Programming Assignment Signal Flow Graphs & Routh Stability Criterion

[GitHub Repository]

Team Members:

Mahmoud Hesham Mohamed	22011201
Abdelrahman Amgad Hassan	22010871
Ayman Ibrahim Mohamed Kotb	22010656
Mohamed Elsayed Mohamed	22011102
Ali mostafa mohamed	22010939

Problem Statement:

Develop a software application that aids in the analysis of linear control systems using Signal Flow Graphs and the Routh Stability Criterion. The program should provide a graphical interface that allows users to visually construct and analyze a signal flow graph, as well as determine the stability of a given system using its characteristic equation.

Technologies:

The application is web-based , developed using Python Flask for back-end and React for the front-end.

Modules:

The application is web-based and developed into the following modules exposed through different endpoints which will be discussed in further details later in the report:

- Graph Solving Module
- Routh Criteria Module

Part 1

Signal Flow Graph Analysis

Features:

Users can build the signal flow graph by dragging and dropping nodes to add them to the canvas, then connecting them with edges.

- Edge curvature can be adjusted by clicking on an edge and dragging the control point that appears above it.
- To delete nodes or edges, select them and press the Shift key.
- You can select multiple elements by holding the Shift key while clicking.
- Each edge allows you to assign a numerical gain value.

When the "Analyze" button is pressed, the system will:

- Automatically detect all loops, non-touching loop combinations, and forward paths.
- Display this information in organized tables, including the computed values of Δ, Δ_1, Δ_2, ..., and the overall transfer function using Mason's Gain Formula.

Modules and Data Structures

Classes Package:

The Classes package encapsulates the core components of the Signal Flow Graph Solver. It contains the essential data structures and logic necessary to represent and analyze signal flow graphs. The key modules within this package are:

1. Graph Class

Role: Serves as the primary data structure for representing the signal flow graph.

Structure: Utilizes a hashmap of hashmaps (Dict[str, Dict[str, float]]) to represent adjacency lists, ensuring O(1) time complexity for edge lookups.

Functionality:

- Graph construction from input data
- Identification of start nodes and end nodes
- Extraction of all paths between two nodes
- Utility functions for interacting with graph elements
- Detection of loops in the graph

2. Loop Class

Role: Represents a feedback loop within the signal flow graph.

Structure:

- A list of nodes involved in the loop
- The gain value of the loop (a float).

3. Path Class

Role: Represents a forward path from the source to the destination node.

Structure:

- A list of nodes traversed by the path
- The total gain of the path (a float)

Note: While similar in structure to Loop, it is semantically used to represent forward signal paths.

4. SignalGraphSolver Class

Role: Implements the algorithms required to solve the signal flow graph.

Functionality:

Leverages the Graph, Loop, and Path classes

Applies signal flow graph solving techniques, such as Mason's Gain Formula, to compute the overall transfer function of the system.

Messages Package:

Contains the message classes in which the front and back-end communicates.

Classes:

1. InputDTO , contains the graph representation sent by the front-end which will be processed to be used by back-end

Algorithms used:

- **Class `SignalGraphSolver`** solves signal flow graphs using Mason's Gain Formula.
- **Constructor** initializes loops, paths, and first-order (individual) loops.
- **`get_non_touching_loops` method** finds combinations of mutually non-touching loops and calculates their gain products.
- **`are_mutually_non_touching` method** checks if loops share any nodes.
- **`combo_gain` method** multiplies gains of loops in a combination.
- **`get_deltas` method**:
  - For each path, filters loops not touching the path.
  - Computes delta for each filtered set using a temporary solver.
- **`get_delta` method** calculates the overall graph delta using gains of non-touching loop sets.
- **`solve` method**:
  - Gets deltas for each path and the overall delta.
  - Computes final result using Mason's Gain Formula.

Pseudo code:

