OrnsteinUhlenbeck1D with output

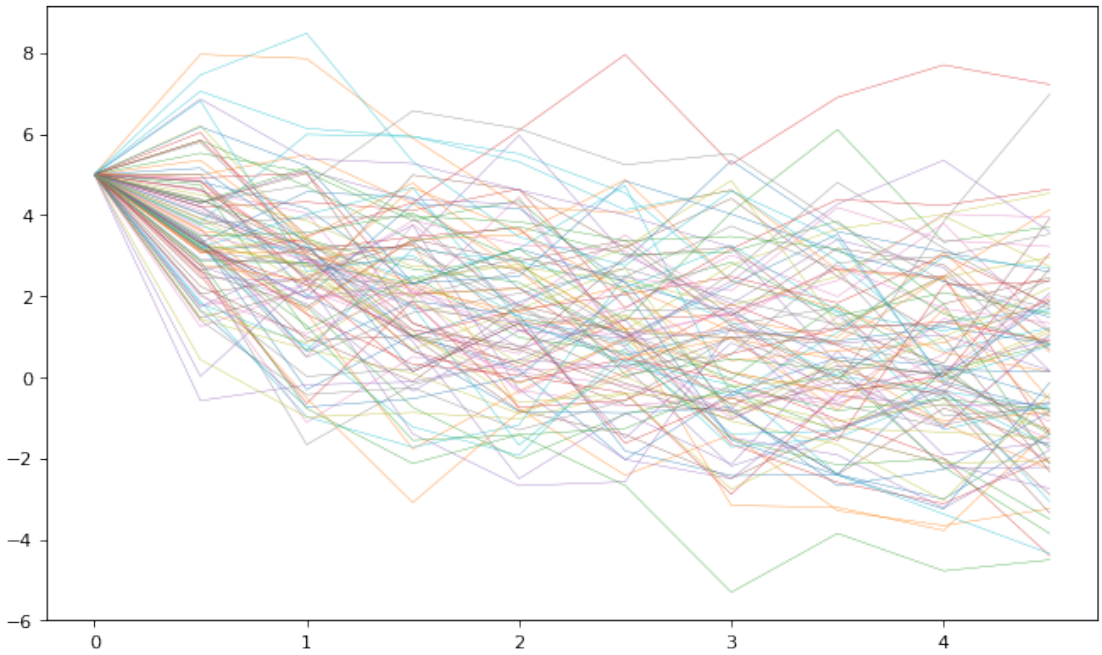
October 22, 2018

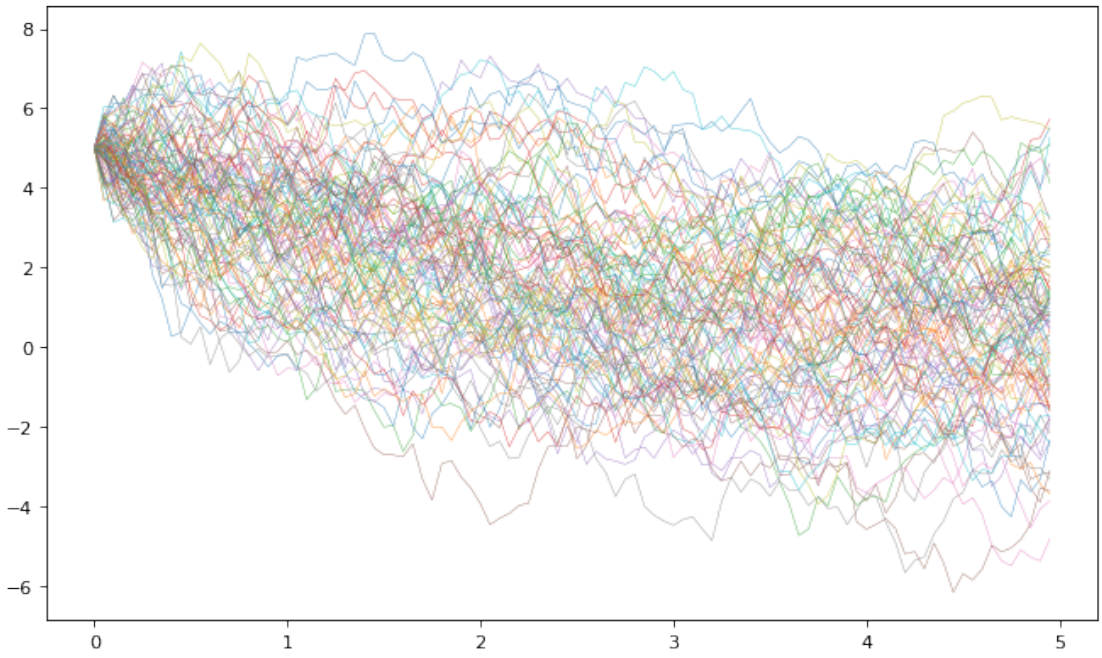
1 SDE approximation by Euler scheme

