Autumn 2022 STAT 31120 Solutions

Hongli Zhao

November 2022

Abstract

STAT 31120 / CAAM 31120 at the University of Chicago is a graduate-level course on Numerical Methods for Stochastic Differential Equations. The course covers fundamental concepts concerning strong and weak convergence of random variables, stochastic Taylor expansions (Taylor-Ito expansion), difference between Ito and Stratonovich integrals, and explicit numerical methods for stochastic integration. I had the fortune to work closely with Prof. Zhongjian Wang as the course reader for Winter 2021 and Autumn 2022 offerings of the course. The course webpage is updated here. Textbook used was Kloeden & Platen, *Numerical Solution of Stochastic Differential Equations.*

This document details updated solutions to course projects (5 in total) along with a small portion of the final. Code is available on the associated Github page, in Python and Julia. Any comments and suggestions are welcome, please send an email to `honglizhaobob@uchicago.edu`.

Disclaimer: If you are viewing this document (or Github page) as an active UChicago student, please use this document at your own risk, add appropriate references, and adhere to UChicago's academic integrity policy. Please email me or your course instructor if you are unsure.

1 Project 1

1.1 Uniform Random Number Generation

Generate $N = 10^4$ uniformly distributed pseudo random numbers on $(0, 1)$. Partition the interval into subintervals I_j of equal length 0.05. Count the number of random numbers in I_j as N_j. Plot the relative frequencies:

$$p_j = \frac{N_j}{N \cdot I_j}$$

to create a histogram. Compare this histogram with the density of $U(0, 1)$.

In addition, compute sample average and sample variance and compare them to the exact expectation and variance 1/2 and 1/12.

Solution: The Julia code for creating histogram and computing moments of $\mathcal{U}[0, 1]$ is as follows.

30

Uniform Random Number Generation

December 27, 2022

[1]:
```julia
# libraries
using Random
Random.seed!(3); # for reproducibility
using Plots
pyplot();
```

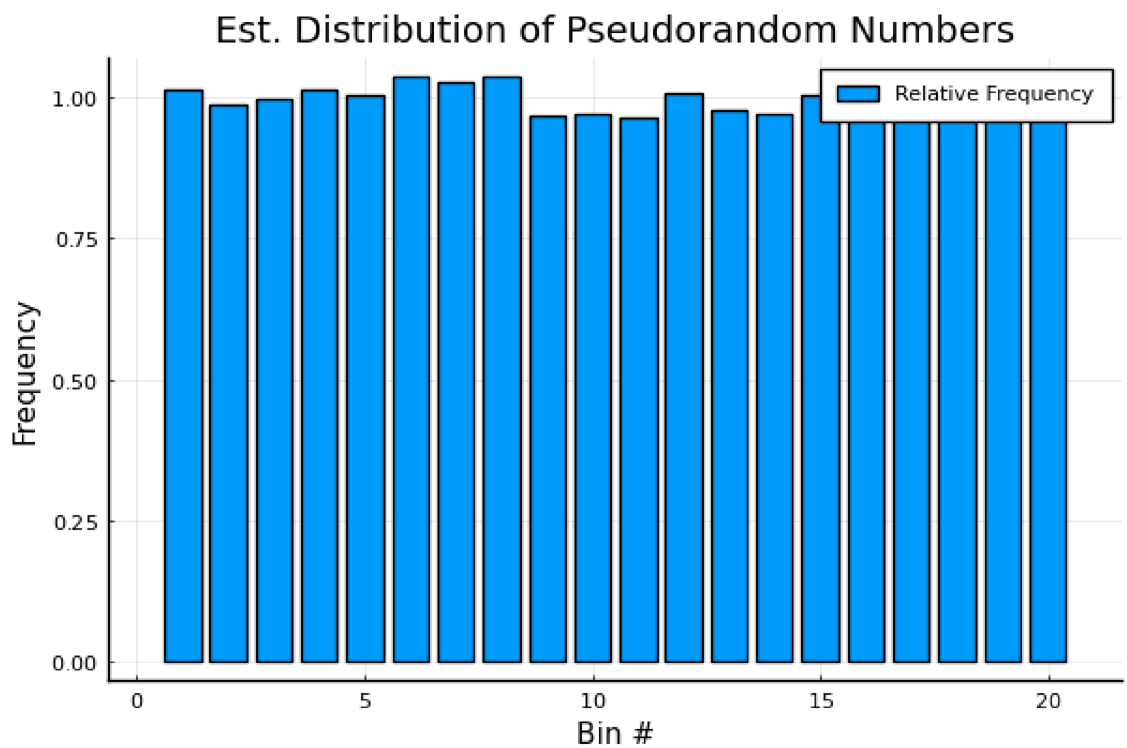
[2]:
```julia
# generate random numbers from uniform [0,1]
unif_numbers = rand(10^4);

"""
Helper function, finds the index of the sub-interval
x falls in.
"""
function find_interval(intervals, x)
    i = searchsortedlast(intervals, x)
    i == length(intervals) && (i = 0)
    return(i)
end

"""
Partitions the interval [lower, upper] based on
nbins, and count the relative frequencies of rand_nums
in each bin.
"""
function count_frequency(rand_nums, lower=0, upper=1, nbins=20)
    N = length(unif_numbers)
    interval_length = (upper - lower) / nbins
    all_bins = collect(lower:interval_length:upper)
    all_counts = Vector{Float64}(undef, nbins)
    # find which bin each number is in
    all_bin_nums = zeros(0)
    for x in rand_nums
        append!(all_bin_nums, find_interval(all_bins, x))
    end
    all_bin_nums = Vector{Int64}(all_bin_nums)
    # relative frequency
```

```
    all_counts = Dict{Int64, Float64}([(i, count(x->x==i, all_bin_nums)) for i in all_bin_nums])
    all_counts = sort(collect(all_counts), by = x->x[1])
    all_counts = Dict{Int64, Float64}([(x[1], x[2]) for x in all_counts])
    all_freq = ( collect(values(all_counts)) / N ) / interval_length
    return(all_freq)
end
```

[2]:
```
count_frequency
```

[5]:
```
frequencies = count_frequency(unif_numbers);
bar(frequencies, label="Relative Frequency")
title!("Est. Distribution of Pseudorandom Numbers", xlabel="Bin #", ylabel="Frequency")
```

[5]:



[4]:
```
est_mean = sum(unif_numbers) / length(unif_numbers);
est_variance = sum((unif_numbers .- est_mean).^2) / ( length(unif_numbers) - 1);
println("*> Est. Mean = ", est_mean)
println("*> Est. Var  = ", est_variance)
```

```
*> Est. Mean = 0.49997059832859697
*> Est. Var  = 0.0836583264092236
```

1.2 Custom Density Random Number Generation

Repeat the previous question for the following density supported on $[0, 2]$.

$$p(x) = \frac{1}{4}x^3$$

Solution: We first verify the theoretical mean and variance for p. Let $X \sim p$

$$\mathbb{E}\big[X\big] = \int_0^2 xp(x)dx = \frac{1}{4}\int_0^2 x^4 dx = \frac{8}{5}$$

similarly,

$$\mathbb{E}\big[X^2\big] = \frac{8}{3}, \mathrm{Var}[X] = \mathbb{E}\big[X^2\big] - \mathbb{E}\big[X\big]^2 = \frac{8}{75}$$

To sample from the distribution p, we generate random seeds from $\mathcal{U}[0, 1]$ and apply the inverse CDF method. The CDF is given by:

$$P(x) = \int_0^x p(z)dz = \frac{1}{16}x^4$$

then $P^{-1}(z) = 2z^{1/4}$ for $z \sim \mathcal{U}[0, 1]$. The code for generating the samples are as follows.

35

Inverse CDF

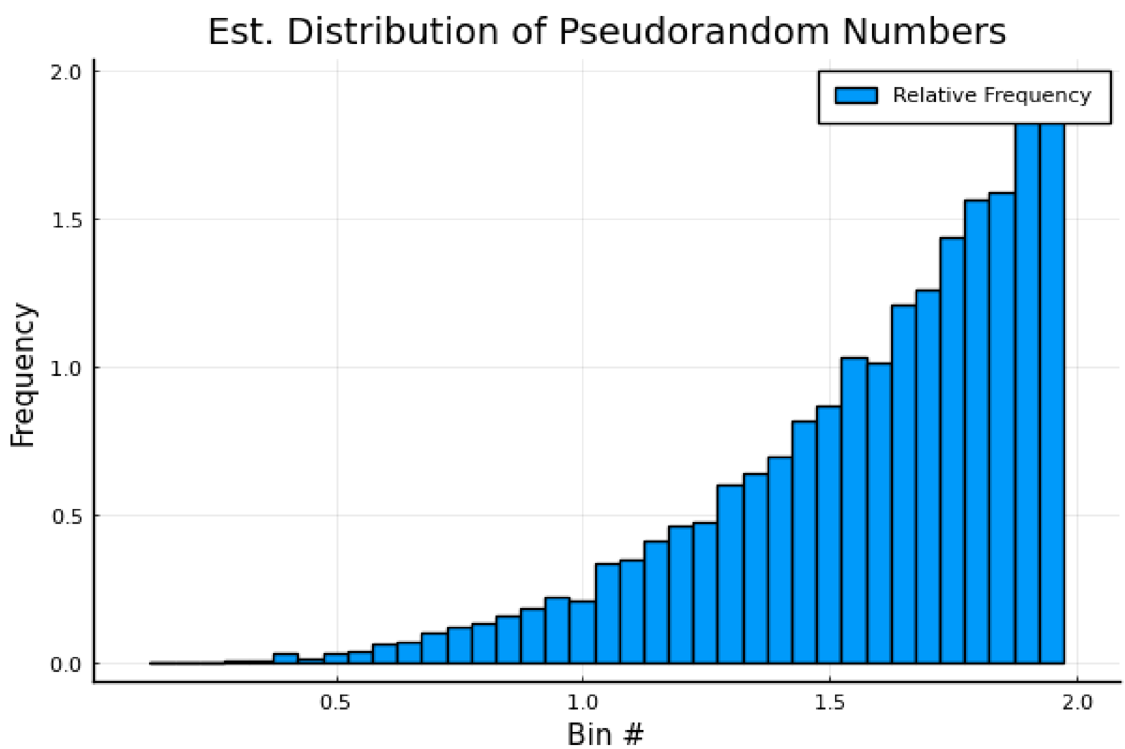
December 27, 2022

[1]:
```julia
# libraries
using Random
Random.seed!(3); # for reproducibility
using Plots
pyplot();
```

[2]:
```julia
# repeat the find interval helper functions as in part 1
"""
Helper function, finds the index of the sub-interval
x falls in.
"""
function find_interval(intervals, x)
    i = searchsortedlast(intervals, x)
    i == length(intervals) && (i = 0)
    return(i)
end

"""
Partitions the interval [lower, upper] based on
nbins, and count the relative frequencies of rand_nums
in each bin.
"""
function count_frequency(rand_nums, lower=0, upper=1, nbins=20)
    N = length(unif_numbers)
    interval_length = (upper - lower) / nbins
    all_bins = collect(lower:interval_length:upper)
    all_counts = Vector{Float64}(undef, nbins)
    # find which bin each number is in
    all_bin_nums = zeros(0)
    for x in rand_nums
        append!(all_bin_nums, find_interval(all_bins, x))
    end
    all_bin_nums = Vector{Int64}(all_bin_nums)
    # relative frequency
    all_counts = Dict{Int64, Float64}([(i, count(x->x==i, all_bin_nums)) for i in all_bin_nums])
    all_counts = sort(collect(all_counts), by = x->x[1])
```

```
    all_counts = Dict{Int64, Float64}([(x[1], x[2]) for x in all_counts])
    all_freq = ( collect(values(all_counts)) / N ) / interval_length
    return(all_freq)
end
        36
# generate random seed
unif_numbers = rand(10^4);
# apply inverse transform
x = 2*(unif_numbers.^(1/4));
# plot histogram
histogram(x, bins = range(minimum(x)-0.05,maximum(x), step = 5 / 100),
    normalize = true, label = "Relative Frequency")
title!("Est. Distribution of Pseudorandom Numbers", xlabel="Bin #",␣
 ↪ylabel="Frequency")
```

[2]:



[12]:
```
est_mean = sum(x) / length(x);
est_variance = sum((x .- est_mean).^2) / ( length(x) - 1 );
println("*> Est. Mean = ", est_mean)
println("*> Est. Var  = ", est_variance)
```

```
*> Est. Mean = 1.6053308315363621
*> Est. Var  = 0.10365923540971544
```

1.3 Box-Muller Distribution

Show that two random numbers N_1, N_2, generated by the Box-Muller method, are Gaussian with zero mean and identity variance, when the seeds U_1, U_2 are independent $U(0,1)$ distributed.

Solution:

The Box-Muller method uses the following nonlinear mapping:

$$N_1 = \sqrt{-2\ln U_1} \cdot \cos(2\pi \cdot U_2)$$

$$N_2 = \sqrt{-2\ln U_1} \cdot \sin(2\pi \cdot U_2)$$

where U_1, U_2 are Unif$[0,1]$.

Consider polar coordinates defined by:

$$\begin{aligned} r = \sqrt{N_1^2 + N_2^2} &= \sqrt{-2\ln U_1 \cdot \cos^2(2\pi \cdot U_2) - 2\ln U_2 \cdot \sin(2\pi \cdot U_1)} \\ &= \sqrt{-2\ln U_1}(\cos^2(2\pi U_2) + \sin^2(2\pi U_2)) = \sqrt{-2\ln U_1} \\ \theta = \tan^{-1}(\frac{N_2}{N_1}) &= \tan^{-1}\left[\frac{\sin(2\pi \cdot U_2)}{\cos(2\pi \cdot U_2)}\right] \\ &= \tan^{-1}\tan(2\pi U_2) = 2\pi U_2 \end{aligned}$$

Here we have converted $(N_1, N_2) = (\sqrt{-2\ln U_1} \cdot \cos(2\pi \cdot U_2), \sqrt{-2\ln U_1} \cdot \sin(2\pi \cdot U_2))$ into polar coordinates (r, θ), it is enough to verify that the joint distribution of r, θ satisfies the polar form of a standard normal in $\mathbb{R}^2$.

