Assignment 10

Reid Ginoza

4/19/2020

Ornstein-Uhlenbeck Maximum Likelihood Estimate

The analytical problem is to derive the maximum likehood estimate for the Ornstein-Uhlenbeck process. This is Exercise 11.4 from (Särkkä and Solin 2019), and we are guided by the authors' instructions. The Ornstein-Uhlenbeck process is defined as follows:

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

where λ is unknown and β has an unknown diffiusion constant q. Thus, we define the unknown parameters as the vector $\theta = (\lambda, q)$. The authors also previously provided the transition density:

$$p\left(x\left(t+\Delta t\right)|x\left(t\right)\right) = \mathrm{N}\left(x\left(t+\Delta t\right)|\exp\left(-\lambda\Delta t\right)x\left(t\right), \frac{q}{2\lambda}\left[1-\exp\left(-2\lambda\Delta t\right)\right]\right) \tag{2}$$

The negative log-likelihood is:

$$\ell(\lambda, q) = \sum_{k=0}^{T-1}\left[\frac{1}{2}\log\left(2\pi\frac{q}{2\lambda}\left[1-\exp\left(-2\lambda\Delta t\right)\right]\right) + \frac{\lambda}{q\left[1-\exp\left(-2\lambda\Delta t\right)\right]}\left(x(t_{k+1}) - \exp\left(-\lambda\Delta t\right)x(t_k)\right)^2\right] \tag{3}$$

and using this change of variables:

$$a = \exp\left(-\lambda\Delta t\right) \tag{4}$$

$$\Sigma = \frac{q}{2\lambda}\left[1-\exp\left(-2\lambda\Delta t\right)\right] \tag{5}$$

we arrive at:

$$\ell(a, \Sigma) = \sum_{k=0}^{T-1}\left[\frac{1}{2}\log\left(2\pi\Sigma\right) + \frac{1}{2\Sigma}\left(x(t_{k+1}) - ax(t_k)\right)^2\right] \tag{6}$$

To find the maximum likelihood, we find where ℓ has a gradient of 0.

$$\frac{\partial \ell}{\partial a} = -\frac{1}{\Sigma} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a x(t_k) \right) x(t_k) = 0 \tag{7}$$

$$\implies 0 = \sum_{k=0}^{T-1} x(t_{k+1}) x(t_k) - a \sum_{k=1}^{T-1} \left(x(t_k) \right)^2 \tag{8}$$

$$\implies a_{\mathrm{ML}} = \frac{\sum_{k=0}^{T-1} x(t_{k+1}) x(t_k)}{\sum_{k=1}^{T-1} \left(x(t_k) \right)^2} \tag{9}$$

$$\tag{10}$$

$$\frac{\partial \ell}{\partial \Sigma} = \sum_{k=0}^{T-1} \left[\frac{1}{2\Sigma} - \frac{1}{2\Sigma^2} \left(x(t_{k+1}) - a x(t_k) \right)^2 \right] = 0 \tag{11}$$

$$0 = \frac{T}{2\Sigma} - \frac{1}{2\Sigma^2} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a x(t_k) \right)^2 \tag{12}$$

$$\implies \frac{T}{2\Sigma} = \frac{1}{2\Sigma^2} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a x(t_k) \right)^2 \tag{13}$$

$$\implies T\Sigma = \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a x(t_k) \right)^2 \tag{14}$$

$$\implies \Sigma_{\mathrm{ML}} = \frac{1}{T} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a_{\mathrm{ML}} x(t_k) \right)^2 \tag{15}$$

And to reverse the change of variables, we used:

$$a_{\mathrm{ML}} = \exp\left(-\lambda_{\mathrm{ML}} \Delta t \right) \tag{16}$$

$$\Sigma_{\mathrm{ML}} = \frac{q_{\mathrm{ML}}}{2\lambda} \left[1 - \exp\left(-2\lambda_{\mathrm{ML}} \Delta t \right) \right] \tag{17}$$

which gives

$$a_{\mathrm{ML}} = \frac{\sum_{k=0}^{T-1} x(t_{k+1}) x(t_k)}{\sum_{k=1}^{T-1} \left(x(t_k) \right)^2} \tag{18}$$

$$\exp\left(-\lambda_{\mathrm{ML}} \Delta t \right) = \frac{\sum_{k=0}^{T-1} x(t_{k+1}) x(t_k)}{\sum_{k=1}^{T-1} \left(x(t_k) \right)^2} \tag{19}$$

