Langevin Equation Test Case

May 15, 2019

1 Langevin Equation Test Case

1.1 Producing the data

For this notebook we consider the stochastic overdamped Langevin equation with a double-welled potential $V(x) = \frac{1}{4}(x^2 - 1)^2$. So, we consider the following SDE.

$$dX_t = -X_t(X_t^2 - 1)dt + dB_t$$

To find a numerical solution we first use the Euler-Maruyama scheme

$$X_{n+1} = X_n - hX_n(X_n^2 - 1) + \nu_n$$

here

$$\nu_n \sim (B_{t_{n+1}} - B_{t_n}) \sim N(0, h) \sim \sqrt{h}N(0, 1).$$

Now we generate the sample path.

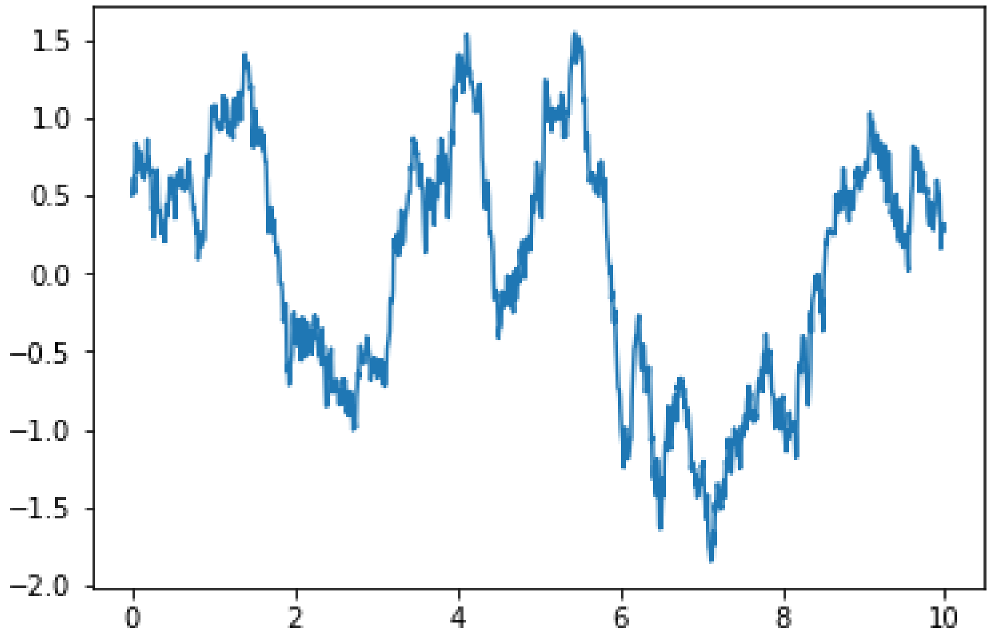
```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import scipy.linalg as sa
        import scipy.optimize as so
```

```
In [2]: start = 0
        stop = 10**4
        steps = stop*100 + 1

        dVdx = lambda x : -x*(x**2 -1)

        x_init = .5
```

```
In [3]: h = (stop - start)/(steps - 1)
        t = np.linspace(start,stop,steps)

        X = np.zeros(steps)
        X[0] = x_init

        for n in range(steps - 1):
            X[n+1] = X[n] + h*dVdx(X[n]) + np.sqrt(h)*np.random.normal(0,1)
```

```
In [4]: plt.plot(t[:1000],X[:1000])
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x1fecf0b5048>]
```