Next, we derive the probability density for r, θ,

$$\begin{aligned} \mathbb{P}(r \le x) = \mathbb{P}(\sqrt{-2\ln U_1} \le x) &= \mathbb{P}\left[U_1 \ge \exp\left(-\frac{1}{2}x^2\right)\right] \\ &= 1 - \mathbb{P}\left[U_1 \le \exp\left(-\frac{1}{2}x^2\right)\right] \\ = 1 - F_{U_1}\left[\exp\left(-\frac{1}{2}x^2\right)\right] &= 1 - \exp\left(-\frac{1}{2}x^2\right) \end{aligned}$$

where F_{U_1} is the CDF of Unif$(0,1)$. Then we have density:

$$f_R(r) = \frac{d}{dx}\mathbb{P}(r \le x) = x\exp\left(-\frac{1}{2}x^2\right)$$

The density of θ is directly renormalized from a uniform distribution:

$$\mathbb{P}(\theta \le x) = \mathbb{P}(2\pi U_2 \le x) = \mathbb{P}(U_2 \le \frac{1}{2\pi}x) = \frac{x}{2\pi}$$

then the density:

$$f_\Theta(\theta) = \frac{d}{dx}\mathbb{P}(\theta \le x) = \frac{1}{2\pi}$$

The joint density of U_1, U_2 is a product measure / independent, this means the joint density of $r := r(U_1), \theta := \theta(U_2)$ must also be a product measure, therefore we obtain finally:

$$f_{R,\Theta}(r,\theta) = \frac{r}{2\pi}\exp\left(-\frac{1}{2}r^2\right)$$

One might recall that this is the polar form of a standard Gaussian PDF in $\mathbb{R}^2$; if not, we may use the following transformation:

$$(r,\theta) \mapsto (r\cos(\theta), r\sin(\theta))$$

and put (r,θ) back to Cartesian coordinates.

1.4 Jointly Gaussian distributions

1.4.1 Linear Transformation

Let $Z = (N_1, N_2)$ for N_1, N_2 standard normal, show that $X = S^T Z + \mu$ is jointly Gaussian with mean μ and covariance $S^T S$. Here S is an invertible matrix.

Solution: We use the change of variables formula for random vectors, here $X = g(Z) = S^T Z + \mu$, then:

$$f_{\mathbf{X}}(X) = f_{\mathbf{Z}}(g^{-1}(X)) \cdot \left|\det \frac{dX}{dZ}\right|^{-1}$$

where $\frac{dX}{dZ}$ denotes the Jacobian matrix of X with respect to Z (i.e. original density + volume correction). The Jacobian of X with respect to Z is calculated by matrix calculus:

$$\frac{dX}{dZ} = \nabla_Z f(Z) = S$$

And:

$$g^{-1}(X) = S^{-T}(X - \mu)$$

Then we have the density:

$$\begin{aligned} f_{\mathbf{X}}(X) &= \frac{1}{2\pi|\det S|}\exp\left(-\frac{1}{2}[S^{-T}(X-\mu)]^T[S^{-T}(X-\mu)]\right) \\ &= \frac{1}{2\pi|\det S^T S|^{1/2}}\exp\left(-\frac{1}{2}(X-\mu)^T(S^T S)^{-1}(X-\mu)\right) \end{aligned}$$

from which we conclude that $\mathbf{X} \sim \mathcal{N}(\mu, S^T S)$.

1.4.2 Generating Joint Gaussian Random Numbers

Write a program to generate a pair of Gaussian pseudorandom numbers X_1, X_2 with zero mean and covariance $\mathbb{E}\left[X_1^2\right] = 1$, $\mathbb{E}\left[X_2^2\right] = 1/3$, $\mathbb{E}\left[X_1 X_2\right] = 1/2$. Generate 1000 pairs of such numbers and compute sample averages and sample covariances.

Solution: This section is meant to confirm the result from the previous section, please do not use built-in samplers for this question. The use of Unif$(0, 1)$ number generation is allowed. The procedure should be roughly:

$$\text{Unif}(0,1) \to \mathcal{N}(0,1) \to \mathcal{N}(\mu, C)$$

We have the covariance matrix:

$$S = \begin{bmatrix} \mathbb{E}[X_1^2] & \mathbb{E}[X_1 X_2] \\ \mathbb{E}[X_2 X_1] & \mathbb{E}[X_2^2] \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$$

59 The code for Box-Muller is as follows.

Box-Muller

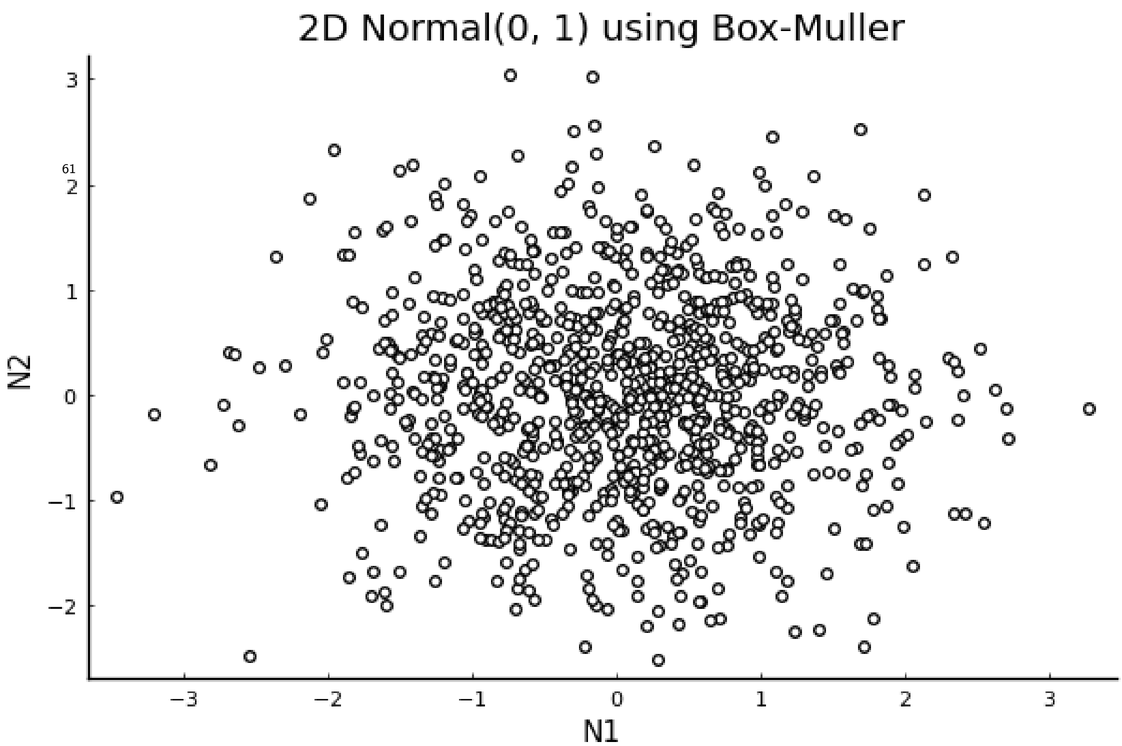
60

November 25, 2022

[1]:
```
# libraries
using Random
Random.seed!(3); # for reproducibility
using Plots
pyplot();
```

[2]:
```
"""
Helper function that generates standard normal
random variables based on uniform seed. Seed
input is assumed to have shape [N, 2].
"""
function box_muller(seed=rand(1000, 2))
    N = size(seed)[1]
    U1 = seed[:, 1]
    U2 = seed[:, 2]
    N1 = sqrt.(-2 * log.(U1)) .* cos.(2 * pi .* U2)
    N2 = sqrt.(-2 * log.(U1)) .* sin.(2 * pi .* U2)
    normal = zeros(N, 2)
    normal[:, 1] = N1
    normal[:, 2] = N2
    return(normal)
end

# check box muller is working
normal_numbers = box_muller()
layout = @layout [a            _
                  b{0.8w,0.8h} c]
default(fillcolor = :lightgrey, markercolor = :white, grid = false, legend = false)
plot(layout = layout, link = :both, size = (500, 500), margin = -10Plots.px)
plot(normal_numbers[:, 1], normal_numbers[:, 2], seriestype = :scatter,
   xlabel="N1", ylabel="N2", title="2D Normal(0, 1) using Box-Muller")
```

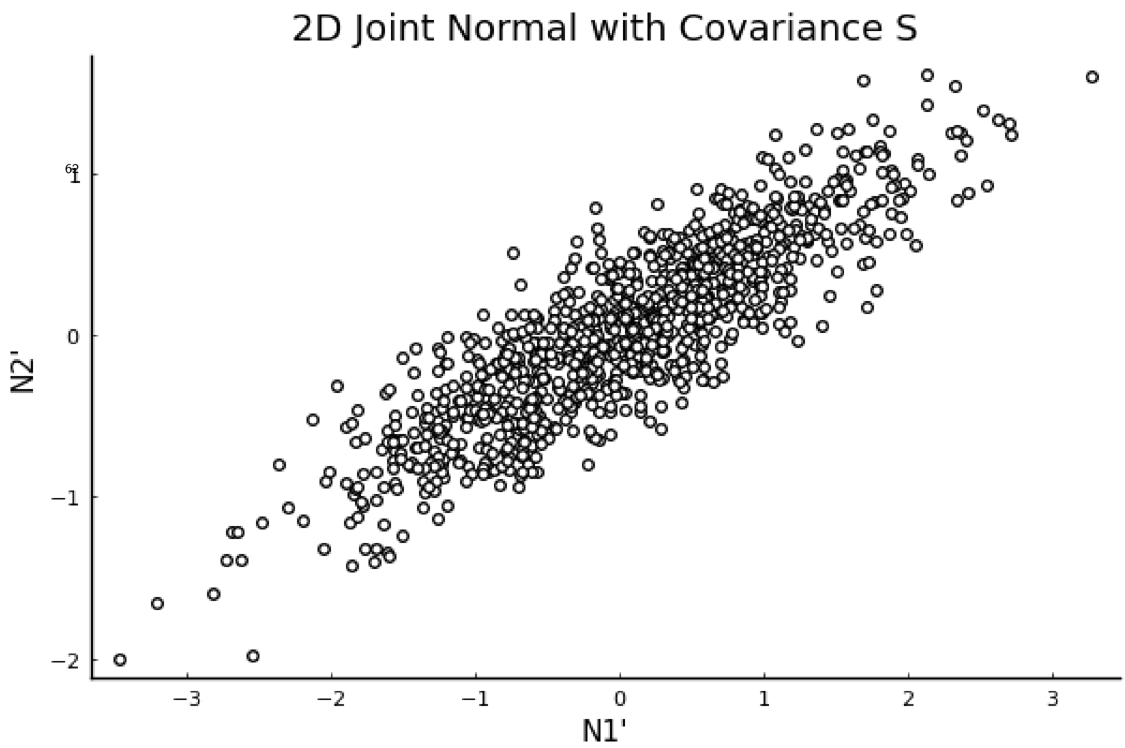
[2]:



Now we have the desired covariance matrix C, we can compute the matrix S such that $C = S^T S$ for our linear transformation. We do so by Cholesky factorization.

[6]:
```
using LinearAlgebra
C = [1 1/2; 1/2 1/3];
S = cholesky(C).U;
# transform normal(0,1)
Y = transpose(transpose(S) * transpose(normal_numbers));
# plot new histogram
plot(Y[:, 1], Y[:, 2], seriestype = :scatter,
   xlabel="N1'", ylabel="N2'", title="2D Joint Normal with Covariance S")
```

[6]:



[4]:
```
using Statistics
est_mean = mean(Y, dims = 1)
```

[4]:
```
1×2 Matrix{Float64}:
 0.0402206  0.0298096
```

[5]:
```
est_cov = cov(Y, dims = 1, corrected = true)
```

[5]:
```
2×2 Matrix{Float64}:
 0.963013  0.479742
 0.479742  0.314302
```

1.4.3 Extension

Is it possible to generate a pair of real random numbers X_1, X_2 (not necessarily Gaussian) with $\mu = 0$ and covariance structure: $\mathbb{E}\big[X_1^2\big] = 1, \mathbb{E}\big[X_2^2\big] = 1/3, \mathbb{E}\big[X_1 X_2\big] = 1$?

Solution: It is not possible to generate such a pair, to see this, it is enough to compute the correlation coefficient that arises from this desired statistical profile:

$$\rho = \frac{\mathbb{E}[X_1 X_2]}{\sqrt{\mathrm{Var}[X_1]} \cdot \sqrt{\mathrm{Var}[X_2]}} = \frac{1}{1 \cdot \sqrt{\frac{1}{3}}} = \sqrt{3} > 1$$

where in the case $\mathbb{E}[X_1] = \mathbb{E}[X_2] = 0$, $\mathrm{Var}[X_1] = \mathbb{E}[X_1^2], \mathrm{Var}[X_2] = \mathbb{E}[X_2^2]$.

2 Project 2

2.1 SDE solution

Let $X_t = \int_0^t f(s,w)dW_s$, show that e^{X_t} is a solution of the SDE:

$$dY_t = \frac{1}{2} f^2(t,w) Y_t dt + f(t,w) Y_t dW_t$$

and $\exp\Big(X_t - \frac{1}{2}\int_0^t f^2(s,w)ds\Big)$ is a solution of the SDE:

$$dY_t = f(t,w) Y_t dW_t$$

Solution: We recall the most general Ito's formula (scalar valued process); we will be using this formula throughout. Suppose X_t satisfies the integral form:

$$X_t = X_0 + \int_0^t A_s ds + \int_0^t C_s dB_s$$

and suppose $f(t,x)$ is C^1 in t, and C^2 in x, Ito's formula gives:

$$df(t,X_t) = \left[\partial_t f(t,X_t) + A_t \partial_x f(t,X_t) + \frac{1}{2} C_t^2 \partial_{xx} f(t,X_t)\right] dt + C_t \partial_x f(t,X_t) dB_t$$

where B_t is standard Brownian motion (sBM); our text uses W_t.