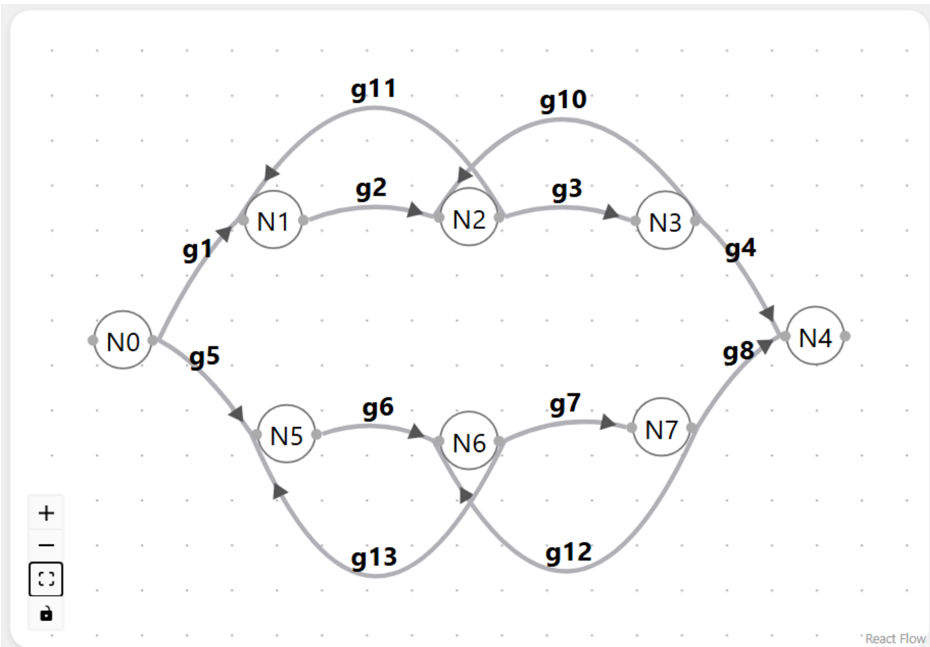
```
CLASS SignalGraphSolver
  CONSTRUCTOR(loops: List of Loop, paths: List of Path)
    INITIALIZE all_loops = loops
    INITIALIZE all_pairof_loops as empty list
    INITIALIZE paths = paths

    // First order loops (individual loops)
    ADD all individual loops to all_pairof_loops[0]

  METHOD get_non_touching_loops(all_loops)
    INITIALIZE gains as empty list
    ADD list of individual loop gains to gains[0]

    SET max_order = length of all_loops

    FOR order FROM 2 TO max_order:
      INITIALIZE current_combinations as empty list
      INITIALIZE current_gains as empty list

      IF order == 2:
        FOR EACH combination of loops (size order):
          IF combination is mutually non-touching:
            ADD combination to current_combinations
            ADD product of combination gains to current_gains
      ELSE:
        SET prev_order_index = order - 2
        FOR EACH prev_combo IN all_pairof_loops[prev_order_index]:
          FOR EACH loop IN all_loops:
            IF loop NOT IN prev_combo:
              CREATE new_combo = prev_combo + loop
              IF new_combo is mutually non-touching:
                SORT new_combo by loop id
                IF new_combo not already in current_combinations:
                  ADD new_combo to current_combinations
                  ADD product of new_combo gains to current_gains

      IF current_combinations is not empty:
        ADD current_combinations to all_pairof_loops
        ADD current_gains to gains
      ELSE:
        BREAK loop
```

```
        RETURN gains, all_pairof_loops

    METHOD are_mutually_non_touching(loops)
        FOR each pair of loops in loops:
            IF loops touch each other:
                RETURN False
        RETURN True

    METHOD combo_gain(loops)
        SET gain = 1.0
        FOR each loop in loops:
            MULTIPLY gain by loop.gain
        RETURN gain

    METHOD get_deltas()
        INITIALIZE deltas as empty list

        FOR each path in paths:
            INITIALIZE non_touching_loops as empty list

            FOR each loop in all_loops:
                SET touches_path = False
                FOR each node in loop.nodes:
                    IF node in path.path:
                        SET touches_path = True
                        BREAK
                IF NOT touches_path:
                    ADD loop to non_touching_loops

            CREATE temp_solver with non_touching_loops
            CALL temp_solver.get_delta()
            ADD resulting delta to deltas

        RETURN deltas

    METHOD get_delta()
        CALL get_non_touching_loops(all_loops) -> gains, loops
        SET res = 0
        SET sign = -1

        FOR each order in gains:
            MULTIPLY sign by -1
            FOR each gain in current order:
                ADD sign * gain to res

        RETURN (1 - res), gains, loops

    METHOD solve()
        CALL get_deltas() -> deltas
        CALL get_delta() -> delta, gains, loops

        SET res = 0
        FOR each path in paths:
            ADD path.gain * corresponding delta to res
        DIVIDE res by delta

        RETURN res, delta, deltas, gains, all_pairof_loops, paths
END CLASS
```

Sample Runs:

Example 1:

Constructed Graph:



Branches Gains

Function	Value
g1	2
g2	1
g3	1
g4	1
g5	3
g6	1
g7	1
g8	1
g10	-1
g11	-1
g12	-1
g13	-1

Example 1 Analysis:

Signal Flow Graph Results

×

Transfer Function

1.6666666666666667

Delta (Δ)

