Math Capstone Project

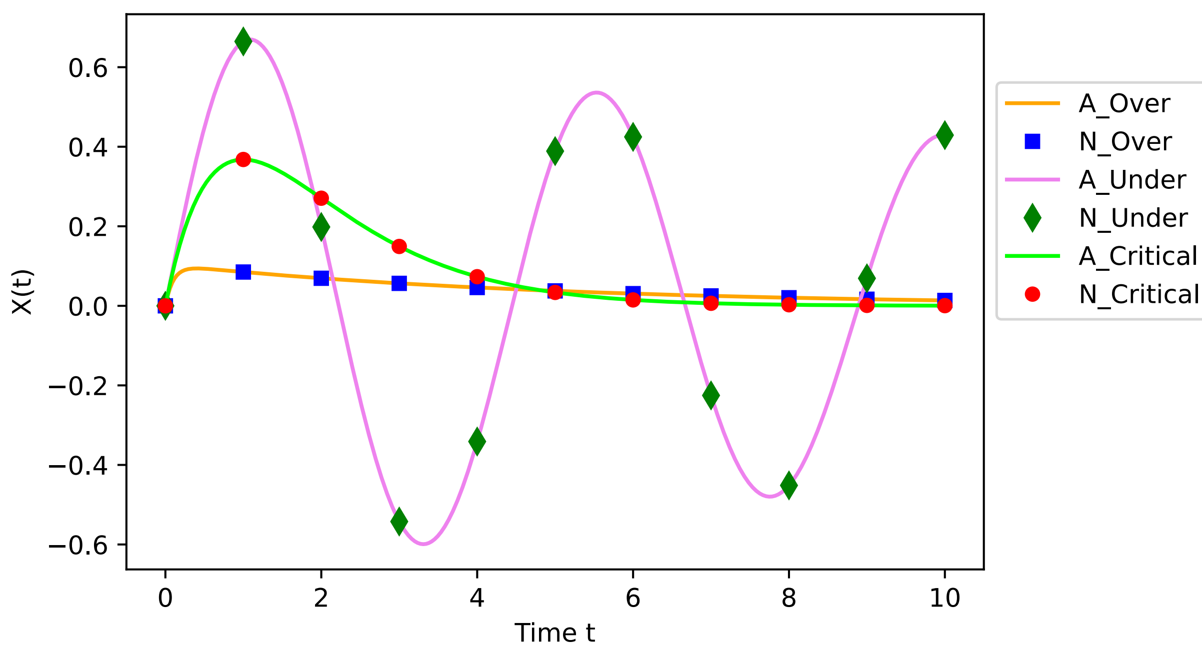
Author: Ho Yi Alexis HO

In [1]:
```
# $X(0) = 0$, $p(0) = 100$, $ h = 0.1$'

import matplotlib.pyplot as plt
# Ouput images with higher resolutions
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
from matplotlib.offsetbox import AnchoredText
from scipy.integrate import odeint
import numpy as np

# Initialized the parameters
h = 0.001 # time step used
interval_len = 10 # the length of the interval
# total num of observed time stamps N = interval_len / h
# And I will chop off the decimal places after N in cases N is a non-integer

m = 1 # mass
K = 2 # spring constant
gamma = 10 # frictional coefficient

# x_n := x(t_n)
# dx_n := the first derivative of x(t_(n-1))
x_0 = 0 # initial position of the particle
p_0 = 1 # initial velocity of the particle

# initialize the values for iterations
x_n = x_0
p_n = p_0

N_ls = [0] # The i list, (which are the index of ti, the observed timestamp)
x_n_ls = [x_n] # contains the trajectory of X at different time ti, i.e. X(t)

# N = truncate_to_int(interval_len/h)
for N in range(1, int(interval_len/h)+1):
    x_n = x_n + p_n*h
    p_n = p_n + h*(-gamma*p_n - K*x_n)/m
    N_ls.append(N)
    x_n_ls.append(x_n)

# change the plot's labels in i (the ith observation) to ti (observed timestamps)
time_axis = [i * h for i in N_ls]

## Notice: for the illustration purpose points are plotted with stepsize = 1000 ##
## to contrast its discreteness with the continuous line ##
# list[start:stop:step]

### We will plot the analytical ODE sol in a continous line to represent the ground tr
# odeint() can define more than one 1st order differential equations
# And we can solve even higher order ODEs by using multiple 1st order ODEs.
def derivatives(initial_values, time_interval):
    x_0 = initial_values[0]
    dxdt_0 = initial_values[1]
```

```
    sec_dxdt_0 = -gamma/m*dxdt_0-K/m*x_0
    return(dxdt_0, sec_dxdt_0)
initial_values = [x_0, p_0]
