ECE538 Assignment 4

tonyw3

11.09.2024

T1:

1. In this exercise, you will approximate a second-order ODE describing the dynamics of a pendulum using a discrete-time LSSM with two hidden states. The pendulum is governed by the following second-order ODE:

$$\frac{d^2\phi}{dt^2} + c\frac{d\phi}{dt} + \sin\phi = u(t) \qquad (*)$$

where $\phi(t)$ is the angle of the pendulum from the vertical at time t, $u(t)$ is an external input torque applied to the pendulum, and c is a friction coefficient. Your goal is to transform this continuous-time system into a discrete-time state-space model using a small discretization step Δt. To do so, follow these steps:

(a) **Define the State Variables**: To convert this second-order ODE into a first-order system, define the following state variables:

$$\mathbf{x}(t) = \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} \phi(t) \\ \omega(t) \end{bmatrix},$$

where $\phi(t)$ is the angle of displacement and $\omega(t) = \frac{d\phi}{dt}$ is the angular velocity. Using this state variable, rewrite the pendulum equation as a system of two first-order ODEs:

$$\begin{cases} \frac{d\phi}{dt} = f_1(\omega, \phi, u(t)) = \omega \\ \frac{d\omega}{dt} = f_2(\omega, \phi, u(t)) = ? \end{cases}$$

Find the explicit form of the function $f_2(\omega, \phi, u(t))$.

(b) **Formulate the State Equations**: Now that we have expressed the system as a first-order system of ODEs, write it in the following state-space form:

$$\begin{bmatrix} \frac{d\phi}{dt} \\ \frac{d\omega}{dt} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \end{bmatrix}}_{\frac{d\mathbf{x}}{dt}} = \underbrace{\begin{bmatrix} f_1(x_2, x_1, u(t)) \\ f_2(x_2, x_1, u(t)) \end{bmatrix}}_{\mathbf{f}(\mathbf{x}, u(t))}.$$

Find the explicit formula for the vector on the right-hand side.

(c) **Discretize the Nonlinear System**: To transform this continuous-time system into a discrete-time model, use Euler's method with a discretization step Δt, i.e.,

$$\mathbf{x}(t + \Delta t) \approx \mathbf{x}(t) + \Delta t \cdot \mathbf{f}(\mathbf{x}(t), u(t)).$$

Define the discrete-time state as $\mathbf{x}_k = \mathbf{x}(k \cdot \Delta t)$ and input $u_k = u(k \cdot \Delta t)$. The state update equation becomes:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t \cdot f(\mathbf{x}_k, u_k),$$

which can be written as the following vector equality:

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \Delta t \cdot \begin{bmatrix} ? \\ ? \end{bmatrix},$$

Fill in the entries of the last vector. Your answer should be a function of $x_{1,k}$, $x_{2,k}$, and u_k.

1(a): $\begin{cases} x_1(t) = \phi(t) & \text{angle} \\ x_2(t) = \omega(t) = \frac{d\phi}{dt} & \text{angular velocity} \end{cases}$

we have $\frac{d^2\phi}{dt^2} = \frac{d\omega}{dt}$, plug in ODE (*)

$\Rightarrow f_2(\omega, \phi, u(t)) = \frac{d\omega}{dt} = \frac{d^2\phi}{dt^2}$

$= \frac{d\omega}{dt} f_1(\omega, \phi, u(t))$

$= u(t) - c\omega - \sin\phi$

1(b): $\begin{cases} \frac{dx_1}{dt} = \frac{d\phi}{dt} = f_1(\omega, \phi, u(t)) = \omega = x_2 \\ \frac{dx_2}{dt} = \frac{d\omega}{dt} = \frac{d^2\phi}{dt^2} = -c\frac{d\phi}{dt} - \sin\phi + u(t) \end{cases}$

So $\frac{d}{dt}\vec{x}(t) = \begin{bmatrix} x_2 \\ u(t) - cx_2 - \sin x_1 \end{bmatrix}$

1(c): $\vec{x}(t + \Delta t) \approx \vec{x}(t) + \Delta t \frac{d\vec{x}}{dt}$

$\Rightarrow$ in Euler's method, as we learned in ESE500,

we can discretize sys as: $\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \end{bmatrix} = \begin{bmatrix} x_{1,k} \\ x_{2,k} \end{bmatrix} + \Delta t \begin{bmatrix} x_{2,k} \\ u_k - c \cdot x_{2,k} - \sin x_{1,k} \end{bmatrix}$

(d) Implement the resulting LSSM in Python and plot the evolution of x_1 as a function of k. In your simulations, use `delta_t = 0.01` and simulate for `num_steps=10000` steps. Set the initial angle to $x_{1,0} = \pi - 0.1$ (close to the vertical position) and the initial angular velocity to zero, i.e., $x_{2,0} = 0$. Set the input to be:

c=0.1

```
# Sinusoidal input
amplitude = 0
frequency = 0.5
# Sinusoidal input over time
u = amplitude*np.sin(frequency*np.arange(num_steps))
```

Provide a physical interpretation of your observations.

(e) Using the same parameters, include a plot where the x-axis is $x_{1,k}$ and the y-axis is $x_{2,k}$. This type of plot is called a **phase space** plot. Provide a physical interpretation of your observations.