9

Forward Paths

Path	Gain
N0 → N1 → N2 → N3 → N4	2
N0 → N5 → N6 → N7 → N4	3

Individual Loops

Loop	Gain
N1 → N2	-1
N2 → N3	-1
N5 → N6	-1
N6 → N7	-1

2 Non-Touching Loops

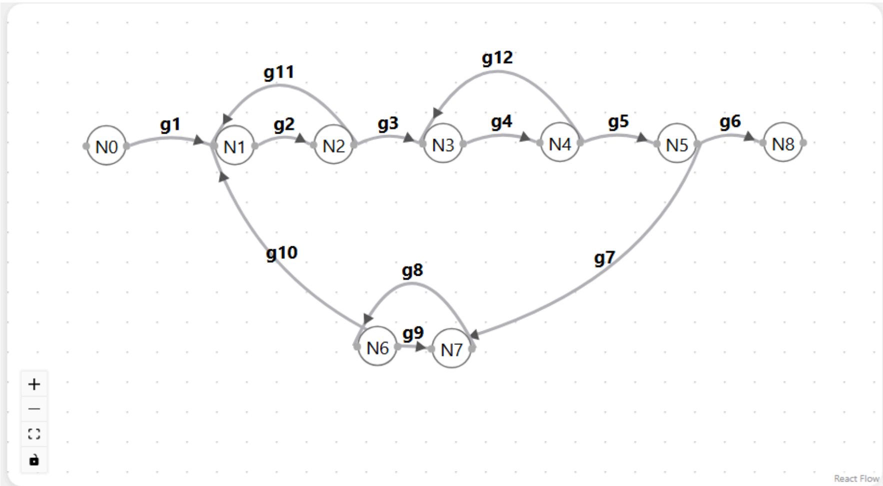
Loops	Gain
N1 → N2 & N5 → N6	1
N1 → N2 & N6 → N7	1
N2 → N3 & N5 → N6	1
N2 → N3 & N6 → N7	1

Delta Values for Each Forward Path

Path	Delta Value
Path 1	3
Path 2	3

Example 2:

Constructed Graph:



Branches Gains

Function	Value
g1	4
g2	1
g3	1
g4	1
g5	3
g6	1
g7	-1
g8	1
g9	2
g10	-2
g11	1
g12	1

Example 2 Analysis:

Signal Flow Graph Results

Transfer Function

2

Delta (Δ)

−6

Forward Paths

Path	Gain
N0 → N1 → N2 → N3 → N4 → N5 → N8	12

Individual Loops

Loop	Gain
N1 → N2 → N3 → N4 → N5 → N7 → N6	6
N1 → N2	1
N3 → N4	1
N7 → N6	2

2 Non-Touching Loops

Loops	Gain
N1 → N2 & N3 → N4	1
N1 → N2 & N7 → N6	2
N3 → N4 & N7 → N6	2

3 Non-Touching Loops

Loops	Gain
N3 → N4 & N1 → N2 & N7 → N6	2

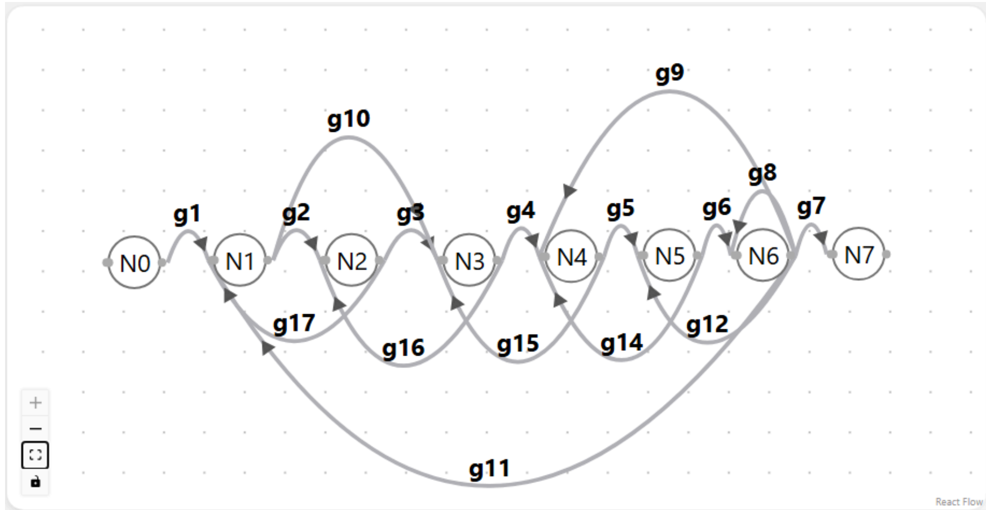
Delta Values for Each Forward Path

Path	Delta Value
Path 1	-1

Example 3:

Constructed Graph:

(Every edge gain is 1)



Graph Analysis:

Signal Flow Graph Results

Transfer Function

1

Delta (Δ)

2

Forward Paths

Path	Gain
N0 → N1 → N2 → N3 → N4 → N5 → N6 → N7	1
N0 → N1 → N3 → N4 → N5 → N6 → N7	1

Individual Loops

Loop	Gain
N1 → N2 → N3 → N4 → N5 → N6	1
N1 → N2	1
N1 → N3 → N4 → N5 → N6	1
N1 → N3 → N2	1
N2 → N3	1
N3 → N4	1
N4 → N5 → N6	1
N4 → N5	1
N5 → N6	1
N6	1

2 Non-Touching Loops

Loops	Gain
N1 → N2 & N3 → N4	1
N1 → N2 & N4 → N5 → N6	1
N1 → N2 & N4 → N5	1
N1 → N2 & N5 → N6	1
N1 → N2 & N6	1
N1 → N3 → N2 & N4 → N5 → N6	1
N1 → N3 → N2 & N4 → N5	1
N1 → N3 → N2 & N5 → N6	1
N1 → N3 → N2 & N6	1
N2 → N3 & N4 → N5 → N6	1
N2 → N3 & N4 → N5	1
N2 → N3 & N5 → N6	1
N2 → N3 & N6	1
N3 → N4 & N5 → N6	1
N3 → N4 & N6	1
N4 → N5 & N6	1