time_interval = np.linspace(0, interval_len, N)
ODE_sol = odeint(derivatives, initial_values, time_interval)
ODE_sol = ODE_sol[:,0]

plt.plot(time_interval,ODE_sol, linestyle ='-', color='orange', label='A_Over' )
step = 1000 # set the step size for ploting (constrast the discrete points with the cc
plt.plot(time_axis[0:len(time_axis):step], x_n_ls[0:len(time_axis):step],
         color='blue', marker = "s", linestyle='None', markersize = 5, label='N_Over')

plt.xlabel("Time t")
plt.ylabel("X(t)")
plt.legend()
plt.savefig(r'C:\Users\alexi\Desktop\Plots\Overdamped.svg')

# Reference:
# https://blog.csdn.net/weixin_42376039/article/details/86485817
# https://www.epythonguru.com/2020/07/second-order-differential-equation.html
# https://apmonitor.com/pdc/index.php/Main/SolveDifferentialEquations

import matplotlib.pyplot as plt
# Ouput images with higher resolutions
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
from matplotlib.offsetbox import AnchoredText
from scipy.integrate import odeint
import numpy as np

# Initialized the parameters
h = 0.001 # time step used
interval_len = 10 # the length of the interval
# total num of observed time stamps N = interval_len / h
# And I will chop off the decimal places after N in cases N is a non-integer

m = 10 # mass
K = 20 # spring constant
gamma = 1 # frictional coefficient

# x_n := x(t_n)
# dx_n := the first derivative of x(t_(n-1))
x_0 = 0 # initial position of the particle
p_0 = 1 # initial velocity of the particle

# initialize the values for iterations
x_n = x_0
p_n = p_0

N_ls = [0] # The i list, (which are the index of ti, the observed timestamp)
x_n_ls = [x_n] # contains the trajectory of X at different time ti, i.e. X(t)

# N = truncate_to_int(interval_len/h)
for N in range(1, int(interval_len/h)+1):
    x_n = x_n + p_n*h
    p_n = p_n + h*(-gamma*p_n - K*x_n)/m
    N_ls.append(N)
    x_n_ls.append(x_n)

# change the plot's labels in i (the ith observation) to ti (observed timestamps)
```

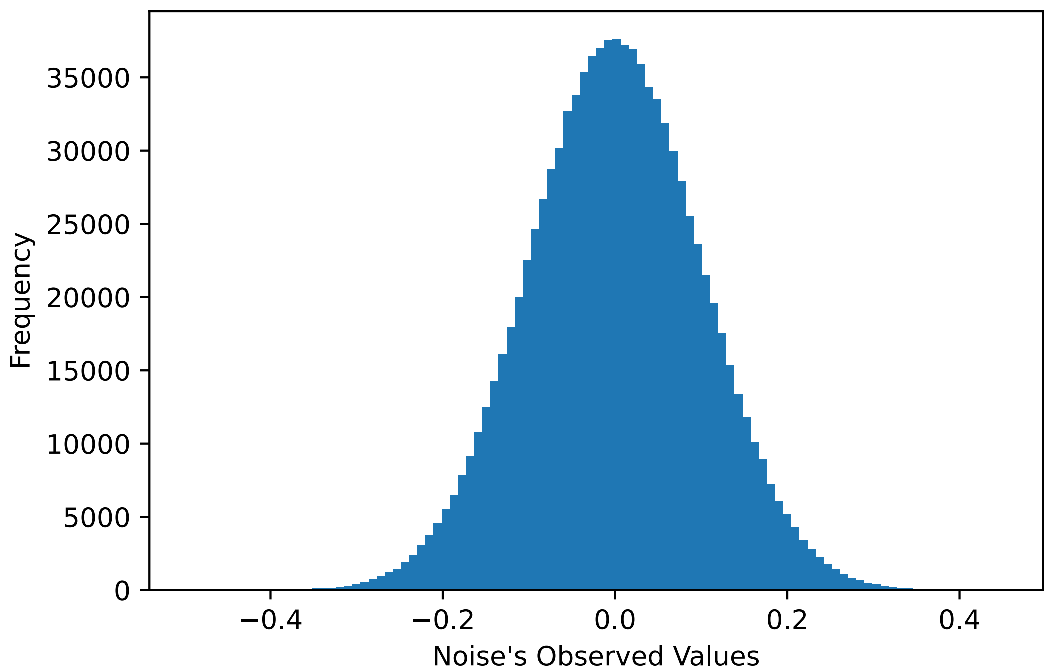
```
time_axis = [i * h for i in N_ls]

### We will plot the analytical ODE sol in a continous line to represent the ground tr
# odeint() can define more than one 1st order differential equations
# And we can solve even higher order ODEs by using multiple 1st order ODEs.
def derivatives(initial_values, time_interval):
    x_0 = initial_values[0]
    dxdt_0 = initial_values[1]
    sec_dxdt_0 = -gamma/m*dxdt_0-K/m*x_0
    return(dxdt_0, sec_dxdt_0)
initial_values = [x_0, p_0]
time_interval = np.linspace(0, interval_len, N)
ODE_sol = odeint(derivatives, initial_values, time_interval)
ODE_sol = ODE_sol[:,0]

plt.plot(time_interval,ODE_sol, linestyle ='-', color='violet', label='A_Under' )
# list[start:stop:step]
step = 1000 # set the step size for ploting (constrast the discrete points with the co
plt.plot(time_axis[0:len(time_axis):step], x_n_ls[0:len(time_axis):step],
         color='green', marker = "d", linestyle='None', markersize = 7, label='N_Under

plt.xlabel("Time t")
plt.ylabel("X(t)")
plt.legend()
plt.savefig(r'C:\Users\alexi\Desktop\Plots\Underdamped.svg')

# Reference:
# https://blog.csdn.net/weixin_42376039/article/details/86485817
# https://www.epythonguru.com/2020/07/second-order-differential-equation.html
# https://apmonitor.com/pdc/index.php/Main/SolveDifferentialEquations

import matplotlib.pyplot as plt
# Ouput images with higher resolutions
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')
from matplotlib.offsetbox import AnchoredText
from scipy.integrate import odeint
import numpy as np

# Initialized the parameters
h = 0.001 # time step used
interval_len = 10 # the length of the interval
# total num of observed time stamps N = interval_len / h
# And I will chop off the decimal places after N in cases N is a non-integer

m = 1 # mass
K = 1 # spring constant
gamma = 2 # frictional coefficient

# x_n := x(t_n)
# dx_n := the first derivative of x(t_(n-1))
x_0 = 0 # initial position of the particle
p_0 = 1 # initial velocity of the particle

# initialize the values for iterations
x_n = x_0
p_n = p_0

N_ls = [0] # The i list, (which are the index of ti, the observed timestamp)
x_n_ls = [x_n] # contains the trajectory of X at different time ti, i.e. X(t)
```

```python
# N = truncate_to_int(interval_len/h)
for N in range(1, int(interval_len/h)+1):
    x_n = x_n + p_n*h
    p_n = p_n + h*(-gamma*p_n - K*x_n)/m
    N_ls.append(N)
    x_n_ls.append(x_n)

# change the plot's labels in i (the ith observation) to ti (observed timestamps)
time_axis = [i * h for i in N_ls]

### We will plot the analytical ODE sol in a continous line to represent the ground tr
# odeint() can define more than one 1st order differential equations
# And we can solve even higher order ODEs by using multiple 1st order ODEs.
def derivatives(initial_values, time_interval):
    x_0 = initial_values[0]
    dxdt_0 = initial_values[1]
    sec_dxdt_0 = -gamma/m*dxdt_0-K/m*x_0
    return(dxdt_0, sec_dxdt_0)
initial_values = [x_0, p_0]
time_interval = np.linspace(0, interval_len, N)
ODE_sol = odeint(derivatives, initial_values, time_interval)
ODE_sol = ODE_sol[:,0]

plt.plot(time_interval,ODE_sol, linestyle ='-', color='lime', label='A_Critical' )
# list[start:stop:step]
step = 1000 # set the step size for ploting (constrast the discrete points with the co
plt.plot(time_axis[0:len(time_axis):step], x_n_ls[0:len(time_axis):step],
         color='red', marker = ".", linestyle='None', markersize = 10, label='N_Critic
plt.xlabel("Time t")
plt.ylabel("X(t)")
plt.legend(bbox_to_anchor=(1, 0.9), loc='upper left', ncol=1)
viewBox="-322 591 150 44"
plt.savefig(r'C:\Users\alexi\Desktop\Plots\3_Cases.svg', bbox_inches='tight')
```



