Computational Finance

AMSTERDAM BUSINESS SCHOOL

Business

Monte Carlo Methods

Brownian Motion

- We saw last week that the binomial tree implies for $X_t \equiv \log S_t$ that

$$X_{i\delta t} = X_{(i-1)\delta t} + R_i \iff \delta X_t = R_i, \qquad (\dagger)$$

  where $R_i = \log u$ or $R_i = \log d$, with probabilities $\mathbb{Q}[u]$ and $\mathbb{Q}[d]$.
- Equation ($\dagger$) is a *stochastic difference equation*.
- Its *solution*

$$X_T = \log S_0 + \sum_{i=1}^{N} R_i = \log S_0 + \sigma\sqrt{\delta t}(2k - N)$$

  is called a *binomial process*, or in the special case with $\mathbb{E}[R_i] = 0$, a *random walk*.

- We also saw that if we let $N \to \infty$ (so that $\delta t \to 0$),

$$X_T - X_0 \xrightarrow{d} N(\mu T, \sigma^2 T), \qquad \mu \equiv r - \frac{1}{2}\sigma^2.$$

- The argument can be repeated for every $X_t, t \leq T$, showing that

$$X_t - X_0 \xrightarrow{d} N(\mu t, \sigma^2 t),$$

  and that for any $0 < t < T, X_t - X_0$ and $X_T - X_t$ are independent.
- As $\delta t \to 0$, $\{X_t\}_{t \geq 0}$ becomes a continuous time process: the indexing set is now given by the entire positive real line.
- This continuous time limit (with $\mu = 0$ and $\sigma^2 = 1$) is called *Brownian motion*, or *Wiener process*.
- From now on, rather than modelling in discrete time and then letting $\delta t \to 0$, we will directly model in continuous time, using Brownian motion as a building block.

- Definition of (standard) *Brownian Motion*: Stochastic process $\{W_t\}_{t\geq 0}$ satisfying
    - $W_0 = 0$;
    - The increments $W_t - W_s$ are independent for all $0 \leq s < t$;
    - $W_t - W_s \sim N(0, t-s)$ for all $0 \leq s \leq t$;
    - Continuous sample paths.

- Properties of Brownian Sample Paths:
  - *Continuity*: by assumption, and also $W_{t+\delta t} - W_t \sim N(0, \delta t) \to 0$ as $\delta t \downarrow 0$;
  - *Nowhere differentiability*: intuitively, this is seen from
$$\frac{W_t - W_{t-\delta t}}{\delta t} \sim N\left(0, \frac{1}{\delta t}\right), \quad \frac{W_{t+\delta t} - W_t}{\delta t} \sim N\left(0, \frac{1}{\delta t}\right);$$
    left and right difference quotients do not have (common) limit as $\delta t \downarrow 0$.
  - *Self-similarity*: Zooming in on a Brownian motion yields another Brownian motion: for any $c > 0, X_t = \sqrt{c} W_{t/c}$ is a Brownian motion.

Simulating Brownian Motion

- In order to simulate Brownian Motion, we need to discretize it. As usual, split the time interval $[0, T]$ into N parts of length $\delta t = T/N$ and let $t_i \equiv i\delta t$.
- As in (†), let

  $$W_{i\delta t} = W_{(i-1)\delta t} + R_i \iff \delta W_t = R_i,$$

  but where now we model R_i as normal rather than binomial:

  $$R_i = \sqrt{\delta t} \cdot Z_i, \qquad Z_i \sim \mathrm{N}(0, 1).$$

- As a slight generalization, *Brownian motion with drift* is obtained from

  $$\delta X_t \equiv X_{i\delta t} - X_{(i-1)\delta t} = \mu\delta t + \sigma\delta W_t = \mu\delta t + \sigma\sqrt{\delta t} Z_i,$$

