SUMMER PROJECT REPORT

by,

S Vishal, Devendra Gupta, Sangeet Choubey

(B.tech, Part IV, Electrical Engineering)

Dr. Gopal Sharma, Supervisor

Department of Electrical Engineering,

IIT(BHU), Varanasi.

DEVELOPMENT OF AN ALGORITHM TO IMPLEMENT MASON'S GAIN FORMULA AND ITS IMPLEMENTATION IN C++

S Vishal, Devendra Gupta, Sangeet Choubey

Indian Institute of Technology (Banaras Hindu University)

Department of Electrical Engineering

Email: vishal.student.eee12@itbhu.ac.in, sangeet.choubey.eee12@itbhu.ac.in, devendra.gupta.eee12@itbhu.ac.in

Abstract- Mason's gain formula (MGF) is a method for finding the transfer function of a linear signal-flow graph (SFG). We present a robust algorithm for its implementation in any high level language by making use of its Object Oriented feature. This implementation can be a very handy tool for an engineer in doing stability analysis of any system and in other fields of its application. This will reduce a lot of tedious hand calculations.

I. INTRODUCTION

Mason's gain formula (MGF) is a method for finding the transfer function of a linear signal-flow graph (SFG). The formula was derived by Samuel Jefferson Mason, whom it is also named after. MGF is an alternate method to finding the transfer function algebraically by labeling each signal, writing down the equation for how that signal depends on other signals, and then solving the multiple equations for the output signal in terms of the input signal. MGF provides a step by step method to obtain the transfer function from a SFG. Often, MGF can be determined by inspection of the SFG. The method can easily handle SFGs with many variables and loops including loops with inner loops. MGF comes up often in the context of control systems and digital filters because control systems and digital filters are often represented by SFGs.[1]

The gain formula is as follows:

$G = Y_{out}/Y_{in} = \sum P_k \Delta_k / \Delta$

$\Delta = 1 - \sum L_i + \sum L_i L_j - \sum L_i L_j L_k + \ldots\ldots + (-1)^m \sum \ldots\ldots + \ldots\ldots$

Where,

- Δ = the determinant of the graph.
- Y_{in} = input-node variable
- Y_{out} = output-node variable
- G = complete gain between Y_{in} and Y_{out}
- P_k = path gain of the kth forward path between Y_{in} and Y_{out}
- L_i = loop gain of each closed loop in the system
- $L_i L_j$ = product of the loop gains of any two non-touching loops (no common nodes)

- $L_iL_jL_k$ = product of the loop gains of any three pairwise non touching loops
- Δ_k = the cofactor value of Δ for the kth forward path, with the loops touching the kth forward path removed.

For the computerized implementation of the above formula, matrix methods can come in handy, but they take up a lot of computer space for their implementation. We, on the other hand have used some efficient data structures and Abstract Data types to store, represent and implement directed graphs and transfer functions. We have also developed a complete library to not only store and represent but also perform operations on transfer functions in a high level language like C++.[2]

II. DEVELOPING ABSTRACT DATA TYPES (ADTs) TO REPRESENT TRANSFER FUNCTIONS AND DIGRAPHS IN C++

C++ currently does not have any library package to represent and perform operations on transfer functions or digraphs. But, being an Object Oriented language, it gives us the flexibility to define our own class with features in the form of functions to perform operations. We made use of this feature of C++ to build certain classes which helped us in representing transfer functions and performing various operations on them like addition, subtraction, multiplication etc. which were useful when implementing the MGF.

A. Transfer Functions

In engineering, a transfer function (also known as the system function or network function and, when plotted as a graph, transfer curve) is a mathematical representation for fit or to describe inputs and outputs of black box models.

A transfer function is of the form: $H(s) = X(s)/Y(s)$, where $X(s)$ and $Y(s)$ are both polynomial functions of s.

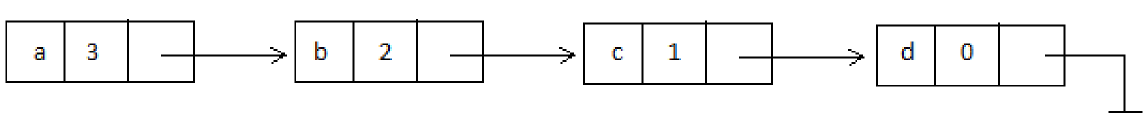
To make a class to represent a transfer function in C++, we need a class to represent polynomials.

Analysis of transfer functions is possible only when its numerator and denominator are in factorized form, it's only then that we can detect its *poles* and *zeros* and analyze our system. So, the polynomial class should not just be able to represent and perform operations on polynomials but it should do the same on *factorized polynomials*. So, we need a class to represent a factorized polynomial as well.

1) Polynomials

We used the linked list data structure (`std::list`) to represent polynomials [3],[4],[8]. For example, to represent the polynomial

$ax^3 + bx^2 + cx + d$, the following linked list representation is used:



The set of values and operations in polynomial ADT are given below:

Values:

```
/** A structure with two
   fields representing
   coefficient and exponents
   of a polynomial */
struct poly
{
     float coeff;
     float exp;
};
/** Class named polynomial
   with a linked list of
   poly type as the node as
   private member */
class polynomial
{
private:
     list<poly> P;
................
........
```

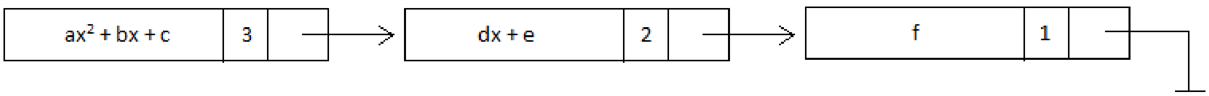
Operations:

```
class polynomial
{
...............
..............
public:
    void declare(void); //operation to declare a polynomial
    void display(void); //operation to display a polynomial
    void add(polynomial,polynomial); //operation to add two
polynomials and store result in third
    void sub(polynomial,polynomial); //operation to subtract two
polynomials and store result in third
    void mult(polynomial,polynomial); //operation to multiply two
polynomials and store result in third
    int siz(void); //operation to know size of the polynomial
    bool isequal(polynomial); //operation to check whether two
polynomials are equal
    void equalto(polynomial); //operation to assign one value to
another
    void inmult(polynomial);  //operation to multiply two
polynomials in-situ
    void unit(void); //operation to assign a unit polynomial
    bool isconst(); //operation to check whether a polynomial is a
constant
    bool isunit(); //operation to check whether a polynomial is a
unit polynomial
};
```

2) Factorized Polynomials

A linked list was again used to represent factorized polynomials [3],[4],[8]. For example, to represent the following:

$(ax^2 + bx + c)^3(dx + e)^2f$, the following linked list is used:



The set of values and operations in factorized polynomial ADT are given below:

<u>Values</u>:

```
/** A structure with
    first field being
    a polynomial defined
    earlier(representing
    factor) and second field
    representing power of
    that factor */
struct foly
{
    polynomial p;
    int exp;
};
/** class named folynomial
    with a linked list of
    foly as the private member */
class folynomial
{
private:
    list<foly> F;
.................
.................
```

<u>Operations:</u>

```
class folynomial
{
.................
.................
public:
    void declare(void); //operation to declare a factorized
polynomial (folynomial)
    void display(void); //operation to display a folynomial
    void add(folynomial,folynomial); //operation to add two
folynomials
    void sub(folynomial,folynomial); //operation to subtract two
folynomials
    polynomial expand(void); //operation to expand a factorized
polynomial
    void equalto(folynomial); //operation to assign value of one
to another
    void mult(folynomial,folynomial); //operation to multiply two
folynomials
    void inmult(folynomial); //operation to multiply two
folynomials in-situ
    void num(folynomial,folynomial); //operation to divide two
folynomials and display numerator
    void den(folynomial,folynomial); //operation to divide two
folynomials and display denominator
    void clean(void); //operation to empty out a folynomial
    void unit(void); //operation to assign a unit folynomial
    bool isunit(void); //operation to check if a folynomial is a
unit folynomial
```

After already having developed an ADT for polynomial, this becomes easy.

3) Transfer Functions

For making a class for transfer functions, we define two private variables viz. N and D for numerator and denominator of the type `folynomial` already defined above.

The set of values and operations in transfer function ADT are given below:

Values:

```
/** A class named transferfunction
    with two members of folynomial
    data type(defined earlier)
    representing the numerator and
    denominator of a transfer function*/

class transferfunction
{
private:
    folynomial N;
    folynomial D;
..................
..................
```

Operations:

```
class transferfunction
{
.................
.................
public:
    void declare(void); //operation to declare a transfer function
    void display(void); //operation to display a transfer function
    void add(transferfunction,transferfunction); //operation to
add two transfer functions
    void sub(transferfunction,transferfunction); //operation to
subtract two transfer functions
    void mult(transferfunction,transferfunction); //operation to
multiply two transfer functions
    void div(transferfunction,transferfunction); //operation to
divide two transfer functions
    void inmult(transferfunction); //operation to multiply two
transfer functions in-situ
    void inadd(transferfunction); //operation to add two transfer
functions in-situ
    void insub(transferfunction); //operation to subtract two
transfer functions in-situ
    void equalto(transferfunction); //operation to assign value of
one transfer function to other
    void cleanup(void); //operation to empty out a transfer
function
    void unit(void); //operation to initialize a unit transfer
function
};
```

B. Digraph

To represent a digraph we use an adjacency list and a weight matrix where an adjacency list is a 2-D vector (`std::vector`) and the weight matrix is a 1-D vector of structure data type where the structure stores 3 fields viz. `from` for source vertex, `to` for destination vertex and `weight` for the corresponding weight of the edge.[9]

The set of values and operations in transfer function ADT are given below:

<u>Values</u>:

```
/
class digraph
{
   private:
s        vector< vector<int> > D; //Adjacency list in form of a 2D
{array
        vector<sparse> W; //A sparse weight matrix
        vector<Gp> paths; //A matrix to represent all paths from a
source to a destination
}       vector<Gp> loops; //A matrix to represent all loops in a
/digraph
        transferfunction delta; //A variable of type
stransferfunction to store delta of a digraph
{
    transferfunction gain;
    vector<int> pth;
};
```

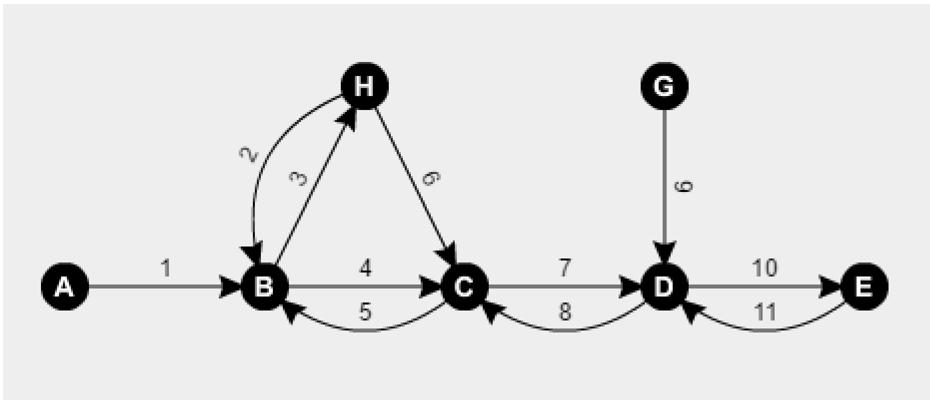
<u>Operations:</u>

```
class digraph
{
    ...............
    ...............
    public:
     void declare(void); //operation to declare a digraph
     void Paths(int,int); //operation to get all paths
from a source to a destination
     void DFS(int); //operation to perform Depth First
Search
     void getloops(void); //operation to get all loops and
corresponding gains
     void getdelta(void); //operation to get delta
     void gettf(void); //operation to get final transfer
function
};
```

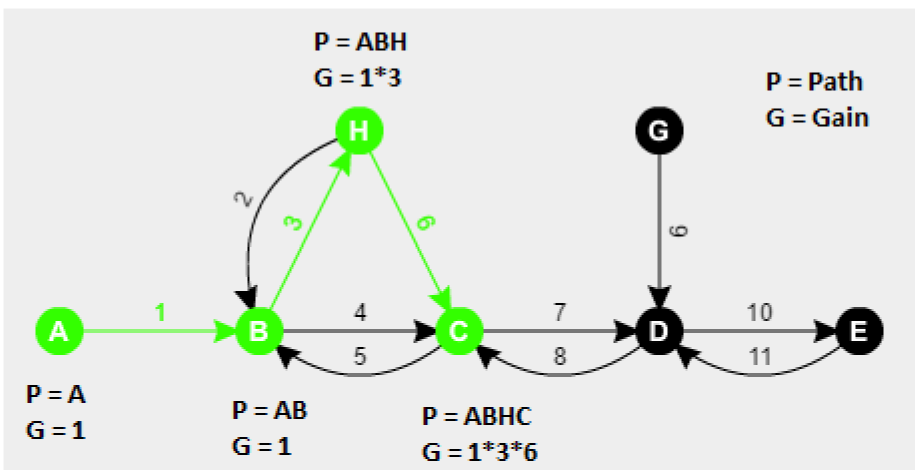
III. SIGNAL FLOW GRAPH ALGORITHMS

A. Finding paths and the corresponding gains from a source node to a destination node

Consider the following graph:



If we want to go from A to B, we visit each node one by one, starting from A and going towards B following the paths in the graph and looking for a potential path from A to B. Every time we reach a node, we color it (visited) and store the path gain and the path till that point locally.