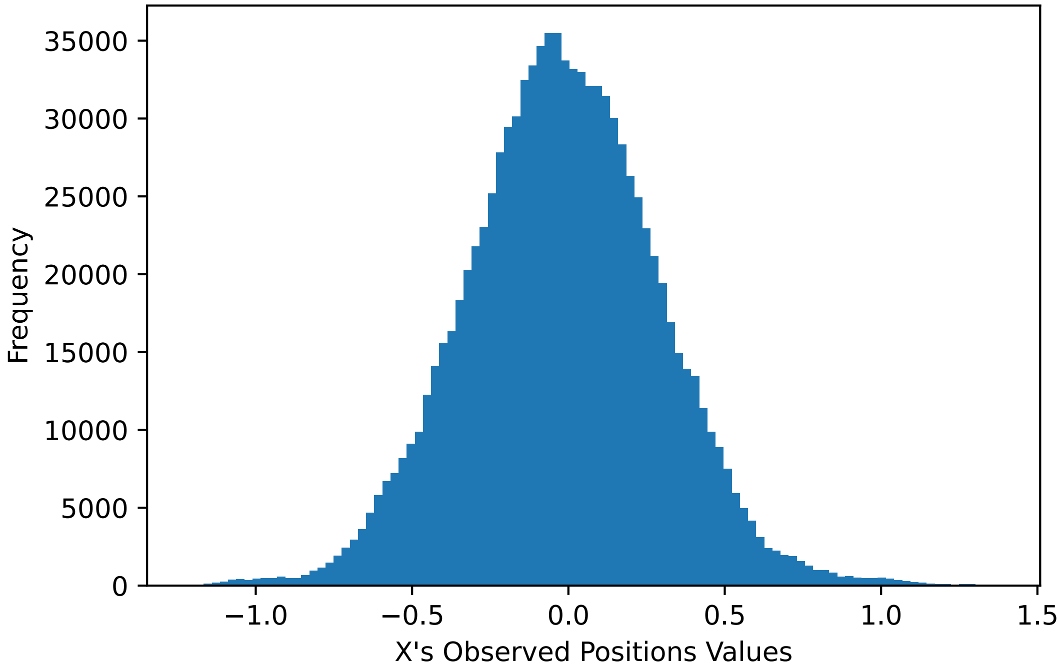
In [4]:
```python
x_0 = 0
# Initialized the parameters
p_0 = 100 # initial velocity of the particle
```

```python
h = 0.001 # time step
interval_len = 1000 # the length of the interval

# mass = 0 and can be obmitted
K = 5 # spring constant
gamma = 10 # frictional coefficient

x_n = x_0
p_n = p_0

N_ls = [0]
x_n_ls = [x_0]

# N = truncate_to_int(interval_len/h)
for N in range(1, int(interval_len/h)+1):
    x_n = x_n + p_n*h
    p_n = p_n + h*(-gamma*p_n - K*x_n)/m
    N_ls.append(N)
    x_n_ls.append(x_n)

mu, sigma = 0, 0.1 # mean and standard deviation
noise_sample = np.random.normal(mu, sigma, N)
plt.hist(noise_sample, bins = 100)
plt.xlabel("Noise's Observed Values")
plt.ylabel("Frequency")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\Noise.svg')
```



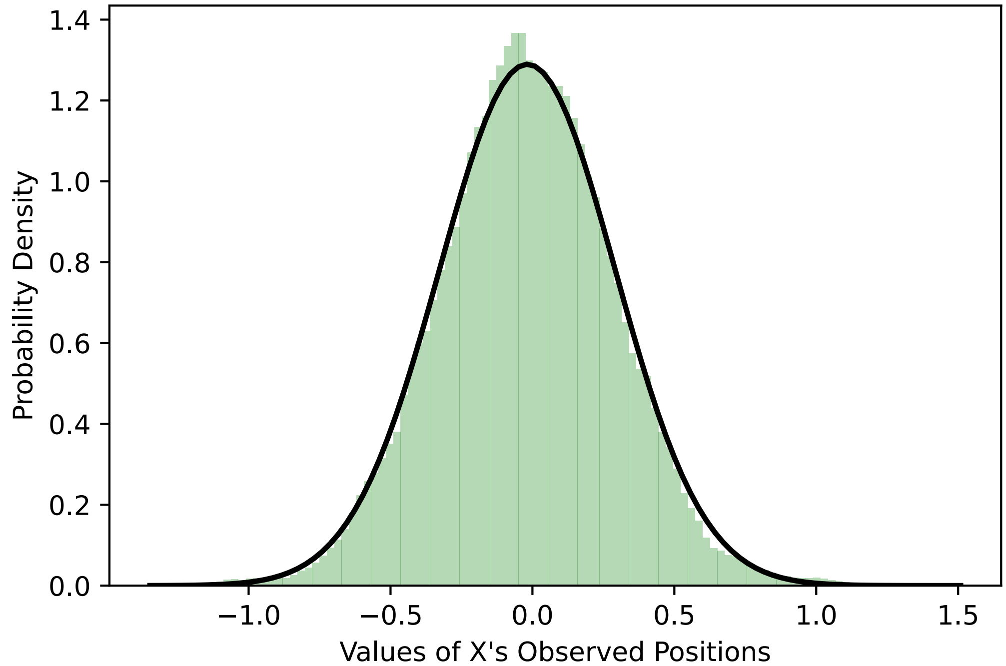
In [5]:

```python
x_noise_i = x_0
x_noise_ls = [x_noise_i]
for i in range(1, int(interval_len/h)+1):
    x_noise_i = x_noise_i - K/gamma*x_noise_i*h + noise_sample[i-1]/gamma
    x_noise_ls.append(x_noise_i)
plt.hist(x_noise_ls, bins = 100)
plt.xlabel("X's Observed Positions Values")
plt.ylabel("Frequency")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\X_with_noise.svg')
```

