Assignment 11

Reid Ginoza

4/26/2020

Scalar SDE Analytical Solution

This is Exercise 8.1 from (Särkkä and Solin 2019). We are given the following SDE:

$$\mathrm{d}x = -c\,x\mathrm{d}t + g\,x\mathrm{d}\beta, \quad x(0) = x_0, \tag{1}$$

where c, g, and x_0 are positive constants and $\beta(t)$ is a standard Brownian motion.

We will use the Itô chain rule, where the drift function is $F(x) = -cx$ and the dispersion function $G(x) = gx$. We want to solve for $u(x) = \ln(x)$.

$$u_t = 0 \tag{2}$$

$$u_x = \frac{1}{x} \tag{3}$$

$$u_{xx} = -\frac{1}{x^2} \tag{4}$$

$$\implies \mathrm{d}u = \left(u_t + u_x F + \frac{1}{2}u_{xx}G^2\right)\mathrm{d}t + u_x\,G\,\mathrm{d}\beta \tag{5}$$

$$= \left(0 - \frac{cx}{x} - \frac{g^2x^2}{2x^2}\right)\mathrm{d}t + \frac{gx}{x}\,\mathrm{d}\beta \tag{6}$$

$$= \left(-c - \frac{g^2}{2}\right)\mathrm{d}t + g\,\mathrm{d}\beta \tag{7}$$

$$\implies u = \left(-c - \frac{g^2}{2}\right)t + g\beta + C \tag{8}$$

$$\ln(x) = \left(-c - \frac{g^2}{2}\right)t + g\beta + C \tag{9}$$

$$\implies x = C\exp\left[\left(-c - \frac{g^2}{2}\right)t + g\beta\right] \tag{10}$$

$$\text{Applying the initial condition,} \qquad x = x_0\exp\left[\left(-c - \frac{g^2}{2}\right)t + g\beta\right] \tag{11}$$

Numerical Solution

We will use the Milstein method for the an SDE in this form:

$$\mathrm{d}x = F(x)\mathrm{d}t + G(x)\mathrm{d}\beta, \tag{12}$$

where x is a scalar, β is the standard Brownian motion, and we given the initial condition $x(0) = x_0$. Given the Brownian motion step $\Delta\beta$, or having drawn it from $\Delta\beta \sim \mathrm{N}(0, \Delta t)$, the Milstein calculates each time step $\hat{x}$ as:

$$\hat{x}(t_{k+1}) = \hat{x}(t_k) + F(\hat{x}(t_k))\Delta t + G(\hat{x}(t_k))\Delta\beta_k + \frac{1}{2}\frac{\partial\,G(\hat{x}(t_k))}{\partial x}G(\hat{x}(t_k))(\Delta\beta_k^2 - \Delta t) \tag{13}$$

We will examine the case where $x_0 = 1$, $c = 0.1$, $g = 0.1$ with the Milstein method. The SDE, therefore, is

$$\mathrm{d}x = -0.1x\mathrm{d}t + 0.1x\mathrm{d}\beta, \quad x(0) = 1. \tag{14}$$

and the known analytical solution is:

$$x = \exp\left[-0.105t + 0.1\beta\right] \tag{15}$$

and the Milstein step, given the Brownian motion step $\Delta\beta_k$ is:

$$\hat{x}(t_{k+1}) = \hat{x}(t_k) - 0.1\hat{x}(t_k)\Delta t + 0.1\hat{x}(t_k)\Delta\beta_k + 0.005\hat{x}(t_k)(\Delta\beta_k^2 - \Delta t) \tag{16}$$

```
import numpy as np
from matplotlib import pyplot as plt

# -- Brownian Motion --
def multiple_brownian_motion(end_time=1., num_tsteps=500, n_trials=1000):
    """Creates multiple 1-D Brownian motion with time as the row index and
    each column as a separate path of Brownian motion.

    This assumes that all Brownian motion starts at 0. Currently only
    implements one-dimensional Brownian motion. This also assumes all
    step sizes are the same size.

    The steps of Brownian motion, ``dw``, are modeled with a Gaussian
    distribution with mean 0 and variance ``sqrt(dt)``, where ``dt``
    is the constant time step size.

    Parameters
    ----------
    end_time : float

    num_tsteps : int
        The number of steps to take. Will calculate the step
        size dt internally. The number of rows in the output of
        Brownian motion will be num_tsteps + 1.

    n_trials : int
        The number of sample paths to create. This will be the number
        of columns in the output.

    Returns
    -------
    t : ndarray
        One-dimensional time ndarray from 0 to ``end_time`` with
        shape (``num_tsteps``+1,)

    w : ndarray
        Two-dimensional ndarray representing ``n_trials`` number of
        sample paths of one-dimensional Brownian motion.
        This will be of shape (``num_tsteps``+1, ``n_trials``).

    dt : float
        The value indicating the step size of t. This is only implemented
        with constant step size.
```

```
    dw : ndarray
        Two-dimensional ndarray representing the steps of Brownian motion.
        The first row is all zeros. Each i-th row of ``dw``, ie. dw[i, :]
        indicates the change in ``w`` from w[i-1, :] to w[i, :]
        This will be the same shape as ``w``, (``num_tsteps``+1, ``n_trials``).

    """

    dt = (end_time - 0) / num_tsteps
    dw = np.random.normal(scale=np.sqrt(dt), size=(num_tsteps+1, n_trials))
    # Brownian motion must start at time 0 with value 0
    dw[0] = np.zeros_like(dw[0])
    w = dw.cumsum(axis=0)
    # t is not used in calculations, but returned to allow user to keep track
    # of points in time
    t = np.linspace(0, end_time, num=num_tsteps+1).reshape((num_tsteps+1, 1))
    assert w.shape[0] == t.shape[0], ('time and position arrays are not the '
                                      'same length. w.shape[0] - t.shape[0] = '
                                      f'{w.shape[0] - t.shape[0]}')
    assert w.shape == dw.shape, ('position and velocity arrays are not the '
                                 'same shape: '
                                 f'w.shape: {w.shape}    dw.shape: {dw.shape}')
    return t, w, dt, dw


def euler_maruyama_nonlinear_vec(f, g, x0, t, dt, dw, M):
    """
    calculates the EM approximation on the nonlinear one-dimensional SDE,
    vectorized for multiple trials based on the shape of ``dw``.

    SDE is of the form:
    dX = f(X)dt + G(X)dW; X(0) = X0

    :param f: shift function in SDE. Must be passed as a function of x
    :param g: drift/dispersion function in SDE
    :param x0: Initial condition
    :param t: time one dimensional nd-array
    :param dt: step size of time array, float
    :param dw: White noise associated with the Brownian motion, ndarray
    :param M: multiple of dt for Euler-Maruyama step size. Do not make this too large.
    :return: time array and solution x array
    """

    if M < 1:
        raise ValueError('M must be greater than or equal to 1')

    Dt = M * dt  # EM step size
    L = (t.shape[0] - 1) / M  # number of EM steps

    if not L.is_integer():
        raise ValueError('Cannot handle Step Size that is not a multiple of M')

    L = int(L)  # needed for range below
```