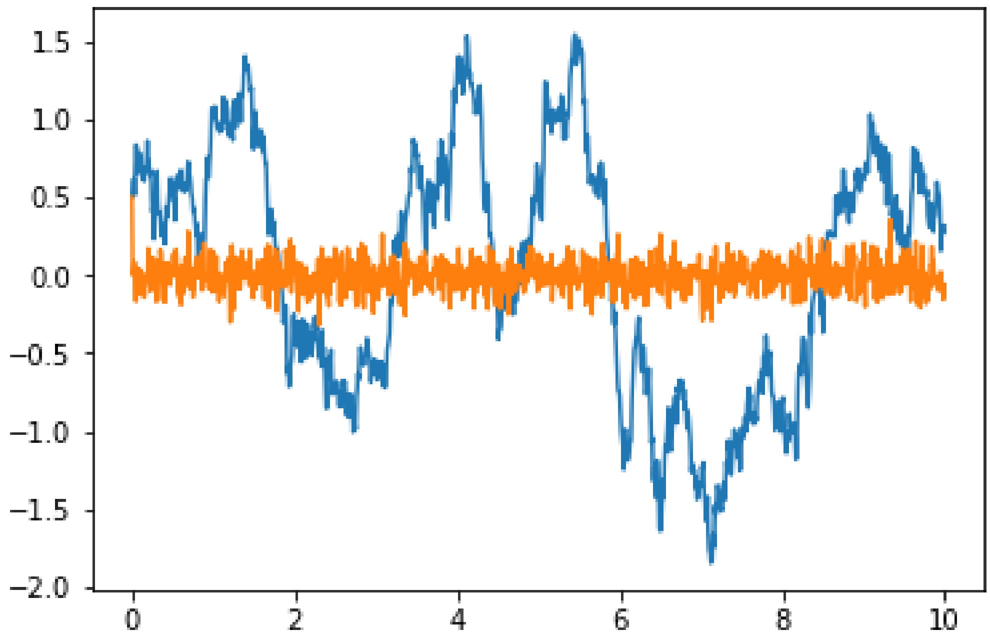

```
In [5]: def sim(X,Psi,A_init):
            N = len(X)

            A_sol = A_init

            # Then we run it
            a = A_sol[0]
            b = A_sol[1:]

            Y = np.zeros(N)
            y = np.zeros(N-1)
            Y[:1] = X[:1]
            y[0] = X[1]
            for i in range(1,N-1):
                Y[i+1] = y[i] + np.sqrt(abs(b[1]))*np.random.normal(0,1)
                y[i] = -a*y[i-1] + b[1]*Psi(Y[i]) + b[0]*Psi(Y[i-1])
            return Y
```

The graph below display two realizations of the same stochastic process.

```
In [7]: Psi = dVdx
        A_init = [-1,0,h]
```

```
Y = sim(X,Psi,A_init)
plt.plot(t[:1000],X[:1000])
plt.plot(t[:1000],Y[:1000])
```

Out[7]: [<matplotlib.lines.Line2D at 0x1fecf0e4828>]



1.2 Proposing the test

We seek to fit something of the form

$$
\begin{aligned}
Y_{n+1} &= y_n + \xi_{n+1} && (1)\\
y_n &= -a y_{n-1} + b_0 \Psi(Y_{n-1}) + b_1 \Psi(Y_n) && (2)\\
& && (3)
\end{aligned}
$$

where $\Psi(x) = -\frac{dV}{dx} = -x(x^2-1)$, $Y_{0:1} = X_{0:1}$, and $y_0 = X_1$. This coresponds to a NARMAX model.

To fit we use for our loss function the mean squared one-step error:

$$\frac{1}{N}\sum_{n=1}^{N} \|X_n - y_{n-1}\|^2.$$

So that the mimimization problem may be rendered as follows:

$$\min_{a,b} \frac{1}{N} \sum_{n=1}^{N} \|X_n - y_{n-1}\|^2 \tag{4}$$

$$\text{s.t. } y_0 = X_1 \tag{5}$$

$$y_n = -ay_{n-1} + b_1\Psi(X_n) + b_0\Psi(X_{n-1}) \qquad \text{for } n = 1, \ldots, N-1 \tag{6}$$

$$\tag{7}$$

It can be shown that with this form above if $a = -1$, $b = (b_0, b_1) = (0, h)$ and $\xi_n \sim \sqrt{h}N(0,1)$ and are iid, then

$$Y_n =_d X_n \qquad \text{for all } n = 0, 1, \ldots, N.$$

And so, we expect there to be a local minimum to the optimization problem at $A = (-1, 0, h)$ since The optimization problem above is coded as follows.

```
In [173]: def modReduction(X,Psi,A_init):
              N = len(X)

              def aux_fun(A, X, Psi, N):
                  a = A[0]
                  b = A[1:]

                  y = np.zeros(N-1)
                  y[0] = X[1]
                  for i in range(1,N-1):
                      y[i] = -a*y[i-1] + b[1]*Psi(X[i]) + b[0]*Psi(X[i-1])
                  return X[1:] - y

              obj_fun = lambda A : aux_fun(A, X, Psi, N)

              A_sol = so.least_squares(obj_fun,A_init).x

              # Then we run it
              a = A_sol[0]
              b = A_sol[1:]

              Y = np.zeros(N)
              y = np.zeros(N-1)
              Y[:1] = X[:1]
              y[0] = X[1]
              for i in range(1,N-1):
                  Y[i+1] = y[i] + np.sqrt(abs(b[1]))*np.random.normal(0,1)
                  y[i] = -a*y[i-1] + b[1]*Psi(Y[i]) + b[0]*Psi(Y[i-1])
              return [Y,A_sol]

In [175]: def sim(X,Psi,A_init):
              N = len(X)
```

