

Control Signal-flow Graphs solver and stability testing

Github repo: [Control-Systems-Solver](#)

Osama Belal - 20010269

Michael Monir - 20011168

Joseph Shokry - 20010439

Ahmed Abdullah - 20010169

Omar Tammam - 20011016

Problem Statement

It was required to develop a program that solves signal flow graphs and implements Routh's criteria for stability testing which has the following features:

- Calculates the transfer function of a signal-flow graph.
- Lists forward paths, individual loops, and all combinations of n non-touching loops.
- Calculates the values of all deltas, which is used in calculating the transfer function.

Main Features

- Easy graph drawing
- Calculation of the transfer function of a signal-flow graph
- Visual representation of all forward paths, loops, and every combination of n non-touching loops.
- Full representation and calculation of the final Routh table for a given characteristic equation.
- Stability testing of a given system

Bonus Features

We didn't just stop there, our implementation is very different and unique. So we made **BONUS** features:

- A modern clean-looking **GUI**, that not only is functional but also is usable and easy to use.
- **Animation** of all forward paths that shows each forward path in a modern way, not just listing them.
- **Animation** of all loops.
- Robust Back-end.

Data structures

Backend

1. Signal Flow graphs

```
int size;

private List<pair>[] roads;

public List<List<Integer>> loops;

private List<Double> loopsWeights;

private boolean[][] loopsMatrix;

private Double[] loopMasks;

public List<List<String>> nonIntersectingLoopsEveryPath;

public List<String> paths;

public List<Double> deltasForEachPath;

public List<String> loopsForDeltaTotal;
```

- roads: list of pairs of all pairs of nodes connected stored as an adjacency list of directed edges.
- loops: store all different loops as a list of lists where each list is the list of nodes in the cycle
- loopsWeights: stores the product of edges in each loop according to the order of loops
- loopsMasks: array of masks of loops which can be taken together and store the product of all edges of all those loops. So, for example, the mask 10 (which is 1010 in binary) means that we take only loops with indices 3 and 1.
- nonIntersectingLoopsEveryPath: for every path gets a string of every set of non-Intersecting loops. But it is only used for outputting the result of the program.
- Path: another data structure used only to output the result to every path which returns a string showing nodes in that path.
- deltasforEachPath: keeps the delta calculation for each path and prints it with the results.

- loopsforDeltaTotal: stores loops of each delta but only stored for printing the result not in other calculations.

2. Routh Criteria

```
Private double[][] routhTable  
Private int rowSize  
Private int maxPower  
Private int rootCount  
Private static final double eps = 1e-10
```

- routhTable: a two-dimensional array of integers => stores the routh table of the given system and use its values for routh calculations to calculate the empty rows.
- rowSize: an integer variable => stores the dim y of the routh table and calculated using the following formula $(int) \text{Math.ceil}(\text{equation.size()} / 2.0)$
- maxPower: an integer variable => helper variable stores the max power of the given system equation to use it in routh calculations.
- rootCount: an integer variable => stores the number of roots in the right half plane.
- eps: a constant double variable => replaces the zeros while applying the routh algorithm.

Main modules

Backend:

1. Signal Flow graphs

```
private void getAllLoops()
{...}
private void dfsLoop(int x, double gain, List<Integer> stack)
{...}
boolean checkLoopNotCountedBefore(List<Integer> stack)
{...}
boolean checkTwoLoopsSame(List<Integer> stack, int index)
{...}
private void getIntersectingMatrices()
{...}
private void getDP()
{...}
public double calculateGraph()
{...}
double dfsPath(Integer x, double gain, List<Integer> stack, List<Boolean> taken)
{...}
double calculateDeltaForPath(List<Integer> stack)
{...}
double calculateDeltaTotal()
{...}
double calculateDelta(boolean[] validLoops)
{...}
boolean[] getAllValidLoopsForPaths(List<Integer> stack)
{...}
boolean checkLoopForPath(List<Integer> stack, int loopIndex)
{...}
private String getNonIntersectingLoops(int mask)
{...}
private String getPathsString(List<Integer> stack)
{...}
private class pair
{...}
```

First we go to the constructor which initializes all the edges and then use the calculate graph function which calculates the total delta of the graph and increments to the answer the number that is returned from each path. This is done using the DFS path function which gets all the path. To get all the non-intersecting

loops we first get all the loops using the getallloops function that uses the dfsloop and then we filter out the repeated ones using the checktwoloopssame function and then check for every pair of loops whether they are intersecting or not then we calculate the delta of every path using the masks that we obtained from the getDP function.

