Report : Signal Flow Graph & Stability Solver

➔ Problem Statement :~

- The problem is separated in two parts :-

1st part :

- The main objective is to draw the signal flow graph representing a control system, being able to edit it , and more importantly analyzing in order to get :~
  1. The Forward Paths.
  2. The Main Loops
  3. The Delta Values {Δ, Δ1, Δ2, Δ3, …. Δm} where m = the number of the forward paths
  4. Transfer Function Value.

2nd part :

- The main objective is to test the stability of the transfer function, given the denominator of the transfer function in the S-plane domain, it's essential to draw the Routh table used to get the answer.

➔ Main Modules & Algorithms used :~

➢ Folder 1 : Backend

○ Inside " api " Folder :- routes.py

Snapshot of the code :-

```
from flask import Blueprint, request, jsonify
from services.signal_flow_graph import analyze_signal_flow_graph
from services.stability_analysis import routh_stability_analysis
from services.signal_flow_graph import convert_sympy_to_str
api_bp = Blueprint('api', __name__)
```

Main Import Libraries	Use
Flask	Used for getting the requests from the frontend side, and acquire essential data to start analyzing the graph { Edges & Nodes }
Services.signal_flow_graph -> analyze_signal_flow_graph	Custom file in backend made to analyze the acquired data from the frontend and get the essential values needed to get the transfer function.
Services.stability_analysis -> routh_stability_analysis	Custom file in backend made to analyze the acquired data from the frontend and get the essential values needed to determine the stability of the system using Routh table.
Services.signal_flow_graph -> convert_sympy_to_str	Custom Class in services->signal_flow_graph used to change python sympy library variables to string in order to be able to multiply and add variable path weights (EX: a1*a2*a3*….*am)

```python
@api_bp.route('/analyze-graph', methods=['POST'])
def analyze_graph():
    """
    Analyze a signal flow graph
    Expected JSON input:
    {
        "nodes": [list of node IDs],
        "branches": [
            {"from": "node1", "to": "node2", "gain": 1.0}
        ]
    }
    """
    data = request.json
    try:
        result = analyze_signal_flow_graph(data['nodes'], data['branches'])
        safe_result = convert_sympy_to_str(result)
        print("Safe Result:", safe_result)
        return jsonify(safe_result)
    except Exception as e:
        print("Error:", e)
        return jsonify({"error": str(e)}), 400

@api_bp.route('/stability-analysis', methods=['POST'])
def stability():
    """
    Perform Routh stability analysis
    Expected JSON input:
    {
        "degree": 3,
        "equation": [list of coefficients] from highest power to lowest # Example: [1, 1, 10, 72, 152, 240]
    }
    """
    data = request.json

    try:
        result = routh_stability_analysis(data['equation'], data['degree'])
        return jsonify(result)
    except Exception as e:
        return jsonify({"error": str(e)}), 400
```

- The data is acquired from the system when the button << Analyze Graph >> or <<Analyze Stability>> (In case in Stability Analysis using Routh table)
- If the route of the program is in **‘ ./analyze-graph ‘**, the following course of actions happens :
- The result that is in the form of the python dictionary is given in the shape {
  "forward_paths": forward_paths,
  "forward_path_gains": forward_path_gains,
  "loops": loops,
  "loop_gains" : loop_gains,
  "delta" : delta,
  "delta_values" : delta_values,
  "transfer_function" : transfer_function}
- The the result is then stringified to handle sympy values and be able to printed
- If the route of the program is in **‘ ./stability-analysis ‘**, the following course of actions happens :
- The result that is in the form of the python dictionary is given in the shape {
  "is_stable" : is_stable,
  "routh_array" : routh_array.tolist(),
  "rhs_poles" : sign_changes,}

- **Inside “ Mason ” Folder :- DeltaCalculations.py**

```
from Mason.nontouchingdetection import find_non_touching_loops
def calculate_delta_values(forward_paths, loops, loop_gains):
    """
```

- From the same file **“ Mason ”** , the class **Find_non_touching_loops** is imported to get all the non – touching loops for further use
- The calculation function uses three parameters (
  forward_paths -> include all the forward paths of the system
  loops -> includes all the loops of the system in literals (not using gains, EX : 1 -> 2 -> 3 ->..)
  loop_gains -> includes all the loops of the system in gains)

```
non_touching_loops = find_non_touching_loops(loops)
print("Non-touching loops detected:", non_touching_loops)
# Calculate the main delta value
delta = 1.0

# Subtract sum of all individual loop gains
for i, gain in enumerate(loop_gains):
    delta -= gain

# Add sum of products of gains of all combinations of two non-touching loops
if 2 in non_touching_loops:
    for i, j in non_touching_loops[2]:
        delta += loop_gains[i] * loop_gains[j]

# Subtract sum of products of gains of all combinations of three non-touching loops
if 3 in non_touching_loops:
    for i, j, k in non_touching_loops[3]:
        delta -= loop_gains[i] * loop_gains[j] * loop_gains[k]

# Add sum of products of gains of all combinations of four non-touching loops
if 4 in non_touching_loops:
    for i, j, k, l in non_touching_loops[4]:
        delta += loop_gains[i] * loop_gains[j] * loop_gains[k] * loop_gains[l]

# Subtract sum of products of gains of all combinations of five non-touching loops
if 5 in non_touching_loops:
    for i, j, k, l, m in non_touching_loops[5]:
        delta -= loop_gains[i] * loop_gains[j] * loop_gains[k] * loop_gains[l] * loop_gains[m]
```

- The non – touching loops are acquired using **Find_non_touching_loops**
- The delta value is defined and then for each non – touching loops, their resultant is obtained and then substracted from the delta value.
- The system handles up to 5 non – touching loops’ control systems.

```python
# Calculate delta values for each forward path
delta_values = []

for path_idx, path in enumerate(forward_paths):
    # For each forward path, we need to remove loops that touch this path
    path_nodes = set(path)

    # Initialize delta value for this path
    path_delta = 1.0

    # Find loops that don't touch this path
    non_touching_path_loops = []
    non_touching_path_gains = []

    for loop_idx, loop in enumerate(loops):
        # Check if loop touches the path (remove last node which is duplicate of first)
        loop_nodes = set(loop[:-1])
        if len(path_nodes.intersection(loop_nodes)) == 0:
            non_touching_path_loops.append(loop)
            non_touching_path_gains.append(loop_gains[loop_idx])

    # If there are no non-touching loops, delta_k = 1
    if not non_touching_path_loops:
        delta_values.append(1.0)
        continue
```

- A delta values list is defined to insert the delta values in.
- The path nodes is then acquired from each forward path in the list forward_paths
- The delta value is then defined as **path_delta**
- For each path, two new lists defined to hold the loops & loops' gains of each path
- The program then detect the existence of an intersaction between the the path and the loops
- If an intersection is detected, the system ignore this loop as if it's touching the current path
- Otherwise, it's inserted in its loops & loops' gains list as it's not touching the path

- **Inside " Mason " Folder :- nontouchingdetection.py**

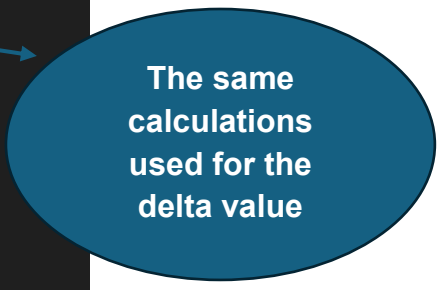
```python
def find_non_touching_loops(loops):
    """
    Find all combinations of non-touching loops in a signal flow graph.

    Args:
        loops: List of loops, where each loop is a list of nodes

    Returns:
        Dictionary with keys as the number of loops in combination (2, 3, etc.)
        and values as lists of tuples, where each tuple contains indices of non-touching loops
    """
    non_touching_loops = {2: [], 3: [], 4: [], 5: []}  # Store combinations by size

    # Check if two loops are non-touching
    def are_non_touching(loop1, loop2):
        # Remove the duplicate end node from each loop for comparison
        nodes1 = set(loop1[:-1])
        nodes2 = set(loop2[:-1])
        # If they share no common nodes, they are non-touching
        return len(nodes1.intersection(nodes2)) == 0
```

- The non – touching loops' list is initiated for 5 non – touching loops as a limit.
- For each two loops, we detect if there's an intersection.

```
# Find pairs of non-touching loops
n_loops = len(loops)
for i in range(n_loops):
    for j in range(i+1, n_loops):
        if are_non_touching(loops[i], loops[j]):
            non_touching_loops[2].append((i, j))

# Find triplets of non-touching loops
for i, j in non_touching_loops[2]:
    for k in range(max(i, j)+1, n_loops):
        if are_non_touching(loops[i], loops[k]) and are_non_touching(loops[j], loops[k]):
            non_touching_loops[3].append((i, j, k))

# Find quadruplets of non-touching loops
for i, j, k in non_touching_loops[3]:
    for l in range(max(i, j, k)+1, n_loops):
        if (are_non_touching(loops[i], loops[l]) and
            are_non_touching(loops[j], loops[l]) and
            are_non_touching(loops[k], loops[l])):
            non_touching_loops[4].append((i, j, k, l))

# Find quintuplets of non-touching loops
for i, j, k, l in non_touching_loops[4]:
    for m in range(max(i, j, k, l)+1, n_loops):
        if (are_non_touching(loops[i], loops[m]) and
            are_non_touching(loops[j], loops[m]) and
            are_non_touching(loops[k], loops[m]) and
            are_non_touching(loops[l], loops[m])):
            non_touching_loops[5].append((i, j, k, l, m))

# Remove empty categories
for key in list(non_touching_loops.keys()):
    if not non_touching_loops[key]:
        del non_touching_loops[key]

return non_touching_loops
```