Once we reach our destination, we append the path and the path gain at that point in another vector which will be used to store all paths and path gains. Also, whenever we reach a dead end, i.e. a node with no neighbors or the destination node, we color it black again (unvisited) and backtrack i.e. go back to the previous node. (In the code, this would correspond to the end of one recursive function call and the resumption of the calling function.) This is done so that all possible paths can be stored. [6],[7]

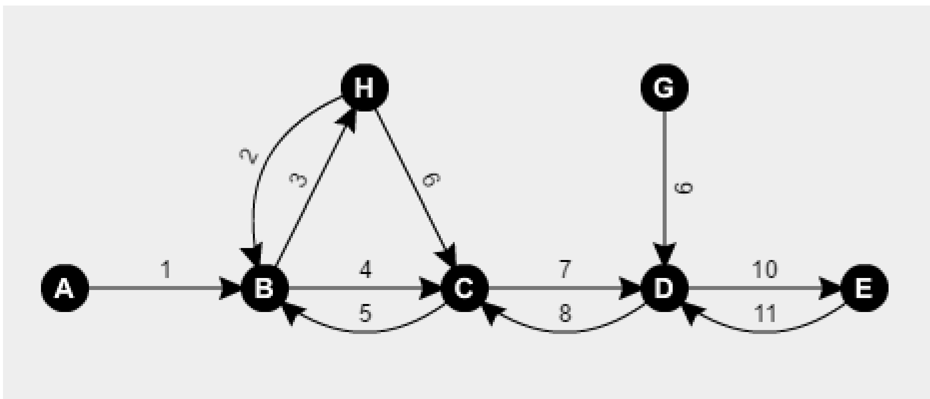
Pseudo Code

```
Declare a variable p for storing gain and a variable x for storing path
Procedure Paths(a,b):
    Store path gain and path traversed in local variables q and y
    if a=b
```

```
        A path is found. Append the path and the corresponding path gain to a
vector and return
    Mark node a visited
    for all neighbours w of a do
        if w is not marked visited
            p <- p*weight of path from a to weight
            Append w to x
        recursively call Paths(w,b)
        p <- q
        x <- y
    Mark a as unvisited again
```

B. Finding all loops and the corresponding loop gains in a Signal Flow Graph

Again consider the following graph:

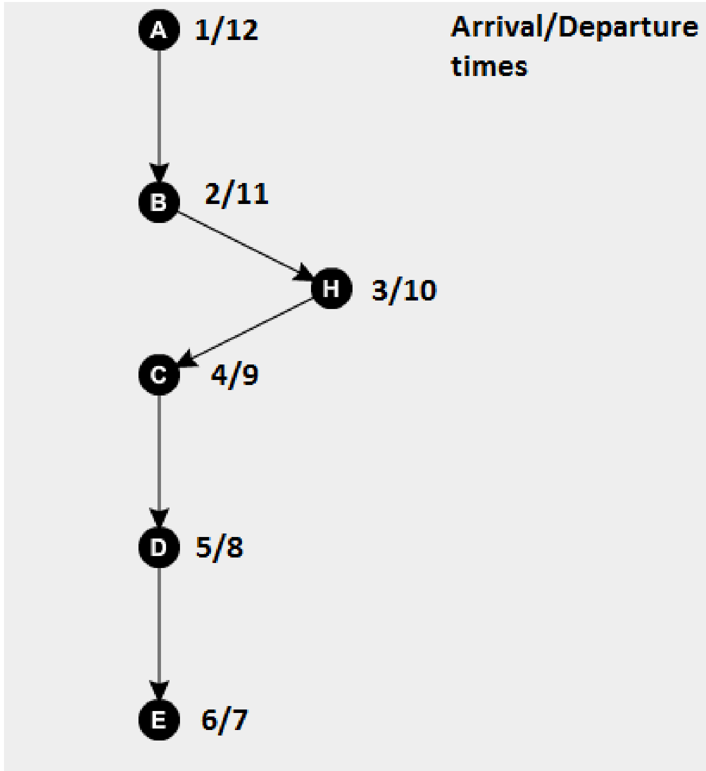


To detect a loop, we first perform a Depth First Search (DFS) on this graph and mark the *arrival times* and the *departure times* for every node in the DFS tree obtain on performing DFS.

Arrival time: The time at which the vertex was explored for the first time during a DFS. [6]

Departure time: The time at which we have explored all the neighbors of the vertex and we are ready to backtrack. [6]

After performing a DFS on the above graph and after labeling the arrival and departure times for each vertex, we obtain the following DFS tree:

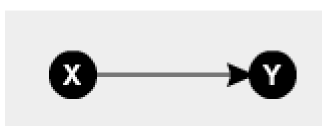


In any graph, consider the following two types of edges:

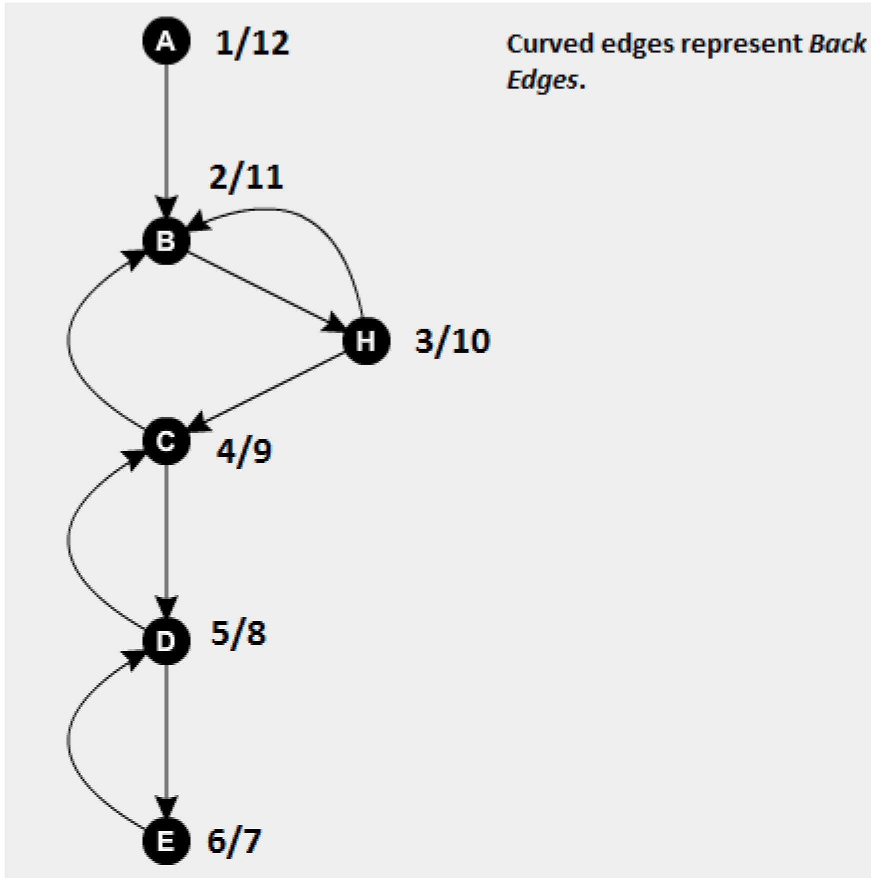
Tree Edge: Edges through which we reached (discovered) a vertex for the first time, i.e., edges leading to an unexplored vertex. These are the edges of the DFS tree.

