

Advanced Control for Robotics: Homework #1

Shang Yangxing

January 19, 2021

Contents

1	ODE and Its Simulation	1
1.1	Equation of Pendulum Motions	1
1.2	Simulation of Pendulum	2
2	Matrix calculus	3
2.1	Tutorial	3
2.2	Derivative of Trace	4
2.3	Derivation	4
3	Inner product	5
3.1	Angle between Two Vectors	5
3.2	Compute the Angle	5
4	Some linear algebra	5
4.1	Condition on A	5
4.2	Compute Rank and Null	5
4.3	Projection Expression	5
5	Gradient Flow	5
5.1	State Space Form	5
5.2	Characterize the Equilibrium	6
5.3	Simulation of Gradient	6

1 ODE and Its Simulation

1.1 Equation of Pendulum Motions

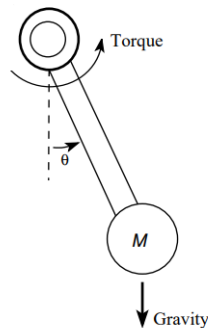


Figure 1: pendulum model

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

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

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 $x_1 = \theta$, $x_2 = \dot{\theta}$:

$$\dot{x}_1 = x_2, \quad \dot{x}_2 = -\frac{g}{l}\sin x_1 - \alpha x_2 + \frac{T}{ml^2}. \quad (2)$$

Written in standard state-space form:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{l}\sin x_1 - \alpha x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} T \quad (3)$$

$$\mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \mathbf{x} \quad (4)$$

1.2 Simulation of Pendulum

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

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -g\sin x_1 - \alpha x_2 + T \end{bmatrix} \quad (5)$$

according the equation, we code the simulation as following:

```

1  # -*- coding: utf-8 -*-
2
3  """
4  This code simulate the pendulum system using
5  scipy.integrate.odeint package
6  """
7
8  import numpy as np
9  from scipy.integrate import odeint
10 import matplotlib.pyplot as plt
11
12 def pendulum(var_x, unused_t, grav, damping_constant, torque):
13     """
14     pendulum system vector-space function
15     """
16     var_x1, var_x2 = var_x
17     dxdt = [var_x2, \
18             -grav*np.sin(var_x1) - damping_constant*var_x2 + torque]
19     return dxdt
20
21 # initial condition
22 G = 9.8 # gravitational constant
23
24 # damping constant alpha collection of two different cases
25 ALPHA_COLLECTION = [0.3, 0.7]
26 T = 0 # the control input
27
28 # initial theta collection of two different cases
29 X1_0_COLLECTION = [np.pi*3/4, np.pi/4]
```

```

30 X2_0 = 0 # initial omega
31
32 # simulation setup
33 SIM_TIME = np.linspace(0, 9.9, 400)
34 # y = [] # the output collection of four cases
35
36 plt.subplots(2, 2, sharex='all', sharey='all', figsize=(14, 8))
37 # plt.figure()
38
39 # four cases
40 for i in range(4):
41     # choose x1_0 with rem,
42     # when i = 0 or 2, x1_0 is in the first case,
43     # when i = 1 or 3, in another one
44     x0 = [X1_0_COLLECTION[i%2], X2_0]
45
46     # choose alpha with mod,
47     # when i = 0 or 1, alpha is in the first case,
48     # when i = 2 or 3, in another one
49     alpha = ALPHA_COLLECTION[i//2]
50
51     # solve
52     y = odeint(pendulum, x0, SIM_TIME, args=(G, alpha, T))
53
54     # plot
55     plt.subplot(2, 2, i+1)
56     plt.plot(SIM_TIME, y[:, 0], label='x1:theta')
57     plt.plot(SIM_TIME, y[:, 1], label='x2:omega')
58     plt.title('x1_0={:.2f}, x2_0={:.2f}, alpha={:.2f}, T={:.2f}'.\
59             .format(x0[0], x0[1], alpha, T))
60     plt.legend(loc='best')
61     plt.ylim(-6, 6)
62     if i >= 2:
63         plt.xlabel('time')
64     plt.grid()
65
66 # save and show
67 plt.savefig(r'./HW1/img/pendulum_sim.png')
68 plt.show()

