Notes

July 15, 2021

1 Stochastic Volatility - Notebook

1.1 Table of contents

1.2 Introduction

- Table of contents

1.3 Geometric Brownian Motion

- Table of contents

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:

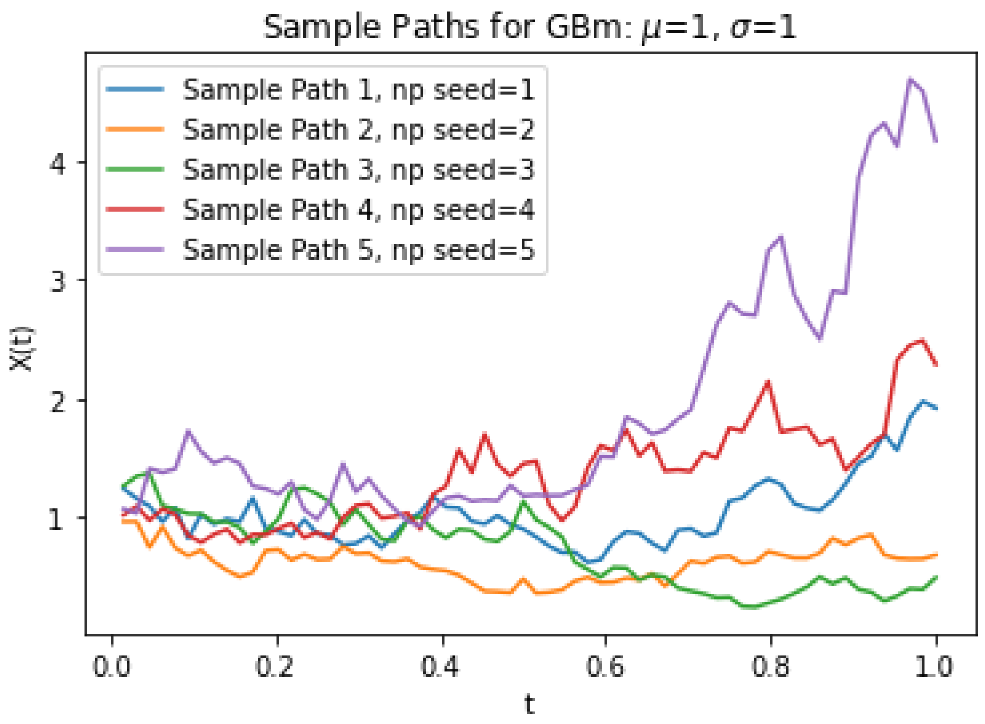
$$X(t) = X(0)\exp\left(\left(\mu - \frac{\sigma^2}{2}\right)t + \sigma W(t)\right)$$

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).

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

```
[2]: # TO DO!! - SET THIS SO THAT PATHS START AT 0!!! LEN=64=2**6... includes no 0 at␣
      ↪front...

     # set model parameters
     mu, sigma, X_0 = 1, 1, 1 # we cannot start X at 0 or the solution is 0! If␣
      ↪mu>>sigma, we see a linear trend as the drift is the dominant term

     # 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
     # t_grid = np.insert(t_grid,0,0)

     # 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)
     #     X = np.insert(X,0,X_0)

         # Add line to plot
         plt.plot(t_grid, X, label = "Sample Path " + str(i+1)+", np seed="+str(i+1))

     #Plotting
     plt.title('Sample Paths for GBm: $\mu$=' + str(mu)+', $\sigma$='+str(sigma))
     plt.ylabel('X(t)'); plt.xlabel('t')
     # Add legend
     plt.legend()
```

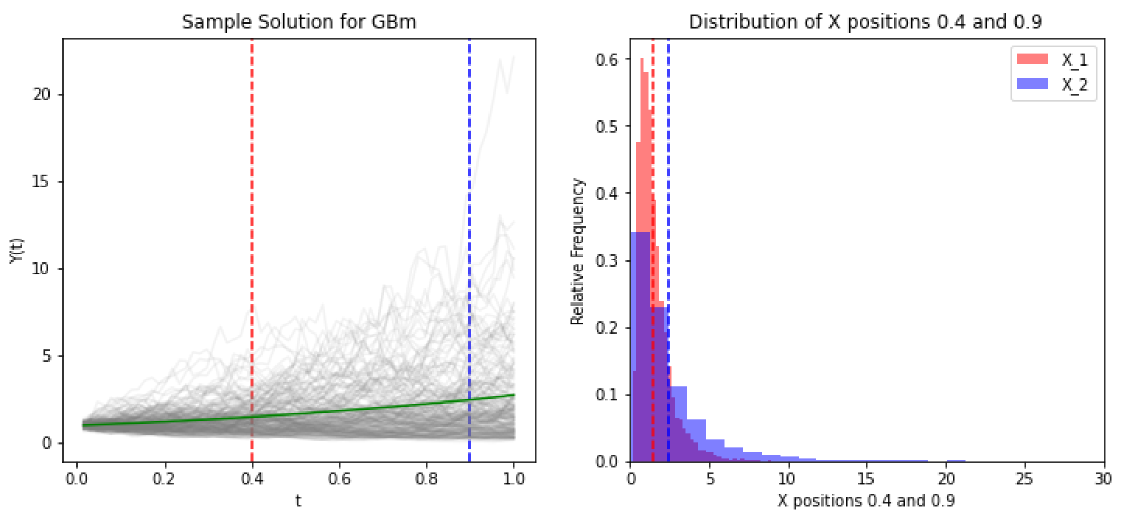
```
[2]: <matplotlib.legend.Legend at 0x7fc39c1109d0>
```



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.

[3]:
```
#NB: volatility and drift from prior cells carried through
%matplotlib inline
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='red')
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 = X_0 * 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
    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="Mean " + str(i),color='green')


# 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, label='X_1', color = 'red')
plt.hist(X_2,bins=num_bins,density=1,alpha=0.5, label='X_2', color = 'blue')
# 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')
plt.legend()
```

```
[3]: <matplotlib.legend.Legend at 0x7fc39c55a760>
```



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 is quite clear as drift does not dominate the diffusion and we see our Wiener process contribution clearly. If we have say $\mu = 1, \sigma = 0.1$ the drift term now dominates, and the diffusion term appears as a small perturbation on top of it, and it is approximately linearly increasing.

