Imperial College London
Department of Mathematics

Stochastic Differential Equation simulations and uniform peacock problem

Ran Zihuan(Alex)
Supervisor: Professor Damiano Brigo

July 2018

Abstract

In this report we will discuss stochastic differential equations and simulations of their roots. After that, a special SDE equation will be introduced and simulated using both Euler and Milstein method.

Acknowledgements

I would like to thank my supervisor professor Damiano Brigo for giving his valuable time and effort in helping me complete this project. I really appreciate the support and guidance he has given and it has been a pleasure working with him.

Contents

Chapter 1

Overview

Stochastic Differential Equations (SDE) have long been a hot topic that make progress with the help of computing technology. In this UROP project I am going to explore this combination by implement simulations of a series of interesting SDE. This simulation will be written in python and plots will be shown to reveal the distribution of solutions.

There are many applications of Stochastic differential equations, especially in mathematical finance. Such applications, as mentioned above, are closely related to the use of computational languages and tools. In the first part I will solve SDEs using Euler's method in python, producing and exploring the plots and their meanings. Finally I will investigate a problem with a series of special but interesting SDE that produce a uniform distribution.

Chapter 2

Simulations using Euler's method

A simple model

Consider the following scenario where we have a simple model for population growth:

$$\frac{dX(t)}{dt} = 0.1X(t) + \sigma dW_t, \tag{2.1}$$

$$X(0) = 2 \tag{2.2}$$

This SDE illustrates a model with a constant change of population size at 10% at that very same moment.

Aiming at solving it for up to 10 years, we are going to use Euler-Maruyama method(1).

Euler-Maruyama method

In *Itô* calculus, the Euler-Maruyama method (also called the Euler method) is a method for the approximate numerical solution of a stochastic differential equation (SDE). It is a simple generalization of the Euler method for ordinary differential equations to stochastic differential equations. It is named after Leonhard Euler and Gisiro Maruyama. Unfortunately, the same generalization cannot be done for any arbitrary deterministic method [1].

Consider the autonomous *Itō* stochastic differential equation:

$$dX_t = a(X_t) + b(X_t)dW_t, \tag{2.3}$$

with initial condition $X_0 = x_0$. Here W_t stands for the Wiener process on time interval [0, T].

Then the Milstein method follows the procedures below to approximate the numerical value of solution:

1. partition the interval [0, T] into N equal subintervals of width $\Delta t > 0\Delta t > 0$:

$$0 = \tau_0 < \tau_1 < \cdots < \tau_N = T \text{ with } \tau_n := n\Delta t \text{ and } \Delta t = \frac{T}{N}$$

2. set $Y_0 = x_0$;

3. recursively define Y_n for $1 \leq n \leq N$ by

$$Y_{n+1} = Y_n + a(Y_n)\Delta t + b(Y_n)\Delta W_n,$$

where $\Delta W_n = W_{\tau_{n+1}} - W_{\tau_n}$ are independent and identically distributed normal random variables with expected value zero and variance Δt.(4)