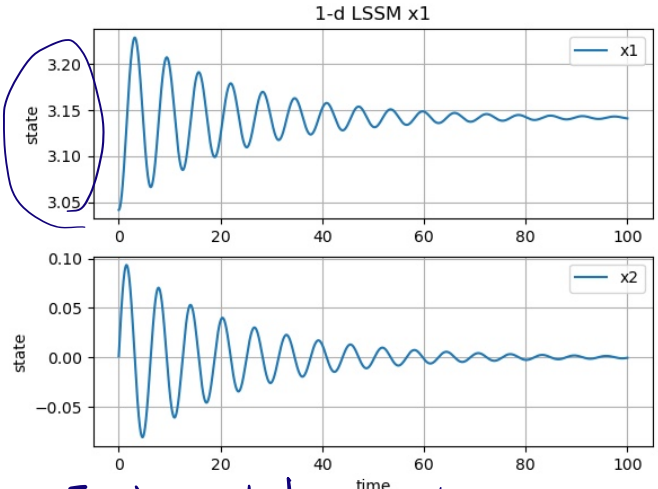
(f) In a pendulum, you can induce complex behavior by choosing the right set of parameters. For example, plot the phase space with the same initial conditions and the following set of parameters:

```
# Parameters
delta_t = 0.01  # Time step
num_steps = 100000  # Number of time steps to simulate
c = 0.01  # Friction coefficient

# Sinusoidal input
amplitude = 1
frequency = 0.01
```

What does the pendulum do in physical terms? Does it stabilize to an equilibrium point? Does it oscillate periodically? Does it oscillate irregularly?

(d): As shown below,
initialy, the pendulum oscillate in damp due to friction, the it coverges to 3.14 in expential decay way
⇒ sys is stable and small interactin while go equilibrium finally

```
### 1-d LSSM
num_steps  = 10000
delta_t = 0.01
x1_0 = np.pi - 0.1
x2_0 = 0.0
C = 0.1
amplitude = 0
frequency = 0.5
u = amplitude *np.sin( frequency *np. arange ( num_steps ) )

x1  = np.zeros(num_steps)
x2 = np.zeros(num_steps)

for i in range(num_steps):
    x1[i] = x1_0 + delta_t * x2_0
    x2[i] = x2_0 + delta_t * (np.sin(x1_0) - C *x2_0 + u[i])
    x1_0 = x1[i]
    x2_0 = x2[i]

plt.figure()
plt.subplot(2,1,1)
plt.plot(np.arange(num_steps)*delta_t, x1, label='x1')

plt.title('1-d LSSM x1')
plt.ylabel('state')

plt.legend()
plt.grid()

plt.subplot(2,1,2)
plt.plot(np.arange(num_steps)*delta_t, x2, label='x2')

plt.xlabel('time')
plt.ylabel('state')

plt.legend()
plt.grid()
plt.show()
```

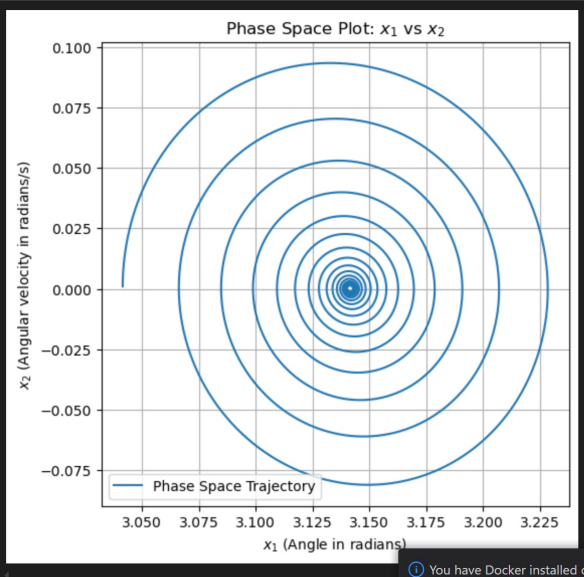


I also plot x2 here

(e): As shown on the right, it's a spiral inward pattern
⇒ pendulum motion is damping to zero as time goes,

```
# 1-e Phase space plot (x1 vs x2)
plt.figure(figsize=(6, 6))
plt.plot(x1, x2, label='Phase Space Trajectory')
plt.title('Phase Space Plot: $x_1$ vs $x_2$')
plt.xlabel('$x_1$ (Angle in radians)')
plt.ylabel('$x_2$ (Angular velocity in radians/s)')
plt.legend()
plt.grid(True)
plt.show()
```

✓ 0.2s

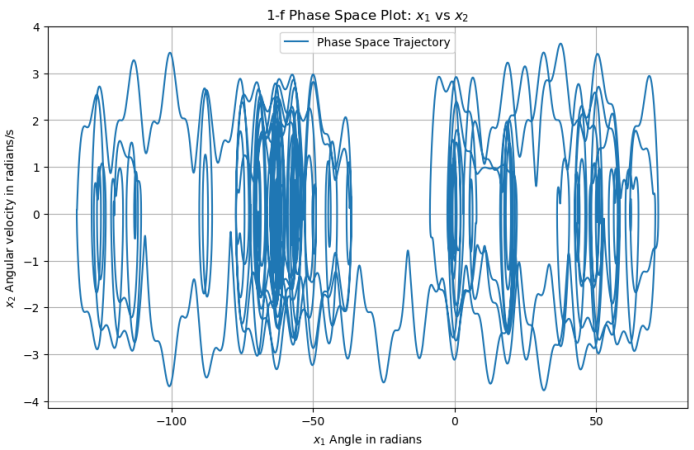


(f): As shown below, the phase space is of chaotic under complex plane:

$$\begin{cases} x_k^1 = x_{k-1}^1 + \Delta t \cdot x_{k-1}^2 \\ x_k^2 = x_{k-1}^2 + \Delta t\left(-C x_{k-1}^2 - \sin(x_{k-1}^1) + u_{k-1}\right) \end{cases}$$

$$u_k = A_u \sin(f_u \cdot k)$$

$$= 1 \cdot \sin(0.01\, k)$$



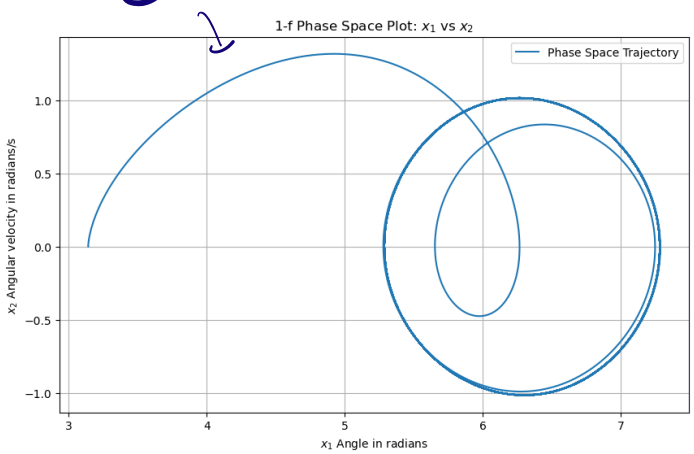
This is because $A_u \gg C$, where the input u_k interation destabilize system, making it oscillating irregularity

⇒ it could be chaotic moving along circle

But if $C = 1 = A_u$, then system go stable again:

'C=1 case

```
# (f)
delta_t = 0.01 # Time step
num_steps = 100000 # Number of time steps to simulate
c = 0.01  # friction coefficient
# Sinusoidal input
amplitude = 1
frequency = 0.01
u = amplitude * np.sin(frequency * np.arange(num_steps))

x1 = np.zeros(num_steps)
x2 = np.zeros(num_steps)
x1[0] = x1_0
x2[0] = x2_0

for k in range(1, num_steps):
    x1[k] = x1[k-1] + delta_t * x2[k-1]
    x2[k] = x2[k-1] + delta_t * (-c * x2[k-1] - np.sin(x1[k-1]) + u[k-1])

plt.figure(figsize=(10, 6))
plt.plot(x1, x2, label='Phase Space Trajectory')
plt.title('1-f Phase Space Plot: $x_1$ vs $x_2$')
plt.xlabel('$x_1$ Angle in radians')
plt.ylabel('$x_2$ Angular velocity in radians/s')
plt.legend()
plt.grid(True)
plt.show()
```



T2: HMM: Hidden Markov Model

observable signals. Suppose the machine can be in one of three hidden states at any given time: **Normal Operation** (State 1), **Minor Fault** (State 2), or **Severe Fault** (State 3). Although these states are hidden, they influence observable sensor readings. The sensor outputs one of three observable levels of vibration: **Low Vibration** (Observation 1), **Medium Vibration** (Observation 2), or **High Vibration** (Observation 3). Your goal is to analyze an HMM that represents this system, then simulate the model's response to a given observation sequence, and interpret the results. The initial distribution, state transition matrix, and emission matrix of the HMM are as follows:

$$\boldsymbol{\pi} = \begin{bmatrix} 0.8 & 0.15 & 0.05 \end{bmatrix}^\intercal, \ \mathbf{A} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.05 & 0.15 & 0.8 \end{bmatrix}, \ \mathbf{B} = \begin{bmatrix} 0.9 & 0.08 & 0.02 \\ 0.3 & 0.6 & 0.1 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}.$$

Follow the steps below to construct the model.

(a) **Simulate the Observation Sequence**: Using these HMM parameters, simulate the hidden states and corresponding observations over time. Write a function in `Python` that generates a sequence of hidden states and observations based on the transition and emission probabilities.

(b) **Decode the Observation Sequence**: Build a function in `Python` that, for each observation, identifies the most likely hidden state by finding the highest emission probability in the emission matrix for each state. For example, if you observe a high vibration level, the most likely hidden state would be **Severe Fault**, since that state has the highest probability of producing this observation.

(c) **Apply the Model to a Given Observation Sequence**: Using the following observation sequence:

`[1, 1, 2, 3, 3, 2, 1, 1, 2, 3]`

use your decoding function in `Python` to estimate the hidden state sequence that most likely produced these observations.

