# Dairy Bike: Task 1

task-1

saurabhp e-Yantra Staff 11d

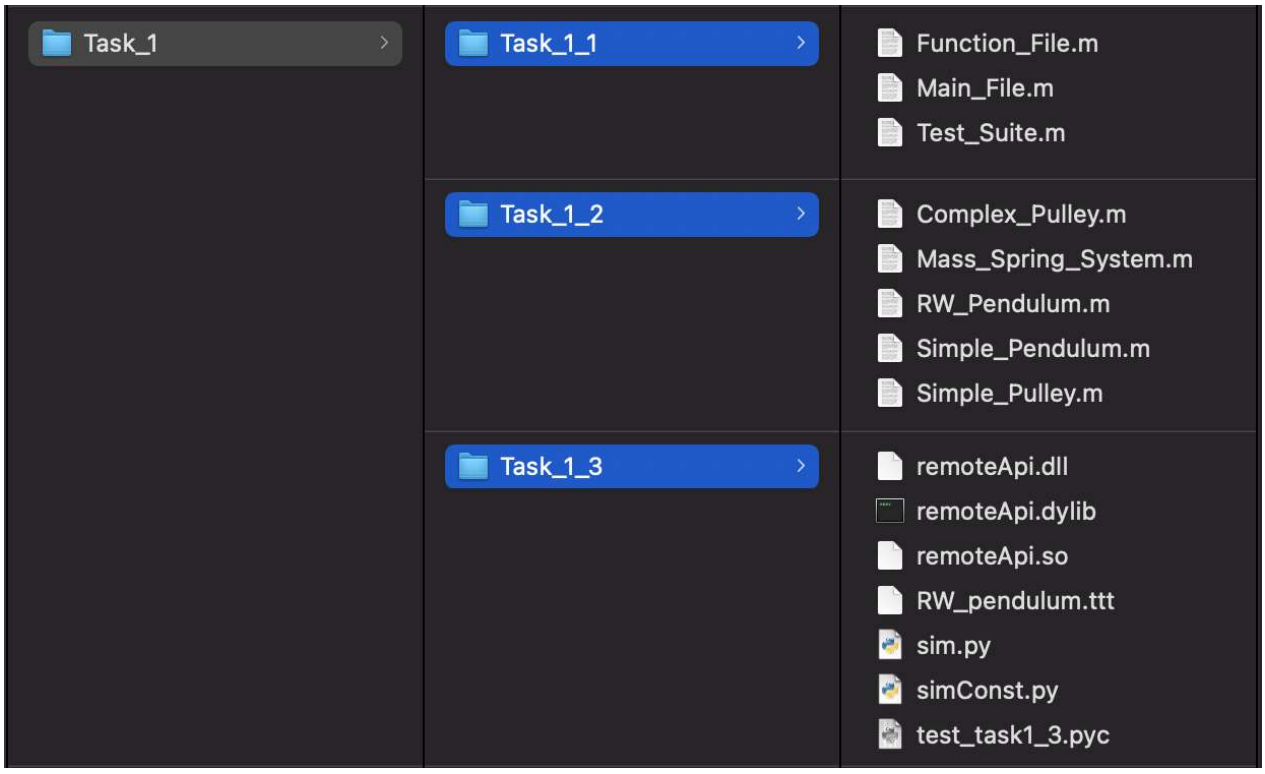
Table of contents

## Task 1

Welcome to TASK 1!

In this task we will be learning about non-linear systems, understanding equilibrium points and stability of systems. The software that you have installed in the previous task will be used throughout this task.
Download and extract **this** .zip file to get all the required files for this task.

In this .zip file you will find folder named Task_1 with the folder structure as below:




## This task is divided into three sub tasks:

### 1. Task 1.1

### 2. Task 1.2

### 3. Task 1.3

Skip to main content

**This task is divided into three sub tasks:**

1. Task 1.1 777
2. Task 1.2 398
3. Task 1.3 264

---

**UNLISTED ON OCT 13**

---

**LISTED ON OCT 18**

---

**PINNED ON OCT 19**

---

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.1 Theory (Part 1)

task-1

Hyperactive e-Yantra Staff 11d

Table of contents

## Modeling of non-linear Dynamical Systems

In mathematics and science, a non-linear system is a system in which the change of the output is not linearly proportional to the change of the input. Non-linear problems are of interest to engineers, biologists, mathematicians etc because most systems that occur in nature are inherently non-linear.

Non-linear dynamical systems that describe changes in variable over time may often appear chaotic, unpredictable or counter-intuitive in nature, contrasting with much simpler linear systems.

Typically the behavior of a non-linear system is described as a set of simultaneous equations in which the unknowns (or unknown functions in the case of differential equations) appear as variables of a polynomial of degree higher than one. Such a system is called a **non-linear system of equations.**

We will deal with dynamical systems that are modeled by a finite number of coupled first order ordinary differential equations

$$\dot{x}_1 = f_1(t, x_1, \ldots\ldots, x_n, u_1, \ldots\ldots, u_p)$$

$$\dot{x}_2 = f_2(t, x_1, \ldots\ldots, x_n, u_1, \ldots\ldots, u_p)$$

$$\cdot\cdot$$

$$\cdot\cdot$$

$$\dot{x}_n = f_n(t, x_1, \ldots\ldots, x_n, u_1, \ldots\ldots, u_p)$$

Here $\dot{x}_1, \dot{x}_2, \ldots \dot{x}_n$ denote the derivative of $x_1, x_2 \ldots x_n$ respectively with respect to time variable t and $u_1, u_2, \ldots u_p$ etc are specified input variables. We call the variables $x_1, x_2, \ldots x_n$ the **state variables**.

**State Variables** are used to to represent the memory the dynamical system has of its past or the desired variable of interest. We usually use vector notation to write these equations in a compact form.

Skip to main content

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}, u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ \vdots \\ u_p \end{bmatrix}, f(t,x,u) = \begin{bmatrix} f_1(t,x,u) \\ f_2(t,x,u) \\ \vdots \\ \vdots \\ f_n(t,x,u) \end{bmatrix}$$

We can rewrite the n first-order differential equations as one n-dimensional first-order vector differential equation

$$\dot{x} = f(t, x, u)$$

We call above equation as the **State Equation** of the system and refer to *x* as the **state** and *u* as the **input.**

---

Take the following quiz before moving forward:

## Task 1.1 Part 1: Quiz

**Modeling of non-linear Dynamical Systems**

**Task 1.1 Part 1: Quiz 564**

---

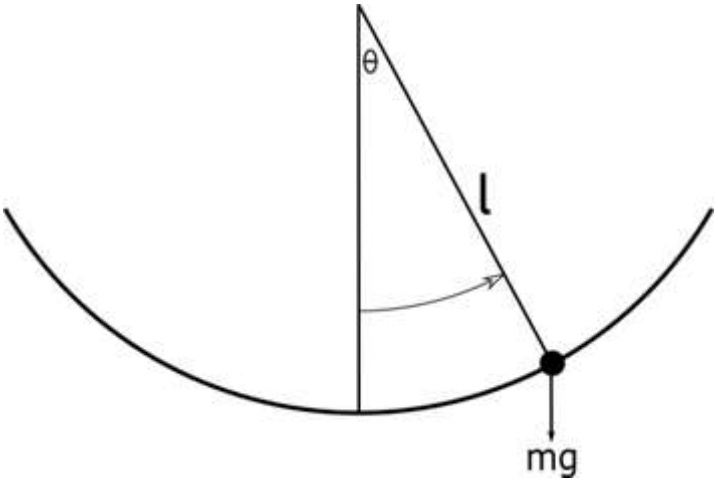
**UNLISTED ON OCT 13**

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.1 Theory (Part 2)

Hyperactive e-Yantra Staff 8d

## Modeling of a Simple Pendulum




We will start with the mathematical modeling of a simple system.

Consider a simple pendulum shown in Fig 1, where *l* denotes the length of rod and *m* denotes the mass of the bob. Assume the rod is rigid and has zero mass. Let *θ* denote the angle subtended by the rod and the vertical axis through the pivot point.

The pendulum is free to swing in the vertical plane, The bob of the pendulum moves in a circle of radius *l*. To write the equations of motion, let us identify the forces acting on the bob.

1. Downward gravitational force **mg** where **g** is acceleration due to gravity.
2. Frictional force resisting motion which can be assumed to be proportional to the speed of the bob with a coefficient of friction **k**.

Using the Newton's second law of motion, the equation of motion for the bob in the tangential direction of motion can be written as

$$ml\ddot{\theta} = -mg\sin\theta - kl\dot{\theta}$$

To obtain the State Equation for the pendulum, the state variables can be assumed as

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

The state equations for the pendulum model are:

Skip to main content

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -\frac{g}{l}\sin x_1 - \frac{k}{m}x_2$$

It is possible to find the **equilibrium points** of this system by setting

and then solving for x1 and x2.

$$0 = x_2$$

$$0 = -\frac{g}{l}\sin x_1 - \frac{k}{m}x_2$$

The equilibrium points are located at (nπ,0) for n = ±1,±2,… . From the physical descriptions of the pendulum, it is clear that there are only two equilibrium positions (0,0) and (π,0). The rest of equilibrium points are just repetitions based on number of full swings of the pendulum.

Hence this is how a simple physical system is modeled.

## Stable and Unstable Equilibrium Points

A typical problem that arises while dealing with non-linear dynamical systems is to check if a system is stable or unstable at a given equilibrium point.

**Equilibrium Point** of a system is the point at which the state of the system doesn't change. The equilibrium points can be estimated by setting $\dot{x}_1=0$ and $\dot{x}_2 = 0$ and solving the given equations for $x_1$ and $x_2$.

**Stable Equilibrium** - If a system always returns to the equilibrium point after small perturbations.

**Unstable Equilibrium** - If a system moves away from equilibrium point after small perturbations.

Consider the following set of coupled equations as given below.

$$\dot{x}_1 = -x_1 + 2x_1^3 + x_2$$

$$\dot{x}_2 = -x_1 - x_2$$

We will discuss the steps in order to compute the stability of system.

1. Calculate the equilibrium points of the system by setting the values of $\dot{x}_1 = \dot{x}_2 = 0$.

$$0 = -x_1 + 2x_1^3 + x_2$$

$$0 = -x_1 - x_2$$

Skip to main content

By solving the above two equations, the values of equilibrium point are as follows:

2. Linearize the set of equations by calculating the Jacobian.

   Behavior of Non-linear systems is very hard to analyse. Linearization is a method which involves creating a linear approximation of a non-linear system that is valid in a small region around the operating point (in this case, the equilibrium points).

   In order to linearize a set of equations, we need to first calculate the Jacobian for a set of equations. You can read more about Jacobian **here**

   Consider the following:

   The Jacobian *J* for the set of equations can be calculated as:

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 6x_1^2 - 1 & 1 \\ -1 & -1 \end{bmatrix}$$

3. Substitute the value of the equilibrium points in the matrix J to find 3 matrices $J_1$, $J_2$ and $J_3$.

$$J_1 = \begin{bmatrix} 6(0)^2 - 1 & 1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}$$

$$J_2 = \begin{bmatrix} 6(1)^2 - 1 & 1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ -1 & -1 \end{bmatrix}$$

$$J_3 = \begin{bmatrix} 6(-1)^2 - 1 & 1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ -1 & -1 \end{bmatrix}$$

4. Calculate the eigenvalues of each of the matrices.

   Eigen values of state matrices represent poles of a system. The poles decides the stability of a system and if the poles are on the negative half of the complex plane i.e. they have the negative real part then the system is stable and if the real part is positive the system is unstable. A system is marginally stable if it has simple poles (non repeated) on imaginary axis and unstable if it is repeated. Hence by intuition you can see that for simple pendulum case the pendulum in downward position is stable and in upright position it is unstable. Find the Jacobian around the two equilibrium points and verify the same.

**Skip to main content**

The eigenvalues can be calculated by constructing the characteristic equation of the matrix and equating it to zero.

For equilibrium point (0,0) :-

$$\left|sI - J_1\right| = 0$$

$$\left|s\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} - \begin{bmatrix}-1 & 1\\ -1 & -1\end{bmatrix}\right| = 0$$

$$\begin{vmatrix}(s+1) & -1\\ 1 & (s+1)\end{vmatrix} = 0$$

$$(s+1)^2 + 1 = 0$$

$$s = (-1+i), (-1-i)$$

The eigen values for J1 are () and ().

For equilibrium point (-1,1) and (1,-1):

$$\left|sI - J_2\right| = 0$$

$$\left|s\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} - \begin{bmatrix}5 & 1\\ -1 & -1\end{bmatrix}\right| = 0$$

$$\begin{vmatrix}(s-5) & -1\\ 1 & (s+1)\end{vmatrix} = 0$$

$$(s-5)(s+1) + 1 = 0$$

$$s = (2+2\sqrt{2}), (2-2\sqrt{2})$$

5. Check the real part of the eigenvalues calculated for each equilibrium points.

For each Equilibrium point

● If all the eigenvalues have negative real part, the system is **Stable** at the given Equilibrium point.