- We detect if the two selected loops are touching or not
- Then, we check if other loops are touching this two or not
- Following this model as a recursion to detect if there's multiple non – touching loops

```python
# Subtract sum of all individual non-touching loop gains
for gain in non_touching_path_gains:
    path_delta -= gain

# Find non-touching combinations within the non-touching path loops
path_non_touching = find_non_touching_loops(non_touching_path_loops)

# Map the original loop indices to the new indices in non_touching_path_loops
loop_index_map = {loops.index(loop): i for i, loop in enumerate(non_touching_path_loops)}

# Add sum of products of gains of combinations of two non-touching loops
if 2 in path_non_touching:
    for i, j in path_non_touching[2]:
        path_delta += non_touching_path_gains[i] * non_touching_path_gains[j]

# Subtract sum of products of gains of combinations of three non-touching loops
if 3 in path_non_touching:
    for i, j, k in path_non_touching[3]:
        path_delta -= (non_touching_path_gains[i] *
                       non_touching_path_gains[j] *
                       non_touching_path_gains[k])

# Add sum of products of gains of combinations of four non-touching loops
if 4 in path_non_touching:
    for i, j, k, l in path_non_touching[4]:
        path_delta += (non_touching_path_gains[i] *
                       non_touching_path_gains[j] *
                       non_touching_path_gains[k] *
                       non_touching_path_gains[l])

# Subtract sum of products of gains of combinations of five non-touching loops
if 5 in path_non_touching:
    for i, j, k, l, m in path_non_touching[5]:
        path_delta -= (non_touching_path_gains[i] *
                       non_touching_path_gains[j] *
                       non_touching_path_gains[k] *
                       non_touching_path_gains[l] *
                       non_touching_path_gains[m])
```



- **Inside " Mason " Folder :- mason.py**

```python
def mason_gain(forward_paths, forward_path_gains, loops, loop_gains, delta, delta_values):
    """
    Compute the overall transfer function using Mason's Gain Formula.

    Parameters:
    - forward_paths: List of forward paths (not used directly in computation, for reference)
    - forward_path_gains: List of gains for each forward path [P1, P2, ...]
    - loops: List of loops (not used directly, for reference)
    - loop_gains: List of individual loop gains
    - delta: Overall delta of the system
    - delta_values: List of delta values corresponding to each forward path [Δ1, Δ2, ...]

    Returns:
    - Transfer function T (float or symbolic, depending on input)
    """

    if len(forward_path_gains) != len(delta_values):
        raise ValueError("Mismatch between number of forward path gains and delta values.")

    numerator = sum(P * D for P, D in zip(forward_path_gains, delta_values))
    transfer_function = numerator / delta

    return transfer_function
```

- **Inside " Services " Folder :- signal_flow_graph.py**

```python
import networkx as nx
import numpy as np
import sympy as sp
from Mason.mason import mason_gain
from Mason.DeltaCalculations import calculate_delta_values
from sympy import Basic

def analyze_signal_flow_graph(nodes, branches):
```

Main Imports	Use
networkx	creating, manipulating, and analyzing complex networks or graphs. In the provided code, networkx is used to represent and analyze the Signal Flow Graph (SFG)
Numpy & Sympy	Used for handling variable path weights and calculations of the gains
Mason -> mason_gain	Used to calculate the main gain of mason (value fo the transfer function)
Mason –> DeltaCalculations	Used to calculate all the delta values

```python
# Create a directed graph
G = nx.DiGraph()

# Add nodes and edges
for node in nodes:
    G.add_node(node)

for branch in branches:
    if isinstance(branch['gain'], str):
        gain = sp.Symbol(branch['gain'])
    else:
        gain = branch['gain']
    G.add_edge(branch['from'], branch['to'], weight=gain)
# Find forward paths (assuming first node is input and last node is output)
source = nodes[0]
sink = nodes[-1]
forward_paths = list(nx.all_simple_paths(G, source=source, target=sink))
print("Forward Paths:", forward_paths)
# Calculate forward path gains
forward_path_gains = []
for path in forward_paths:
    gain = 1.0
    for i in range(len(path) - 1):
        edge_data = G.get_edge_data(path[i], path[i+1])
        gain *= edge_data['weight']
    forward_path_gains.append(gain)
print("Forward Path Gains:", forward_path_gains)
# Find all loops
loops = []
for cycle in nx.simple_cycles(G):
    if len(cycle) > 1:  # Exclude self-loops
        # Add the start node at the end to represent the cycle
        cycle_with_return = cycle + [cycle[0]]
        loops.append(cycle_with_return)
```

- Defining the directed graph G that will hold the signal flow graph for later use.
- Checking every edge and include it in the directed graph G using the source node, target node, and the weight of this edge.
- We get each forward path gain for later use
- we detect each loop in the directed graph and appended in a new list for all the loops

```python
# Calculate loop gains
loop_gains = []
for loop in loops:
    gain = 1.0
    for i in range(len(loop) - 1):
        edge_data = G.get_edge_data(loop[i], loop[i+1])
        gain *= edge_data['weight']
    loop_gains.append(gain)

# TODO: Implement non-touching loops detection
# TODO: Calculate delta values
# TODO: Calculate transfer function // finished

# Placeholder for now
delta = 1.0
delta_values = [1.0] * len(forward_paths)
delta , delta_values = calculate_delta_values(forward_paths, loops, loop_gains)
transfer_function = mason_gain(forward_paths, forward_path_gains, loops, loop_gains, delta, delta_values)

if not isinstance(transfer_function, (int, float)):
    transfer_function = sp.simplify(transfer_function)
print("Transfer Function:", transfer_function)
print("Delta:", delta)
print("Delta Values:", delta_values)
return {
    "forward_paths": forward_paths,
    "forward_path_gains": forward_path_gains,
    "loops": loops,
    "loop_gains": loop_gains,
    "delta": delta,
    "delta_values": delta_values,
    "transfer_function": transfer_function
}
```

(variable) forward_paths: list

- we get all the loops' gains from the directed graph G
- we use already defined function in **" Services "** , to calculate all the needed variables {delta values, transfer function & delta}

```python
def convert_sympy_to_str(obj):
    if isinstance(obj, Basic):
        return str(obj)
    elif isinstance(obj, list):
        return [convert_sympy_to_str(i) for i in obj]
    elif isinstance(obj, dict):
        return {k: convert_sympy_to_str(v) for k, v in obj.items()}
    else:
        return obj
```

- Converting all sympy variables into string to be able to display it after the analyzation.

- **Inside " Services " Folder :- stability_analysis.py**

```python
def routh_stability_analysis(equation, degree):
    """
    Perform Routh stability analysis on a characteristic equation.
    Args:
        equation: List of coefficients from highest power to lowest
        degree: Degree of the polynomial
    Returns:
        Dictionary with stability status and details
    """
    n = int(degree + 1)  # Number of coefficients
    coefficients = equation

    # Ensure we have the right number of coefficients
    if len(coefficients) != n:
        return {"error": f"Expected {n} coefficients for degree {degree}, got {len(coefficients)}"}

    # Initialize Routh array
    routh_array = np.zeros((n, (n + 1) // 2))

    # Fill first two rows with coefficients
    counter1, counter2 = 0, 0
    for i in range(n):
        if i % 2 == 0:
            routh_array[0, counter1] = float(coefficients[i])
            counter1 += 1
        else:
            routh_array[1, counter2] = float(coefficients[i])
            counter2 += 1
```

- The routh stability analysis function is defined and takes two parameters provided in the request coming from the frontend GUI {
  Equation : the denominator of the transfer function who determines the stability
  Degree : the highest power given to S that determines the degree of equation}
- Starting by defining the number of coefficients which is the **degree** incremented by 1.
- Defining the Routh array which represent the Routh table using the dimension:
  n -> representing the rows, as the power of each S in this row (from highest to lowest)
  (n+1) // 2 -> representing the columns, as the coefficients of each consequent oddly powered & evenly powered S's is stated in the same column (For example if the degree = 6, which means the number of coefficients = 7).

