Advanced Control for Robotics: Homework #1

Shang Yangxing

January 19, 2021

Contents

1 ODE and Its Simulation

1.1 Equation of Pendulum Motions

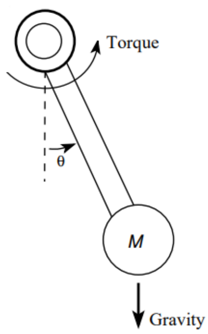

Figure 1: pendulum model

By applying the Newton's law of dynamics, a pendulum with no external force can be formulated as:

$$ml^2\ddot{\theta} + ml^2\alpha\dot{\theta} + mgl\sin\theta - T = 0. \tag{1}$$

in which,
m is mass of the ball
l is length of the rod
α is the damping constant
g is the gravitational constant
θ is angle measured between the rod and the vertical axis
T is torque of the joint, which is also the control input u
to a system of two first order equation by letting $x_1 = \theta$, $x_2 = \dot{\theta}$:

$$\dot{x_1} = x_2, \quad \dot{x_2} = -\frac{g}{l}\sin x_1 - \alpha x_2 + \frac{T}{ml^2}. \tag{2}$$

Written in standard state-space form:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{l}\sin x_1 - \alpha x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} T \tag{3}$$

$$\boldsymbol{y} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \boldsymbol{x} \tag{4}$$

1.2 Simulation of Pendulum

When assuming $m = l = 1$ with proper unit, equation (3) can be simplified as:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -g\sin x_1 - \alpha x_2 + T \end{bmatrix} \tag{5}$$

according the equation, we code the simulation as following:

```python
# -*- coding: utf-8 -*-

"""
This code simulate the pendulum system using
scipy.integrate.odeint package
"""

import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

def pendulum(var_x, unused_t, grav, damping_constant, torque):
    """
    pendulum system vector-space function
    """
    var_x1, var_x2 = var_x
    dxdt = [var_x2, \
    -grav*np.sin(var_x1) - damping_constant*var_x2 + torque]
    return dxdt

# inital condition
G = 9.8  # gravitational constant

# damping constant alpha collection of two different cases
ALPHA_COLLECTION = [0.3, 0.7]
T = 0  # the control input

# inital theta collection of two different cases
X1_0_COLLECTION = [np.pi*3/4, np.pi/4]
```

```
X2_0 = 0  # inital omega

# simulation setup
SIM_TIME = np.linspace(0, 9.9, 400)
# y = []  # the output collection of four cases

plt.subplots(2, 2, sharex='all', sharey='all', figsize=(14, 8))
# plt.figure()

# four cases
for i in range(4):
    # choose x1_0 with rem,
    # when i = 0 or 2, x1_0 is in the first case,
    # when i = 1 or 3, in another one
    x0 = [X1_0_COLLECTION[i%2], X2_0]

    # choose alpha with mod,
    # when i = 0 or 1, alpha is in the first case,
    # when i = 2 or 3, in another one
    alpha = ALPHA_COLLECTION[i//2]

    # solve
    y = odeint(pendulum, x0, SIM_TIME, args=(G, alpha, T))

    # plot
    plt.subplot(2, 2, i+1)
    plt.plot(SIM_TIME, y[:, 0], label='x1:theta')
    plt.plot(SIM_TIME, y[:, 1], label='x2:omega')
    plt.title('x1_0={:.2f},x2_0={:.2f},alpha={:.2f},T={:.2f}'\
              .format(x0[0], x0[1], alpha, T))
    plt.legend(loc='best')
    plt.ylim(-6, 6)
    if i >= 2:
        plt.xlabel('time')
    plt.grid()

# save and show
plt.savefig(r'./HW1/img/pendulum_sim.png')
plt.show()
```

and getting the results showed in the Figure 2

2 Matrix calculus

2.1 Tutorial

$$\frac{\partial}{\partial X} f(X) = \begin{bmatrix} \frac{\partial f(X)}{\partial X_{11}} & \cdots & \frac{\partial f(X)}{\partial X_{1m}} \\ \vdots & \frac{\partial f(X)}{\partial X_{ij}} & \vdots \\ \frac{\partial f(X)}{\partial X_{n1}} & \cdots & \frac{\partial f(X)}{\partial X_{nm}} \end{bmatrix} \tag{6}$$

Derivative of scalar function $f(X)$ can be calculated by taking derivatives of the scalar function with respect to each entry X_{ij} of the matrix X separately, showing as above equation (6).