Simulations

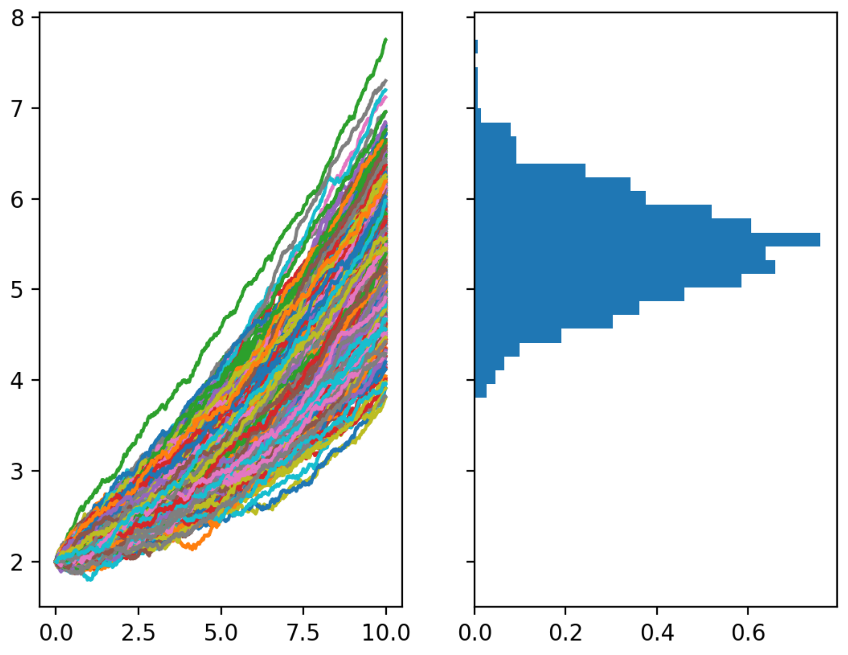

Figure 2.1: plot for sde 2.1 with 1000 trajectories

Above is a plot with 1000 trajectories and a histogram. The trajectories are numerically approximated solutions given by Euler's method with a step size of 0.01. The histogram, on the other hand, is indicating the distribution of resulting values after the period of 10 years.

From the plot we can see that as we have set, all trajectories starts The histogram on the right illustrates that the resulting distribution after 10 years indeed follows a normal distribution centering y = 2

Variations

On the basis of the above simulation, I further changed some parameters to see different results.

Variation 1

First is to increase the number of path to 10000 for a better illustration in the histogram for resulting distribution. In order to improve the performance and reduce running time, results

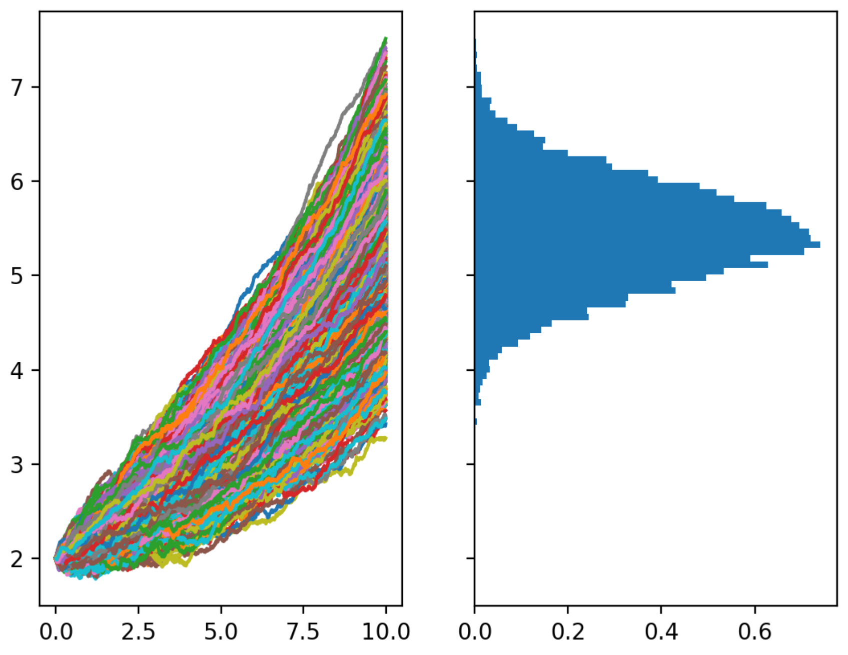

Figure 2.2: plot for sde 2.1 with 10000 trajectories

are stored in vectors.

Variation 2.1

Next I move to the case that is martingale:

$$a(x) = 0 \tag{2.4}$$

$$b(x) = 0.4x \tag{2.5}$$

$$x(0) = 100 \tag{2.6}$$

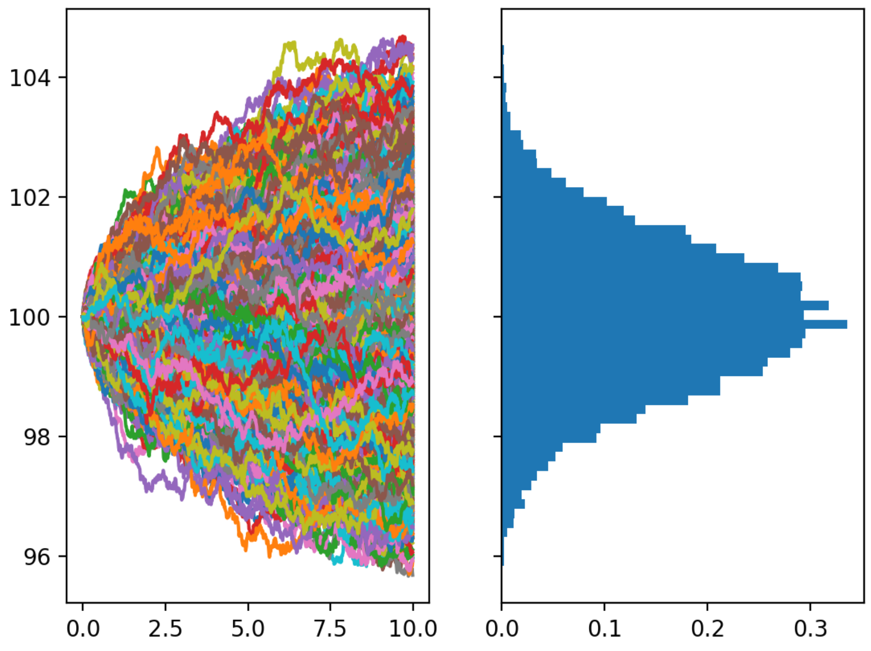

Figure 2.3: plot with initial conditions 2.3,2.4,2.5 and 10000 trajectories

Variation 2.2

I also simulated the case where:

$$a(x) = -0.05x \tag{2.7}$$

$$b(x) = 0.4x \tag{2.8}$$

$$x(0) = 100 \tag{2.9}$$

Listing 2.1: python Code for numerically solving SDE 2.1

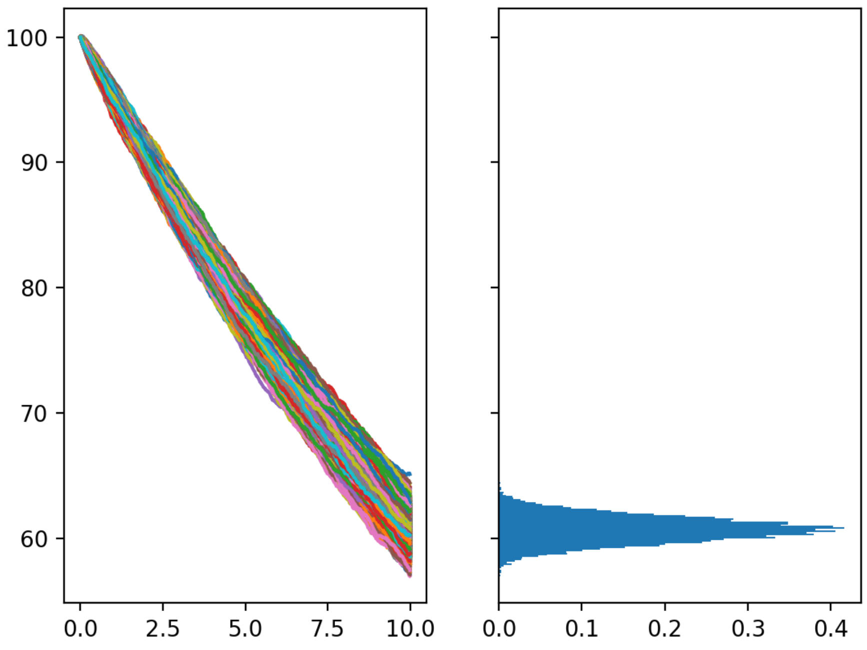

Figure 2.4: plot with and 10000 trajectories

```python
import numpy as np
import matplotlib.pyplot as plt
import time
%matplotlib notebook
start_time = time.time()


### how many trajectories
num_sims = 1000

###info from the equation
t_init = 0
t_end  = 10
N      = 1000 ### how many points in one trajectory
dt     = float(t_end - t_init) / N
x_init = 2
sigma = 0.1

a_x = 0.1
b_x = sigma

###functions describing C_a
def a(x, t):
    """update of a(X_ti).""" ## a = 0.1X_t
    return a_x*x