The first problem is a verification of Ito's formula. Here $X_t = \int_0^t f(s,\omega)dW_s$. Then:

$$dX_t = f(t,\omega)dW_t$$

Let $Z_t = \exp(X_t)$, the exponential function is smooth in x, then applying Ito's formula we obtain:

$$\begin{aligned} dZ_t &= \frac{1}{2} f^2(t,\omega)\exp(X_t)dt + f(t,\omega)\exp(X_t)dW_t \\ &= \frac{1}{2} f^2(t,\omega) Z_t dt + f(t,\omega) Z_t dW_t \end{aligned}$$

thus the first part is verified.

Similarly, let now:

$$Z_t = e^{X_t - \frac{1}{2}\int_0^t f^2(s,\omega)ds}$$

The function $g(t,x) = e^{x - \frac{1}{2}\int_0^t f^2(s,\omega)ds}$ is C^1 in t and smooth in x, and:

$$\frac{\partial}{\partial t} g(t,x) = \frac{\partial}{\partial t}\left[e^x \cdot e^{-\frac{1}{2}\int_0^t f^2 ds}\right]$$

$$= -\frac{1}{2} f^2(t,\omega)\exp\left(x - \frac{1}{2}\int_0^t f^2 ds\right) = -\frac{1}{2} f^2(t,\omega) g(t,x)$$

$$\partial_x g(t,x) = \partial_{xx} g(t,x) = g(t,x)$$

Then:

$$dZ_t = dg(t,X_t) = \left[\partial_t g(t,X_t) + \frac{1}{2} f^2(t,\omega)\partial_{xx} g(t,X_t)\right]dt + f(t,\omega)\partial_x g(t,X_t)dW_t$$

$$= \underbrace{\left[-\frac{1}{2} f^2(t,\omega) Z_t + \frac{1}{2} f^2(t,\omega) Z_t\right]dt}_{=0} + f(t,\omega) Z_t dW_t$$

2.2 Langevin equation

Derive the second moment equation for general linear Ito SDE, and find first and second moments of the Langevin equation.

Solution: The general linear Ito SDE has form:

$$dX_t = (a_1(t)X_t + a_2(t))dt + (b_1(t)X_t + b_2(t))dW_t$$

And integral form:

$$X_t = X_0 + \int_0^t (a_1(s)X_s + a_2(s))ds + \int_0^t (b_1(s)X_s + b_2(s))dW_s$$

Define $m(t) = E[X_t]$, then taking expectation on both sides of the integral form (assuming bounded total variation), we have:

$$m(t) = E[X_0] + \int_0^t a_1(s)E[X_s]ds + \int_0^t a_2(s)ds + 0$$

the zero comes from the fact that Ito integral is a martingale (see proof in Chapter 3 of textbook).

Differentiating both sides with respect to t yields:

$$m'(t) = a_1(t)m(t) + a_2(t)$$

Now define $Y_t = f(X_t) = X_t^2$; f is C^2 in x, therefore we use Ito's formula again:

$$dY_t = \left[2(a_1(t)X_t + a_2(t)) \cdot X_t + (b_1(t)X_t + b_2(t))^2\right]dt + 2(b_1(t)X_t + b_2(t))X_t dW_t$$

$$= \left[(2a_1 + b_1^2)X_t^2 + (2a_2 + 2b_1b_2)X_t + b_2^2\right]dt + (2b_1X_t^2 + 2b_2X_t)dW_t$$

The integral form:

$$Y_t = Y_0 + \int_0^t [(2a_1 + b_1^2)X_s^2 + (2a_2 + 2b_1b_2)X_s + b_2^2]ds + \int_0^t [2b_1X_s^2 + 2b_2X_s]dW_s$$

Let $D(t)$ denote $E[Y_t] = E[X_t^2]$, then:

$$D(t) = E[X_0^2] + \int_0^t [(2a_1 + b_1^2)D(s) + (2a_2 + 2b_1b_2)m(s) + b_2^2]ds + 0$$

$$D(t) = E[X_0^2] + \int_0^t (2a_1(s) + b_1^2(s))D(s)ds + \int_0^1 (2a_2(s) + 2b_1(s)b_2(s))m(s)ds + \int_0^t b_2^2(s)ds$$

Take derivative with respect to t on both sides, the constant terms vanish, and:

$$D'(t) = [2a_1(t) + b_1^2(t)]D(t) + [2a_2(t) + 2b_1(t)b_2(t)]m(t) + b_2^2(t)$$

77 the solution can also be found on textbook page 113, equation (2.11).

The Langevin equation reads:

$$dX_t = -aX_tdt + bdW_t$$

Let $m(t) = E[X_t]$, then taking expectation on both sides:

$$E[X_t] = E[X_0] - a\int_0^t E[X_s]ds \Leftrightarrow m(t) = E[X_0] - a\int_0^t m(s)ds$$

then:

$$m'(t) = -am(t)$$

Integral form:

$$X_t = X_0 - a\int_0^t X_sds + b\int_0^t dW_s$$

which has the analytic solution:

$$m(t) = m(0) \cdot e^{-at} = E[X_0] \cdot \exp(-at)$$

Define $Y_t = X_t^2$, then:

$$dY_t = [-2aX_t^2 + b^2]dt + 2bX_tdW_t = [-2aY_t + b^2]dt + 2bX_tdW_t$$

$$Y_t = Y_0 + \int_0^t (-2aY_s + b^2)ds + 2b\int_0^t X_s dW_s$$

$$E[Y_t] = E[Y_0] - 2a\int_0^t E[Y_s]ds + b^2\int_0^t ds + 0$$

$$E[Y_t] = E[Y_0] - 2a\int_0^t E[Y_s]ds + b^2 t$$

$$D'(t) = \frac{d}{dt}E[Y_t] = -2aD(t) + b^2$$

This ODE has analytic solution:

$$D(t) = \frac{b^2}{2a} + (E[X_0^2] - \frac{b^2}{2a})e^{-2at}$$

2.3 OU process generation

Generate the Ornstein-Uhlenbeck process numerically by discretizing the integral representation:

$$X_t = e^{-2t}X_0 + 2\int_0^t e^{-2(t-s)}dW_s$$

with left hand rule (which yields Ito), for a small grid size ds, for $t \in [0,1]$. Here X_0 is a $\mathcal{N}(0,1)$ random variable independent of the Brownian path $W_t, t > 0$.

Compute the covariance $\mathbb{E}\left[X_t X_s\right]$ numerically, and compare with the exact covariance $e^{-2|t-s|}$, to help guide choosing a good choice of ds.

Plot a sample path of the solution on $[0,1]$.

Solution: The derivation and code are as follows.

85

OU Process Generation

November 25, 2022

[1]:
```python
# numerical libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
%matplotlib inline
```

We have the integral:

$$X_t = e^{-2t}X_0 + 2\int_0^t e^{-2(t-s)}dW_s = e^{-2t}\left(X_0 + 2\int_0^t e^{2s}dW_s\right)$$

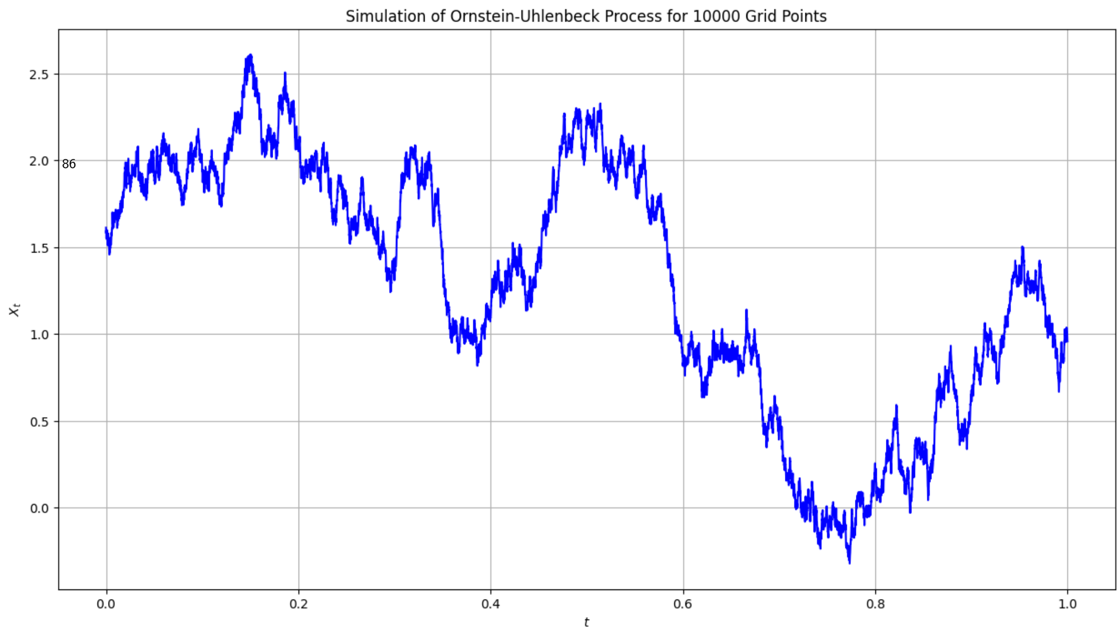
with $X_0 \sim \mathcal{N}(0, 1)$.

The only part that involves randomness is the simulation of the stochastic integral. Let $t_0, t_1, t_2, \cdots, t_N$ be a time grid with $t_0 = 0, t_{j+1} - t_j = \Delta t$, then:

$$\int_0^{t_j} e^{2s}dW_s \approx \sum_{i=0}^{j-1} e^{2t_i}[B_{t_{i+1}} - B_{t_i}]$$

Here $B_{t_{i+1}} - B_{t_i} \sim \mathcal{N}(0, \Delta t)$ are i.i.d. We implement one sample path on $t \in [0, 1]$ below; the process is simulated with 10000 grid points.

[3]:
```python
# for reproducibility
np.random.seed(1)
N = 10000
# initial condition
x0 = np.random.normal(0, 1, 1)
t_grid = np.linspace(0, 1, N)
dt = t_grid[1]-t_grid[0]
dWt = np.random.normal(loc=0.0, scale=np.sqrt(dt), size=N)
# Ito integral on grid points
ito = np.exp(-2*t_grid) * (x0 + 2 * np.cumsum(np.exp(2*t_grid) * dWt))
# plotting
plt.figure(1, figsize=(15, 8))
plt.plot(t_grid, ito, color='blue');
plt.grid(True); plt.xlabel(r'$t$'); plt.ylabel(r'$X_t$');
plt.title("Simulation of Ornstein-Uhlenbeck Process for {} Grid Points".
 ↪format(N));
```



0.1 Verifying Covariance

The resulting visualization will be a 3D plane, since we have access to discrete values X_{t_j} on grid points. We compute the estimated covariance using 10000 sample paths, each path is generated using 10000 grid points.

[4]:
```
# simulate 5000 paths
np.random.seed(12)
N = 10000
sample_size = 10000
t_grid = np.linspace(0, 1, N)
data = np.zeros([N, sample_size])
for i in range(sample_size):
    # draw a sample path
    x0 = np.random.normal(0, 1, 1)
    dWt = np.random.normal(loc=0.0, scale=np.sqrt(dt), size=N)
    ito = np.exp(-2*t_grid) * (x0 + 2 * np.cumsum(np.exp(2*t_grid) * dWt))
    data[:, i] = ito
```

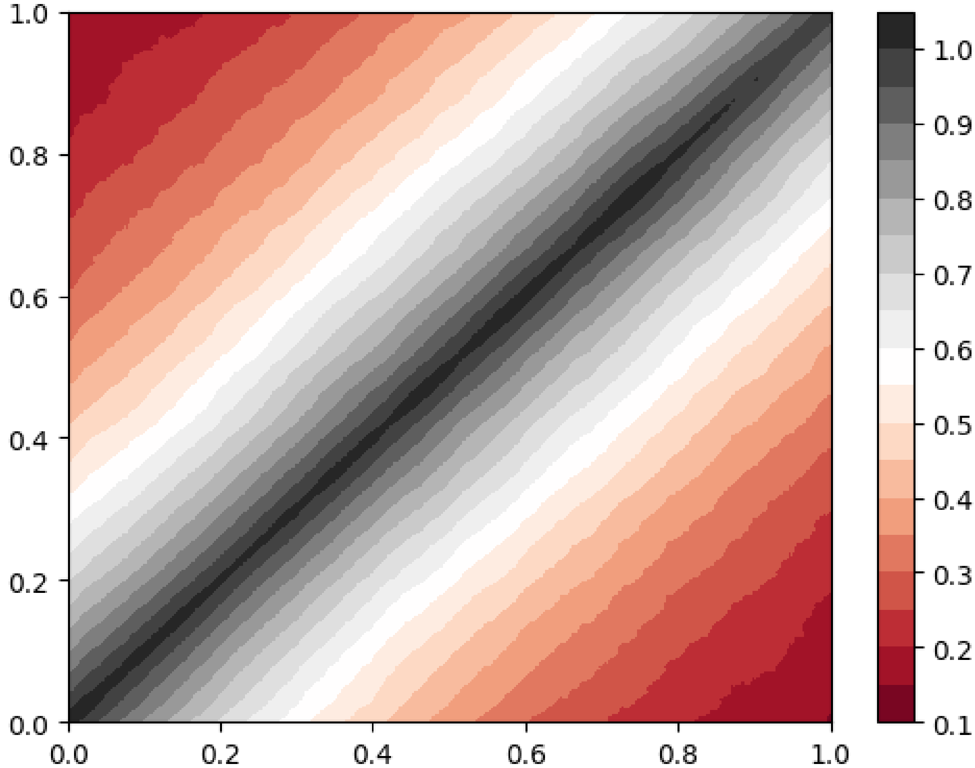
[5]:
```
data.shape
```

[5]: (10000, 10000)

