times (if a step process) or an equal partition of *startTime* to t.

- `sample_position(t, N = 1)`

  Returns the position at time t of N processes.

- `mean(t)`

  Returns the expected value of the process at time t.

- `var(t)`

  Returns the variance of the process at time t.

- `plot(t, N=1)`

  plot N sample paths along points in t (if array), or from *startTime* to time t (if float).

· `expected_value(f, t)`

  calculate the expected value of the process $f(X_t, t)$, where f is a Python function.

· `sample_jumps(t)`

  (Step processes only.) Returns N samples of jump times.

· `transition_pdf(x, t, y)`

  (Diffusion and Jump-Diffusion processes only.) Returns the value of the transition density function of being at y, after time t, given you start at x.

More details are available in the official documentation.

3 Exact sampling algorithms

PyProcess takes pride in being able to efficiently sample the processes *exactly*, that is sample the process with no bias. This requires sampling from some very exotic distributions. Included in *PyProcess* are non-standard distributions like the Incomplete Gamma distribution and Inverse Gaussian distribution. These distributions behave similarly to the distributions found in Scipy.Stats, and expose an $rvs()$ method to sample from the distributions.

Many algorithms come from articles written on exact sampling from a particular family of processes, see [2–4, 6–8]. One very interesting simulation algorithm is the *Exact Algorithm*.

3.1 The Exact Algorithm

Presented in [3] as recently as 2006, the *Exact Algorithm* is a new technique to sample from an arbitrary diffusion model that satisfy somewhat mild conditions. The idea is to use retrospective sampling, i.e. sample a skeleton first, and then fill in the gaps later. Retrospective sampling in most powerful in a infinite dimensional setting, where its main competitors are approximate and computationally expensive. In contrast, retrospective methods are often computationally inexpensive and exact. We will outline the main algorithm here, and note that *PyProcess* uses it to sample from the nonlinear defined:

$$dX_t = \psi \sin(X_t + \theta)\, dt + dW_t$$

In general, consider sampling from the following diffusion, defined:

$$dX_t = \alpha(X_t)dt + dW_t$$

This is not a restrictive form, as any diffusion model $dX_t = \mu(X_t)dt + \sigma(X_t)dW_t$ can be transformed to look like this by the transformation

$$\nu(x) = \int_z^x \frac{1}{\sigma(u)}\, du$$

Assume the following conditions hold:

1. the drift function α is differentiable.
2. the function $\exp\left(A(u) - \frac{(u-x)^2}{2T}\right), u \in \mathbf{R}$, for $A(u) = \int_0^u \alpha(y)\, dy$, is integrable.
3. the function $\phi = (\alpha^2 + \alpha')/2$ is bounded above and below. (Though this can be relaxed slightly)

The algorithm relies on the following theorem, proof available in [3]:

Theorem 3.1. *Let ω be any element of $C([0,T], \mathbf{R})$, and $M(\omega)$ an upper bound for the mapping ϕ, t in$[0,T]$. If Φ is a homogenous Poisson process of unit intensity on $[0,T] \times [0, M(\omega)]$, and $N =$ the number of points of Φ found below the graph $(t, \phi(B_t)|t \in [0,T]$, then:*

$$P(N = 0|\omega) = \exp\left(-\int_0^T \phi(B_t)\, dt\right)$$

The combination of the above theorem, we can sample the a skeleton of the desired process X_t as follows:

---

while True **do**
  Produce a realization $\{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}$ of the process Φ on $[0,T] \times [0, M]$.
  Simulate a skeleton of Brownian motion at times $\{\tau_1, \tau_2, \ldots, \tau_n\}$
  Evaluate N
  if $N = 0$ **then**
    Output the skeleton as distributed according to X_t.
  end if
end while

As we are sampling from the Weiner measure, we can connect the sampled skeleton using Brownian bridges and the resulting process is still sampled according to X_t. The entire result is actually just a generalization of the Acceptation-Rejection algorithm to diffusion measures, as shown in [3], where the Radon-Nikodym derivative is supplied by Girsanov's Theorem.