2. Routh Criteria

```
package com.accursed.controlsolver.routhCriteria;

import lombok.Setter;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

3 usages  ↗ josephShokry +1
@Setter
public class RouthSolver {
    private double[][] routhTable;
    private int rowSize;
    private int maxPower;
    private int rootCount;
    private static final double eps = 1e-10;

    1 usage  ↗ josephShokry
    public RouthSolver(List<Double> equation) { setEquation(equation); }

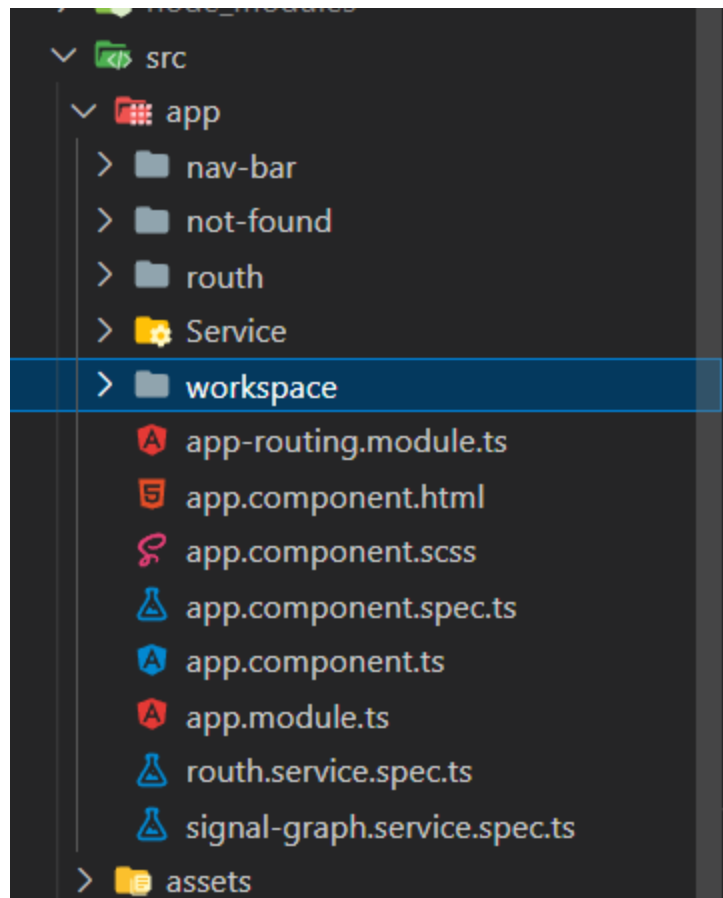
    1 usage  ↗ josephShokry +1
    private void setEquation(List<Double> equation) {...}
    1 usage  ↗ josephShokry +1
    public RouthDTO solve() {...}
    1 usage  ↗ josephShokry +1
    private boolean isStable(List<Double> routhOut) {...}
    1 usage  ↗ josephShokry
    private void zeroRow(int row) {...}
}
```

First in construction user sets the equation and the module runs the set equation Algorithm (all algorithms are explained in the algorithms section)

then the user calls the solve function to run the routh algorithm and the routh algorithm uses all the other functions as explained later.

Frontend

- **Nav-bar:**
 - Code for the circular menu
- **Not-Found:**
 - Handles errors in routing
- **Routh:**
 - Code for Routh table
- **Workspace:**
 - Code for the main signal-flow graph.