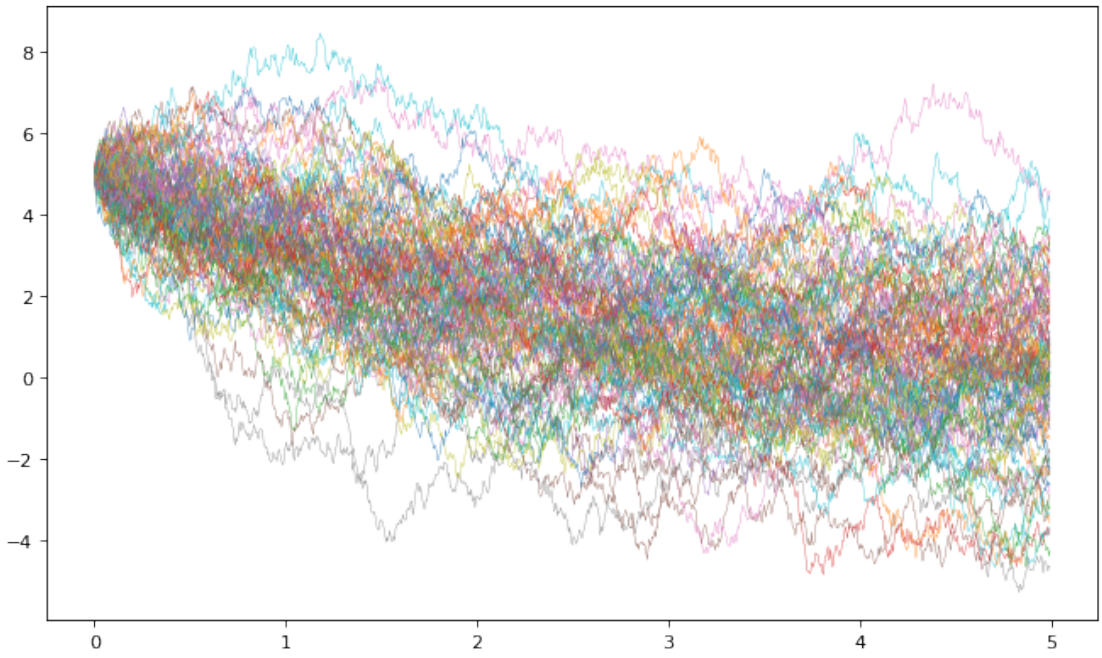
We simulate a one dimensional Ornstein-Uhlenbeck process by using an Euler approximation of the solution to the corresponding stochastic differential equation.