  and where we allow X_0 to be an arbitrary value.
- This implies

  $$X_t \equiv X_0 + \mu t + \sigma W_t,$$

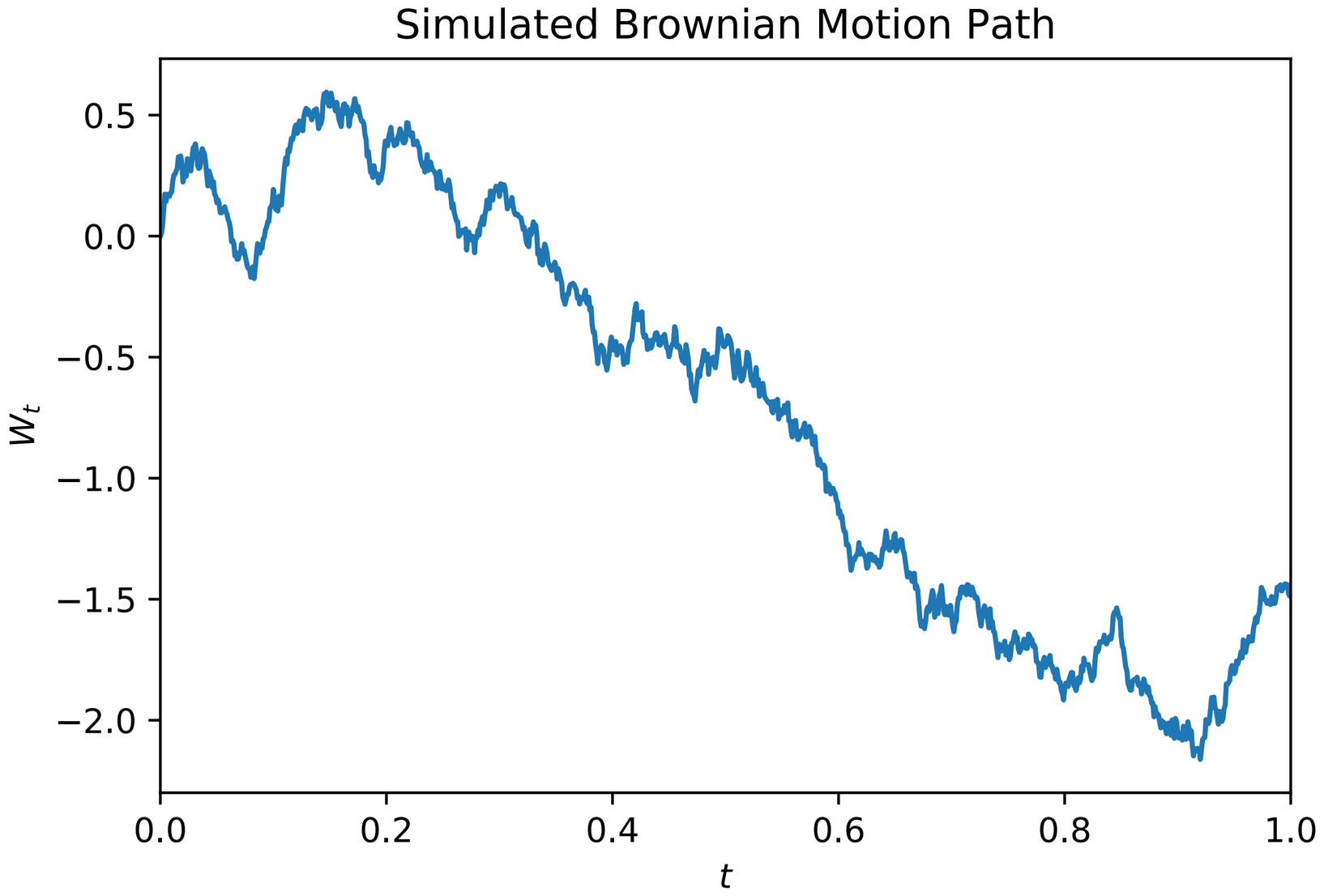
  so that $\mathbb{E}[X_t] = X_0 + \mu t, \mathrm{Var}[X_t] = \sigma^2 t$. Hence the average upwards (or downwards if $\mu < 0$) tendency over a time interval δt is $\mu\delta t$.