(d) **Probability of the Hidden State Sequence**: Using the HMM parameters, calculate in `Python` the probability of the hidden state sequence you estimated in the previous step.


```python
pi = np.array([0.8, 0.15, 0.05])  # Initial state distribution
A = np.array([
    [0.7, 0.2, 0.1],
    [0.1, 0.8, 0.1],
    [0.05, 0.15, 0.8]
])  # Transition matrix
B = np.array([
    [0.9, 0.08, 0.02],
    [0.3, 0.6, 0.1],
    [0.1, 0.3, 0.6]
])  # Emission matrix

states = ['Normal Operation', 'Minor Fault', 'Severe Fault']
observations = ['Low Vibration', 'Medium Vibration', 'High Vibration']

def HMM(n_steps):
    hidden_states = []
    observed_sequence = []

    current_state = np.random.choice([0, 1, 2], p=pi)
    hidden_states.append(current_state)

    observation = np.random.choice([0, 1, 2], p=B[current_state])
    observed_sequence.
                      (parameter) n_steps: Any

    for _ in range(1, n_steps):
        current_state = np.random.choice([0, 1, 2], p=A[current_state])
        hidden_states.append(current_state)
        observation = np.random.choice([0, 1, 2], p=B[current_state])
        observed_sequence.append(observation)
    return hidden_states, observed_sequence

n_steps = 10
hidden_states, observed_sequence = HMM(n_steps)

def translate_to_string(hidden_states, observed_sequence):
    s_list = [states[i] for i in hidden_states]
    o_list = [observations[i] for i in observed_sequence]
    print('States:\n- ',hidden_states,'\n- ', s_list)
    print('Observations:\n- ',observed_sequence,'\n- ', o_list)

translate_to_string(hidden_states, observed_sequence)
```

✓ 0.0s

```
States:
- [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
- ['Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault']
Observations:
- [0, 1, 1, 0, 1, 2, 0, 1, 1, 1]
- ['Low Vibration', 'Medium Vibration', 'Medium Vibration', 'Low Vibration', 'Medium Vibration', 'High Vibration', 'Low Vibration', 'Medium Vibration', 'Medium Vibration', 'Medium Vibration']
```

(a):

States:
- [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
- ['Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault', 'Minor Fault']

Observations:
- [0, 1, 1, 0, 1, 2, 0, 1, 1, 1]
- ['Low Vibration', 'Medium Vibration', 'Medium Vibration', 'Low Vibration', 'Medium Vibration', 'High Vibration', 'Low Vibration', 'Medium Vibration', 'Medium Vibration', 'Medium Vibration']

(b):

```python
def decode_observations(B, observations):
    decoded_states = []
    for obs in observations:
        state = np.argmax( B[:, obs])
        decoded_states.append(state)
    return decoded_states

decoded_states = decode_observations(B, observations)

print("Decoded States:")
for t in range(num_steps):
    print(f"{t}\t{state_names[decoded_states[t]]}\t {o
```

✓ 0.0s

```
Decoded States:
0       Normal Operation!       Low Vibration
1       Normal Operation!       Low Vibration
2       Normal Operation!       Low Vibration
3       Severe Fault    High Vibration
4       Minor Fault     Medium Vibration
5       Minor Fault     Medium Vibration
6       Minor Fault     Medium Vibration
7       Minor Fault     Medium Vibration
8       Severe Fault    High Vibration
9       Normal Operation!       Low Vibration
10      Normal Operation!       Low Vibration
11      Normal Operation!       Low Vibration
12      Normal Operation!       Low Vibration
13      Minor Fault     Medium Vibration
14      Minor Fault     Medium Vibration
15      Minor Fault     Medium Vibration
16      Minor Fault     Medium Vibration
17      Normal Operation!       Low Vibration
18      Minor Fault     Medium Vibration
19      Minor Fault     Medium Vibration
```

(c):

```python
given_seq = [1, 1, 2, 3, 3, 2, 1, 1, 2, 3]
# adjust to 0-based indexing
ob_seq = [i-1 for i in given_seq]
print(ob_seq)

dec_seq = decode_observations(B, ob_seq)

print("Estimated hidden states: ")
for t,ob in enumerate(ob_seq):
    print(f"{t}\t{observation_names[ob]}\t {state_names[dec_seq[t]]}")
```

[33] ✓ 0.0s

```
[0, 0, 1, 2, 2, 1, 0, 0, 1, 2]
Estimated hidden states:
0       Low Vibration    Normal Operation!
1       Low Vibration    Normal Operation!
2       Medium Vibration         Minor Fault
3       High Vibration   Severe Fault
4       High Vibration   Severe Fault
5       Medium Vibration         Minor Fault
6       Low Vibration    Normal Operation!
7       Low Vibration    Normal Operation!
8       Medium Vibration         Minor Fault
9       High Vibration   Severe Fault
```

(d) Prob = 1.8816×10^{-6}

```python
# 2-d
def compute_seq_prob(pi, A, B, states):
    prob = pi[states[0]]
    for t in range(1, len(states)):
        prob *= A[states[t-1], states[t]]
    return prob
probability = compute_seq_prob(pi, A, B, dec_seq)

print(f"Probability of the estimated hidden state sequence: {probability:.6e}")
```

✓ 0.0s

```
Probability of the estimated hidden state sequence: 1.881600e-06
```

T3

3. Consider a dynamical system characterized by the following system of recursions:

$$x_{1,k+1} = x_{1,k}/2 + x_{2,k}/4 + x_{3,k}/8 + \sin(3k),$$
$$x_{2,k+1} = x_{1,k},$$
$$x_{3,k+1} = x_{2,k},$$

with the observation equation:

$$y_k = x_{1,k} + x_{2,k} + x_{3,k}.$$

(a) Write the system in matrix form.

