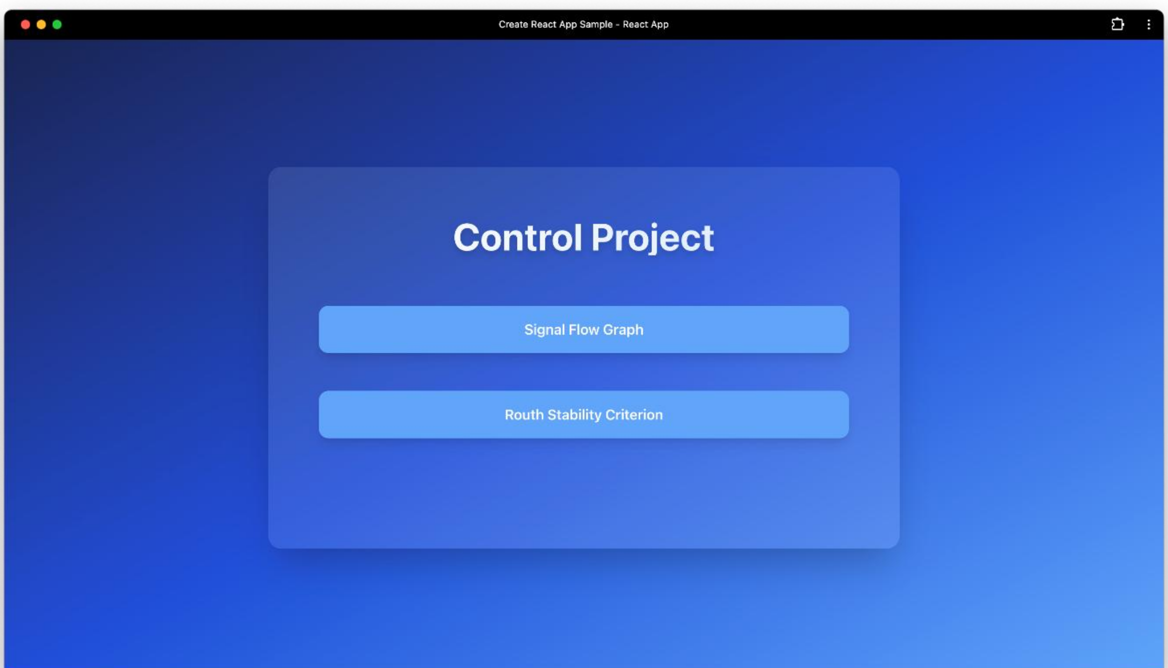
جامعة الإسكندرية
ALEXANDRIA
UNIVERSITY
كلية الهندسة

Computer & Systems Engineering Department

Control

Programming Assignment

Signal Flow Graphs & Routh Stability Criterion



Contributors:

	Name	ID
1	عمر طاهر حسين زيدان	22010966
2	عمر خالد حسين السيد	22010962
3	يوسف أحمد عبدالغني حسب الله	22011369
4	ياسين أسعد أحمد فريد محمود	22011349
5	عبدالرحمن السيد سعد سليمان نوار	22010869
6	علي الدين ماهرعبدالمنعم	22010934

GitHub Repository: SignalFlowGraph-RouthStabilityCriterion

1.Problem Statement

Control systems engineering requires powerful tools to analyze system behavior and stability. This project addresses two fundamental challenges:

1. **Signal Flow Analysis**: Visualizing and calculating transfer functions from complex system representations
2. **Stability Determination**: Assessing system stability through characteristic equation analysis

Traditional manual methods for these analyses are time-consuming and error-prone, especially for large systems. This tool automates both processes, providing accurate visualizations and calculations to support control system design and analysis.

2. Main Features

Signal Flow Graph Module

- **Interactive graph construction**: Add nodes and edges with specified gains
- **Automatic path detection**: Identifies all forward paths between input and output
- **Loop identification**: Finds all feedback loops in the system
- **Mason's Gain Formula implementation**: Computes the overall transfer function
- **Non-touching loop analysis**: Identifies groups of non-interacting loops
- **Visual optimization**: Aggregates parallel edges for cleaner representation

Routh-Hurwitz Stability Module

- **Characteristic equation input**: Accepts both coefficient arrays and symbolic expressions
- **Routh array construction**: Builds the complete stability matrix
- **Special case handling**: Manages zero rows and divisions by zero
- **Stability determination**: Counts sign changes in the first column
- **Root analysis**: Identifies unstable poles when the system is unstable
- **Numerical stability**: Uses small epsilon values (1e-10) for near-zero cases

Additional Options

- **JSON serialization**: All results can be exported for further processing
- **Symbolic computation**: Handles both numeric and symbolic system representations
- **Error handling**: Graceful degradation for invalid inputs
- **Automatic node detection**: Identifies input/output nodes when not specified

3. Data Structures

✅ Class 1: `SignalFlowAnalyzer`

1. Input Data

- `network`:
  Type: `dict` of `{node: List[Tuple[destination, gain]]}`
  Purpose: Adjacency list representing the signal flow graph.