Back Edge: Edges which lead back to a vertex already visited.

In any graph, the presence of a back edge indicates the presence of a loop. In other words, for every back edge, there is always a loop. So, our task reduces to detecting back edges.



If departure time of X > departure time of Y, then the edge XY is a back edge. [6]



Therefore, in the above graph, the edges HB, CB,DC and ED are back edges and hence correspond to loops in the graph.

Thus, to find loops in any graph:

- Perform a DFS on the graph labeling the arrival and departure times for each node.
- Detect all the back edges of the graph. A back edge is an edge (XY) for which the departure time of X is greater than the departure time of Y.
- For every back edge (XY), find all paths from Y to X and the corresponding path gains and then multiply the path gain of edge XY to that to get the loop gain.

C) Finding Δ *for a given graph*

In Mason's gain formula, the term delta is defined as [1]:

$$\Delta = 1 - \sum L_i + \sum L_i L_j - \sum L_i L_j L_k + \ldots\ldots + (-1)^m \sum \ldots\ldots + \ldots\ldots$$

Where,

- L_i = loop gain of each closed loop in the system
- $L_i L_j$ = product of the loop gains of any two non-touching loops (no common nodes)
- $L_i L_j L_k$ = product of the loop gains of any three pairwise non touching loops

We first define a few terms and notations before formally giving an algorithm to calculate **Δ.**

L_j = A structure (loop structure) with two fields, one to store the nodes in loop j in form of a vector and the other to store loop gain of loop j.

L_j(G) = Loop gain of the j^{th} loop.

L_j(V) = Vector containing nodes of loop j.

L_i(V) ∩ L_j(V) = Set of all common vertices in L_i(V) and L_j(V).

L_i(V) U **L_j(V) =** Union of the set of vertices from vector L_i(V) and L_j(V).

{L_1, L_2, L_3, …., L_n} = A set of loop structures (defined above).

L_i^L_j = L_k which is a loop structure with L_k(G) = L_i(G).L_j(G) and L_k(V) = L_i(V) U L_j(V).

Pseudo code:

```
Set global k <- 1
Set global D <- 1 - ∑ Lj

procedure delta()
    Set S <- {L1, L2, L3, …., Ln}
    Set d = 0
    for all loops structures j from 1 to n do
        d <- d + del(j,S)
    return d

procedure del(i,s)
    set k <- k + 1
    for all loop structures from Li+1 to Ln do
        if Li(V)∩ Lj(V) = Ø
        set D <- D + (-1)^k.Li(G).Lj(G)
        set q <- k
        set s2 <- {L1, L2, …., Li, …., Lj-1, Li^Lj, Lj+1, …., Ln}
        recursively call del(j,s2)
        set k <- q
```

IV. IMPLEMENTATION

The flow graph algorithms given in section III. can be easily combined to implement Mason's Gain Formula in any high level language. Our implementation in C++ is given in the Appendix.

The input and output formats for our implementations are given in this section.

1) Input Format

a) Polynomial

The representation of a polynomial in the form of linked list has already been given in section II.

The following steps are needed to be followed to enter a polynomial in our implementation:

- Enter the number of terms in the polynomial.
- For each term, enter the coefficients and exponents separately to form a linked list of the form already discussed.

b) Transfer Function

The representation of a transfer function has already been given in section II.

The following steps are needed to be followed to enter a transfer function in our implementation:

- Enter the numerator. The numerator and denominator are both factorized polynomials.
    - Enter the factorized polynomial in the numerator.
        - Enter the number of factors in the numerator.
        - Enter each factor which is a polynomial, one by one. In our implementation, a constant is treated like a separate factor and hence like a polynomial.
- Enter the denominator the same way as the numerator.

c) Digraph

The signal flow graph of any control system is a directed graph whose representation in C++ has already been given in section II.

The following steps are needed to be followed to enter a digraph in our implementation:

- Enter the number of vertices in the graph.
- Enter the number of neighbors for each vertex and while doing so, also specify the neighbors. This will create an adjacency list.
- Enter the number of edges in the graph.
- For every edge, specify the starting and ending vertex and the weight (transfer function) associated with that vertex. This will create a weight matrix which is a 1-D vector of structure data type where the structure stores 3 fields viz. from for source vertex, to for destination vertex and weight for the corresponding weight of the edge.

d) Starting and ending vertices

These are the nodes corresponding to the input and the output in our signal flow graph (cause and effect).

2) Output format

The output is the required transfer function of the signal flow graph entered.

V. FUTURE WORK

The major problem with our implementation is the inconvenient and inefficient way of inputting transfer functions. The input format given in section IV. becomes very time consuming when the system and hence the signal flow graph is very large. Although the answer is correct every time, the representation format of transfer functions through factorized polynomials causes the input format to become highly inefficient. So, we need to work on a better way to store transfer functions without compromising on the fact that both the numerator and the denominator of the transfer function should be in factorized form, which is a necessary constraint so that the poles and zeros can be clearly seen as much as possible and a comment on the stability of the system can be made.

We also need to work on building a Graphical User Interface which can be a helpful tool for making the best use of our work without the need to see the command prompt every time a transfer function is needed to be calculated.

VI. REFERENCES

[1] https://en.wikipedia.org/wiki/Mason%27s_gain_formula

[2] N. Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice Hall Inc., 1974, pp. 416-424.

[3] N. Karumanchi, Data Structures and Algorithms Made Easy, CareerMonk publications, 2014, pp.36-49, 227-250.

[4] Y. Kanetkar, Data Structures Through C, BPB publications, 2012, pp. 115-241.

[5] E. Balaguruswami, Object Oriented Programming with C++, Tata McGraw Hill publication, 1995, pp. 65-93.

[6] C. Papamanthou, "Depth First Search & Directed Acyclic Graphs", A Review for the Course Graph Algorithms, Department of Computer Science, University of Crete, 2004.