Algorithms used

Backend:

1. Signal Flow graphs

Dfs: we use dfs for two purposes:

- Dfs to find all paths:

We keep track of the stack of nodes that are visited until the current time, when finished visiting some node we remove it from the stack. If we visit a node that is already in the stack. This means that this is a cycle that we aren't interested in at the moment. We only care about the path. The condition that we stop at is when the last node is the last node of the graph. When this happens, we calculate the delta of this path and multiply it with the product of nodes, and return this value.

```
double dfsPath(Integer x, double gain, List<Integer> stack, List<Boolean> taken)
{
    stack.add(x);
    taken.set(x, true);
    if (x == size - 1) {
        paths.add(getPathsString(stack));
        return gain * calculateDeltaForPath(stack);
    }
    double ans = 0;
    for (pair p : roads[x])
    {
        if (taken.get(p.first)) continue;
        ans += dfsPath(p.first, gain * p.second, stack, taken);

        stack.remove(stack.size() - 1);
        taken.set(p.first, false);
    }
    return ans;
}
```


- Dfs to find all paths:

```
private void getAllLoops()
{
    List<Integer> emptyList = new ArrayList<>();

    for (int i = 0; i < size; i++)
        dfsLoop(i, 1, emptyList);

    getIntersectingMatrices();
    getDP();
}

private void dfsLoop(int x, double gain, List<Integer> stack)
{
    if (stack.size() > 0 && stack.get(0) == x)
    {
        List<Integer> temp = List.copyOf(stack);
        if (!checkLoopNotCountedBefore(stack))
        {
            loops.add(temp);
            loopsWeights.add(gain);
        }
        return;
    }

    for (int i = 1; i < stack.size(); i++)
        if (stack.get(i) == x) return;

    stack.add(x);

    for (pair p : roads[x])
        dfsLoop(p.first, gain * p.second, stack);

    stack.remove(stack.size() - 1);
}
```

Here we also use the stack thing where we put all the nodes in the current path of dfs in the stack. But there are two big differences in that dfs. First of all, we perform dfs from every node in the graph as we need to get all the loops starting from every node. The second difference is that this time we have two stopping conditions: when the last node is the first node. This means that we have a closed loop and now we can start calculating it. The second condition is that the last node is repeated but in the middle. This means that the loop is not correct. So we terminate without counting that loop.

```
private void getIntersectingMatrices()
{
    loopsMatrix = new boolean[loops.size()][loops.size()];
    for (int i = 0; i < loops.size(); i++)
    {
        HashSet<Integer> sett = new HashSet<>();
        for (Integer element : loops.get(i))
            sett.add(element);
        for (int j = 0; j < loops.size(); j++)
        {
            boolean flag = true;
            if (i == j)
            {
                this.loopsMatrix[i][i] = true;
                continue;
            }
            for (Integer element : loops.get(j))
                if (sett.contains(element))
                    flag = false;
            loopsMatrix[i][j] = flag;
        }
    }
}
```

This code checks whether every two loops are intersecting or not. This is done in $O(n*m*m)$ where n is the number of nodes and m is the number of loops. We also use a HashSet to reduce the complexity. We store the results in the Loopsmatrix to use the results in the future.

```
boolean checkLoopNotCountedBefore(List<Integer> stack)
{
    for (int i = 0; i < loops.size(); i++)
        if (checkTwoLoopsSame(stack, i))
            return true;
    return false;
}

boolean checkTwoLoopsSame(List<Integer> stack, int index)
{
    List<Integer> list1 = new ArrayList<>(stack);
    List<Integer> list2 = new ArrayList<>(loops.get(index));

    Collections.sort(list1);
    Collections.sort(list2);
    if (list1.size() != list2.size()) return false;
    for (int i = 0; i < list1.size(); i++)
        if (list1.get(i) != list2.get(i))
            return false;
    return true;
}
```

The code of finding loops has a big flow, that is it may count the same loop multiple times. So to solve that problem we just check the loop that wasn't counted before we insert it in the loop list.

```

private void calculateAllMasks()
{
    this.loopMasks = new Double[(1 << loops.size())];
    Arrays.fill(this.loopMasks, (double)0);
    loopMasks[0] = (double)1;

    for (int ii = 1; ii < (1 << loops.size()); ii++)
    {
        List<Integer> takenLoops = new ArrayList<>();
        for (int i = 0; i < loops.size(); i++)
        {
            int bit = (1 << i);
            if ((ii & bit) == 0) continue;
            takenLoops.add(i);
        }

        int lastone = takenLoops.get(takenLoops.size() - 1);
        boolean flag = true;
        for (int i : takenLoops)
            if (loopsMatrix[i][lastone] == false)
                flag = false;
        if (!flag) continue;

        this.loopMasks[ii] = this.loopMasks[ii - (1 << lastone)] *
this.loopsWeights.get(lastone);
    }
}

```

This function is used to calculate the masks of all the loops. We use bitmasks to store all the possible combinations of intersecting loops.