(b) Is the system stable? *Hint*: The state matrix is stable if its eigenvalues are inside the unit circle.

(c) Determine whether the LSSM is in controllable or observable canonical form. *Hint*: Refer to Example 6 for guidance on canonical forms.

(a): let $\vec{x}_k = \begin{bmatrix} x_{1,k} \\ x_{2,k} \\ x_{3,k} \end{bmatrix}$ ⇒ And SSM as: $\vec{x}_{k+1} = A\vec{x}_k + B_c\vec{u}_k$

So $A = \begin{bmatrix} \frac{1}{2}, \frac{1}{4}, \frac{1}{8} \\ 1, 0, 0 \\ 0, 1, 0 \end{bmatrix}$ $\vec{B}_c = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ $\vec{u}_k = \sin 3k$

observation

$\vec{y}_k = C\vec{x}_k \Rightarrow C = [1, 1, 1]$

```
A = np.array([
    [0.5, 0.25, 0.125],
    [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0]
])
B = np.array([[1], [0], [0]])
C = np.array([1.0, 1.0, 1.0])  # Observation matrix

eigenvalues = np.linalg.eigvals(A)
print("Eigenvalues of A:")
print(eigenvalues)
```

```
Eigenvalues of A:
[ 0.91964338+0.j         -0.20982169+0.30314536j -0.20982169-0.30314536j]
```

(b): First let's solve state matrix A's eigenvalues, i.e. $\det(A - \lambda I) = 0$

$$\Rightarrow \det\begin{bmatrix} \frac{1}{2}-\lambda, \frac{1}{4}, \frac{1}{8} \\ 1, -\lambda, 0 \\ 0, 1, -\lambda \end{bmatrix} = 0$$

$$\Rightarrow \left(\tfrac{1}{2}-\lambda\right)\left[(-\lambda)(-\lambda) - 0\cdot 1\right] - \frac{1}{4}\left[1\cdot(-\lambda) - 0\cdot 0\right] + \frac{1}{8}\left[1\cdot 1 - 0\cdot(1-\lambda)\right] = 0$$

$$\Rightarrow \frac{1}{2}\lambda^2 - \lambda^3 + \frac{1}{4}\lambda + \frac{1}{8} = 0$$

$$\Rightarrow 8\lambda^3 - 4\lambda^2 - 2\lambda - 1 = 0$$

So $\lambda_1 = 0.92$ Since $|\lambda_k| < 1$ for $\forall k \in \{1,2,3\}$

$\lambda_2 = -0.21 + 0.3i$ So all eigenvalues of A are within

$\lambda_3 = -0.21 - 0.3i$ the unit circle ⇒ system is stable

(c): Sys is both controllable & observable, as rank(A) = 3 is full

As $A = \begin{bmatrix} \frac{1}{2}, \frac{1}{4}, \frac{1}{8} \\ 1, 0, 0 \\ 0, 1, 0 \end{bmatrix}$ $\vec{B}_c = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ shape fit with CCF as Example 6 says

$u_k = \sin 3k$

⇒ we say System is in Controllable Canonical Form

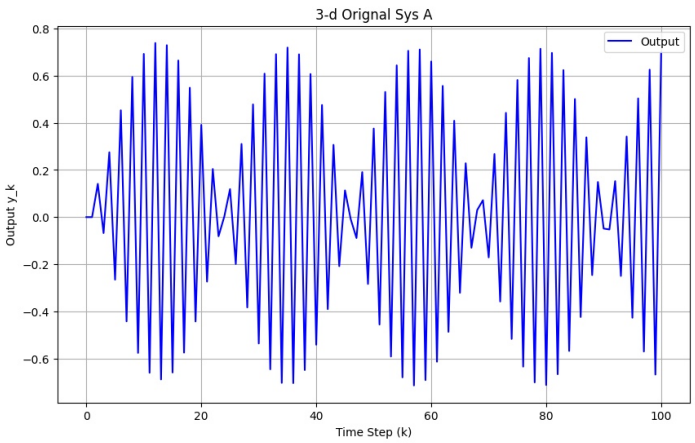
(d) Simulate the system in Python for 100 time steps, using the initial condition $\mathbf{x}_0 = \mathbf{0}$. Plot the evolution of y_k over time. Verify whether the system is stable.

(e) Write the state and output equations of the system in observable canonical form.

(f) Simulate the transformed system in Python for 100 time steps, using the initial condition $\boldsymbol{\xi}_0 = \mathbf{0}$. Plot the evolution of the output y_k over time.

(g) Compare the simulations of the original and transformed systems. Justify your observations.