Checking Implementation for GBm

- Table of contents

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

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

$$= \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)t\right] + \mathbb{E}\left[\sigma\sqrt{t}Z\right]$$

$$= \mathbb{E}\left[(\mu - \frac{1}{2}\sigma^2)t\right] = (\mu - \frac{1}{2}\sigma^2)t$$

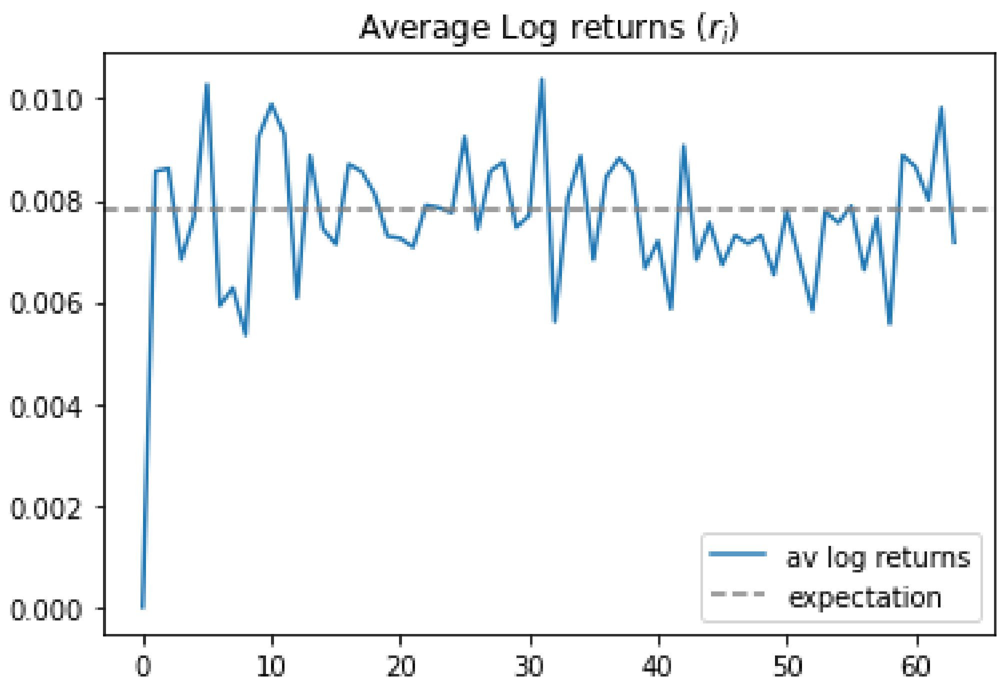
as the expectation of stochastic integral terms is zero due to the fact they are a martingale. Therefore we expect that the log returns (LHS) are equal to the final term on the RHS. The average log-returns are calculated below and compared (graphically) to the last RHS term.

[4]:
```
# 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)
print('X_stack.shape === ', X_stack.shape)
print('X_stack[0].shape === ', X_stack[0].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()

# Calculate average 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)

# Expectation/analytical mean
expectation = (mu - 0.5 * sigma**2)*dt #from equation
```

```
X_stack.shape ===  (10000, 64)
X_stack[0].shape ===  (64,)
```

[5]:
```
plt.plot(av_log_returns, label='av log returns')
plt.axhline(expectation,linestyle='--', color = 'gray', label = 'expectation')
plt.title('Average Log returns ($r_i$)')
plt.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7fc39d275400>
```



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

$$\mathbb{E}[r_i] = \mathbb{E}\Big[\ln\big(\frac{X_{t_i}}{X_{t_{i+1}}}\big)\Big] = \mathbb{E}\Big[(\mu - \frac{1}{2}\sigma^2)T\Big] = (\mu - \frac{1}{2}\sigma^2)T$$

is representative of the the mean of the log returns as illustrated by the above plot, showing that the average log returns is indeed equal to the expectation, or analytical mean shown.

[6]:
```
# sigma * np.sqrt(dt) #Uh oh... whats this for again? Seriously cant remember...
```

So now say we know that $\mu = 1$(which it is in our setup), using our solution can we recover the volatility? or mu? Rearranging above (r_i) to get:

$$\hat{\mu} = \frac{\mathbb{E}[r_i]}{\Delta t} + \frac{1}{2}\sigma^2$$

and,

$$\hat{\sigma} = \sqrt{\frac{var(r_i)}{\Delta t}}$$

and $var(r_i)$ is the sample variance of the log returns r_i, given by the standard formula $\sigma^2_{\hat{\mu}} = \frac{1}{n-1}\sum_{i=1}^{N}(r_i - \hat{\mu})^2$.

[7]:
```
# mu, sigma, T from above
print('T ==',T,', mu ==',mu,', sigma ==',sigma)
decimal_places = 4
```

```
mu_hat = (np.mean(av_log_returns)/dt) + 0.5*sigma**2
print('Getting original mu == ' , round(mu_hat,decimal_places) )

# recovered_sigma = np.sqrt( 2*((mu-np.mean(av_log_returns))/T))
sigma_hat = np.sqrt( np.var(log_returns)/dt )
print('Getting original sigma == ' , round(sigma_hat,decimal_places) )

print('{mu_hat}/{mu} == ', round(mu_hat/mu,decimal_places-1))
print('{sigma_hat}/{sigma}', round(sigma_hat/sigma,decimal_places-1))
```

```
T == 1.0 , mu == 1 , sigma == 1
Getting original mu ==  0.9869
Getting original sigma ==  0.9932
{mu_hat}/{mu} ==  0.987
{sigma_hat}/{sigma} 0.993
```

This has shown recovery of the original parameters to a good degree - for $\mu = \sigma = 1$ we get a recovery of 98.7% and 99.3% respectively. For $\mu = 1$, $\sigma = 0.1$ we get a recovery of 98.4% and 100.5% respectively In summary, we set μ and σ for use in the discretised GBm model, and obtained a path of random increments, and modelled the log returns. We then used the mean and variance of these log returns to see if we can recover the original solutions analytically.