[7] http://algs4.cs.princeton.edu/42directed/

[8] http://www.cplusplus.com/reference/list/list/?kw=list

[9] http://www.cplusplus.com/reference/vector/vector/?kw=vector

APPENDIX

POLYNOMIAL HEADER FILE

```
#ifndef POLYNOMIAL_H
#define POLYNOMIAL_H
#include <iostream>
#include <list>
#include <cmath>
using namespace std;
struct poly
{
    float coeff;
    float exp;
};
class polynomial
{
private:
    list<poly> P;
public:
    void declare(void);
    void display(void);
    void add(polynomial,polynomial);
    void sub(polynomial,polynomial);
    void mult(polynomial,polynomial);
    int siz(void);
    bool isequal(polynomial);
    void equalto(polynomial);
    void inmult(polynomial);
    void unit(void);
    bool isconst();
    bool isunit();
    float value(void);
    void assignconst(float);
};
#endif // POLYNOMIAL_H
```

POLYNOMIAL CPP FILE

```
#include "polynomial.h"
#include <iostream>
#include <list>
#include <cmath>
using namespace std;

void polynomial:: declare()
{
   float c,e;
   poly f;
   int n;
```

```
    cout << "enter no. of terms in the polynomial" << endl;
    cin >> n;
    cout << "enter successive coefficients and exponents" << endl;
    for(int i=0; i<n; i++)
     {

             cin >> c;
             cin >> e;
             f.coeff = c;
             f.exp = e;
             P.push_back(f);

     }
}
void polynomial:: display()
{
     if(P.empty())
     {
         cout << "no polynomial entered." << endl;
         return;
     }
     list<poly>::iterator i;
     for(i=P.begin(); i!=P.end(); i++)
    {
        if(i->exp == 0)
        {
            if(i == P.begin())
            {
                 cout <<  i->coeff;
            }
            else
            {
                if(i->coeff < 0)
                 cout <<  i->coeff;
                else
                 cout << "+" << i->coeff;
            }
        }
        else if(i->exp == 1)
        {
            if(i->coeff != 1)
            {
            if(i == P.begin())
            {

            if(i->coeff!=-1)
             cout << i->coeff << "s";
             else
                 cout << "-s";
            }
            else
            {
```

```
                if(i->coeff > 0)
                 cout << "+" << i->coeff << "s";
                else if(i->coeff != -1)
                 cout << i->coeff << "s";
                 else
                     cout << "-s";
            }
        }
        else
        {
            if(i == P.begin())
             cout << "s";
            else
            {
                if(i->coeff > 0)
                 cout << "+" << "s";
                else
                 cout << "s";
            }
        }
        }
        else
        {
            if(i->coeff != 1)
            {
            if(i == P.begin())
            {
                if(i->coeff!=-1)
                 cout << i->coeff << "s^" << i->exp;
                 else
                     cout << "-s^" << i->exp;
            }
            else
            {
                if(i->coeff < 0)
                {

                if(i->coeff != -1)
                 cout << i->coeff << "s^" << i->exp;
                 else
                     cout << "-s^" << i->exp;
                }
                else
                 cout << "+" << i->coeff << "s^" << i->exp;
            }
        }
        else
        {
            if(i == P.begin())
            {
                 cout << "s^" << i->exp;
            }
```

```
           else
           {
               if(i->coeff < 0)
                cout << "s^" << i->exp;
               else
                cout << "+" << "s^" << i->exp;
           }
       }
   }

}
}
void polynomial:: add(polynomial p1, polynomial p2)
{
    if((p1.P).empty() && (p2.P).empty())
    {
        cout << "both polynomials empty" << endl;
        return;
    }
    else if((p1.P).empty())
    {
        P.assign((p2.P).begin(),(p2.P).end());
        return;
    }
    else if((p2.P).empty())
    {
        P.assign((p1.P).begin(),(p1.P).end());
        return;
    }
    list<poly>:: iterator i = (p1.P).begin();
    list<poly>:: iterator j = (p2.P).begin();
    while(i!=(p1.P).end() && j!=(p2.P).end())
    {
        if(i->exp > j->exp)
        {
            P.push_back((*i));
            ++i;
        }
        else if(j->exp > i->exp)
        {
            P.push_back((*j));
            ++j;
        }
        else
        {
            poly f;
            f.coeff = i->coeff + j->coeff;
            f.exp = i->exp;
            P.push_back(f);
            ++i;
            ++j;
        }
```

```
    }
    if(i!=(p1.P).end())
    {
        P.insert(P.end(),i,(p1.P).end());
    }
    else if(j!=(p2.P).end())
    {
      P.insert(P.end(),j,(p2.P).end());
    }
}
void polynomial:: sub(polynomial p1, polynomial p2)
{
    for(list<poly>:: iterator i=(p2.P).begin(); i!=(p2.P).end(); ++i)
    {
        i->coeff = (-1)*(i->coeff);
    }
    polynomial p3;
    p3.add(p1,p2);
    list<poly>:: iterator j=(p3.P).begin();
    int flag;
    while(j!=(p3.P).end())
    {
        flag = 0;
        if(j->coeff == 0)
        {
            j = (p3.P).erase(j);
            flag = 1;
        }
        if(flag!=1)
            j++;
    }
    P.assign((p3.P).begin(),(p3.P).end());
}
void polynomial:: mult(polynomial p1, polynomial p2)
{
    for(list<poly>:: iterator i=(p1.P).begin(); i!=(p1.P).end(); ++i)
    {
        for(list<poly>:: iterator j=(p2.P).begin(); j!=(p2.P).end(); ++j)
        {
            poly f;
            f.coeff = (i->coeff)*(j->coeff);
            f.exp = (i->exp) + (j->exp);
            int flag = 0;
            for(list<poly>:: iterator k=P.begin(); k!=P.end(); ++k)
            {
                if(k->exp == f.exp)
                {
                    k->coeff += f.coeff;
```

```
                        flag = 1;
                        break;
                    }
                }
                if(flag==0)
                    P.push_back(f);
                }
        }
        list<poly>:: iterator k=P.begin();
                while(k!=P.end())
                {
                    if(k->coeff == 0)
                        k = P.erase(k);
                    else
                        ++k;
                }
}
int polynomial:: siz()
{
    return (P.size());
}
bool polynomial:: isequal(polynomial p2)
{
   list<poly>:: iterator i = P.begin();
   list<poly>:: iterator j = (p2.P).begin();
   int flag = 0;
   while(i!=P.end() || j!=(p2.P).end())
   {
       if(((i)->coeff) != ((j)->coeff) || ((i)->exp) != ((j)->exp))
       {
           flag = 1;
           break;
       }
       i++;
       j++;
   }
   if(i!=P.end() || j!=(p2.P).end())
    return false;
   else if(flag == 1)
    return false;
   else
    return true;
}
void polynomial:: equalto(polynomial p)
{
    for(list<poly>:: iterator i = (p.P).begin(); i!=(p.P).end(); i++)
        P.push_back((*i));
}
void polynomial:: inmult(polynomial p)
{
```

```
    polynomial p1,p2;
    (p2.P).assign(P.begin(),P.end());
    p1.mult(p2,p);
    P.assign((p1.P).begin(),(p1.P).end());
}
void polynomial:: unit()
{
    poly p;
    p.coeff = 1;
    p.exp = 0;
    P.push_back(p);
}
bool polynomial:: isconst()
{
    if(P.size() == 1)
    {
        if((P.begin())->exp == 0)
            return true;
        else
            return false;
    }
    else
        return false;
}
bool polynomial:: isunit()
{
    if(P.size() == 1)
    {
        if((P.begin())->exp == 0 && (P.begin())->coeff == 1)
            return true;
        else
            return false;
    }
    else
        return false;
}
float polynomial:: value()
{
    return ((P.begin())->coeff);
}
void polynomial:: assignconst(float n)
{
    P.clear();
    poly p;
    p.coeff = n;
    p.exp = 0;
    P.push_back(p);
}
```