3 Non-Touching Loops

Loops	Gain
N3 → N4 & N5 → N6 & N1 → N2	1
N3 → N4 & N6 & N1 → N2	1
N6 & N4 → N5 & N1 → N2	1
N6 & N4 → N5 & N1 → N3 → N2	1
N6 & N4 → N5 & N2 → N3	1

Delta Values for Each Forward Path

Path	Delta Value
Path 1	1
Path 2	1

Part 2

Routh Stability Criterion

Features:

- **Stability Check Using Routh-Hurwitz Criterion:** Determines whether the system is stable based on its characteristic equation by analyzing the signs of the first column in the Routh array. The program fully handles all special cases, including rows of all zeros and zeros in the first column, using appropriate auxiliary equations and epsilon substitution techniques.

- **Unstable Poles Detection:** If the system is found to be unstable, the program identifies and displays the number and exact values of poles located in the right half of the s-plane. Special cases in the Routh table are also handled correctly to ensure accurate pole detection.

Modules and Data Structure:

- `Routh_criteria.py`

  Role:

  Serves as the core module to analyze system stability using the Routh-Hurwitz criterion.

  Structure:

  - Accepts the coefficients of the characteristic equation as input.
  - Uses **NumPy arrays** to construct the Routh table and store intermediate results.
  - Outputs a dictionary containing:
    - `"message"`: Stability status (stable/unstable)
    - `"matrix"`: The Routh array (2D array)
    - `"poles"`: A list of unstable poles, if any

Main Modules:

- `routh_criteria(coeffs, n)`
  - Inputs:
    - `coeffs`: List of polynomial coefficients in descending order of powers.
    - `n`: Order (degree) of the polynomial.
  - Processes:
    - Builds the Routh array row by row.
    - Applies auxiliary equation handling and epsilon substitution for special rows.
    - Analyzes the first column for sign changes.
    - Calculates roots and identifies unstable poles if needed.
  - Outputs:
    - Stability message
    - Full Routh array
    - Unstable poles (if applicable)

Algorithms used:

1- Routh-Hurwitz Criterion

- Classical control theory method to determine system stability without solving for roots.
- Determines the number of right-half-plane poles from sign changes in the first column of the Routh table.

2- Auxiliary Equation Technique

- Used when a row of zeros appears in the Routh array.
- The auxiliary polynomial is derived from the previous row, then differentiated to form a new row.

3- Epsilon Substitution

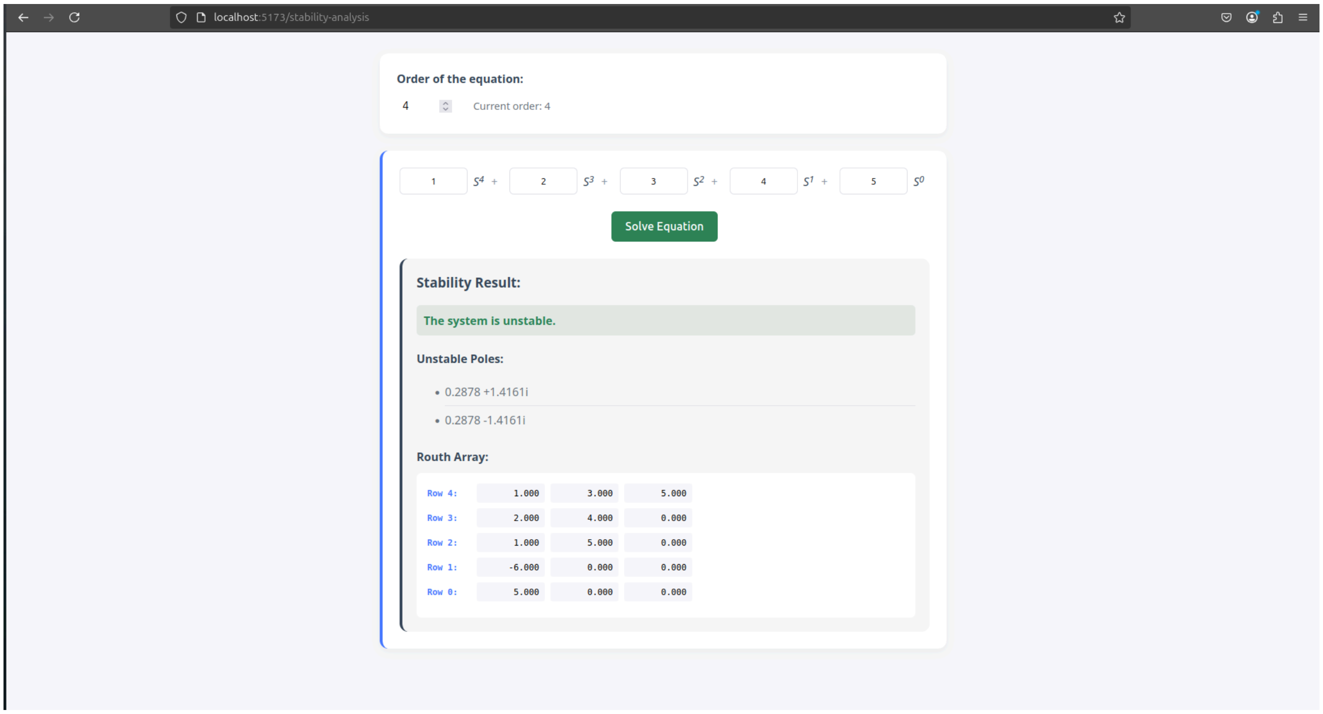
- Used when the first element in a row is zero (but the rest are not).
- Substitutes a small epsilon (`1e-10`) to continue computation safely and preserve matrix structure.