2. Instance Variables / Internal Data Structures

Variable	Data Structure	Purpose
`forward_paths`	`List[List[str or int]]`	All forward paths from input to output.
`loops`	`List[List[str or int]]`	List of loops (cycles) in the graph.
`loop_gains`	`List[float]`	Gain values for each loop.
`path_gains`	`List[float]`	Gain values for each forward path.
`path_deltas`	`List[float]`	Path-specific deltas excluding touching loops.
`non_touching_loop_groups`	`List[List[List[int]]]`	Groups of non-touching loops (indexes of loops).
`visited`	`dict` of `{node: bool}`	Used for DFS traversal to track visited nodes.

3. Methods & Their Additional Data Structures

Method	Extra Structures Used	Purpose
`optimize_network()`	`defaultdict(int)`	To merge edges with the same destination.
`find_non_touching_loop_groups()`	`List[List[int]]`	Indices of non-touching loops.
`calculate_gain()`	—	Uses network list to compute gain.

Method	Extra Structures Used	Purpose
`calculate_total_system_delta()`	`int`, `float`, `List`	For delta computation using loop gains.
`dfs_find_paths()`	`List`, recursion	DFS pathfinding.

✅ Class 2: `RouthHurwitz`

1. Input Data

- `coefficients`:
  Type: `dict` of `{s-power (symbol or int): value (float/int)}`
  Purpose: Coefficients of the characteristic polynomial.

2. Instance Variables / Internal Data Structures

Variable	Data Structure	Purpose
`parsed_expression`	Usually a `sympy` expression	For symbolic math (if parsing expressions).
`symbols_dict`	`dict`	Possibly for storing variables from parsing.
`coefficients`	`dict`	Input polynomial coefficients.

3. Methods & Their Additional Data Structures

Method	Extra Structures Used	Purpose
`normalize_coeff_dict()`	`dict` → `dict`	Renames keys like `s` or `1` into `s**1`, `s**0`.
`initialize_routh_matrix()`	`List[List[float]]`	Initializes a 2D list (matrix) for Routh Array.
`compute_routh_matrix()`	`List[List[float]]` + special handling	Computes the full Routh Array with edge cases handled.

🔁 Summary Table

Class Name	Key Data Structures Used
`SignalFlowAnalyzer`	`dict`, `list`, `defaultdict`, `tuple`, `set` (implicitly via loop check), recursion
`RouthHurwitz`	`dict`, `list of lists` (matrix), optional symbolic expressions (e.g., via `sympy`)

4.Main Modules

Frontend (React)

- `SignalFlowGraphPage.jsx`: Graph UI and user input handling.
- `RouthTablePage.jsx`: Stability input and output rendering.
- `SolveSignalFlowGraphService.js`: Axios-based API call logic.

Backend (Flask)

- `app.py`: Flask server and API routing.
- `signal_flow_solver.py`: Mason's Gain Formula algorithm.
- `routh_solver.py`: Routh-Hurwitz stability table generation.

SignalFlowAnalyzer Class

- `optimize_network()`: Combines parallel edges
- `find_loops()`: DFS-based loop detection
- `find_forward_paths()`: Finds all input-to-output paths
- `calculate_transfer_function()`: Implements Mason's formula
- `to_dict()`: Serializes results for API response

RouthHurwitz Class

- `initialize_routh_matrix()`: Sets up initial coefficient rows
- `compute_routh_matrix()`: Completes the array with stability checks
- `determine_stability()`: Analyzes first column sign changes
- `find_roots()`: Solves characteristic equation
- `to_dict()`: Prepares stability analysis results

