Task 0.2 - 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 0.2 folder, you will find the following scripts.

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

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

Main_File.m

Main_File.m

```
1 pkg load symbolic
2 syms x1 x2

3 x1_dot = -x1 + 2*x1^3 + x2;
4 x2_dot = -x1 - x2;

5   [equilibrium_points   jacobians   eigen_values   stability]   =
    main_function(x1_dot, x2_dot);
```

Line 1 - 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 2 - Declares 2 symbolic variables x1 and x2 (to denote x_1 and x_2)

Line 3-4 - 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 5 - 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 will be used to test your code for different sets of equations. 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.**

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 eqbm_points = find_equilibrium_points(x1_dot, x2_dot)
2   x1_dot == 0;
3   x2_dot == 0;
    ###################### ADD YOUR CODE HERE ######################
4   eqbm_points = ;
    ################################################################
5 endfunction
```

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 **eqbm_points**.
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 **eqbm_points.**

When you display **eqbm_points** (using disp() function in octave), the structure should be as shown:

```
Command Window
>> disp(eqbm_points)
{
  [1,1] =

    scalar structure containing the fields:

      x1 =

        <class sym>

      x2 =

        <class sym>


  [1,2] =

    scalar structure containing the fields:

      x1 =

        <class sym>

      x2 =
```

find_jacobian_matrices()

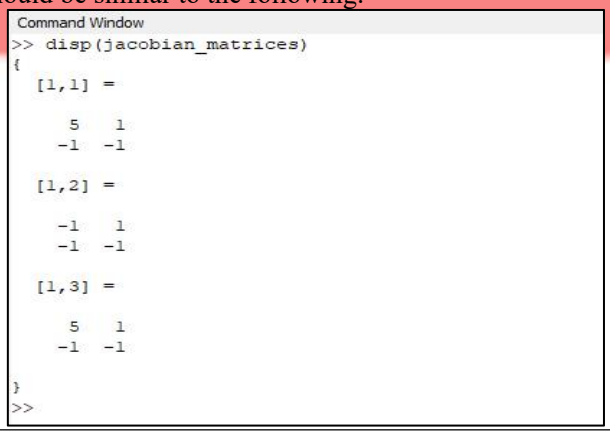
```
1 function jacobian_matrices = find_jacobian_matrices(eqbm_points,
    x1_dot, x2_dot)
2   syms x1 x2
3   solutions = {};
4   jacobian_matrices = {};
5   for k = 1:length(eqbm_points)
6     x_1 = eqbm_points{k}.x1;
7     x_2 = eqbm_points{k}.x2;
8     soln = [x_1 x_2];
9     solutions{k} = soln;
10   endfor

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


12 ##############################################################
13 endfunction
```

This function takes the **eqbm_points** 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 (stored in **eqbm_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.
Lines 3 and 4 - 2 empty cell arrays solutions and **jacobian_matrices** are initialized.
Lines 5 to 10 - A for loop iterates over the length of **eqbm_points**. For each equilibrium point, the values x1 and x2 are extracted (and stored in **x_1** and **x_2**). They are stored as a pair in the cell array solutions.

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**.
- When you display 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(eqbm_pts,
     jacobian_matrices)
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",double(eqbm_pts{k}.x1),double(eqbm_pts{k}.x2));
9        stability{k} = "Stable";
10     else
11       fprintf("The system is unstable for equilibrium point (%d, %d)
     \n",double(eqbm_pts{k}.x1),double(eqbm_pts{k}.x2));
12       stability{k} = "Unstable";
13     endif
14   endfor
15 endfunction
```