Scalar function $f(X)$ project matrix variable $X \in \mathbb{R}^{n\times m}$ to a scalar $y \in \mathbb{R}^1$, so its derivative is the partial derivative, except that its results are arranged in form of a matrix, who has the same shape as X.

For instance, let's say $X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$, $f(X) = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$. So $y = f(X) = X_{11} +$

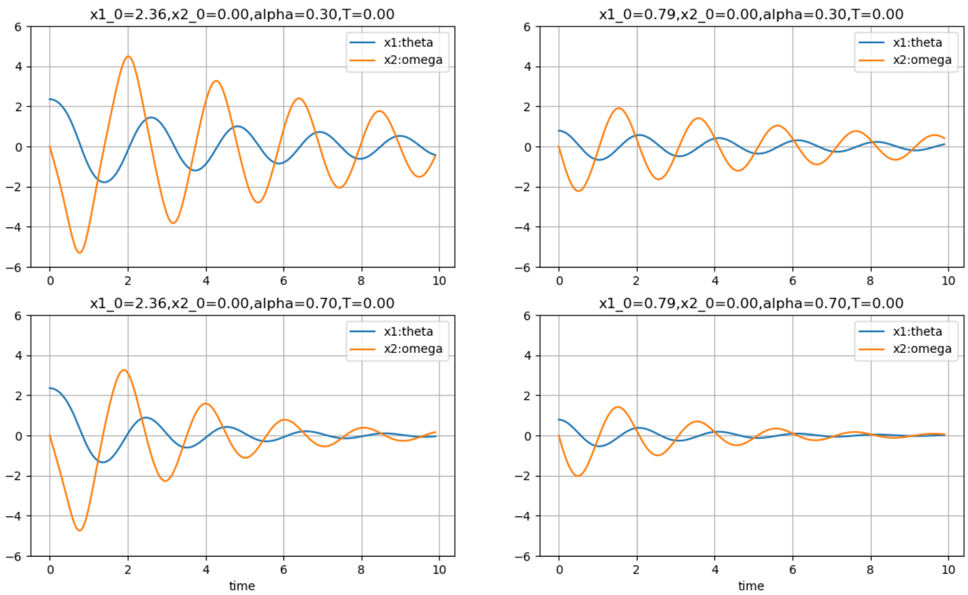

Figure 2: pendulum simulation output

$X_{12} + X_{21} + X_{22}$. And the partial derivative of $f(X)$ is

$$\frac{\partial}{\partial X} f(X) = \begin{bmatrix} \frac{\partial f(X)}{\partial X_{11}} & \frac{\partial f(X)}{\partial X_{12}} \\ \frac{\partial f(X)}{\partial X_{21}} & \frac{\partial f(X)}{\partial X_{22}} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{7}$$

2.2 Derivative of Trace

$$\begin{aligned} \frac{\partial}{\partial X} tr(AX) &= \frac{\partial}{\partial X} tr(\begin{bmatrix} A_{11}X_{11} & \cdots & A_{1m}X_{m1} \\ \vdots & A_{ij}X_{ji} & \vdots \\ A_{n1}X_{1n} & \cdots & A_{nm}X_{mn} \end{bmatrix}) \\ &= \frac{\partial}{\partial X}(A_{11}X_{11} + \cdots + A_{ij}X_{ji} + \cdots + A_{nm}X_{mn}) \\ &= \begin{bmatrix} A_{11} & \cdots & A_{n1} \\ \vdots & A_{ji} & \vdots \\ A_{1m} & \cdots & X_{mn} \end{bmatrix} = A^T \end{aligned} \tag{8}$$

in which, $\frac{\partial}{\partial X_{ij}}(A_{11}X_{11} + \cdots + A_{ij}X_{ji} + \cdots + A_{nm}X_{mn}) = A_{ji}$