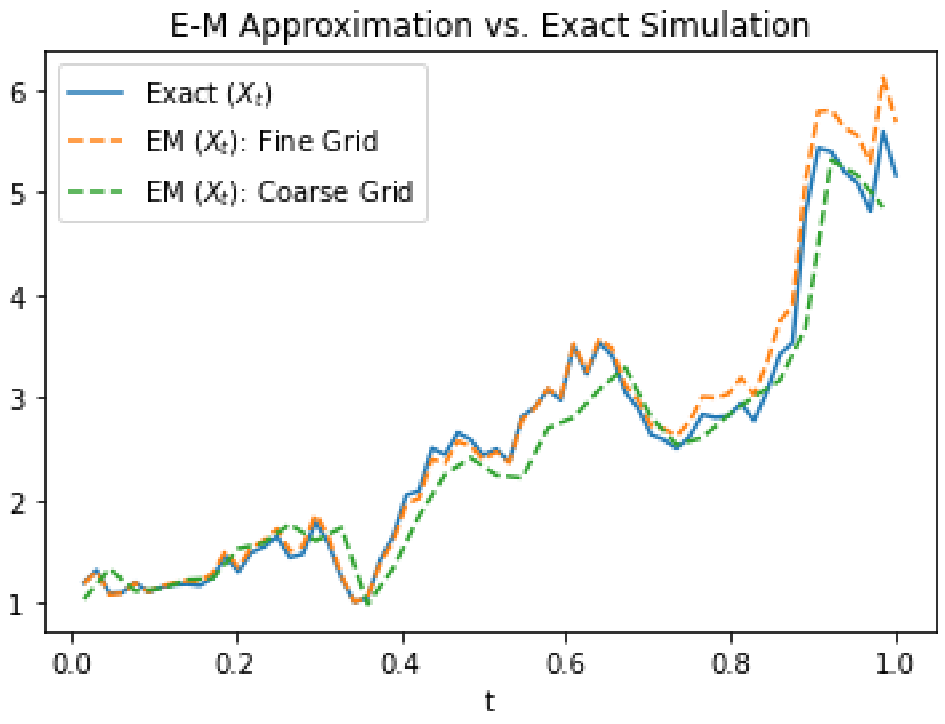
#### Euler Maramaya Approximation for GBm and effects of coarse or fine Δt * Table of contents

Showing that we get a better approximation for a smaller time discretisation - as Δt decreases, the closer the approximation is to the true solution.

[8]:
```
# Create Brownian Motion
np.random.seed(10)
dW = np.sqrt(dt) * np.random.randn(N) #N above
W  = np.cumsum(dW)

# Exact Solution
X_true = X_0 * np.exp((mu - 0.5*sigma**2)*t_grid + (sigma * W))

# 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)
#     if j<10:
#         print(X, t_grid[j])

# Coarse dt - Use R to make more coarse - dt = R*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)
```

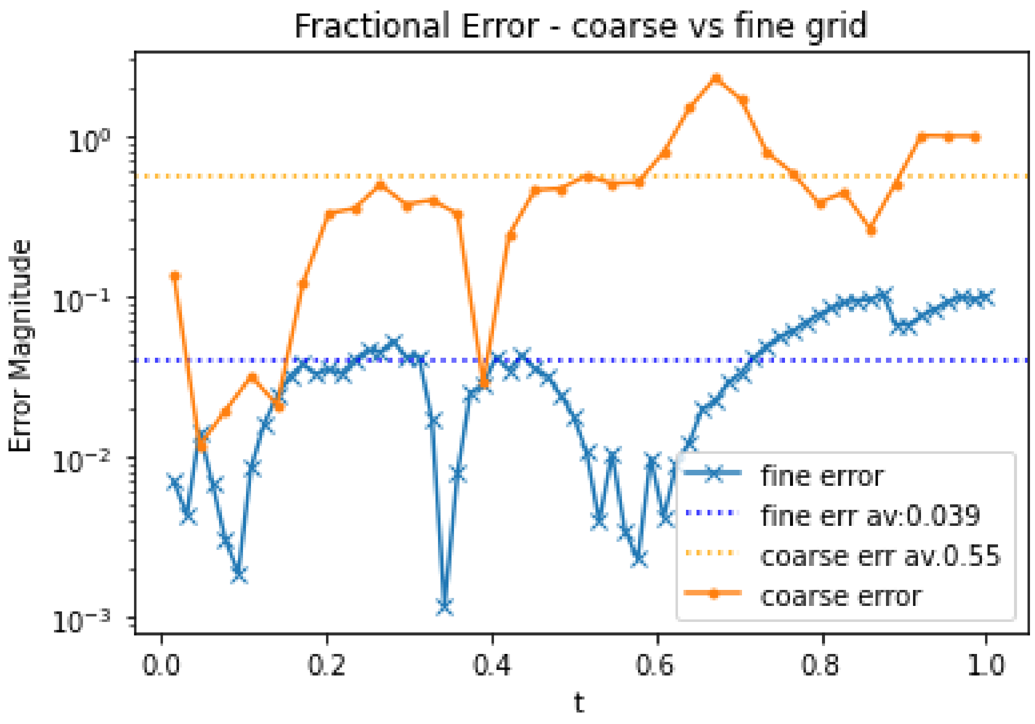
```
[8]: <matplotlib.legend.Legend at 0x7fc39d2c7c40>
```



ibid. Large Δt results in a poor approximation - not a surprise, and reminds us to always keep Δt small within the computational limits of hardware. Ask now how much error there is between the two, can I quantify how bad this has gotten with course grids, versus finer ones:

```
[9]: # 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')
     # plt.axhline(1,linestyle='--', color = 'gray', label = '100% error')
     plt.axhline(np.mean(fine_err),linestyle=':', color = 'blue', label = 'fine err␣
      ↪av:'+str(round(np.mean(fine_err),3)))
     plt.axhline(np.mean(coarse_err),linestyle=':', color = 'orange', label =␣
      ↪'coarse err av.'+str(round(np.mean(coarse_err),2)))

     # plt.axhline(0.01,linestyle='--', color = 'gray', label = '1% error')
     #Log plot to show up the differences better
     plt.semilogy(coarse_grid, coarse_err, marker='.', label = 'coarse error')
     plt.title('Fractional Error - coarse vs fine grid')
     plt.xlabel('t')
     plt.ylabel('Error Magnitude')
     plt.legend(loc = 0)
```

```
[9]: <matplotlib.legend.Legend at 0x7fc39c34e130>
```



The plot above shows that for the coarse grid, the fractional error (i.e. distance between true and approximated as a proportion of the true) are consistently higher than that of the fine grid (see legend for values), which is just a more quantified way of showing what we can ‘eyeball’ from the previous plot. With the exception of a few isolated periods, but these dont seem to form any kind of pattern and seem arbitrary - propose this is an artefact of the random variation?