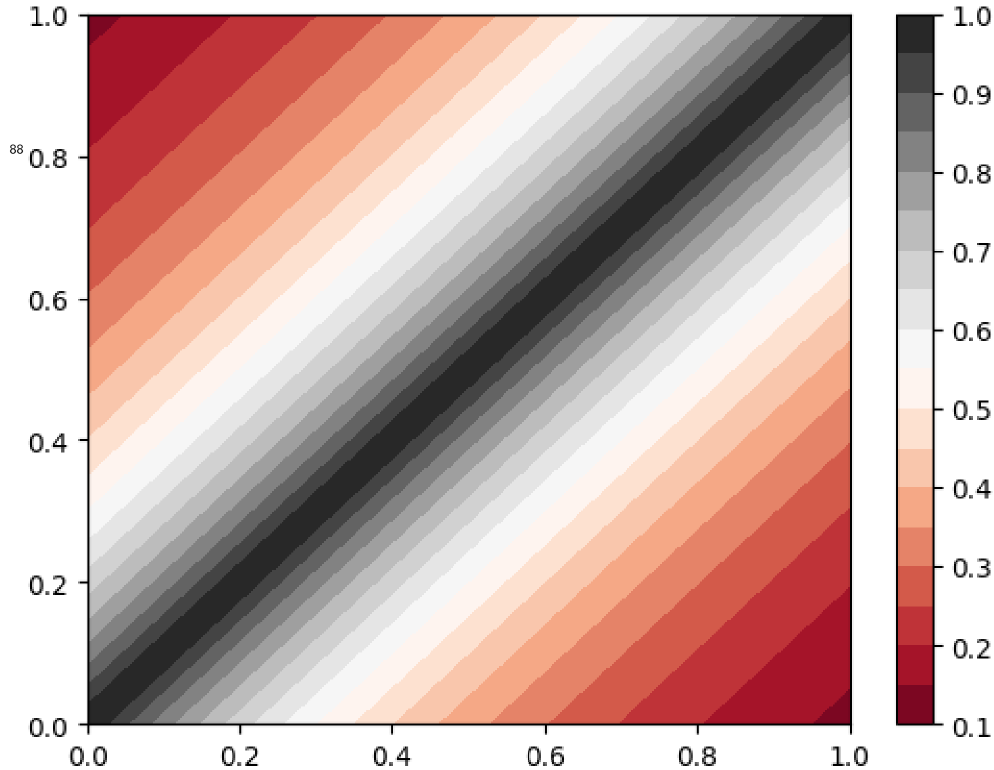
[6]:
```
# compute approximate covariance (result should be N x N grid)
numerical_cov = np.cov(data)
numerical_cov.shape
```

[6]: (10000, 10000)

[7]:
```
# plot grid
[s_mesh, t_mesh] = np.meshgrid(t_grid, t_grid)
plt.contourf(s_mesh, t_mesh, numerical_cov, 20, cmap='RdGy')
plt.colorbar();
```



[8]:
```
# plot exact covariance
exact = np.exp(-2*np.abs(s_mesh-t_mesh))
plt.contourf(s_mesh, t_mesh, exact, 20, cmap='RdGy')
plt.colorbar();
```



1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1

3 Project 3

3.1 Convergence of Linear SDE

Consider the SDE:

$$dX_t = aX_t dt + bX_t dW_t$$

with a, b constant. Discretize the SDE with Euler scheme and find the order of convergence for its third and fourth order moments.

Solution: More details can be found in the textbook section 9.7. We say a discrete time approximation Y^δ converges to X weakly at time T as $\delta \downarrow 0$ with respect to a class $\mathcal{C}$ of test functions if:

$$\lim_{\delta \to 0} |E(g(X_T)) - E(g(Y^\delta(T)))| = 0$$

for all $g \in \mathcal{C}$. Furthermore, Y^δ is said to converge weakly with order $\beta > 0$ if for each $g \in \mathcal{C}_P^{2(\beta+1)}(\mathbb{R}^d, \mathbb{R})$ (the class of $2(\beta + 1)$ times differentiable functions) if there is a constant $C > 0, \delta_0 < \infty$ such that:

$$|E[g(X_T)] - E[g(Y^\delta)]| \leq C\delta^\beta$$

for all $\delta \in (0, \delta_0)$.

From Theorem 9.7.4 we can verify that our constant coefficient SDE satisfies the assumptions with Euler scheme. Theorem 14.1.5 shows that Euler is at least order $\beta = 1$ weakly convergent. Thus we have:

$$|E[g(X_T)] - E[g(Y^\delta)]| \leq C\delta$$

for all $g \in \mathcal{C}_P^4$. Let $g_1(x) = x^3, g_2(x) = x^4$, it is clear that $g_1, g_2 \in \mathcal{C}_P^4$. On the other hand, this shows:

$$|E[X_T^3] - E[(Y^\delta)^3]| \leq C\delta, |E[X_T^4] - E[(Y^\delta)^4]| \leq C\delta$$

Respectively, this implies that the third and fourth moments of Y^δ are order-1 convergent. Note that it is incorrect to say "weak convergence" here because third and fourth moments are deterministic.

3.2 Propagating Front IBVP

Consider the initial-boundary value problem:

$$u_t = 0.0025u_{xx} + e^{\xi(x,w)}u(1-u), x \in [0, 15]$$

$$u(t, 0) = 1, u(t, 15) = 0, u(0, x) = \chi_{[0,1]}(x)$$

where $\xi(x, w)$ is the OU process with $\mathcal{N}(0, 1)$ as initial condition, and covariance $\mathbb{E}\big[\xi(x)\xi(0)\big] = e^{-2x}$. Use backward time centered in space scheme with proper stepsizes h, k ($h \leq 0.01$) to discretize the PDE.

3.2.1 Sample solutions

Plot sample solutions of u for $t = 0, 4, 8, 12, 16, 20$.

3.2.2 Sample generation

Generate $N \geq 1000$ samples. For each ensemble solution $u(\cdot, \cdot; \omega)$, define another random process: $X_t(\omega)$ such that $u(X_t(\omega), t; \omega) = 1/2$. Plot a histogram of $\eta_1(\omega) = X_{20}(\omega)/20$.

3.2.3 Summary statistic

Compute $c = \mathbb{E}\left[\eta_1\right]$ and:

$$c' = 2\sqrt{0.025 \cdot \mathbb{E}\left[e^{\xi}\right]}$$

c' is a naive estimate of the random front velocity. Compare c and c'.

Solution: The derivation of finite differencing and Python code are as follows.

111

Propagating Front IBVP

December 27, 2022

[2]:
```
# numerical libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
%matplotlib inline
```

Let T denote the discretized time grid, N denote the discretized spatial domain. Backward-time-centered-space refers to expanding around U_N^{T+1} using:

$$\partial_t u \approx \frac{U_N^{T+1} - U_N^T}{k}$$

$$\partial_{xx} u \approx \frac{U_{N+1}^{T+1} - 2U_N^{T+1} + U_{N-1}^{T+1}}{h^2}$$

Original equation:

$$u_t = 0.025 u_{xx} + e^{\xi(x,\omega)} u(1-u)$$

$$u(0,x) = \chi_{[0,1]}(x), u(t,0) = 1, u(t,T=15) = 0$$

where $\xi(x,\omega) = X_t$ is an OU-process with $X_0 \sim \mathcal{N}(0,1)$.

Rearranging, we obtain the implicit scheme (let i, j denote space and time indices); notice that we actually should use backward for the nonlinear term $u(1-u)$ and solve a nonlinear equation with, for instance, Newton's method. But most students implemented a forward scheme, and it is okay too.

$$\frac{1}{k}(u_{i,j+1} - u_{i,j}) = \frac{1}{40h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + \exp[\xi(x_i,\omega)]u_{i,j}(1-u_{i,j})$$

$$u_{i,j+1} = u_{i,j} + \frac{k}{40h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + k\exp[\xi(x_i,\omega)]u_{i,j}(1-u_{i,j})$$

[3]:
```
# fix seed
np.random.seed(1)
h = 0.01
k = 0.0001

x = np.arange(0,15,h)
t_end = 20 # 0, 4, 8, 12, 16, 20.
```

```
t = np.arange(0,t_end,k)

# number of spatial points
num_space = len(x)
# number of time points
num_time = len(t)

# pre-allocate
u = np.zeros([num_time, num_space])
# initial conditions
u[:, 0] = 1
u[:, -1] = 0
indicator_idx = np.where(x == 1)[0][0]
u[0, 0:indicator_idx+1] = 1
# generate Brownian motion
dWt = np.sqrt(h)*np.random.randn(num_space)
# Simulate OU-Process (See Project 2 Solutions)
x0 = np.random.randn()
xi_t = np.exp(-2*x)*(x0 + 2*np.cumsum(np.exp(2*x) * dWt))
# forward time stepping
for i in np.arange(1, num_time):
    u[i, 1:num_space-1] = u[i-1, 1:num_space-1] + (k/(40*(h**2))) * (u[i-1, 2:
 ↪num_space] - \
                                                                  2*u[i-1, 1:
 ↪num_space-1] + \
                                                                  u[i-1, 0:
 ↪num_space-2]) + \
                                                                  k * np.
 ↪exp(xi_t[1:num_space-1]) * \
                                                                      u[i-1, 1:
 ↪num_space-1] * \
                                                                      (1 - u[i-1,␣
 ↪1:num_space-1])
# end for loop
```

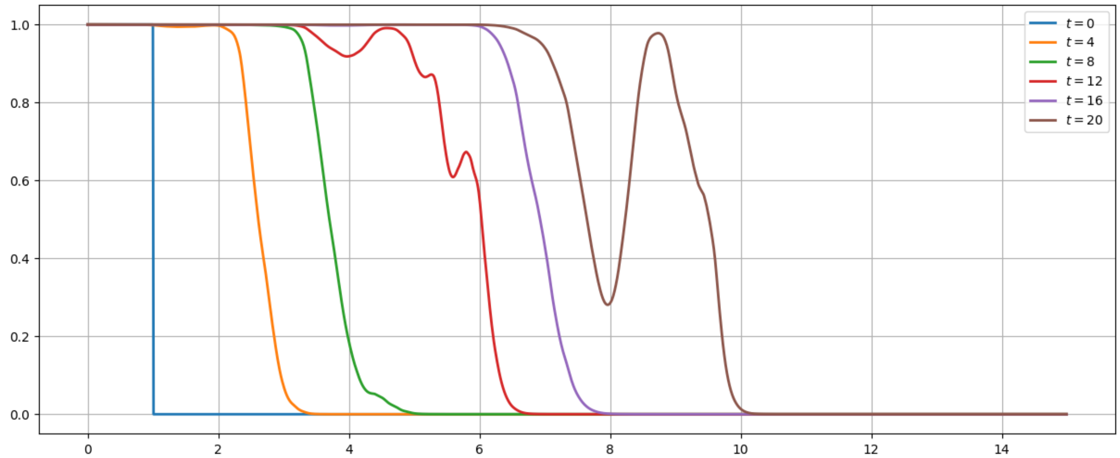
(a) Plot sample solution

[4]:

```
# find indices of t
idx4 = np.where(t == 4)[0][0]
idx8 = np.where(t == 8)[0][0]
idx12 = np.where(t == 12)[0][0]
idx16 = np.where(t == 16)[0][0]
plt.figure(1, figsize=(15, 6));
plt.plot(x, u[0, :], lw=2, label=r"$t=0$");
plt.plot(x, u[idx4, :], lw=2, label=r"$t=4$");
plt.plot(x, u[idx8, :], lw=2, label=r"$t=8$");
plt.plot(x, u[idx12, :], lw=2, label=r"$t=12$");
```

```
plt.plot(x, u[idx16, :], lw=2, label=r"$t=16$");
plt.plot(x, u[-1, :], lw=2, label=r"$t=20$");
plt.grid(True);
plt.legend();
```

113



(b) Histogram Due to randomness, u will not be exactly 2, we thus find all solutions such that:

$$|u(\cdot, \cdot; \omega) - \frac{1}{2}| < \epsilon$$

where we choose ϵ small. We take $N = 2000$ as our sample size.