(n+1)//2
->
(7+1)//2
-> 4 ✓

A_S^6	A_S^4	A_S^2	A_S^0
A_S^5	A_S^3	A_S^1	**0.0**

- Defining the first two rows of the Routh array with the coefficients

```
# Calculate remaining rows
for i in range(2, n):
    # Check if previous row is all zeros
    if np.all(np.abs(routh_array[i-1, :]) < 1e-10):
        # Replace with derivative of the row above it
        for j in range((n + 1) // 2 - 1):
            if j + (i-2)//2 + 1 < (n+1)//2:  # Ensure we're within array bounds
                # Calculate coefficient for the auxiliary polynomial
                power = degree - (i-2) - 2*j
                if power >= 0:
                    routh_array[i-1, j] = routh_array[i-2, j] * power

    # Check if first element is zero (or very small)
    if abs(routh_array[i-1, 0]) < 1e-10:
        routh_array[i-1, 0] = 1e-6  # Replace with small value

    # Calculate the row
    for j in range((n + 1) // 2 - 1):
        if j+1 < (n+1)//2:  # Ensure we don't go out of bounds
            try:
                routh_array[i, j] = (
                    routh_array[i-1, 0] * routh_array[i-2, j+1] -
                    routh_array[i-2, 0] * routh_array[i-1, j+1]
                ) / routh_array[i-1, 0]
            except:
                routh_array[i, j] = 0  # Handle any calculation errors

# Check for sign changes in first column
first_column = routh_array[:, 0]
sign_changes = 0

# Filter out zero values for sign change calculation
nonzero_elements = [val for val in first_column if abs(val) > 1e-10]
```

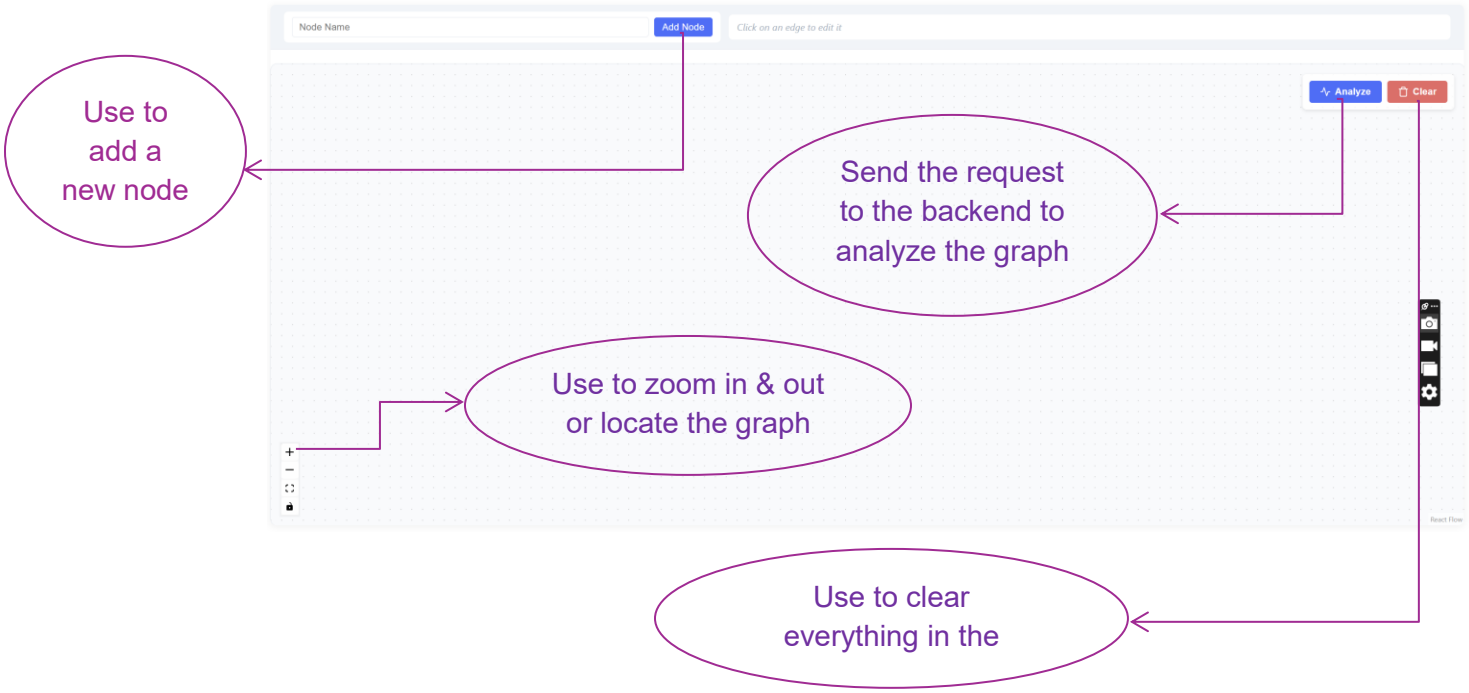
- Calculating the Routh remaining values using the formulas provided in the lecture:

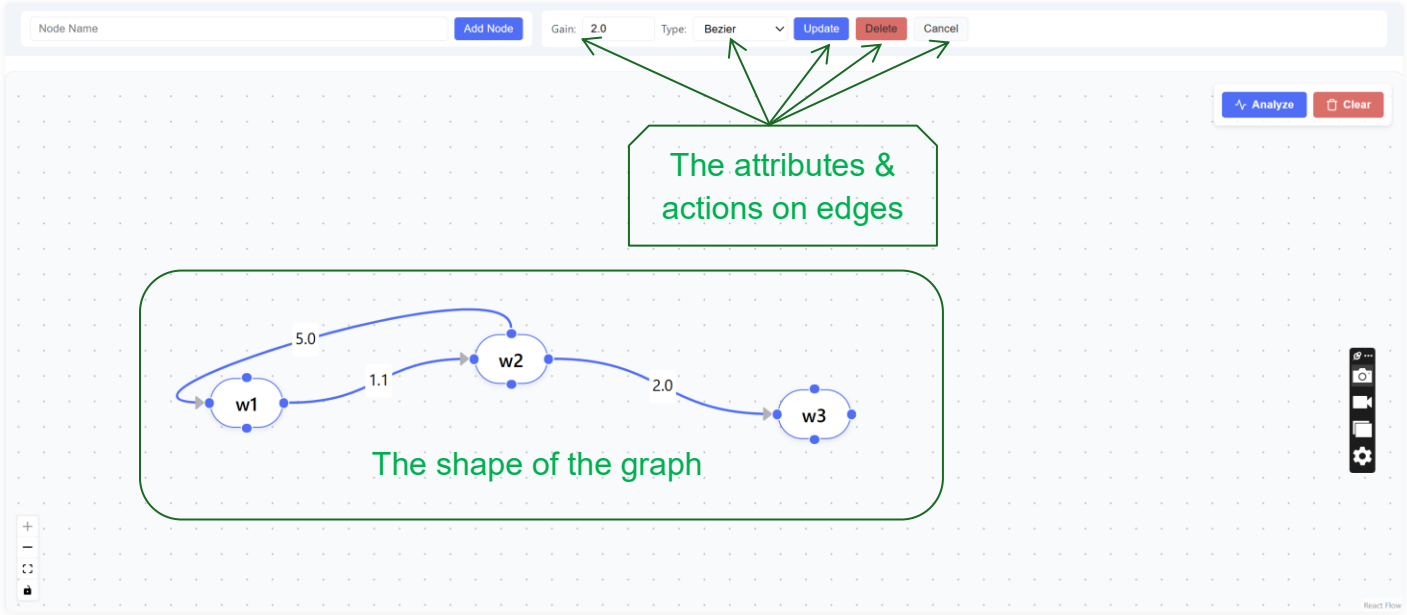
s^4	a_4	a_2	a_0	**0**
s^3	a_3	a_1	0	**0**
s^2	$b_1 = \dfrac{\begin{vmatrix} a_4 & a_2 \\ a_3 & a_1 \end{vmatrix}}{a_3}$	$b_2 = \dfrac{\begin{vmatrix} a_4 & a_0 \\ a_3 & 0 \end{vmatrix}}{a_3}$	$\mathbf{b3} = \dfrac{\begin{vmatrix} a_4 & 0 \\ a_3 & 0 \end{vmatrix}}{a_3}$	
s^1	$c_1 = \dfrac{\begin{vmatrix} a_3 & a_1 \\ b_1 & b_2 \end{vmatrix}}{b_1}$	$\mathbf{c_2} = \dfrac{\begin{vmatrix} a_3 & 0 \\ b_1 & 0 \end{vmatrix}}{b_1}$	$\mathbf{c_3} = \dfrac{\begin{vmatrix} a_3 & 0 \\ b_1 & 0 \end{vmatrix}}{b_1}$	
s^0	$d_1 = \dfrac{\begin{vmatrix} b_1 & b_2 \\ c_1 & 0 \end{vmatrix}}{c_1}$	$\mathbf{d_2} = \dfrac{\begin{vmatrix} b_1 & 0 \\ c_1 & 0 \end{vmatrix}}{c_1}$	$\mathbf{d_3} = \dfrac{\begin{vmatrix} b_1 & 0 \\ c_1 & 0 \end{vmatrix}}{c_1}$	

- Then we observe the 1st column of the Routh array then find the sign changes
- If there's sign change then the system is unstable, otherwise, it's stable

```python
for i in range(1, len(nonzero_elements)):
    if nonzero_elements[i-1] * nonzero_elements[i] < 0:
        sign_changes += 1

is_stable = sign_changes == 0 and all(abs(val) > 1e-10 for val in first_column)

return {
    "is_stable": is_stable,
    "routh_array": routh_array.tolist(),
    "rhs_poles": sign_changes,
}
```