4- Root Calculation (NumPy)

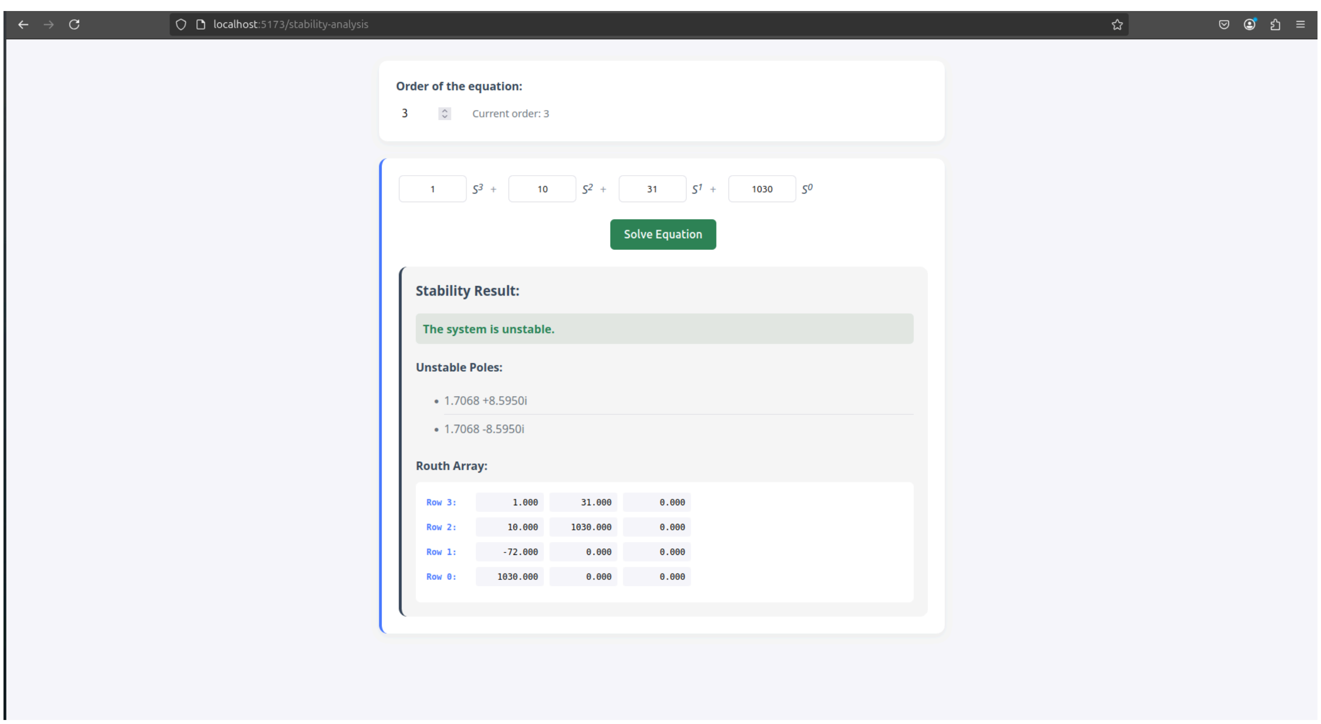
- When instability is detected, roots of the polynomial are calculated using `np.roots(coeffs)`.
- Real parts of the roots are checked to identify and display unstable poles.