5. Algorithms Used

Signal Flow Graph

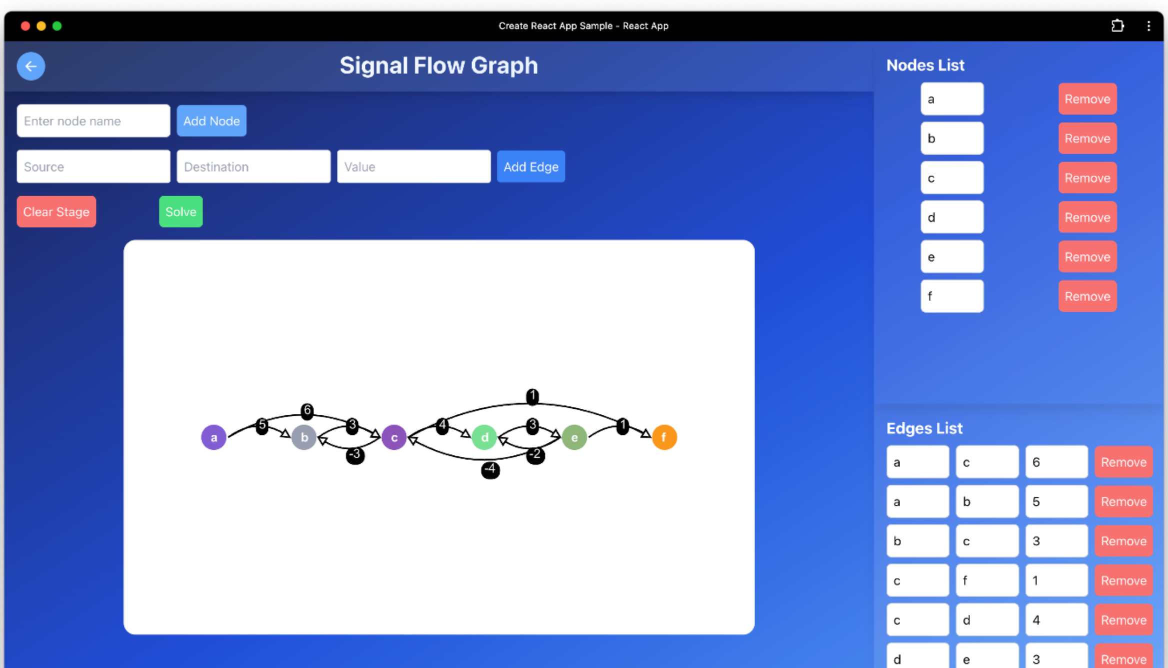
1. **Depth-First Search (DFS)**: For path and loop detection
2. **Mason's Gain Formula**:
3. `T = Σ(Pk·Δk)/Δ`
4. `Where:`
5. `Pk = kth forward path gain`
6. `Δ = 1 - ΣLn + ΣLmLn - ... (determinant)`
7. `Δk = Δ excluding loops touching Pk`
8. **Combinatorial Analysis**: For non-touching loop combinations

Routh-Hurwitz

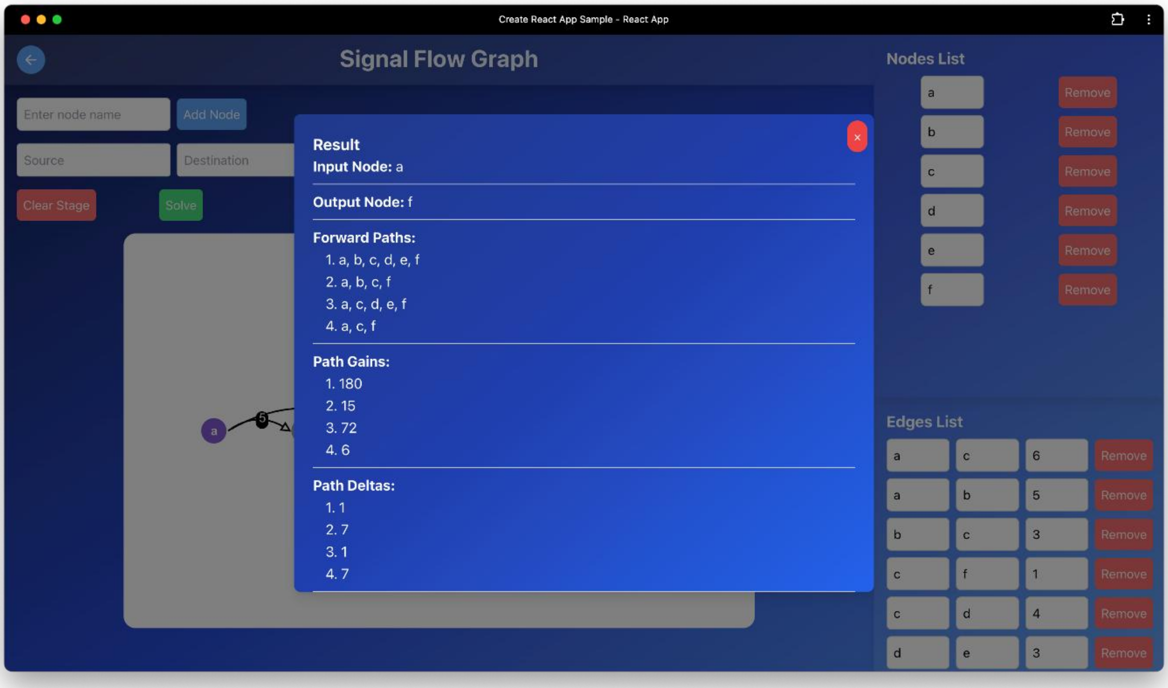
1. **Routh Array Construction**:
2. `For row i ≥ 2:`
3. `b(i,j) = [a(i-1,1)*a(i-2,j+1) - a(i-2,1)*a(i-1,j+1)]/a(i-1,1)`
4. **Auxiliary Polynomial Method**: For zero rows
5. **Root Finding**: Using SymPy's symbolic solver