#### Milstein Method: GBm * Table of contents

Adding a second-order "correction" term

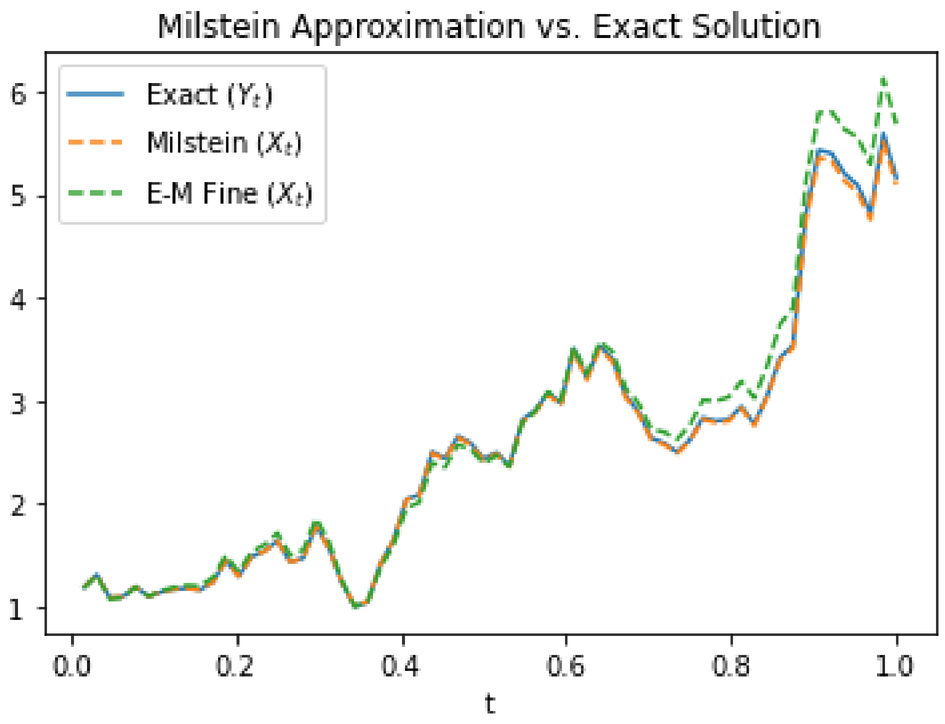
$$X_{n+1} - X_n = a(X_n)\Delta t + b(X_n)\Delta W_n + \frac{1}{2} b(X_n)\, \partial_X b(X_n) \left((\Delta W_n)^2 - \Delta t \right) + \tilde{R}$$

where $a(X_n) = \mu X_n$ and $b(X_n) = \sigma X_n$, $\tilde{R}$ is remainder terms from the Ito Taylor expansion used to obtain this scheme. This implies the following for our Geometric Brownian Motion example

$$X_{n+1} = X_n + \mu X_n \Delta t + \sigma X_n \Delta W_n + \frac{1}{2} \sigma^2 X_n \left((\Delta W_n)^2 - \Delta t \right)$$

[10]:
```
# Milstein Approximation
Xmilstein, X_milstein_err, 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)
    X_milstein_err.append((np.abs(X_true[j]-X))/X_true[j])


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

[10]:
```
<matplotlib.legend.Legend at 0x7fc39e356190>
```



The Milstein scheme gives a better fit to the path used here, shown in the above plot.

#### Ornstein-Uhlenbeck Process by Euler Maramaya * Table of contents

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.

Commonly, an extra 'time constant' is introduced so I modeled this, and had a little play with the parameter 'tau'. Another variation has a multiplied of $\sqrt{\frac{2}{\tau}}$ which again, im not sure I understand the significance of.

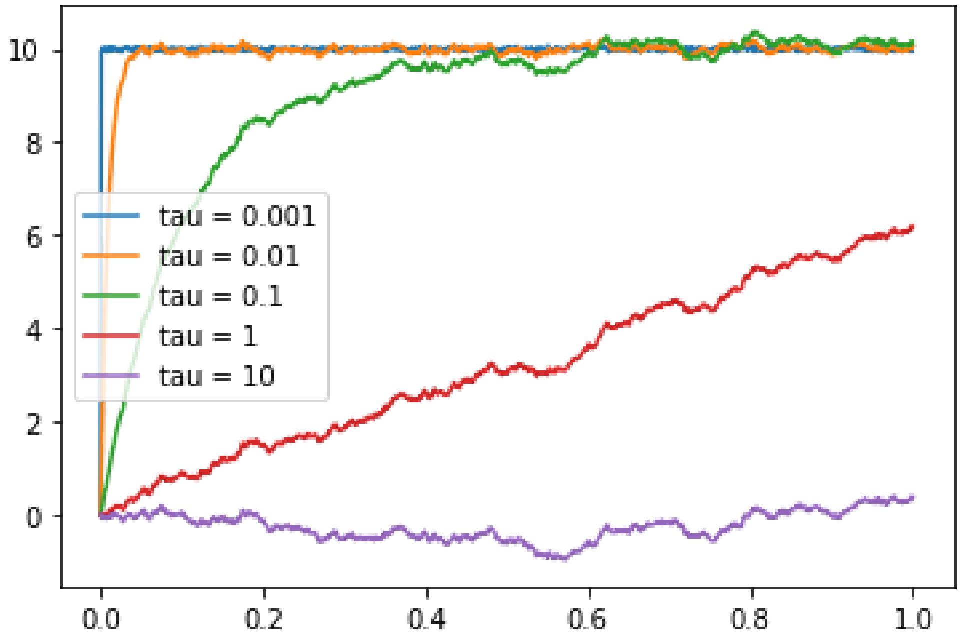
$$dX = -\frac{1}{\tau}(x - \mu)dt + \sigma dW$$

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

[14]:
```
# OU Process with time constant tau