[10]:
```
mc = 2000

h = 0.01
k = 0.001

x = np.arange(0,15,h)
t_end = 20 # 0, 4, 8, 12, 16, 20.
t = np.arange(0,t_end,k)

# number of spatial points
num_space = len(x)
# number of time points
num_time = len(t)

# ensemble solution
save_x = np.zeros([mc, num_space])

# preallocate
eta1 = []
u_mean = np.zeros((num_time,num_space))
```

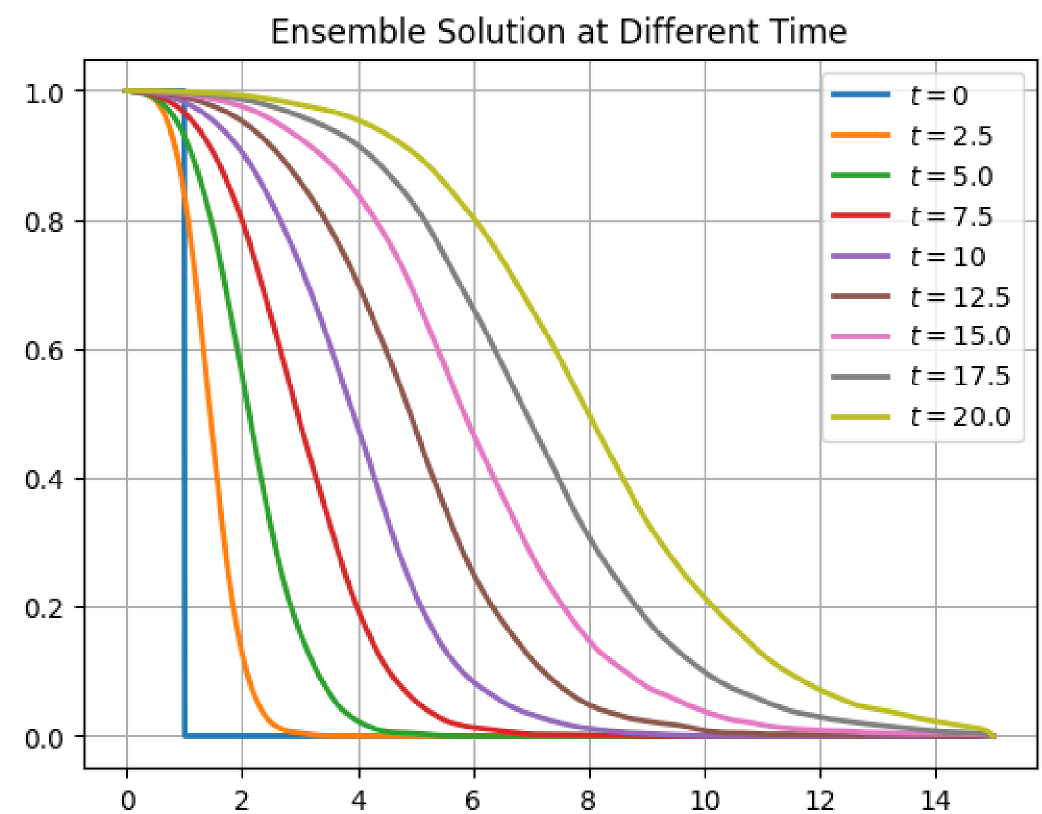
```python
for idx in range(mc):
    if idx % 100 == 0:
        print("[#] sample {}".format(idx))
    # pre-allocate
    u = np.zeros([num_time, num_space])
    # initial conditions
    u[:, 0] = 1
    u[:, -1] = 0
    indicator_idx = np.where(x == 1)[0][0]
    u[0, 0:indicator_idx+1] = 1
    # generate Brownian motion
    dWt = np.sqrt(h)*np.random.randn(num_space)
    # Simulate OU-Process (See Project 2 Solutions)
    x0 = np.random.randn()
    xi_t = np.exp(-2*x)*(x0 + 2*np.cumsum(np.exp(2*x) * dWt))

    # forward time stepping
    for i in np.arange(1, num_time):
        u[i, 1:num_space-1] = u[i-1, 1:num_space-1] + (k/(40*(h**2))) * (u[i-1, 2:num_space] - \
                                                                          2*u[i-1, 1:num_space-1] + \
                                                                          u[i-1, 0:num_space-2]) + \
                                                                          k * np.exp(xi_t[1:num_space-1]) * \
                                                                              u[i-1, 1:num_space-1] * \
                                                                              (1 - u[i-1, 1:num_space-1])
    u_mean += u / mc

    # after solving u, take 1/2-process
    minimum_idx = np.argmin(np.abs(u[-1, :]-0.5))
    eta1.append(x[minimum_idx])
```

```
[#] sample 0
[#] sample 100
[#] sample 200
[#] sample 300
[#] sample 400
[#] sample 500
[#] sample 600
[#] sample 700
[#] sample 800
[#] sample 900
[#] sample 1000
```

```
[#] sample 1100
[#] sample 1200
[#] sample 1300
[#] sample 1400
[#] sample 1500
[#] sample 1600
[#] sample 1700
[#] sample 1800
[#] sample 1900
```

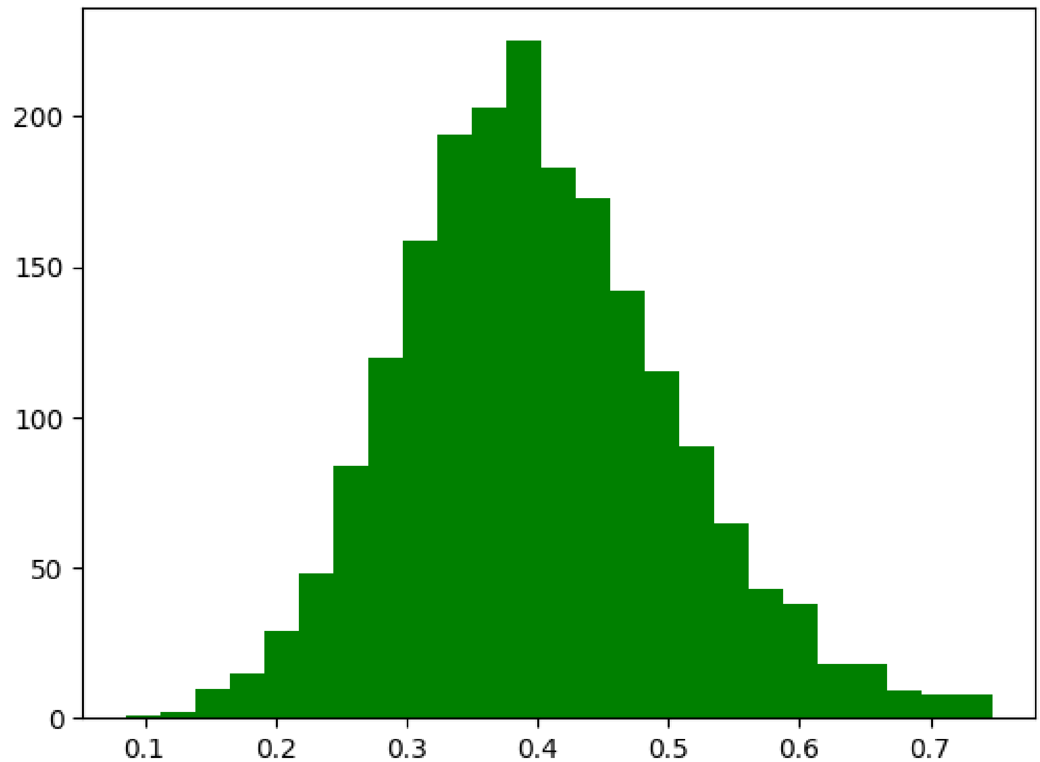
As a sanity check, the ensemble solution should correspond to roughly a pure advection-diffusion.

[24]:
```
plt.plot(x, u_mean[0, :], lw=2, label=r"$t=0$");
plt.plot(x, u_mean[2500, :], lw=2, label=r"$t=2.5$");
plt.plot(x, u_mean[5000, :], lw=2, label=r"$t=5.0$");
plt.plot(x, u_mean[7500, :], lw=2, label=r"$t=7.5$");
plt.plot(x, u_mean[10000, :], lw=2, label=r"$t=10$");
plt.plot(x, u_mean[12500, :], lw=2, label=r"$t=12.5$");
plt.plot(x, u_mean[15000, :], lw=2, label=r"$t=15.0$");
plt.plot(x, u_mean[17500, :], lw=2, label=r"$t=17.5$");
plt.plot(x, u_mean[-1, :], lw=2, label=r"$t=20.0$");
plt.title("Ensemble Solution at Different Time")
plt.grid(True);
plt.legend();
```



(c) Estimate speed

[21]:
```
# histogram
plt.figure(2);
eta1 = np.array(eta1)/20
plt.hist(eta1, color='green', bins=25);
```



[ ]:
```
c = np.mean(eta1)
# simulate exp(xi) N times
mc = 10000
ensemble_exp_xi = 0
for idx in range(mc):
    dWt = np.sqrt(h)*np.random.randn(num_space)
    x0 = np.random.randn()
    xi_t = np.exp(-2*x)*(x0 + 2*np.cumsum(np.exp(2*x) * dWt))
    ensemble_exp_xi += np.exp(xi_t)[-1]
ensemble_exp_xi /= mc
c_prime = 2 * np.sqrt(0.025 * ensemble_exp_xi)
print("[*] Compare c = {}, c' = {}".format(c, c_prime))
```

3.3 SDE solution

Consider the SDE:

$$dX_t = aX_t dt + bX_t dW_t$$

which has exact solution:

$$X_t = X_0 \exp\left((a - \frac{b^2}{2})t + bW_t\right)$$

let $X_0 = 1$, $a = 1.5, b = 1$, solve the above SDE using Euler and Milstein schemes for t up to 1, using time discretizations $\delta = 2^{-3}, 2^{-4}, 2^{-5}, 2^{-6}$.

3.3.1 Visualization

Plot and compare sample solutions together with the exactly solution.

3.3.2 Sample generation

Generate 20000 samples for each value of δ, and compute the absolute error $\epsilon = \mathbb{E}\big[|X(1) - Y_\delta(1)|\big]$, where $Y_\delta(1)$ is the final solution at $t = 1$ numerically computed with discretization level δ. Plot $\epsilon(\delta)$ against different choices of δ, and discuss the order of accuracy of Euler method and Milstein method.

Solution: (Presented on the next page)

128

SDE Solution

December 27, 2022

0.1 Compare Euler and Miltstein Solver

(a) Visualization Simulate:

$$dX_t = aX_t dt + bX_t dW_t, X_0 = 1$$

whose exact solution is:

$$X_t = \exp\left[(a - \frac{b^2}{2})t + bW_t\right]$$

Recall Euler scheme:

$$X_{n+1} = X_n + aX_n\Delta t + bX_n(W_{n+1} - W_n), W_{n+1} - W_n = \sqrt{\Delta t}Z, Z \sim \mathcal{N}(0, 1)$$

And Milstein scheme:

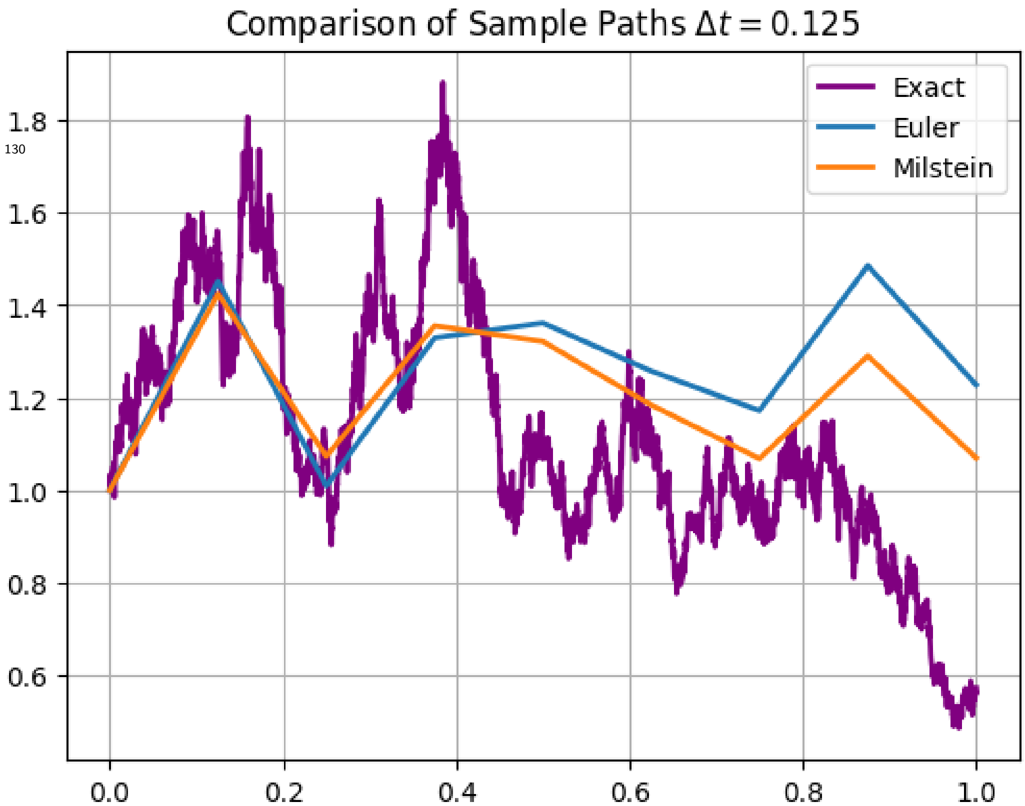
$$X_{n+1} = X_n + aX_n\Delta t + bX_n\Delta W_n + \frac{1}{2}b^2 X_n((\Delta W_n)^2 - \Delta t)$$

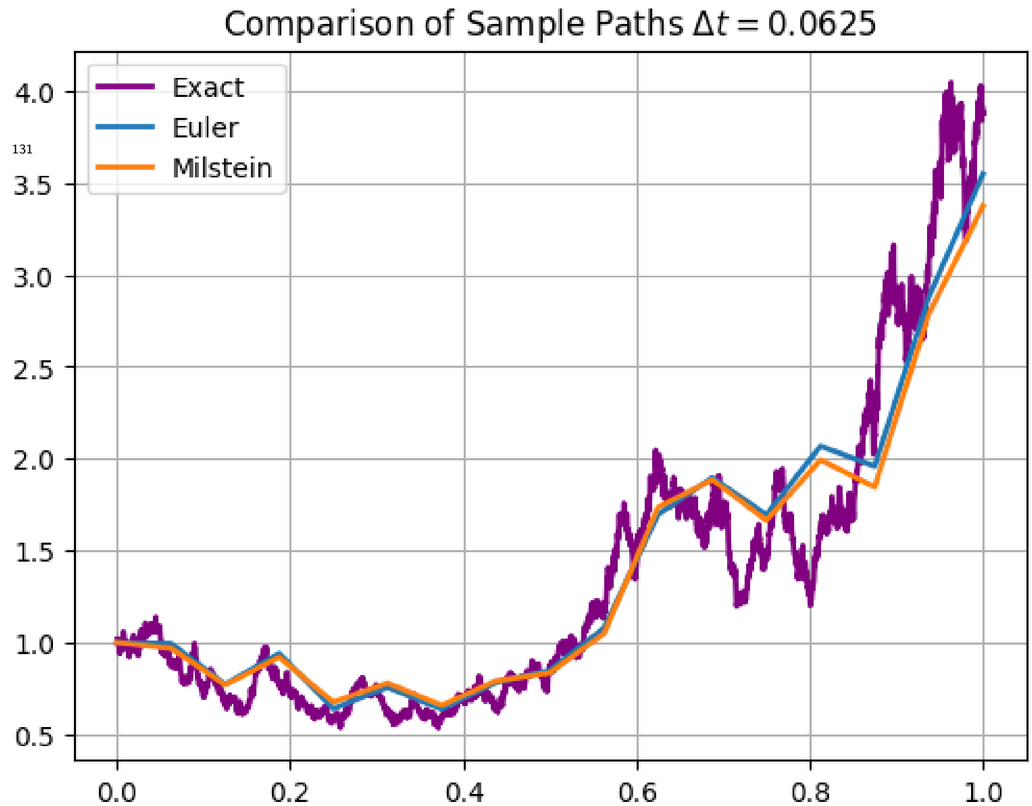
where $\Delta W_n = W_{n+1} - W_n$.