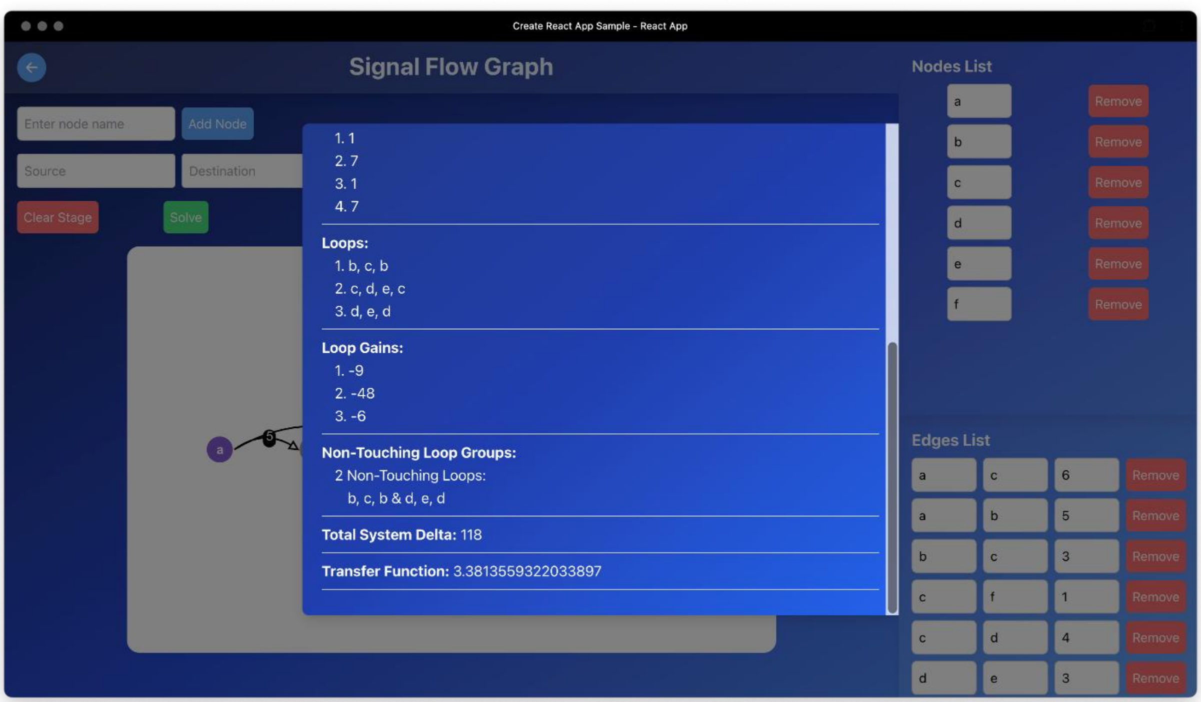
6. Sample Runs

Signal Flow Graph Example 1

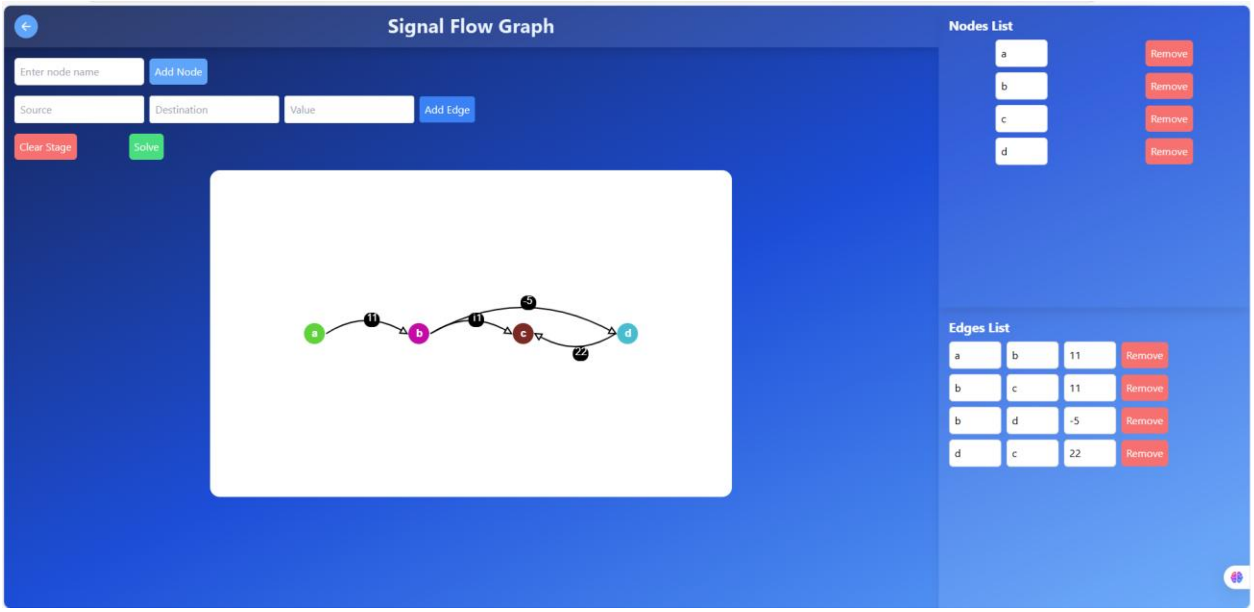


Output:


Create React App Sample - React App
Signal Flow Graph
Enter node name
Add Node
Source
Destination
Clear Stage
Solve
Result
Input Node: a
Output Node: f
Forward Paths:
1. a, b, c, d, e, f
2. a, b, c, f
3. a, c, d, e, f
4. a, c, f
Path Gains:
1. 180
2. 15
3. 72
4. 6
Path Deltas:
1. 1
2. 7
3. 1
4. 7
Nodes List
a Remove
b Remove
c Remove
d Remove
e Remove
f Remove
Edges List
a c 6 Remove
a b 5 Remove
b c 3 Remove
c f 1 Remove