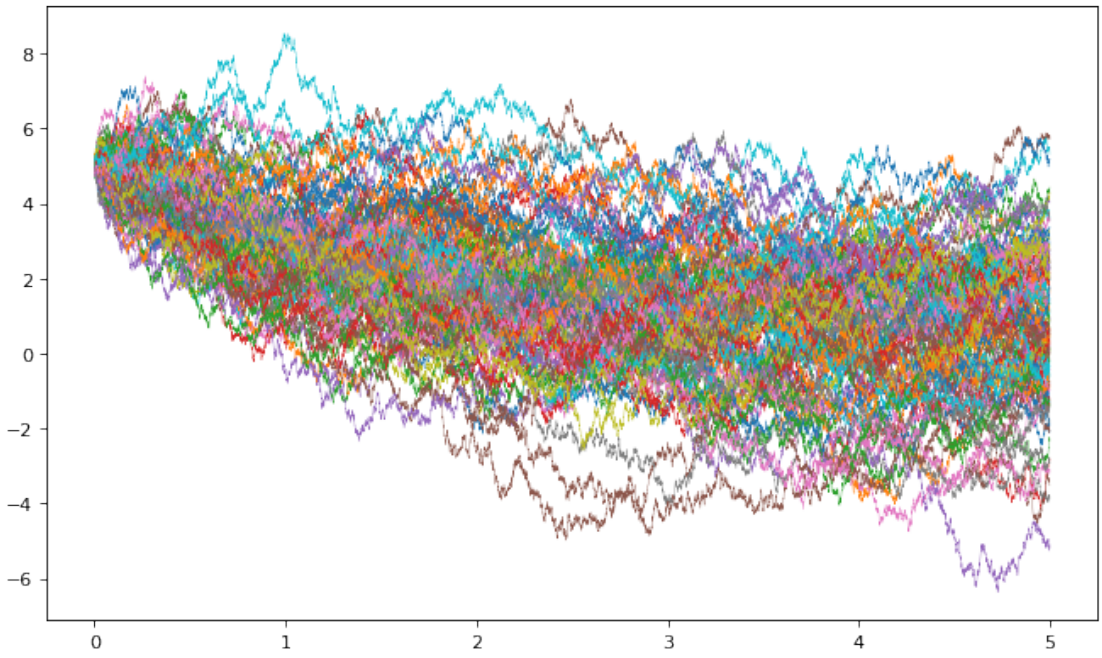
In [9]:
```
# Euler-Maryuama Approximation for solution of SDE  dX_t = -gamma X_t dt + sigma dB_t

import numpy as np
# makes numpy routines and data types available as np.[name ouf routine or data type]

import matplotlib.pyplot as plt
# makes plotting command available as plt.[name of command]

x0 = 5.
# initial value at time t=0
sigma = 2.
# volatility
gamma = .5
# friction coefficient

tmax = 5.
# simulation of process from time 0 to tmax
stepslist = [10,100,1000,10000]
# produce simulations with step numbers chosen from steplist

for steps in stepslist:
    h = tmax/steps
    # stepsize for time discretization
    std = np.sqrt(h)
    # standard deviation for the distribution of each step

    k = 100
    # number of samples that will be simulated

    noise = np.random.randn(steps,k)*std
    # create a steps times k dimensional matrix of normal random numbers
    # with variance h
```

```
sde = np.ones((steps,k))
sde = sde*x0
# initialize the array sde with the initial condition x0

for n in range(steps-1):
    sde[n+1] = sde[n]-gamma*h*sde[n]+sigma*noise[n]

t = np.arange(0,steps,1)*h
# creates vector of time points

plt.figure(figsize=(10,6), dpi=80)
# sets size of plot
plt.plot(t,sde,linewidth=0.3)
# produces a plot of the sample solutions in the components of sde
# versus t with thin lines
plt.show()
# output of plot
```




8
6
4
2
0
−2
−4
−6
0
1
2
3
4
5


8
6
4
2
0
−2
−4
0
1
2
3
4
5



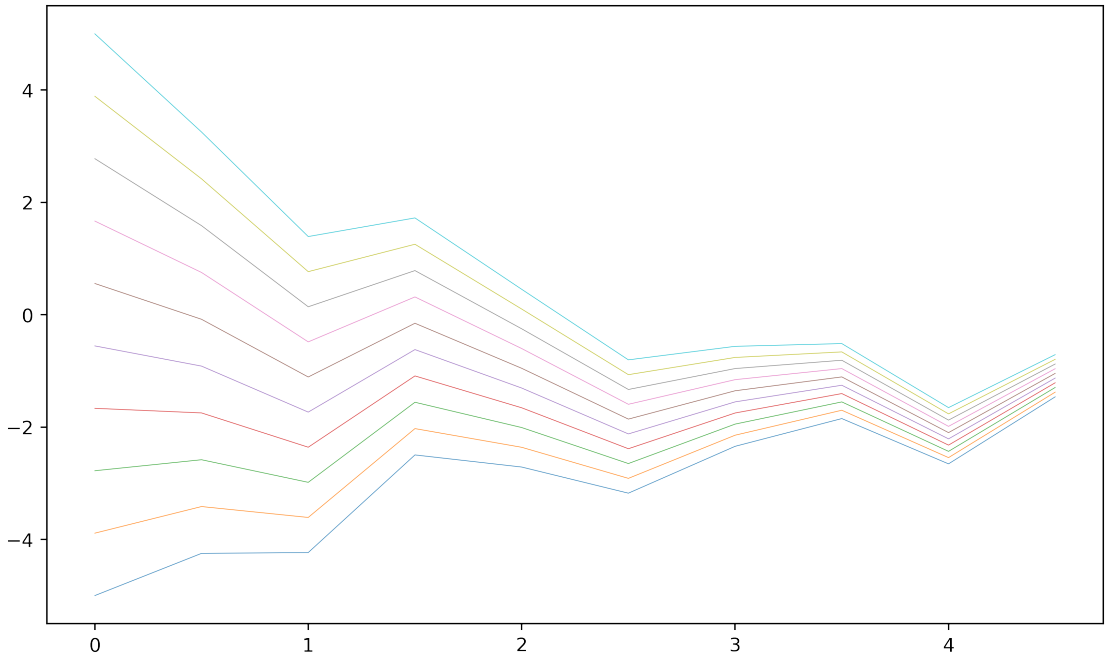
After some time the process seems to approach an equilibrium distribution.