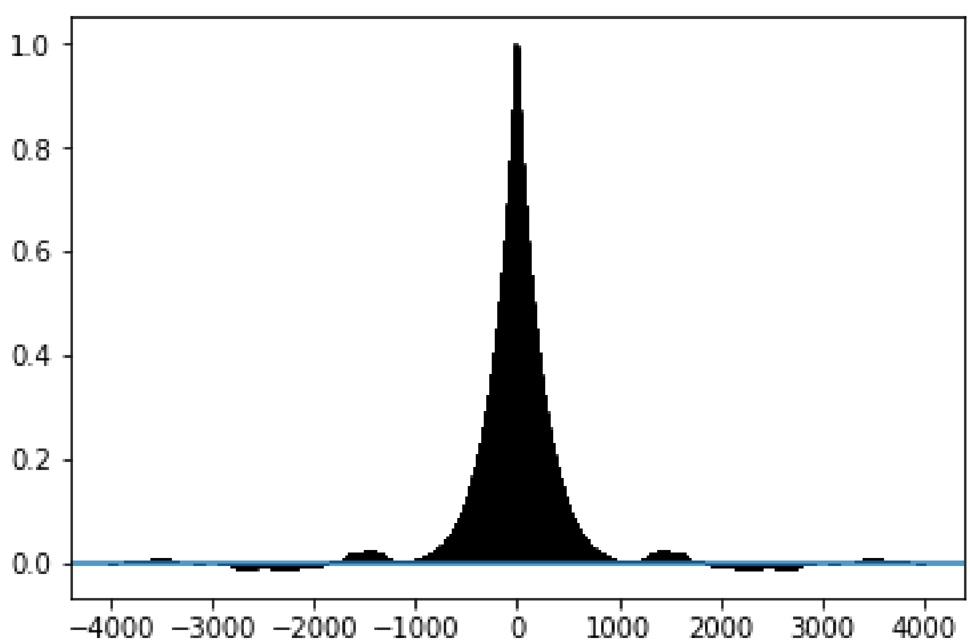
```
    A_sol = A_init

    # Then we run it
    a = A_sol[0]
    b = A_sol[1:]

    Y = np.zeros(N)
    y = np.zeros(N-1)
    Y[:1] = X[:1]
    y[0] = X[1]
    for i in range(1,N-1):
        Y[i+1] = y[i] + np.sqrt(abs(b[1]))*np.random.normal(0,1)
        y[i] = -a*y[i-1] + b[1]*Psi(Y[i]) + b[0]*Psi(Y[i-1])
    return Y
```

Now to implement this. First, we set Ψ and an initial guess at the parameters a and b, we let $A = (a, b)$.

1.3 Preparing for fitting

The above fit will work if the data is stationary. To ensure stationarity we estimate the exponential autocorrelation time τ_{exp}. To do that, we estimate the autocorrelation function. Here this is done by using the `acorr` function in `matplotlib`. It is ploted and we observe the exponential decay of the transient behavior to the mean zero, noise drive behavior.

```
In [152]: maxlags = 4000
          [lags,c] = plt.acorr(X,maxlags = maxlags)[:2]
```



We truncate the autocorrelation function to isolate the exponetial.