- **Folder 2 : Frontend**
  - **Inside " components " Folder :- GraphEditor.jsx**





  - **Inside " components " Folder :- ResultDisplay.jsx**

Signal Flow Graph Analysis Results

Forward Paths

- Path 1: 1 → 2 → 3 (Gain: 2.200)

Individual Loops

- Loop 1: 1 → 2 → 1 (Gain: 5.500)

Delta Values

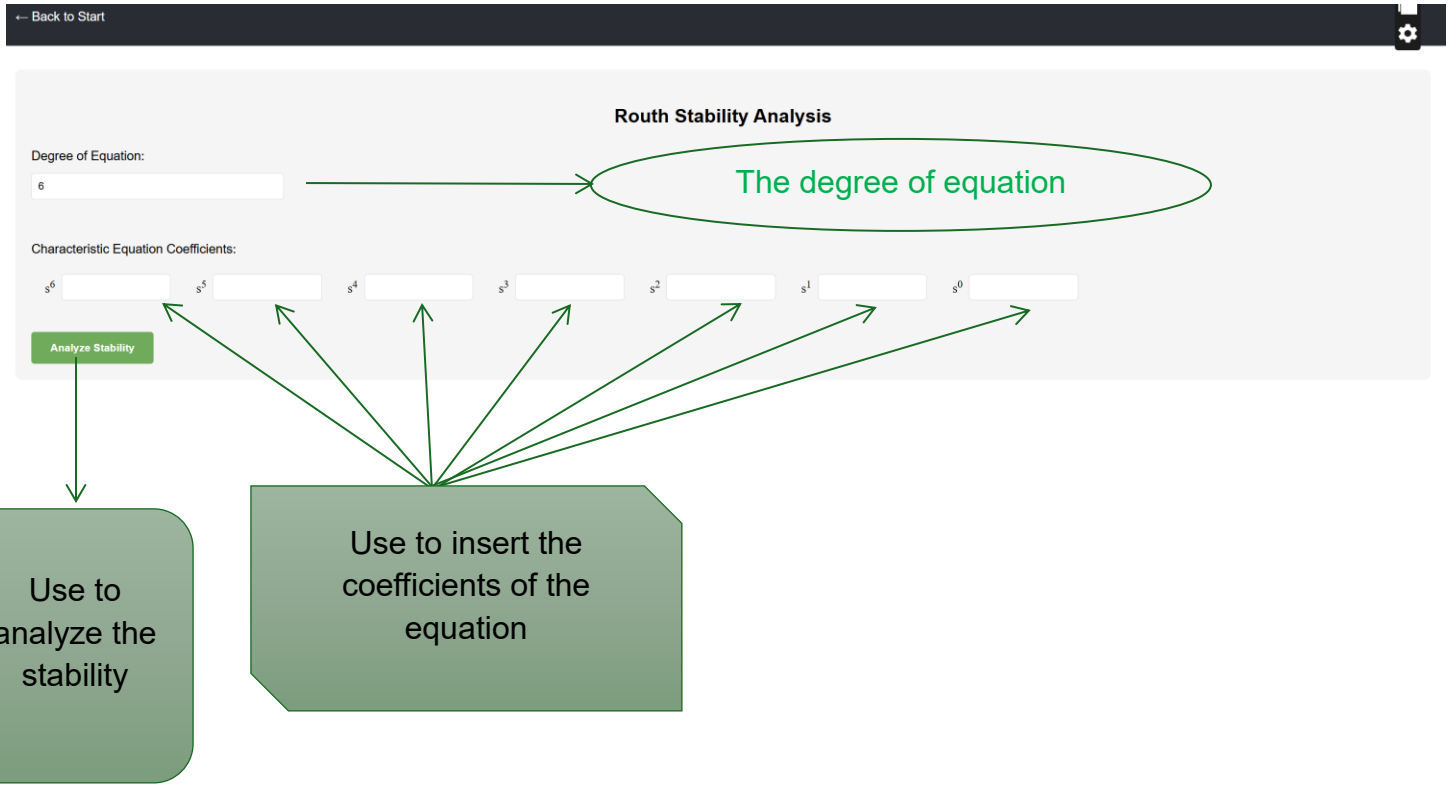
Δ = -4.500

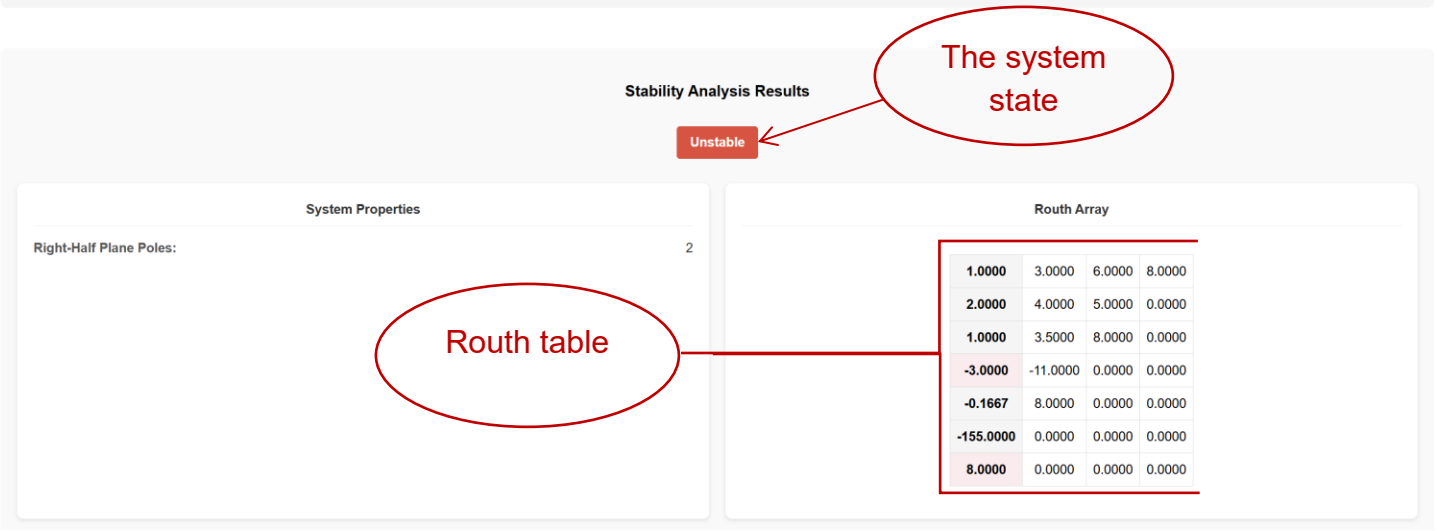
- Δ1 = 1.000

Transfer Function

-0.48888888888888893

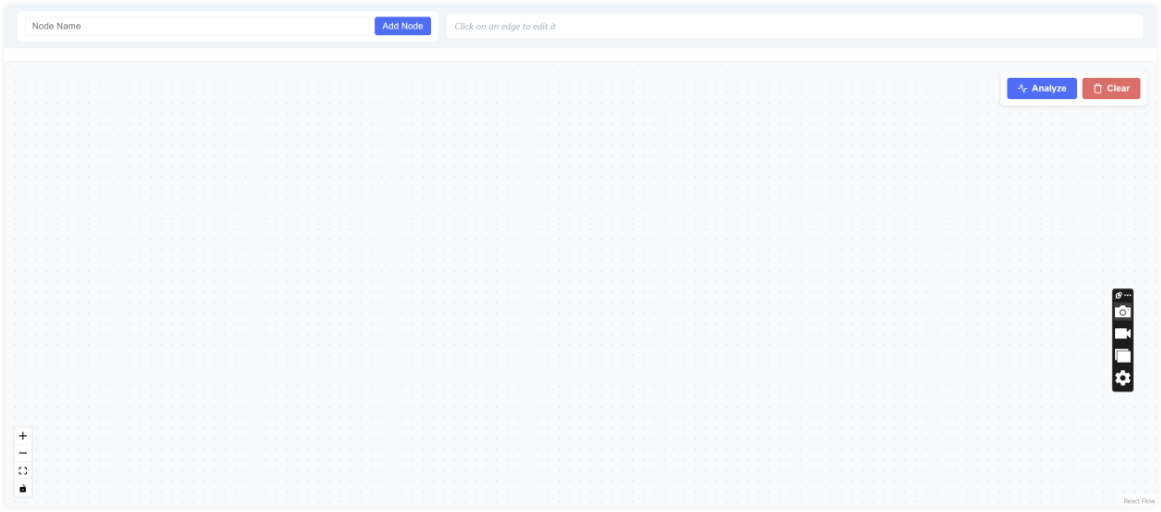
- **Inside " components " Folder :- StabilityAnalysis.jsx**