- In order to implement this, we need a way of drawing random samples from the normal distribution.
- Computers are deterministic machines. They cannot generate true random numbers.
- Instead, they construct sequences of pseudo-random numbers from a specified distribution that *look* random, in the sense that they pass certain statistical tests.
- E.g., NumPy's `np.random.randn(d0[, d1, ...])` constructs an array of standard normal pseudo random numbers.
- Random number generators use a *seed* value for initialization. Given the same seed, the same pseudo-random sequence will be returned.
- NumPy picks the seed automatically. To force it to use a specific seed, use `np.random.seed(n)`. Putting this line at the beginning of your Monte-Carlo program ensures that you get exactly the same results every time the program is run.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
```

In [2]:
```
def bmsim(T, N, X0=0, mu=0, sigma=1):
    """Simulate a Brownian motion path.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(N+1)  #N+1 is one more than we need, actually. This way we won't have to grow dX by X0
    dX = mu*deltaT + sigma*np.sqrt(deltaT)*z  #X[j+1]-X[j]=mu*deltaT + sigma*np.sqrt(deltaT)*z[j].
    dX[0] = 0.
    X = np.cumsum(dX)
    X += X0
    return tvec, X
```

In [3]:
```
np.random.seed(0)
tvec, W = bmsim(1, 1000)
W = pd.Series(W, index=tvec)
W.plot()
plt.title('Simulated Brownian Motion Path')
plt.xlabel("$t$"); plt.ylabel("$W_t$");
plt.savefig("img/BMpath.svg"); plt.close()
```


Simulated Brownian Motion Path
0.5
0.0
−0.5
−1.0
−1.5
−2.0
W_t
0.0
0.2
0.4
0.6
0.8
1.0
t

Ito Processes

- Ito processes generalize Brownian motion with drift by allowing the drift and volatility to be time-varying:

$$\delta X_t \equiv X_{i\delta t} - X_{(i-1)\delta t} = \mu_{t_{i-1}}\delta t + \sigma_{t_{i-1}}\delta W_t.$$

- We allow μ_{t_i} and σ_{t_i} to be stochastic; e.g., they may depend on X_{i-1}, as in

$$\delta X_t = \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\delta W_t.$$

- One can show that under appropriate conditions, the process has a well-defined limit as $N \to \infty$ (so that $\delta t \downarrow 0$), for which we write

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t.$$

  This is known as a stochastic differential equation (SDE).

- If σ_t is stochastic, then the distribution of X_T implied by the SDE is no longer normal. One can show that if N is large enough, then this distribution is correctly reproduced by the discretized version (which is then known as as the *Euler approximation*).

- Remark: in continuous time, a process $\{X_t\}_{t\geq 0}$ is a *martingale* if
    - $\mathbb{E}[|X_t|] < \infty$, for all $t \geq 0$;
    - $\mathbb{E}[X_t|\mathcal{F}_s] = X_s$, for all $t > s \geq 0$, where $\mathcal{F}_t$ denotes the information on X_t up to time t. An Ito process is a martingale if and only if the drift is zero, because (using $s = t_{i-1}$)

$$\begin{aligned}\mathbb{E}[\delta X_t|\mathcal{F}_{t_{i-1}}] &= \mathbb{E}[\mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\delta W_t|\mathcal{F}_{t_{i-1}}] \\ &= \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\mathbb{E}[\delta W_t|\mathcal{F}_{t_{i-1}}] \\ &= \mu(t_{i-1}, X_{i-1})\delta t + \sigma(t_{i-1}, X_{i-1})\mathbb{E}[W_{t_i} - W_{t_{i-1}}|\mathcal{F}_{t_{i-1}}] = \mu(t_{i-1}, X_{i-1})\delta t.\end{aligned}$$

- **Example**: Take $\mu(t, S_t) \equiv \mu S_t$ and $\sigma(t, S_t) \equiv \sigma S_t$. The resulting process

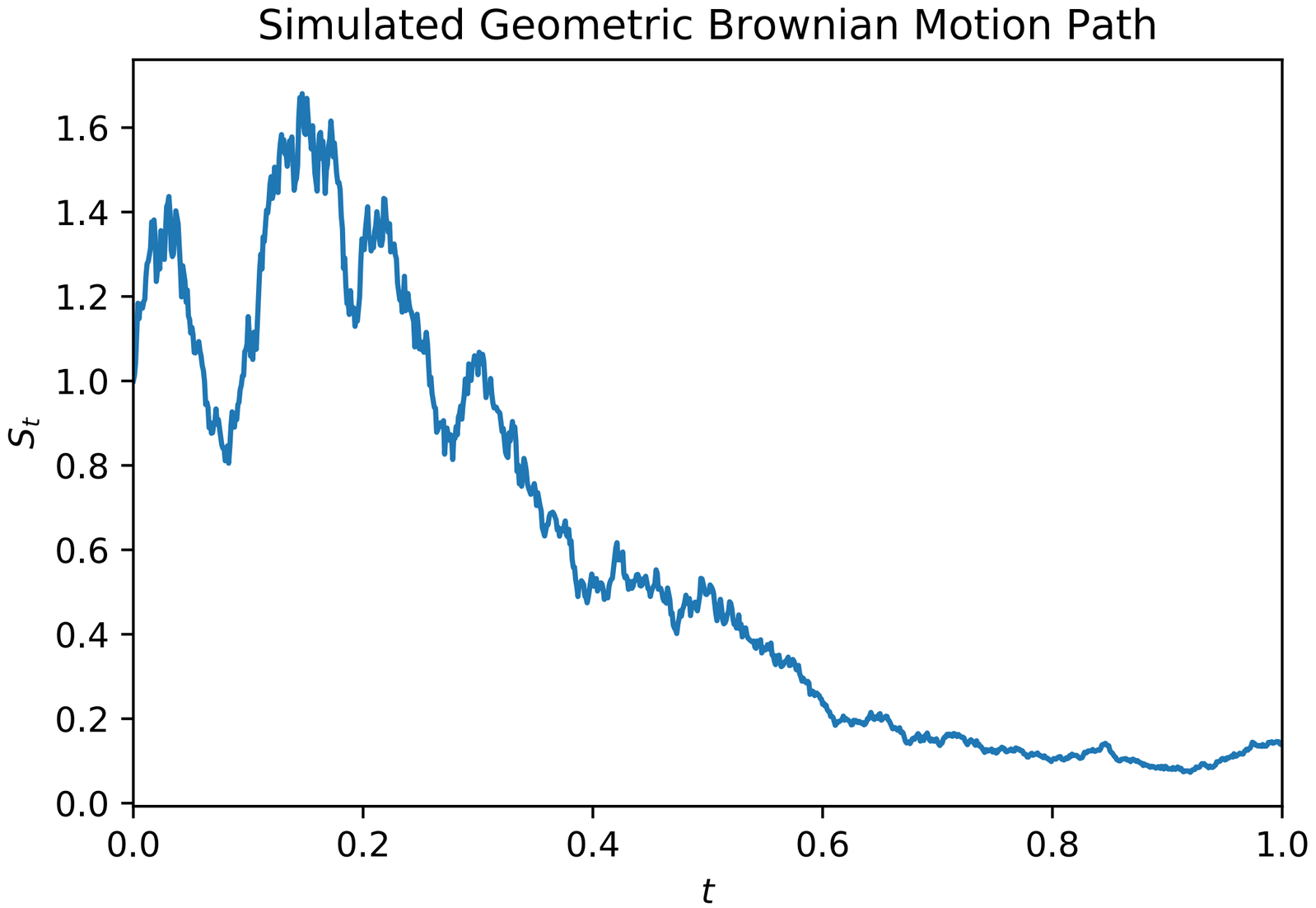
$$dS_t = \mu S_t dt + \sigma S_t dW_t$$

is known as *Geometric Brownian Motion*. Its Euler approximation is

$$\delta S_t \equiv S_i - S_{i-1} = \mu S_{i-1} \delta t + \sigma S_{i-1} \sqrt{\delta t} Z_i .$$

In [4]:
```
def gbmsim(T, N, S0=1, mu=0, sigma=1):
    """Simulate a Geometric Brownian motion path.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(N+1)  #Again one more than we need. This keeps it comparable to bmsim.
    S = np.zeros_like(z)
    S[0] = S0
    for i in xrange(0, N): #Note: we can no longer vectorize this, because S[:, j] is needed for S[:, j+1].
        S[i+1] = S[i] + mu*S[i]*deltaT + sigma*S[i]*np.sqrt(deltaT)*z[i+1]
    return tvec, S
```

In [5]:
```
np.random.seed(0)
tvec, S = gbmsim(1, 1000)
S = pd.Series(S, index=tvec)
S.plot()
plt.title('Simulated Geometric Brownian Motion Path')
plt.xlabel("$t$"); plt.ylabel("$S_t$")
plt.savefig("img/GBMpath.svg"); plt.close()
```


Simulated Geometric Brownian Motion Path
S_t
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
t

Ito's Lemma

- Ito's lemma answers the question: if X_t is an Ito process with given dynamics (SDE), then what are the dynamics of a function $f(t, X_t)$?
- It can be stated as follows: Let $\{X_t\}_{t\geq 0}$ be an Ito process satisfying $dX_t = \mu_t dt + \sigma_t dW_t$, and consider a function $f : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ with continuous partial derivatives

$$\dot{f}(t,x) = \frac{\partial f(t,x)}{\partial t}, \qquad f'(t,x) = \frac{\partial f(t,x)}{\partial x}, \qquad f''(t,x) = \frac{\partial^2 f(t,x)}{\partial x^2}.$$