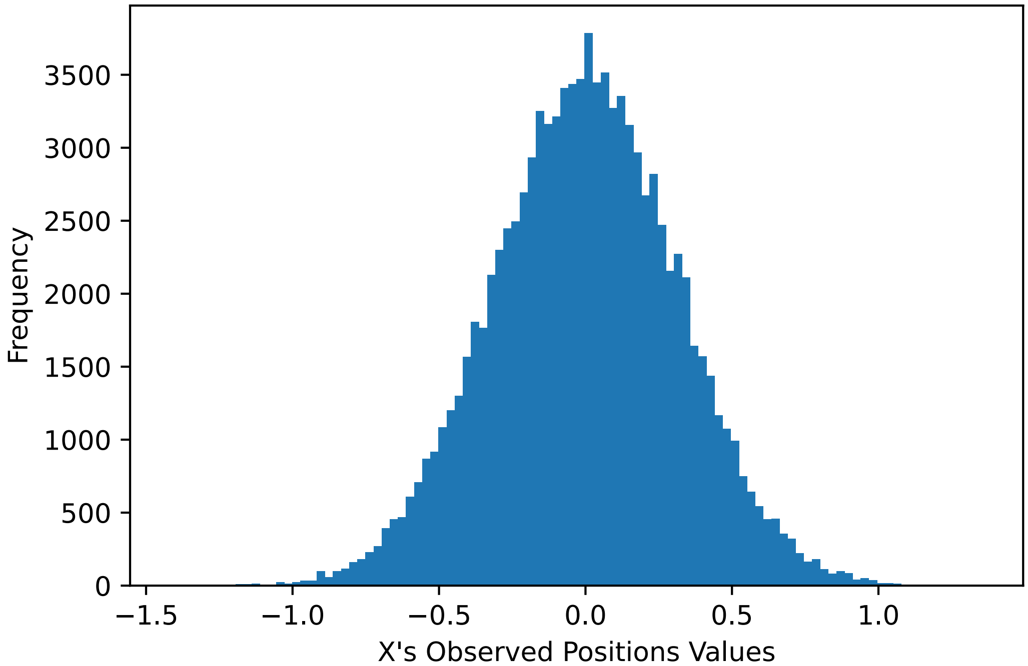


In [6]:
```
x0 = 0
# Reference:
# https://stackoverflow.com/questions/20011122/fitting-a-normal-distribution-to-1d-dat
import numpy as np
from scipy.stats import norm

## Find the best fitted normal distribution to X(t) ##
mu, std = norm.fit(x_noise_ls) # x_noise_ls: contains the trajectory of X(t)

## Plot the histogram of X(t) ##
# density: transfer the frequence on y-axis into probability density
# alpha: constrols the histogram's transparency
plt.hist(x_noise_ls, bins = 100, density=True, alpha=0.3, color='green')
xmin, xmax = plt.xlim() # set the boundary for x_axis
# larger N is, the higher the resolutions of (smoother) the fitted line of the PDF val
selected_pts_on_x_axis = np.linspace(xmin, xmax, 100) # 100 is the number of equally s
corresponding_pdf_values = norm.pdf(selected_pts_on_x_axis , mu, std)
plt.plot(selected_pts_on_x_axis, corresponding_pdf_values, 'k', linewidth=2) # k:= (co
# title = "Estimated Paramters: $\hat{\mu}$ = %.2f, $\hat{\sigma}$ = %.2f" % (mu, std)
plt.xlabel("Values of X's Observed Positions")
plt.ylabel("Probability Density")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\Bestfit.svg')
mu, std
```

Out[6]:
```
(-0.01803108724688858, 0.3093262955469327)
```

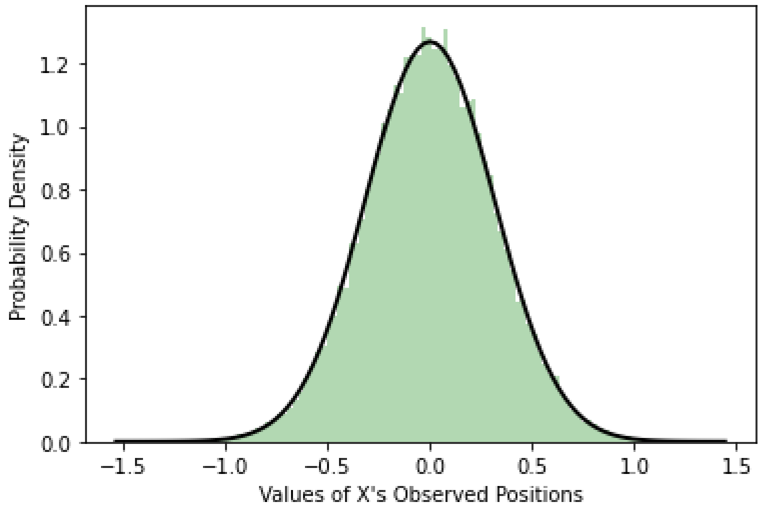


In [7]:
```
import matplotlib.pyplot as plt
import random
import math
kBT = 0.5
K = 5 # spring constant
E = lambda x: (K*x**2/2) # elastic potential energy
x = 0 # initial position
chain = [x]
for i in range(1, 100000):
    x_c = x + random.uniform(-1, 1) # trial step size
    z = random.uniform(0, 1)
    prob = math.exp(-(E(x_c)-E(x))/kBT)
    if E(x_c) < E(x):
        x = x_c
    elif z <= prob:
        x = x_c
    else:
        x = x
    chain.append(x)
plt.hist(chain, bins = 100)
plt.xlabel("X's Observed Positions Values")
plt.ylabel("Frequency")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\MC_hist.svg')
```