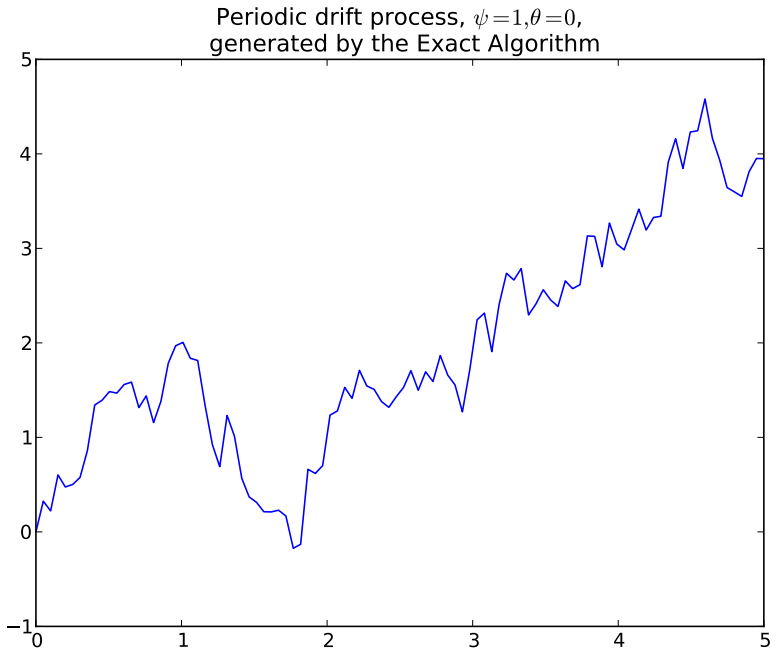

Figure 1: Example output of the Exact Algorithm applied to the periodic drift process 3.1.

4 Conditional sampling and conditional expected value

4.1 Conditional sampling

Sometimes the distribution of $X_t|X_T = y$ is not known, or is too slow or difficult to implement. An alternative approach is to sample a *time reversed* process from $Y_t = X_{T-t},\ Y_0 = y$, and a standard process x_t and determine their intersection point (if any). The time reversed process Y_t has the same dynamics as the original process [5]. See figure 4.1 for an example.

There is a bias in this method though. As we only compute a finite number of skeleton points, we are cannot be absolutely sure if the process *did not* intersect. And if they did intersect, we are never absolutely sure where they crossed.

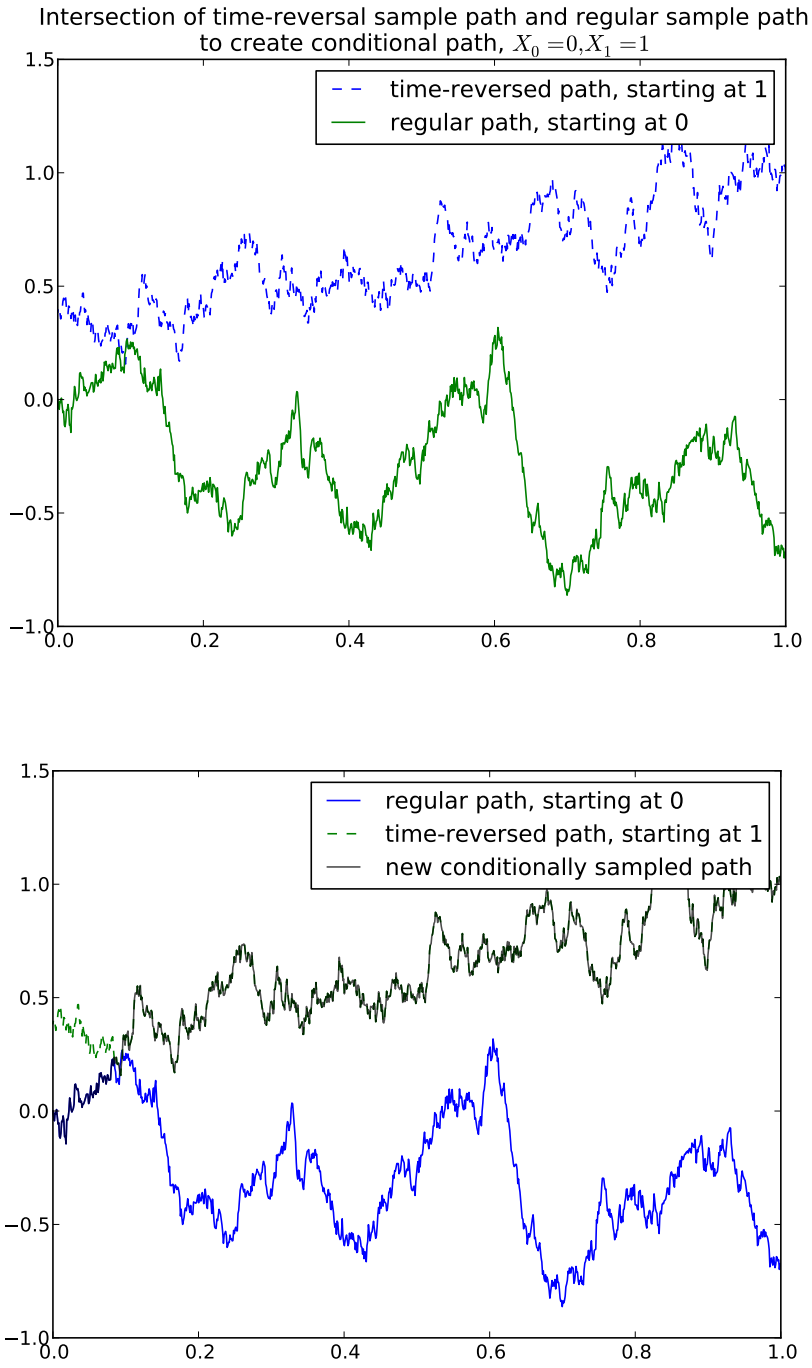

Figure 2: To sample a conditioned process, we sample a process starting at the end point, and reverse its trajectory, and sample a non-time-reversed process starting at the original start point. We join the two processes at their intersection (if any), and return the new joined process as the sampled conditional process. The process created here was Brownian motion pinned at $X_1 = 1$.