FACTORIZED POLYNOMIAL HEADER FILE

```cpp
#ifndef FOLYNOMIAL_H
#define FOLYNOMIAL_H
#include <iostream>
#include <list>
#include <cmath>
#include "polynomial.h"
using namespace std;
struct foly
{
    polynomial p;
    int exp;
};
class folynomial
{
private:
    list<foly> F;
    void sim(void);
public:
    void declare(void);
    void display(void);
    void add(folynomial,folynomial);
    void sub(folynomial,folynomial);
    polynomial expand(void);
    void equalto(folynomial);
    void mult(folynomial,folynomial);
    void inmult(folynomial);
    void num(folynomial,folynomial);
    void den(folynomial,folynomial);
    void clean(void);
    void unit(void);
    bool isunit(void);
};

#endif // FOLYNOMIAL_H
```

FACTORIZED POLYNOMIAL CPP FILE

```cpp
#include "folynomial.h"
#include <iostream>
#include <list>
#include <cmath>
#include "polynomial.h"
using namespace std;

void folynomial:: declare()
{
    int m,n;
    cout << "enter number of factors in the polynomial:" << endl;
    cin >> m;
    for(int i=1; i<=m; i++)
```

```
    {
        foly f;
        cout << "enter factor no." << i << ":" << endl;
        (f.p).declare();
        cout << "enter power of this factor:" << endl;
        cin >> n;
        f.exp = n;
        F.push_back(f);
    }
}
void folynomial:: display()
{
  if(!isunit())
    sim();
    if(F.empty())
    {
        cout << "no polynomial entered" << endl;
        return;
    }
    list<foly>:: iterator i;
    for(i=F.begin(); i!=F.end(); i++)
    {
        if((i->p).siz() != 1)
        {
            cout << "(";
            (i->p).display();
            if(i->exp == 1)
                cout << ")";
            else
                cout << ")^" << i->exp;
        }
        else
        {
            (i->p).display();
            if(i->exp != 1 && !(i->p).isconst())
                cout << "^" << i->exp;
        }
    }
}
void folynomial:: add(folynomial f1, folynomial f2)
{
   list<foly>:: iterator i = (f1.F).begin();
   list<foly>:: iterator j;
   int flag;
   while(i!=(f1.F).end())
   {
       j = (f2.F).begin();
       flag = 0;
       while(j!=(f2.F).end())
       {
           if((i->p).isequal((j->p)))
           {
```

```
                foly f;
                if(i->exp > j->exp)
                {
                    (f.p).equalto((j->p));
                    f.exp = j->exp;
                    F.push_back(f);

                    (i->exp) = (i->exp) - (j->exp);
                    j = (f2.F).erase(j);
                    goto label;
                }
                else if(i->exp < j->exp)
                {
                    (f.p).equalto((j->p));
                    f.exp = i->exp;
                    F.push_back(f);

                    (j->exp) = (j->exp) - (i->exp);
                    flag = 1;
                    i = (f1.F).erase(i);
                    goto label;
                }
                else
                {
                    (f.p).equalto((j->p));
                    f.exp = j->exp;
                    F.push_back(f);
                    i = (f1.F).erase(i);
                    j = (f2.F).erase(j);
                    flag = 1;
                    goto label;
                }
            }
            else
             ++j;
        }
        label:
            if(flag!=1)
            {
                ++i;
            }

    }
    for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
    {
        if((i->p).isconst())
        {
            for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
            {
                if((j->p).isconst())
                {
                    if((i->p).value()<(j->p).value())
```

```
                    {
                        (j->p).assignconst(((j->p).value())/((i->p).value()));
                        F.push_front((*i));
                        i = (f1.F).erase(i);
                        break;
                    }
                    else if((i->p).value()>(j->p).value())
                    {
                        (i->p).assignconst(((i->p).value())/((j->p).value()));
                        F.push_front((*j));
                        j = (f2.F).erase(j);
                        break;
                    }
                    else
                    {
                        F.push_front((*i));
                        i = (f1.F).erase(i);
                        j = (f2.F).erase(j);
                    }
                }
            }
        }
    }

    polynomial p1,p2,p3;
    p1 = f1.expand();
    p2 = f2.expand();
    p3.add(p1,p2);
    foly f;
    (f.p).equalto(p3);
    f.exp = 1;
    F.push_back(f);
}
void folynomial:: sub(folynomial f1, folynomial f2)
{
    list<foly>:: iterator i = (f1.F).begin();
    list<foly>:: iterator j;
    int flag;
    while(i!=(f1.F).end())
    {
        j = (f2.F).begin();
        flag = 0;
        while(j!=(f2.F).end())
        {
            if((i->p).isequal((j->p)))
            {
                foly f;
                if(i->exp > j->exp)
                {
                    (f.p).equalto((j->p));
```

```
                    f.exp = j->exp;
                    F.push_back(f);

                    (i->exp) = (i->exp) - (j->exp);
                    j = (f2.F).erase(j);
                    goto label;
                }
                else if(i->exp < j->exp)
                {
                    (f.p).equalto((j->p));
                    f.exp = i->exp;
                    F.push_back(f);

                    (j->exp) = (j->exp) - (i->exp);
                    flag = 1;
                    i = (f1.F).erase(i);
                    goto label;
                }
                else
                {
                    (f.p).equalto((j->p));
                    f.exp = j->exp;
                    F.push_back(f);
                    i = (f1.F).erase(i);
                    j = (f2.F).erase(j);
                    flag = 1;
                    goto label;
                }
            }
            else
              ++j;
        }
        label:
            if(flag!=1)
            {
                ++i;
            }

    }
    for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
    {
        if((i->p).isconst())
        {
            for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
            {
                if((j->p).isconst())
                {
                    if((i->p).value()<(j->p).value())
                    {
                        (j->p).assignconst(((j->p).value())/((i->p).value()));
                        F.push_front((*i));
```

```
                        i = (f1.F).erase(i);
                        break;
                    }
                    else if((i->p).value()>(j->p).value())
                    {
                        (i->p).assignconst(((i->p).value())/((j->p).value()));
                        F.push_front((*j));
                        j = (f2.F).erase(j);
                        break;
                    }
                    else
                    {
                        F.push_front((*i));
                        i = (f1.F).erase(i);
                        j = (f2.F).erase(j);
                    }
                }
            }
        }
    }

    polynomial p1,p2,p3;
    p1 = f1.expand();

    p2 = f2.expand();
    p3.sub(p1,p2);
    foly f;
    (f.p).equalto(p3);
    f.exp = 1;
    F.push_back(f);
}

polynomial folynomial:: expand()
{
    polynomial q;
    q.unit();
    for(list<foly>:: iterator i = F.begin(); i!=F.end(); ++i)
    {
        for(int j=1; j<=(i->exp); j++)