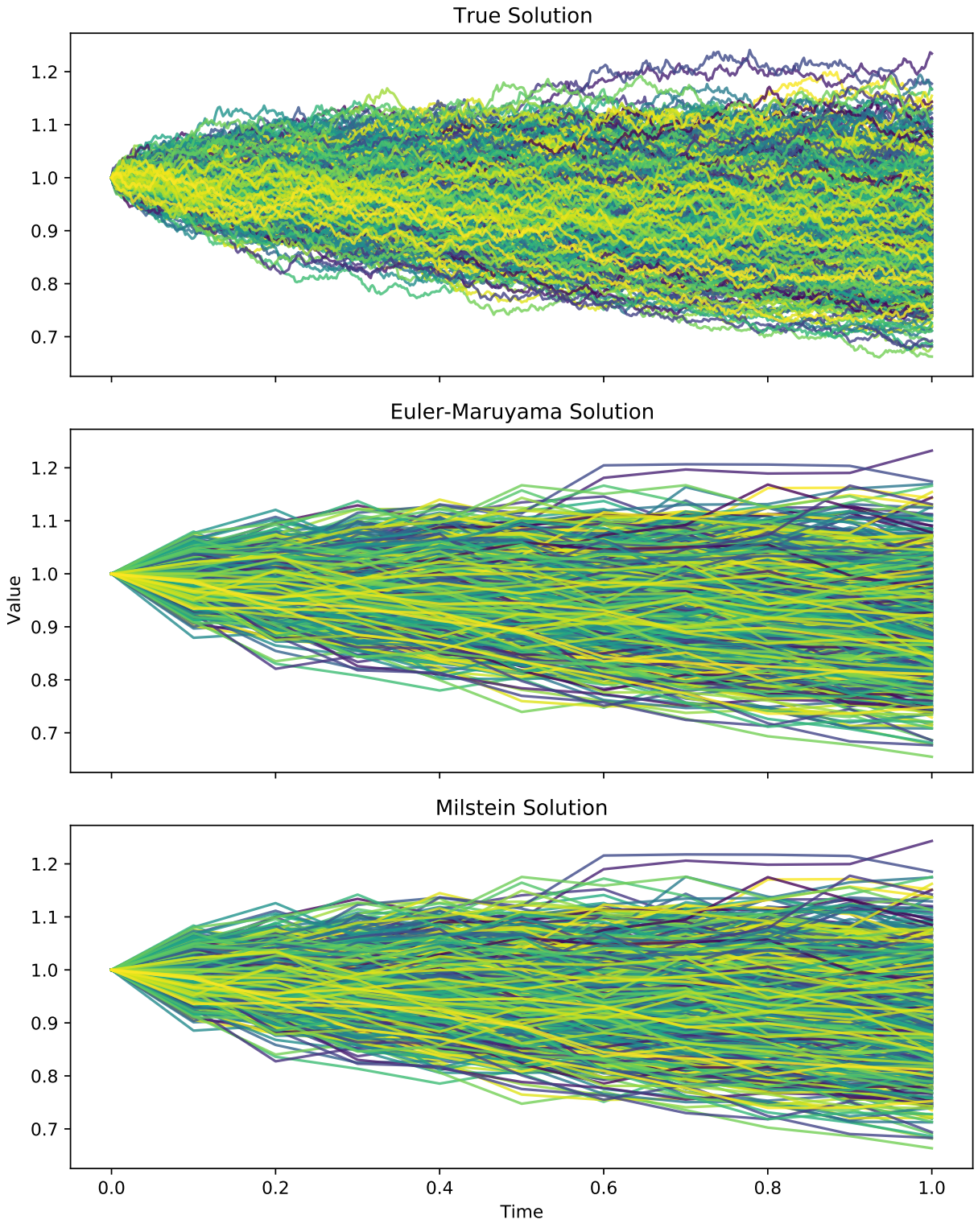
```
    x = [np.full((dw.shape[1],), x0)]
    T = [0]
    for i in range(1, L+1):
        # DW is the step of Brownian motion for EM step size
        DW = (dw[M * (i - 1) + 1:M * i + 1, :]).sum(axis=0).reshape(dw.shape[1], )
        x.append(x[i-1] + Dt * f(x[i-1]) + g(x[i-1]) * DW)
        T.append(T[i-1] + Dt)

    return np.array(T), np.array(x)


def milstein(f, g, dg, x0, t, dt, dw, M):
    if M < 1:
        raise ValueError('M must be greater than or equal to 1')

    Dt = M * dt  # EM step size
    L = (t.shape[0] - 1) / M  # number of EM steps

    if not L.is_integer():
        raise ValueError('Cannot handle Step Size that is not a multiple of M')

    L = int(L)  # needed for range below

    x = [np.full((dw.shape[1],), x0)]
    T = [0]
    for i in range(1, L+1):
        # DW is the step of Brownian motion for EM step size
        DW = (dw[M * (i - 1) + 1:M * i + 1, :]).sum(axis=0).reshape(dw.shape[1], )
        x.append(x[i-1] + Dt * f(x[i-1]) + g(x[i-1]) * DW
                 + 0.5 * dg(x[i-1]) * g(x[i-1]) * (DW**2 - dt))
        T.append(T[i-1] + Dt)

    return np.array(T), np.array(x)


# -- Input Deck --
def f(x):
    return - 0.1 * x


def g(x):
    return 0.1 * x


def dg(x):
    del x  # unused
    return 0.1


# Initial Condition
x0 = 1

# Known Analytical Solution
```

```
def x_true(t, w):
    return np.exp(-0.105*t + 0.1*w)


# simulation variables
END_TIME = 1
NUM_TSTEPS = 500
N_TRIALS = 1000
APPROX_STEP = 50

# plotting
color_rotate = 10
color_num = int(N_TRIALS/color_rotate)
colors = plt.get_cmap('viridis', lut=color_num)
```

Sample Paths

This section shows sample paths generated by the true solution given 1000 samples of brownian motion, and the approximations from the Euler Maruyama solution and Milstein solution, both using only 0.2 times the number of steps as the true solution.