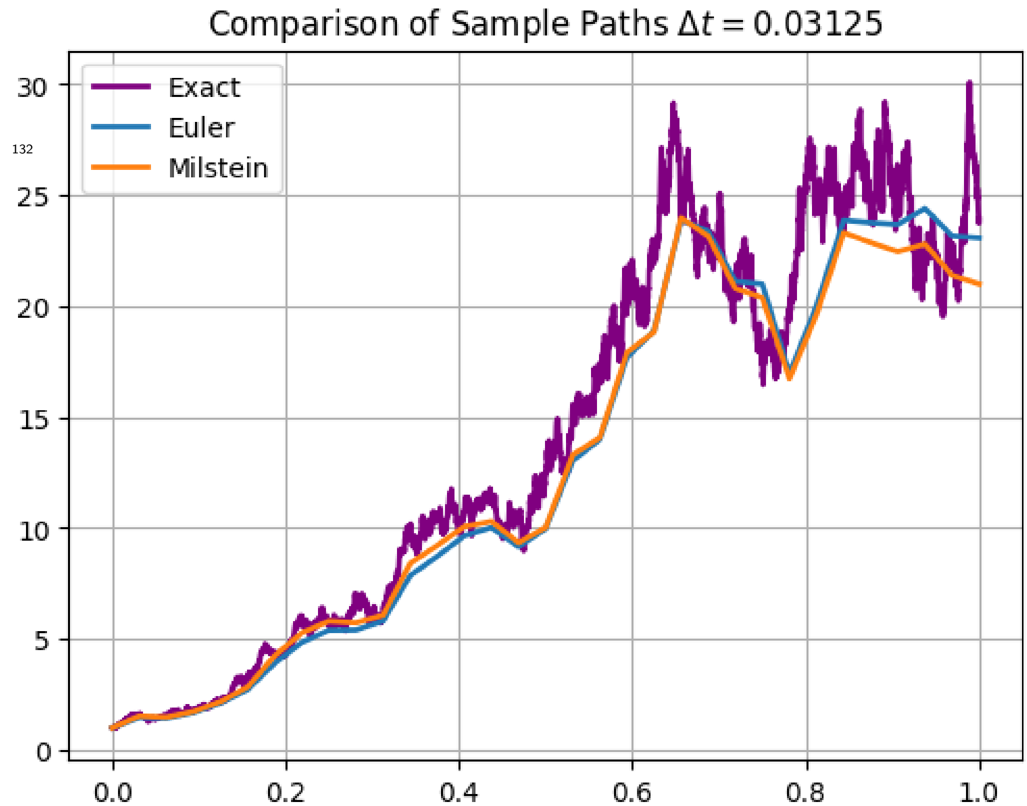
For sufficient resolution, we estimate the exact solution with a small step size, such as $\Delta t = 0.0001$.

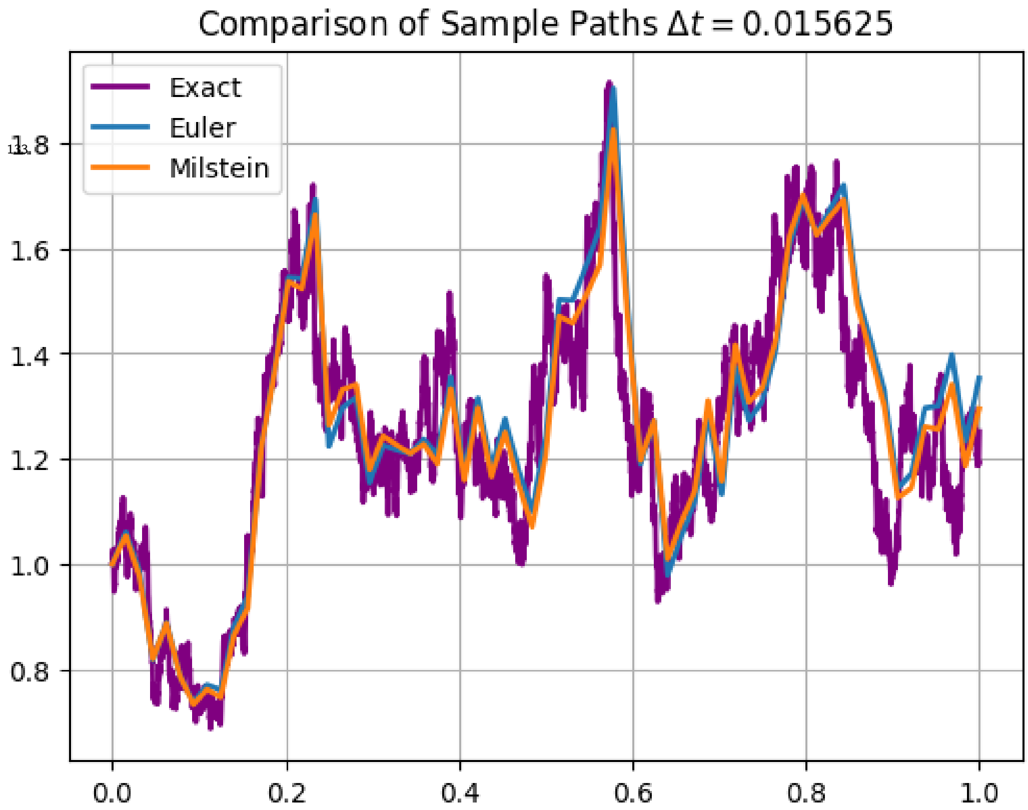
[1]:
```python
# numerical libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
%matplotlib inline

# helper function
def exact_solution(dWt):
    nt = len(dWt)
    dt = 1/nt
    X_exact = [0]
    for i in range(nt):
        X_exact.append(X_exact[-1]+dt+dWt[i])
    X_exact = [np.exp(i) for i in X_exact]
    return X_exact
```

```python
# fix seed
np.random.seed(10)
# fixed params
a = 1.5
b = 1   129
# configure
all_n = np.array([3, 4, 5, 6])
num_trials = len(all_n)
all_dt = np.array([1/(2**s) for s in all_n])
dt_exact = 2**(-12)
t_exact = np.arange(0, 1+dt_exact, dt_exact)
nt_exact = len(t_exact)

for i in range(num_trials):
    dt = all_dt[i]
    # time grid
    t = np.arange(0, 1+dt, dt)
    nt = len(t)

    # simulate a Wiener process using the fine discretization
    dWt = np.sqrt(dt_exact)*np.random.randn(int(1/dt_exact))
    # exact solution
    X_exact = exact_solution(dWt)
    # need to query the Wiener process at coarser leve;
    reduce = int(nt_exact/nt)
    sub_dWt = np.array([np.sum(dWt[(reduce*i):(reduce*(i+1))]) for i in range(nt)])

    X_euler = np.zeros(nt)
    X_mil = np.zeros(nt)
    X_euler[0] = 1
    X_mil[0] = 1

    for j in np.arange(0, nt-1):
        # Euler scheme
        X_euler[j+1] = X_euler[j] + a*X_euler[j]*dt + b*X_euler[j]*sub_dWt[j]
        # Milstein scheme
        X_mil[j+1] = X_mil[j] + a*X_mil[j] * dt + b*X_mil[j] * sub_dWt[j] + 0.5*X_mil[j]*((sub_dWt[j])**2-dt)
    # plotting
    plt.figure(i);
    plt.plot(t_exact, X_exact, lw=2, color='purple')
    plt.plot(t, X_euler, t, X_mil, lw=2);
    plt.legend(['Exact', 'Euler', 'Milstein']); plt.title(r"Comparison of Sample Paths $\Delta t = {}$".format(dt))
    plt.grid(True);
```


Comparison of Sample Paths $\Delta t = 0.125$
Exact
Euler
Milstein
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.0
0.2
0.4
0.6
0.8
1.0


Comparison of Sample Paths $\Delta t = 0.0625$
Exact
Euler
Milstein
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.2
0.4
0.6
0.8
1.0


Comparison of Sample Paths $\Delta t = 0.03125$
Exact
Euler
Milstein
0
5
10
15
20
25
30
0.0
0.2
0.4
0.6
0.8
1.0



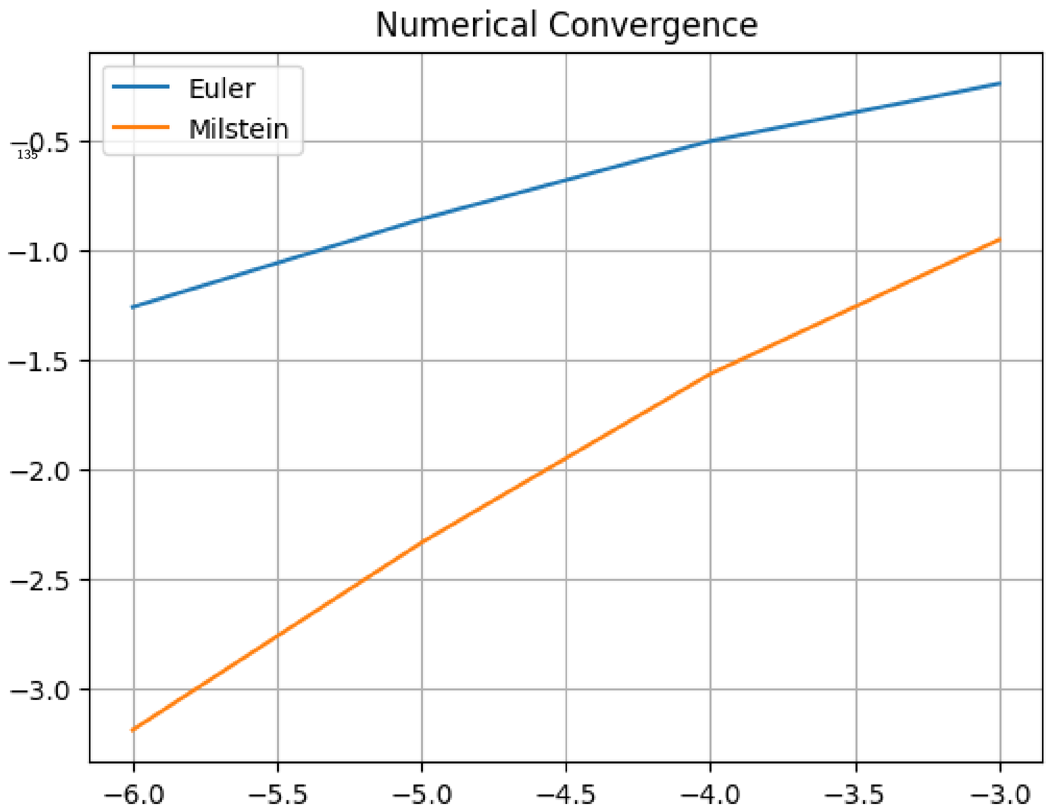
(b) Sample Generation To compare at the final time, the exact solution is computed using a fine discretization, $\Delta t = 2^{-12}$.

[2]:
```
# fix seed
mc = 5000

err_euler = np.zeros([num_trials, mc])
err_mil = np.zeros([num_trials, mc])
all_dt = [2**(-i) for i in range(3, 7)]
dt_exact = 2**(-12)
t_exact = np.arange(0, 1+dt_exact, dt_exact)
nt_exact = len(t_exact)
for i in range(mc):
    # generate exact solution using a fine level
    dWt = np.sqrt(dt_exact)*np.random.normal(loc=0, scale=1, size=nt_exact)
    Wt = np.cumsum(dWt)
    # exact solution
    X_exact = exact_solution(dWt)
    for k in range(3, 7):
        dt = 1/(2**k)
```

```
        t = np.arange(0, 1+dt, dt)
        nt = len(t)
        # compute Euler and Milstein solutions
        reduce = int(nt_exact/nt)
134     sub_dWt = np.array([np.sum(dWt[(reduce*zz):(reduce*(zz+1))]) for zz in range(nt)])
        # exact solution query points
        X_exact_query = np.array([X_exact[(reduce*zz)] for zz in range(nt)])
        X_euler = np.zeros(nt)
        X_mil = np.zeros(nt)
        X_euler[0] = 1
        X_mil[0] = 1
        for j in np.arange(0, nt-1):
            # Euler scheme
            X_euler[j+1] = X_euler[j] + a*X_euler[j]*dt + b*X_euler[j]*sub_dWt[j]
            # Milstein scheme
            X_mil[j+1] = X_mil[j] + a*X_mil[j] * dt + b*X_mil[j] * sub_dWt[j] + 0.5*X_mil[j]*((sub_dWt[j])**2 - dt)
        # compute final error
        err_euler[k-3, i] = np.max(np.abs(X_euler-X_exact_query))
        err_mil[k-3, i] = np.max(np.abs(X_mil-X_exact_query))
```

Convergence plot

[3]:
```
plt.figure(1);
plt.plot(np.log2(all_dt), np.log2(err_euler.mean(1)), np.log2(all_dt), np.log2(err_mil.mean(1)));
plt.legend(['Euler', 'Milstein']);
plt.title('Numerical Convergence');
plt.grid(True);
```


Numerical Convergence
Euler
Milstein
−0.5
−1.0
−1.5
−2.0
−2.5
−3.0
−6.0
−5.5
−5.0
−4.5
−4.0
−3.5
−3.0

4 Project 4

4.1 Multiple dimensional Wiener process

Given:

$$dX_t = adt + \sum_{j=1}^{m} b^j dW_t^j$$

determine: $\underline{f}_{(j_1,j_2,j_3,j_4)}$ and $f_{(j_1,j_2,j_3,j_4)}$ for $j_1, j_2, j_3, j_4 \in \{1, \dots, m\}$.