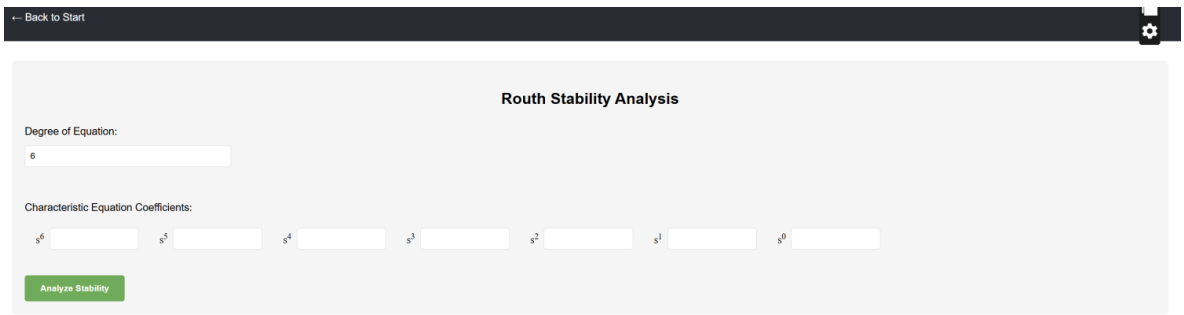


- **Inside " Pages " Folder :~**

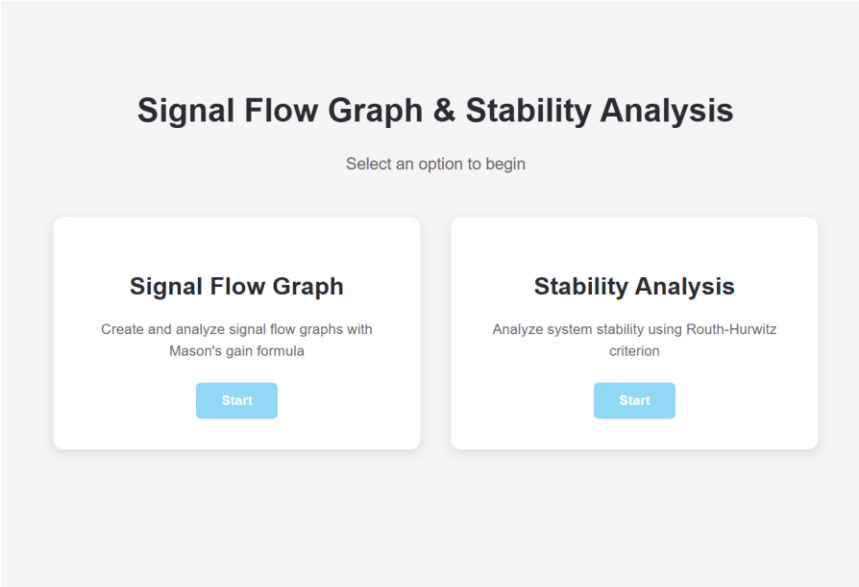
  > SignalFlowPage :



> StabilityPage :



> StartPage :



➔ Data Structures :~

Data Structures	Objective
Lists & Python dictionaries & strings	Used for common printing & returning, it also thw best option to display the results in the program
Numpy Array	Used to calculate with the numbers in the stabilization problems as there's no variable coefficients
Sympy Array	Used to calculate with the variable coefficients (or number ones) in the graph analyzation as there's a possibility to use variable coefficients
Networkx Graphs	Easy to use for calculation of the loops & paths In the graph as it provide a variety of functions to handle calculation in shape of graph

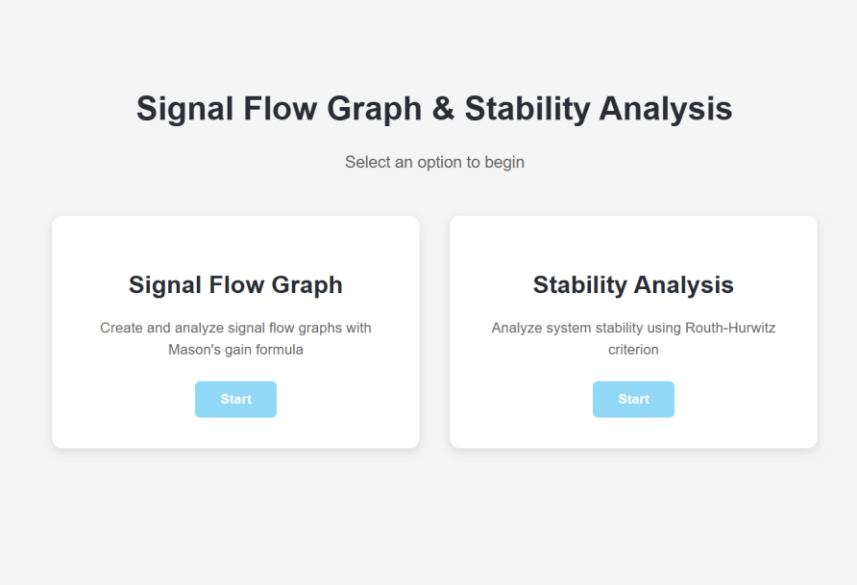
➔ Simple Guide to Run & Understand:~

- Reaching the file << SignalFlowGraph_and_RouthCriteria-master >>.
- Opening the terminal, then to run the frontend by running the commands :

  . cd front

  . npm run dev

- Copy the localhost site link to a new tab { In our case : " **http://localhost:5173/** "
- To be able to run the frontend, we need to activate the backend
- Using any IDE, we enter the file **" ./back/app.py "**
- Then we click on the run button : 
- The first encountered page is the following :



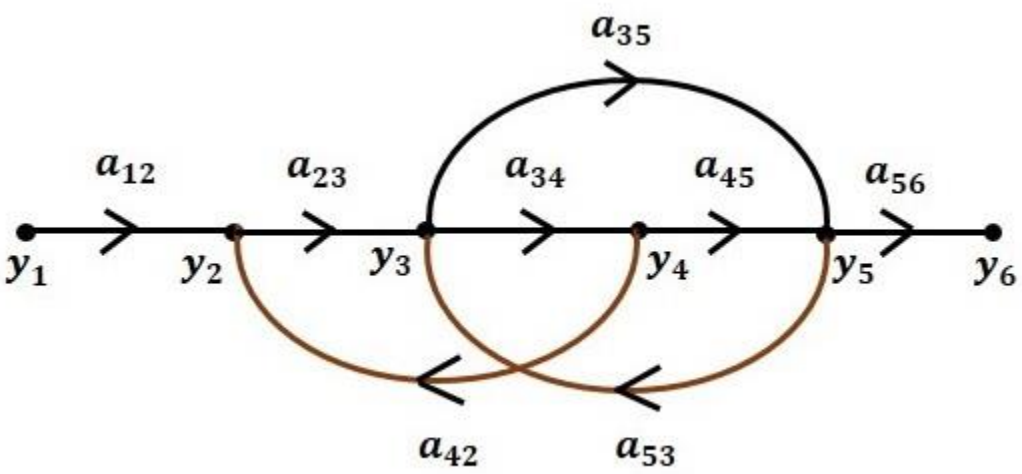
- In case of need of Signal Flow Graph, click on " Start " button on the section << Signal Flow Graph >>
- In case of need of Routh Table Analysis, click on " Start " button on the section << Stability Analysis >>

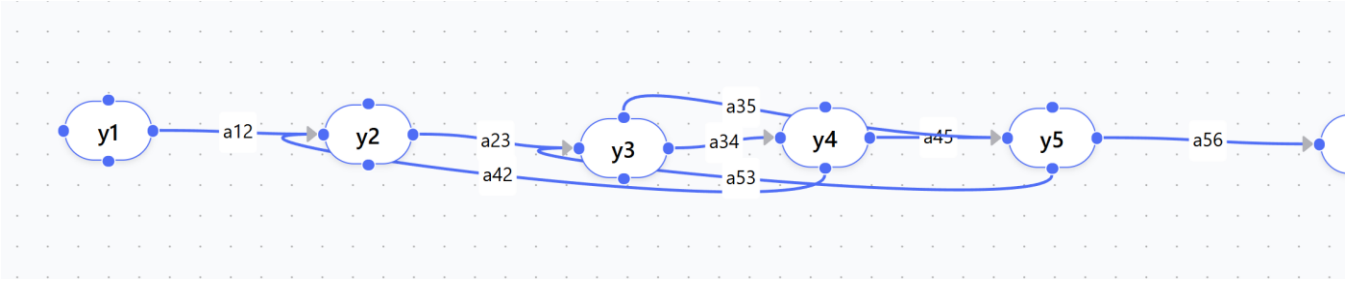
Remark:~

For Signal Flow graph , it 's always assume that the first node is the entering node, and the last one as the last result

➔ Sample Runs :~

○ Problem 1:~





Signal Flow Graph Analysis Results

Forward Paths

- Path 1: 1 → 2 → 3 → 4 → 5 → 6 (Gain: 1.0*a12*a23*a34*a45*a56)
- Path 2: 1 → 2 → 3 → 5 → 6 (Gain: 1.0*a12*a23*a35*a56)