```

```
def b(x, t):
    """Update of b(X_ti).""" ## b = \sigma*X_t
    return b_x


def dW(delta_t):
    """Sample a random number at each call."""
    return np.random.normal(loc = 0.0, scale = np.sqrt(delta_t))

###initialze
ts     = np.arange(t_init, t_end, dt)
xs     = np.zeros((N,num_sims))
xs[0]  = x_init*np.ones(num_sims)

###set up subplot
f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=↩
    True, subplot_kw=None, gridspec_kw=None)

###Euler's Method
for j in range(0,num_sims):
    for i in range(1, ts.size):
        t = (i-1) * dt
        x = xs[i-1,j]
        xs[i,j] = x + a(x, t) * dt + b(x, t) * dW(dt)
    ##trajectories
    ax1.plot(ts, xs[:,j])

##histogram
ax2.hist(xs[-1], density = True, bins ='auto',orientation='↩
    horizontal')

plt.show()

##timer
print("--- %s seconds ---" % (time.time() - start_time))
```

Chapter 3

SDEs with uniform distributions

SDE with uniform distribution and Conic Supports

In this section I will consider the following SDE, which will generate a uniform distribution:(2)

$$dX_t = \mathbb{1}_{X_t \in [-b(t), b(t)]} \left(\frac{\dot{b}(t)}{b(t)} (b(t)^2 - X_t^2) \right)^{1/2} dW_t, X_0 = 0 \tag{3.1}$$

3.0.1 Euler - Maruyama method

Aiming to approximate solutions to SDE (3.2) numerically, Euler's method, as introduced in chapter 2, is continued to be used in this uniform peacock problem.

Linear case

Consider the linear case where $b(t) = kt$, then equation 3.1 becomes:

$$dX_t = \mathbb{1}_{X_t \in [-kt, kt]} \frac{1}{\sqrt{t}} \sqrt{(kt)^2 - X_t^2} dW_t, X_0 = 0 \tag{3.2}$$

By implementing Euler's method to $b(t) = t$ namely when$k = 1$, we will have the following plot and histogram.