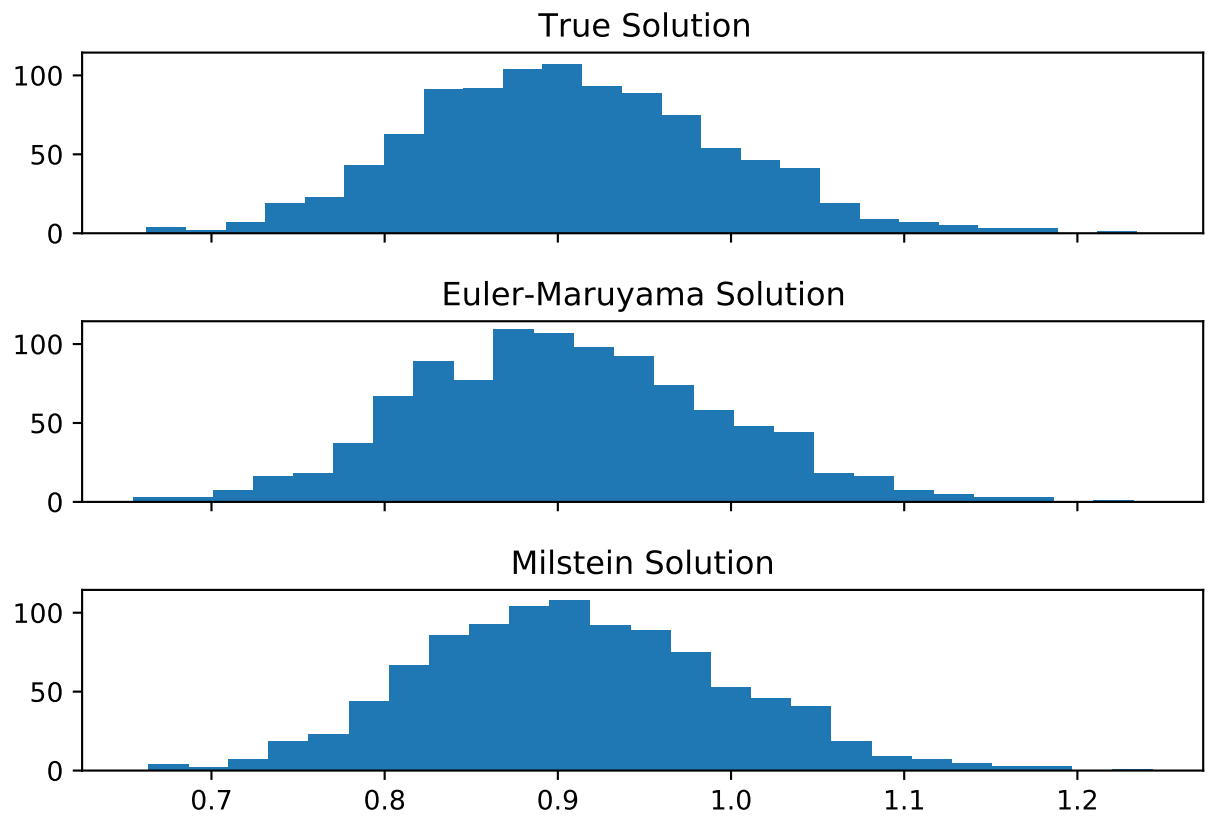
```
# start run
t, w, dt, dw = multiple_brownian_motion(END_TIME, NUM_TSTEPS, N_TRIALS)
full_true = x_true(t, w)

t_em, x_em = euler_maruyama_nonlinear_vec(f, g, x0, t, dt, dw, M=APPROX_STEP)
t_mil, x_mil = milstein(f, g, dg, x0, t, dt, dw, M=APPROX_STEP)

# plot true solution

fig_path, ax_path = plt.subplots(nrows=3, sharex=True, sharey=True, figsize=(8, 10))
for idx, trial in enumerate(full_true.transpose()):
    ax_path[0].plot(t, trial, color=colors(idx%color_num), alpha=0.8)
ax_path[0].set_title('True Solution')

for idx, trial in enumerate(x_em.transpose()):
    ax_path[1].plot(t_em, trial, color=colors(idx%color_num), alpha=0.8)
ax_path[1].set_ylabel('Value')
ax_path[1].set_title('Euler-Maruyama Solution')

for idx, trial in enumerate(x_mil.transpose()):
    ax_path[2].plot(t_mil, trial, color=colors(idx%color_num), alpha=0.8)
ax_path[2].set_title('Milstein Solution')
ax_path[2].set_xlabel('Time')
# fig_path.subplots_adjust(top=0.88)
# fig_path.suptitle('Paths')
plt.tight_layout()
plt.show()
```


True Solution
Euler-Maruyama Solution
Milstein Solution
Value
Time
1.2
1.1
1.0
0.9
0.8
0.7
0.0
0.2
0.4
0.6
0.8
1.0

Distribution at the End Time

Now we show the distributions at the end time $t = 1$. Surprisingly, even with the large time steps, both the Euler-Maruyama and the Milstein method perform well in matching the distribution.

```
# Look at end_time
fig, ax = plt.subplots(nrows=3, sharex=True, sharey=True)
h0 = ax[0].hist(full_true[-1:].flatten(), bins='auto')
ax[0].set_title('True Solution')
h1 = ax[1].hist(x_em[-1:].flatten(), bins='auto')
ax[1].set_title('Euler-Maruyama Solution')
h2 = ax[2].hist(x_mil[-1:].flatten(), bins='auto')
ax[2].set_title('Milstein Solution')
# fig.subplots_adjust(top=0.88)
# fig.suptitle(f'Distribution at Time $t={END_TIME}$')
plt.tight_layout()
plt.show()
```



References

Särkkä, Simo, and Arno Solin. 2019. *Applied Stochastic Differential Equations*. Vol. 10. Cambridge University Press.