2.3 Derivation

According to *The Matrix Cookbook* equation (81), we have

$$\frac{\partial x^T Q x}{\partial x} = (Q + Q^T)x \tag{9}$$

and we can derive that

$$\frac{\partial tr(xx^T)}{\partial x} = \frac{\partial}{\partial x}(x_1^2 + x_2^2 + \cdots + x_n^2) = \begin{bmatrix} 2x_1 \\ 2x_2 \\ \vdots \\ 2x_n \end{bmatrix} = 2x \tag{10}$$

comprehensive above, we get

$$\begin{aligned}\frac{\partial}{\partial x}f(x) &= \frac{\partial x^TQx}{\partial x} + \frac{\partial tr(xx^T)}{\partial x} \\ &= (Q+Q^T)x + 2x\end{aligned} \tag{11}$$

3 Inner product

3.1 Angle between Two Vectors

The inner product of two vectors is $<x, y>= \|x\|\|y\|\cos\theta$, so the angle θ equal to $\arccos\frac{<x,y>}{\|x\|\|y\|}$

3.2 Compute the Angle

Using the way to calculate the angle above, we get

$$\theta = \arccos\frac{<A, B>}{\|A\|\|B\|} \tag{12}$$

and we find that

$$<A, B>= tr(A^TB) = tr(\begin{bmatrix}-1 & 2 & 1\\ -1 & 0 & 1\\ -1 & 2 & 1\end{bmatrix}) = 0 \tag{13}$$

so, the angle between A and B is $\frac{\pi}{2}$.

4 Some linear algebra

4.1 Condition on A

Take row reducion to $Ax = b$, if any row come up with the situation that left-hand side of the equation are zeros, while the right-hand side is not, then equation $Ax = b$ has no solution, else it has at least one solution.

4.2 Compute Rank and Null

A has two linearly independent columns, so $rank(A) = 2$. Knowing $a_3 + a_1 = a_2$ and $a_4 - a_3 = a_1$, so we can get $a_3 = a_2 - a_1$ and $a_4 = a_1 + a_3 = a_1 + a_2 - a_1 = a_2$, so

$$A = [a_1, a_2, a_3, a_4] = [a_1, a_2, a_2 - a_1, a_2] \tag{14}$$

We can easily find two independent vectors satisfying $Ax = 0$

$$x_1 = \begin{bmatrix}1\\-1\\1\\0\end{bmatrix}, x_2 = \begin{bmatrix}1\\0\\1\\-1\end{bmatrix} \tag{15}$$

So, $Null(A) = \{x_1, x_2\}$

4.3 Projection onto Subspace

According to a document[1] from MIT's Course Linear Algebra, when project a vector y onto the column space of A, the projection matrix P is

$$P = A(A^TA)^{-1}A^T \tag{16}$$

So the projection from y to A is

$$p = A(A^TA)^{-1}A^Ty \tag{17}$$

[1] $https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/least-squares-determinants-and-eigenvalues/projections-onto-subspaces/MIT18\_06SCF11\_Ses2.2sum.pdf$

5 Gradient Flow

5.1 State Space Form

Let $x_1 = \omega, x_2 = \dot{\omega}, x = [x_1, x_2]^T$, we get

$$\dot{x} = \begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -\nabla l(x_1) - Ax_2 \end{bmatrix} \tag{18}$$

$$y = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{19}$$

5.2 Characterize the Equilibrium

The equilibrium point satisfies $\dot{x} = 0$, i.e.

$$\begin{cases} x_2 = 0 \\ -\nabla l(x_1) - Ax_2 = 0 \end{cases} \Rightarrow \begin{cases} \dot{\omega} = 0 \\ \nabla l(\omega) = 0 \end{cases} \tag{20}$$