Solution:

We have:

$$dX = adt + \sum_{j=1}^{m} b^j dW^j$$

Please see Lecture 8 and Lecture 10 for a detailed discussion of the simplified notations (Ito-Taylor and Stratonovich-Taylor forms). The solution to this question is quite mechanical and it is enough to carry out the recursive definition. Note that b^j is indexing the coefficient for each process, not power.

$$f_\alpha = \begin{cases} f : l = 0 \\ L^{j_1} f_{-\alpha}, l \geq 1 \end{cases}$$

where:

$$L^j = \sum_{k=1}^{d} b^{k,j} \frac{\partial}{\partial x^k} = b^j \frac{\partial}{\partial x}$$

here $d = 1$, thus the last equality.

$$L^0 = \frac{\partial}{\partial t} + a \frac{\partial}{\partial x} + \frac{1}{2} \sum_{j=1}^{m} (b^j)^2 \frac{\partial^2}{\partial x^2}$$

Thus:

$$f_{(j_1,j_2,j_3,j_4)} = L^{j_1} f_{(j_2,j_3,j_4)} = \dots = L^{j_1} L^{j_2} L^{j_3} L^{j_4} f$$

and notice $j_1, j_2, j_3, j_4 \geq 1, f(t,x) = x, \frac{\partial f}{\partial x} = 1$:

$$L^{j_4} f = b^{j_4}$$

$$L^{j_3}(L^{j_4} f) = L^{j_3}(b^{j_4}) = b^{j_3} \frac{\partial}{\partial x} b^{j_4} = b^{j_3} b^{j_4'}$$

$$L^{j_2}(L^{j_3} L^{j_4} f) = L^{j_2}(b^{j_3} b^{j_4'}) = b^{j_2} \frac{\partial}{\partial x} b^{j_3} b^{j_4'} = b^{j_2}(b^{j_3'} b^{j_4'} + b^{j_3} b^{j_4''}) = b^{j_2} b^{j_3'} b^{j_4'} + b^{j_2} b^{j_3} b^{j_4''}$$

$$f_\alpha = L^{j_1} b^{j_2} b^{j_3'} b^{j_4'} + L^{j_1} b^{j_2} b^{j_3} b^{j_4''} = b^{j_1} \frac{\partial}{\partial x} (b^{j_2} b^{j_3'} b^{j_4'} + b^{j_2} b^{j_3} b^{j_4''})$$

$$\frac{\partial}{\partial x} b^{j_2} b^{j_3'} b^{j_4'} = b^{j_2'} b^{j_3'} b^{j_4'} + b^{j_2} b^{j_3''} b^{j_4'} + b^{j_2} b^{j_3'} b^{j_4''}$$

$$\frac{\partial}{\partial x} b^{j_2} b^{j_3} b^{j_4''} = b^{j_2'} b^{j_3} b^{j_4''} + b^{j_2} b^{j_3'} b^{j_4''} + b^{j_2} b^{j_3} b^{j_4'''}$$

Finally:

$$f_\alpha = b^{j_1}(b^{j_2'} b^{j_3'} b^{j_4'} + b^{j_2} b^{j_3''} b^{j_4'} + 2b^{j_2} b^{j_3'} b^{j_4''} + b^{j_2} b^{j_3} b^{j_4''} + b^{j_2} b^{j_3} b^{j_4'''})$$

The Stratonovich-Ito case is omitted.

4.2 Approximation of stochastic integral

Consider:

$$W_1^1 I_{(1,2)}[1]_{0,1}$$

4.2.1 Random coefficients

Find a representation of $I_{(1,2)}$ in terms of some random coefficients.

Solution:

$$l((1,2)) = 2$$

thus:

$$I_{(1,2)} = J_{(1,2)} - \frac{1}{2} I_{\{j_1 = j_2 \neq 0\}} J_{(0)} = J_{(1,2)}$$

because $j_1 \neq j_2$.

$$J_{(1,2)} = \frac{1}{2} W_\Delta^1 W_\Delta^2 - \frac{1}{2}(a_{2,0} W_\Delta^1 - a_{1,0} W_\Delta^2) + \Delta(\frac{\pi}{\Delta} \sum_{r=1}^{\infty} r(a_{1,r} b_{2,r} - b_{1,r} a_{2,r}))$$

Here the coefficients $a_{i,j}, b_{i,j}$ are normally distributed and pairwise independent.

4.2.2 Monte Carlo

Compute $\mathbb{E}\big[W_1^1 I_{(1,2)}[1]_{0,1}\big]$ with Monte Carlo based on a truncation of part 1.

Solution: (See next page)

4.2.3 Direct simulations

Compute $\mathbb{E}\big[W_1^1 I_{(1,2)}[1]_{0,1}\big]$ again directly using Wiener process samples and integrating along the paths.

Solution: (See next page)

Monte Carlo

158

November 26, 2022

[1]:
```python
# numerical libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
%matplotlib inline
```

[3]:
```python
np.random.seed(10) # reproducibility
Delta = 1
dt = 0.001
t = np.arange(0, Delta+dt, dt)
Nt = len(t)
# simulate two Wiener processes
dW1 = np.random.normal(size=Nt)
dW2 = np.random.normal(size=Nt)
W1 = np.cumsum(dW1)
W2 = np.cumsum(dW2)
W1_Delta = np.random.randn()
W2_Delta = np.random.randn()
# terms in approximation
Xi1 = W1_Delta
Xi2 = W2_Delta
# order of approximation
p = 50
mean = 0
nmc = 10**4
for _ in range(nmc):
    # generate all coefficients
    a_jr = np.zeros([2, p])
    b_jr = np.zeros([2, p])
    zeta_jr = np.zeros([2, p])
    eta_jr = np.zeros([2, p])
    for idx in range(p):
        r = idx + 1
        a_jr[:, idx] = np.random.normal(0, 1/(2*((np.pi)**2) * (r**2)), size=2)
        b_jr[:, idx] = np.random.normal(0, 1/(2*((np.pi)**2) * (r**2)), size=2)
        zeta_jr[:, idx] = np.sqrt(2)*np.pi*r*a_jr[:, idx]
        eta_jr[:, idx] = np.sqrt(2)*np.pi*r*b_jr[:, idx]
```

```
    a10 = -np.sqrt(2)/np.pi * np.sum((1 / np.arange(1, p+1)) * zeta_jr[0, :]) # ignore tail series
    a20 = -np.sqrt(2)/np.pi * np.sum((1 / np.arange(1, p+1)) * zeta_jr[1, :])
    Ap_12 = (0.5/np.pi) * np.sum(( (1 / np.arange(1, p+1) ) * (zeta_jr[0, :] * eta_jr[1, :] - \
                                                              eta_jr[0, :] * zeta_jr[1, :])))
    Jp_12 = 0.5*Xi1*Xi2 - 0.5*(a20*Xi1 - a10*Xi2) + Ap_12
    result = W1_Delta * Jp_12
    mean = mean + result / nmc
mean
```

[3]: -0.0009368538586441822

Direct Integration

160

November 26, 2022

[1]:
```
# numerical libraries
import numpy as np
import scipy
import matplotlib.pyplot as plt
%matplotlib inline
```

Path-sampling approximation requires us to simulate:

$$I_{(1,2)}[1]_{0,1} = \int_0^1 W_s^1 dW_s^2$$

[3]:
```
np.random.seed(10)
nmc = 10000
mean2 = 0
for _ in range(nmc):
    # first Wiener process
    dt = 0.001
    Delta = 1
    t = np.arange(0, Delta+dt, dt)
    Nt = len(t)
    dW1 = np.sqrt(dt) * np.random.randn(Nt)
    W1 = np.cumsum(dW1)
    # second Wiener process
    dW2 = np.sqrt(dt) * np.random.randn(Nt)
    # integration
    I_12 = np.sum(W1 * dW2)
    # final result
    mean2 += W1[-1] * I_12/nmc
mean2
```

[3]:
```
0.006803603192876089
```

We see that both methods approximate the true mean 0.

5 Project 5

5.1 L^p strong convergence

Finish the proof of L^p strong convergence.

Solution sketch: Follow the proof for the L^2 case in Platen 10.7. For the L^p generalization, one may cite Doob's maximal inequality for submartingales in L^p on $X_t - Y^\delta(t)$, and derive an analogous inequality as (7.7), then apply the tower property on both sides by taking expectations.

5.2 SDE problem

For $t \geq t_0 = 0$, consider:

$$dX_t = (\frac{2}{1+t}X_t + (1+t)^2)dt + (1+t)^2 dW_t$$

with $X_0 = 1$.

5.2.1 Exact solution

Verify that it has an exact solution:

$$X_t = (1+t)^2(1 + W_t + t)$$

Solution: We use Ito formula to take the differential of X_t:

$$dX_t = \left[3(1+t)^2 + 2(1+t)W_t\right]dt + (1+t)^2 dW_t$$

and noting that:

$$\frac{2}{1+t}X_t + (1+t)^2 = 3(1+t)^2 + 2(1+t)W_t$$

we conclude that the solution indeed matches, with $X_0 = 1 \cdot (1 + 0 + 0) = 1$.

5.2.2 Approximation of final solution

Approximating X_T with the Chang scheme, for $T = 0.5$, in which $J_{(1,1,0)}$ is approximated by $J^p_{(1,1,0)}$ for $p = 15$. Make this approximation for equal step sizes $\delta = 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}$ and record the absolute error.

Solution: Recall the nonautonomous Chang scheme (Chapter 11, Platen):

$$Y_{n+1} = Y_n + \frac{1}{2}\left[a(t_n + \frac{1}{2}\Delta t, \bar{Y}_+) + a(t_n + \frac{1}{2}\Delta t, \bar{Y}_-)\right]\Delta t \tag{1}$$

$$+b(t_n)\Delta W + \frac{1}{\Delta t}(b(t_{n+1}) - b(t_n)) \cdot (\Delta W \Delta t - \Delta Z)$$

where:

$$\bar{Y}_{\pm} = Y_n + \frac{1}{2}\underline{a}(t_n, Y_n)\Delta t + \frac{1}{\Delta t} \cdot b(t_n)\left[\Delta Z \pm \sqrt{2J_{(1,1,0)}\Delta t - (\Delta Z)^2}\right]$$

and $\underline{a}$ is the Stratonovich corrected drift:

$$\underline{a} = a - \frac{1}{2}bb'$$

In this question, we approximate $J_{(1,1,0)}$ as described in Exercise 10.5.1 of Platen, with $p = 15$:

$$J_{(1,1,0)} \approx J^p_{(1,1,0)} = \frac{1}{3!}(\Delta t)^2\zeta_1^2 + \frac{1}{4}(\Delta t)a_{1,0}^2 - \frac{1}{2\pi}(\Delta t)^{3/2}\zeta_1 b_1$$

$$+\frac{1}{4}(\Delta t)^{3/2}a_{1,0}\zeta_1 - (\Delta t)^2 C^p_{1,1}$$

And:

$$C^p_{1,1} = -\frac{1}{2\pi^2}\sum_{r,l=1,r\neq l}^{p}\frac{r}{r^2 - l^2}\left(\frac{1}{l}\xi_{1,r}\xi_{1,l} - \frac{l}{r}\eta_{1,r}\eta_{1,l}\right)$$

with the following definitions:

$$\begin{cases} a_{1,0} = -\frac{1}{\pi}\sqrt{2\Delta t}\sum_{r=1}^{p}\frac{1}{r}\xi_{1,r} - 2\sqrt{(\Delta t)\rho_p}\mu_{1,p} \\ \rho_p = \frac{1}{12} - \frac{1}{2\pi^2}\sum_{r=1}^{p}\frac{1}{r^2} \\ b_1 = \sqrt{\frac{\Delta t}{2}}\sum_{r=1}^{p}\frac{1}{r^2}\eta_{1,r} + \sqrt{(\Delta t)\alpha_p}\phi_{1,p} \\ \alpha_p = \frac{\pi^2}{180} - \frac{1}{2\pi^2}\sum_{r=1}^{p}\frac{1}{r^4} \end{cases}$$

here $\zeta_1, \xi_{1,r}, \eta_{1,r}, \mu_{1,p}, \phi_{1,p}$ for $r = 1, \ldots, p$ are independent standard Gaussian random variables.

The code that implements the Chang scheme is on the next page, along with the exact sample paths.

5.2.3 Visualization

Plot the absolute error against $\log_2 \delta$ and explain order of convergence.

Solution: The plotting is omitted. One may investigate its strong order of convergence by computing an average absolute final error.

185

Chang Scheme

November 27, 2022

Solve:

$$dX_t = \left[\frac{2}{1+t}X_t + (1+t)^2\right]dt + (1+t)^2 dW_t$$

with the Chang scheme.

As a comparison, we simulate $M = 20$ batches each of $N = 100$ sample paths, and vary time steps $\delta = \Delta = 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}$ up to $T = 0.5$.