Individual Loops

- Loop 1: 3 → 4 → 5 → 3 (Gain: 1.0*a34*a45*a53)
- Loop 2: 3 → 4 → 2 → 3 (Gain: 1.0*a23*a34*a42)
- Loop 3: 3 → 5 → 3 (Gain: 1.0*a35*a53)

Delta Values

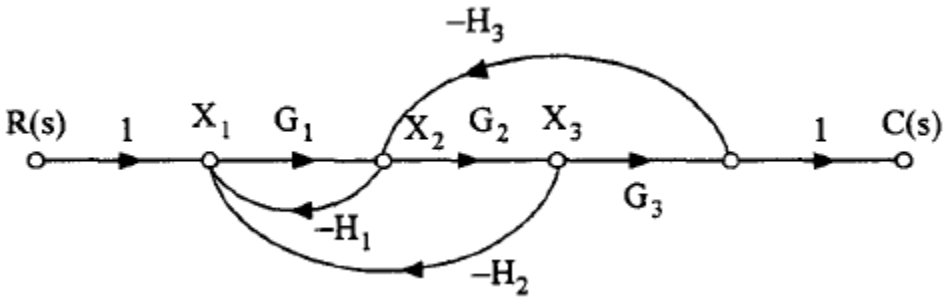
Δ = -1.0*a23*a34*a42 - 1.0*a34*a45*a53 - 1.0*a35*a53 + 1.0

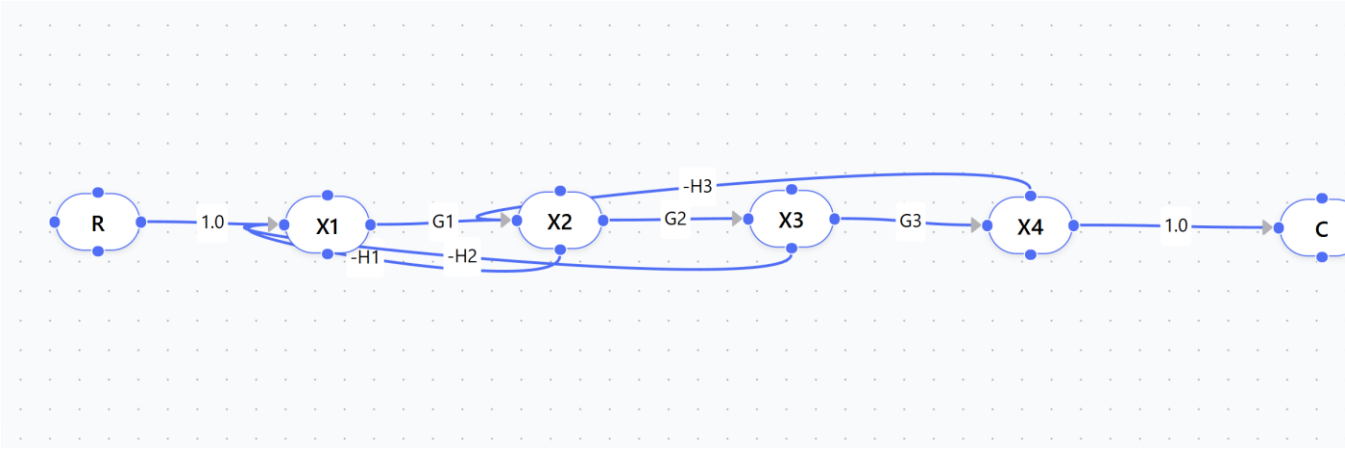
- Δ1 = 1.000
- Δ2 = 1.000

Transfer Function

1.0*a12*a23*a56*(-a34*a45 - a35)/(a23*a34*a42 + a34*a45*a53 + a35*a53 - 1)

- **Problem 2:~**





Signal Flow Graph Analysis Results

Forward Paths

- Path 1: 1 → 2 → 3 → 4 → 5 → 6 (Gain: 1.0*1.0**2*G1*G2*G3)

Individual Loops

- Loop 1: 3 → 4 → 5 → 3 (Gain: 1.0*-H3*G2*G3)
- Loop 2: 3 → 4 → 2 → 3 (Gain: 1.0*-H2*G1*G2)
- Loop 3: 3 → 2 → 3 (Gain: 1.0*-H1*G1)

Delta Values

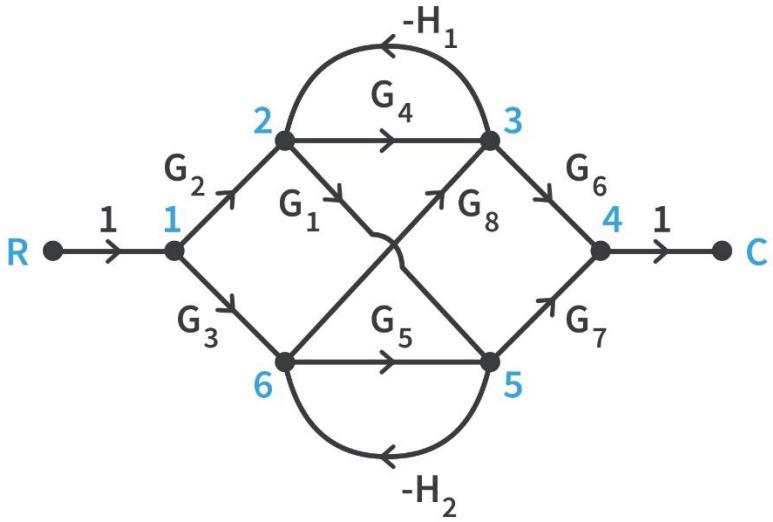
Δ = -1.0*-H1*G1 - 1.0*-H2*G1*G2 - 1.0*-H3*G2*G3 + 1.0

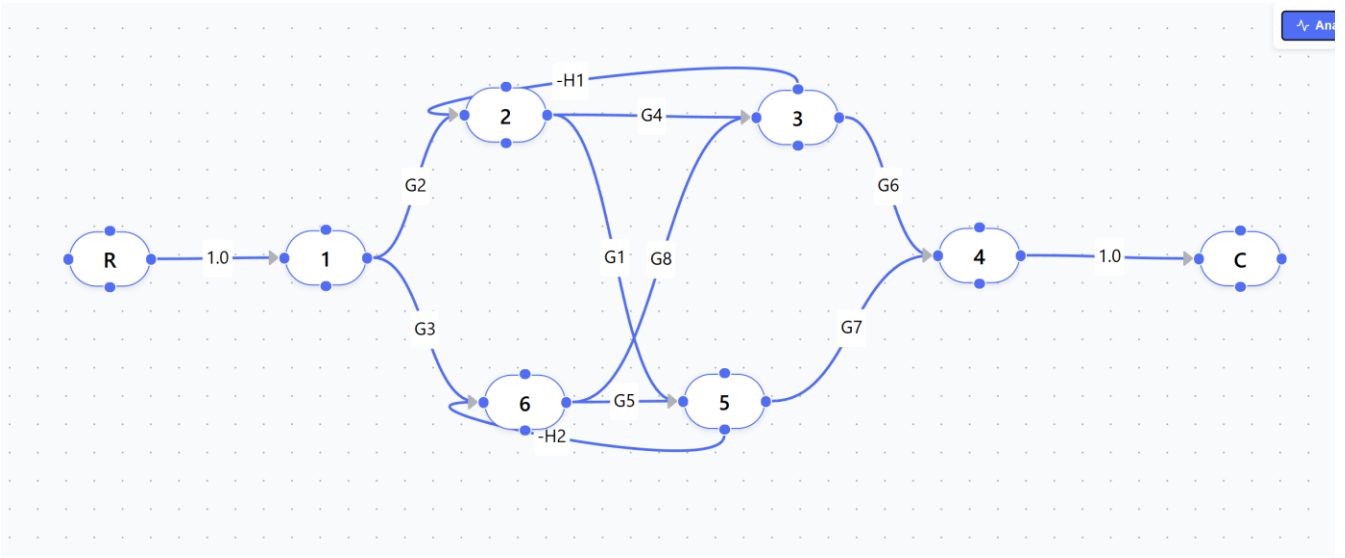
- Δ1 = 1.000

Transfer Function

-1.0*1.0**2*G1*G2*G3/(-H1*G1 - H2*G1*G2 - H3*G2*G3 - 1)

- **Problem 3:~**





Signal Flow Graph Analysis Results

Forward Paths

- Path 1: 1 → 2 → 3 → 4 → 5 → 8 (Gain: 1.0*1.0**2*G2*G4*G6)
- Path 2: 1 → 2 → 3 → 6 → 5 → 8 (Gain: 1.0*1.0**2*G1*G2*G7)
- Path 3: 1 → 2 → 3 → 6 → 7 → 4 → 5 → 8 (Gain: 1.0*-H2*1.0**2*G1*G2*G6*G8)
- Path 4: 1 → 2 → 7 → 6 → 5 → 8 (Gain: 1.0*1.0**2*G3*G5*G7)
- Path 5: 1 → 2 → 7 → 4 → 5 → 8 (Gain: 1.0*1.0**2*G3*G6*G8)
- Path 6: 1 → 2 → 7 → 4 → 3 → 6 → 5 → 8 (Gain: 1.0*-H1*1.0**2*G1*G3*G7*G8)