Also, we check before that for every mask if it contains two loops that are found to be intersecting (as we only count combinations of non-Intersecting loops). To do that quickly we use the matrix that we calculated before for that purpose(`loopsMatrix`). If the current mask was found to have two intersecting vertices, its value becomes zero, otherwise, it's the product of all edges of all the loops that the mask contains.

```

double calculateDeltaForPath(List<Integer> stack)
{...}
double calculateDeltaTotal()
{...}
double calculateDelta(boolean[] validLoops)
{...}

```

Finally, these three functions are the three functions used for calculating the delta either for the whole graph or for one certain path only. They use the same function which is to calculate the delta which only takes the valid loops. When we want to calculate the delta of the whole graph, then all the loops are valid. When a certain loop is invalid then any mask containing that loop isn't considered.

```

double calculateDelta(boolean[] validLoops)
{
    double ans = 1;

    List<String> currentLoops = new ArrayList<>();

    for (int mask = 1; mask < (1 << loops.size()); mask++)
    {
        boolean flag = true;

        double count = 0;

        for (int i = 0; i < loops.size(); i++)
        {
            int bit = (1 << i);

            if ((mask & bit) == 0) continue;

            if (validLoops[i] == false) flag = false;

            count++;
        }

        if (!flag) continue;

        currentLoops.add((getNonIntersectingLoops(mask) + ", " + this.loopMasks[mask]));

        if (count % 2 == 0)

```

```

        ans += this.loopMasks[mask];
    else
        ans -= this.loopMasks[mask];
    }

    nonIntersectingLoopsEveryPath.add(currentLoops);
    return ans;
}

```

That function loops through all the masks. When a certain mask is invalid we don't consider it. Also if it has an odd number of loops then we subtract it from the answer. Otherwise, we add it.

2. Routh Criteria

```

private void setEquation(List<Double> equation){
    maxPower = equation.size();
    rowSize = (int)Math.ceil(equation.size()/2.0);
    routhTable = new double[maxPower][rowSize];
    for(int i=0;i<equation.size();i++){
        if(i%2==0) routhTable[0][i/2] = equation.get(i);
        else routhTable[1][i/2] = equation.get(i);
    }
    rootCount = 0;
}

```

This algorithm is used to set the system equation and initiate the routh table (initiate the values of the first 2 rows of the table with the values of the coefficients of the system equation).

```

public RouthDTO solve(){
    for(int i=2;i<maxPower;i++){
        double val1 = routhTable[i-1][0];
        double val2 = routhTable[i-2][0];
        if(val1==0){
            val1 = eps;
            routhTable[i-1][0] = eps;
        }
        for(int j=0;j<rowSize-1;j++){
            routhTable[i][j] =
((val1*routhTable[i-2][j+1]-val2*routhTable[i-1][j+1])/val1);
        }
        zeroRow(i);
    }
    List<Double> out = new ArrayList<>(Collections.nCopies(maxPower, 0.0));
    for(int i=0;i<maxPower;i++){
        out.set(i,routhTable[i][0]);
    }
    System.out.println(out);
    boolean isStable = isStable(out);
    RouthDTO routhDTO = new RouthDTO();
    routhDTO.isStable = isStable;
    routhDTO.routhSolution = routhTable;
    routhDTO.rootCount = rootCount;
    return routhDTO;
}

```

This is the main function of the routh criteria used to fill the routh table the user set the transfer function of the system.

Applying the routh criterion in this function as it moves row by row and calculates the value of the cell using this formula:

$$routhTable[i][j] = ((val1 * routhTable[i - 2][j + 1] - val2 * routhTable[i - 1][j + 1])/val1)$$

Where

$$val1 = routhTable[i - 1][0]$$

$$val2 = routhTable[i - 2][0]$$

If val1 equals Zero then replace it with eps where eps is defined above in routh DataStructures, it is a very small +ve value to avoid division by zero in the following row operation of routh.

After calculating the row check if the whole row is zeros if so apply “zeroRow” Algorithm explained later, else continue applying the routh algorithm.

```
private boolean isStable(List<Double> routhOut){
    boolean stable = true;
    int rootCount = 0;
    for(int i = 0;i<routhOut.size()-1;i++){
        if(routhOut.get(i)* routhOut.get(i+1)<0) {
            stable = false;
            rootCount++;
        }
    }
    this.rootCount = rootCount;
    return stable;
}
```

This algorithm runs after finishing the routh algorithm (solve algorithm).