Sample runs:

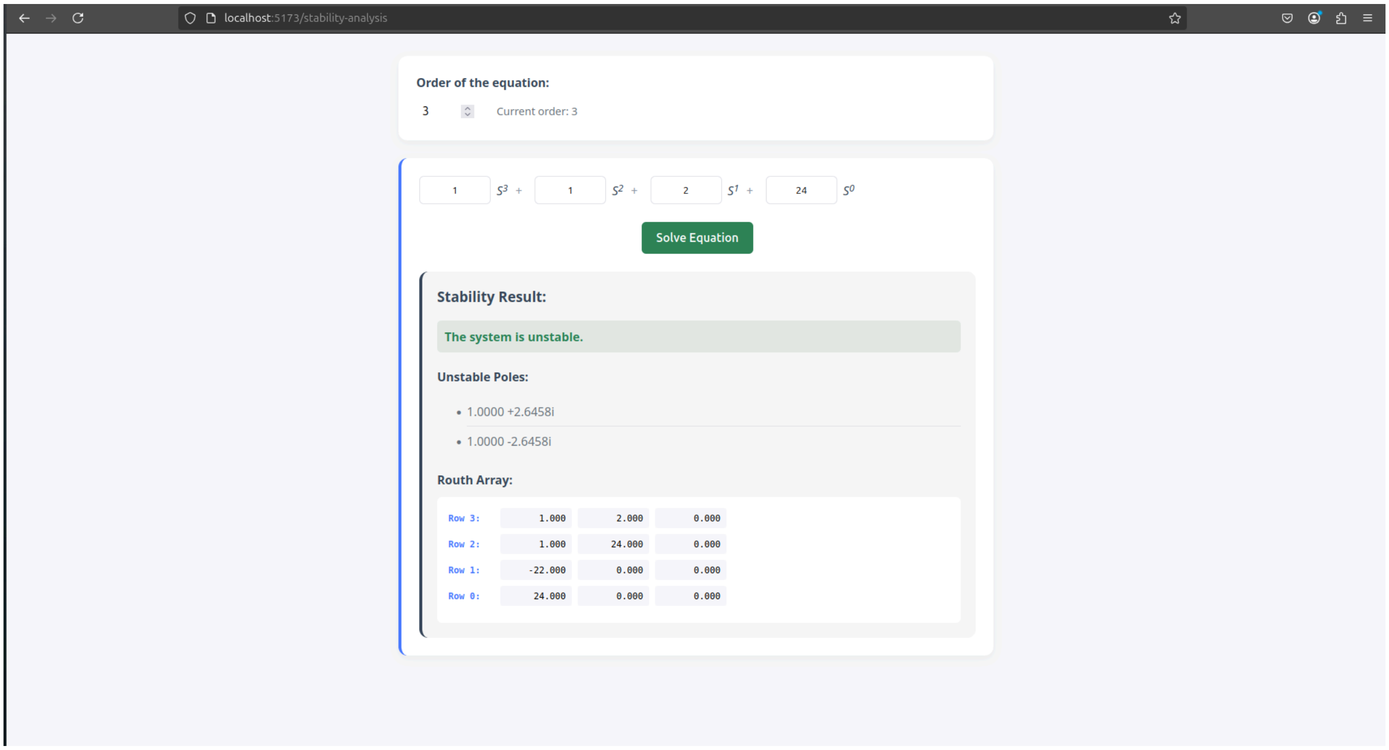
Example 1:



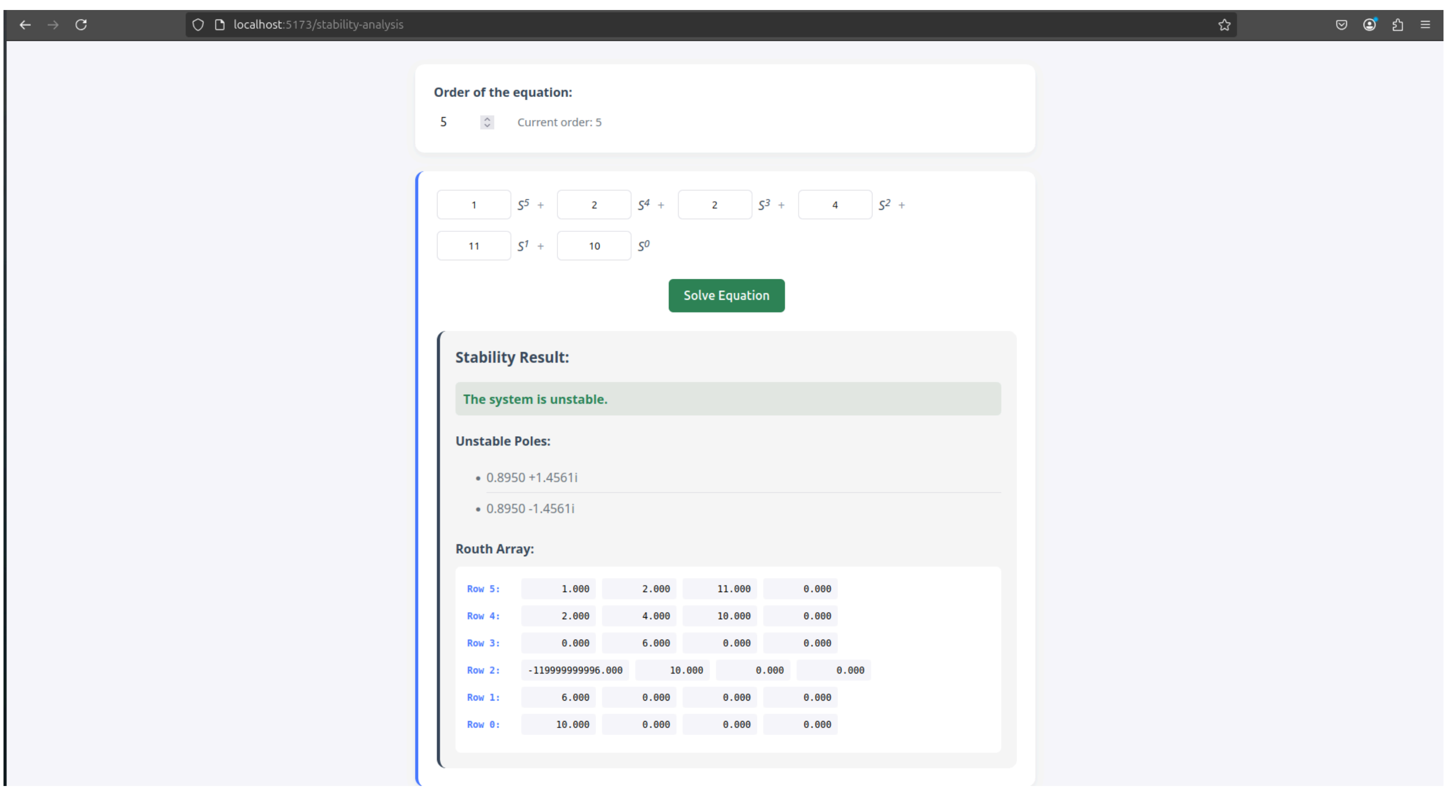
Example 2:



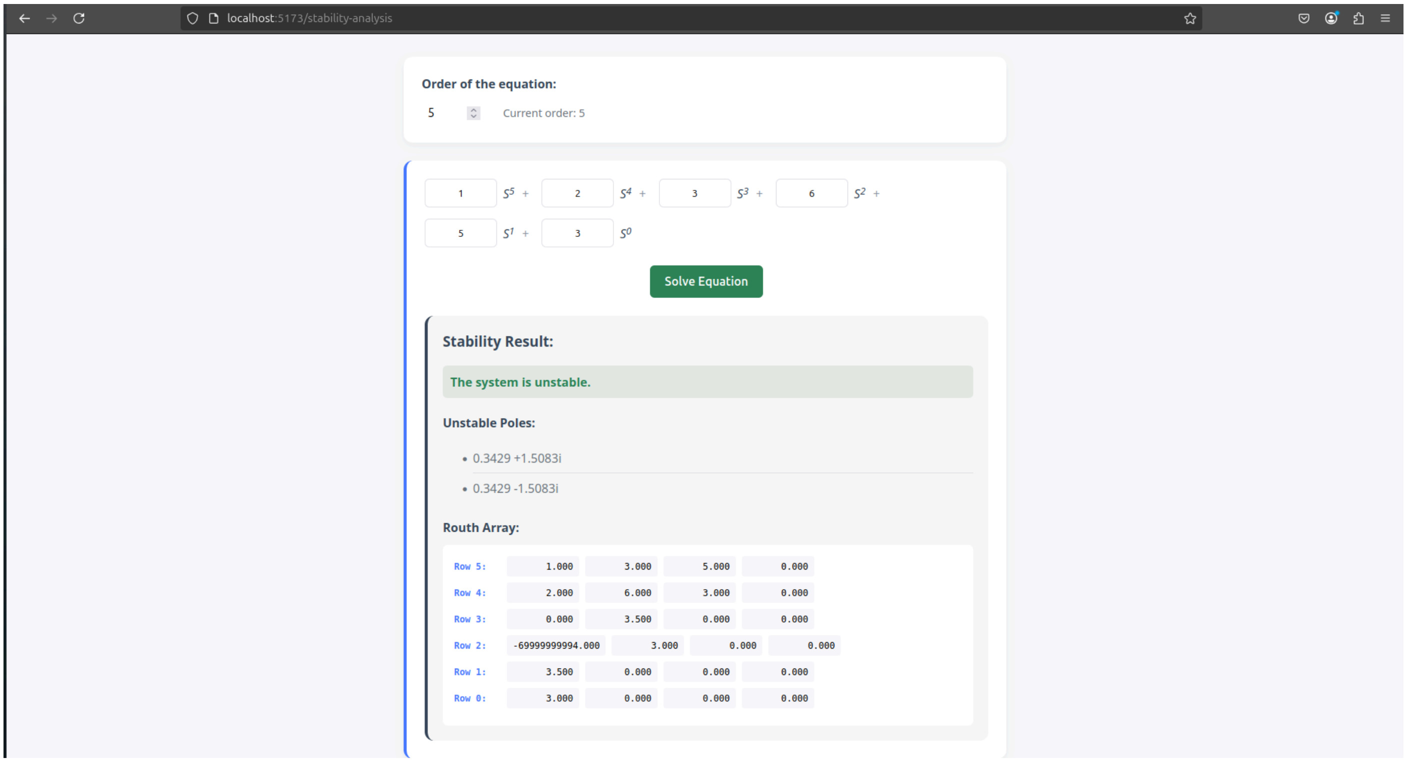
Example 3:



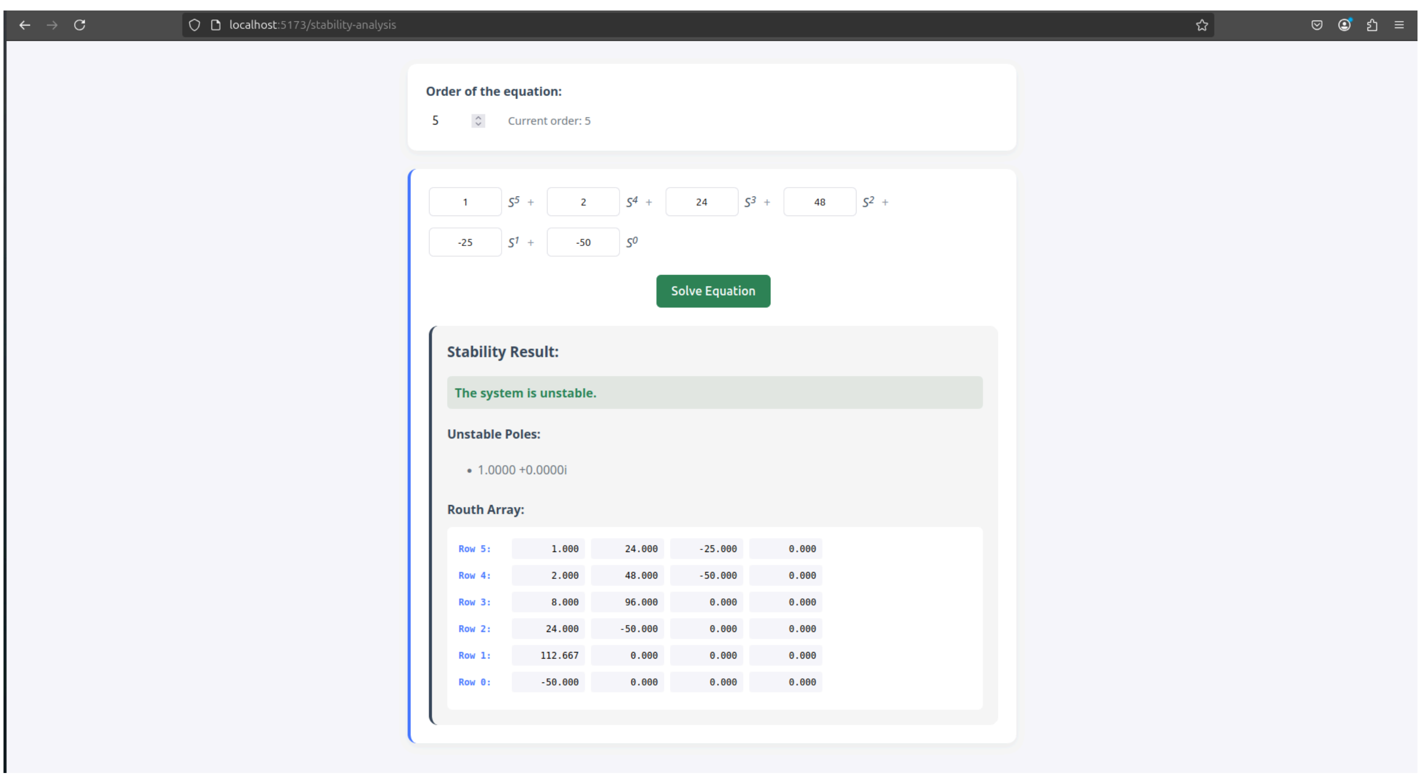
Example 4 (special case) :



Example 5 (special case):



Example 6 (special case) :



User Manual:

First, Clone the GitHub repository then open the backend folder and run this 2 lines:

Assuming you already have python installed.
pip install flask
Python main.py

Second, opened the frontend folder and run this 3 lines:

Assuming you already have react installed with npm
npm install
npm install @xyflow/react
npm install react-router-dom
npm run dev

Now open in your browser this link: http://localhost:5173/

You will find a menu with two pages: one for **Stability Analysis** and another for the **Signal Flow Graph**.

In the Stability Analysis:

First, user should enter the degree of the characteristic equation
Second, fill the coefficients of the equation
Third, press Solve to start the analysis and see results

The result is composed of 3 things:
1. The status of the equation (stable or unstable)
2. Positive real roots if any
3. The routh criteria table

In Signal Flow Graph Analysis:

- User can drag the Node button into the canva to add a node.
- Connect nodes with each other by dragging the handle of a node to the other.
- The right handle of a Node is considered its output while the left handle is the input.
- You can't connect to output handles with each other or input.
- You can click on the edge to adjust its curveture.
- Under the canva user can write the gain of each edge.
- After Constructing the graph you can press the analyse button to get the results

The result is composed of transfer function, nodes in each loop and their gains, paths, non-touching loops and deltas.