Individual Loops

- Loop 1: 7 → 6 → 7 (Gain: 1.0*-H2*G5)
- Loop 2: 7 → 4 → 3 → 6 → 7 (Gain: 1.0*-H1*-H2*G1*G8)
- Loop 3: 3 → 4 → 3 (Gain: 1.0*-H1*G4)

Delta Values

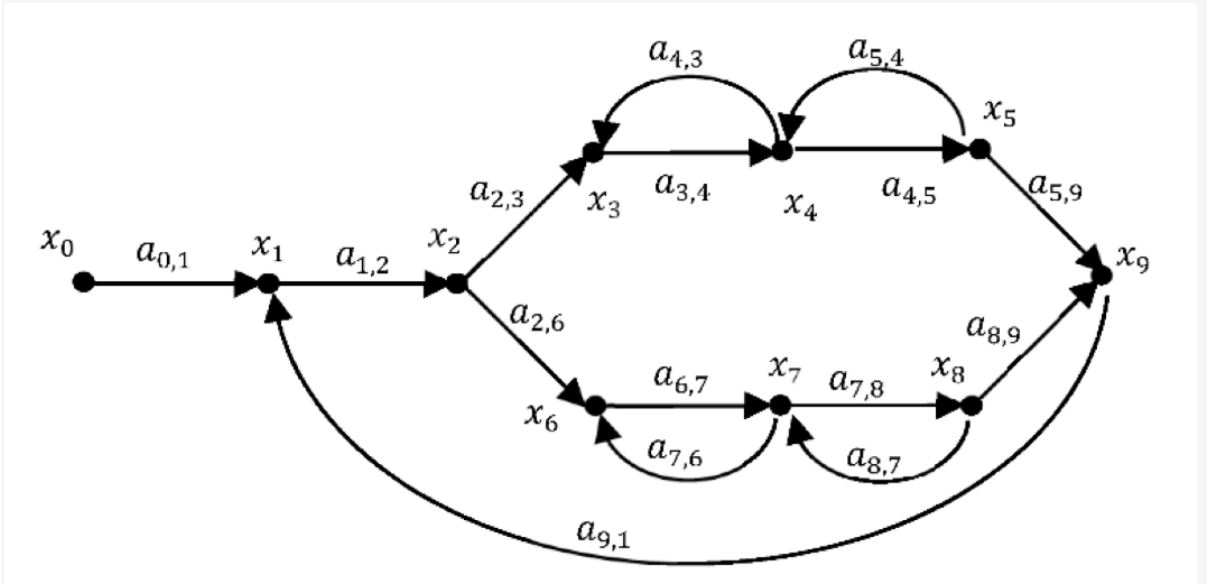
Δ = -1.0*-H1*-H2*G1*G8 + 1.0*-H1*-H2*G4*G5 - 1.0*-H1*G4 - 1.0*-H2*G5 + 1.0

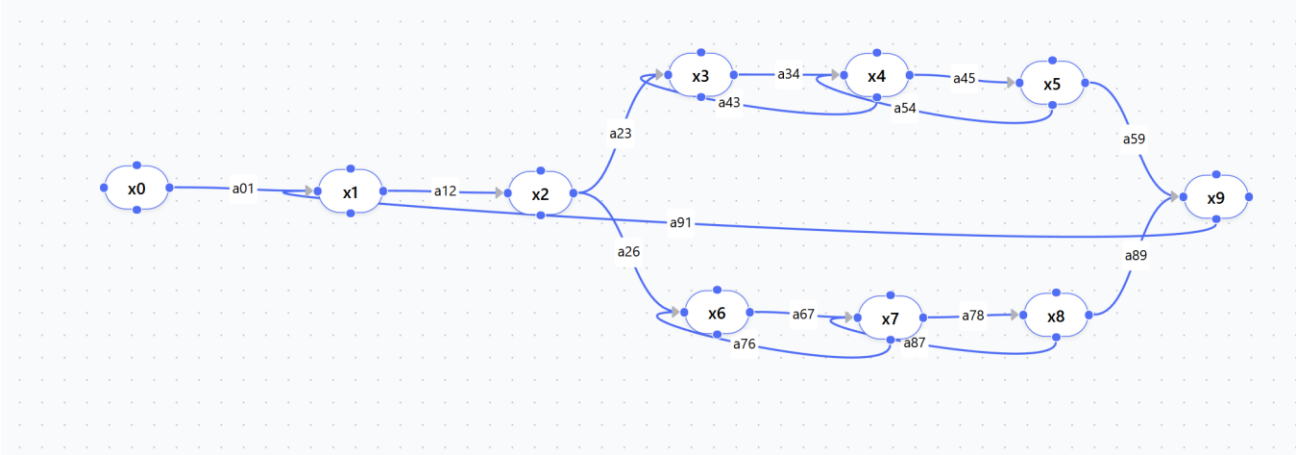
- Δ1 = -1.0*-H2*G5 + 1.0
- Δ2 = 1.000
- Δ3 = 1.000
- Δ4 = -1.0*-H1*G4 + 1.0
- Δ5 = 1.000
- Δ6 = 1.000

Transfer Function

1.0*1.0**2*(--H1*G1*G3*G7*G8 - -H2*G1*G2*G6*G8 - G1*G2*G7 + G2*G4*G6*(-H2*G5 - 1) + G3*G5*G7*(-H1*G4 - 1) - G3*G6*G8)/(-H1*-H2*G1*G8 - -H1*-H2*G4*G5 - H1*G4 - H2*G5 - 1)

- **Problem 4:~**





Signal Flow Graph Analysis Results

Forward Paths

- Path 1: 1 → 2 → 3 → 4 → 5 → 6 → 10 (Gain: 1.0*a01*a12*a23*a34*a45*a59)
- Path 2: 1 → 2 → 3 → 7 → 8 → 9 → 10 (Gain: 1.0*a01*a12*a26*a67*a78*a89)

Individual Loops

- Loop 1: 7 → 8 → 9 → 10 → 2 → 3 → 7 (Gain: 1.0*a12*a26*a67*a78*a89*a91)
- Loop 2: 7 → 8 → 7 (Gain: 1.0*a67*a76)
- Loop 3: 9 → 8 → 9 (Gain: 1.0*a78*a87)
- Loop 4: 4 → 5 → 6 → 10 → 2 → 3 → 4 (Gain: 1.0*a12*a23*a34*a45*a59*a91)
- Loop 5: 4 → 5 → 4 (Gain: 1.0*a34*a43)
- Loop 6: 5 → 6 → 5 (Gain: 1.0*a45*a54)

Delta Values

Δ = 1.0*a12*a23*a34*a45*a59*a67*a76*a91 + 1.0*a12*a23*a34*a45*a59*a78*a87*a91 - 1.0*a12*a23*a34*a45*a59*a91 + 1.0*a12*a26*a34*a43*a67*a78*a89*a91 + 1.0*a12*a26*a45*a54*a67*a78*a89*a91 - 1.0*a12*a26*a67*a78*a89*a91 + 1.0*a34*a43*a67*a76 + 1.0*a34*a43*a78*a87 - 1.0*a34*a43 + 1.0*a45*a54*a67*a76 + 1.0*a45*a54*a78*a87 - 1.0*a45*a54 - 1.0*a67*a76 - 1.0*a78*a87 + 1.0