$$\lambda_{\mathrm{ML}} = -\frac{1}{\Delta t} \log \left[\frac{\sum_{k=0}^{T-1} x(t_{k+1}) x(t_k)}{\sum_{k=1}^{T-1} \left(x(t_k) \right)^2} \right] \tag{20}$$

$$\tag{21}$$

$$\Sigma_{\mathrm{ML}} = \frac{1}{T} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a_{\mathrm{ML}} x(t_k) \right)^2 \tag{22}$$

$$\frac{q_{\mathrm{ML}}}{2\lambda} \left[1 - \exp\left(-2\lambda_{\mathrm{ML}} \Delta t \right) \right] = \frac{1}{T} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - a_{\mathrm{ML}} x(t_k) \right)^2 \tag{23}$$

$$\implies q_{\mathrm{ML}} = \frac{2\lambda_{\mathrm{ML}}}{T \left[1 - \exp\left(-2\lambda_{\mathrm{ML}} \Delta t \right) \right]} \sum_{k=0}^{T-1} \left(x(t_{k+1}) - \exp\left(-\lambda_{\mathrm{ML}} \Delta t \right) x(t_k) \right)^2 \tag{24}$$

Computational Problem

This is problem 11.9 from (Särkkä and Solin 2019) and concerns a model where two parameters θ_1 and θ_2 are unknown:

$$\mathrm{d}x = \theta_1 \sin(x - \theta_2) + \mathrm{d}\beta, \quad x(0) = x_0, \tag{25}$$

where β is a standard Brownian motion (i.e. the diffusion coeffient $q = 1$).

I was restricted in the number of time steps based on the product required to calculate the negative log-likihood. I also only sampled one path. As a result, the numerical method did not find the true parameters very well.

First, I'll show the Euler-Maruyama approximation of the true function. Then I will show the optimal parameters based on the maximum likelihood method based on one sample path.

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.optimize import minimize
from scipy.stats import norm


# -- Plotting Tools --
def low_buff(y_values):
    span = y_values.max() - y_values.min()
    return y_values.min() - 0.1 * span


def plot_on_axis(ax, time, pos, cols, title, color_map, with_mean=False):
    for idx, col in enumerate(cols):
        ax.plot(time, pos[:, col], c=color_map(idx), alpha=0.5)
    ax.set_title(title)
    if with_mean:
        ax.plot(time, pos.mean(axis=1), color='black',
            label=r'Sample Mean $(n={})$'.format(pos.shape[1]), linewidth=2)
        ax.plot(time, pos.mean(axis=1) + 2* pos.std(axis=1), color='black',
                linestyle='dashed')
        ax.plot(time, pos.mean(axis=1) - 2* pos.std(axis=1), color='black',
                linestyle='dashed', label='Sample Two Std. Dev.')
        ax.legend()