● If even one of the eigenvalues has positive real part, the system is **Unstable** at the given Equilibrium point.

For Equilibrium Point (0,0), the eigenvalues are (-1-i) and (-1+i). Since both eigenvalues have negative real part, the system is **Stable**.

For Equilibrium Points (1,-1) and (-1,1), the eigenvalues are (2+2√2) and (2-2√2). Since ues has a positive real part, the system is **Unstable**.

Skip to main content

Take the following quiz before moving forward:

## Task 1.1 Part 2: Quiz

Task 1.1 Part 2: Quiz 376

UNLISTED ON OCT 17

CLOSED 2 DAYS AGO

# Dairy Bike: Task 1.1 Practical

task-1

saurabhp e-Yantra Staff 11d

Table of contents

## Task Instructions

In this task, we will explain how to solve a system of non-linear equations to find the equilibrium points and then check the stability of the system at the given equilibrium points using Octave. The theory of this has already been explained in **"Task_Theory"**. Please make sure you have read and thoroughly understood that document.

If you are new to Octave we would advise you to go through this **link** . Octave syntax is very easy to understand. If you have prior experience with MATLAB, Python (or pretty much any programming language), then it shouldn't be hard to grasp.

In the Task folder, you will find the following scripts.

1. Main_File.m
2. Function_File.m
3. Test_Suite.m

We will examine the following files one-by-one.

---

### 1. Main_File.m

```
1. 1;
2. Function_File;
3. pkg load symbolic                # Load the octave symbolic library
4. syms x1 x2                       # Define symbolic variables x1 and x1
5. x1_dot = -x1 + 2*x1^3 + x2;      # Write the expressions for x1_dot and x2
6. x2_dot = -x1 - x2;               # YOU CAN MODIFY THESE 2 LINES TO TEST OU
7. [x_1 x_2  jacobians  eigen_values  stability] = main_function (x1_dot, x
```

**Lets now look at the main file line by line:**

**Line 3** - It loads the symbolic library for Octave. When we want to define equations containing variables like x, y, z etc, we need to use the symbolic library.

**Line 4** - Declares 2 symbolic variables x1 and x2 (to denote $x_1$ and $x_2$ )

Skip to main content

| **Lets now look at the main file line by line:** |
|---|
| **Line 5-6** - Defines a coupled set of non-linear equations using x1 and x2 (x1_dot and x2_dot are used to denote $\dot{x}_1$ and $\dot{x}_2$). |
| **Line 7** - It is a function call to a function named main_function(). main_function() is defined in Function_File.m and takes 2 arguments (x1_dot and x2_dot). We will explain this function later. |

**Main_File** can be used to test your code for different sets of equations on your side. You can change the x1_dot and x2_dot values to specify different equations for your code to test. **Please do not modify any other line in this script.**

### 2. Function_File.m (This is where the magic happens!!)

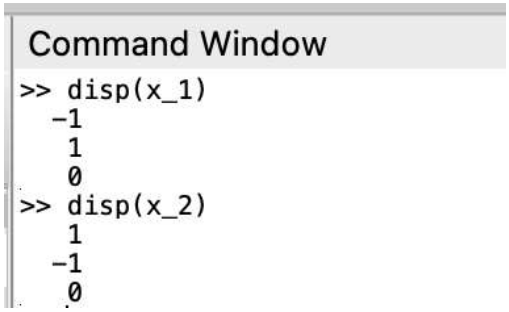
There are 4 functions defined in this script. You are required to complete 3 of them in order to complete the task successfully.

```
find_equilibrium_points()

1. function [x_1, x_2] = find_equilibrium_points(x1_dot, x2_dot)

2. x1_dot == 0;

3. x2_dot == 0;

#######################ADD YOUR CODE HERE############################

#####################################################################
4. x_1=double(x_1);

5. x_2=double(x_2);

6. endfunction
```

| **Lets now look at the function line by line:** |
|---|
| This function is pretty straightforward |
| find_equilibrium_points() takes **x1_dot** and **x2_dot** as arguments. |
| Line 2 and 3 equate the expressions specified by **x1_dot** and **x2_dot** equal to zero. |
| The function returns the set of equilibrium points for **x1_dot = 0** and **x2_dot = 0**. This set is stored in the array **[x_1, x_2]**. |
| Please complete the code in this function so that it calculates the equilibrium points for the set of equations and stores it in the variable **[x_1, x_2].** |

When you display **[x_1, x_2]** (using disp() function in octave), the structure should be as shown:

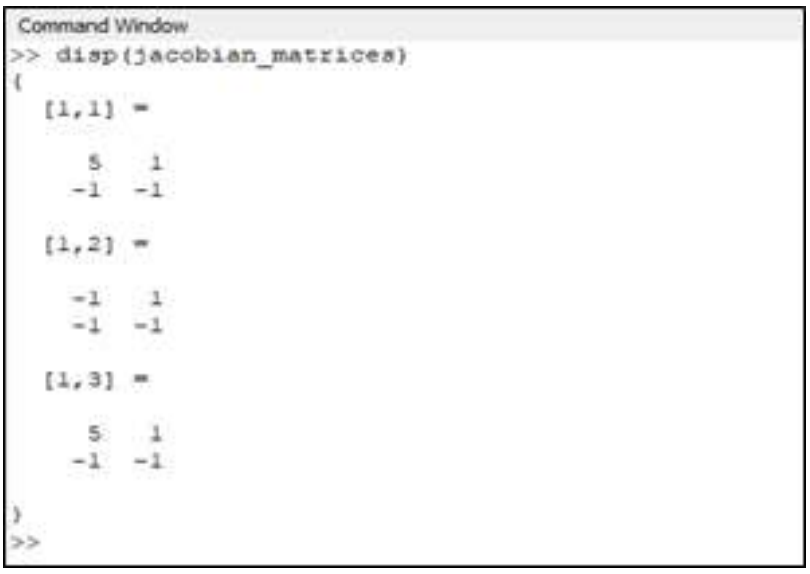
Skip to main content




```
find_jacobian_matrices()

1. function jacobian_matrices = find_jacobian_matrices(x_1, x_2, x1_dot, x2

2. syms x1 x2;

3. solutions = [x_1, x_2];

4. jacobian_matrices = {};

#####################   ADD YOUR CODE HERE  ####################

#########################################################################

5. endfunction
```

- This function takes the **equilibrium points (x_1, x_2)** and **x1_dot** and **x2_dot** as arguments.
- It computes the jacobian matrix for **x1_dot** and **x2_dot** (It should be a 2x2 symbolic array).
- It then substitutes calculated values of **x1** and **x2** for each of the equilibrium points. This function returns a variable called **jacobian_matrices**. It is a cell array in which each element is a 2x2 J matrix calculated for each of the corresponding equilibrium points.
- You are not allowed to change any code already written in this function. You are required to add your own code in the space provided.
- Line 3 - solution x_1 & x_2 are combined in one array **solutions**
- Line 4 - empty cell array **jacobian_matrices** is initialized.
- You are required to add code which does the following:
  - Computes the jacobian of **x1_dot** and **x2_dot**.
  - For each of the equilibrium points, substitute the calculated values of x1 and x2 in the jacobian and form a 2x2 matrix.
  - Store that jacobian matrix as an element of the cell array **jacobian_matrices**.

Skip to main content y the jacobian matrices cell array (using disp() function in octave), the cell array structure should be similar to the following:



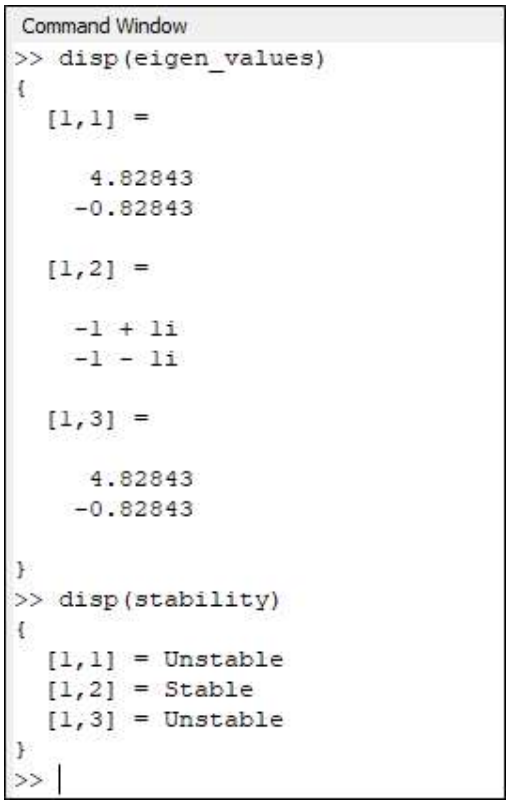

check_eigen_values()

```
1. function [eigen_values stability] =  check_eigen_values(x_1, x_2, jacobi
2. stability = {};
3. eigen_values = {};
4. for k = 1:length(jacobian_matrices)
5.      matrix = jacobian_matrices{k};
6.      flag = 1;
#####################   ADD YOUR CODE HERE  ######################
#########################################################################
7.      if flag == 1
8.          fprintf("The system is stable for equilibrium point(%d, %d)\n",
9.          stability{k} = "Stable";
10.     else
11.         fprintf("The system is unstable for equilibrium point (%d, %d)
12.         stability{k} = "Unstable";
13.     endif
14. endfor
```

Skip to main content

15. endfunction

- This function takes the **x_1, x_2** and **jacobian_matrices** as input. For each jacobian matrix stored in **jacobian_matrices**, the eigenvalues of matrix are calculated and stored in the cell array **eigen_values**. Subsequently the eigenvalues are checked in this function. If for any jacobian matrix the eigenvalues have positive real part, the system is unstable at the corresponding equilibrium point. If all eigenvalues have negative real part, the system is stable at the corresponding equilibrium point.
- Two empty cell arrays **stability** and **eigen_values** are defined. A for-loop is iterated through the length of **jacobian_matrices**. Within the for-loop, flag = 1 is initialized. You are required to write code which does the following:
  - Find out the eigenvalues for the current jacobian matrix (value stored in matrix)
  - Check real part of all eigenvalues of the matrix. If all eigenvalues have negative real part, then flag is set equal to 1.
  - If even one eigenvalue is positive (greater than zero), the flag is set to 0.
  - Store the eigenvalues calculated in the cell array **eigen_values**.
- Based on value of flag, the stability of system is reported in the if-else statement.
- When you display the **eigen_values** and **stability** cell arrays (using disp() in octave) the output should be similar to the following:




```
main_function()

1. function [x_1 x_2 jacobians eigen_values stability] = main_function(x1_d

2.          pkg load symbolic;

3.          syms x1 x2;
```

Skip to main content `_2] = find_equilibrium_points(x1_dot, x2_dot);`

```
5.          jacobians = find_jacobian_matrices(x_1, x_2, x1_dot, x2_dot);

6.          [eigen_values stability] = check_eigen_values (x_1, x_2, jacobi

7. endfunction
```

- This function puts together all the pieces.
- It takes x1_dot and x2_dot as argument. First the equilibrium points are calculated. For each equilibrium point, the jacobian matrix is calculated. Then the stability for each equilibrium point is determined by computing the eigen_values and checking the real parts of eigen values.
- This equation returns **x_1, x_2**, **jacobians**, **eigen_values**, **stability**.
- You are not allowed to make any changes to this function. You need to run it as it is.
- After you have modified the functions explained above as instructed. You need to test your solution.
- To test your script, you need to run Main_File.m in octave. If your solution is correct you will see the following octave prompt:

Skip to main content

```
>> Main_File

The system is unstable for equilibrium point (-1, 1)
The system is unstable for equilibrium point (1, -1)
The system is stable for equilibrium point (0, 0)
x_1 =

  -1
   1
   0

x_2 =

   1
  -1
   0

jacobians =
{
  [1,1] =

     5   1
    -1  -1

  [1,2] =

     5   1
    -1  -1

  [1,3] =

    -1   1
    -1  -1

}

eigen_values =
{
  [1,1] =

    4.82843
   -0.82843

  [1,2] =

    4.82843
   -0.82843

  [1,3] =

   -1 + 1i
   -1 - 1i

}

stability =
{
  [1,1] = Unstable
  [1,2] = Unstable
  [1,3] = Stable
}
```

Skip to main content your solution)

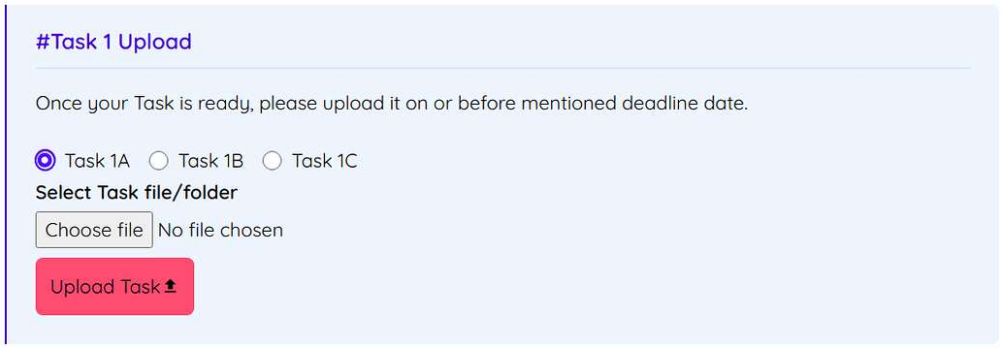
- Test_Suite.m is used for testing your solution. You are not allowed to make any changes in this script.
- The Test_Suite script has a set of non-linear equations. If your code runs successfully for the given equations then the output should be as follows:

```
>> Test_Suite

The system is unstable for equilibrium point (-1, 1)
The system is unstable for equilibrium point (1, -1)
The system is stable for equilibrium point (0, 0)

Checking output values with sample output
Output matched


Checking datatype for the output genereated

checking dataype of elements of x_1
datatype matched
datatype matched
datatype matched

checking dataype of elements of x_2
datatype matched
datatype matched
datatype matched

checking dataype of elements of jacobians
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched

checking dataype of elements of eigen_values
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched
datatype matched

Checking datatype of elements of stability
datatype matched
datatype matched
datatype matched
>>
```

- If your Test_Suite runs successfully, your **Function_File.m** file is ready to be submitted
  - For successful completion of **Task 1_1**, upload the **Function_File.m** file on the **portal.**
  - Now, open the portal and go to **Task 1**. In the **Task 1 Upload** section select **Task 1A.**
  - Now select **Choose file** button to upload the file. From the dialogue box, select the

Skip to main content **en.**

- You shall see the file name **Function_File.m** in text-box besides the **Choose file** button. Click on **Upload Task** button to submit the file.