Then

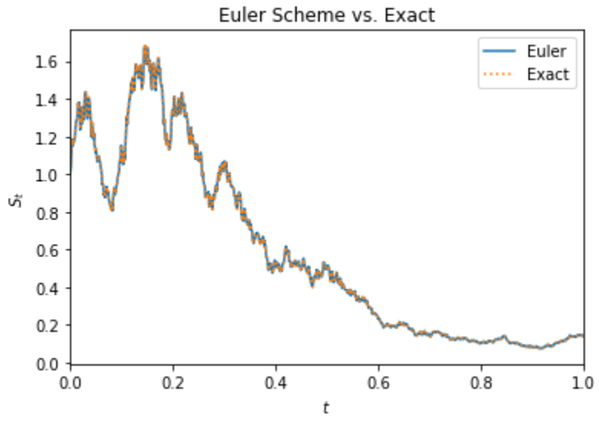
$$df(t,X_t) = \dot{f}(t,X_t)dt + f'(t,X_t)dX_t + \frac{1}{2}f''(t,X_t)\sigma_t^2 dt.$$

- **Example**: Geometric Brownian Motion. Let $dS_t = \mu S_t dt + \sigma S_t dW_t$ and $X_t = f(S_t) = \log S_t$. Then $\dot{f}(S_t) = 0, f'(S_t) = 1/S_t, f''(S_t) = -1/S_t^2$, and

$$\begin{aligned} dX_t = df(S_t) &= \dot{f}(S_t)dt + f'(S_t)dS_t + \frac{1}{2}f''(S_t)(S_t\sigma)^2 dt \\ &= \frac{1}{S_t}dS_t - \frac{1}{2S_t^2}(S_t\sigma)^2 dt \\ &= \frac{1}{S_t}(\mu S_t dt + \sigma S_t dW_t) - \frac{1}{2}\sigma^2 dt \\ &= \nu dt + \sigma dW_t, \qquad \nu = \mu - \frac{1}{2}\sigma^2. \end{aligned}$$

- So we find that $S_t = \exp(X_t) = S_0 \exp(\nu t + \sigma W_t)$. This gives us an alternative way to simulate GBM, which avoids the error introduced by the Euler approximation.
- We see that S_T has a lognormal distribution in the GBM model, and that S_t is a martingale (driftless process) if and only if $\mu = 0$, so that $\nu = -\frac{1}{2}\sigma^2$.

In [6]:
```
N=1000  #Try changing N to 100, then 10!
np.random.seed(0)
tvec, S1 = gbmsim(1, N)  #mu = 0, sigma = 1
np.random.seed(0)  #Use the same seed, otherwise we'd get different paths.
tvec, X = bmsim(1, N, 0, -.5)  #nu = mu - .5*sigma**2 = -.5
S2 = np.exp(X)
S1 = pd.Series(S1, index=tvec)
S2 = pd.Series(S2, index=tvec)
S1.plot()
S2.plot(linestyle=":")
plt.title("Euler Scheme vs. Exact")
plt.xlabel("$t$")
plt.ylabel("$S_t$")
plt.legend(["Euler", "Exact"]);
```



- Intuition for Ito's lemma: (see Hull, 2012, Appendix to Ch. 13): In standard calculus, the total differential

$$df = \dot{f}(t, g(t))dt + f'(t, g(t))dg(t)$$

  is the linear part of a Taylor expansion; the remaining terms are of smaller order as $dt, dg(t) \to 0$, so the total differential is a local linear approximation to f.
- If $g(t) = X_t$, an Ito process, take a 2nd order Taylor approximation:

$$\begin{aligned}\delta f \approx \dot{f}(t, X_t)\delta t + f'(t, X_t)\delta X_t \\ + \frac{1}{2}\left[\frac{\partial^2 f}{\partial t^2}(\delta t)^2 + 2\frac{\partial^2 f}{\partial t \partial X_t}(\delta t)(\delta X_t) + \frac{\partial^2 f}{\partial X_t^2}(\delta X_t)^2\right].\end{aligned}$$

- We have that $\delta X_t \equiv (X_t - X_{t-\delta t}) \approx \mu_{t-\delta t}\delta t + \sigma_{t-\delta t}\delta W_t \sim N(\mu_{t-\delta t}\delta t, \sigma^2_{t-\delta t}\delta t)$. Thus, $\mathbb{E}[(\delta X_t)^2] \approx (\mu_{t-\delta t}\delta t)^2 + \sigma^2_{t-\delta t}\delta t \approx \sigma^2_{t-\delta t}\delta t$; i.e., the 2nd order term is of the same order of magnitude as the 1st order term δt.
- It can be shown that as $\delta t \to 0, (\delta X_t)^2$ can be treated as non-stochastic: $(dX_t)^2 = \sigma_t^2 dt$. Together with $(dt)^2 = 0$ and $(dt)(dX_t) = 0$, this gives the result.

The Black-Scholes Model

- Black and Scholes assumed the following model:
  - The stock $\{S_t\}_{t\in[0,T]}$ follows GBM:
    $$dS_t = \mu S_t dt + \sigma S_t dW_t.$$
  - The stock pays no dividends.
    - Cash bond price $B_t = e^{rt} \iff dB_t = rB_t dt$; riskless lending and borrowing at the same rate r.
    - European style derivative with price C_t and payoff $C_T = (S_T)$.
    - Trading may occur continuously, with no transaction costs.
    - No arbitrage opportunities.
- The problem is to find the price $C_t, t \in [0, T]$.

- It can be shown that the FTAP holds in continuous time as well: if the market is arbitrage free, then there exists a risk neutral measure $\mathbb{Q}$ under which all assets earn the risk free rate (on average), and the price of a claim is the discounted expected payoff under $\mathbb{Q}$. If the market is complete, then $\mathbb{Q}$ is unique. This gives us a pricing formula for general European claims:

$$C_t = e^{-r(T-t)}\mathbb{E}^{\mathbb{Q}}\left[C_T | \mathcal{F}_t\right].$$

- This implies that if we can simulate the stock price under the measure $\mathbb{Q}$, then we can price the claim by Monte Carlo simulation.

- In the BS model, it can be shown that under the risk-neutral measure $\mathbb{Q}$,

$$dS_t = rS_t dt + \sigma S_t dW_t.$$

- Note that by Ito's lemma, the discounted stock price $\tilde{S}_t \equiv e^{-rt} S_t =: f(t, S_t)$ satisfies

$$\begin{aligned} d\tilde{S}_t &= \dot{f}(t, S_t)dt + f'(t, S_t)dS_t + \frac{1}{2} f''(t, S_t)\sigma^2 S_t^2 dt \\ &= -re^{-rt} S_t dt + e^{-rt} dS_t + 0 \\ &= -r\tilde{S}_t dt + e^{-rt}(rS_t dt + \sigma S_t dW_t) \\ &= \sigma \tilde{S}_t dW_t. \end{aligned}$$

- I.e., $\tilde{S}_t$ is an Ito process without drift, and thus a martingale. This is the reason that $\mathbb{Q}$ is also called the equivalent martingale measure.

- We can extend the BS model by assuming that the underlying pays a continuous dividend at rate δ (realistic only for indices, not individual stocks). Then a position of 1 share generates an instantaneous dividend stream $\delta S_t dt$, in addition to the capital gains dS_t. Note that only the holder of the underlying receives the dividend.