```

and getting the results showed in the Figure 2

2 Matrix calculus

2.1 Tutorial

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

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 $y = f(X) = X_{11} +$

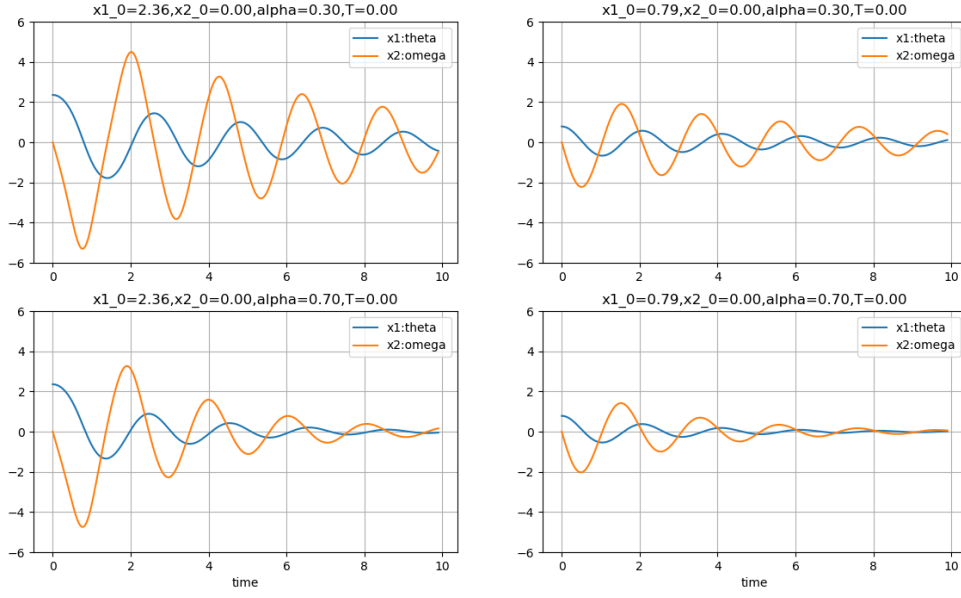


Figure 2: pendulum simulation output

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

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

2.2 Derivative of Trace

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

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

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

and we can derive that

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

comprehensive above, we get

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

3 Inner product

3.1 Angle between Two Vectors

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

3.2 Compute the Angle

Using the way to calculate the angle above, we get

$$\theta = \arccos \frac{\langle A, B \rangle}{\|A\| \|B\|}\tag{12}$$

and we find that

$$\langle A, B \rangle = \text{tr}(A^T B) = \text{tr}\left(\begin{bmatrix} -1 & 2 & 1 \\ -1 & 0 & 1 \\ -1 & 2 & 1 \end{bmatrix}\right) = 0\tag{13}$$

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

4 Some linear algebra

4.1 Condition on A

Take row reduction 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 $\text{rank}(A) = 2$. 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, $\text{Null}(A) = \{x_1, x_2\}$

4.3 Projection Expression

5 Gradient Flow

5.1 State Space Form

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

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\nabla l(x_1) - Ax_2 \end{bmatrix}\tag{16}$$

$$y = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\tag{17}$$

5.2 Characterize the Equilibrium

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

$$\begin{cases} x_2 = 0 \\ -\nabla l(x_1) - Ax_2 = 0 \end{cases} \Rightarrow \begin{cases} \dot{\omega} = 0 \\ \nabla l(\omega) = 0 \end{cases} \quad (18)$$

5.3 Simulation of Gradient

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

$$\nabla l(w) = \frac{\partial}{\partial w}(w^T Q w + b^T w) = (Q + Q^T)w + b \quad (19)$$

so the system equations can be written as

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -(Q + Q^T)x_1 - Ax_2 - b \end{bmatrix} \quad (20)$$

with the system equations, we code the following

```
1  # -*- coding: utf-8 -*-
2
3  """
4  This code simulate the gradient algorithm system
5  using scipy.integrate.solve_ivp,
6  which is recommeded by the official
7  """
8
9  import numpy as np
10 from scipy.integrate import solve_ivp
11 import matplotlib.pyplot as plt
12
13 # initialize the condition
14 A = np.array([[1, 1], [1, 1]])
15 # choose a case
16 CASE = 2
17 if CASE == 1:
18     # case 1
19     Q = np.array([[5, 3], [3, 2]])
20     B = np.array([1, -1])
21 elif CASE == 2:
22     # case 2
23     Q = np.array([[1, 2], [3, 4]])
24     B = np.array([1, -1])
25 else:
26     raise Exception("Don't exist case {0}, else \
27     choose between 1 and 2".format(CASE))
28 X0 = np.array([1, 1, 0, 0])
29
30 # simulation time
31 SIM_LEN = 50
32 def accelerated_gradient(unused_t, var_x):
33     """
34     accelerated gradient algorithm system.
35     Args:
36         unused_t: used by solver.
37         var_x: the input x should be one-dimension.
38     Returns:
39         An array which is the next epoch var_x, so having the same
40         shape.
```

```

40     """
41     x_tmp = -np.matmul(Q+Q.T, [var_x[0], var_x[1]])-\
42         np.matmul(A, [var_x[2], var_x[3]])-B
43     return np.array([var_x[2], var_x[3], x_tmp[0], x_tmp[1]])
44
45 def compute_loss(weights):
46     """
47     compute the loss, given a series of weight
48     Args:
49         weights: shape(4, n), n is the number of time points
50     Returns:
51         A list who has the length of n.
52     """
53     computed_loss = []
54     for i in range(weights.shape[1]):
55         weight = np.array(weights[0:2, i])
56         computed_loss.append(np.matmul(weight.T, \
57             np.matmul(Q, weight))+np.matmul(B.T, weight))
58     return computed_loss
59
60 SOLUTION = solve_ivp(accelerated_gradient, [0, SIM_LEN], X0, \
61     method='LSODA', dense_output=True)
62
63 TIME_SERIES = np.linspace(0, SIM_LEN, SIM_LEN*30)
64 WEIGHTS = SOLUTION.sol(TIME_SERIES)
65 LOSS = compute_loss(WEIGHTS)
66
67 plt.subplot(2, 1, 1)
68 plt.plot(TIME_SERIES, LOSS)
69 plt.legend(['loss'])
70 plt.grid()
71 plt.title('Accelerated_Gradient_Algorithm_System_Case_{}'.format(
72     CASE))
73
74 plt.subplot(2, 1, 2)
75 plt.plot(TIME_SERIES, WEIGHTS[0:2, :].T)
76 plt.legend(['w1', 'w2'])
77 plt.grid()
78 plt.xlabel('time')
79
80 plt.savefig(r'./HW1/img/accelerated_gradient_simulation_case_{}.png'.
81     \
82     .format(CASE))
83 plt.show()