That's it !! Task 1.1 is complete!!

**Congrats!!**

**Next >>**

**Task Instructions**

1. **Main_File.m**
2. **Function_File.m (This is where the magic happens!!)**
3. **Test_Suite.m (Testing your solution)**

**UNLISTED ON OCT 13**

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.2 Theory (Part 1)

task-1

saurabhp e-Yantra Staff 11d

Table of contents

## Mathematical Modeling of a system

In the previous Task, we had discussed about the various steps involved in testing the stability of a system. In this document, we will be discussing the **Euler-Lagrange method** to derive the equations of motion of a given system.

Before you start, you might need to recapitulate a few topics if you want to fully understand what we are going to explain here.

You will need a good understanding of classical mechanics. *"Concepts of Physics by Prof.* H.C. *Verma"* is a great place to brush up on those concepts.

You will also need to understand mathematical concepts like partial differentiation, jacobians, solving equations with two or more variables etc.

### Euler-Lagrange Method

The Euler-Lagrange method states that the equations of motion of a system can be obtained by solving the following equation:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = 0$$

(Assuming that there are no non-conservative forces acting on the system)

Here,

L is the Lagrangian which is the difference between the Kinetic energy and Potential energy of the system.

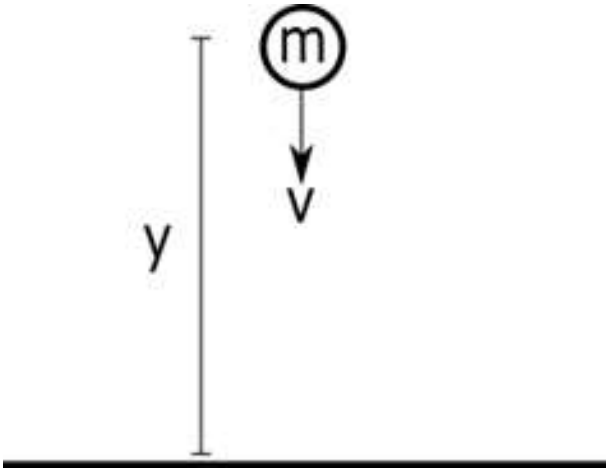
Hence

$$L = K.E - P.E$$

x and ẋ are the state variables (In our case here, position and velocity respectively) in generalized coordinate system.

We will try to understand this using an example.

( Skip to main content stem:




A point mass m is raised to height y. We need to calculate the equations of motion using the Euler-Lagrange method.

Firstly we calculate the KE and PE. Then use those values to calculate the Lagrangian L.

$$PE = mgy \qquad (1)$$

$$KE = \frac{1}{2}mv^2 = \frac{1}{2}m\dot{y}^2 \qquad (2)$$

$$L = KE - PE = \frac{1}{2}m\dot{y}^2 - mgy \qquad (3)$$

Equation (1) is self explanatory. The mass is raised to height y. So the potential energy stored in mass will by mgy.

Equation (2) is slightly tricky to understand. We know that kinetic energy of a point mass is (1/2) x *mass* x (velocity)^2. Now velocity v is nothing but rate of change of y with respect to t. Hence v can be written as dy/dt or ẏ

Equation (3) represents the Lagrangian (L) which is the difference between the KE and PE of the system.

Now we calculate the Euler Lagrange equations of motion.

$$\frac{\partial L}{\partial y} = \frac{\partial}{\partial y}\left(\frac{1}{2}m\dot{y}^2 - mgy\right) = -mg \qquad (4)$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{y}}\right) = \frac{d}{dt}\left(\frac{\partial}{\partial \dot{y}}\left(\frac{1}{2}m\dot{y}^2 - mgy\right)\right) = m\ddot{y} \qquad (5)$$

*Hence*

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{y}}\right) - \frac{\partial L}{\partial y} = 0$$

$$\Rightarrow m\ddot{y} - (-mg) = 0$$

$$\Rightarrow \ddot{y} = -g \qquad (6)$$

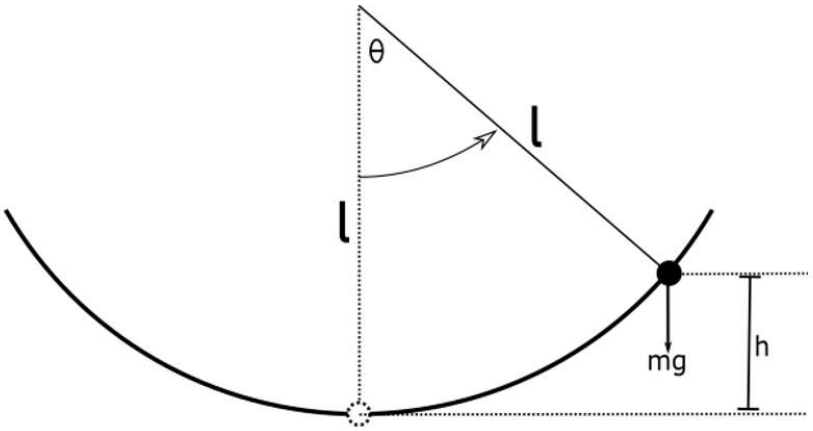
Skip to main content

Equation (6) gives the final answer. Here ÿ represents the acceleration.

Equation (6) makes sense as only gravitational force is acting on the system. Hence the acceleration of the system is the acceleration due to gravity.

If we had used the Newton's laws of motion, we would have arrived at the same result, albeit in a different way.

Let us now consider a somewhat more complex example.




We have our pendulum equation whose equations of motion we demonstrated in previous Task using Newtons Laws of motions. Now we will demonstrate the same using Euler-Lagrangian method.

In this system, we have a pendulum. The mass of the bob is given as m. The length of rod to which the bob is attached to is l. We have assumed the rod to be rigid and have no mass. So all the mass is concentrated to the bob.

While swinging, at any arbitrary point in the pendulum's trajectory, the pendulum can assumed to be at a height h from the bottom. h can be written as a function of θ where θ is the angle the pendulum bob makes with the vertical.

$$h = l - l\cos\theta$$

First, we need to calculate the Lagrangian L for this system. For that we need to compute the kinetic energy and potential energy of this system.

Calculating the potential energy is pretty straightforward.

$$PE = mgh = mg(l - l\cos\theta) = mgl(1 - \cos\theta) \qquad (7)$$

The kinetic energy will be defined by (1/2) x mass x velocity^2. Here the velocity is the tangential velocity of the bob. We can take x and y components of velocity v and solve for kinetic energy using those equations.

However, we can use rotational mechanics to make our calculations simpler. Since the pendulum bob is oscillating in a circular trajectory, the kinetic energy can be given by

Skip to main content

$$KE = \frac{1}{2} I\omega^2$$

Where I is the moment of inertia and ω is the angular velocity.

But we know

$$I = ml^2 \qquad and \qquad \omega = \dot{\theta}$$

Angular velocity ω can be written as rate of change of θ with respect to time. Hence we can write ω= (dθ/dt).

Therefore we have the expression for kinetic energy

$$KE = \frac{1}{2} I\omega^2 = \frac{1}{2} ml^2\dot{\theta}^2 \qquad (8)$$

Now we have the expressions for PE and KE we will calculate the Lagrangian L and use it to calculate the equations of motion for this system.

$$L = KE - PE = \frac{1}{2} ml^2\dot{\theta}^2 - mgl(1-\cos\theta) \qquad (9)$$

Since L is a function of θ, we need to select $\theta$ and θ as state variables of the system. Hence the equations of motion can be calculated as:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = 0$$

$$\frac{d}{dt}\left(\frac{\partial}{\partial \dot{\theta}}\left(\frac{1}{2} ml^2\dot{\theta}^2 - mgl(1-\cos\theta)\right)\right) - \frac{\partial}{\partial \theta}\left(\frac{1}{2} ml^2\dot{\theta}^2 - mgl(1-\cos\theta)\right) = 0$$

$$\Rightarrow \qquad ml^2\ddot{\theta} + mgl\sin\theta = 0$$

$$\ddot{\theta} = \frac{-g}{l}\sin\theta \qquad (10)$$

In previous Task, we had used the same pendulum example and calculated the equations of motion for pendulum using Newton's laws. We can confirm that the same equations have been derived using the Euler-Lagrange method.

Suppose we take

$$x_1 = \theta$$

Skip to main content

$$x_2 = \dot{\theta}$$

We can express the equations we formed in the following way

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{-g}{l}\sin x_1$$

In this way we have a two equations that govern our system.

What happens if there is any external force acting on the system?

Consider the following system as given in Fig 3.

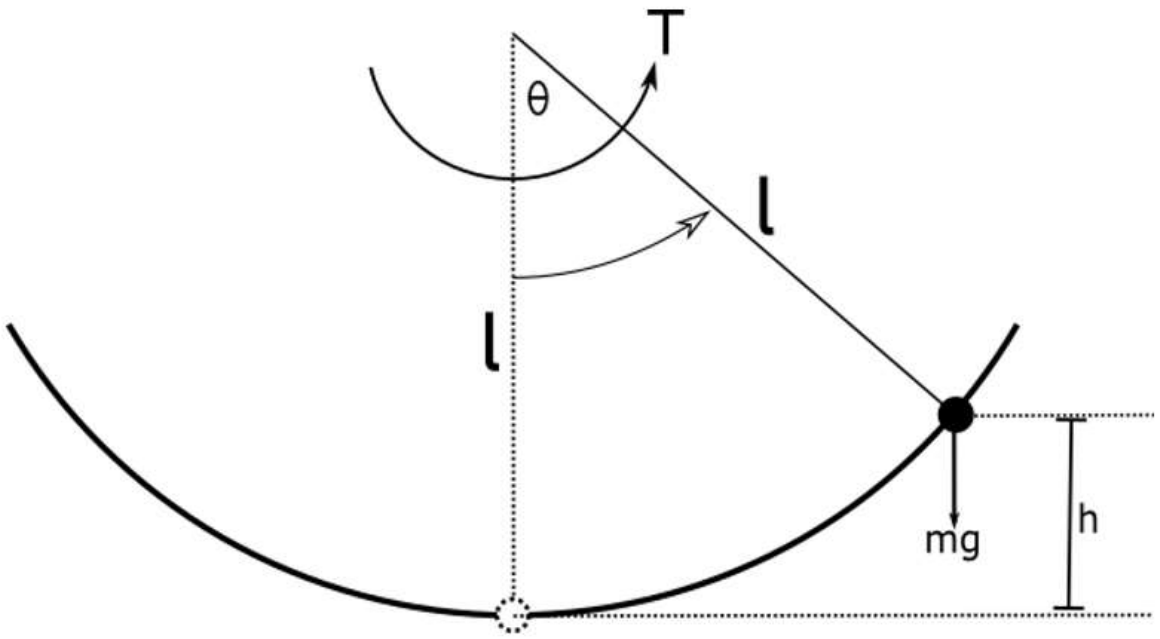



Figure 3: Pendulum with external torque

In the given simple pendulum system, we have applied an external torque to the system.

In this case, the Euler-Lagrange equation formed will be slightly different.

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = T$$

Any non-conservative force acting on the system (Since states chosen are angular position and velocity that's why force should also be taken as angular force i.e. Torque. In case we use linear motions as in the first example then we'll use external linear force on the right hand side.) appears on the right side of the Euler-Lagrange equation. Consequently the equations of motion derived for this pendulum system will be as follows:

Skip to main content

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{-g}{l}\sin x_1 + \frac{1}{ml^2}T$$

You can see that there is an additional term in the second equation. We can check if this term is dimensionally correct.