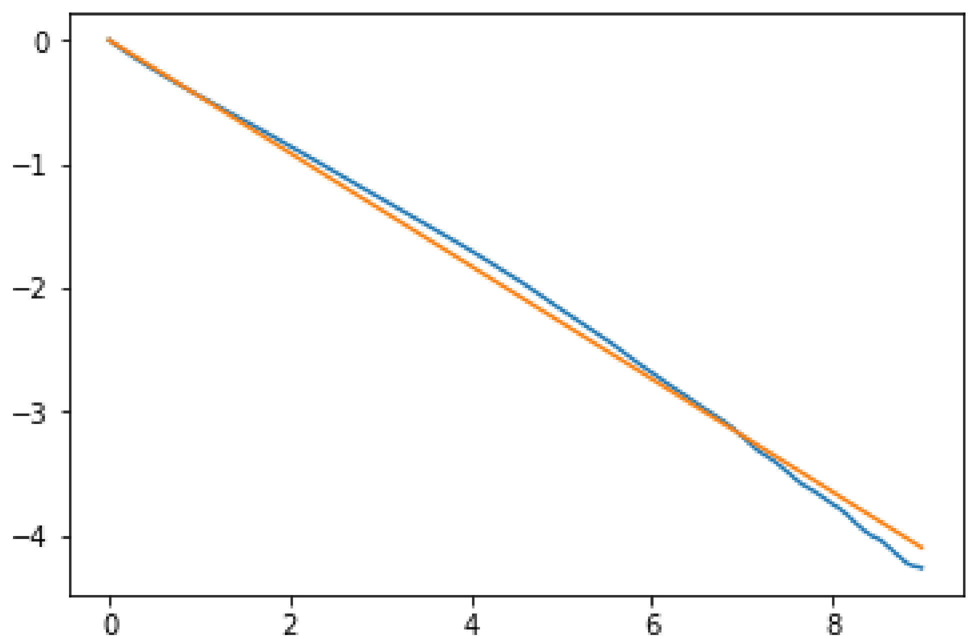
In [153]:
```
trunc = 900
lags_tr = lags[maxlags:maxlags + trunc]
c_tr = c[maxlags:maxlags + trunc]
```

Then we fit a line to the log of the exponetial to recover the exponential autocorrelation time.

In [154]:
```
x = np.array(range(trunc))*h
y = np.log(c_tr)
```

In [155]:
```
plt.plot(x,y)

A = np.matrix([[xx] for xx in x])
B = sa.pinv(A) @ np.matrix(y).T.A1
Y = [B*xx for xx in x]

plt.plot(x,Y)

tau = -1/B[0]
tau
```

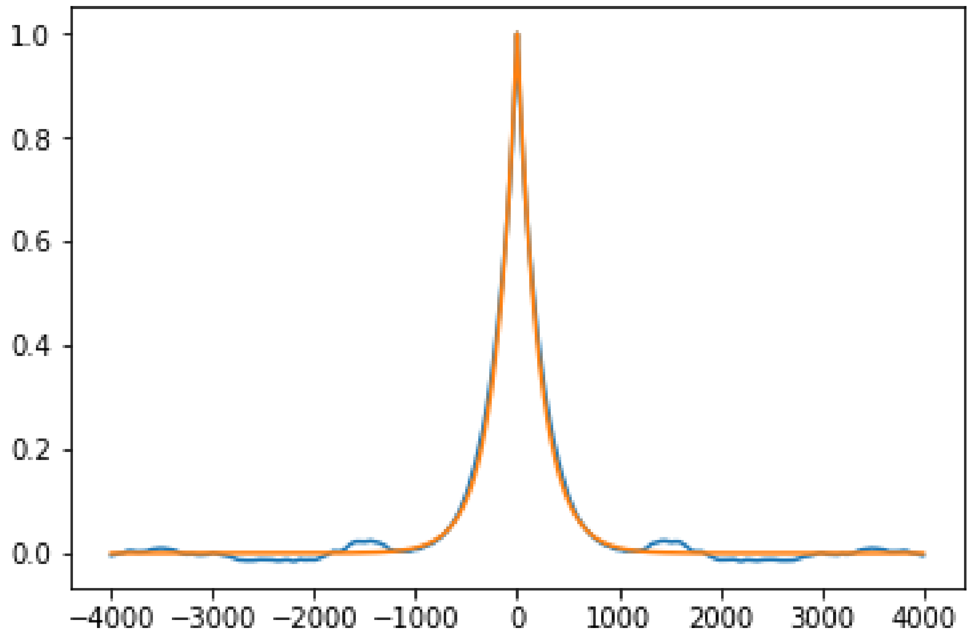
Out[155]: 2.1918542356064505



Here we can check the fit.