- Δ1 = -1.0*a67*a76 - 1.0*a78*a87 + 1.0
- Δ2 = -1.0*a34*a43 - 1.0*a45*a54 + 1.0

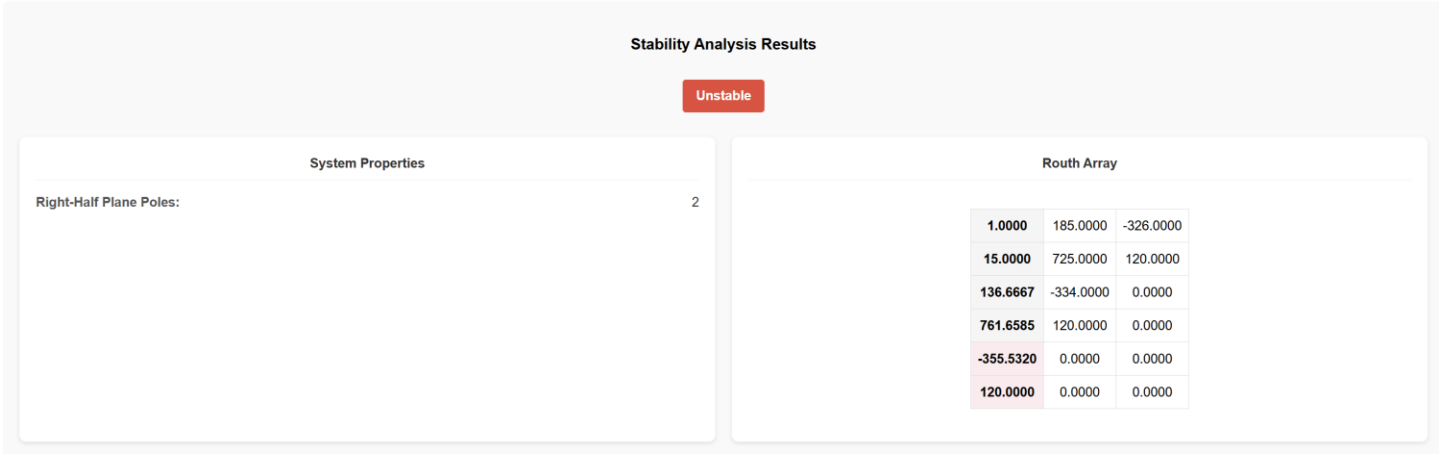
Transfer Function

1.0*a01*a12*(-a23*a34*a45*a59*(a67*a76 + a78*a87 - 1) - a26*a67*a78*a89*(a34*a43 + a45*a54 - 1))/(a12*a23*a34*a45*a59*a67*a76*a91 + a12*a23*a34*a45*a59*a78*a87*a91 - a12*a23*a34*a45*a59*a91 + a12*a26*a34*a43*a67*a78*a89*a91 + a12*a26*a45*a54*a67*a78*a89*a91 - a12*a26*a67*a78*a89*a91 + a34*a43*a67*a76 + a34*a43*a78*a87 - a34*a43 + a45*a54*a67*a76 + a45*a54*a78*a87 - a45*a54 - a67*a76 - a78*a87 + 1)

- **Problem 1 Routh:~**

$$\Delta_{CL}(s) = s^5 + 15s^4 + 185s^3 + 725s^2 - 326s + 120 = 0$$

s^5:	1	185	-326
s^4:	15	725	120
s^3:	136.7	-334	
s^2:	761.7	120	
s^1:	-355.5		
s^0:	120		

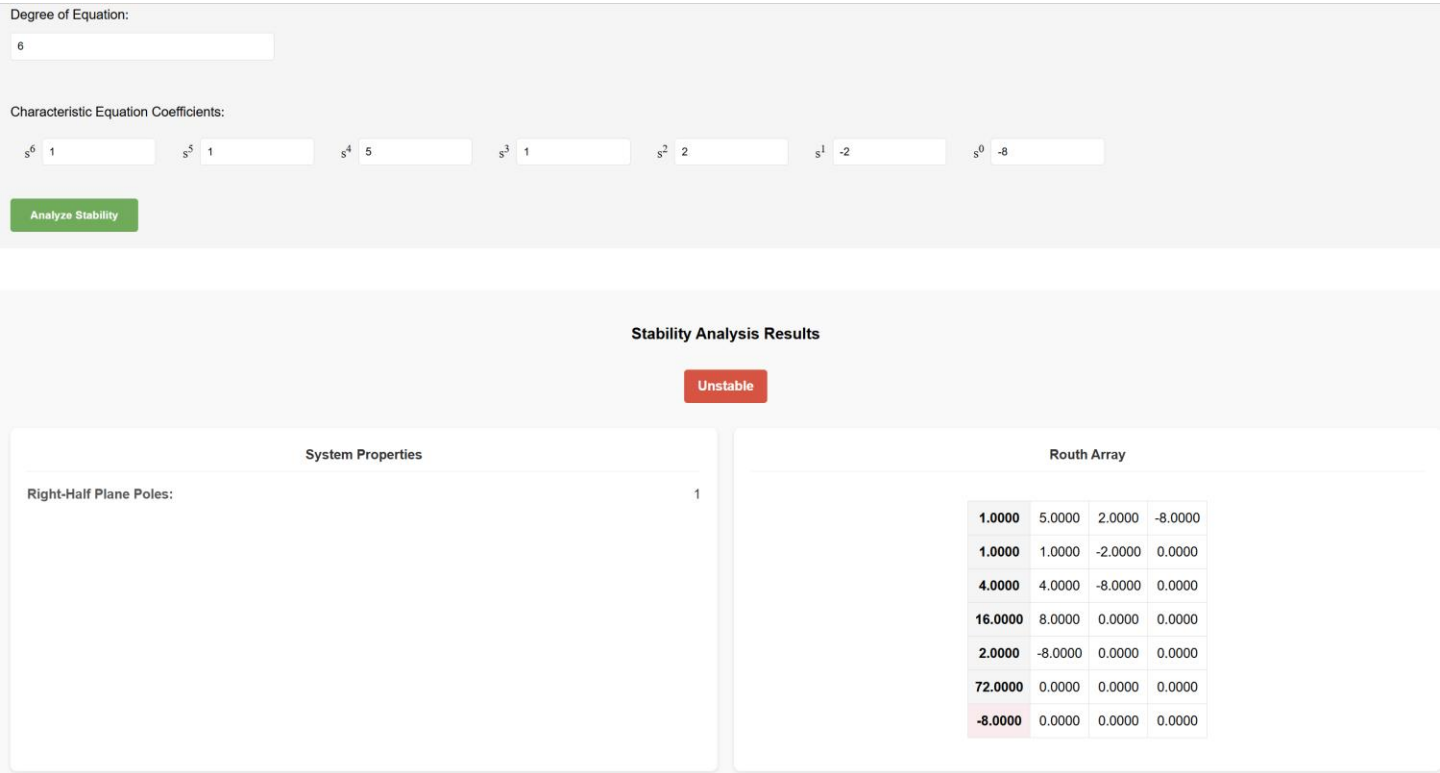


- **Problem 2 Routh:~**

Routh table

$$q(s) = s^6 + s^5 + 5s^4 + s^3 + 2s^2 - 2s - 8$$

s^6	1	5	2	−8
s^5	1	1	−2	
s^4	4	4	−8	← EP
s^3	0	0		← ROZ



- **Problem 3 Routh:~**

$$P(s) = 3s^7 + 9s^6 + 6s^5 + 4s^4 + 7s^3 + 8s^2 + 2s + 6$$

Degree of Equation:

7

Characteristic Equation Coefficients:

s^7 3 s^6 9 s^5 6 s^4 4 s^3 7 s^2 8 s^1 2 s^0 6

Analyze Stability

Stability Analysis Results

Unstable

System Properties

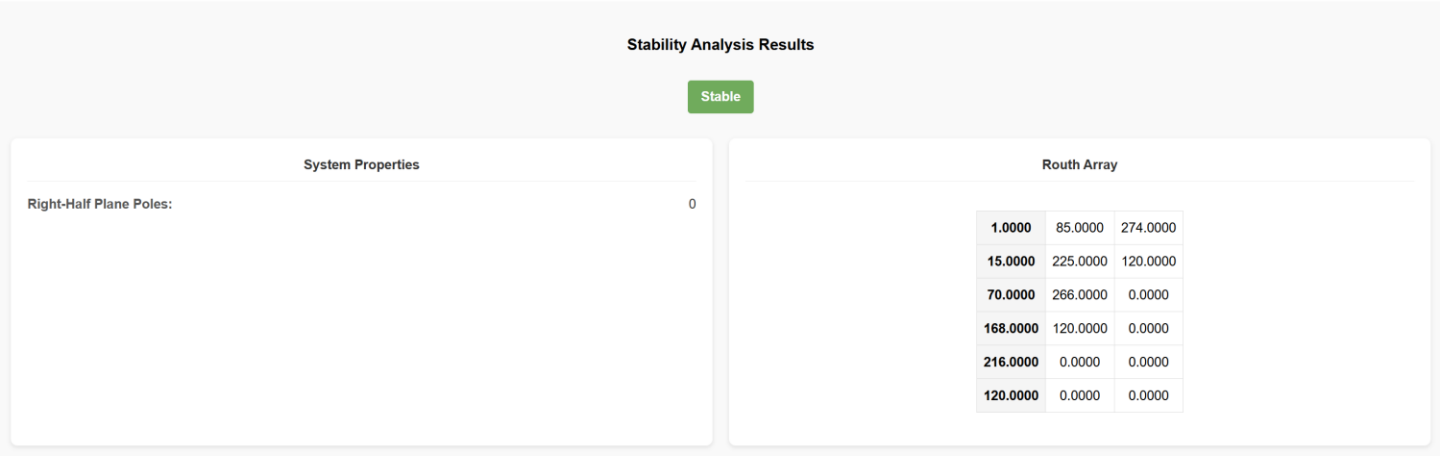
Right-Half Plane Poles: 4

Routh Array

3.0000	6.0000	7.0000	2.0000
9.0000	4.0000	8.0000	6.0000
4.6667	4.3333	0.0000	0.0000
-4.3571	8.0000	6.0000	0.0000
12.9016	6.4262	0.0000	0.0000
10.1703	6.0000	0.0000	0.0000
-1.1852	0.0000	0.0000	0.0000
6.0000	0.0000	0.0000	0.0000

- **Problem 4 Routh:~**

s^5	1	85	274
s^4	15	225	120
s^3	70	266	0
s^2	168	120	0
s^1	216	0	
s^0	120	0	



➔ Github directory:~

- https://github.com/braamost/SignalFlowGraph_and_RouthCriteria