5.3 Simulation of Gradient

According to the results in Problem 2.3, i.e. equation (9), we can derive that

$$\nabla l(w) = \frac{\partial}{\partial w}(w^T Q w + b^T w) = (Q + Q^T)w + b \tag{21}$$

so the system equations can be written as

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -(Q + Q^T)x_1 - Ax_2 - b \end{bmatrix} \tag{22}$$

with the system equations, we code the following

```python
# -*- coding: utf-8 -*-

"""
This code simulate the gradient algorithm system
using scipy.integrate.solve_ivp,
which is recommonded by the official
"""

import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

# initialize the condition
A = np.array([[1, 1], [1, 1]])
# choose a case
CASE = 2
if CASE == 1:
    # case 1
    Q = np.array([[5, 3], [3, 2]])
    B = np.array([1, -1])
elif CASE == 2:
    # case 2
    Q = np.array([[1, 2], [3, 4]])
    B = np.array([1, -1])
else:
     raise Exception("Don't exist case {}, else \
        choose between 1 and 2".format(CASE))
X0 = np.array([1, 1, 0, 0])
```

```

# simulation time
SIM_LEN = 50
def accelerated_gradient(unused_t, var_x):
    """
    accelerated gradient algorithm system.
    Args:
        unused_t: used by solver.
        var_x: the input x should be one-dimension.
    Returns:
        An array which is the next epoch var_x, so having the same
            shape.
    """
    x_tmp = -np.matmul(Q+Q.T, [var_x[0], var_x[1]])-\
        np.matmul(A, [var_x[2], var_x[3]])-B
    return np.array([var_x[2], var_x[3], x_tmp[0], x_tmp[1]])

def compute_loss(weights):
    """
    compute the loss, given a series of weight
    Args:
        weights: shape(4, n), n is the number of time points
    Returns:
        A list who has the length of n.
    """
    computed_loss = []
    for i in range(weights.shape[1]):
        weight = np.array(weights[0:2, i])
        computed_loss.append(np.matmul(weight.T, \
            np.matmul(Q, weight))+np.matmul(B.T, weight))
    return computed_loss

SOLUTION = solve_ivp(accelerated_gradient, [0, SIM_LEN], X0, \
    method='LSODA', dense_output=True)

TIME_SERIES = np.linspace(0, SIM_LEN, SIM_LEN*30)
WEIGHTS = SOLUTION.sol(TIME_SERIES)
LOSS = compute_loss(WEIGHTS)

plt.subplot(2, 1, 1)
plt.plot(TIME_SERIES, LOSS)
plt.legend(['loss'])
plt.grid()
plt.title('Accelerated Gradient Algorithm System Case {}'.format(
    CASE))

plt.subplot(2, 1, 2)
plt.plot(TIME_SERIES, WEIGHTS[0:2, :].T)
plt.legend(['w1', 'w2'])
plt.grid()
plt.xlabel('time')

plt.savefig(r'./HW1/img/accelerated_gradient_simulation_case_{}.png'
    \
    .format(CASE))
plt.show()
```

Due to don't knowing the matrix A, we assuming that

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{23}$$

by switch the cases in the code (line 16), i.e. change the different matrix Q and b, we get the results showed in Figure 3 and Figure 4