# -- 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):
    """
```

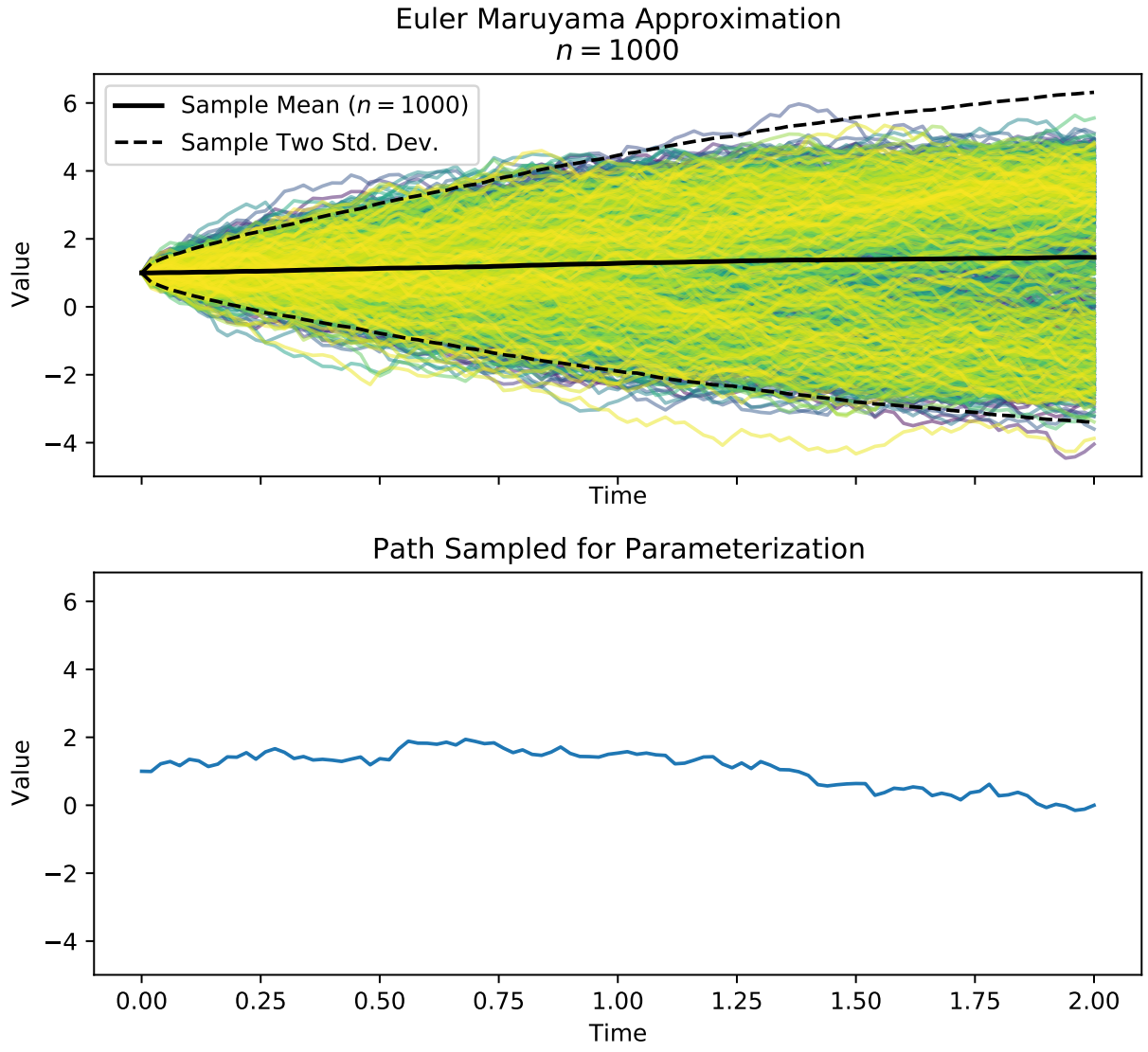
```
    calculates the EM approximation on the nonlinear one-dimensional SDE,
    vectorized for multiple trials based on the shape of ``dw``.

    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 approx_llh(sample, dt, f, g, theta):
    loc = sample[:-1] + f(sample[:-1], theta) * dt
    scale = np.sqrt(g(sample[:-1], theta) * dt)
    return norm(loc=loc, scale=scale).pdf(sample[1:]).prod()


def ell(theta, llh, sample, dt, f, g):
    return - (np.log(llh(sample, dt, f, g, theta))).sum()


# == INPUT DECK ==
# -- True Process --
"""
True Process:
dx = THETA_1 sin(x - THETA_2)dt + dB
```

```
needs to be in the form:
dx = f(x) dt + g(x) dB
"""
```

```
## '\nTrue Process:\ndx = THETA_1 sin(x - THETA_2)dt + dB\nneeds to be in the form:\ndx = f(x) dt + g(x
```

```
THETA_1 = 1.5
THETA_2 = np.pi/4


def f(x, theta):
    theta_1 = theta[0]
    theta_2 = theta[1]
    return theta_1 * np.sin(x - theta_2)


def g(x, theta):
    del x, theta  # unused
    return 1


def known_f(x):
    return f(x, (THETA_1, THETA_2))


def known_g(x):
    return g(x, (THETA_1, THETA_2))


# Initial Condition
x0 = 1

# Only one trial for the sample, but creating multiple trials
# to study the true solution
END_TIME = 2
NUM_TSTEPS = 100
N_TRIALS = 1000
M = 1



# -- Plotting Variables
viridis_em = plt.get_cmap('viridis', lut=N_TRIALS)


# -- Start run --
# -- Generate Samples using Euler-Maruyama Method
t, w, dt, dw = multiple_brownian_motion(END_TIME, NUM_TSTEPS, N_TRIALS)
t_em, x_em = euler_maruyama_nonlinear_vec(known_f, known_g, x0, t, dt, dw, M)
fig, ax = plt.subplots(nrows=2, sharex=True, sharey=True, figsize=(7, 6.5))
plot_on_axis(ax[0], t_em, x_em, np.arange(N_TRIALS), 'Euler Maruyama Approximation\n'
                                                     r'$n=' f'{N_TRIALS}'+r'$',
             color_map=viridis_em, with_mean=True)
```