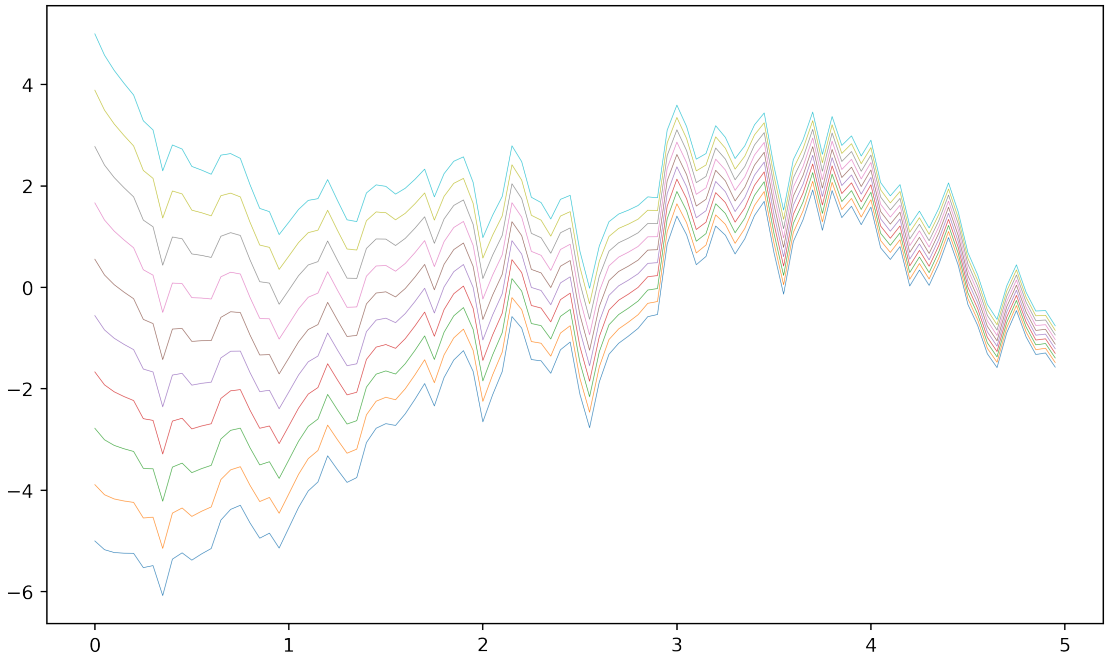
2 Solution flow of Ornstein-Uhlenbeck SDE

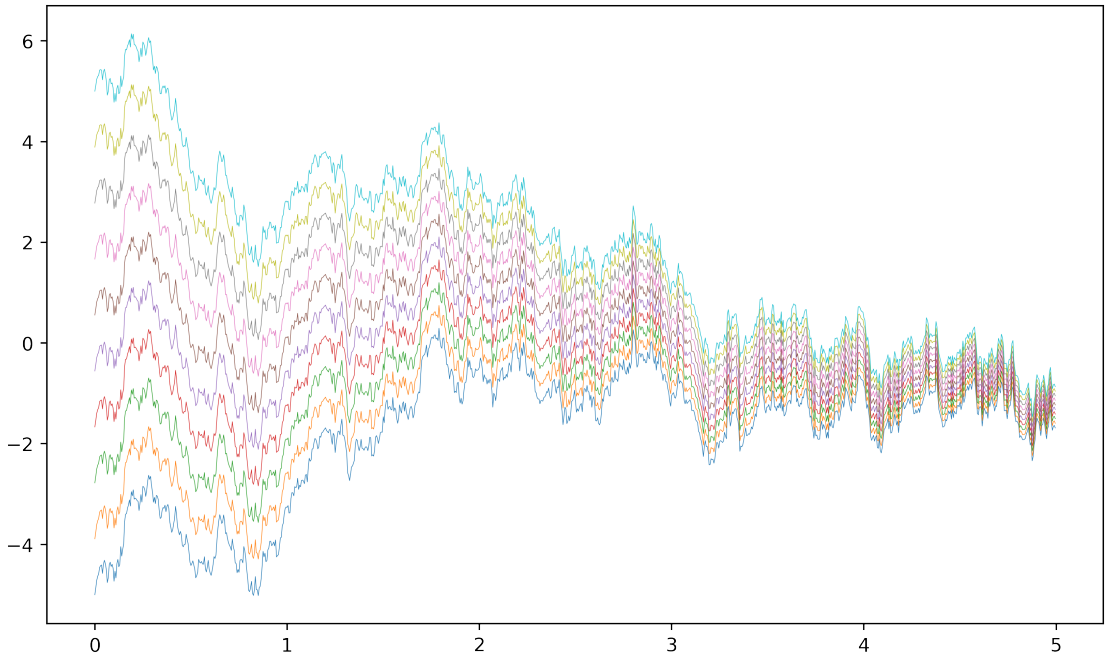
We now simulate simultaneously the solutions starting from different initial conditions where the same noise random variables are applied in each case.