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()
```

```
[14]: <matplotlib.legend.Legend at 0x7f90d6963f10>
```



This shows one realisation of the process for several values of tau to see what that does. In theory the 1/tau is the mean reversion speed, the higher tau is, the smaller 1/tau is so the slower

#### The Ornstein Uhlenbeck process post nightmares * Table of contents

So now we have the equation…

$$dX_t = \kappa(\theta - X_t)dt + \sigma dW_t$$

where κ is the speed of mean reversion, θ and σ are the long term mean and volatility of the process respectively, and X_t is the value of the process at time t. The process forms the basis of the Vasicek model for short interest rates, and was first proposed in 1977. Note here that if we have conditions such that $\theta < X_t$ then the drift term becomes negative, and the drift forces the process in the direction of the long term mean.

The analytical solution to this is as follows.

$$X_T = X_0e^{-kT} + \theta(1 - e^{-kT}) + \sigma \int_0^T e^{-k(T-t)}dW_t$$

With mean and variance,

$$\mathbb{E}[X] = X_0e^{-kT} + \theta(1 - e^{-kT})var[X] = \frac{\sigma^2}{2k}(1 - e^{-2kT})$$

We also know from taking the limit as $T \to \infty$

$$\lim_{T\to\infty} \mathbb{E}[X] = \lim_{T\to\infty} \left[X_0e^{-kT}+\theta(1-e^{-kT})\right] = X_0 \lim_{T\to\infty} e^{-kT}+\theta(1-\lim_{T\to\infty} e^{-kT}) = \theta \lim_{T\to\infty} var[X] = \lim_{T\to\infty} [\frac{\sigma^2}{2k}(1-e^{-2kT})$$

$$\lim_{T\to\infty} var[X] = \lim_{T\to\infty} [\frac{\sigma^2}{2k}(1 - e^{-2kT})]$$

The limit of the mean here displays the mean reverting property, but the variance is dependent on the speed of mean reversion, k. The higher the speed of mean reversion, the higher the drift towards the mean (quicker the convergence) and the lower the variance. Therefore this inverse relationship is entirely expected.

For a general start time t $\neq$ 0, and increment size Δt we have:

$$\mathbb{E}[X_{t+\Delta t}] = X_te^{-k\Delta t} + \theta(1 - e^{-k\Delta t})var[X_{t+\Delta t}] = \frac{\sigma^2}{2k}(1 - e^{-2k\Delta t})$$

We know that for a fixed value of X, we have normally distribution as

$$X_{t+\Delta t} = X_te^{-k\Delta t} + \theta(1 - e^{-k\Delta t}) + \sigma\sqrt{\frac{(1 - e^{-2k\Delta t})}{2k}}Z$$

where $Z \sim N(0, 1)$ and Δt is the increment between timesteps, and is a constant. This is comparable to an autroregressive process of order 1 (time series?)

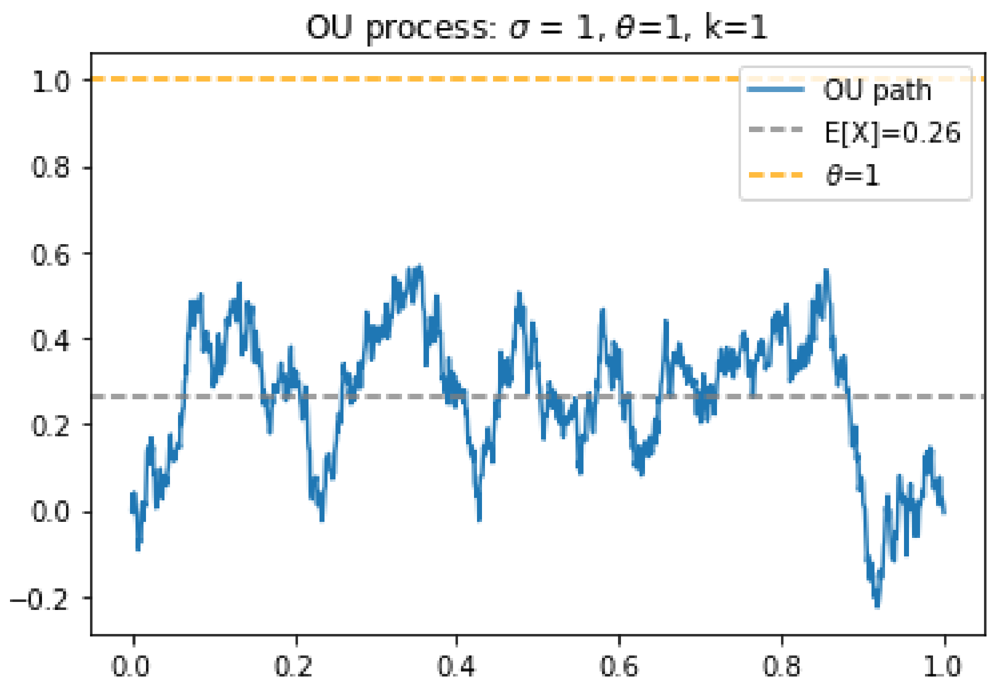
$$y_{i+1} = by_i + a + \epsilon_{i+1}b = e^{-k\Delta t}a = \theta\ (1 - e^{-k\Delta t})\epsilon = \sigma\sqrt{\frac{(1 - e^{-2k\Delta t})}{2k}}Z$$

If we can obtain a and b from data, and then using our knowledge of k and Δt we can obtain estimates for θ and σ

[128]:
```
#Ornstein Uhlenbeck Process - Euler Marumaya discretisation

sigma, theta, k = 1, 1, 1 # Standard deviation, mean, mean reversion speed

T, N = 1.0, 1000
dt = T/N  # Time step
```

```
Z = np.random.randn(N)

t_grid = np.linspace(0., T, N) # Vector of times.
X = np.zeros(N)
for i in range(N-1):
    X[i+1] = X[i]*np.exp(-k*dt) + theta*(1-np.exp(-k*dt)) + sigma*np.sqrt(␣
 ↪(1-np.exp(-2*k*dt))/(2*k) )*Z[i]

#mean and variance of process:
expectation = np.mean(X)
variance = np.var(X)
#variance from analytical solution:(mean exact is theta - see plot)
var_exact = (sigma**2/(2*k)) * (1-np.exp(-2*k*dt))

plt.plot(t_grid, X, label = 'OU path')
plt.axhline(expectation,linestyle='--', color = 'gray', label =␣
 ↪'E[X]='+str(round(expectation,2)))
plt.axhline(theta,linestyle='--', color = 'orange', label =␣
 ↪'$\\theta$='+str(theta))
plt.title('OU process: $\sigma$ = ' + str(sigma)+', $\\theta$='+str(theta)+',␣
 ↪k='+str(k))
plt.legend()

print('Variance Calcs:\nnp.var(X) = ', round(variance,5))
print('var_exact', round(var_exact,6))
```