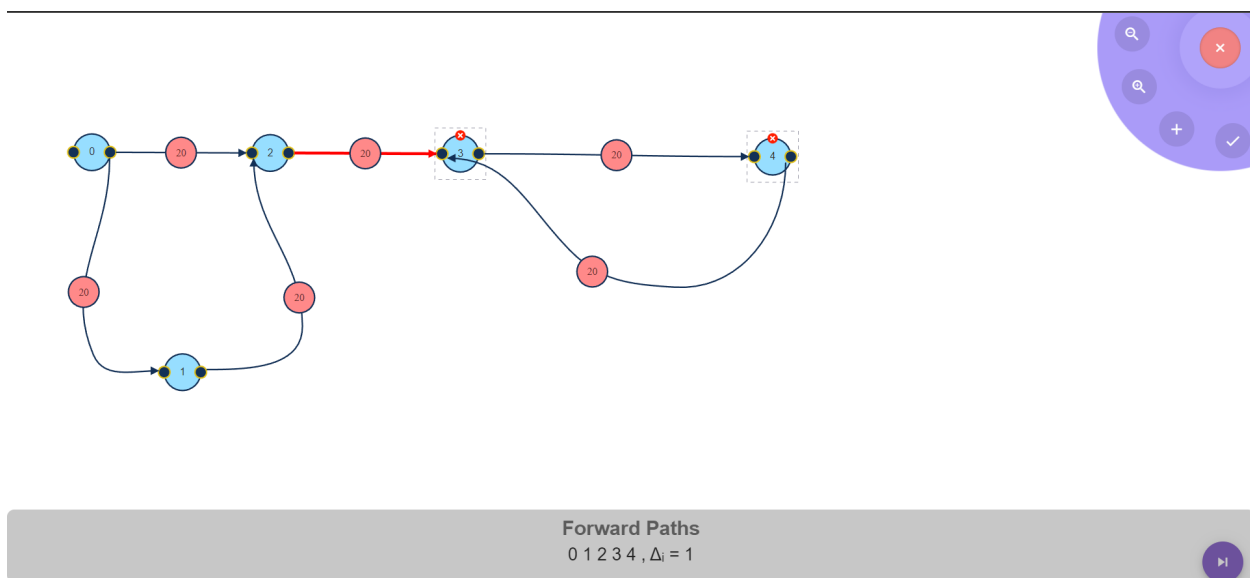
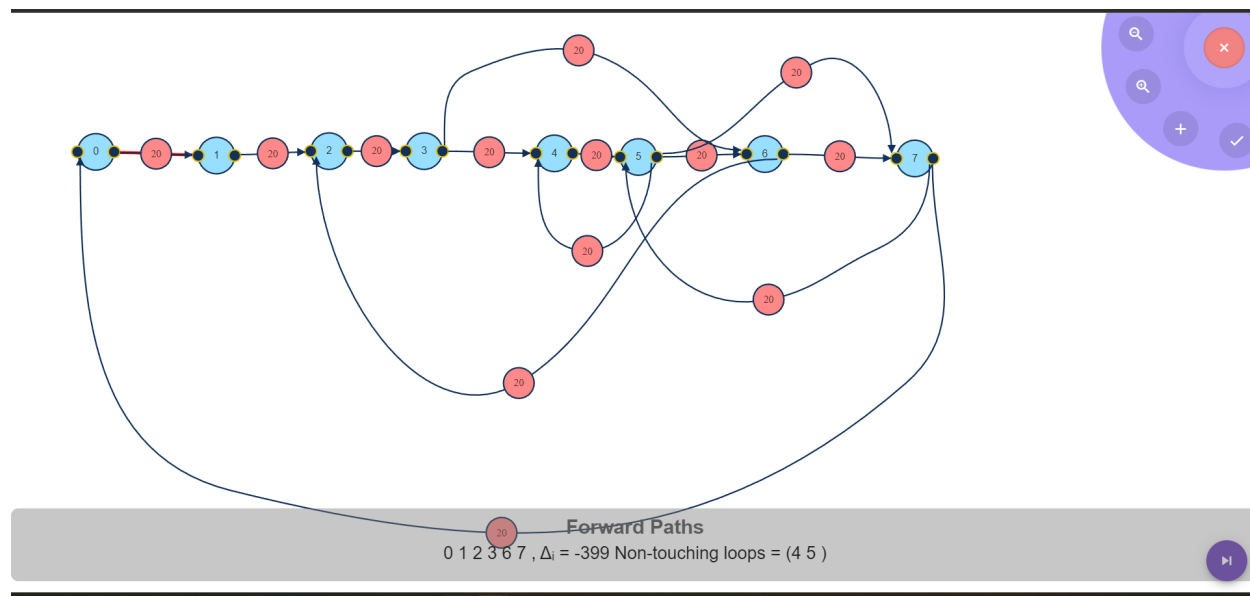
Checks if the generated routh table is stable by checking the first column if it finds any change of signs between 2 consecutive rows it states that the system is unstable and increments the count of the root count variable that indicates the number of roots in the right half plane.