```
ax[0].set_xlabel('Time')
ax[0].set_ylabel('Value')

ax[1].plot(t_em, x_em[:, 1])
ax[1].set_title('Path Sampled for Parameterization')
ax[1].set_xlabel('Time')
ax[1].set_ylabel('Value')

fig.tight_layout()
plt.show()
```

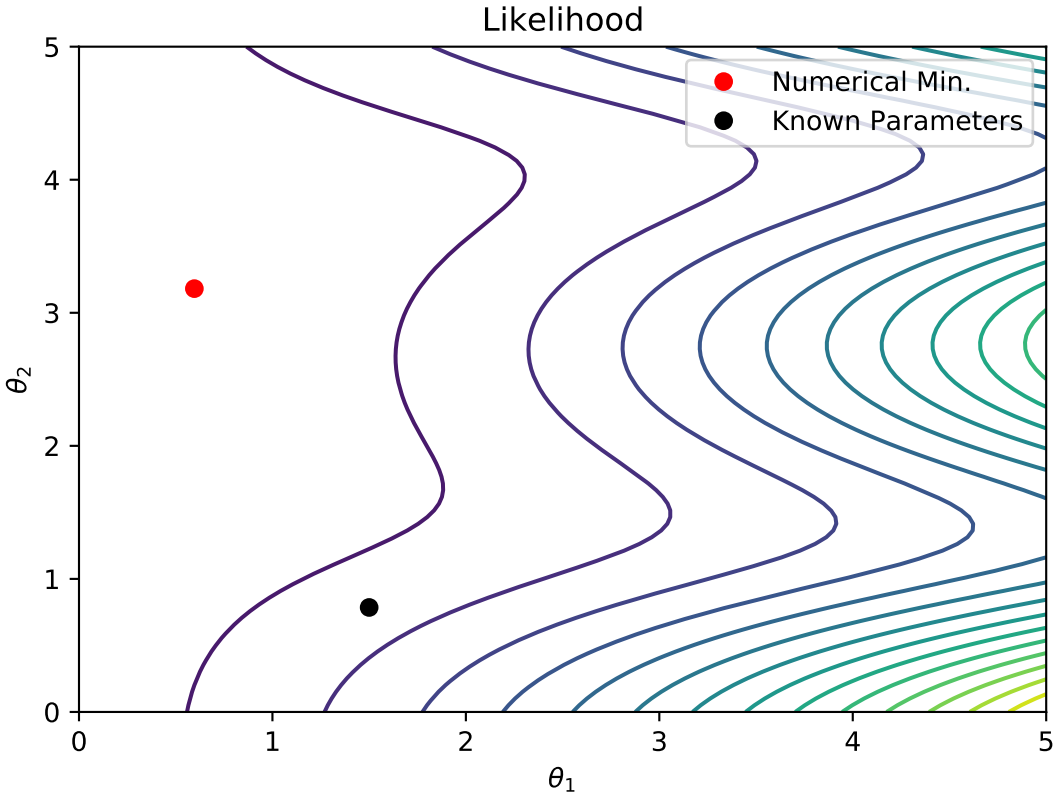



```
sample_time = t_em
sample_value = x_em[:, 1].copy()


# -- Approximate Likelihood
```

```
def approx_ell(theta):
    return ell(theta, approx_llh, sample_value, dt, f, g)


ml_res = minimize(approx_ell, x0=(0.5, 3), bounds=[(0, 5), (0, 5)])

plt.figure()
plot_theta_1 = np.linspace(0, 5, 100)
plot_theta_2 = np.linspace(0, 5, 100)
th_1_xx, th_2_yy = np.meshgrid(plot_theta_1, plot_theta_2)
plot_likelihood = np.zeros_like(th_2_yy)

for x_idx, th_1 in enumerate(plot_theta_1):
    for y_idx, th_2 in enumerate(plot_theta_2):
        plot_likelihood[y_idx, x_idx] = approx_ell((th_1, th_2))

plt.contour(th_1_xx, th_2_yy, plot_likelihood, levels=20)
```

```
## <matplotlib.contour.QuadContourSet object at 0x1249bb8b0>
```

```
plt.xlabel(r'$\theta_1$')
plt.ylabel(r'$\theta_2$')
plt.plot(ml_res.x[0], ml_res.x[1], 'ro', label='Numerical Min.')
plt.plot(THETA_1, THETA_2, 'ko', label='Known Parameters')
plt.legend()
plt.title('Likelihood')
```



References

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