We know $x_1$ is the angular position θ of the pendulum bob (with respect to vertical) and $x_2$ is the angular velocity *theta.dot* of the bob. Hence $\dot{x}_1 = x_2$ will correspond to the angular velocity *theta.dot* and $\dot{x}_2$ will be the angular acceleration *theta.double.dot*. Let the angular acceleration be denoted by α.

The units and dimensions of α are $rad/s^2$ and [T^(-2) ] respectively.

We know T=Iα (where T= torque, I = moment of inertia, α=angular acceleration). And $I = ml^2$.

(T/ml^2) equals angular acceleration α. Hence the last term is an angular acceleration term which is consistent with the equation. We can also calculate the dimensions of this term to verify. It will always come as [T^(-2) ]. This method is helpful to verify if or equations are valid.

---

Take the following quiz before moving forward:

## Task 1.2 Part 1: Quiz

**Mathematical Modeling of a system**

**Euler-Lagrange Method**

**Task 1.2 Part 1: Quiz 200**

**UNLISTED ON OCT 13**

last visit

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.2 Theory (Part 2)

Hyperactive e-Yantra Staff 7d

## Mathematical Modeling of a system

## Introduction to State Space Analysis

We had briefly covered State Variables and State Equations in previous Task. In this section we will further elaborate on that topic and discuss the various control techniques that are associated with that.

In control engineering, a **state-space representation** is a mathematical model of a physical system as a set of input, output and state variables related by first-order differential equations or difference equations. **State variables** are variables whose values evolve through time in a way that depends on the values they have at any given time and also depends on the externally imposed values of input variables. Output variables' values depend on the values of the state variables.

The state space equations for a linear time invariant system (LTI) system can be given as follows:

$$\text{State Equation} \Rightarrow \dot{x}(t) = Ax(t) + Bu(t)$$
$$\text{Output Equation} \Rightarrow y(t) = Cx(t) + Du(t)$$

Here

$x(t)$- State Vector (n x 1 matrix)

$y(t)$- Output Vector (p x 1 matrix)

$u(t)$- Input Vector (m x 1 matrix)

A - State (or system) matrix (n x n matrix)

B - Input matrix (n x m matrix)

C - Output Matrix (p x n matrix)

D - Feed-forward matrix (p x m matrix)

where p, m, n are:

We won't go into the theory of how these equations came into being. That's a lot of complicated math that cannot be covered here. You can refer to good Control Systems books.

Skip to main content

Consider a set of equations:

$$\dot{x}_1 = x_1 x_2 - x_2$$
$$\dot{x}_2 = 2x_1 - x_2^2 \qquad (1)$$

We want to express this set of equations into the form

$$\dot{x} = Ax \qquad (2)$$

Notice, we have neglected the **Bu** term in this equation. That's because our system doesn't have any input. It only has state variables $x_1$ and $x_2$.

Can we express the set of equations (1) in terms of (2)??

The answer is no, we cannot. (1) is a set of non linear equations while (2) is a set of linear equations. However, if we linearize (1), it might be possible to express (1) in terms of (2).

How do we linearize (1)? That was basically the whole point of previous Task.

**1. Find the equilibrium points**

We want to find the point around which the system is stable. To find the equilibrium points we need to set $\dot{x}_1 = 0$ and $\dot{x}_2 = 0$. Then solve the equations for

$x_1$ and $x_2$.

$$0 = x_1 x_2 - x_2$$

$$0 = 2x_1 - x_2^2 \ldots\ldots\ldots\ldots\ldots\ldots (3)$$

If we solve (3) for $x_1$ and $x_2$ we will get the equilibrium points as (0,0), (1,√2) and (1,-√2).

**2. Calculate the jacobian of the system of equations**

The jacobian J for the system of equations (3) will be:

$$J = \begin{bmatrix} \frac{\partial(x_1 x_2 - x_2)}{\partial x_1} & \frac{\partial(x_1 x_2 - x_2)}{\partial x_2} \\ \frac{\partial(2x_1 - x_2^2)}{\partial x_1} & \frac{\partial(2x_1 - x_2^2)}{\partial x_2} \end{bmatrix}$$

$$J = \begin{bmatrix} x_2 & x_1 - 1 \\ 2 & -2x_2 \end{bmatrix}$$

**3. For each equilibrium point, calculate the value of the jacobian.**

The values of jacobian for each equilibrium point will be given as:

Skip to main content

$$J_{(0,0)} = \begin{bmatrix} 0 & 0-1 \\ 2 & -2(0) \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 2 & 0 \end{bmatrix}$$
$$J_{(1,\sqrt{2})} = \begin{bmatrix} \sqrt{2} & 1-1 \\ 2 & -2(\sqrt{2}) \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ 2 & -2\sqrt{2} \end{bmatrix}$$
$$J_{(1,-\sqrt{2})} = \begin{bmatrix} -\sqrt{2} & 1-1 \\ 2 & -2(-\sqrt{2}) \end{bmatrix} = \begin{bmatrix} -\sqrt{2} & 0 \\ 2 & 2\sqrt{2} \end{bmatrix}$$

**4. Construct the state equation for each equilibrium point.**

The state equation for equilibrium point (0,0) will be:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Here,

$$\dot{x} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$A = \begin{bmatrix} 0 & -1 \\ 2 & 0 \end{bmatrix}$$

Therefore the set of equations has been expressed in the form:

$$\dot{x} = Ax$$

It is very important to note that this approximation of the set of non-linear equations given in (3) will only hold true for point close to the equilibrium point (0,0).

This means that around the vicinity of the equilibrium point (0,0), the non-linear system will behave like a linear system and the state equation given above will hold true around the vicinity of that point.

Likewise, the state equations for equilibrium points (1,√2) and (1,-√2) are:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ 2 & -2\sqrt{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

and

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -\sqrt{2} & 0 \\ 2 & 2\sqrt{2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

## Stability

Skip to main content

We can find out whether the system is stable or unstable at each of the equilibrium points by finding out the eigenvalues of the A matrix. If any of the eigenvalues have a positive real part,

the system will be unstable.

So, for equilibrium point (1, √2) of the system, the eigenvalues will be -2.824 and 1.414. Hence system will be unstable.

**Introducing Control Input**

Let us consider the Pendulum with external applied torque system.

We derived the equations for this system as:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \frac{-g}{l}\sin x_1 + \frac{1}{ml^2}T$$

We can apply the same linearization technique explained above.

**1. Find the equilibrium points.**

If we set $\dot{x}_1 = 0$ and $\dot{x}_2 = 0$, we will find the equilibrium points of this system as (nπ,0) where n=0,±1,±2,… .From the physical descriptions of the pendulum, it is clear that there are only two equilibrium positions (0,0) and (π,0). The rest of equilibrium points are just repetitions based on number of full swings of the pendulum.

The equilibrium point (0,0) will be when the pendulum bob is vertically downwards.

The equilibrium point (π,0) will be when the pendulum bob is vertically upwards.

Intuitively, we can guess that the system will be stable at equilibrium point (0,0) and unstable at equilibrium point (π,0). Let us see if our intuition is correct.

**2. Calculate the jacobian of the system of equations.**

The jacobian $J_1$ for the A matrix of the state equation will be:

$$J_1 = \begin{bmatrix} \frac{\partial(x_2)}{\partial x_1} & \frac{\partial(x_2)}{\partial x_2} \\ \frac{\partial(\frac{-g}{l}\sin x_1 + \frac{1}{ml^2}T)}{\partial x_1} & \frac{\partial(\frac{-g}{l}\sin x_1 + \frac{1}{ml^2}T)}{\partial x_2} \end{bmatrix}$$

$$J_1 = \begin{bmatrix} 0 & 1 \\ \frac{-g}{l}\cos x_1 & 0 \end{bmatrix}$$

Since our system has input, we also need to calculate jacobian $J_2$ for the B matrix.

$J_2$ will be:

Skip to main content

$$J_2 = \begin{bmatrix} \frac{\partial f_1}{\partial u} \\ \frac{\partial f_2}{\partial u} \end{bmatrix}, \quad f_1 = \dot{x}_1, f_2 = \dot{x}_2$$

Since T is input, we replace T with u

$$J_2 = \begin{bmatrix} \frac{\partial(x_2)}{\partial u} \\ \frac{\partial(\frac{-g}{l}\sin x_1 + \frac{1}{ml^2}u)}{\partial u} \end{bmatrix}$$

$$J_2 = \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix}$$

**3. For each equilibrium point, substitute value of ($x_1$,$x_2$) in the jacobian and calculate the A and B matrix.**

The values of A matrix for each equilibrium point will be given as:

$$A_{(0,0)} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix}, \qquad A_{(\pi,0)} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix}$$

The values of B matrix for all equilibrium points will be:

$$B = \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix}$$

**4. Construct the state equation for each equilibrium point.**

The state equation for equilibrium point (0,0) will be:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} u$$

The state equation for equilibrium point (π,0) will be:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} u$$

Skip to main content

**5. Check the stability of the system at each equilibrium point.**

At equilibrium point (0,0) the eigenvalues will be:

$$\lambda = \pm\sqrt{\frac{g}{l}}i$$

At equilibrium point (π,0) the eigenvalues will be:

$$\lambda = \pm\sqrt{\frac{g}{l}}$$

The eigenvalues for (0,0) will be purely imaginary. Hence the system will be marginally stable. Marginally stable means that system will be continue to oscillate about the equilibrium point indefinitely.

The eigenvalues for (π,0) will be purely real. One of the eigenvalues will have positive real part. Hence the system will be unstable.

Hence we proved that our earlier intuitions about the stability of the system are correct. The system will be stable for (0,0) and unstable for (π,0).

## Controllability and Observability

In control theory, controllability and observability are two very important properties of the system.

**Controllability** is the ability to drive a state from any initial value to a final value in finite amount of time by providing a suitable input. A matrix which determines if a system is fully controllable or not is called the controllability matrix.

**Observability** is the property of the system that for any possible sequnce of state and control inputs, the current state can be determined in finite time using only the outputs. A matrix which determines if a system is fully observable or not is called the observability matrix. A fully observable system means that it is possible to know all the state variables from the system outputs.

Controllability matrix (R) of a system is given by the following:

$$R = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix}$$

Observability matrix (O) of a system is given by the following:

Skip to main content

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

If a system is fully controllable, then

$$rank(R) = n$$

If a system is fully observable, then

$$rank(O) = n$$

Here, n is the number of state variables.

Rank of a matrix is defines as the maximum number of linearly independent rows or columns in a matrix.

In the pendulum example (with external torque), we have the A and B matrix available to us. Hence we can calculate the controllability of the system.

$$R = \begin{bmatrix} 0 & \frac{1}{ml^2} \\ \frac{1}{ml^2} & 0 \end{bmatrix}$$

The rank of R is 2 which is equal to the number of state variables. Hence the system is fully controllable.

---

Take the following quiz before moving forward:

## Task 1.2 Part 2: Quiz

**Mathematical Modeling of a system**

**Introduction to State Space Analysis**

**Stability**

**Controllability and Observability**

**Skip to main content** 168

**UNLISTED ON OCT 17**

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.2 Theory (Part 3)

Hyperactive e-Yantra Staff 7d

## Mathematical Modeling of a system

## Controller Design

So far we have discussed the basics of state space analysis. In this section, we will discuss different types of controller design.

Consider the State Space Equations of a system:

$$\dot{x}(t) = Ax(t) + Bu(t)$$
$$y(t) = Cx(t) + Du(t)$$

This system can be represented in form of a block diagram as follows:

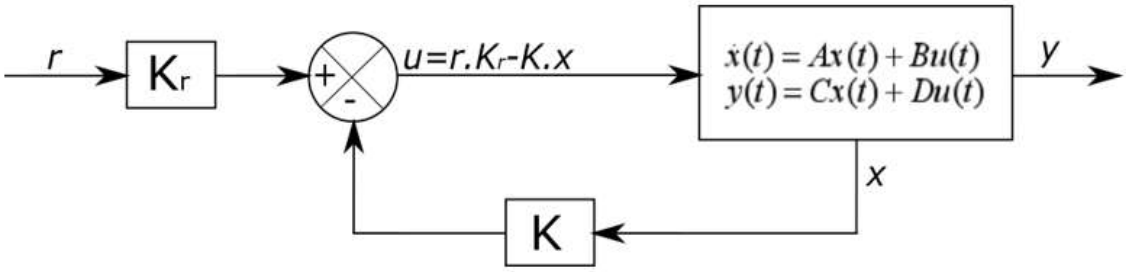



Figure 4: System block diagram

Here

r - Reference point

Kr - Gain by which reference is multiplied

x - State Vector

y -Output Vector

u - Input to system

K - Gain by which input is multiplied.

In this system we have taken the state vector x, multiplied that with some gain matrix K and fed that as feedback input to the system.

Skip to main content

The State Equation for the system can be written as follows:

$$\dot{x} = Ax + B(rK_r - Kx)$$
$$\Rightarrow \dot{x} = Ax - BKx + BrK_r$$
$$\Rightarrow \dot{x} = \underbrace{(A - BK)}_{New\ State\ Matrix} x + BrK_r$$

The new state matrix (A-BK) defines the dynamics of the system where -Kx is fed as input. The system stability can be calculated by finding the eigenvalues of the (A-BK) matrix.