            q.inmult((i->p));
    }
    return q;
}
void folynomial:: equalto(folynomial f)
{
    F.clear();
    F.assign((f.F).begin(),(f.F).end());
}
void folynomial:: mult(folynomial f1, folynomial f2)
{
    if((f1.F).size()==1 && (f2.F).size()==1){
```

```
        if((((f1.F).begin())->p).isunit() && (((f2.F).begin())->p).isunit()){
            foly f;
            (f.p).unit();
            f.exp = 1;
            F.push_back(f);
            return;
        }
    }
    list<foly>:: iterator i = (f1.F).begin();
    list<foly>:: iterator j;
    int flag;
    while(i!=(f1.F).end())
    {
        flag = 0;
        j = (f2.F).begin();
        while(j!=(f2.F).end())
        {
            if((i->p).isequal(j->p))
            {
                foly f;
                (f.p).equalto(i->p);
                f.exp = (i->exp) + (j->exp);
                F.push_back(f);
                i = (f1.F).erase(i);
                j = (f2.F).erase(j);
                flag = 1;
                break;
            }
            else
                ++j;
        }
        if(flag == 0)
            ++i;
    }
    for(i=(f1.F).begin(); i!=(f1.F).end(); ++i)
        F.push_back((*i));
    for(j=(f2.F).begin(); j!=(f2.F).end(); ++j)
        F.push_back((*j));
}
void folynomial:: inmult(folynomial f)
{
    folynomial f1,f2;
    (f2.F).assign(F.begin(),F.end());
    f1.mult(f2,f);
    F.assign((f1.F).begin(),(f1.F).end());
}
void folynomial:: num(folynomial f1,folynomial f2)
{
   list<foly>:: iterator i = (f1.F).begin();
   list<foly>:: iterator j;
   int flag;
```

```
    while(i!=(f1.F).end())
    {
        j = (f2.F).begin();
        flag = 0;
        while(j!=(f2.F).end())
        {
            if((i->p).isequal((j->p)))
            {
                if(i->exp > j->exp)
                {
                    (i->exp) = (i->exp) - (j->exp);
                    j = (f2.F).erase(j);
                    goto label;
                }
                else if(i->exp < j->exp)
                {
                    (j->exp) = (j->exp) - (i->exp);
                    flag = 1;
                    i = (f1.F).erase(i);
                    goto label;
                }
                else
                {
                    i = (f1.F).erase(i);
                    j = (f2.F).erase(j);
                    flag = 1;
                    goto label;
                }
            }
            else
             ++j;
        }
        label:
            if(flag!=1)
            {
                ++i;
            }

    }
    for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
    {
        if((i->p).isconst())
        {
            for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
            {
                if((j->p).isconst())
                {
                    if((i->p).value()<(j->p).value())
                    {
                        (j->p).assignconst(((j->p).value())/((i->p).value()));
                        i = (f1.F).erase(i);
```

```
                        break;
                    }
                    else if((i->p).value()>(j->p).value())
                    {
                        (i->p).assignconst(((i->p).value())/((j->p).value()));
                        j = (f2.F).erase(j);
                        break;
                    }
                    else
                    {
                        i = (f1.F).erase(i);
                        j = (f2.F).erase(j);
                    }
                }
            }
        }
    }
    if((f1.F).empty())
    {
        foly f;
        (f.p).unit();
        f.exp = 0;
        (f1.F).push_back(f);
    }
    if((f2.F).empty())
    {
        foly f;
        (f.p).unit();
        f.exp = 0;
        (f2.F).push_back(f);
    }

    F.assign((f1.F).begin(),(f1.F).end());
}
void folynomial:: den(folynomial f1, folynomial f2)
{
        list<foly>:: iterator i = (f1.F).begin();
    list<foly>:: iterator j;
    int flag;
    while(i!=(f1.F).end())
    {
        j = (f2.F).begin();
        flag = 0;
        while(j!=(f2.F).end())
        {
            if((i->p).isequal((j->p)))
            {
                if(i->exp > j->exp)
                {
                    (i->exp) = (i->exp) - (j->exp);
                    j = (f2.F).erase(j);
```

```
                    goto label;
                }
                else if(i->exp < j->exp)
                {
                    (j->exp) = (j->exp) - (i->exp);
                    flag = 1;
                    i = (f1.F).erase(i);
                    goto label;
                }
                else
                {
                    i = (f1.F).erase(i);
                    j = (f2.F).erase(j);
                    flag = 1;
                    goto label;
                }
            }
            else
             ++j;
        }
        label:
            if(flag!=1)
            {
                ++i;
            }

    }
    for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
    {
        if((i->p).isconst())
        {
            for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
            {
                if((j->p).isconst())
                {
                    if((i->p).value()<(j->p).value())
                    {
                        (j->p).assignconst(((j->p).value())/((i->p).value()));
                        i = (f1.F).erase(i);
                        break;
                    }
                    else if((i->p).value()>(j->p).value())
                    {
                        (i->p).assignconst(((i->p).value())/((j->p).value()));
                        j = (f2.F).erase(j);
                        break;
                    }
                    else
                    {
                        i = (f1.F).erase(i);
```

```
                        j = (f2.F).erase(j);
                    }
                }
            }
        }
    }
    if((f1.F).empty())
    {
        foly f;
        (f.p).unit();
        f.exp = 0;
        (f1.F).push_back(f);
    }
    if((f2.F).empty())
    {
        foly f;
        (f.p).unit();
        f.exp = 0;
        (f2.F).push_back(f);
    }

    F.assign((f2.F).begin(),(f2.F).end());
}
void folynomial:: clean()
{
     F.clear();
}
void folynomial:: sim()
{
     int flag = 0;
     polynomial p1,p2;
     p1.unit();
     p2.unit();
     list<foly>:: iterator i = F.begin();
     while(i!=F.end())
     {
         flag = 0;
         if((i->p).isconst())
         {
             p1.inmult(i->p);
             i = F.erase(i);
             flag = 1;
         }
         if(flag != 1)
             ++i;
     }
         if(!p1.isequal(p2))
         {
             foly f;
             (f.p).equalto(p1);
             f.exp = 0;
             F.push_front(f);
```

```
        }
        i = F.begin();
        list<foly>:: iterator j;
        while(i!=F.end())
        {
            j = ++i;
            --i;
            while(j!=F.end())
            {
                flag = 0;
                if((i->p).isequal(j->p))
                {
                    (i->exp)++;
                    j = F.erase(j);
                    flag = 1;
                }
                if(flag==0)
                    ++j;
            }
            ++i;
        }
    }
    void folynomial:: unit()
    {
        foly f;
        (f.p).unit();
        f.exp = 1;
        F.push_back(f);
    }
    bool folynomial:: isunit()
    {
        if(F.size()==1)
        {
            if(((F.begin())->p).isunit())
                return true;
        }
        return false;
    }