4.2 Conditional expected values

According to [1], the *folklore facts* of stochastic processes are results concerning conditional, or *pinned*, processes. Consider a time-homogeneous diffusion process $(X_t, \mathbb{P}^X)$ with transition probability function $p(x, t, y)$. One can define the pinned, or conditional, process by $X_t | X_T = y$, and the corresponding transition density function is given by:

$$H_{T,y}(s, z; t, w) = \frac{p(z, t-s, w)p(w, T-t, y)}{p(z, T-s, y)}$$

for all $0 \leq s < t < T$.

One can show that the processes defined by:

$$N_t = \frac{p(X_t, T-t, y)}{p(x, T, y)}$$

is a non-negative martingale, with $E[N_t] = N_0 = 1$:

Proof.

$$\begin{aligned} E[N_t | X_s = x_s] &= \int_\Omega \frac{p(v, T-t, y)}{p(x, T, y)} p(x_s, t-s, v)\, dv \\ &= \frac{1}{p(x, T, y} \int_\Omega p(v, T-t, y) p(x_s, t-s, v)\, dv \\ &= \frac{p(x_s, T-s, y)}{p(x, T, y)} \text{ by the Law of Total Probability} \\ &= N_s \end{aligned}$$

□

Therefore, define a probability $\mathbb{P}_T^{x,y}$ by

$$\left.\frac{d\mathbb{P}_T^{x,y}}{d\mathbb{P}^x}\right|_{X_t} = \frac{p(X_t, T-t, y)}{p(x, T, y)}$$

for all $t < T$, called the conditional probability of X_t s.t. $X_0 = x$ and $X_T = y$. An immediate application is if we wish to compute an expected value of a conditional process, we can use

$$\begin{aligned} E^{\mathbb{P}_T^{x,y}}[f(X_t)] =& E^{\mathbb{P}^x}[f(X_t) \frac{d\mathbb{P}_T^{x,y}}{d\mathbb{P}^x}] \\ &= E^{\mathbb{P}^x}[f(X_t) \frac{p(X_t, T-t, y)}{p(x, T, y)}] \end{aligned}$$

Hence we do not need to sample from the conditional distribution at all! This is simply a change of measure, or in Monte Carlo terminology, this is similar to importance sampling For this useful reason, each diffusion process in *PyProcess* has a transition density function method that can be used to perform this trick.

5 Conclusion

We explored some of the architecture and tricks used in the new *PyProcess* library. The previous release of *PyProcess* had API issues, speed issues and Python integration issues. With the extensions and improvements put into the project, we are confident that users of the library will find it very simple to use.

There are still many known processes to implement, and we would like to give the user the opportunity to create their own process based on the fundamental models presented here. On the list of process to implement are multivariate processes that have a specified correlation between the individual processes (or the underlying Brownian Motion in the case of diffusion processes). Also, we would like to extend the use of the Exact Algorithm to a more general framework. In [3], the author applies the Exact Algorithm to the diffusion analog to the logistic growth model. We would like to include this process, and others that rely on the Exact Algorithm.

The code is not included with this report, as it spans over 1700 lines of code, but it is available to be viewed on GitHub. The purpose of putting it in GitHub is to invite other developers to easily contribute to the project, and for researchers to include their own specialized processes.

References

[1] Qian, Zhongmin, and Weian Zheng. "A representation formula for transition probability densities of diffusions and applications." Stochastic processes and their applications 111.1 (2004): 57-76.

[2] Makarov, Roman N., and Devin Glew. "Exact simulation of Bessel diffusions." Monte Carlo Methods and Applications 16.3-4 (2010): 283-306.

[3] Beskos, Alexandros, Omiros Papaspiliopoulos, and Gareth O. Roberts. "Retrospective exact simulation of diffusion sample paths with applications." Bernoulli 12.6 (2006): 1077-1098.

[4] Avramidis, Athanassios N., Pierre L'ecuyer, and P-A. Tremblay. "Efficient simulation of gamma and variance-gamma processes." Simulation Conference, 2003. Proceedings of the 2003 Winter. Vol. 1. IEEE, 2003.

[5] Bladt, Mogens, and Michael Srensen. "Simple simulation of diffusion bridges with application to likelihood inference for diffusions." CREATES Research Papers (2010).

[6] Gillespie, Daniel T. "Exact numerical simulation of the Ornstein-Uhlenbeck process and its integral." Physical review E 54.2 (1996): 2084.

[7] Madan, Dilip B., Peter P. Carr, and Eric C. Chang. "The variance gamma process and option pricing." European Finance Review 2.1 (1998): 79-105.

[8] Ribeiro, Claudia, and Nick Webber. "A Monte Carlo method for the normal inverse Gaussian option valuation model using an inverse Gaussian bridge." (2003).