## Pole Placement

One method to ensure that the system is stable is to select the gain matrix K in such a way so that the eigenvalues of the (A-BK) matrix are purely real and negative.

We can select the desired eigenvalues for the system and calculate the K matrix such that (A-BK) has our desired eigenvalues.

We again go back to the pendulum(with external torque) system with the following state equation:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} u$$

The equilibrium point for this state equation is (π,0). We have already proved that this system is unstable as one of the eigenvalues of A matrix is real and positive.

Let the gain matrix K = [$k_1$ $k_2$]

If we assume u=-Kx,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} \begin{bmatrix} k_1 & k_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

(A-BK) matrix will be:

$$(A - BK) = \begin{bmatrix} 0 & 1 \\ \frac{g}{l} & 0 \end{bmatrix} - \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} \begin{bmatrix} k_1 & k_2 \end{bmatrix}$$
$$(A - BK) = \begin{bmatrix} 0 & 1 \\ \left(\frac{g}{l} - \frac{k_1}{ml^2}\right) & \frac{-k_2}{ml^2} \end{bmatrix}$$

The characteristic equation for this system will be:

Skip to main content

$$\begin{vmatrix} A-BK-\lambda I \end{vmatrix} = 0$$

$$\begin{vmatrix} -\lambda & 1 \\ \left(\frac{g}{l}-\frac{k_1}{ml^2}\right) & \frac{-k_2}{ml^2}-\lambda \end{vmatrix} = 0$$

$$\lambda^2+\frac{k_2}{ml^2}\lambda+\left(\frac{k_1}{ml^2}-\frac{g}{l}\right)=0 \qquad (13)$$

Let us select some arbitrary eigenvalues.

$$\lambda_1=-1 \qquad \lambda_2=-2$$

These eigenvalues are purely real and negative. Hence if (A-BK) has these eigenvalues, the system will be stable.

If the eigenvalues are as given above, the characteristic equation for the system will be

$$(\lambda+2)(\lambda+1)=0$$
$$\lambda^2+3\lambda+2=0 \qquad (14)$$

Since (13) and (14) represent the same system, we can calculate the values of $k_1$ and $k_2$.

$$\frac{k_2}{ml^2}=3 \;; \quad \frac{k_1}{ml^2}-\frac{g}{l}=2$$
$$k_2=3ml^2; \quad k_1=2ml^2+mgl$$

Hence,

$$K=\begin{bmatrix} 2ml^2+mgl & 3ml^2 \end{bmatrix}$$

Hence we found the **K** matrix for which the system is **stable.**

## Linear Quadratic Regulator (LQR)

So far we have seen that if we have a system which is controllable, then we can place its eigenvalues anywhere in the left half plane by choosing appropriate gain matrix K. But the main question is where should we place our eigenvalues?

Till now we have only discussed about the stability of the system. But nowhere have we asked that what is our performance measure?
In this section, we'll see how to optimize the value of gain matrix K to get the desired performance
measure from the system.

Linear Quadratic Regulator is a powerful tool which helps us choose the K matrix according to our desired response. Here we use a cost function.

Skip to main content

$$J = \int_0^\infty (x^T Q x + u^T R u)$$

where, Q and R are positive semi-definite diagonal matrices (positive semi-definite matrices are those matrices whose all the eigenvalues are greater than or equal to zero). Also to remind you that for a diagonal matrix, the diagonal entries are its eigenvalues. x and u are the state vector and input vector respectively.

Let us say that you have your system with four states and one input.

Then $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$ and input *u*. Let

$$Q = \begin{bmatrix} Q_1 & 0 & 0 & 0 \\ 0 & Q_2 & 0 & 0 \\ 0 & 0 & Q_3 & 0 \\ 0 & 0 & 0 & Q_4 \end{bmatrix}$$

Then $x^T Q x + u^T R u = Q_1 x_1^2 + Q_2 x_2^2 + Q_3 x_3^2 + Q_4 x_4^2 + R u^2$ . Thus, you may see that the system taken here is our usual system represented as

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

The controller is of the form *u = -Kx* which is a **Linear** controller and the underlying cost function is **Quadratic** in nature and hence the name **Linear Quadratic Regulator.**
With a careful look at the integrand of the cost function J, we may observe that each Qi are the weights for the respective states xi.

So, the trick is to choose weights Qi for each state xi so that the desired performance criteria is achieved. Greater the state objective is, greater will be the value of Q corresponding to the said state variable. We can choose R = 1 for single input system. In case we have multiple inputs, we could use similar arguments for weighing the inputs as well.

In case of inverted pendulums, we know that angle with the vertical and the angular velocity _ is very important and hence the weights corresponding to them should be more as compared to linear position x and linear velocity x_ .

Skip to main content

LQR minimizes this cost function J based on the chosen matrices Q and R. Its a bit complicated to find out matrix K which minimizes this cost function. This is usually done by solving Algebraic Riccatti Equation (ARE). We'll not go into details of how to solve ARE, as it is not required in our tasks. There is inbuilt **lqr** command in octave to find K matrix. What is required to be done is to choose the Q and R matrix appropriately to get the desired performance.

---

**Now that you have understood the theory, let's move on to implementation of Task 1.2 Click on the link given below.**

# Task 1.2 Practical

Mathematical Modeling of a system

Controller Design

Pole Placement

Linear Quadratic Regulator (LQR)

Task 1.2 Practical 150

---

UNLISTED ON OCT 17

---

CLOSED 2 DAYS AGO

# Dairy Bike: Task 1.2 Practical

task-1

saurabhp e-Yantra Staff 11d

Table of contents

## Task Instructions

In this task, we will be implementing Pole Placement Controller and the LQR controller on different types of physical systems. You will be required to find out the equations of motion for each of the physical systems and modify code as instructed in order to simulate each of the physical systems in a proper manner. You are also required to answer a set of questions relating to each physical system so as to ensure us that you have thoroughly understood the task given.

For every Physical System that we will discuss here, you will find following files in **Task_1/Task_1_2** folder:

1. Simple_Pendulum.m
2. Simple_Pulley.m
3. Complex_Pulley.m
4. Mass_Spring_System.m
5. RW_Pendulum.m

## Physical System 1 - Simple Pendulum

We will revisit the Simple Pendulum (with external torque) once again.

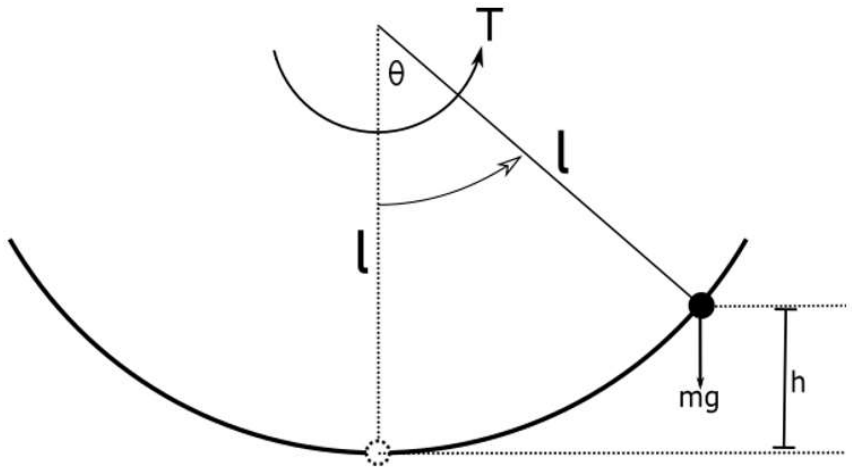



Figure 1: Simple Pendulum (with external torque)

We also derived the equations of motion of this system as:

Skip to main content

**ndulum.m** using Octave.

In this file you will find the following functions defined:

- draw_pendulum()
- pendulum_dynamics()
- sim_pendulum()
- pendulum_AB_matrix()
- pole_place_pendulum()
- lqr_pendulum()
- simple_pendulum_main()

---

**draw_pendulum():**

This function is used to draw the pendulum on a 2D plot. **You are not allowed to make any changes to this function.**

---

**pendulum_dynamics():**

This function is used to define the dynamics of the system by using the equations of motion you have derived for this system.

| | **Lets now understand this function:** |
|---|---|
| 1. | **function** dy = pendulum_dynamics ( y, m, L, g, u ) |
| 2. | sin_theta = sin( y(1) ) ; |
| 3. | cos_theta = cos( y(1) ) ; |
| 4. | dy (1,1) = y (2) ; |
| 5. | dy (2,1) = ; |
| 6. | **endfunction** |

In this function,

- y - denotes the state vector. The state vector consists of 2 state variables y(1) and y(2).
- y(1) corresponds to x1 (or theta) in the system of equations.
- y(2) corresponds to x2 (or theta_dot) in the system of equations.
- dy (1,1) corresponds to ẋ1 (or theta_dot) in the system of equations.
- dy (2,1) corresponds to ẋ2 (or theta_dot_dot) in the system of equations.
- m is the mass of pendulum bob.
- g is the acceleration due to gravity.
- L is the length of pendulum bob.
- u is the input to the system.
- Line 4 denotes the first equation of motion for the system of equations
- You are required to complete Line 5 by filling in the second equation of motion in the space provided.

---

**sim_pendulum():**

This function simulates the Simple Pendulum with no input.

Skip to main content

**Lets now understand this function:**

```
1. function [t,y] = sim_pendulum(m, g, L, y0)
2. tspan = 0:0.1:10;                                  ## Initialize time step
3. u = 0;                                             ## No Input
4. [t,y] = ode45(@(t,y)pendulum_dynamics(y, m, L, g, u),tspan,y0);
5. endfunction
```

- y0 is the initial condition of the system.
- Line 4 integrates the differential equation defined in the pendulum_dynamics() function across the length of tspan with initial condition y0. Line 4 returns an output [t, y].
- t is the time step array.
- y is the solution array where each element in y is the solution of the first order differential equation and corresponds to an element in t.
- [t,y] is returned as output of the sim_pendulum() function.

Note- **You are not allowed to make any changes to this function.**

---

**pendulum_AB_matrix():**

This function returns the A and B matrices for the system.

**Lets now understand this function:**

```
1. function [A,B] = pendulum_AB_matrix(m, g, L)
2. A = [ 0 1 ; g/L 0] ;
3. B = ;
4. endfunction
```

- Declare the A and B matrices for the Simple Pendulum system.

---

**pole_place_pendulum()**

This function simulates the Simple Pendulum with **u = -Kx** input. K matrix is calculated using pole placement.

**Lets now understand this function:**

```
1. function [t,y] = pole_place_pendulum (m, g, L, y_setpoint, y0 )
2. [A,B] = ;
3. eigs = ;
4. K = ;
5. tspan = 0 : 0.1 : 10;
6. [t,y] = ode45(@ (t,y) pendulum_dynamics (y, m, L, g, -K* (y-y_setpoint)), tspan, y0);
7. endfunction
```

Skip to main content

You are required to make the following changes in this function:

- Line 2 - Use pendulum_AB_matrix() to return A and B matrix.
- Line 3 - Initialize a 2x1 matrix with two eigenvalues. The eigenvalues should be selected so that the system is stable.
- Line 4 - Calculate the K matrix. Use the **place()** function in octave to calculate K.
- **y0** is the **initial state** or **initial condition** of the system. Which means the system will start from this state.
- **y_setpoint** is the final required state of the system. The pole placement controller will try to drive the system to this state. Here the y_setpoint is (π,0) which is an equilibrium point.
- Line 6 integrates the differential equation defined in the pendulum_dynamics() function across the length of tspan with initial condition y0 and u = -Kx

---

**lqr_pendulum()**

This function simulates the Simple Pendulum with **u = -Kx** input. K matrix is calculated using LQR.

| | **Lets now understand this function:** | |
|---|---|---|
| 1. | **function** [t,y] = lqr_pendulum(m, g, L, y_setpoint, y0) | |
| 2. | [A, B] = ; | ## Initialize A and B matrix |
| 3. | Q = ; | ## Initialize Q matrix |
| 4. | R = ; | ## Initialize R |
| 5. | K = ; | ## Calculate K matrix from A,B,Q,R matrices |
| 6. | tspan = 0:0.1:10; | ## Initialize time step |
| 7. | [t,y] = ode45(@(t,y)pendulum_dynamics(y, m, L, g, -K*(y-y_setpoint)),tspan,y0); | |
| 8. | **endfunction** | |

You are required to make the following changes in this function:

- Line 2 - Use pendulum_AB_matrix() to return A and B matrix.
- Line 3 - Choose an appropriate Q matrix.
- Line 4 - Choose an appropriate value of R.
- Line 5 - Calculate the K matrix using the *lqr()* function in octave.
- **y0** is the **initial state** or **initial condition** of the system. Which means the system will start from this state.
- **y_setpoint** is the final required state of the system. The LQR controller will try to drive the system to this state. Here the y_setpoint is (π, 0) which is an equilibrium point.
- The last two lines are same as pole_place_pendulum(). Line 8 integrates the differential equation defined in the pendulum_dynamics() function across the length of tspan with

Skip to main content ınd u = -Kx

**simple_pendulum_main()**

This function is used for testing our code by calling the various functions.

| | **Lets now understand this function:** | |
|---|---|---|
| 1. | **function** simple_pendulum_main() | |
| 2. | m = 1; | |
| 3. | g = 9.8; | |
| 4. | L = 1; | |
| 5. | y_setpoint = [ pi; 0 ] ; | |
| 6. | y0 = [pi/6; 0]; | |
| 7. | [t,y] = sim_pendulum(m,g,L, y0); | ## Test Simple Pendulum |
| 8. | ## [t,y] = pole_place_pendulum(m,g,L, y_setpoint, y0); | ** |
| 9. | ## [t,y] = lqr_pendulum(m,g,L, y_setpoint, y0); | *** |
| 10. | **for** k = 1:length(t) | |
| 11. | draw_pendulum (y( k, : ), L ); | |
| 12. | **endfor** | |
| 13. | **endfunction** | |

**## Test Simple Pendulum with Pole Placement Controller

***## Test Simple Pendulum with LQR Controller

- This function can be used to test out the 3 functions *sim_pendulum()*, *lqr_pendulum()* and pole_place_pendulum()* individually by uncommenting one function and commenting out the rest.
- You can modify the system parameters like mass, gravity, length of pendulum etc. Line 10 - runs a for-loop to animate the pendulum behavior on a 2D plot.
- The required behavior of the pendulum due to the three functions can be seen here at this **link.**

---

**Note:** In order to execute **simple_pendulum_main()** from the Octave command window, first run the **Simple_Pendulum** script in the command window. This will enable Octave to recognize the functions defined in Simple_Pendulum. These functions can then be directly called from the command window.