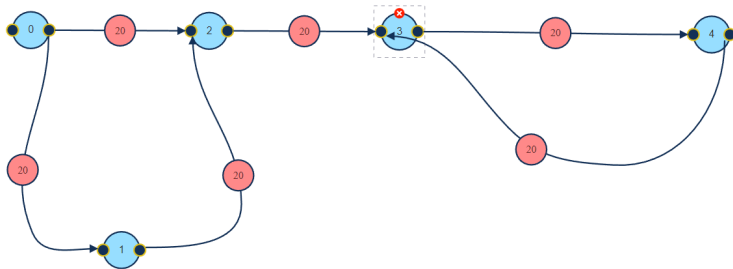

```
private void zeroRow(int row){
    int sum = 0;
    int pow = maxPower - row;
    for(int i=0;i<rowSize;i++)sum+=routhTable[row][i];
    if(sum==0){
        for(int i = 0;i<rowSize;i++){
            routhTable[row][i] = routhTable[row-1][i]*pow;
            pow-=2;
        }
    }
}
```

This algorithm is applied when applying routh algorithm and find a whole row with zeros.

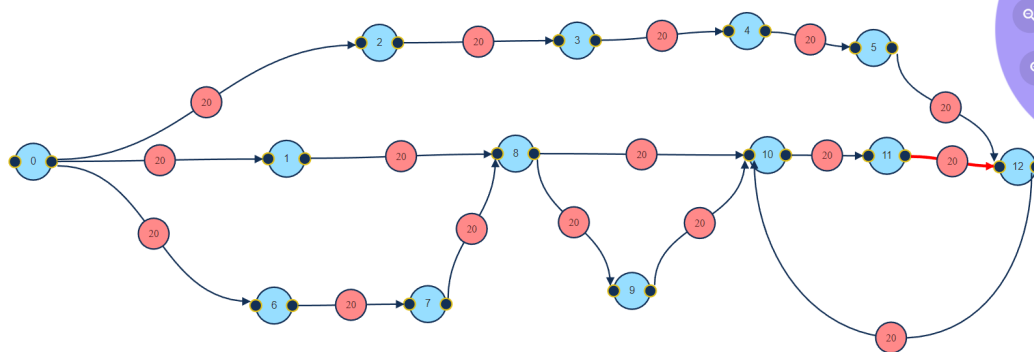
It simulates differentiating the previous row from the improper row by multiplying the power of the cell with the coefficient of the cell in the previous row and putting it instead of the zero (just applying this formula: $routhTable[row][i] = routhTable[row - 1][i] * pow$)