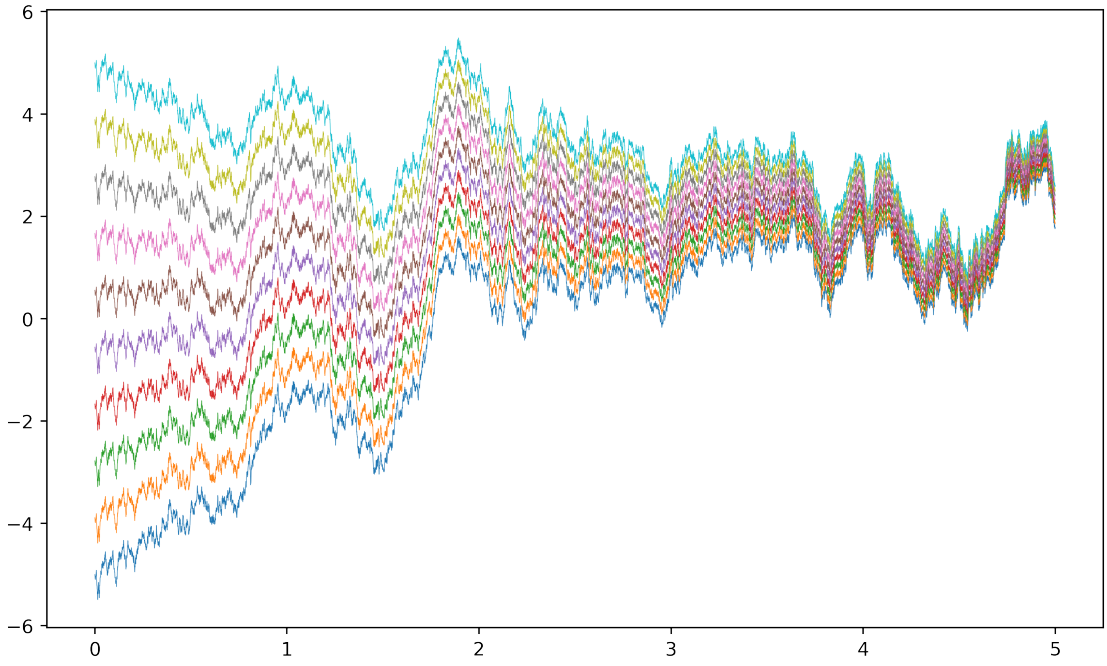
In [7]:
```
# Euler approximation for flow of SDE  dX_t = -gamma X_t dt + sigma dB_t

import numpy as np
# makes numpy routines and data types available as np.[name ouf routine or data type]

import matplotlib.pyplot as plt
# makes plotting command available as plt.[name of command]

sigma = 2.
# volatility
gamma = .5
# friction coefficient

tmax = 5.
# simulation of process from time 0 to tmax
stepslist = [10,100,1000,10000]
# produce simulations with step numbers chosen from steplist

for steps in stepslist:
```

```
h = tmax/steps
# stepsize for time discretization
std = np.sqrt(h)
# standard deviation for the distribution of each step

k = 10
# number of initial values to be considered
x0 = np.linspace(-5.,5.,k)
# simulate solutions for this list of initial values

noise = np.random.randn(steps)*std
# create a steps dimensional vector of normal random numbers with variance h


sde = np.zeros((steps,k))
sde[0] = x0
# initialize the array sde with the initial conditions in x0

for n in range(steps-1):
    sde[n+1] = sde[n]-gamma*h*sde[n]+sigma*noise[n]

t = np.arange(0,steps,1)*h
# creates vector of time points

plt.figure(figsize=(10,6), dpi=400)
# sets size of plot
plt.plot(t,sde,linewidth=0.3)
# produces a plot of the sample solutions in the components of sde
# versus t with thin lines
plt.show()
# output of plot
```


4
2
0
−2
−4
0
1
2
3
4


4
2
0
−2
−4
−6
0
1
2
3
4
5





We observe that the solutions for different initial values approach each other ("loss of memory of initial condition").

In [ ]: