Notes

June 16, 2021

0.1 Notebook - 16/6/21

Geometric Brownian Motion Starting with simulating an SDE of the form of Geometric Brownian Motion:

$$dX(t) = \mu X(t)dt + \sigma X(t)dW(t)$$

where $\mu\left(t, X_t\right) = \mu X_t$ and $\sigma\left(t, X_t\right) = \sigma X_t$ are the drift and diffusion functions respectively, and therefore we have the explicit solution:

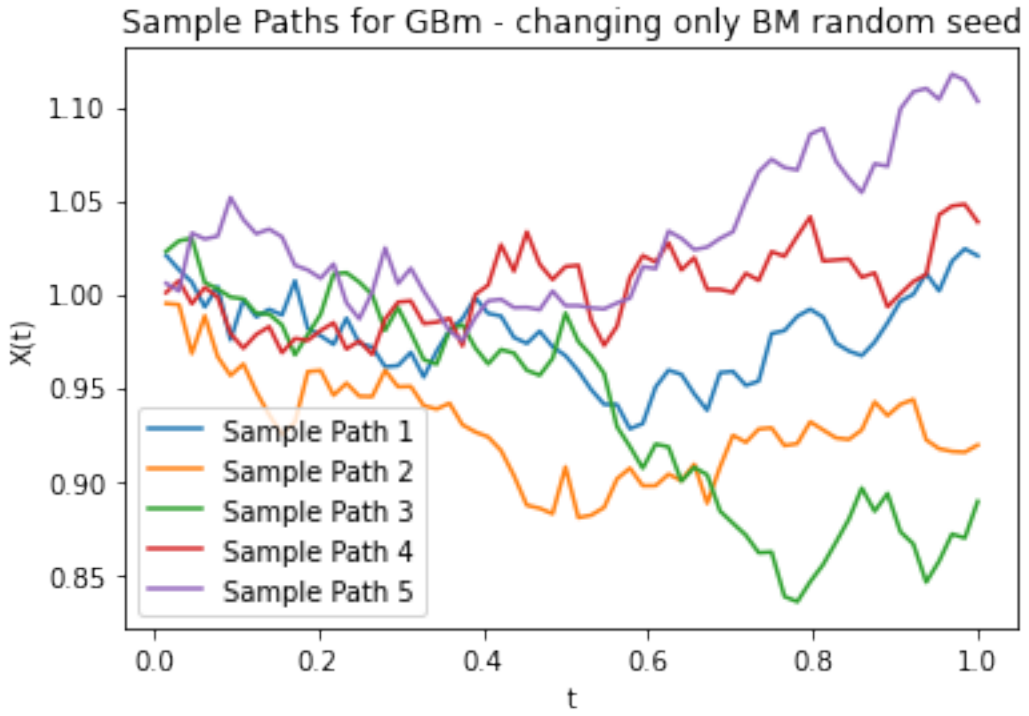
$$X(t) = X(0)\exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W(t)\right)$$

We start to simulate the integrals given here in differential form, by discretising the time into an evenly spaced mesh/grid on interval $[0, T]$ into N intervals like so: $0 = t_0 < t_1 < \cdots < t_N = T$ so the change in time between these discrete points as dt (a.k.a. h).

[508]:
```
import numpy as np
import matplotlib.pyplot as plt

# set model parameters
mu, sigma, X_0 = .01, .10, 1 # we cannot start X at 0 or the solution is 0!

# time discretisation
T, N = 1.0, 2**6 #Force T as float, and 2**n for setting timesteps seems tractible
dt = T / N # Horizon time of 1 for normalisation. N arbitrary.
t_grid = np.arange(dt, dt+1, dt) #start, stop, step

# Create and plot sample paths
num_paths = 5 # Number of paths we want to simulate (arbitrary)
for i in range(num_paths):
    # Create Brownian Motion
    np.random.seed(i+1)
    dW = np.sqrt(dt) * np.random.randn(N) # Increment: sqrt(dt)*random sample␣
 ↪from N(0,1)
    W  = np.cumsum(dW) # Path = sum of increments

    # exact solution
    X = X_0 * np.exp((mu - 0.5 * sigma**2) * t_grid + sigma * W)

    # Add line to plot
```

```
    plt.plot(t_grid, X, label = "Sample Path " + str(i+1))

#Plotting
plt.title('Sample Paths for GBm - changing only BM random seed')
plt.ylabel('X(t)'); plt.xlabel('t')
# Add legend
plt.legend()
```

[508]: <matplotlib.legend.Legend at 0x7fc1a6c9e910>

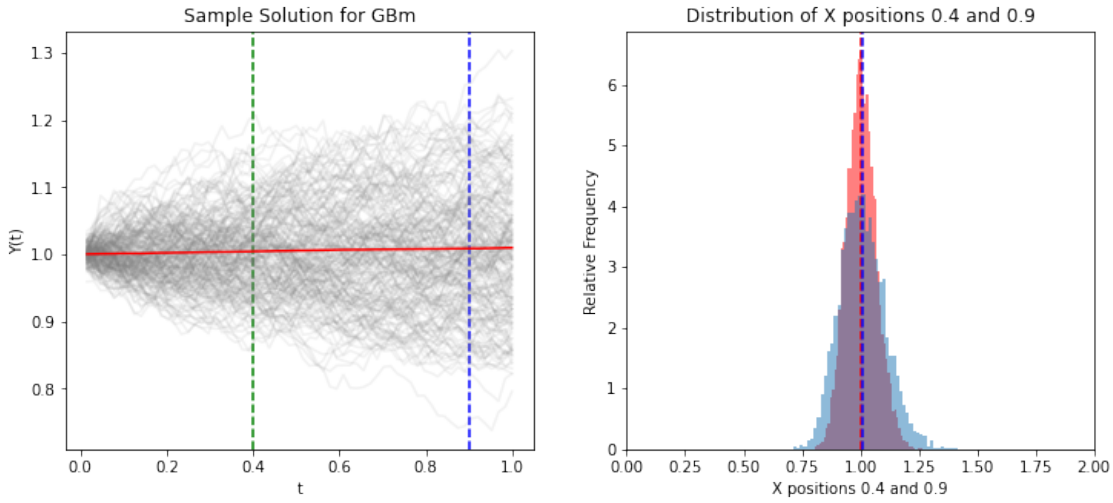


So now we have GBm working, we get a different trajectory each time due only to the random term W, which is our Wiener process (Brownian motion). Now we scale this up for many more paths using `num_paths` and we take a cross section across each of these at two seperate times to show that the mean of the paths is increasing as time goes on. The left plot shows the mean plotted where we can see the upward trend, and the right plot shows this for two specific positions as a histogram and you can see the peak also shifted to higher values for later times. NB - when I did this for mu = sigma = 1 this was quite clear but now I have different mu and sigma values, this is not so clear, or perhaps true?

[511]:
```
#NB: volatility and mean from prior cells carried through

fig = plt.figure(figsize=(12,5))
ax = fig.add_subplot(121)
plt.xlabel('t')
```

```
plt.ylabel('Y(t)')
plt.title('Sample Solution for GBm')

# Select and highlight cross section points: (recall T=1)
xpos1 = 0.4
xpos2 = 0.9
plt.axvline(x=xpos1, linestyle='--',color='green')
plt.axvline(x=xpos2, linestyle='--',color='blue')

# Simulate sample paths
X_1, X_2, X_total = [], [], []
num_paths = 10000
for i in range(num_paths): #ith path
    # Create Brownian Motion as above
    np.random.seed(i)
    dW = np.sqrt(dt) * np.random.randn(N) # ~N(0,1)*sqrt(dt) = N(0,t)
    W = np.cumsum(dW)

    # Exact Solution
    X = X0 * np.exp(((mu - 0.5 * sigma**2) * t_grid) + (sigma * W))
    X_total.append(X) #save all trajectories to allow mean calc [num_paths rows,␣
↪N points]

    X_1.append(X[int(xpos1 * N)]) #extract the X at particular points to allow␣
↪mean calc.
    X_2.append(X[int(xpos2 * N)]) # as above

    # Plot first 200 sample paths on left plot
    if i < 200:
        ax.plot(t_grid, X, label = "Sample Path " + str(i), color = 'gray',␣
↪alpha=0.1)

# Plot average line on left plot
ax.plot(t_grid, np.mean(X_total, 0), label="Sample Path " + str(i),color='red')

# Histogram plotting (NB: all fig 2 stuff below here)
fig.add_subplot(122)
num_bins = 50
plt.xlabel('X positions '+str(xpos1)+' and ' + str(xpos2))
plt.ylabel('Relative Frequency')
plt.xlim(0,30)
plt.title('Distribution of X positions '+str(xpos1)+' and ' + str(xpos2))
plt.hist(X_1,bins=num_bins,density=1,alpha=0.5, color='red')
plt.hist(X_2,bins=num_bins,density=1,alpha=0.5)
plt.xlim([0, 2])
plt.axvline(np.mean(X_total, 0)[int(xpos1 * N)],linestyle='--', color = 'red')
plt.axvline(np.mean(X_total, 0)[int(xpos2 * N)],linestyle='--', color = 'blue')
```

[511]: <matplotlib.lines.Line2D at 0x7fc1a8359250>



Checking Implementation for GBm To check whether we are able to recover the initial parameters in the model we do the following:

$$X_t = X_0 \exp\left((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t\right)$$

Taking natural log both sides:

$$\ln\left(\frac{X_t}{X_0}\right) = (\mu - \frac{1}{2}\sigma^2)t + \sigma\sqrt{t}Z$$

where $\Delta W_t = \sigma\sqrt{t}Z$ and $Z \approx N(0,1)$. Taking expectation

$$\begin{aligned}\mathbb{E}\left[\ln\left(\frac{X_t}{X_0}\right)\right] &= \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)t + \sigma\sqrt{t}Z\right] \\ &= \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)t\right] + \mathbb{E}\left[\sigma\sqrt{t}Z\right] \\ &= \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)t\right]\end{aligned}$$

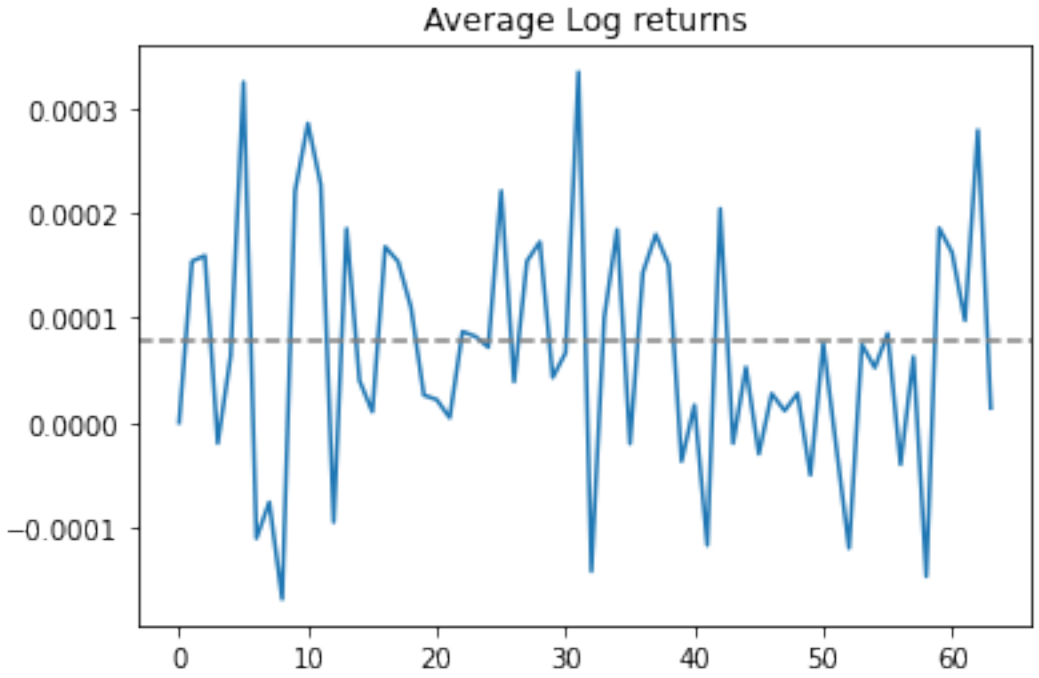
as the expectation of stochastic integral terms is zero due to the fact they are a martingale. Therefore we expect that the log returns (LHS) are equal to the final term on the RHS. The average log-returns are calculated below and compared (graphically) to the last RHS term.

[466]:
```
# Calculating average log returns - using the X_total from previous cell which␣
 ↪is
# a big array of all of the paths (rows = paths, cols = time)
X_stack = np.vstack(X_total)
X_stack.shape
```

```
X_stack_mean = np.mean(X_stack,0)
# plt.plot(X_stack_mean, label = 'stack')
# plt.plot(np.mean(X_total, 0), label = 'total', linestyle='--')
# plt.legend()

#initialise array for log returns
log_returns = np.zeros((num_paths,len(X_stack[0]))) # (5rows, 64 cols)
for i in np.arange(num_paths): #for each path i
    for j in np.arange(1,len(X_stack[i])): # for time j
        rets_Xij = np.log(X_stack[i][j]/X_stack[i][j-1]) #calculate log of␣
 ↪returns Xj/Xj-1
        log_returns[i][j]=rets_Xij
av_log_returns = np.mean(log_returns, axis=0)
```

[467]:
```
expectation = (mu - 0.5 * sigma**2)*dt
plt.plot(av_log_returns)
plt.axhline(expectation,linestyle='--', color = 'gray')
plt.title('Average Log returns')
```

[467]: Text(0.5, 1.0, 'Average Log returns')



Plot shows here that the gray line, which is plotted at,

$$X^* = \ln\left(\frac{X_{t_i}}{X_{t_{i+1)}}}\right) = \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)dt\right] \tag{1}$$

is representative of the the mean of the log returns as illustrated by the above plot.

[469]:
```
sigma * np.sqrt(dt) #Uh oh... whats this for again?! AGH!
```

[469]:
```
0.0125
```

So now say we know that $\mu = 1$, using our solution can we recover the volatility? or mu? Rearranging above (X^*) to get:

$$\mu = \big[\ln\big(\frac{X_t}{X_{t-1}}\big) - \frac{1}{2}\sigma^2\big]/T$$

and,

$$\sigma = \sqrt{2\big(\frac{mu - X^*}{T}\big)}$$

[482]:
```
# mu, sigma, X_0 = 2, 10, 1 from above
print('T == ', T,'mu == ', mu,'sigma == ', sigma)
recovered_mu = (np.mean(av_log_returns) + 0.5*sigma**2)/T
print('Getting original mu == ' , recovered_mu )
recovered_sigma = np.sqrt( 2*((mu-np.mean(av_log_returns))/T))
print('Getting original sigma == ' , recovered_sigma )

print('recovered_mu / mu == ', round(recovered_mu/mu,3))
print('recovered_sigma / sigma', round(recovered_sigma/sigma,3))
```

```
T ==  1.0 mu ==  0.01 sigma ==  0.1
Getting original mu ==  0.005068718577008561
Getting original sigma ==  0.1409346048562342
recovered_mu / mu ==  0.507
recovered_sigma / sigma 1.409
```

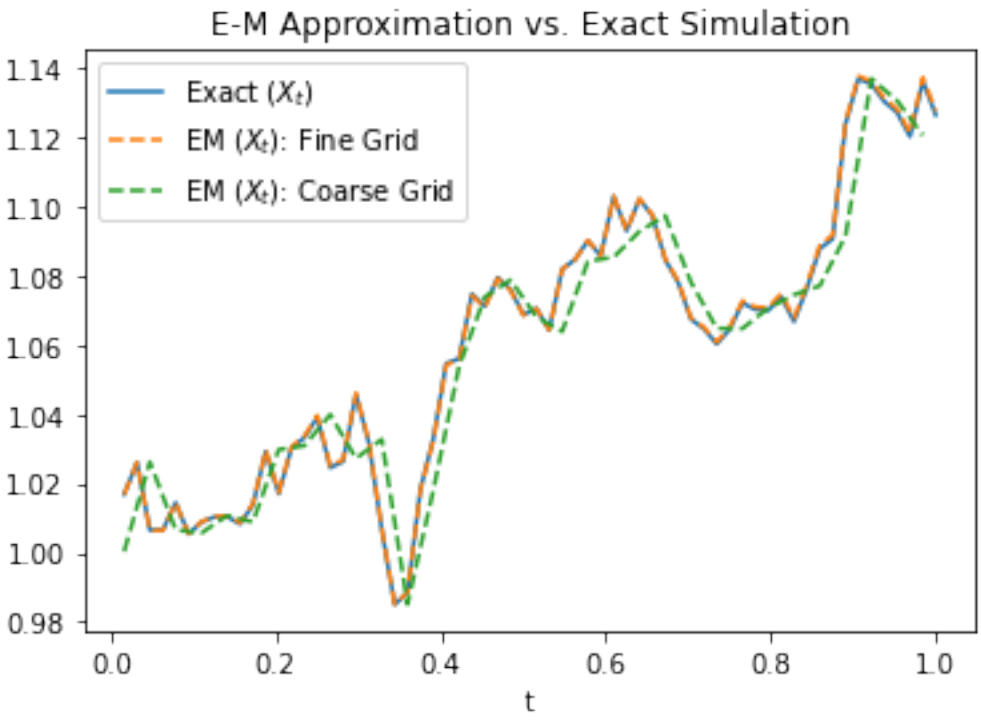
This has shown recovery of the original parameters to some degree, but it was not expected that they be recovered super accurately! but 50% seems high?!?!

Euler Maramaya Approximation for GBm and effects of coarse or fine Δt Showing that we get a better approximation for a smaller time discretisation - as Δt decreases, the closer the approximation is to the true solution.

[433]:
```
# Create Brownian Motion
np.random.seed(10)
dW = np.sqrt(dt) * np.random.randn(N)
W  = np.cumsum(dW)

# Exact Solution
X_true = X0 * np.exp((mu - 0.5*sigma**2)*t_grid + (sigma * W))

# EM Approximation - fine dt
X_EM_FINE, X = [], X_0
```

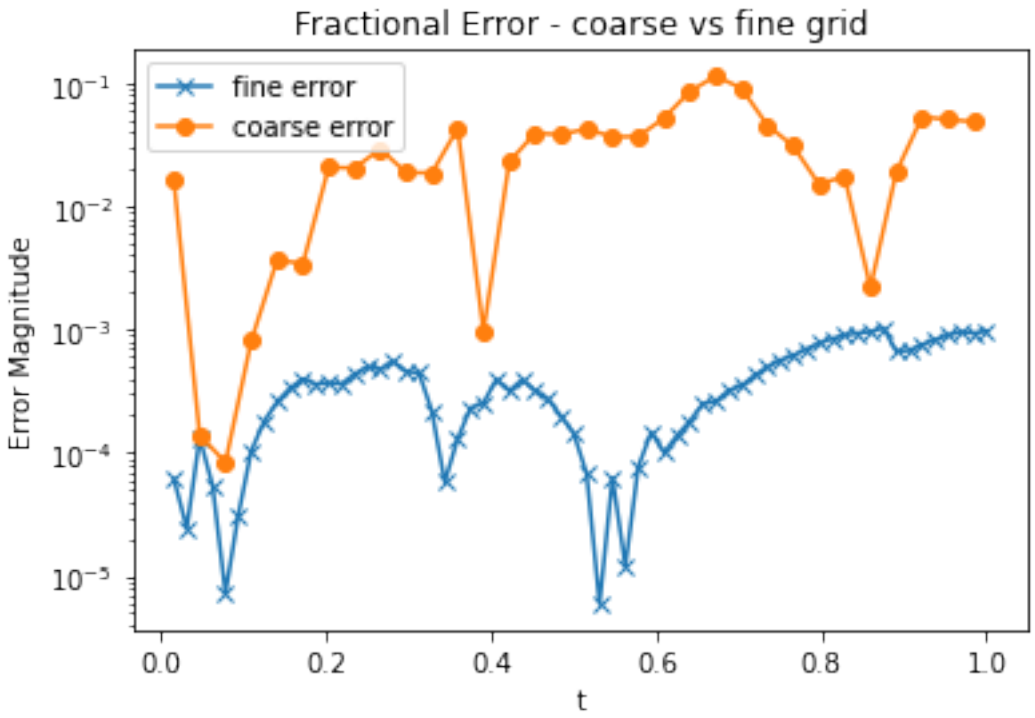
```
#fine_grid = np.arange(dt, 1 + dt, dt)  # this is t from above, omit, and use t␣
 ↪if you like
for j in range(N):
    X += mu*X*dt + sigma*X*dW[j]
    X_EM_FINE.append(X)

# EM Approximation - big dt
X_EM_COARSE, X, R = [], X_0, 2
coarse_grid = np.arange(dt,1+dt,R*dt)
coarse_err=[]
for j in range(int(N/R)):
    X += mu*X*(R*dt) + sigma*X*np.sum(dW[R*(j-1):R*j])
    X_EM_COARSE.append(X)
    coarse_err.append((np.abs(X_true[j]-X))/X_true[j])
    #capture error as a fraction of the true here as cant do it after!

# plotting
plt.plot(t_grid, X_true, label="Exact ($X_t$)")
plt.plot(t_grid, X_EM_FINE, label="EM ($X_t$): Fine Grid", ls='--')
plt.plot(coarse_grid, X_EM_COARSE, label="EM ($X_t$): Coarse Grid", ls='--')
plt.title('E-M Approximation vs. Exact Simulation')
plt.xlabel('t')
plt.legend(loc = 2)
```

[433]: <matplotlib.legend.Legend at 0x7fc1c69e0670>



[487]:
```
# Fractional Errors due to timestep
fine_err = (np.abs(X_true - X_EM_FINE))/X_true
plt.plot(t_grid, fine_err, marker='x', label='fine error')
#Log plot to show up the differences better
plt.semilogy(coarse_grid, coarse_err, marker='o', label = 'coarse error')
plt.title('Fractional Error - coarse vs fine grid')
plt.xlabel('t')
plt.ylabel('Error Magnitude')
plt.legend(loc = 2)
```

[487]: <matplotlib.legend.Legend at 0x7fc1a4b60a00>



The plot above shows that for the coarse grid, the fractional error (i.e. distance between true and approximated as a propoertion of the true) are pretty consistently higher than that of the fine grid, which is just a more quantified way of showing what we can 'eyeball' from the previous plot. There is a lot more variation in the error from the coarse grid, and it is pretty consistently higher, except for a few isolated areas of t - these dont seem to form any kind of pattern and seem arbitrary - propose this is an artefact of the fact we have a random variation?

Ornstein Uhlenbeck Process by Euler Maramaya The general form of the OU process is given as

$$dX_t = (a - bX_t)dt + \sigma dW_t$$

as opposed to GBm, this process (also known as the Vasicek model) has a drift term which depends on the current value of the process, and contains an extra term.

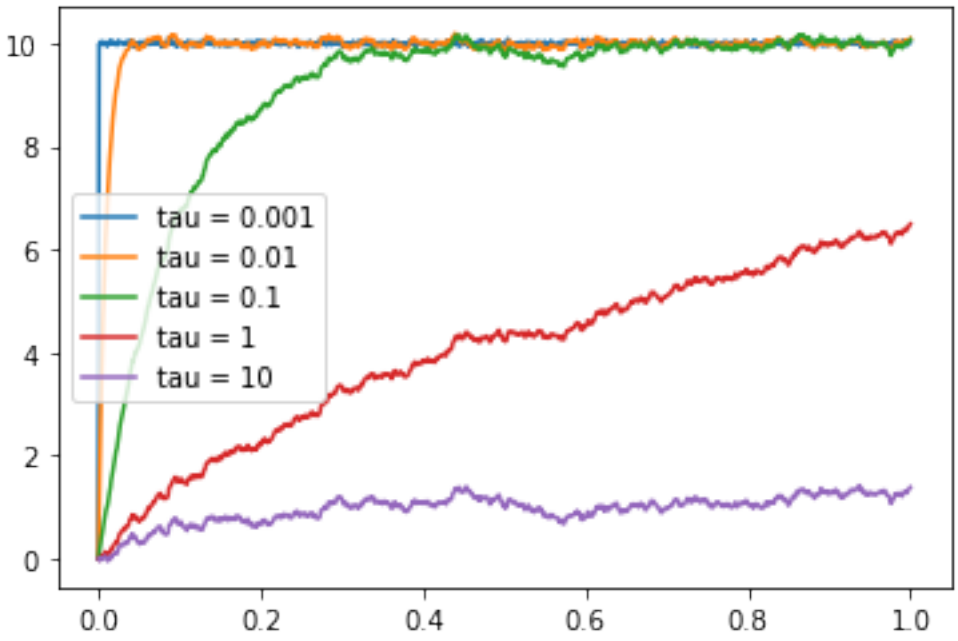
I also saw this one which has this extra 'time constant' so I modeled that one for the time being, and had a little play with the parameter 'tau' as is written here. Another variation has a multiplied of $\sqrt{\frac{2}{\tau}}$ which again, im not sure I understand the significance of.

$$dX = -\frac{1}{\tau}(x - \mu)dt + \sigma dW$$

where $dW = \sqrt{dt} \times N(0, 1)$.

[506]:
```
# OU Process by Euler

sigma, mu = 1., 10  # Standard deviation, mean
tau = 1  # Time constant
tau_var = [0.001, 0.01,0.1, 1, 10]

T, N = 1.0, 1000
dt = T/N  # Time step
dW = np.sqrt(dt) * np.random.randn(N)

t_discrete = np.linspace(0., T, N) # Vector of times.
X = np.zeros(N)
for tau_ in tau_var:
    for i in range(N - 1):
    #     X[i+1] = X[i] + (-(X[i] - mu) * dt / tau) + (sigma * np.sqrt(2/tau)) *␣
 ↪dW[i]
        X[i+1] = X[i] + (-(X[i] - mu)/tau_) * dt + sigma * dW[i]
    plt.plot(t_discrete, X, label='tau = '+str(tau_))
plt.legend()
```

[506]:
```
<matplotlib.legend.Legend at 0x7fc1a5f86b80>
```



This shows one realisation of the process for several values of tau to see what that does.

Milstein Method: GBm Adding a second-order "correction" term

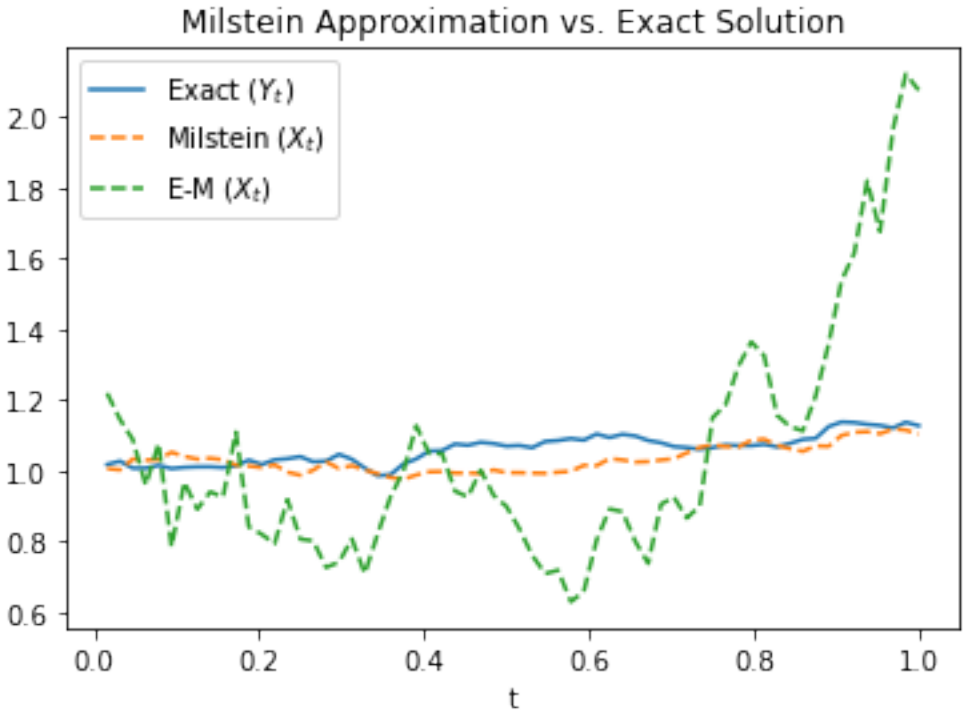
$$X_{n+1} - X_n = a(X_n)\Delta t + b(X_n)\Delta B_n + 0.5 b'(X_n) b(X_n) \left((\Delta B_n)^2 - \Delta t\right)$$

which implies the following for our Geometric Brownian Motion example

$$X_{n+1} - X_n = a(X_n)\Delta t + b(X_n)\Delta B_n + 0.5 \sigma^2 X_n \left((\Delta B_n)^2 - \Delta t\right)$$

[509]:
```
# Milstein Approximation
Xmilstein, X = [], X_0
for j in range(N):
    X += mu*X*dt + sigma*X*dW[j] + 0.5 * sigma**2 * X * (dW[j] ** 2 - dt)
    Xmilstein.append(X)

# Plot
plt.plot(t_grid, X_true, label="Exact ($Y_t$)")
plt.plot(t_grid, Xmilstein, label="Milstein ($X_t$)",ls='--')
plt.plot(t_grid, X_em_small, label="E-M ($X_t$)",ls='--')
plt.title('Milstein Approximation vs. Exact Solution')
plt.xlabel('t')
plt.legend(loc=2)
```

[509]: <matplotlib.legend.Legend at 0x7fc1a6bd5790>



I see pretty clearly there is something wrong here! working on it!