Let’s move on to the second physical system.

## Physical System 2 - Mass Spring System

---

We will examine a new system now. In the given figure we have a Mass Spring system.

Skip to main content

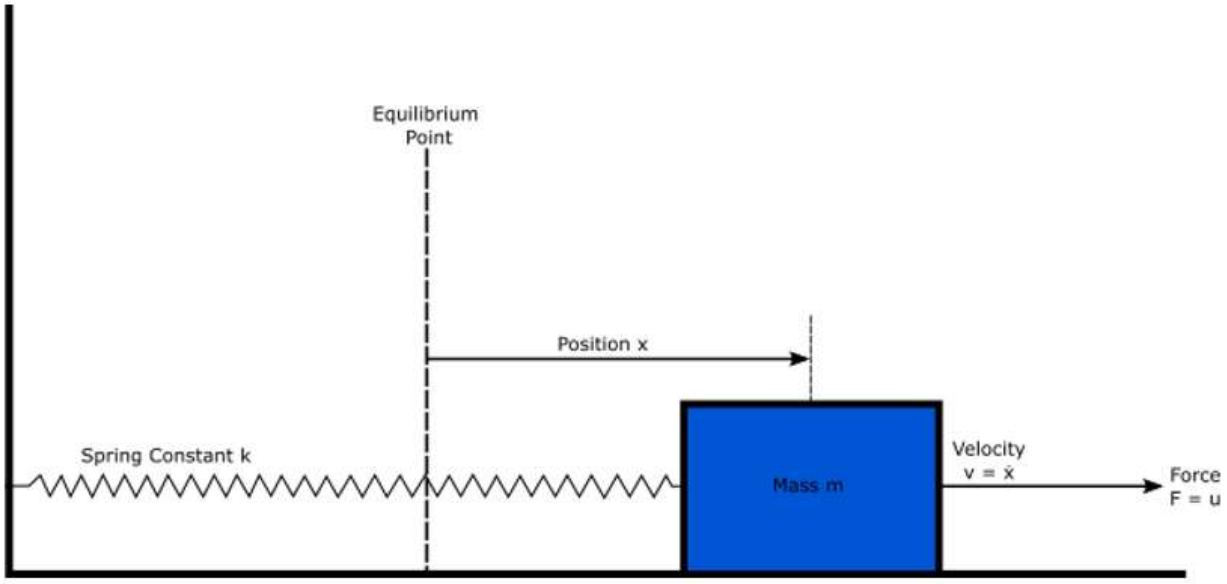



Figure 2: Mass Spring System

The mass-spring system has block of **mass m** which is connected to wall with a spring of **spring constant k**. The equilibrium point is the position where the spring is neither expanded nor contracted. The **position** of mass with respect to equilibrium point is denoted by **x**. A **force u** is acting on the the mass in horizontal direction. The velocity of the mass is denoted by **ẋ**.

You are required to do the following:

1. Derive the equations of motion for this system using the Euler-Lagrangian method we discussed earlier. The state variables for this system will be the position of the mass $\mathbf{x_1 = x}$ and velocity of mass $\mathbf{x_2 = \dot{x}}$.

2. Derive the A and B matrices for the system.

3. Open **Mass_Spring_System.m**. In this file you will find the following functions:

   - *draw_mass_spring()*
   - *mass_spring_dynamics()*
   - *sim_mass_spring()*
   - *mass_spring_AB_matrix()*
   - *pole_place_mass_spring()*
   - *lqr_mass_spring()*
   - *mass_spring_main()*

   a) **Do no edit draw_mass_spring() function.**

   b) *mass_spring_dynamics()* determines the dynamics of the system. You will need to fill in the equations of motion you have derived in this function (similar to as you did in *pendulum dynamics()* function).

Skip to main content

c) *sim_mass_spring()* is similar to *sim_pendulum()*. It will simulate the behaviour of the system under condition of no input.

d) *mass_spring_AB_matrix()* returns the A and B matrix of the system. You need to fill in the A and B matrices you have derived.

e) *pole_place_mass_spring()* and *lqr_mass_spring()* are similar to *pole_place_pendulum()* and *lqr_pendulum()*. You need to complete the code in a similar way.

f) *mass_spring_main()* can be called from command window to test the behaviour of the system. All the constants are defined in this function. **y0** is the **initial state** or **initial condition** of the system. Which means the system will start from this state. Here y0 is [-0.3;0] which means the initial position of block is -0.3 and initial velocity is 0.

g) **y_setpoint** is the final required state of the system. The pole placement or LQR controller will try to drive the system to this state. Here final required state is [0.7; 0]. That means the final position of block should be 0.7 and final velocity should be 0.

You can view the expected behaviour of the system due to the three functions **here.**

## Physical System 3 - Simple Pulley System

The third system is given as follows:

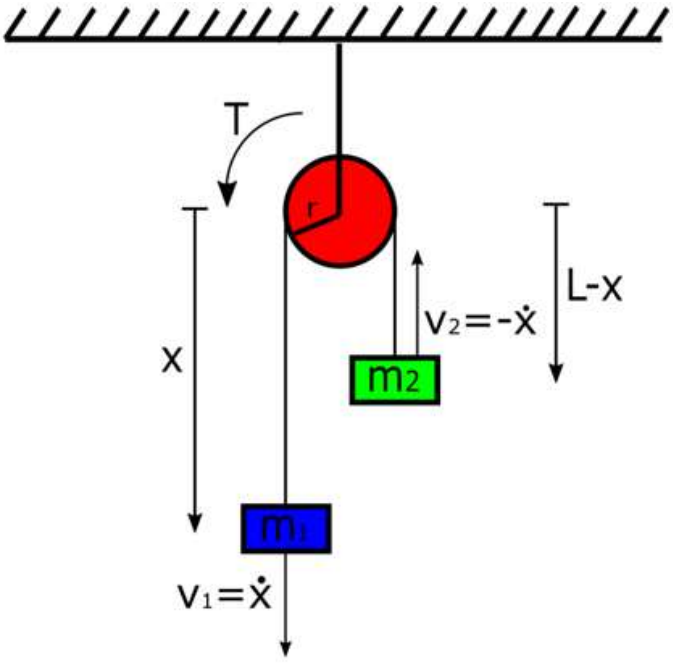



Figure 3: Simple Pulley System

A Simple Pulley system is shown. A pulley of **radius r** hangs from the ceiling.

The pulley is considered mass-less. There are two blocks of **mass $m_1$** and **$m_2$** which hang on either side of the pulley connected by a in-extensible string of **length L**. The **position** of mass $m_1$ with respect to pulley is **x** and **position** of $m_2$ with respect to pulley is **L-x**. The **velocity** of $m_1$ (downward direction is taken as positive). The **velocity** of $m_2$ is considered to be **$-\dot{x}$**. A torque T is applied as input to the system.

You are required to do the following:

1. Derive the equations of motion for this system. Make sure your system of equations are dimensionally consistent. The state variables for this system will be the position of the mass (with respect to pulley) $\mathbf{x_1 = x}$ and velocity of mass $\mathbf{x_2 = \dot{x}}$.

2. Derive the A and B matrices of the system.

3. Open the file **Simple_Pulley.m.** In this file you will find the following functions:

   - *draw_pulley()*
   - *pulley_dynamics()*
   - *sim_pulley()*
   - *pulley_AB_matrix()*
   - *pole_place_pulley()*
   - *lqr_pulley()*
   - *simple_pulley_main()*

   **a) Do not edit the draw_pulley() function.**

   b) The rest of the functions are similar to their corresponding functions in the last two cases. That is, *pulley_dynamics()* functions similar to *mass_spring_dynamics()* and *pendulum_dynamics()* and so on. Hence these functions need to completed according to the earlier instructions.

   c) *Simple_pulley_main()* can be called from command window to test the behaviour of the system. **y_setpoint** and **y0** are defined here.

You can view the expected behaviour of the system due to the three functions **here.**

## Physical System 4 - Complex Pulley System

The fourth system is given as follows:

Skip to main content

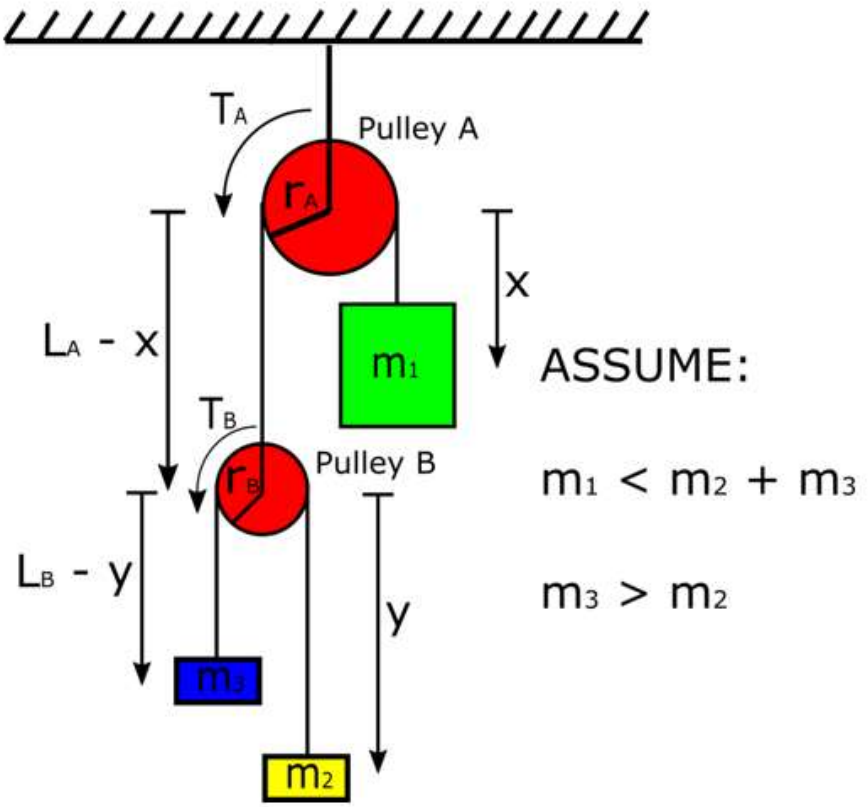



Figure 4: Complex Pulley System

A complex pulley system is shown.

Pulley A with **radius** x_1 hangs from a ceiling. A block of **mass** m_1 hangs from one side of the pulley A. On the other side of the pulley A, another Pulley B is hanging. Pulley B has radius r_b. Pulley B and mass m_2 are connected by an in-extensible string of length L_A. Mass m_2 and m_3 are hanging on opposite sides of pulley B and connected by another in-extensible string of length L_B.

The position of m_1 with respect to pulley A is **x**. The position of m2 with respect to pulley B is **y**.

We have made the following assumptions related to the masses m_1, m_2, m_3.

There are two torques T_A and T_B applied to pulley A and pulley B respectively as input.

You are required to do the following:

1. Derive the equations of motion for this system. Make sure your system of equations are dimensionally consistent. The state variables for this system will be:
    - Position of mass with respect to pulley A,
    - Velocity of mass with respect to pulley A,
    - Position of mass with respect to pulley B,
    - Velocity of mass with respect to pulley B,
2. Derive the A and B matrices of the system. Keep in mind that the number of state variables and inputs in this system is 4 and 2 respectively (as opposed to 2 state variables and 1 input in earlier systems). Hence the dimensions of A and B system will [illegible] case.