```
Variance Calcs:
np.var(X) =  0.02528
var_exact 0.000999
```



We see in this plot that the calculated mean from the generated path, and the analytical mean are not close, unless the value of k is high. Given a set of parameters, how long is the horizon time before we observe our mean reversion? I.e, how long does it take for our process to show the property $X \sim N(\theta, \frac{\sigma^2}{2k})$. Using the concept of half life, we aim to calculate a value of t, namely $t_{1/2}$ that satisfies:

$$\mathbb{E}[X_{t_{1/2}}] = X_0 + \frac{\theta - X_0}{2}$$

given that this is the half way point between the value of the process at X_0, and the long term mean θ. Using this expression we can observe the following:

$$\mathbb{E}[X_{t_{1/2}}] = X_0 + \frac{\theta - X_0}{2} X_0 e^{-kt_{1/2}} + \theta(1 - e^{-kt_{1/2}}) = X_0 + \frac{1}{2}(\theta - X_0) X_0 e^{-kt_{1/2}} + \theta - \theta e^{-kt_{1/2}} = X_0 + \frac{1}{2}(\theta - X_0) e^{-kt_{1/2}} =$$

Therefore;

$$t_{1/2} = \frac{-ln(1/2)}{k}$$

This means that the horizon needed to observe mean reversion is inversely proportional to the mean reversion speed which makes perfect sense - for a higher speed of mean reversion, the shorter the time needed to observe the process reach its mean value. For $k = 3$, we therefore calculate $t_{1/2} =$

1.4 Maximum Likelihood Estimation

Maximum Likelihood Estimation (MLE) is a method by which parameters within a distribution can be estimated. The method assumes inherently that probability distribution function is available, and that the log likelihood function (below) is differentiable with respect to the parameters to be estimated. The aim of MLE is to maximise the likelihood function, or equivalently to minimise the negative log likelihood function, L.

$$L(\theta|X) = \prod_{t=1}^{N} f(X_t|\theta)$$

where θ is a vector of parameters, and X_t is the process with probability density function f. Taking the logarithm of the likelihood function does not affect the position at which the function would be maximised and it allows for easier manipulation of the function. Therefore maximising the log likelihood is equivalent to maximising the likelihood, f.

$$\ln\big(L(\theta|X)\big) = \ln\Big(\prod_{t=1}^{N} f(X_t|\theta)\Big)$$

If we assume for illustration that $\theta = (\mu, \sigma)^T$, we maximise the log likelihood function setting our differentiation with respect to each parameter to zero, and solve to obtain estimates for parameters in θ.

$$\frac{\partial}{\partial\theta}\ln\big(L(\theta|X)\big) = 0\frac{\partial}{\partial\mu}\ln\big(L(\theta|X)\big) = 0$$

1.5 CKLS Model

- Table of contents

The CKLS (Chan, Karolyi, Longstaff, Sanders) model includes both Ornstein Uhlenbeck processes (i.e. the Vasicek model) and the CIR model, and was presented as a more general short rate model.

The model goes a little like this…

$$dX_t = (\alpha - \beta X_t)dt + \sigma X_t^{\gamma} dW_t$$

where W_t is a standard wiener process, distributed as $N(0,t)$. When the parameter $\gamma = \frac{1}{2}$, we can see that the model reduces to the CIR (square root) model. We can further see that when $\gamma = 0$ we recover the Vasicek model. Finally, with $\alpha = 0$ and $\gamma = 1$, we have Geometric Brownian Motion.

The CKLS model posesses no closed form solution, so we must look immediately to discretise this equation. Once again we can simulate this using the Euler-Maramaya method resulting in the following discretisation.

$$X_t - X_{t-1} = (\alpha - \beta X_{t-1})\Delta t + \sigma X_{t-1}^{\gamma}\sqrt{\Delta t}Z\Delta X = (\alpha - \beta X_{t-1})\Delta t + \sigma X_{t-1}^{\gamma}\sqrt{\Delta t}Z$$

where Z is a standard normally distributed process distributed $N(0,1)$, such that $W_t \equiv Z\sqrt{t}$, and $\Delta X = X_t - X_{t-1}$.

From this we can derive a transition density,

$$f(X_t|X_{t-1}) = ...$$

which leads to a likelihood function given by

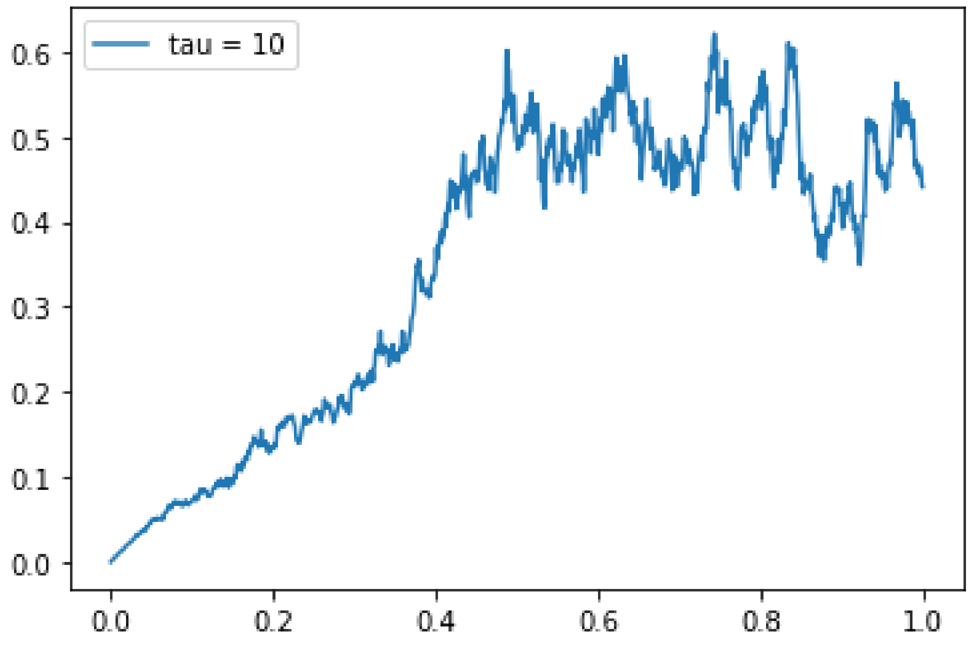
$$L(\{X_i\}_{i=1}^N|\theta) = ...$$

and log likelihood...

$$\ln(L(\{X_i\}_{i=1}^N|\theta)) = ...$$

[133]:
```
# CKLS Process by Euler

alpha, beta, sigma, gamma = 1., 1., 1., 1.
T, N = 1.0, 1000
dt = T/N
dW = np.sqrt(dt) * np.random.randn(N) # root(dt) * Wt
t_discrete = np.linspace(0., T, N)