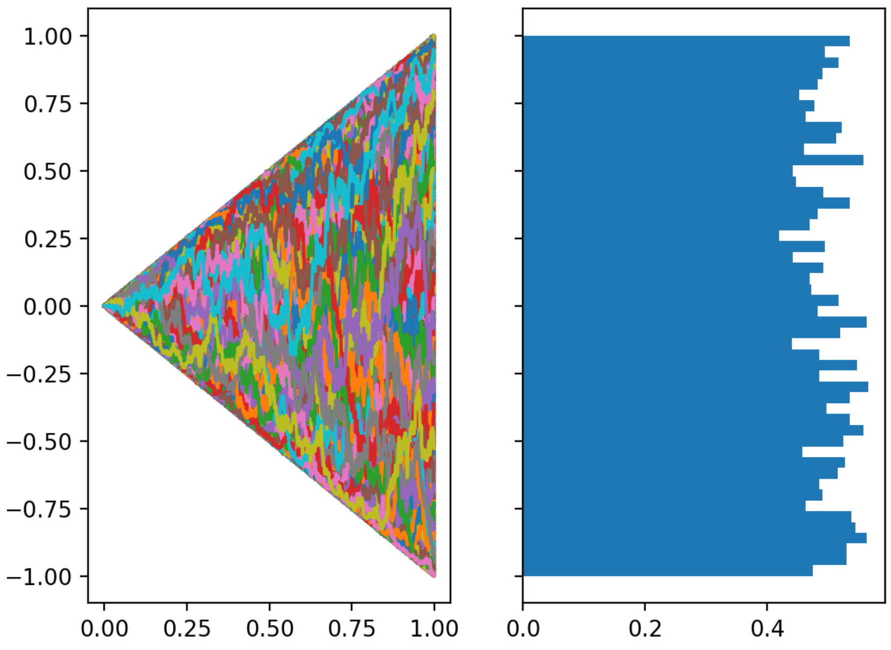

Figure 3.1: plot of simulation using Euler's method and 10000 trajectories

Listing 3.1: Euler-Maruyama Method

```
import numpy as np
import matplotlib.pyplot as plt
import time
import math
%matplotlib notebook
start_time = time.time()


### how many trajectories
num_sims = 10000

###info from the equation
t_init = 0
t_end  = 1
N      = 1000 ### how many points in one trajectory
dt     = float(t_end - t_init) / N
x_init = 0
#sigma = 0.1

a_x = 0


###functions describing C_a
def a(x, t):
```

```python
    """update of a(X_ti).""" ## a = 0.1X_t
    return a_x*x

def b(x, t):
    """Update of b(X_ti).""" ## b = \sigma*X_t
    b_t = t
    if abs(b_t)>=x>=-abs(b_t):
        indicator = 1
        b_x = indicator*(t**2-x**2)**0.5/math.sqrt(t)
    else:
        indicator  = 0
        b_x = indicator
    #(0.5*t**(-0.5)*(b_t**2-x**2)/b_t)**0.5
    return b_x

def dW(delta_t):
    """Sample a random number at each call."""
    return np.random.normal(loc = 0.0, scale = np.sqrt(delta_t))

###initialze
ts    = np.arange(t_init, t_end, dt)
xs    = np.zeros((N,num_sims))
xs[0] = x_init*np.ones(num_sims)

###set up subplot
f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=True, subplot_kw=None, gridspec_kw=None)

###Euler's Method
for j in range(0,num_sims):
    for i in range(1, ts.size):
        t = (i) * dt
        x = xs[i-1,j]
        xs[i,j] = x + b(x, t) * dW(dt)
    ##trajectories
    ax1.plot(ts, xs[:,j])

##histogram
ax2.hist(xs[-1], density = True, bins = 50, orientation='horizontal')

plt.show()

##timer
print("--- %s seconds ---" % (time.time() - start_time))
```

square-root case

Again consider SDE 3.1, if we take instead $b(t = \sqrt{t})$, then itcan be written as:

$$dX_t = \frac{1}{\sqrt{2}}\sqrt{1 - \frac{X_t^2}{t}}\mathbb{1}_{X_t \in [-\sqrt{t}, \sqrt{t}]} dW_t, X_0 = 0 \tag{3.3}$$

Then, using Euler's method, the following ploto with 10000 trajectories and 50 bins are produced:

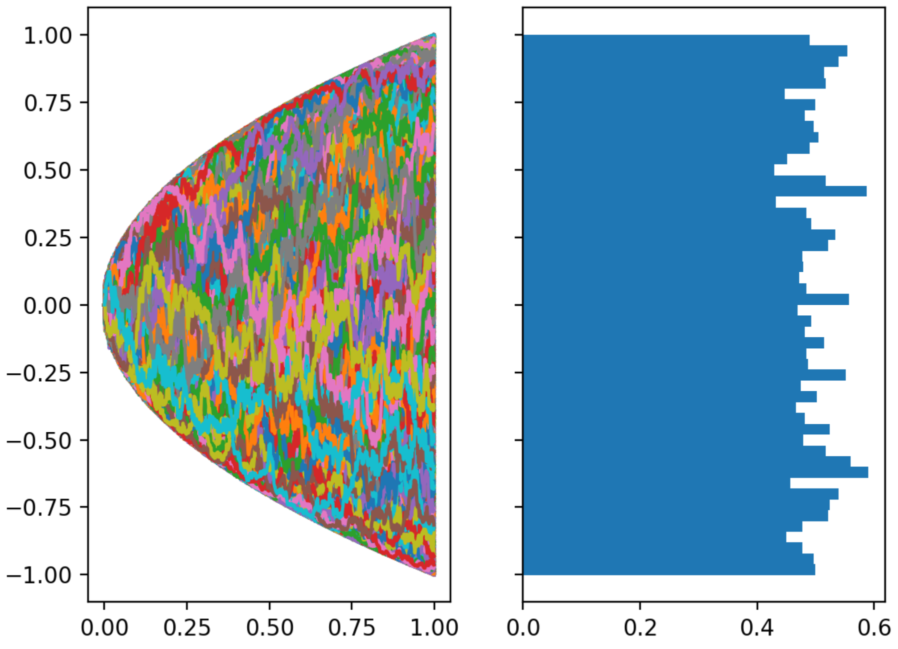

Figure 3.2: plot of root-square case using Euler method and 10000 trajectories

Listing 3.2: square-root case with 10000 trajectories and 50 bins

```
import numpy as np
import matplotlib.pyplot as plt
import time
import math
%matplotlib notebook
start_time = time.time()


### how many trajectories
num_sims = 10000

###info from the equation
```

```python
t_init = 0
t_end  = 1
N      = 1000 ### how many points in one trajectory
dt     = float(t_end - t_init) / N
x_init = 0
#sigma = 0.1

a_x = 0


###functions describing equation
def a(x, t):
    """update of a(X_ti)."""
    return a_x*x

def b(x, t):
    """Update of b(X_ti)."""
    b_t = math.sqrt(t)
    if abs(b_t)>=x>=-abs(b_t):##conic support
        indicator = 1
        b_x = indicator*math.sqrt(1 - x**2/t)/math.sqrt(2)
    else:
        indicator  = 0
        b_x = indicator
    return b_x

def dW(delta_t):
    """Sample a random number at each call."""
    return np.random.normal(loc = 0.0, scale = np.sqrt(delta_t))

###initialze
ts    = np.arange(t_init, t_end, dt)
xs    = np.zeros((N,num_sims))
xs[0] = x_init*np.ones(num_sims)

###set up subplot
f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=↩
    True, subplot_kw=None, gridspec_kw=None)

###Euler's Method
for j in range(0,num_sims):
    for i in range(1, ts.size):
        t = (i) * dt
        x = xs[i-1,j]
        xs[i,j] = x + b(x, t) * dW(dt)
    ##trajectories
    ax1.plot(ts, xs[:,j])

##histogram
ax2.hist(xs[-1], density = True, bins = 50, orientation='horizontal'↩
    )
```

```

plt.show()

##timer
print("--- %s seconds ---" % (time.time() - start_time))
```

3.0.2 Milstein method

It is named after Grigori N. Milstein who first published the method in 1974.(3) Consider the autonomous *Itō* stochastic differential equation:

$$dX_t = a(X_t) + b(X_t)dW_t, \tag{3.4}$$

with initial condition $X_0 = x_0$. Here W_t stands for the Wiener process on time interval [0, T].

Then the Milstein method follows the procedures below to approximate the numerical value of solution:

1. partition the interval [0, T] into N equal subintervals of width $\Delta t > 0 \Delta t > 0$:

$$0 = \tau_0 < \tau_1 < \cdots < \tau_N = T \text{ with } \tau_n := n\Delta t \text{ and } \Delta t = \frac{T}{N}$$

2. set $Y_0 = x_0$;

3. recursively define Y_n for $1 \leq n \leq N$ by

$$Y_{n+1} = Y_n + a(Y_n)\Delta t + b(Y_n)\Delta W_n + \frac{1}{2}b(Y_n)b'(Y_n)\left((\Delta W_n)^2 - \Delta t\right),$$

where b' denotes the derivative of b(x) with respect to x and $\Delta W_n = W_{\tau_{n+1}} - W_{\tau_n}$ are independent and identically distributed normal random variables with expected value zero and variance Δt.(4)

linear case

In linear case where $b(t) = kt, k = 1$:

$$dX_t = \mathbb{1}_{X_t \in [-kt,kt]} \frac{1}{\sqrt{t}} \sqrt{(kt)^2 - X_t^2 dW_t}, X_0 = 0 \tag{3.5}$$