In [156]:
```
y = np.exp(-np.abs(lags)*h/tau)
plt.plot(lags,c)
plt.plot(lags,y)
```

Out[156]: [<matplotlib.lines.Line2D at 0x24f2b040978>]



So, we find that the exponential autocorreltation time is around

```
In [157]: n_disc = int(tau*20) +1
          tau, n_disc
```

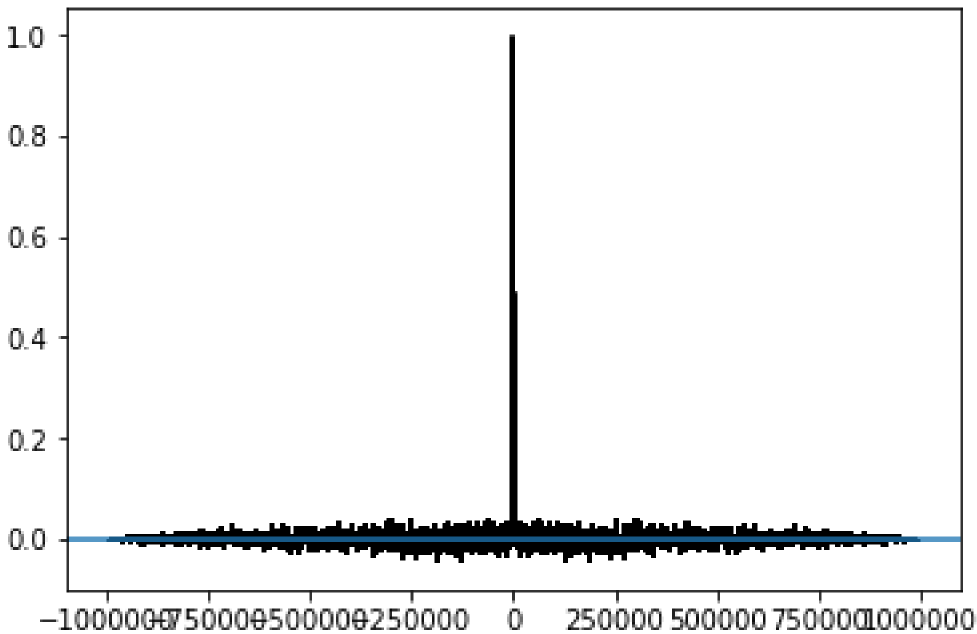
```
Out[157]: (2.1918542356064505, 44)
```

Suggesting that discarding $20\tau \approx 39$ samples will safely exclude the transient behave resulting from starting with a point distribution.

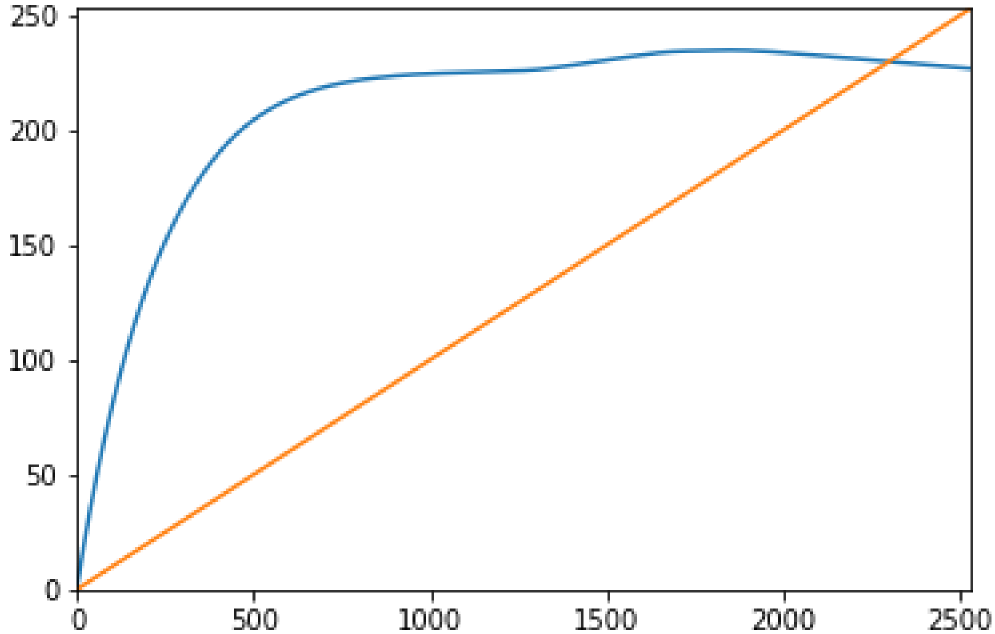
Now, to determine how long we should run the initial system we seek the integrated autocorrelation time τ_{int}, defined as