(d): Shown below, the sys is clearly stable

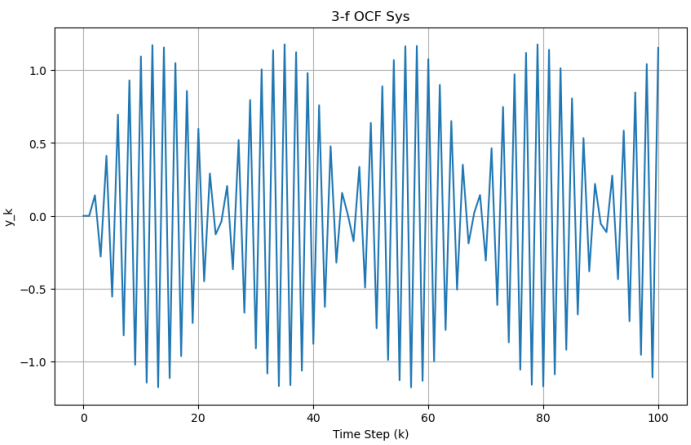



```python
# 3-d CCF case
def sys_dynamic_3(A, C, name, num_steps=100, x0=None):
    def u(k):
        return np.sin(3 * k)
    x = np.zeros((A.shape[0], num_steps + 1))
    y = np.zeros(num_steps + 1)
    x[:, 0] = np.zeros(3) if x0 is None else x0

    for k in range(num_steps):
        x[:, k+1] = A @ x[:, k] + np.array([u(k), 0, 0])
        y[k] = C @ x[:, k]
    y[num_steps] = C @ x[:, num_steps]  # Final output

    plt.figure(figsize=(10, 6))
    plt.plot(range(num_steps + 1), y, 'b-', label='Output')
    plt.xlabel('Time Step (k)')
    plt.ylabel('Output y_k')
    plt.title(name)
    plt.grid(True)
    plt.legend()
    plt.show()

    return x, y

x, y = sys_dynamic_3(A, C, '3-d Orignal Sys A')
```

(e): OCF:

$$A=\begin{bmatrix} 0 & 0 & -\frac{1}{8} \\ 1 & 0 & -\frac{1}{4} \\ 0 & 1 & -\frac{1}{2} \end{bmatrix} \quad \vec{B}=\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \vec{C}=\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

(f):




```python
# 3-f OCF case
A_o = np.array([
    [0.0, 0.0, -0.125],
    [1.0, 0.0, -0.25],
    [0.0, 1.0, -0.5]
])
b_o = np.array([1.0, 0.0, 0.0])
C_o = np.array([1.0, 0.0, 0.0])
xi0 = np.zeros(3)

num_steps = 100
xi = np.zeros((3, num_steps + 1))
y_o = np.zeros(num_steps + 1)
xi[:, 0] = xi0

def u(k):
    return np.sin(3 * k)

for k in range(num_steps):
    xi[:, k+1] = A_o @ xi[:, k] + b_o * u(k)
    y_o[k] = C_o @ xi[:, k]

y_o[num_steps] = C_o @ xi[:, num_steps]

plt.figure(figsize=(10, 6))
plt.plot(range(num_steps + 1), y_o)
plt.xlabel('Time Step (k)')
plt.ylabel('y_k')
plt.title('3-f OCF Sys')
plt.grid(True)
plt.show()
```

(g): As shown above, CCF & OCF form has same plot for systen in shape

T4

4. Consider an LSSM with a single hidden state, governed by the following equations:

$$x_{k+1} = x_k/2 + u_k,$$
$$y_k = x_k.$$

Answer the questions below:

(a) Starting with the initial condition $x_0 = 2$, derive an expression for the output y_k in terms of x_0 and the sequence of inputs $\{u_0, u_1, \ldots, u_{k-1}\}$.

(b) Suppose the input u_k is a step function, defined as $u_k = 1$ for all $k \geq 0$ and 0 otherwise. Using your answer from the previous question, derive an expression for the output sequence y_k under this step input.

(c) Compute the sequence of Markov parameters H_i for this system.

(d) Using the Markov parameters $\{H_0, H_1, H_2, \ldots\}$ and the step function as the input sequence, compute the convolution of the Markov parameters with the input. Express the resulting sequence $\{y_0, y_1, y_2, \ldots\}$ as a function of k.

Define a new state variable ξ_k by the invertible transformation $\xi_k = \frac{x_k}{2}$.

(e) Derive the transformed state-space matrices $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}$ and the transformed initial condition ξ_0 for the new state-space model in terms of a, b, c, and d.

(f) Using the step function as an input, compute the output y_k of the transformed LSSM for $k = 0, 1, 2, \ldots$

(a): Case for $x_0 = 2$

$x_1 = x_0/2 + u_0 = u_0 + 1$

$x_2 = x_1/2 + u_1 = u_1 + \frac{1}{2}u_0 + \frac{1}{2}$

$x_3 = x_2/2 + u_2 = u_2 + \frac{1}{2}u_1 + \frac{1}{4}u_0 + \frac{1}{4}$

$\vdots$

$x_k = x_{k-1}/2 + u_{k-1} = u_{k-1} + \frac{1}{2}u_{k-2} + \cdots + \frac{1}{2^{k-1}}u_0 + \left(\frac{1}{2}\right)^k$

$\Rightarrow x_k = \left(\frac{1}{2}\right)^{k-1} + \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-i-1}u_i$

$y_k = \left(\frac{1}{2}\right)^k x_0 + \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-i-1}u_i$

where $y_k^h = \left(\frac{1}{2}\right)^{k-1}$ as homogeneous sol

particular sol is $u_k * \left(\frac{1}{2}\right)^k$

(a)' Case for $x_0 = 0$

$x_1 = x_0/2 + u_0 = u_0$

$x_2 = x_1/2 + u_1 = u_1 + \frac{1}{2}u_0$

$x_3 = x_2/2 + u_2 = u_2 + \frac{1}{2}u_1 + \frac{1}{4}u_0$

$\vdots$

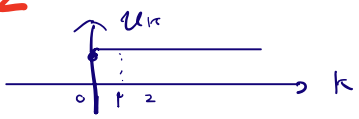
$x_k = x_{k-1}/2 + u_{k-1} = u_{k-1} + \frac{1}{2}u_{k-2} + \cdots + \frac{1}{2^{k-1}}u_0$