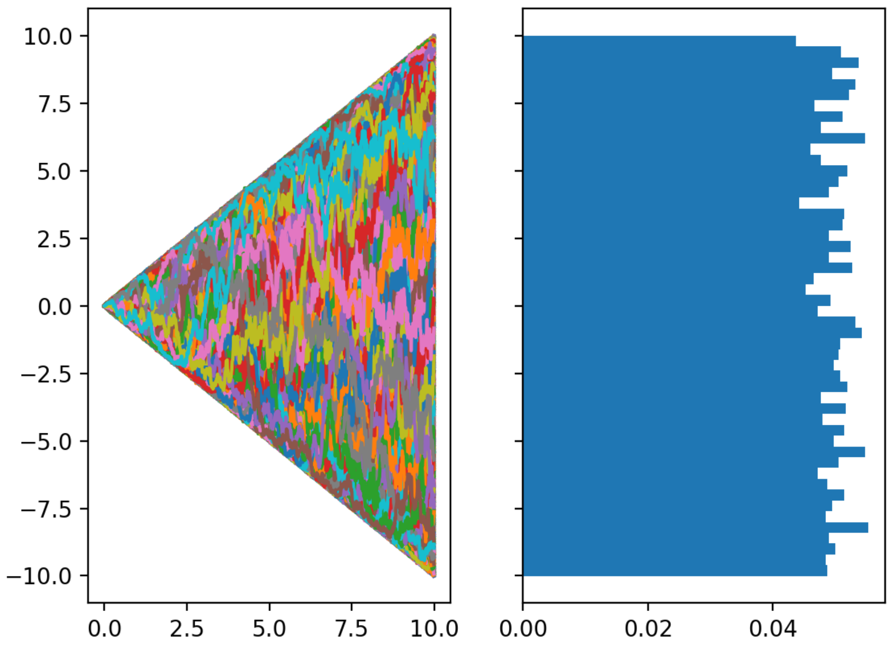

Figure 3.3: plot of simulation using Milstein method and 10000 trajectories

code

Listing 3.3: Milstein Method

```
import numpy as np
import matplotlib.pyplot as plt
import time
%matplotlib notebook
start_time = time.time()


### how many trajectories
num_sims = 10000

###info from the equation
t_init = 0
t_end  = 10
N      = 1000 ### how many points in one trajectory
dt     = float(t_end - t_init) / N
x_init = 0

a_x = 0

###functions describing C_a
def a(x, t):
    """update of a(X_ti).""" ## a = 0.1X_t
    return a_x*x

def b(x, t):
    """Update of b(X_ti).""" ## b = \sigma*X_t
    b_t = t
    if abs(b_t)>=x>=-abs(b_t):
        indicator = 1
        b_x = indicator*(t**2-x**2)**0.5/math.sqrt(t)
    else:
        indicator  = 0
        b_x = indicator
    #(0.5*t**(-0.5)*(b_t**2-x**2)/b_t)**0.5
    return b_x

def b_1(x, t):
    """Update of b(X_ti).""" ## b = \sigma*X_t
    b_t = t
    if abs(b_t)>=x>=-abs(b_t):
        indicator = 1
        b_x = indicator*0.5*math.sqrt(t/(t**2-x**2))*(1+x**2/t**2)
    else:
        indicator  = 0
        b_x = indicator
    #(0.5*t**(-0.5)*(b_t**2-x**2)/b_t)**0.5
    return b_x

def dW(delta_t):
    """Sample a random number at each call."""
    return np.random.normal(loc = 0.0, scale = np.sqrt(delta_t))

###initialze
ts    = np.arange(t_init, t_end, dt)
xs    = np.zeros((N,num_sims))
xs[0] = x_init*np.ones(num_sims)

###set up subplot
f, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharex=False, sharey=↩
    True, subplot_kw=None, gridspec_kw=None)

###Milstein's Method
for j in range(0,num_sims):
    for i in range(1, ts.size):
        t = i * dt
        x = xs[i-1,j]
        xs[i,j] = x + a(x, t) * dt + b(x, t) * dW(dt) + 0.5*b(x,t)*↩
            b_1(x,t)*(dW(dt)**2-dt)
    ##trajectories
    ax1.plot(ts, xs[:,j])

##histogram
```

```
ax2.hist(xs[-1], density = True, bins = 50,orientation='horizontal')

plt.show()

##timer
print("--- %s seconds ---" % (time.time() - start_time))
```

For the square-root case, Euler's method is preferred because of the risk of running into singularity when differentiating $b(t)$.

Bibliography

[1] Kloeden, P.E. & Platen, E. (1992). *Numerical Solution of Stochastic Differential Equations.* Springer, Berlin. ISBN 3-540-54062-8.

[2] Damiano Brigo, Monique Jeanblanc, Frederic Vrins. *SDEs with uniform distributions: Peacocks, Conic martingales and mean reverting uniform diffusions.* arXiv:1606.01570

[3] Mil'shtein, G. N. (1974). *Approximate integration of stochastic differential equations.* Teor. Veroyatnost. i Primenen (in Russian). 19 (3): 583588.

[4] Milshtein, G. N. (1975). *Approximate Integration of Stochastic Differential Equations.* Theory of Probability & Its Applications. 19 (3): 557000. doi:10.1137/1119062.