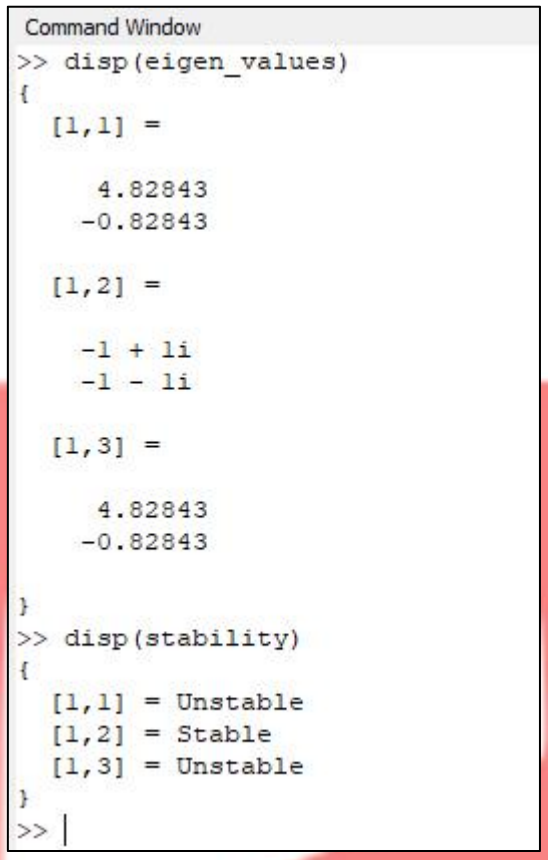
This function takes the **eqbm_points** and **jacobian_matrices** as input. For each jacobian matrix stored in **jacobian_matrices**, the eigenvalues of matrix are calculated and stored in the cell array **eigen_values**. Subsequently the eigenvalues are checked in this function. If for any jacobian 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.

2 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 [equilibrium_points jacobians eigen_values stability] =
    main_function(x1_dot, x2_dot)
2   pkg load symbolic
3   syms x1 x2
4   equilibrium_points = find_equilibrium_points(x1_dot, x2_dot);
5   jacobians = find_jacobian_matrices(equilibrium_points, x1_dot,
    x2_dot);
6   [eigen_values stability] = check_eigen_values
    (equilibrium_points, jacobians);
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 **equilibrium_points**, **jacobians**, **eigen_values**, **stability**. Each of these is a cell array.
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:

```
Command Window
>> Main_File
The system is unstable for equilibrium point (-1, 1)
The system is stable for equilibrium point (0, 0)
The system is unstable for equilibrium point (1, -1)
>>
```

Figure 1: Desired Output

Test_Suite.m (Testing your solution)

Test_Suite.m is used for testing your solution. You are not allowed to make any changes in this script.