c d 4 Remove
d e 3 Remove

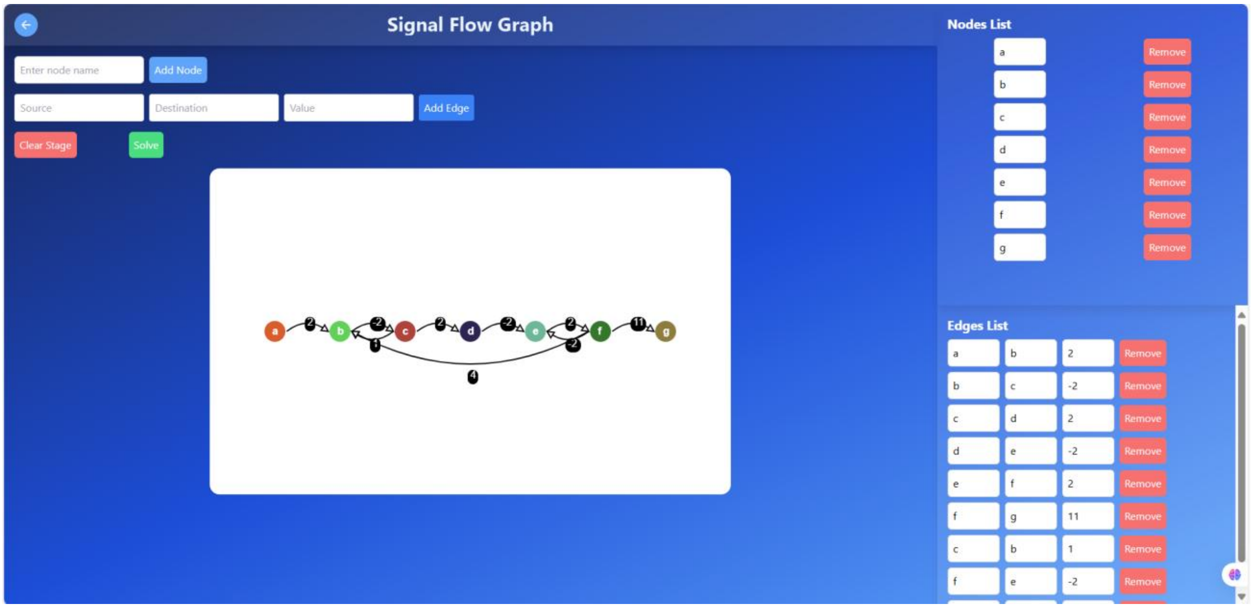

Create React App Sample - React App
Signal Flow Graph
Enter node name
Add Node
Source
Destination
Clear Stage
Solve
1. 1
2. 7
3. 1
4. 7
Loops:
1. b, c, b
2. c, d, e, c
3. d, e, d
Loop Gains:
1. -9
2. -48
3. -6
Non-Touching Loop Groups:
2 Non-Touching Loops:
b, c, b & d, e, d
Total System Delta: 118
Transfer Function: 3.3813559322033897
Nodes List
a Remove
b Remove
c Remove
d Remove
e Remove
f Remove
Edges List
a c 6 Remove
a b 5 Remove
b c 3 Remove
c f 1 Remove
c d 4 Remove
d e 3 Remove