X = np.zeros(N)

for i in range(N - 1):
    X[i+1] = X[i] + (alpha - beta * X[i]) * dt + sigma * X[i]**gamma * dW[i]

plt.plot(t_discrete, X, label='tau = '+str(tau_))
plt.legend()
```

[133]:
```
<matplotlib.legend.Legend at 0x7fc389992130>
```



1.5.1 Convergence

- Table of contents

We think of convergence as the change in 'distance' between a numerical approximation of a process, and the true value. If over time the approximation approaches the true solution i.e. our distance decreases, the numerical scheme is said to converge with time to the true value, with a rate β. In the Euler Marumaya (EM) method, as we decrease the Δt the numerical approximation becomes an increasingly closer match to the true solution, i.e. the two converge as $\Delta t \to 0$. In this case, we can have that either $\Delta t \to 0$ or $N \to \infty$ to achieve convergence. The rate of the convergence depends on how we quantify the distance between the two processes. Calculating the strong and weak error, are two ways to quantify this difference, resulting in two different rates of convergence.

Strong error The strong error considers the errors between the solution generated by the numerical scheme, and the true solution for every value of time in a sample path, given that both are based on the same brownian increments. We are essentially looking at the pathwise difference between two random variables. The strong error is

$$\mathrm{e}_{\mathbf{s}}(h) := \sup_{0 \le t_n \le T} \mathbb{E}\big|X_n^{(h)} - X_{t_n}\big|$$

where X_n^h is interpreted as the numerical approximation of a random variable X, by a numerical scheme of timstep/partition size $h = \Delta t = t_{i+1} - t_i$ at time t_n. The term X_{t_n}, is the true solution to which the numerical approximation is compared at time t_n. The process is said to be strongly convergent if $\mathrm{e}_{\mathbf{s}}(h) \to 0$ as $h \to 0$. The strong error is interpreted as the expectation/mean of the pathwise errors.

Weak error In the weak error we are capturing the average behaviour of a process with time, i.e. the first moment. Given the function ϕ in the class of polynomials up to degree k, the weak error of the processes $X_n^{(h)}$ and X_{t_n} defined above is

$$\mathrm{e}_w^{(h)} = \sup_{0 \le t_n \le T} |\mathbb{E}[\phi(X_n)] - \mathbb{E}[\phi(X_{t_n})]|$$

This error is interpreted as the error between the means, or first moments. A scheme converges weakly with rate β if $\exists c > 0$. Further, this convergence is of order p if:

$$e_{\Delta t}^{weak} \le K \Delta t^p$$

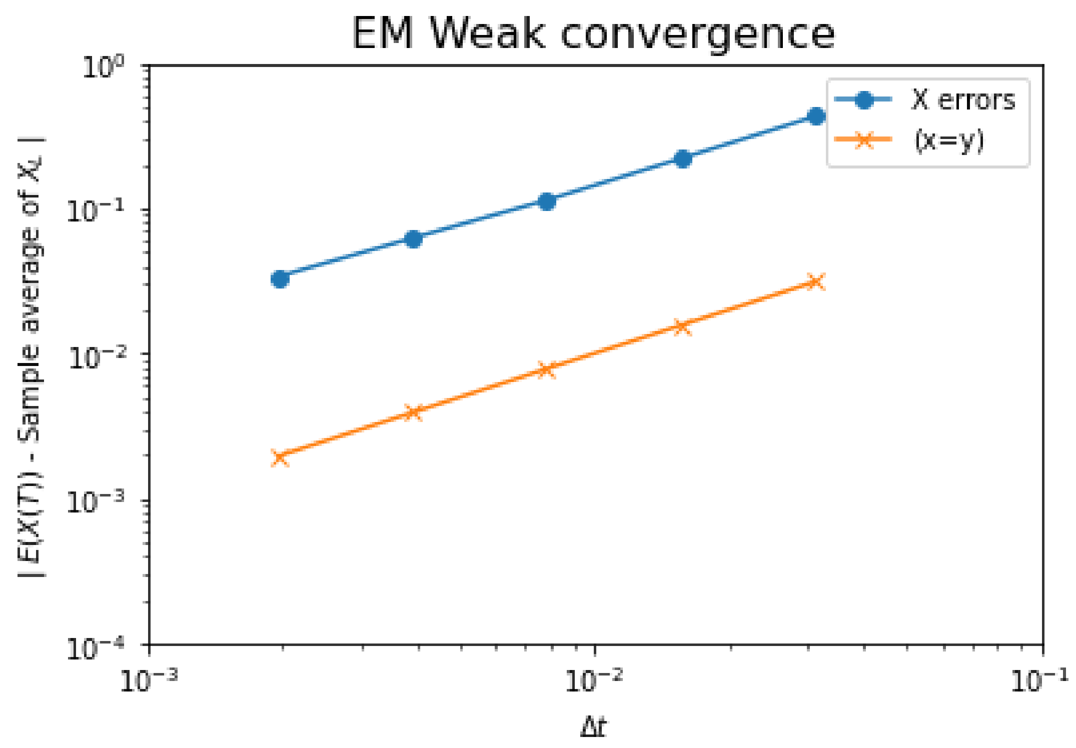
There are two types of convergence, strong and weak. A process is said to be weakly convergent if the following is true:

$$|\mathbb{E}[X_t^h] - \mathbb{E}[X(t_n)]| \le$$

#### Weak Convergence * Table of contents

[13]:
```
# Test weak convergence of Euler-Maruyama
# SDE is dX = mu*X dt + sigma*X dW,   X(0) = X_0
#       where mu = 2, sigma = 0.1, and X_0 = 1
```

```
#
# E-M uses 5 different timesteps: 2^(p-10),  p = 1,2,3,4,5.
# Examine weak convergence at T=1:  | E (X_L) - E (X(T)) |.
#
# Different paths are used for each E-M timestep.
# Adapted from
# Desmond J. Higham "An Algorithmic Introduction to Numerical Simulation of
#                     Stochastic Differential Equations"
# http://www.caam.rice.edu/~cox/stoch/dhigham.pdf

np.random.seed(102)

mu=2
sigma=0.1
X_0=1
T=1
M=50000

Xem=np.zeros((5,1))
for p in range(1,6): # for all Dt values
    Dt = 2**(p-10)
    L=float(T)/Dt
    Xtemp=X_0*np.ones((M,1))

    for j in range(1,int(L)+1): #for each time along the path
        Winc=np.sqrt(Dt)*np.random.randn(M) #random increment
        Xtemp += Dt*mu*Xtemp + sigma*np.multiply(Xtemp.T,Winc).T
    Xem[p-1] = np.mean(Xtemp,0)
Xerr = np.abs(Xem - np.exp(gamma))