$\Rightarrow x_k = \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-i-1}u_i$

$y_k = x_k = \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-i-1}u_i$

Case for $x_0 = 2$

(b): $u_k = 1$ for $\forall k \geq 0$



$\Rightarrow y_k = \left(\frac{1}{2}\right)^{k-1} + \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-1-i} \cdot 1$

$= \left(\frac{1}{2}\right)^{k-1} + \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{i}$

$= \left(\frac{1}{2}\right)^{k-1} + \frac{1-\left(\frac{1}{2}\right)^k}{1-\frac{1}{2}}$

$= \left(\frac{1}{2}\right)^{k-1} + 2 - \left(\frac{1}{2}\right)^{k-1}$

$= 2$

So for $\forall k \geq 0$, $y_k = 2$ under $u_k \equiv 1$ input

Case for $x_0 = 0$

$y_k = \sum_{i=0}^{k-1}\left(\frac{1}{2}\right)^{k-i-1} \cdot 1$

$= 2 - \left(\frac{1}{2}\right)^{k-1}$

Plese don't change sys init conditn....

(c): In HMM, we have:

$$\begin{cases} H_0 = D = 0 \\ H_i = C A^{i-1} B \text{ for } i \geq 1 \\ A = \frac{1}{2},\ B = 1,\ C = 1,\ D = 0 \end{cases}$$

So $H_i = \begin{cases} \left(\frac{1}{2}\right)^{i-1}, & i \geq 1 \\ 0, & i = 0 \end{cases}$

(d): $y_k = H_k * u_k$

$$= \sum_{i=0}^{k} H_i \cdot u_{k-i}$$

$$= 0 \cdot 1 + \sum_{i=1}^{k} \left(\frac{1}{2}\right)^{i-1} \cdot 1$$

$$= \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^{i}$$

$$= 2 - \left(\frac{1}{2}\right)^{k-1}, \text{ which is}$$

identical to (b) as $u_k = 1$ for $k \geq 0$

(e): Transform $\xi_k = x_k / 2$, then $x_k = 2\xi_k$

then $\begin{cases} 2\xi_{k+1} = \frac{1}{2} \cdot 2\xi_k + u_k \\ y_k = 2\xi_k \end{cases} \Rightarrow \begin{cases} \xi_{k+1} = \frac{1}{2}\xi_k + \frac{1}{2}u_k \\ y_k = 2\xi_k \end{cases}$

So $\begin{cases} \tilde{A} = \frac{1}{2} \\ \tilde{B} = \frac{1}{2} \\ \tilde{C} = 2 \\ \tilde{D} = 0 \end{cases}$

And $\xi_0 = \frac{x_0}{2} = 0$ unchanged

$$\xi_k = \sum_{i=0}^{k} \left(\frac{1}{2}\right)^{k} u_k$$

(f): for $u_k = \text{step}(k) = 1$, $k \geq 0$

$$\Rightarrow \xi_k = \sum_{i=0}^{k} \left(\frac{1}{2}\right)^{k} u_k$$

$$= \sum_{i=0}^{k} \left(\frac{1}{2}\right)^{k} = 1 - \left(\frac{1}{2}\right)^{k}$$

$$y_k = 2 - \left(\frac{1}{2}\right)^{k-1}$$

5. Consider an LSSM model with system matrices (A, B, C, D). Prove the following identities:

$$\frac{\partial \|\mathbf{y}_k - \hat{\mathbf{y}}_k\|_2^2}{\partial \hat{\mathbf{y}}_k} = 2(\hat{\mathbf{y}}_k - \mathbf{y}_k)^\intercal, \quad \frac{\partial \hat{\mathbf{y}}_k}{\partial \mathbf{x}_k} = C \text{ and } \frac{\partial \mathbf{x}_{j+1}}{\partial \mathbf{x}_j} = A \text{ for all } j \geq 1.$$

6. Prove that

$$\frac{\partial (UXV)}{\partial X} = U \otimes V^\intercal$$

LSSM model:

$$\begin{cases} x_{k+1} = Ax_k + Bu_k \\ \hat{y}_k = Cx_k + Du_k \end{cases}$$

5° (1) Prove $\frac{\partial \|y_k - \hat{y}_k\|_2^2}{\partial \hat{y}_k} = 2(\hat{y}_k - y_k)^T$.