$$\tau_{int} = \sum_{k=-\infty}^{\infty} \overline{C}k.$$

```
In [158]: [lags,c] = plt.acorr(X,maxlags = None)[:2]
```




```
In [159]: maxlags = int((len(c) - 1) / 2)
          C = 10
          m = lags[maxlags:]
          Y1 = np.cumsum(c[maxlags:])
          Y2 = m/C
          M = np.nonzero((Y2 - Y1) > 0)[0][0]
          plt.plot(m,Y1,m,Y2)
          plt.axis([0, 1.1*M, 0, 1.1*M/C])

Out[159]: [0, 2527.8, 0, 252.78000000000003]
```




```
In [160]: tau_int = np.sum(c[maxlags - M : maxlags + M ])
          (steps - n_disc)/tau_int
```

```
Out[160]: 2180.427479447864
```

Which means we should have effectively about $N/\tau_{int} \approx 2136$ independent samples.

1.4 Fitting

To fit we use the information above which suggests we discard the first 40 seconds of samples.

```
In [161]: Z = X[n_disc:]
```

```
In [188]: Psi = lambda x : x**3
          A_init = [-1,0,-h]Y = sim(X,Psi,A_init)
          plt.plot(t[:1000],Z[:1000])
          plt.plot(t[:1000],Y[:1000])
```

```
In [190]: def aux_fun(A, X, Psi, N):
              a = A[0]
              b = A[1:]

              y = np.zeros(N-1)
              y[0] = X[1]
              for i in range(1,N-1):
                  y[i] = -a*y[i-1] + b[1]*Psi(X[i]) + b[0]*Psi(X[i-1])
              return X[1:] - y
```

```
In [191]: N = len(X)
          np.linalg.norm(aux_fun(A_init, X, Psi, N))
```

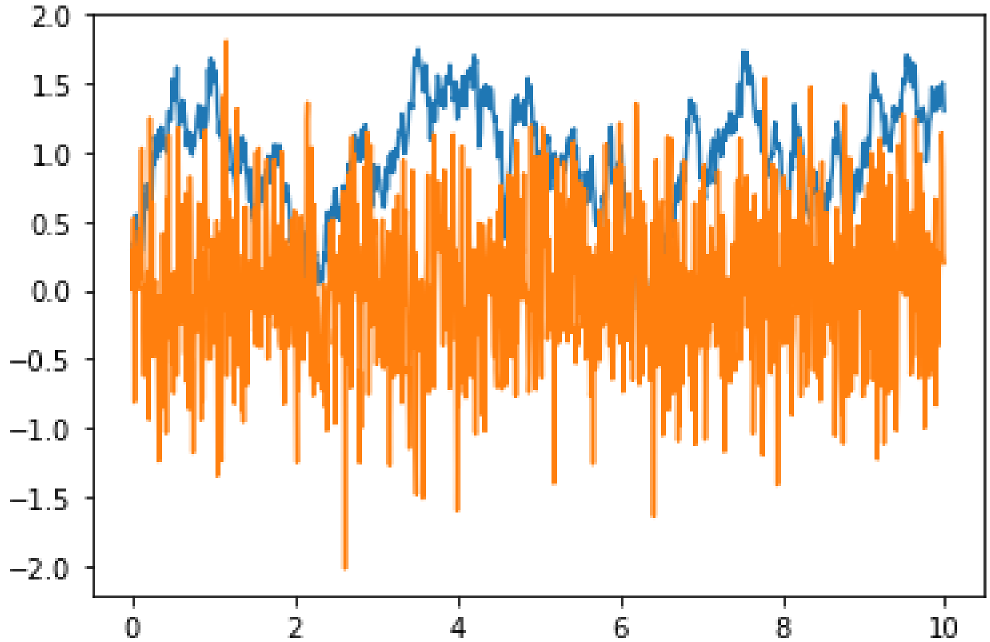
```
Out[191]: 141499.7031003841
```

```
In [192]: [Y,A_sol] = modReduction(Z,Psi,A_init)
          A_sol
```

```
Out[192]: array([-0.98066263, -0.30733008,  0.31910109])
```

```
In [193]: plt.plot(t[:1000],Z[:1000])
          plt.plot(t[:1000],Y[:1000])
```

```
Out[193]: [<matplotlib.lines.Line2D at 0x24f1a1e0e48>]
```




```
In [ ]:
```