- The pricing formula remains the same, but now the risk-neutral dynamics of S_t are
$$dS_t = (r - \delta)S_t dt + \sigma S_t dW_t.$$
- The expected growth rate of the underlying under $\mathbb{Q}$ is $r - \delta$, so the expected return from holding it (capital gains plus dividend yield) is r.
- The price of a call is now
$$C_t = e^{-\delta(T-t)} S_t \Phi(d_1) - e^{-r(T-t)} K \Phi(d_2),$$
where
$$d_{1,2} = \frac{\log(S_t/K) + [(r - \delta) \pm \frac{1}{2}\sigma^2](T - t)}{\sigma\sqrt{T - t}}.$$

Monte Carlo Pricing

- The goal in Monte Carlo simulation is to obtain an estimate of
$$\theta \equiv \mathbb{E}[X],$$
for some random variable X with finite expectation. The assumption is that we have a means of sampling from the distribution of X, but no closed-form expression for θ.
- Suppose we have a sample $\{X_i\}_{i\in\{1,\ldots,n\}}$ of *independent* draws for X, and let
$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$
- The sample average $\bar{X}_n$ is an *unbiased estimator* of θ: $\mathbb{E}[\bar{X}_n] = \theta$.
- The *weak law of large numbers* states that
$$\bar{X}_n \xrightarrow{p} \theta,$$
where the arrow denotes *convergence in probability*; i.e., as the sample size grows, $\bar{X}_n$ becomes a better and better estimate of θ.

- Thus, our strategy is to use a computer to draw n (pseudo) random numbers X_i from the distribution of X, and then estimate θ as the sample mean of the X_i.
- n is called the number of *replications*.
- For finite n, the sample average will be an approximation to θ.
- It is usually desirable to have an estimate of the accuracy of this approximation. Such an estimate can be obtained from the *central limit theorem* (CLT), which states that

$$\sqrt{n}(\bar{X}_n - \theta) \xrightarrow{d} N\left(0, \sigma^2\right),$$

  provided that σ^2, the variance of X, is finite. The arrow denotes convergence in distribution; this implies that for large n, $\bar{X}_n$ has approximately a normal distribution.
- Of course σ^2 is unknown, but we can estimate it as

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\bar{X}_n - X_i\right)^2.$$

- A 95% *confidence interval* (CI) is an interval $[c_l, c_u]$ with $\mathbb{P}[c_l \leq \theta \leq c_u] = 0.95$.
- The CLT implies that, in the limit as $n \to \infty$,

$$\mathbb{P}[-1.96\sigma \leq \sqrt{n}(\bar{X}_n - \theta) \leq 1.96\sigma] = 0.95 \Leftrightarrow$$
$$\mathbb{P}[\bar{X}_n - 1.96\frac{\sigma}{\sqrt{n}} \leq \theta \leq \bar{X}_n + 1.96\frac{\sigma}{\sqrt{n}}] = 0.95.$$

- Hence $c_l = \bar{X}_n - 1.96\frac{\sigma}{\sqrt{n}}$ and $c_u = \bar{X}_n + 1.96\frac{\sigma}{\sqrt{n}}$ is an asymptotically valid CI.
- Note that c_l and c_u are random variables; we should interpret this as "before the experiment is performed, there is a 95% chance that a CI computed according to this formula will contain θ". After performing the experiment, this statement is not valid anymore; the interval is now fixed, and contains θ with probability either 0 or 1.
- The unknown parameter σ can be consistently estimated by $\sqrt{\hat{\sigma}^2}$.

Application: Asian Options

- The payoff of Asian options depends on the *average* price of the underlying, $\bar{S}_T$. Types:
  - Average price Asian call with payoff $(\bar{S}_T - K)^+$;
  - Average price Asian put with payoff $(K - \bar{S}_T)^+$;
  - Average strike Asian call with payoff $(S_T - \bar{S}_T)^+$;
  - Average strike Asian put with payoff $(\bar{S}_T - S_T)^+$.
- It is important to specify how the average is computed: the continuous, arithmetic, and geometric averages are, respectively,

$$\frac{1}{T}\int_0^T S_t dt, \qquad \frac{1}{N}\sum_{i=1}^N S_{t_i}, \qquad \text{and} \qquad \left(\prod_{i=1}^N S_{t_i}\right)^{1/N},$$

  where the t_i are a set of N specified dates.

- Exact Black-Scholes type pricing formulas for Asian options only exist in special cases (e.g., the geometric average Asian call, see next week), so we rely on Monte Carlo simulation.
- Our pricing formula

$$C_t = e^{-r(T-t)}\,\mathbb{E}^{\mathbb{Q}}\left[C_T \middle| \mathcal{F}_t\right]$$

  is exactly in the required form.
- As an example, consider pricing an arithmetic average price call with payoff

$$C_T = (\bar{S}_T - K)^+, \quad \text{where} \quad \bar{S}_T = \frac{1}{N}\sum_{i=1}^{N} S_{t_i},$$

  which cannot be priced analytically.
- The payoff is path-dependent, so we need to simulate the entire asset price path, not just S_T.

In [7]:
```
from scipy.stats import norm
def asianmc(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Monte Carlo price of an arithmetic average Asian call.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    payoffs = np.zeros(numsim)
    for i in xrange(numsim):
        _, X = bmsim(T, N, X0, nu, sigma)  #Convention: underscore holds value to be discarded.
        S = np.exp(X)
        payoffs[i] = max(S[1:].mean()-K, 0.)
    g = np.exp(-r*T)*payoffs
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C-zq/np.sqrt(numsim)*s
    Cu = C+zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```