Sample runs

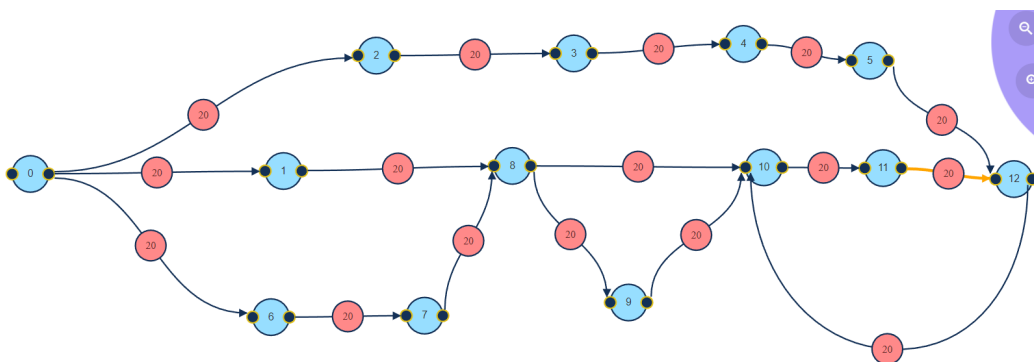




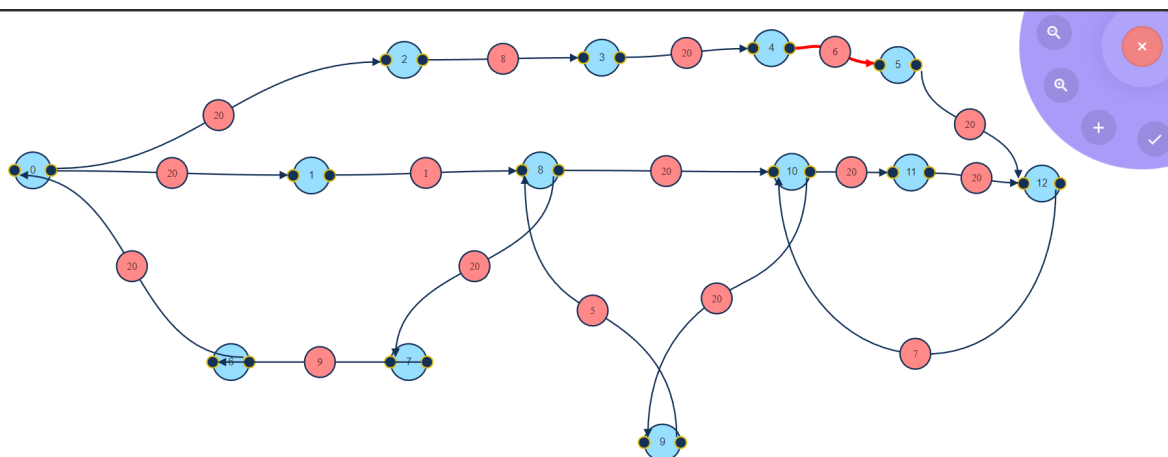
Result Transfer function
-421.05263157894734



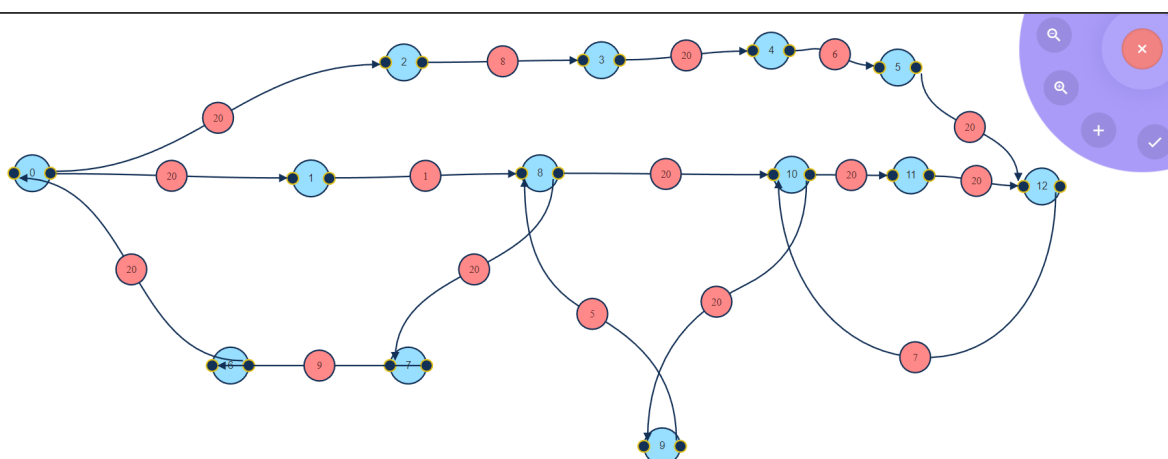
Forward Paths
0 1 8 10 11 12, $\Delta_i = 1$