The Ito SDE gives:

$$a(t,x) = \frac{2x}{1+t} + (1+t)^2, b(t) = (1+t)^2$$

then the Stratonovich corrected drift is:

$$\underline{a}(t,x) = a(t,x) - \frac{1}{2}b(t)b'(t) = \frac{2x}{1+t} + (1+t)^2 - \frac{1}{2}\cdot(1+t)^2\cdot 2(1+t) = \frac{2x}{1+t} + (1+t)^2 - (1+t)^3$$

[22]:

```python
import numpy as np
np.random.seed(10)
import matplotlib.pyplot as plt

# helper functions
def a(t, x):
    return (2*x)/(1+t) + (1+t)**2
def b(t):
    return (1+t)**2
def b_prime(t):
    return 2*(1+t)
def a_strat(t, x):
    return a(t, x) - 0.5 * b(t) * b_prime(t)

# coefficients for computing multiple stochastic integral
def chang_onestep(tn, Yn, dt, p=15):
    # pre-generate all Gaussian r.v.s
    zeta1 = np.random.randn()
    xi1r = [np.random.randn() for _ in np.arange(1, p+1)]
    eta1r = [np.random.randn() for _ in np.arange(1, p+1)]
    mu1p = np.random.randn()
    phi1p = np.random.randn()
    # a_10
```

```
    r_inv = 1/np.arange(1, p+1)
    r2_inv = 1/(np.arange(1, p+1)**2)
    rhop = (1/12)-(1/(2*(np.pi**2)))*np.sum(r2_inv)
    a_10 = (-1/np.pi)*np.sqrt(2*dt)*np.sum(r_inv*xi1r)-2*np.sqrt(dt*rhop)*mu1p
    # b1
    r4_inv = 1/(np.arange(1, p+1)**4)
    alphap = (np.pi**2)/180 - (0.5/(np.pi**2))*np.sum(r4_inv)
    b_1 = np.sqrt(dt/2)*np.sum(r2_inv*eta1r)+np.sqrt(dt*alphap)*phi1p
    # C_p_11
    xi1l = [np.random.randn() for _ in np.arange(1, p+1)]
    eta1l = [np.random.randn() for _ in np.arange(1, p+1)]
    C_p_11 = 0
    for i in range(p):
        r = i + 1 # 1, ..., p
        for j in range(p):
            l = j + 1 # 1, ..., p
            if r != l:
                C_p_11 = C_p_11 + (r/(r**2-l**2))*((1/l)*xi1r[i]*xi1l[j]-(l/r)*eta1r[i]*eta1l[j])
    J_110p = (1/(3*2*1))*(dt**2)*(zeta1**2)+(1/4)*dt*(a_10**2)-(1/(2*np.pi))*(dt**(3/2))*zeta1*b_1 + \
                (1/4)*(dt**(3/2))*a_10*zeta1 - (dt**2)*C_p_11
    # compute Brownian increments
    dW = np.sqrt(dt)*zeta1
    dZ = 0.5*dt*(np.sqrt(dt)*zeta1+a_10)
    # take one step
    Ybar_plus = Yn+0.5*a_strat(tn, Yn)*dt+(1/dt)*b(tn)*(dZ+np.sqrt(np.abs(2*J_110p*dt-(dZ**2))))
    Ybar_minus = Yn+0.5*a_strat(tn, Yn)*dt+(1/dt)*b(tn)*(dZ-np.sqrt(np.abs(2*J_110p*dt-(dZ**2))))
    Ynp1 = Yn + 0.5*(a(tn+0.5*dt, Ybar_plus) + a(tn+0.5*dt, Ybar_minus))*dt \
        + b(tn)*dW + (1/dt)*(b(tn+dt)-b(tn))*(dW*dt-dZ)
    return Ynp1
def chang_scheme(Y0, tgrid):
    dt = tgrid[1]-tgrid[0]
    Nt = len(tgrid)
    all_sol = np.zeros(Nt)
    all_sol[0] = Y0
    Yn = Y0
    for i in range(Nt-1):
        tn = tgrid[i]
        Yn = chang_onestep(tn, Yn, dt)
        all_sol[i+1] = Yn
    return all_sol

def chang_final_solution(Y0, tgrid):
    return chang_scheme(Y0, tgrid)[-1]
```

[23]:
```
# number of batches
M = 20
# number of samples for each batch
N = 100
# all step sizes
all_dt = 2**np.array([-1., -2., -3., -4.])
# final time
T = 0.5
# initial condition
Y0 = 1
# store all solutions (only final solutions)
all_paths = np.zeros([M, N, len(all_dt)])
for i in range(len(all_dt)):
    dt = all_dt[i]
    tgrid = np.arange(0, T+dt, dt)
    Y0 = 1
    for idx1 in range(M):
        for idx2 in range(N):
            all_paths[idx1, idx2, i] = chang_final_solution(Y0, tgrid)
```

As a comparison, we simulate the exact solution using the above timesteps.

$$X_t = (1+t)^2(1+W_t+t)$$

for $t = T = 0.5$:

$$X_T = 2.25(W_T + 1.5)$$

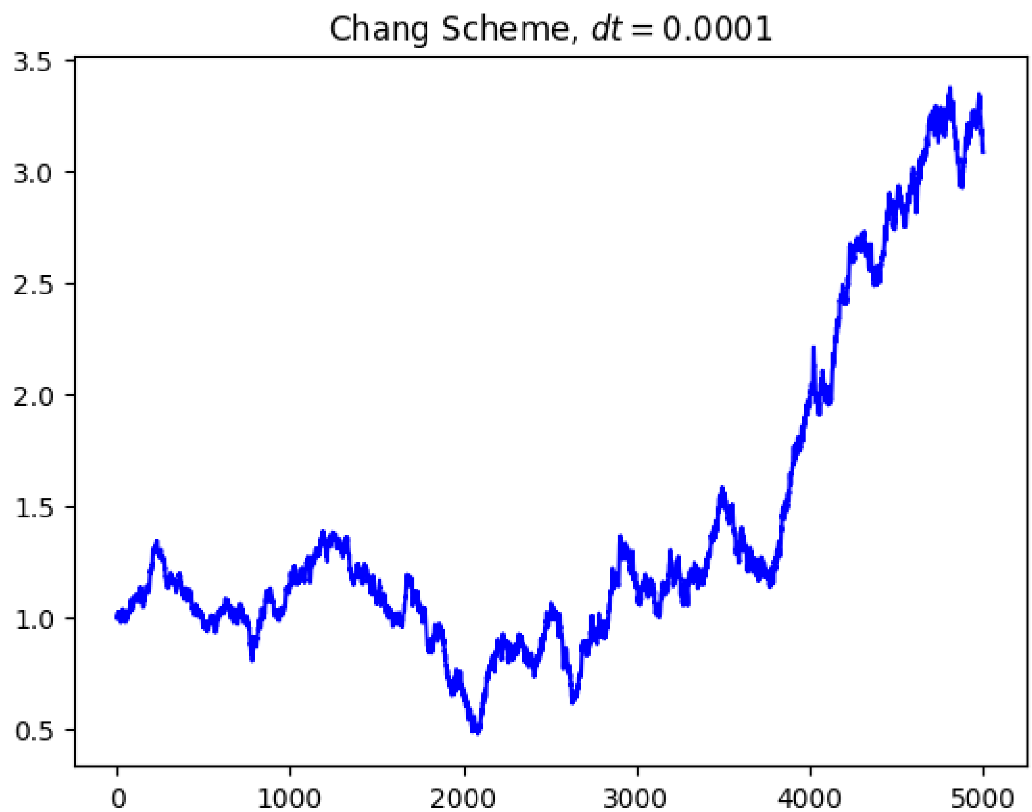
where $W_T \sim \mathcal{N}(0, 0.5)$.

[24]:
```
def exact_solution(Y0, tgrid):
    dt = tgrid[1]-tgrid[0]
    Nt = len(tgrid)
    # simulate Brownian motion
    dWt = np.sqrt(dt)*np.random.normal(loc=0, scale=1, size=Nt)
    Wt = np.cumsum(dWt)
    return (((1+tgrid)**2)*(1+Wt+tgrid))
def exact_final_solution(Y0, tgrid):
    return exact_solution(Y0, tgrid)[-1]

# number of batches
M = 20
# number of samples for each batch
N = 100
# all step sizes
all_dt = 2**np.array([-1., -2., -3., -4.])
# final time
T = 0.5
# initial condition
Y0 = 1
```

```
# store all solutions (only final solutions)
all_paths_exact = np.zeros([M, N, len(all_dt)])
for i in range(len(all_dt)):
    dt = all_dt[i]
    tgrid = np.arange(0, T+dt, dt)
    Y0 = 1
    for idx1 in range(M):
        for idx2 in range(N):
            all_paths_exact[idx1, idx2, i] = exact_final_solution(Y0, tgrid)
```

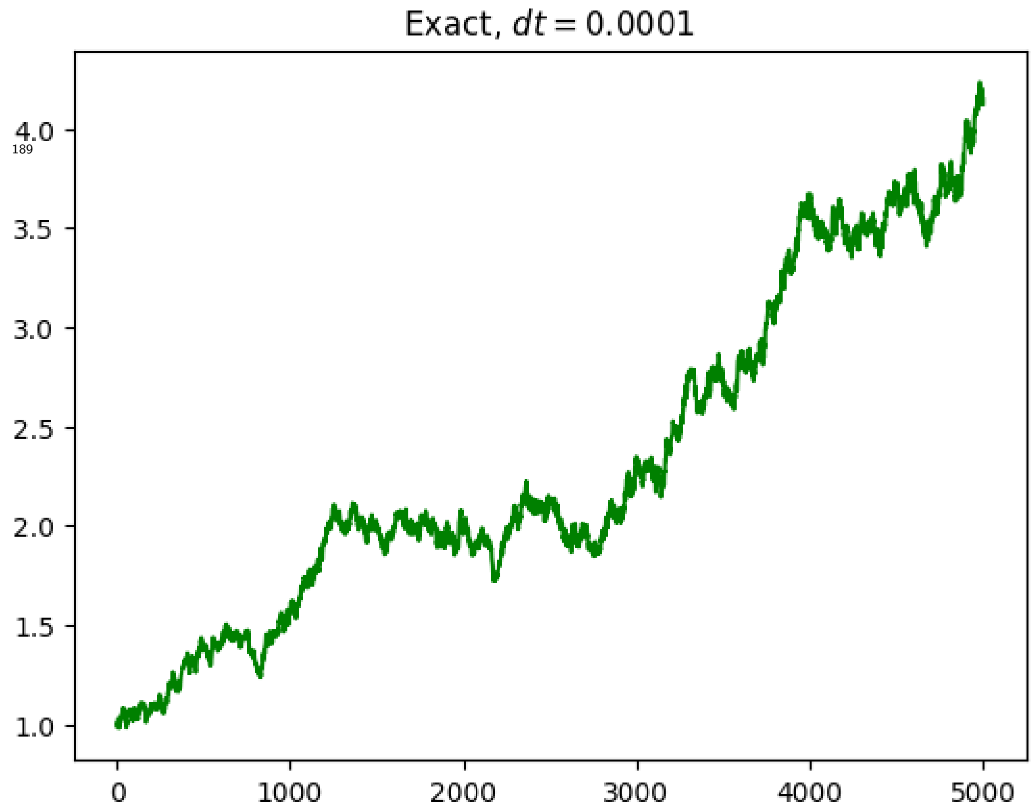
[63]:
```
test = chang_scheme(1, np.arange(0, 0.5+0.0001, 0.0001))
```

[64]:
```
plt.plot(test, color='blue');
plt.title(r"Chang Scheme, $dt=0.0001$");
```



[69]:
```
test2 = exact_solution(1, np.arange(0, 0.5+0.0001, 0.0001))
```

[70]:
```
plt.plot(test2, color='green');
plt.title(r"Exact, $dt=0.0001$");
```


Exact, $dt = 0.0001$
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0
1000
2000
3000
4000
5000

6 Final (Partial)

6.1 Multiple Brownian Motions

Consider 1D SDE driven by two independent Wiener processes:

$$dX_t = X_t dt + X_t dW_t^1 + X_t dW_t^2, X_0 = 1 \tag{2}$$

6.2 Strong solution

We may consider thie as a generalization of Lecture 4, part 4 for general linear SDEs. We first change the two stochastic differentials into Stratonovich form:

$$\begin{aligned} dX_t &= (X_t - 2 \cdot \frac{1}{2} X_t)dt + X_t \circ dW_t^1 + X_t \circ dW_t^2 \qquad (3) \\ &= X_t \circ dW_t^1 + X_t \circ dW_t^2 \end{aligned}$$

Then following (4.13)-(4.16),

$$\Phi_{0,t} = \exp\left(\int_0^t dW_s^1 + \int_0^t dW_s^2\right) = \exp\left(W_t^1 + W_t^2\right)$$

Since $X_0 = 1$, we have:

$$X_t = \exp\left(W_t^1 + W_t^2\right) \tag{4}$$

6.3 Integral form and Expectation

The integral form reads:

$$X_t = 1 + \int_0^t X_s ds + \int_0^t X_s dW_s^1 + \int_0^t X_s dW_s^2$$

To compute $\mathbb{E}\big[X_T\big]$, since W_t^1, W_t^2 are independent Wiener processes, W_T^1, W_T^2 are i.i.d. $\mathcal{N}(0, T)$ random variables. Then:

$$\mathbb{E}\big[X_T\big] = \mathbb{E}\big[\exp\big(W_T^1 + W_T^2\big)\big] = \mathbb{E}\big[\exp\big(W_T^1\big)\big] \cdot \mathbb{E}\big[\exp\big(W_T^2\big)\big] \tag{5}$$

Since W_T^1, W_T^2 are normal random variables, $\exp\big(W_T^1\big), \exp\big(W_T^2\big)$ are log-normally distributed. Therefore we conclude the expectation is the following (using the formula derived from the log-normal density):

$$\mathbb{E}\big[X_T\big] = (e^{\frac{1}{2}T})^2 = e^T \tag{6}$$