In [8]:
```
S0 = 11; K = 10; T = 3/12.; r = 0.02; sigma = .3; delta = 0.01; N = 10
np.random.seed(0)
C0, Cl, Cu = asianmc(S0, K, T, r, sigma, delta, N); C0, Cl, Cu
```

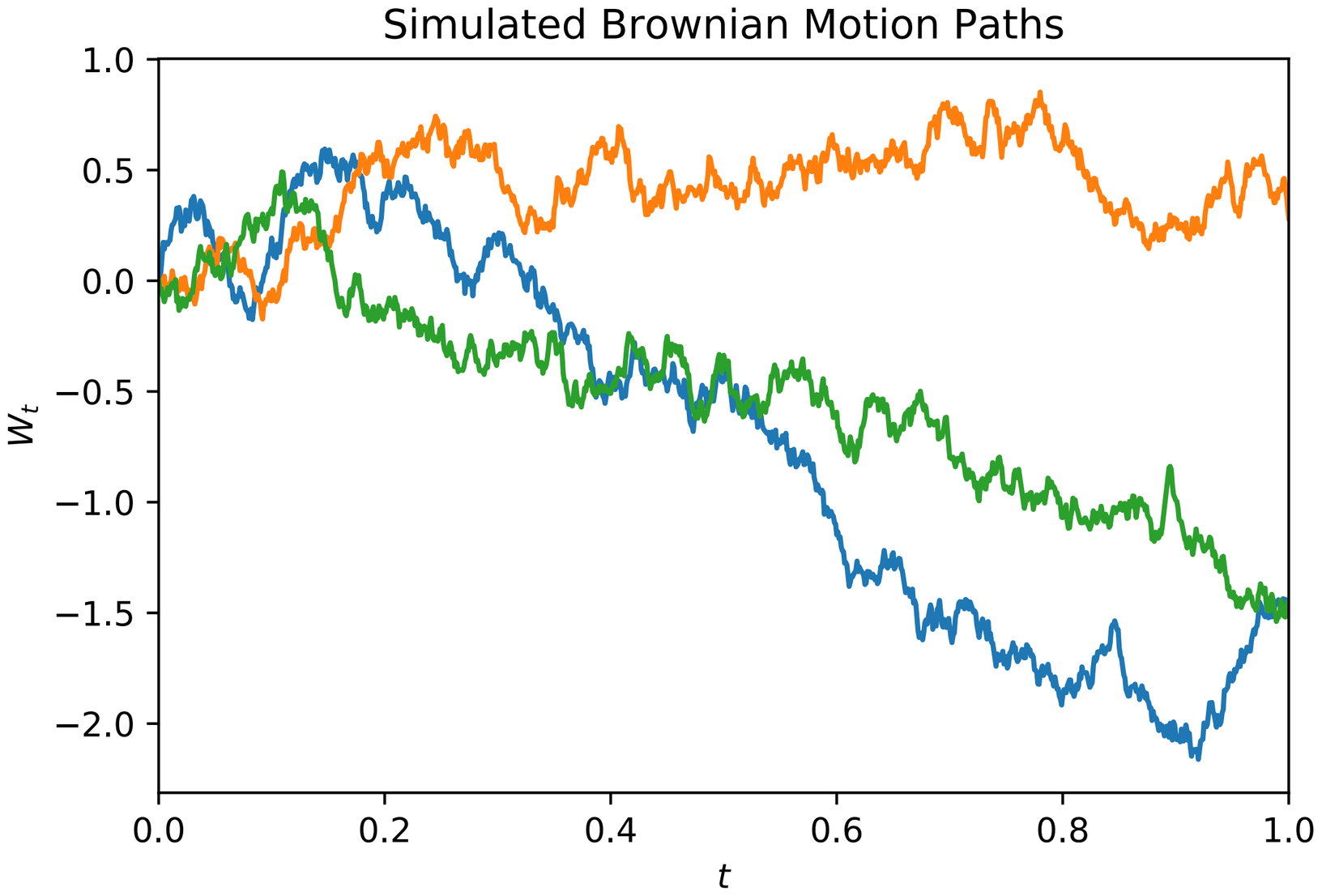
Out[8]:
```
(1.0927262054551385, 1.0747653929130998, 1.1106870179971773)
```