Loops in graph
10 11 12 10


**Forward Paths**

0 2 3 4 5 12, $\Delta_i = -1999$ Non-touching loops = (8 10 9)

**Result Transfer function**

0.0007932538225958323



Enter Degree of the C/CS Equation 

The C/CS Equation Formula

Write down the coefficients of your equation

s^4 + s^3 + s^2 + s^1 + s^0


[Next](#)

Routh Table Result

Here's the final routh-hurwitz table of this equation

Stability Test

Check your transfer function stability

Enter Degree of the C/CS Equation 

The C/CS Equation Formula

Write down the coefficients of your equation

Routh Table Result


Here's the final routh-hurwitz table of this equation

s^4	1	10	0
s^3	22	2	0
s^2	9.909090909090908	0	0
s^1	2	0	0
s^0	2	0	0

[Previous](#) [Next](#)

Stability Test

Check your transfer function stability

Enter Degree of the C/CS Equation 

The C/CS Equation Formula

Write down the coefficients of your equation

Routh Table Result

Here's the final routh-hurwitz table of this equation

Stability Test

Check your transfer function stability

This System is Stable

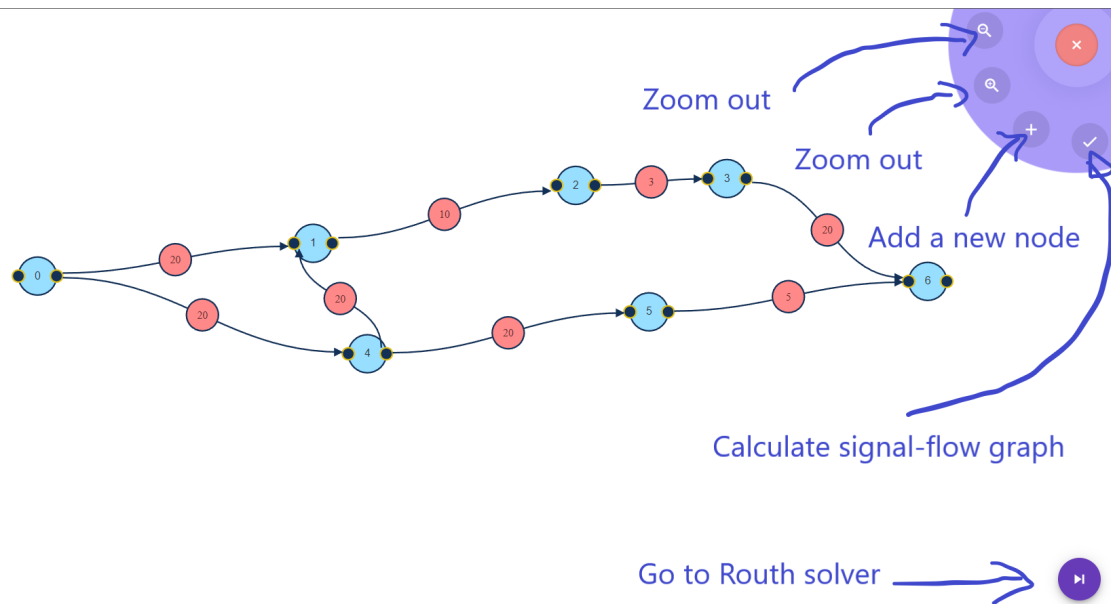
[Previous](#) [Next](#)

User guide

Important information for using the application

- All nodes must be labeled numeric values.
- All nodes should be in increasing order.
- Input/First node should be labeled 0.
- Output/Last node should be the largest value in the graph.
- Each node has 2 ports, the left one is used only for incoming edges, while the right one is used for outgoing edges.
- No self loops allowed.

Usability Information



Enter Degree of the C/CS Equation

The C/CS Equation Formula Write down the coefficients of your equation

s^3 + s^2 + s^1 + s^0

Routh Table Result Here's the final routh-hurwitz table of this equation

Stability Test Check your transfer function stability

Reset table

Lower degree

Increase degree

Calculate Routh table

Visual information

- Color coding for the animation is as follows:
 - **Red**: For visualizing forward paths.
 - **Orange**: For visualizing loops.
- A box is shown that shows the following:
 - The gain for each forward path.
 - n non-touching loops for each forward path.