```

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

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad (21)$$

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

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.

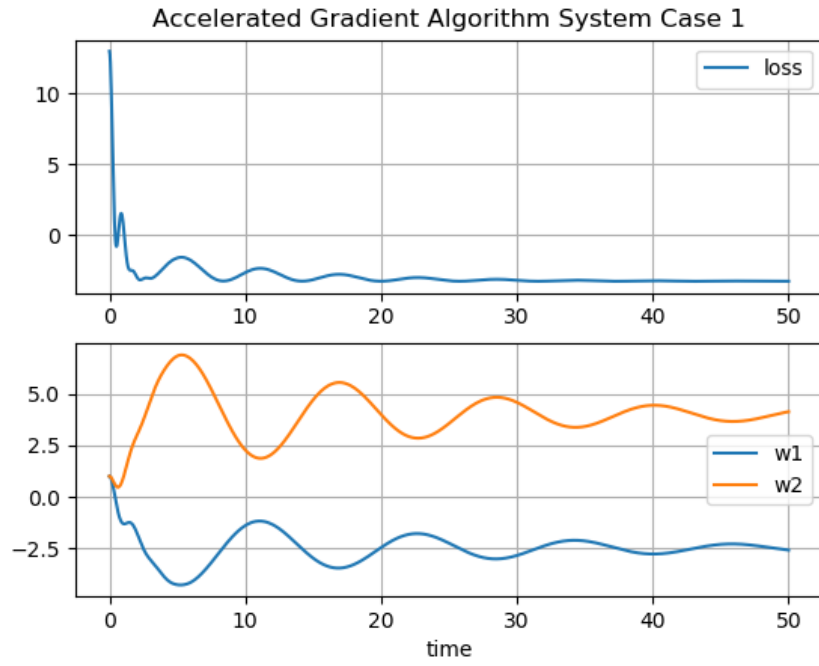


Figure 3: Accelerated gradient system simulation of case 1

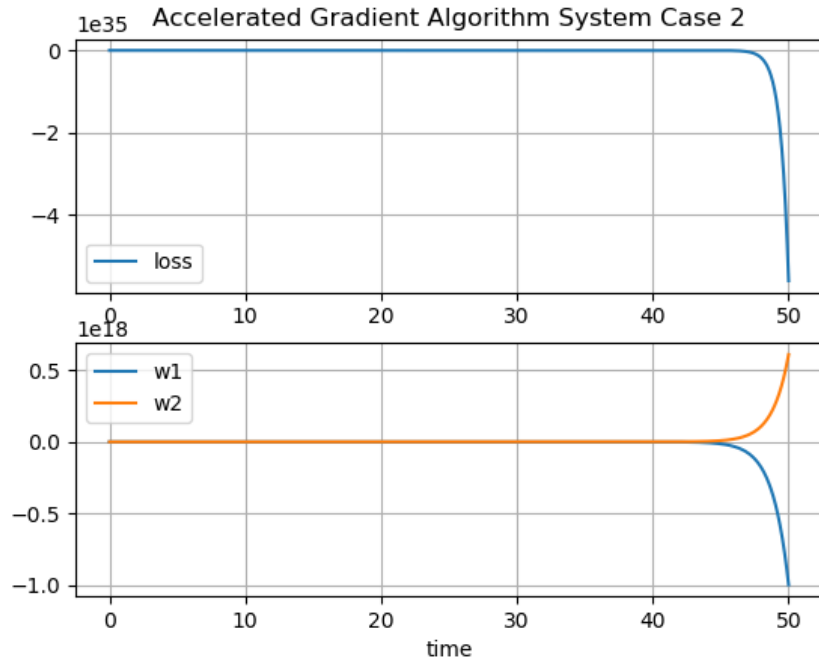


Figure 4: Accelerated gradient system simulation of case 2

Also, there is a trick when coding the simulation, 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.

Source code can be find here: <https://github.com/LoveThinkinghard/Advanced-Control-for-Robotics-Homework>