Code Optimization

- Our code for pricing the Asian option is likely inefficient, because it contains a loop.
- The code can be 'vectorized' to speed it up.
- First step: simulate a bunch of Brownian paths in one shot.
- The resulting code is actually almost identical:

In [9]:
```
def bmsim_vec(T, N, X0=0, mu=0, sigma=1, numsim=1):  #Note new input: numsim, the number of paths.
    """Simulate `numsim` Brownian motion paths.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    z = np.random.randn(numsim, N+1)  #(N+1)->(numsim, N+1)
    dX = mu*deltaT + sigma*np.sqrt(deltaT)*z
    dX[:, 0] = 0.  #dX[0]->dX[:, 0]
    X = np.cumsum(dX, axis=1)  #cumsum(dX)->cumsum(dX, axis=1)
    X += X0
    return tvec, X
```

In [10]:
```
np.random.seed(0)
tvec, W = bmsim_vec(1, 1000, numsim=3)
W = pd.DataFrame(W.transpose(), index=tvec)
W.plot().legend().remove()
plt.title('Simulated Brownian Motion Paths')
plt.xlabel("$t$"); plt.ylabel("$W_t$");
plt.savefig("img/BMpaths.svg"); plt.close()
```


Simulated Brownian Motion Paths
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
W_t
0.0
0.2
0.4
0.6
0.8
1.0
t

- Here is the vectorized code for the Asian option:

In [11]:
```
def asianmc_vec(S0, K, T, r, sigma, delta, N, numsim=10000):
    """Monte Carlo price of an arithmetic average Asian call.
    """
    X0 = np.log(S0)
    nu = r-delta-.5*sigma**2
    #simulate all paths at once:
    _, X = bmsim_vec(T, N, X0, nu, sigma, numsim)
    S = np.exp(X)
    payoffs = np.maximum(S[:, 1:].mean(axis=1)-K, 0.)  #S[1:]->S[:, 1:], max->maximum, mean()->mean(axis=1)
    g = np.exp(-r*T)*payoffs
    C = g.mean(); s = g.std()
    zq = norm.ppf(0.975)
    Cl = C - zq/np.sqrt(numsim)*s
    Cu = C + zq/np.sqrt(numsim)*s
    return C, Cl, Cu
```

- Let's see if it works:

In [12]:
```
np.random.seed(0)
C0_vec, _, _ = asianmc_vec(S0, K, T, r, sigma, delta, N)
np.allclose(C0_vec, C0)
```

Out[12]: True

- And time it:

In [13]:
```
%timeit asianmc(S0, K, T, r, sigma, delta, N)
```

```
1 loop, best of 3: 376 ms per loop
```