Skip to main content

3. Open the file **Complex_Pulley.m**.In this file, you will find the following functions:

- *draw_complex_pulley()*
- *complex_pulley_dynamics()*
- *sim_complex_pulley()*
- *complex_pulley_AB_matrix()*
- *pole_place_complex_pulley()*
- *lqr_complex_pulley()*
- *complex_pulley_main()*

**a) Do not edit the *draw_complex pulley()* function.**

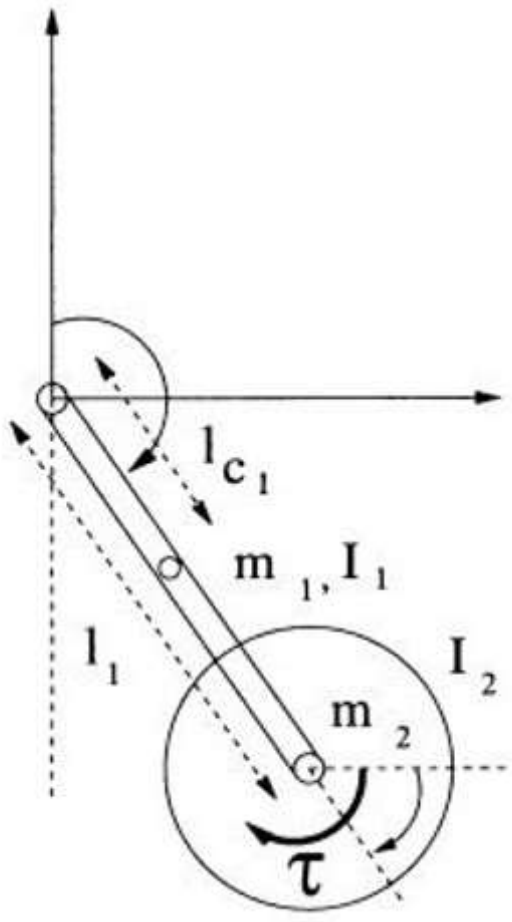
b) The rest of the functions need to be completed in similar way as earlier systems.

c) *Complex_pulley_main()* can be called from command window to test the behaviour of the system. **y_setpoint** and **y0** are defined here.

You can view the expected behaviour of the system due to the three functions **here.**

## Physical System 5- Reaction Wheel Pendulum System

The fifth system is given as follows:




Skip to main content Figure 5: Reaction Wheel Inverted Pendulum

This is an **Inverted Reaction Wheel Pendulum** system.

A reaction wheel with **mass m2** is mounted on the top of a pendulum bar of **mass m 1** suspended vertically upwards and connected to the ground with a hinge through a rigid rod of **length l1**. The **angle** which pendulum bar makes with respect to vertical is θ. and the angle by which reaction wheel rotate with respect to pendulum bar is denoted by **α**. The torque is provided to the reaction wheel via a motor attached to it. The pendulum bar is free to rotate at hinge on the ground.

You are required to do the following:

1. Derive the equations of motion for this system. Make sure your system of equations are dimensionally consistent. The state variables for this system will be:
    - Angular position of reaction wheel,
    - Angular velocity of reaction wheel,
    - Angular position of pendulum bar,
    - Angular velocity of pendulum bar,
2. Derive the A and B matrices of the system. Keep in mind that the number of state variables and inputs in this system is 4 and 1 respectively (as opposed to 2 state variables and 1 input in earlier systems). Hence the dimensions of A and B system will be different in this case.
3. Open the RW_Pendulum.m. In this function, you will find the following functions:
    - *draw_RW_pendulum()*
    - *RW__pendulum_dynamics()*
    - *sim_RW_pendulum()*
    - *RW_pendulum_AB_matrix()*
    - *pole_place_RW_pendulum()*
    - *lqr_RW_pendulum()*
    - *RW_pendulum_main()*

    a) Do not edit the *draw_RW_pendulum()* function.

    b) The rest of the functions need to be completed in similar way as earlier systems.

    c) *RW*_pendulum_main()* can be called from command window to test the behaviour of the system. **y_setpoint** and **y0** are defined here.

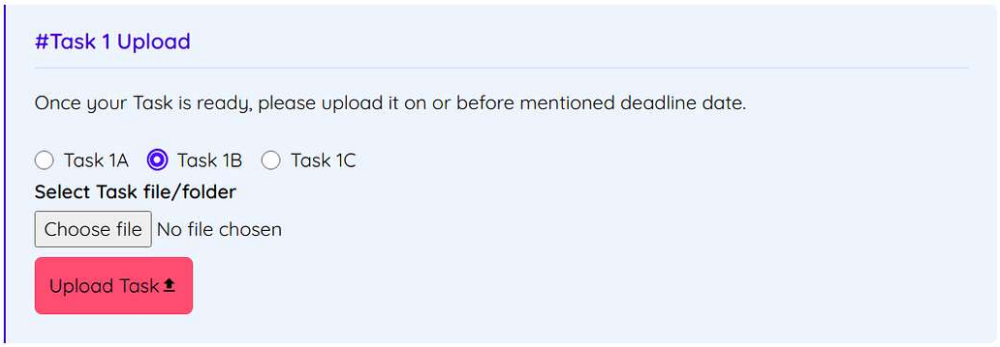
You can view the expected behaviour of the system due to the three functions **here.**

- Once you get your output similar to expected behaviour of the system as shown in the link above.
- Now create a new folder named **Task1_2_Submission**.
- Paste all the solution files (**Simple_Pendulum.m, Simple_Pulley.m, ı, Mass_Spring_System.m, RW_Pendulum.m**) inside the above

Skip to main content

mentioned folder.

- Now compress the folder **Task1_2_Submission** into .zip file.
- For successful completion of **Task 1_2**, upload the **Task1_2_Submission.zip** file on the portal.
- Now, open the portal and go to **Task 1**. In the **Task 1 Upload** section select **Task 1B.**
- Select **Choose file** button to upload the file. From the dialogue box, select the file and click **Open.**
- You shall see the file name **Task1_2_Submission.zip** in text-box besides the **Choose file** button. Click on **Upload Task** button to submit the file.




Task 1.2 is complete!!

**Congrats!!**

**Next >>**

---

## Dairy Bike: Task 1.2 Theory (Part 3)

### Task Instructions

- Physical System 1 - Simple Pendulum
- Physical System 2 - Mass Spring System
- Physical System 3 - Simple Pulley System
- Physical System 4 - Complex Pulley System
- Physical System 5- Reaction Wheel Pendulum System

**UNLISTED ON OCT 13**

**CLOSED 2 DAYS AGO**

# Dairy Bike: Task 1.3 Theory

task-1

Hyperactive e-Yantra Staff 11d

Table of contents

## Getting started with CoppeliaSim - Part 1

- This video introduces the basics of CoppeliaSim software.
- Teams will learn about Features, User Interface, Pages, Views, Scenes, Camera Navigation, Position/Orientation Manipulation, Simulation Settings, Scripts, User Settings, Collections, Layers, Video Recorder, Scene Object Properties etc.

**Now that you have understood the theory, let's move on to implementation of Task 1.3 Click on the link given below.**

## Task 1.3 Practical

Task 1.3 Practical 156

UNLISTED ON OCT 13

last visit

CLOSED 2 DAYS AGO

# Dairy Bike: Task 1.3 Practical

task-1

Hyperactive e-Yantra Staff 11d

Table of contents

## Task Instructions

### 1. Problem Statement

Write a code to implement LQR control strategy **to balance an inverted pendulum at its unstable equilibrium point using Reaction Wheel** in the given CoppeliaSim scene.

Write a code to erect the pendulum from the resting position towards the unstable equilibrium point and then use the LQR control strategy to balance it at the unstable equilibrium as shown in the gif below.

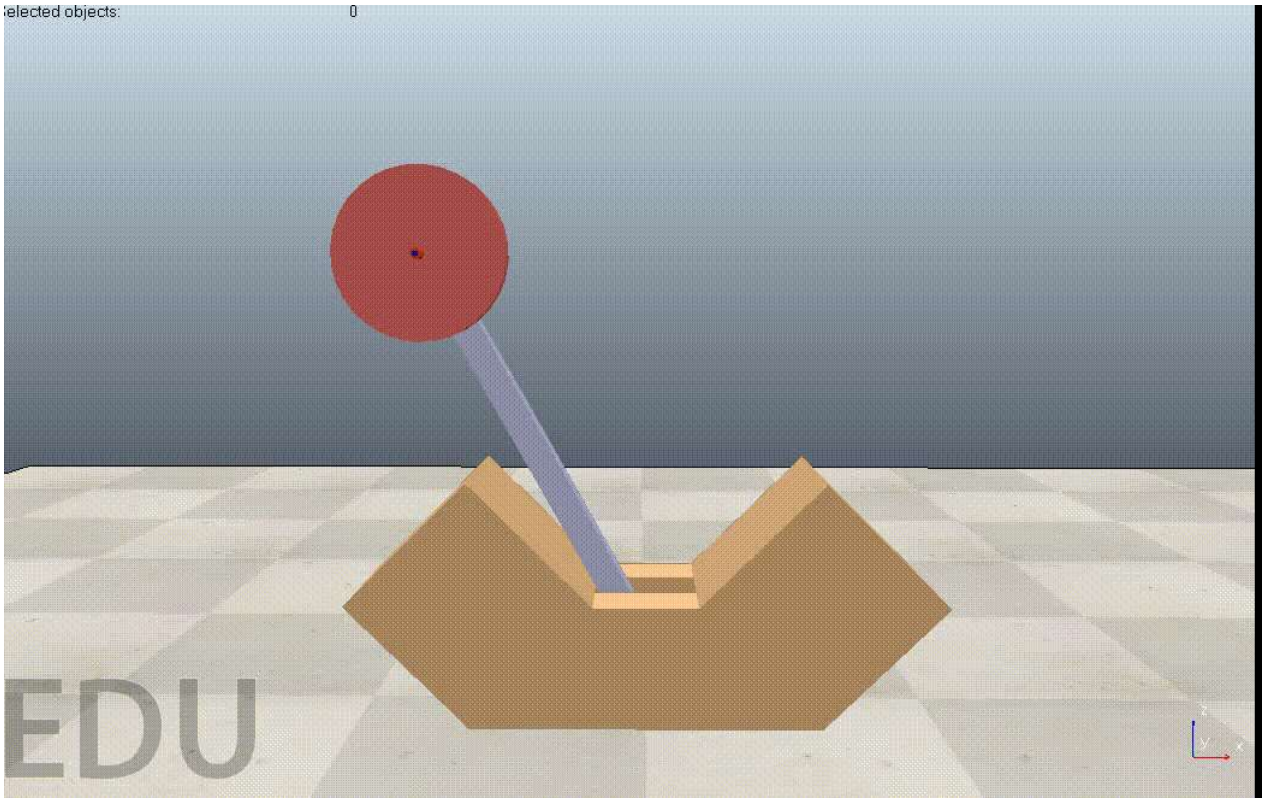



Figure 1: RW Pendulum balancing at unstable equilibrium point

### 2. Given

#### 1. CoppeliaSim's Scene File (RW_pendulum.ttt)

Skip to main content '_pendulum.ttt of CoppeliaSim software as shown in Figure 2.

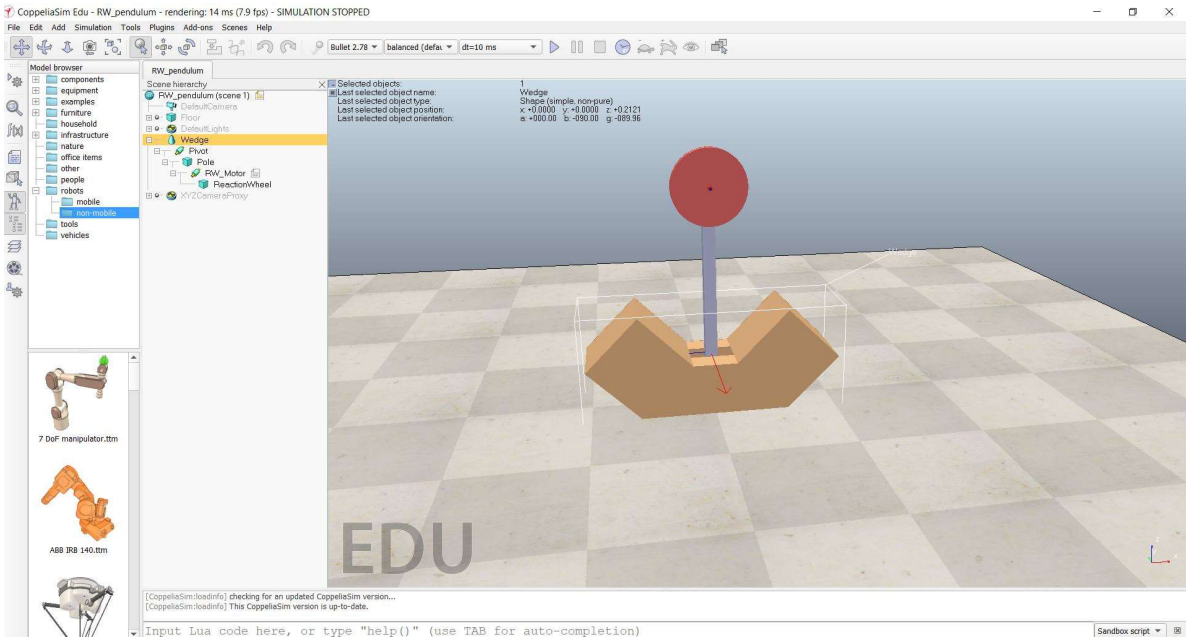



Figure 2: Tree structure of Reaction Wheel Pendulum.