Dtvals=np.power(float(2),[x-10 for x in range(1,6)])
plt.loglog(Dtvals,Xerr, 'o-', label='X errors')
plt.loglog(Dtvals,Dtvals, 'x-', label='(x=y)')
plt.axis([1e-3, 1e-1, 1e-4, 1])
plt.xlabel('$\Delta t$'); plt.ylabel('| $E(X(T))$ - Sample average of $X_L$ |')
plt.title('EM Weak convergence', fontsize=16)
plt.legend()
plt.show()

### Least squares fit of error = C * Dt^q ###
A = np.column_stack((np.ones((p,1)), np.log(Dtvals)))
rhs=np.log(Xerr)
sol = np.linalg.lstsq(A,rhs,rcond=None)[0]
q=sol[1][0]
resid=np.linalg.norm(np.dot(A,sol) - rhs)
print('q = ', q)
print('residual = ', resid)
```

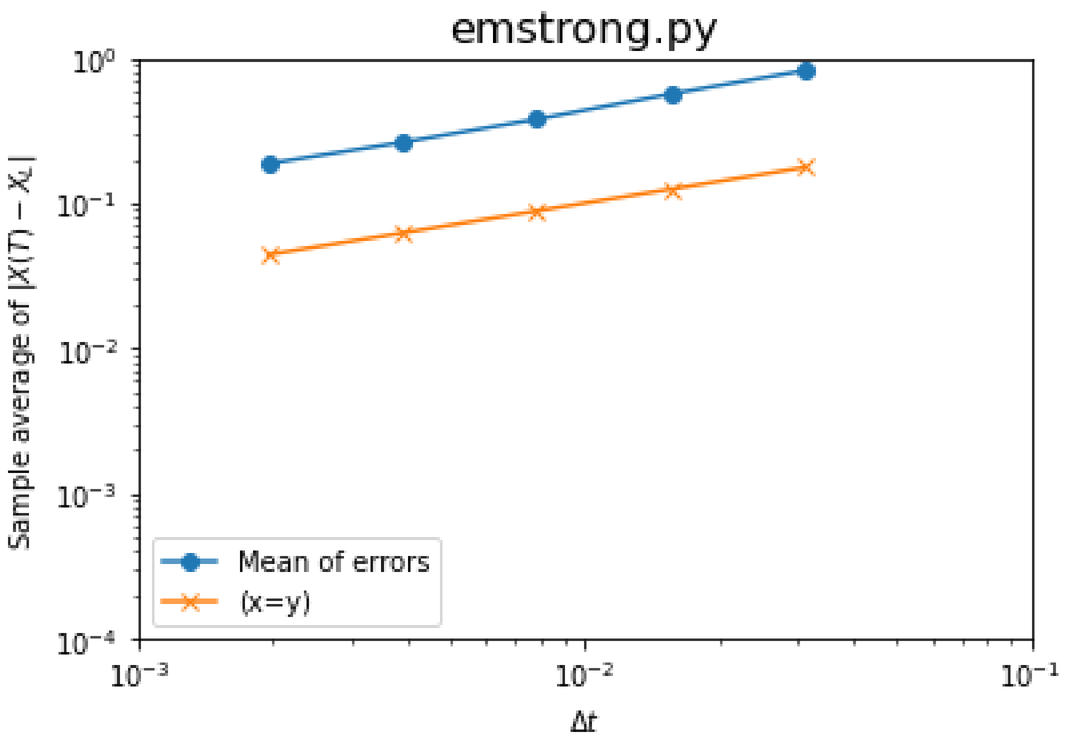


```
q =  0.9165248454248613
residual =  0.051086286431117185
```

#### Strong Convergence * Table of contents

[12]:
```
# EMSTRONG Test strong convergence of Euler-Maruyama
# SDE is dX = mu*X dt + sigma*X dW,  X(0) = X_0
#      where mu = 2, sigma = 1, and X_0 = 1
#
# Discretized Brownian path over [0,1] has dt = 2^(-9).
# E-M uses 5 different timesteps: 16dt, 8dt, 4dt, 2dt, dt.
# Examine strong convergence at T=1:  E | X_L - X(T) |.
#
# Adapted from
# Desmond J. Higham "An Algorithmic Introduction to Numerical Simulation of
#                     Stochastic Differential Equations"
# http://www.caam.rice.edu/~cox/stoch/dhigham.pdf

np.random.seed(100)

mu=2
sigma=1
X_0=1
T=1
```

```
N=2**9
dt = float(T)/N
M=1000 #

Xerr=np.zeros((M,5))
for s in range(M):
    dW=np.sqrt(dt)*np.random.randn(1,N)
    W=np.cumsum(dW)
    Xtrue = X_0*np.exp((mu-0.5*sigma**2)*T+sigma*W[-1])
    for p in range(5): #for each power
        R=2**p
        Dt=R*dt
        L=N/R
        Xem=Xzero
        for j in range(1,int(L)+1):  # for each step in the path
            Winc=np.sum(dW[0][range(R*(j-1),R*j)])
            Xem += Dt*mu*Xem + sigma*Xem*Winc
        Xerr[s,p]=np.abs(Xem-Xtrue)

Dtvals=dt*(np.power(2,range(5)))
plt.loglog(Dtvals,np.mean(Xerr,0),'o-', label='Mean of errors')
plt.loglog(Dtvals,np.power(Dtvals,0.5),'x-',label='(x=y)')
plt.axis([1e-3, 1e-1, 1e-4, 1])
plt.xlabel('$\Delta t$')
plt.ylabel('Sample average of $|X(T)-X_L|$')
plt.title('emstrong.py',fontsize=16)
plt.legend()
plt.show()

### Least squares fit of error = C * Dt^q ###
A = np.column_stack((np.ones((5,1)), np.log(Dtvals)))
rhs=np.log(np.mean(Xerr,0))
sol = np.linalg.lstsq(A,rhs,rcond=None)[0]
q=sol[1]
resid=np.linalg.norm(np.dot(A,sol) - rhs)
print('residual = ', resid)
```



```
residual =  0.034424249204813184
```