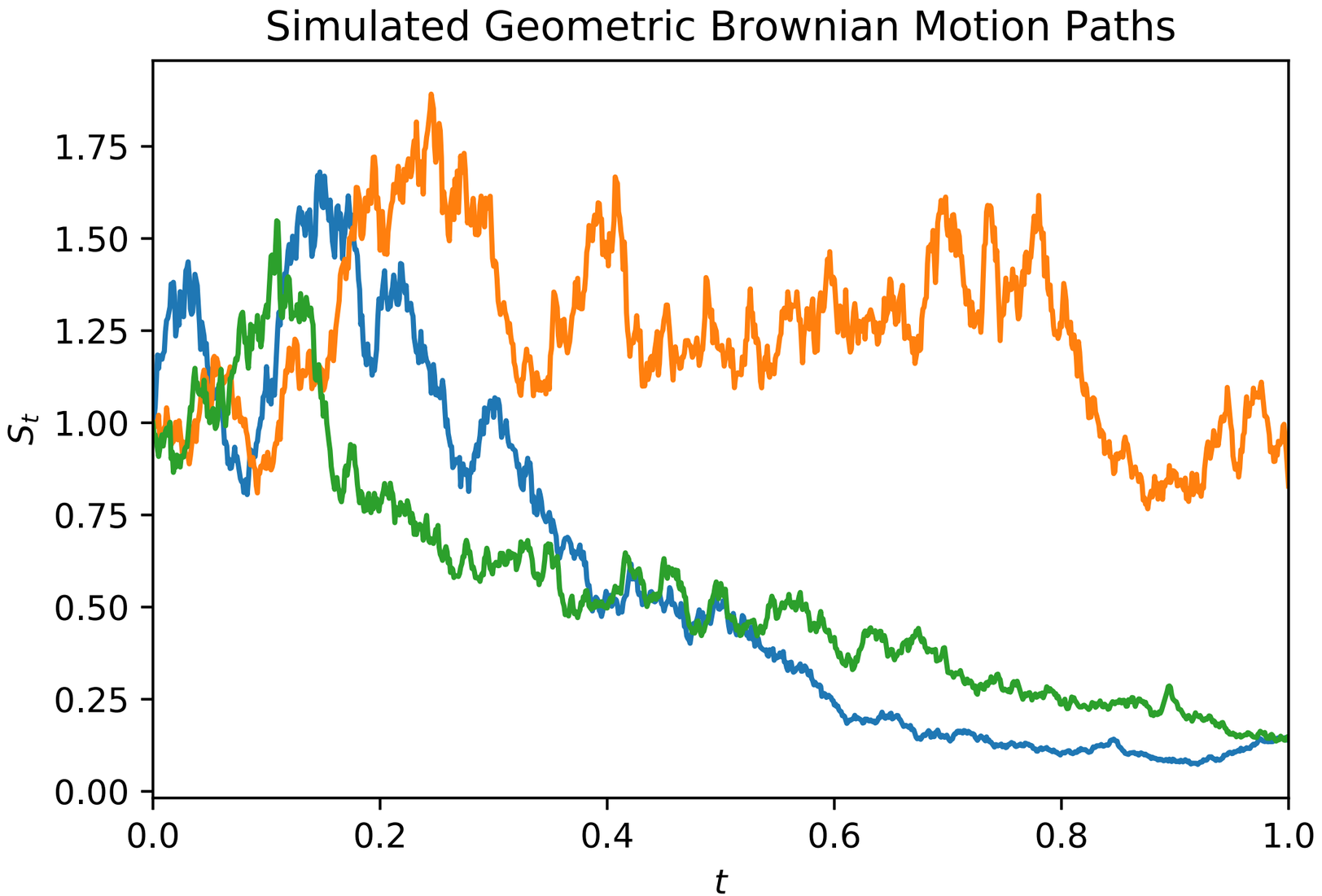
In [14]:
```
%timeit asianmc_vec(S0, K, T, r, sigma, delta, N)
```

```
100 loops, best of 3: 5.93 ms per loop
```

- Our code for the Euler scheme can likewise be adjusted to compute many paths in one shot.
- We're still stuck with the loop over t though, which cannot be vectorized because S_{i+1} depends on S_i.
- We'll use Numba's JIT compiler to speed it up further.

In [15]:
```
from numba import jit
@jit(nopython=True)
def gbmsim_vec(T, N, S0=1, mu=0, sigma=1, numsim=1, seed=0):
    """Simulate `numsim` Geometric Brownian motion paths.
    """
    deltaT = float(T)/N
    tvec = np.linspace(0, T, N+1)
    np.random.seed(seed)  #Note: with jit-compiled functions, the RNG must be seeded INSIDE the compiled code.
    z = np.random.randn(numsim, N+1)
    S = np.zeros_like(z)
    S[:, 0] = S0
    for i in xrange(0, N):
        S[:, i+1]=S[:, i] + mu*S[:, i]*deltaT + sigma*S[:, i]*np.sqrt(deltaT)*z[:, i+1]
    return tvec, S
```

In [16]:
```
tvec, S = gbmsim_vec(1, 1000, numsim=3, seed=0)
S = pd.DataFrame(S.transpose(), index=tvec)
S.plot().legend().remove()
plt.title('Simulated Geometric Brownian Motion Paths')
plt.xlabel("$t$"); plt.ylabel("$S_t$")
plt.savefig("img/GBMpaths.svg"); plt.close()
```


Simulated Geometric Brownian Motion Paths
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
S_t
0.0
0.2
0.4
0.6
0.8
1.0
t

- The compiled code produces the same results:

In [17]:
```
np.random.seed(0)
_, S1 = gbmsim(1, 1000)
_, S2 = gbmsim_vec(1, 1000, seed=0)
np.allclose(S1, S2)
```

Out[17]: True

- But it is quite a bit faster:

In [18]:
```
%%timeit #Cell magic (for timing the entire cell).
for k in xrange(10):  #10 paths.
    gbmsim(1, 1000)
```

```
10 loops, best of 3: 23.3 ms per loop
```

In [19]:
```
%timeit gbmsim_vec(1, 1000, numsim=10)
```

```
The slowest run took 553.56 times longer than the fastest. This could mean that an intermediate result is bei
ng cached.
1 loop, best of 3: 570 µs per loop
```