let $J = \|y_k - \hat{y}_k\|_2^2 = \sum_{i=1}^{n} (y_{k,i} - \hat{y}_{k,i})^T (y_{k,i} - \hat{y}_{k,i})$

then for j, $\left[\frac{\partial J}{\partial \hat{y}_k}\right]_j = \frac{\partial J}{\partial \hat{y}_{k,j}} = \frac{\partial}{\partial \hat{y}_{k,j}} \sum_{i=1}^{n} (y_{k,i} - \hat{y}_{k,i})^T (y_{k,i} - \hat{y}_{k,i})$

$$= \sum_{i=1}^{n} \frac{\partial}{\partial \hat{y}_{k,j}} (y_{k,i} - \hat{y}_{k,i})^T (y_{k,i} - \hat{y}_{k,i})$$

let $z = (y_{k,i} - \hat{y}_{k,i})$

then $\left[\frac{\partial J}{\partial \hat{y}_k}\right]_j = \sum_{i=1}^{n} \left[z \cdot \frac{\partial z^T}{\partial \hat{y}_{k,j}} + z^T \frac{\partial z}{\partial \hat{y}_{k,j}}\right]$

$$= z \frac{\partial z^T}{\partial \hat{y}_{kj}} + z^T \frac{\partial z}{\partial \hat{y}_{kj}} \quad \text{where } i = j, \text{ nonzero}$$

$$= -z^T - z^T$$

$$= -2z^T = 2(\hat{y}_{kj} - y_{kj})^T$$

So for $\frac{\partial J}{\partial \hat{y}_k} = 2(\hat{y}_k - y_k)^T$ (stack vector together)

Prove $\forall j \geq 1$,

(2): $\frac{\partial \hat{y}_k}{\partial x_k} = C, \quad \frac{\partial x_{j+1}}{\partial x_j} = A$

① $\hat{y}_k = Cx_k + Du_k$,

$\because u_k$ is independent of x_k

$\therefore$ for each $i, j \in R$, we have:

$$\frac{\partial [\hat{y}_k]_i}{\partial [x_k]_j} = \frac{\partial}{\partial x_{k,j}} \left[\sum_{d=1}^{c} C_{id} x_{k,d} + D u_{k,i}\right]$$

$\Rightarrow \therefore \frac{\partial \hat{y}_k}{\partial x_k} = C$

② Similar to ①, since

$x_{k+1} = Ax_k + Bu_k$

So $\left[\frac{\partial x_{k-1}}{\partial x_k}\right]_j = A_{jk}$

$\Rightarrow x_k$ is independent of u_k

$\Rightarrow \frac{\partial x_{k+1}}{\partial x_k} = A$

T6: $\frac{\partial UXV}{\partial X} = U \otimes V^T$

let $U \in R^{p \times m}$, $X \in R^{m \times n}$, $V \in R^{n \times q}$

$$f(X) = UXV = \sum_{i=1}^{m} \sum_{j=1}^{n} U_i X_{ij} V_j$$

$$[UXV]_i = \sum_{k_1, k_2}^{m,n} u_{ik_1} X_{k_1 k_2} v_{k_2 i}$$

we then take the partial derivative w/ respect to $\vec{X}$:

$$\left[\frac{\partial UXV}{\partial X}\right]_{i,j,j_2} = \frac{\partial}{\partial x_{j,j_1 k_1 k_2}} \sum^{m,n} u_{ik_1} X_{k_1 k_2} v_{k_2 i}$$

$$= \sum_{k,k_1} u_{ik_1} \frac{\partial x_{k,k_2}}{\partial x_{i,j}} v_{k_2 i} = \sum_{k_1 k_1} u_{ik} v_{k_2 j_1} = U \otimes V^T$$

QED.

7. Prove that

$$\frac{\partial \mathbf{x}_{k+1}}{\partial A} = \mathbb{I}^{(2)} \otimes \mathbf{x}_k + A : \frac{\partial \mathbf{x}_k}{\partial A},$$

where ":" is the symbol for the contraction product using a single index.

8. Explain in your own words the vanishing and exploding gradient problem. What techniques have we covered in this chapter to alleviate this issue?

7° LSSM model:

$$\begin{cases} x_{k+1} = Ax_k + Bu_k \\ y_k = Cx_k + Du_k \end{cases}$$

For Base case $k=0$

$$\Rightarrow \frac{\partial x_1}{\partial A} = \frac{\partial (Ax_0 + Bu_0)}{\partial A} = x_0 = I^{(2)} \otimes x_0$$ ✓

By Induction:

Assume for $k>0$, $\frac{\partial x_k}{\partial A} = \sum_{n=0}^{k-1} A^n \otimes x_{k-n-1}$

for $\frac{\partial x_{k+1}}{\partial A} = \frac{\partial (Ax_k)}{\partial A} + 0$

$$= \frac{\partial A}{\partial A} \otimes x_k + A : \frac{\partial x_k}{\partial A}$$

$$= I^2 \otimes x_k + A : \frac{\partial x_k}{\partial A}$$

So proved

8°: Def: → Cite from my internship @IDEA

- Vanishing gradient: When gradient shrink to None, sys don't fit any more
- Exploding Gradients: when gradient grow to inf, system diverges

- **Vanishing Gradients**: When the eigenvalues of the state matrix A are less than one in magnitude, repeated matrix multiplications during BPTT cause the gradient to decay exponentially. This decay makes it difficult for the network to learn dependencies across long sequences.
- **Exploding Gradients**: Conversely, when the eigenvalues of A are greater than one in magnitude, the gradient grows exponentially over time steps, leading to unstable updates and potential divergence during training.

To alleviate it:

① gradient clipping

if $\|\nabla \mathcal{L}\| > \tau$, then $\nabla \mathcal{L} \leftarrow \tau \frac{\Delta \mathcal{L}}{\|\nabla \mathcal{L}\|}$

② Dropout: randomly set some hidden state to 0

③ Scheduling learning rate lr

④ Layer normalization

⑤ Weight Regularization

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \|A\|_2^2$$