In [26]:
```
import numpy as np
from scipy.stats import norm

## Find the best fitted normal distribution to X(t) ##
mu, std = norm.fit(chain) # chain: contains the trajectory of X(t)

## Plot the histogram of X(t) ##
# density: transfer the frequence on y-axis into probability density
# alpha: constrols the histogram's transparency
plt.hist(chain, bins = 100, density=True, alpha=0.3, color='green')
xmin, xmax = plt.xlim() # set the boundary for x_axis

# larger N is, the higher the resolutions of (smoother) the fitted line of the PDF val
selected_pts_on_x_axis = np.linspace(xmin, xmax, 100) # 100 is the number of equally s
corresponding_pdf_values = norm.pdf(selected_pts_on_x_axis , mu, std)
plt.plot(selected_pts_on_x_axis, corresponding_pdf_values, 'k', linewidth=2) # k:= (cc
# title = "Estimated Paramters: $\hat{\mu}$ = %.2f, $\hat{\sigma}$ = %.2f" % (mu, std)
# plt.title(title)
plt.xlabel("Values of X's Observed Positions")
plt.ylabel("Probability Density")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\MC_Bestfit.svg')
mu, std
```

Out[26]:
```
(-0.0006277811204764891, 0.3142705747302197)
```



In [8]:

```
import numpy as np

gamma = 10 # frictional coefficient
x_0 = 0 # initial position of the particle

h = 0.001 # time step
interval_len = 10 # the length of the interval # t = Nh

# total num of observed time stamps N = interval_len / h
# And I will chop off the decimal places after N in cases N is a non-integer
N = int(interval_len/h) # number of steps
N_ls = [i for i in range(0, N+1)] # The i list, (which are the index of ti, the observ
time_axis = [round(i * h, 3) for i in N_ls] # [h, 2h,... nh=t]

mu, sigma = 0, 0.1 # mean and standard deviation

def form_trajectory():
        noise_sample = np.random.normal(mu, sigma, N)

        x_noise_i = x_0  # initialize the position
        x_noise_ls = [x_noise_i] # contains the trajectory of X at different time ti,

        for i in range(1, int(interval_len/h)+1):
            x_noise_i = x_noise_i + noise_sample[i-1]/gamma
            x_noise_ls.append(x_noise_i)
        return x_noise_ls
```

In [9]:

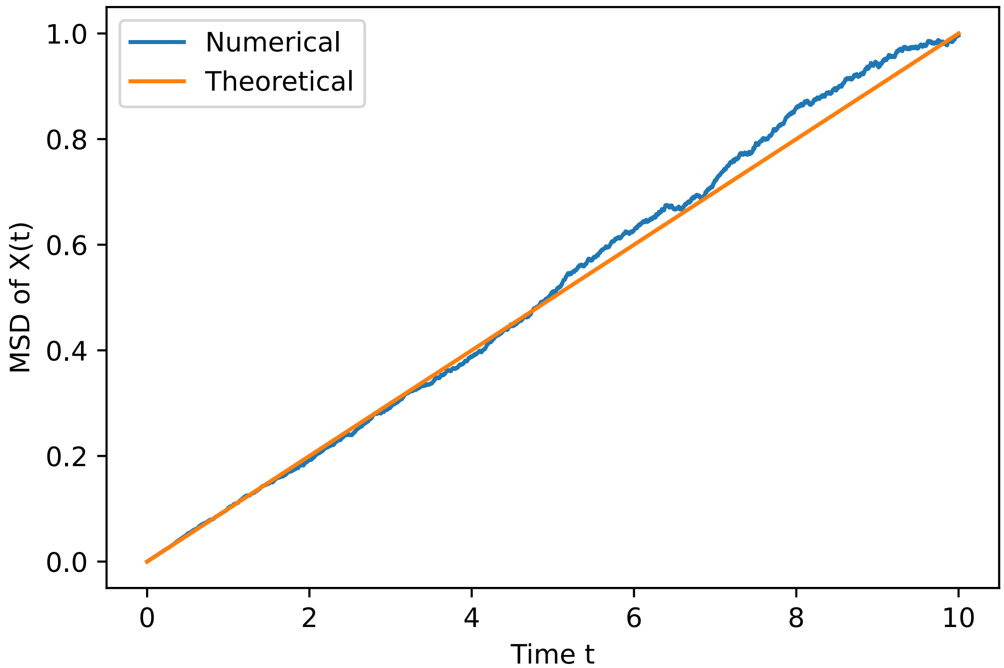
```
import time
start_time = time.time()
from joblib import Parallel, delayed
# ls_Xt is a list of a list resulting from parallel computing
# the inner list represents a trajectory
# the outer list represents an output from the parallel computing
Num_trajectory = 1000 # Number of trajectories to be formed
ls_Xt = Parallel(n_jobs=-2)(delayed(form_trajectory)() for i in range(Num_trajectory))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 14.80647325515747 seconds ---
```

In [10]:

```
# transfer to matrix, where each row stores a trajectory
m = np.array([np.array(row) for row in ls_Xt])
```

In [12]:
```
MSD = np.var(m, axis = 0) # MSD
plt.plot(time_axis, MSD, label='Numerical')
plt.plot(time_axis, 2*kBT/gamma*np.asarray(time_axis), label='Theoretical')

plt.legend()
plt.xlabel("Time t")
plt.ylabel("MSD of X(t)")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\MSD.svg')
```



In [13]:
```
b, a = np.polyfit(time_axis, np.var(m, axis = 0), deg=1) # slope, intercept
```

In [14]:
```
b
```

Out[14]:
```
0.10536669920201379
```

In [15]:
```
a
```

Out[15]:
```
-0.01317263059635735
```

In [16]:
```
2*kBT/gamma
```

Out[16]:
```
0.1
```

In [17]:
```
import numpy as np

gamma = 10 # frictional coefficient
x_0 = 0 # initial position of the particle

h = 0.001 # time step
interval_len = 10 # the length of the interval # t = Nh

# total num of observed time stamps N = interval_len / h
# And I will chop off the decimal places after N in cases N is a non-integer
N = int(interval_len/h) # number of steps
N_ls = [i for i in range(0, N+1)] # The i list, (which are the index of ti, the observ
```

```python
time_axis = [round(i * h, 3) for i in N_ls] # [h, 2h,... nh=t]

mu, sigma = 0, 0.1 # mean and standard deviation

def form_trajectory():
        noise_sample = np.random.normal(mu, sigma, N)

        x_noise_i = x_0  # initialize the position
        x_noise_ls = [x_noise_i] # contains the trajectory of X at different time ti,

        for i in range(1, int(interval_len/h)+1):
            x_noise_i = x_noise_i + noise_sample[i-1]/gamma
            x_noise_ls.append(x_noise_i)
        return x_noise_ls
import time
start_time = time.time()
from joblib import Parallel, delayed
# ls_Xt is a list of a list resulting from parallel computing
# the inner list represents a trajectory
# the outer list represents an output from the parallel computing
Num_trajectory = 10 # Number of trajectories to be formed
ls_Xt = Parallel(n_jobs=-2)(delayed(form_trajectory)() for i in range(Num_trajectory))
print("--- %s seconds ---" % (time.time() - start_time))
import matplotlib.pyplot as plt
for Xt in ls_Xt: plt.plot (time_axis, Xt)
plt.xlabel("Time t")
plt.ylabel("Value of X(t)")
plt.savefig(r'C:\Users\alexi\Desktop\Plots\Trajectories.svg')
```

```
--- 0.40898895263671875 seconds ---
```