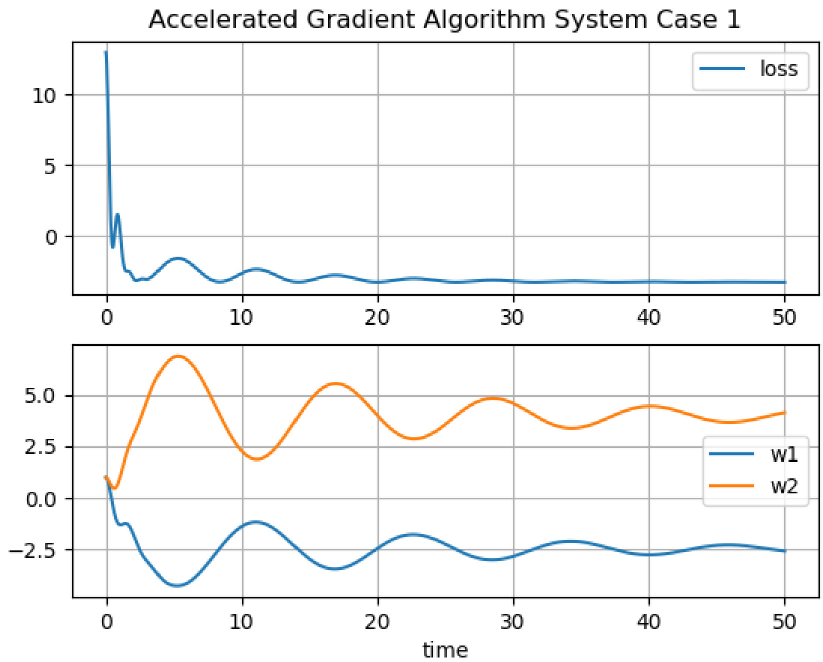

Figure 3: Accelerated gradient system simulation of case 1

In the case 1, $Q = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix}$, while in the case 2, $Q = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Obviously, Figure 3 shows that the learning in case 1 is successful, because the loss decreased to a minima and being stable after some epoches. But case 2 in Figure 4 is not that successful, the loss decreased though but not stable, and the loss value is even under -4×10^{35}, which is ridiculous.

With more experiments, we find that the value of matrix A could also influence whether the learning is successful. But it's going to be discussed here.

Also, there is a trick when coding the simulation[2], specifically, the args var_x of the system function $accelerated\_gradient$ should have one dimension, which is decided by the package $scipy.integrate.solve_ivp$. However, the initial var_x should be a matrix, so the solution is to pass a one-dimension array and reshape it inside the function.

[2]Source code can be find here: https://github.com/LoveThinkinghard/Advanced-Control-for-Robotics-Homework

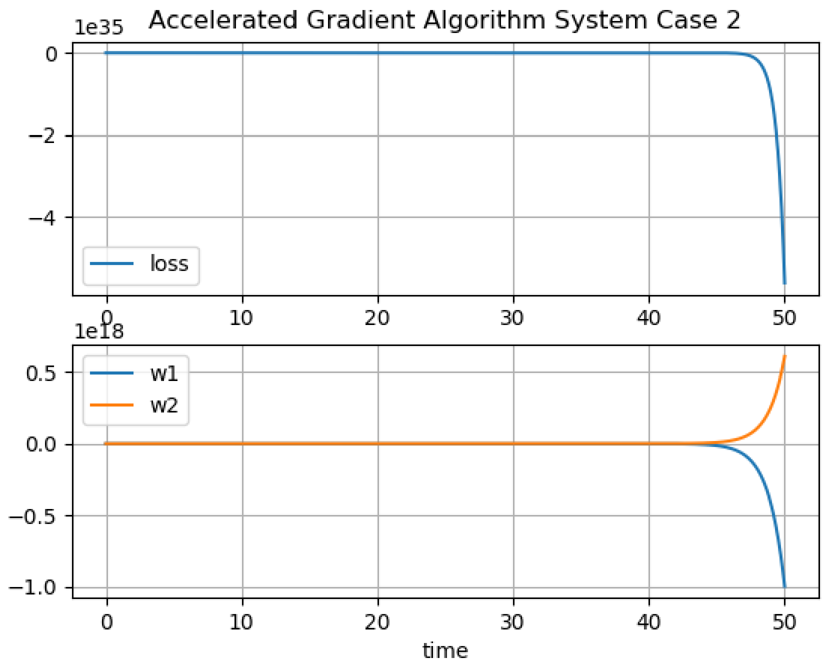

Figure 4: Accelerated gradient system simulation of case 2