- The **objects in the scene** along **with their names and uses** are given in Table 1.

| Objects | Name in the scene | Use(s) |
|---|---|---|
| Base | *Wedge* | Acts like an immovable base for Reaction Wheel Pendulum |
| Pivot | *Pivot* | As a pivot for the pendulum pole around which the pendulum may oscillate |
| Pendulum | *Pole* | Acts as the pendulum |
| Reaction Wheel Motor | *RW_Motor* | The pivot with motor enabled. It provides the input torque to the reaction wheel. |
| Reaction Wheel | *ReactionWheel* | Reaction wheel rotates appropriately to change the orientation of the pendulum and helps balancing the reaction wheel pendulum |

> **NOTE:**
>
> - In this task you are **NOT** allowed to **remove** the above mentioned objects.
> - Any change in the **names of the objects, their parent child relationships, their properties, position, orientation** etc. will result in **poor evaluation** and hence low marks.

- Open the given scene file RW_pendulum.ttt with coppeliaSim.
- You'll find a hierarchical tree structure as shown in Figure 1. The list of objects are given in Table 1.
- **Wedge** is a dynamic respondable object with mass defined to be 80 Kg making the

Skip to main content iovable.

- **Pivot** is a revolute joint placed at center of the wedge connected to the pendulum (denoted here as pole) with motor disabled. This makes the joint free to move. The pivot is shown in Figure 3.

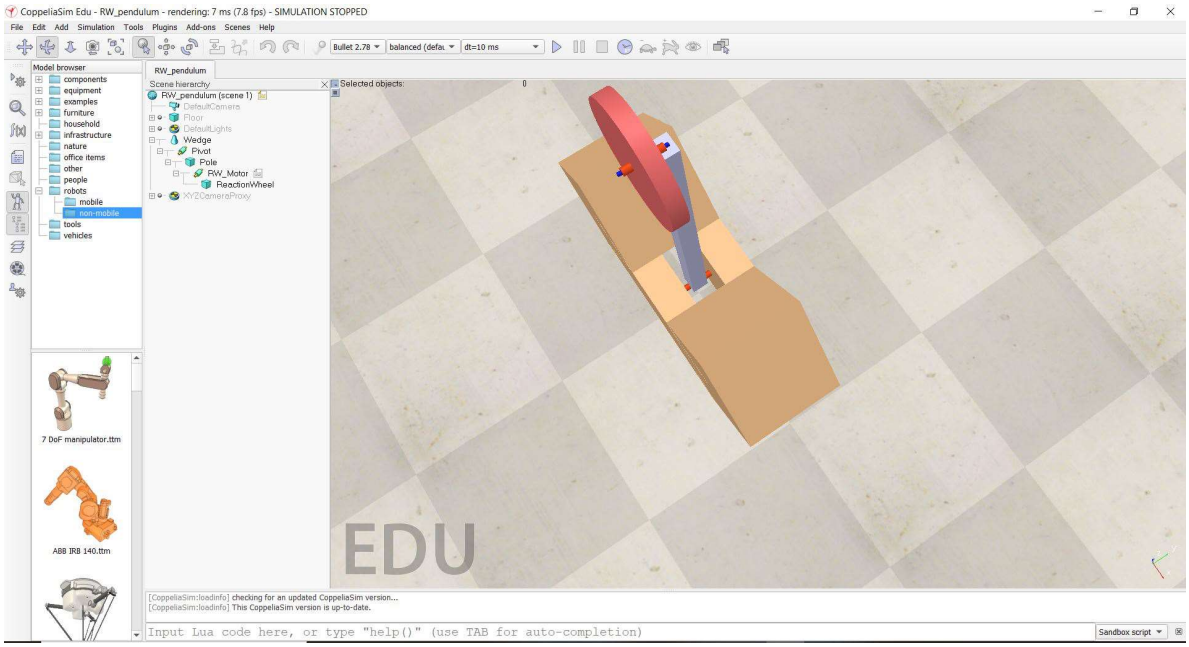
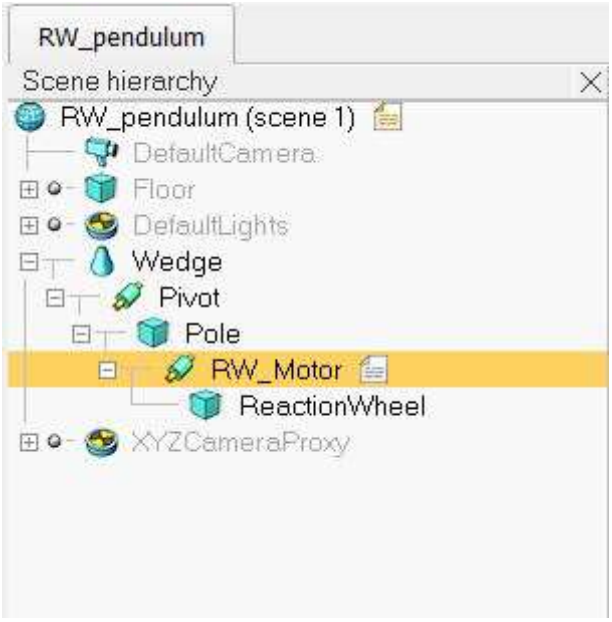



Figure 3: Revolute Joints in the reaction wheel pendulum.

- **Pole** is also a dynamic respondable object acting as the pendulum. The pendulum is connected from wedge to reaction wheel via two revolute joints. The mass of the pendulum is 2 Kg.
- **RW_Motor** is another revolute joint placed on top of the pole joining the pendulum with the Reaction wheel as shown in Figure 3. This joint has motor enabled with max. torque 2.5 N-m.
- **ReactionWheel** is the dynamic respondable object which connects with the pendulum via the revolute joint RW_motor. The motor provides the torque for it to rotate. The mass of Reaction wheel is 4.952 Kg

### Understanding the Task:

- You'll find a script icon near the RW_motor as shown in Figure 4.




Skip to main content

Figure 4: Script Symbol for the Motor

- Click on the script icon. You'll have the script opened as shown in Figure 5. You have to edit this file by writing your code in appropriate function to execute the control loop to balance the inverted pendulum via reaction wheel.

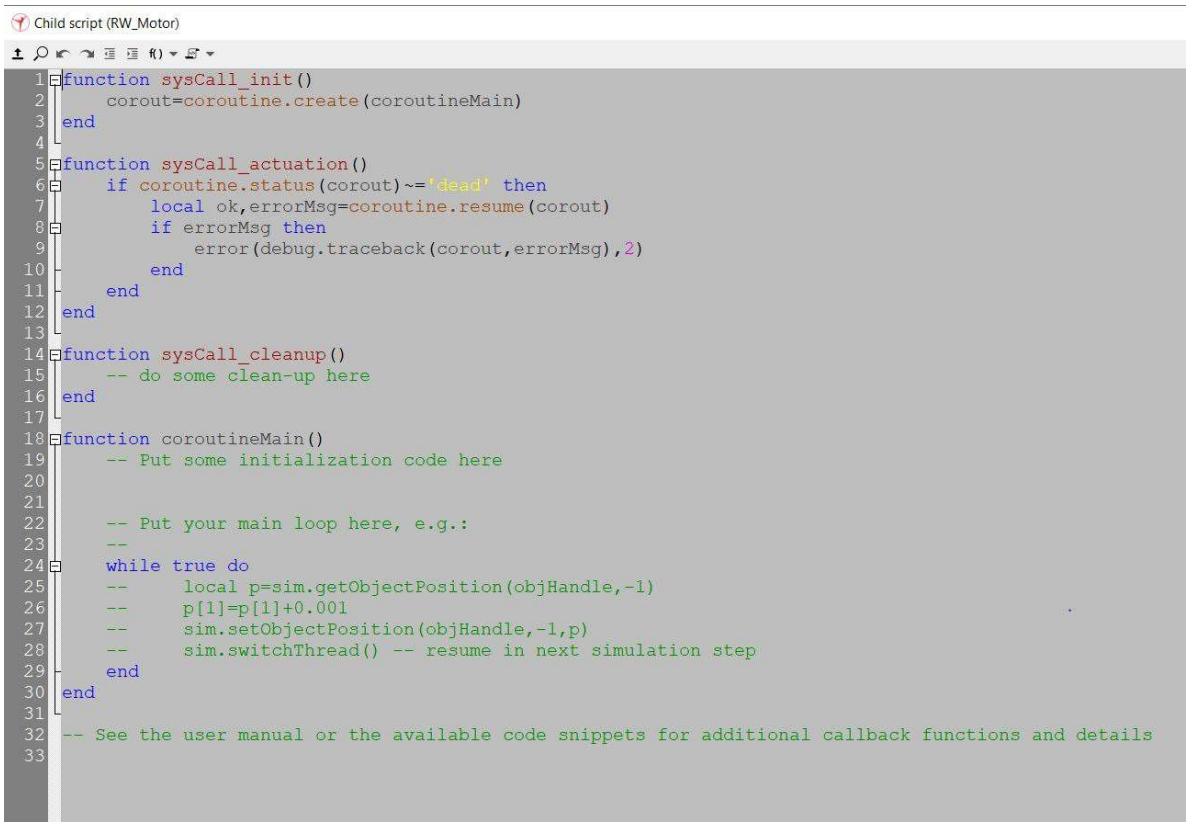



Figure 5: Script Functions

- In the function coroutineMain(), you have to write code for initialization and write appropriate code to implement LQR control strategy to balance the Reaction wheel pendulum in upright position.

```
function coroutineMain()
    -- Put some initialization code here


    -- Put your main loop here, e.g.:
    --
    while true do
    --      local p=sim.getObjectPosition(objHandle,-1)
    --      p[1]=p[1]+0.001
    --      sim.setObjectPosition(objHandle,-1,p)
    --      sim.switchThread() -- resume in next simulation step
    end
end
```

## Submission Instructions

( Skip to main content ully performed this experiment:

1. Open the RW_pendulum.ttt in CoppeliaSim.
2. Now open new Terminal (on Ubuntu OS or MacOS) or Anaconda Prompt (on Windows OS) and navigate to the **Task_1/Task_1_3** folder.
3. Activate your conda environment with the command

   ```
   conda activate DB_<team_id>
   ```

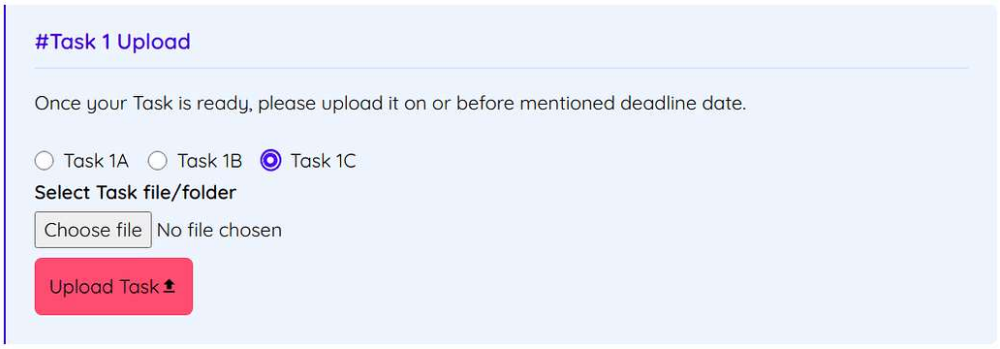
   Example: conda activate DB_9999
4. Run the **test_task1_3.pyc** by running following command

   ```
   python test_task1_3.pyc
   ```
5. When asked, you have to enter your Team ID, such as 9999.
6. It will trigger the simulation of RW_pendulum.ttt using the python remote api and calculate the settling time for the Reaction Wheel Pendulum to balance at the unstable equilibrium as shown below:

Figure 6: Final desired output in CoppeliaSim

7. It will generate **task1_3_output.txt**.
8. For successful completion of **Task 1_3**, upload the **task1_3_output.txt** file on the portal.
9. Now, open the portal and go to **Task 1**. In the **Task 1 Upload** section select **Task 1C**.
10. Select **Choose file** button to upload the file. From the dialog box, select the file and click **Open**.
11. You shall see the file name **task1_3_output.txt** in text-box besides the **Choose file** button. Click on **Upload Task** button to submit the file.

Skip to main content




## Congratulations !! Task 1 is complete!!

### Task Instructions

1. Problem Statement
2. Given

Submission Instructions

**UNLISTED ON OCT 13**

**CLOSED 2 DAYS AGO**