Signal Flow Graph Example 2

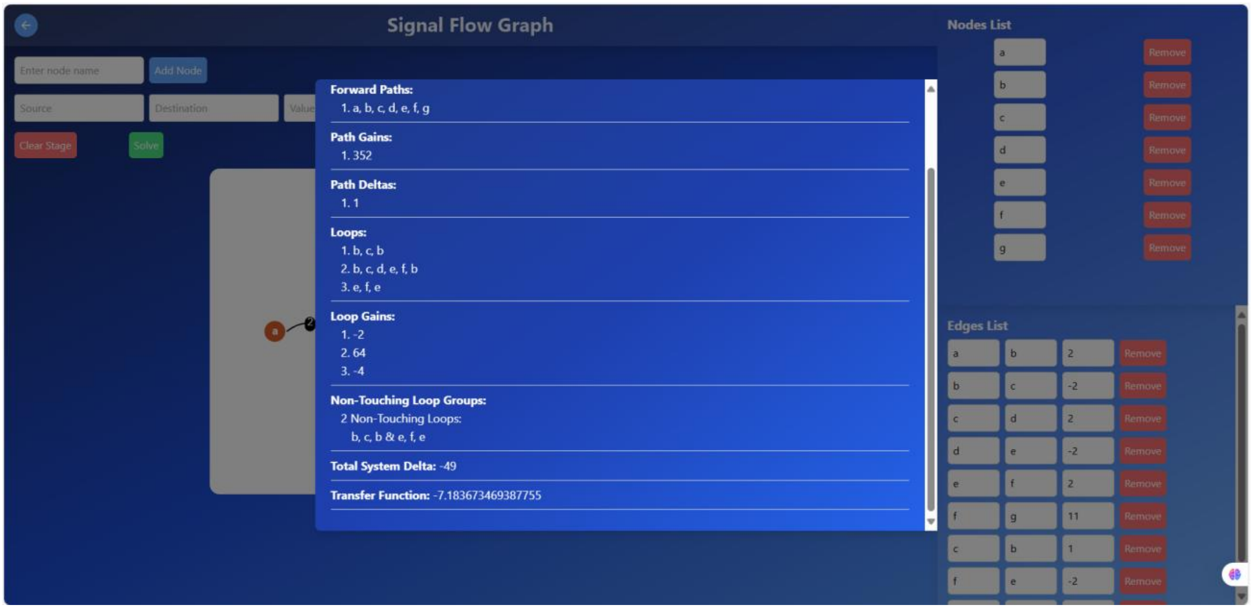


Output:

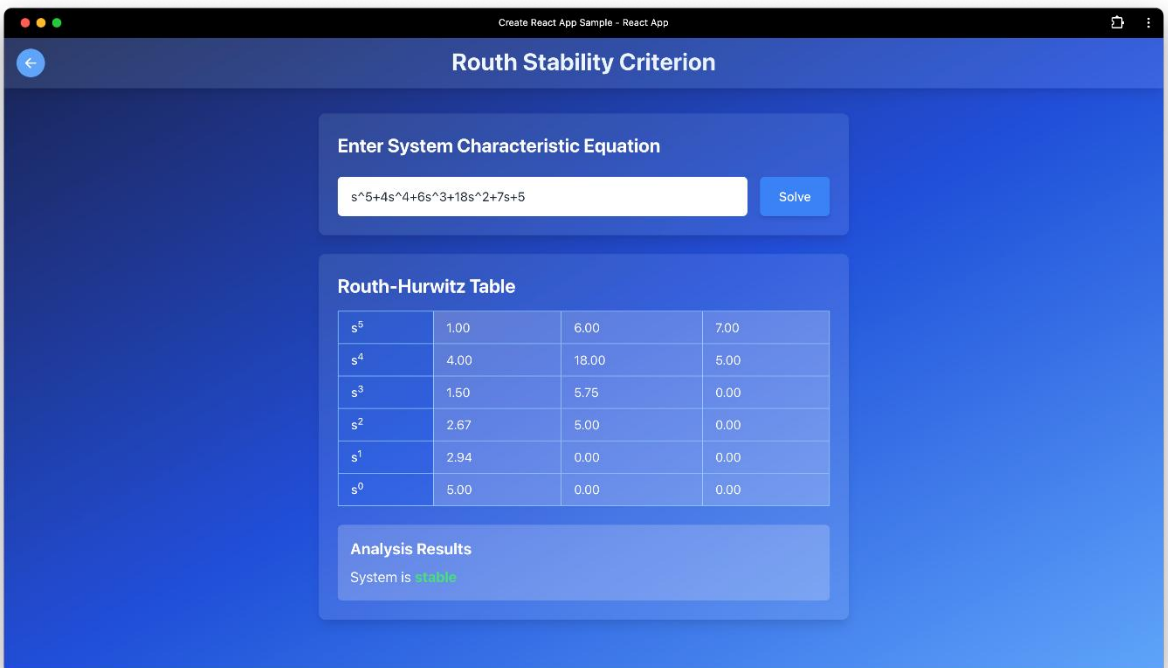
Signal Flow Graph Example 3

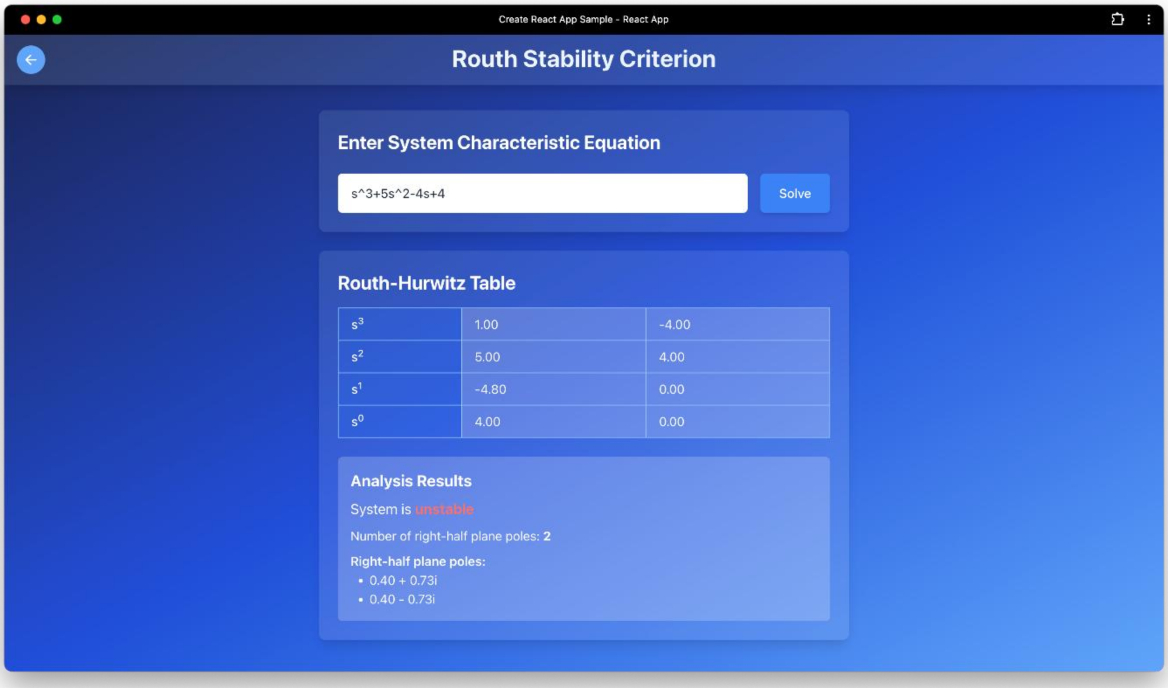


Output

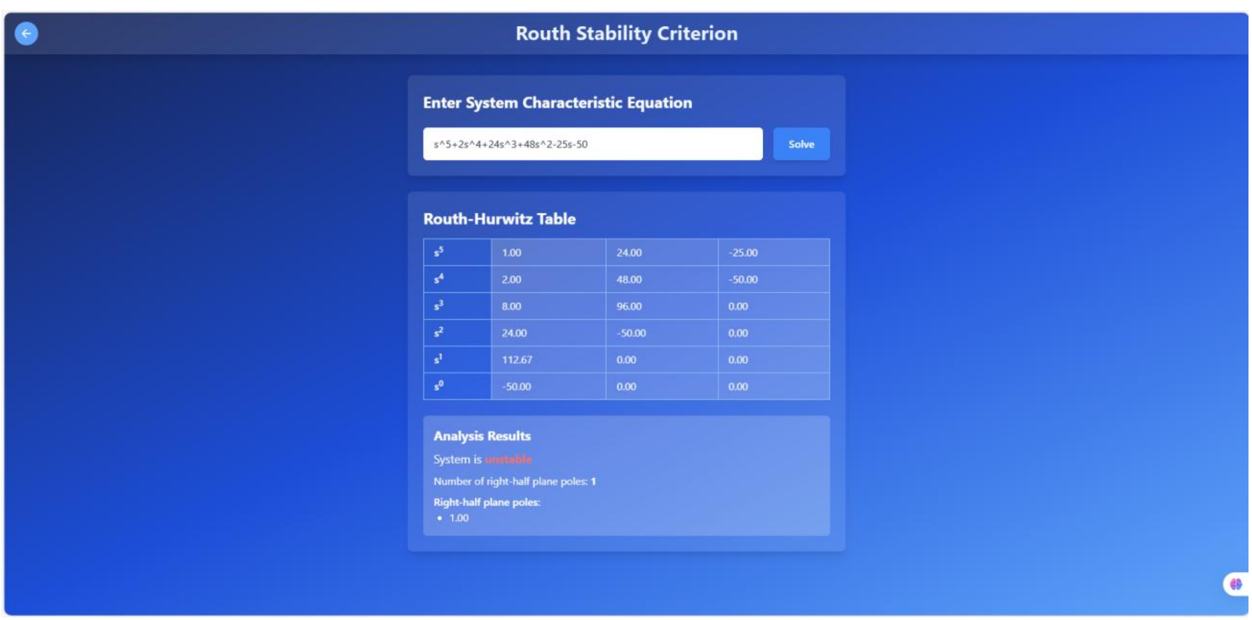


Routh-Hurwitz Example

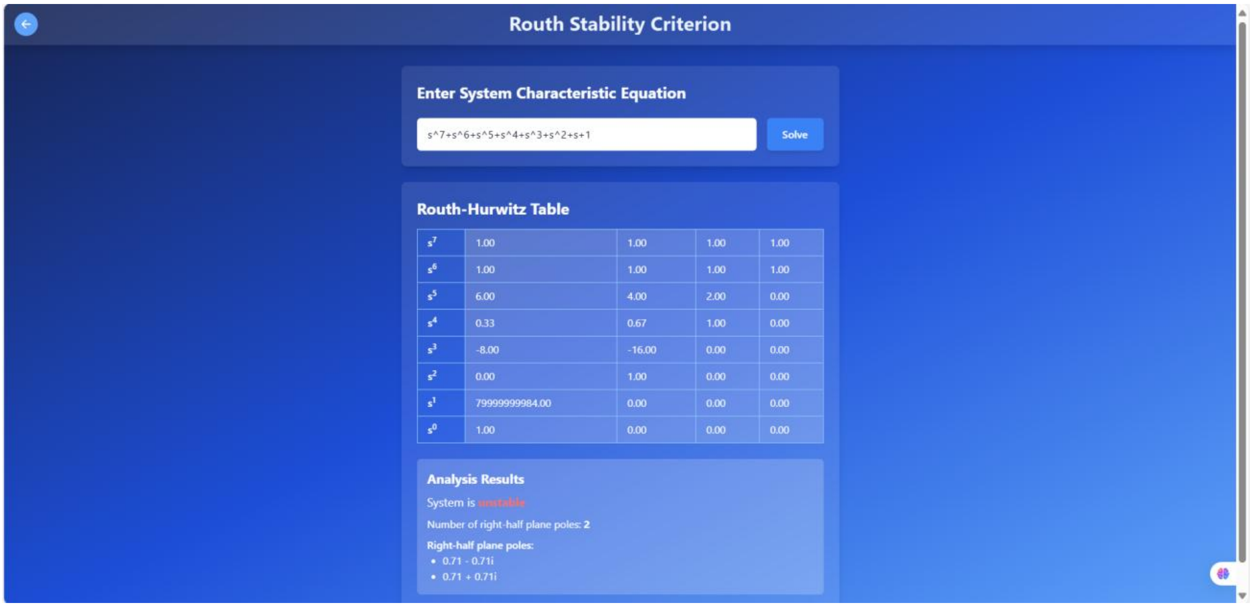




Zero row case



Zero row case & zero in first coloumn case



7. User Guide

🚀 Setup Instructions

Backend (Flask):

```
cd backend
pip install -r requirements.txt
python app.py
```

Frontend (React):

```
cd frontend
npm install
npm start
```

Signal Flow Graph

1. **Add Nodes**: Click "Add Node" and enter node name
2. **Create Edges**: Select source/destination nodes and enter gain value
3. **Analyze**: Click "Solve" to:
    - View all paths and loops

    - See calculated transfer function
4. **Modify**: Remove nodes/edges as needed

Routh-Hurwitz

1. **Input Options**:
    - Enter coefficients directly (e.g., [1,5,2,3])
    - Type symbolic equation (e.g., "s**3 + 5*s**2 + 2*s + 3")
2. **Results**:
    - Complete Routh array
    - Stability conclusion
    - Unstable poles (if any)

Tips

- For large systems, start with major components
- Check console for detailed intermediate results
- Use "Clear" between analyses to avoid interference

Conclusion

This tool provides a comprehensive solution for control system analysis, combining visual representation with rigorous mathematical computation. By automating complex calculations, it enables engineers and students to focus on system design and interpretation rather than manual computation.