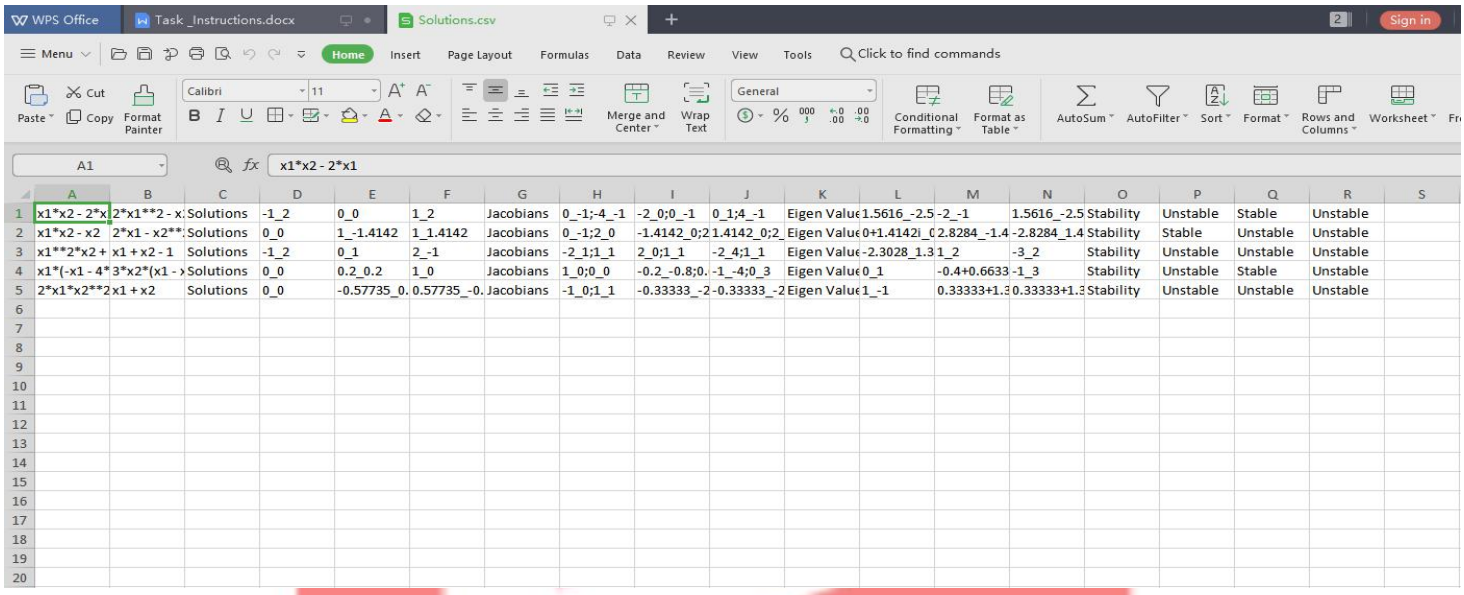
The Test_Suite script has 3 sets of non-linear equations If your code runs successfully for all three sets of equations then the output should be as follows:

```
Command Window
>> Test_Suite
1. Check the stability of system of equations:

   x1_dot = -2*x1 + x1*x2
   x2_dot = 2*x1*x1 - x2
The system is unstable for equilibrium point (-1, 2)
The system is stable for equilibrium point (0, 0)
The system is unstable for equilibrium point (1, 2)
2. Check the stability of system of equations:

   x1_dot = x1*x2 - x2
   x2_dot = 2*x1-x2^2
The system is stable for equilibrium point (0, 0)
The system is unstable for equilibrium point (1, -1.41421)
The system is unstable for equilibrium point (1, 1.41421)
3. Check the stability of system of equations:

   x1_dot = x1^2*x2 + 2*x1
   x2_dot = x2 +x1 - 1
The system is unstable for equilibrium point (-1, 2)
The system is unstable for equilibrium point (0, 1)
The system is unstable for equilibrium point (2, -1)
>>
```

Figure 2: Test Suite Output

If your Test_Suite runs successfully, a ***Solutions.csv*** file should be generated in your Task 0.2 folder. This file should contain all the equilibrium points, jacobians, eigen values and stability calculated for each set of non linear equations.

Figure 3 shows how the file should look like if the Test_Suite has run successfully.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	x1*x2 - 2*x1	2*x1**2 - x2	Solutions	-1_2	0_0	1_2	Jacobians	0_-1;-4_-1	-2_0;0_-1	0_1;4_-1	Eigen Value	1.5616_-2.5	-2_-1	1.5616_-2.5	Stability	Unstable	Stable	Unstable
2	x1*x2 - x2	2*x1 - x2**	Solutions	0_0	1_-1.4142	1_1.4142	Jacobians	0_-1;2_0	-1.4142_0;2	1.4142_0;2_	Eigen Value	0+1.4142i_0	2.8284_-1.4	-2.8284_1.4	Stability	Stable	Unstable	Unstable
3	x1**2*x2 +	x1 + x2 - 1	Solutions	-1_2	0_1	2_-1	Jacobians	-2_1;1_1	2_0;1_1	-2_4;1_1	Eigen Value	-2.3028_1.3	1_2	-3_2	Stability	Unstable	Unstable	Unstable
4	x1*(-x1 - 4*	3*x2*(x1 - x	Solutions	0_0	0.2_0.2	1_0	Jacobians	1_0;0_0	-0.2_-0.8;0.	-1_-4;0_3	Eigen Value	0_1	-0.4+0.6633	-1_3	Stability	Unstable	Stable	Unstable
5	2*x1*x2**2	x1 + x2	Solutions	0_0	-0.57735_0.	0.57735_-0.	Jacobians	-1_0;1_1	-0.33333_-2	-0.33333_-2	Eigen Value	1_-1	0.33333+1.3	0.33333+1.3	Stability	Unstable	Unstable	Unstable

Figure 3: Solutions.csv