```

TRANSFER FUNCTION HEADER FILE

```
#ifndef TRANSFERFUNCTION_H
#define TRANSFERFUNCTION_H
#include <iostream>
#include <list>
#include <cmath>
#include "polynomial.h"
#include "folynomial.h"
using namespace std;
class transferfunction
```

```
{
private:
    folynomial N;
    folynomial D;
public:
    void declare(void);
    void display(void);
    void add(transferfunction,transferfunction);
    void sub(transferfunction,transferfunction);
    void mult(transferfunction,transferfunction);
    void div(transferfunction,transferfunction);
    void inmult(transferfunction);
    void inadd(transferfunction);
    void insub(transferfunction);
    void equalto(transferfunction);
    void cleanup(void);
    void unit(void);
};

#endif // TRANSFERFUNCTION_H
```

TRANSFER FUNCTION CPP FILE

```
#include "transferfunction.h"
#include <iostream>
#include <list>
#include <cmath>
#include "folynomial.h"
#include "polynomial.h"
using namespace std;

void transferfunction:: declare()
{
    cout << "enter numerator:" << endl;
    N.declare();
    cout << "enter denominator:" << endl;
    D.declare();
}
void transferfunction:: display()
{
    N.display();
    cout << "/";
    D.display();
}
void transferfunction:: add(transferfunction t1, transferfunction t2)
{
    folynomial n,n1,n2;
    folynomial d;
    d.mult(t1.D,t2.D);
    n1.mult(t1.N,t2.D);
    n2.mult(t1.D,t2.N);
```

```
    n.add(n1,n2);
    N.num(n,d);
    D.den(n,d);
}
void transferfunction:: sub(transferfunction t1, transferfunction t2)
{
    folynomial n,n1,n2;
    folynomial d;
    d.mult(t1.D,t2.D);
    n1.mult(t1.N,t2.D);
    n2.mult(t1.D,t2.N);
    n.sub(n1,n2);
    N.num(n,d);
    D.den(n,d);
}
void transferfunction:: mult(transferfunction t1, transferfunction t2)
{
    folynomial n,d;
    n.mult((t1.N),(t2.N));
    d.mult((t1.D),(t2.D));
    N.num(n,d);
    D.den(n,d);
}
void transferfunction:: div(transferfunction t1, transferfunction t2)
{
    folynomial n,d;
    n.mult((t1.N),(t2.D));
    d.mult((t1.D),(t2.N));
    N.num(n,d);
    D.den(n,d);
}
void transferfunction:: inmult(transferfunction t)
{
    transferfunction t1,t2;
    (t1.N).equalto(N);
    (t1.D).equalto(D);
    t2.mult(t,t1);
    N.equalto(t2.N);
    D.equalto(t2.D);
}
void transferfunction:: inadd(transferfunction t)
{
    transferfunction t1,t2;
    (t1.N).equalto(N);
    (t1.D).equalto(D);
    t2.add(t,t1);
    N.equalto(t2.N);
    D.equalto(t2.D);
}
```

```
void transferfunction:: insub(transferfunction t)
{
    transferfunction t1,t2;
    (t1.N).equalto(N);
    (t1.D).equalto(D);
    t2.sub(t1,t);
    N.equalto(t2.N);
    D.equalto(t2.D);
}
void transferfunction:: equalto(transferfunction t)
{
    N.equalto(t.N);
    D.equalto(t.D);
}
void transferfunction:: cleanup()
{
    N.clean();
    D.clean();
}
void transferfunction:: unit()
{
    N.unit();
    D.unit();
}
```

DIGRAPH HEADER FILE

```
#ifndef DIGRAPH_H
#define DIGRAPH_H
#include <iostream>
#include <list>
#include <cmath>
#include <vector>
#include "polynomial.h"
#include "folynomial.h"
#include "transferfunction.h"
using namespace std;
struct sparse
{
    int from;
    int to;
    transferfunction weight;
};
struct Gp
{
    transferfunction gain;
    vector<int> pth;
};
class digraph
{
   private:
       vector< vector<int> > D;
```

```
        vector<sparse> W;
        vector<Gp> paths;
        vector<Gp> loops;
        transferfunction delta;

    public:
     void declare(void);
     void Paths(int,int);
     void getpath(void);
     void DFS(int);
     void Paths2(int,int);
     void getloops(void);
     bool disjoint(vector<int>, vector<int>);
     void del(int,vector<Gp>);
     void getdelta(void);
     void gettf(void);
     vector< vector<int> > update_D(vector<int>);
     vector<sparse> update_W(vector<int>);
     bool isperm(vector<int>, vector<int>);
};

#endif // DIGRAPH_H
```

DIGRAPH CPP FILE

```
#include "digraph.h"
#include <iostream>
#include <list>
#include <vector>
#include <cmath>
#include "folynomial.h"
#include "polynomial.h"
#include "transferfunction.h"
using namespace std;
    transferfunction p;
    vector<int> x;
    vector<int> visited;
    vector<int> deptm;
    int t=0;
void digraph:: declare()
{
    int x,k,m;
    cout << "enter no. of nodes in the graph:" << endl;
    cin >> x;
    for(int i=0; i<x; i++)
    {

        vector<int> I;
        cout << "enter no. of neighbours of node no." << i << endl;
        cin >> k;
        for(int j=1; j<=k; j++)
```

```
        {
            cout << "enter neighbour no." << j << endl;
            cin >> m;
            I.push_back(m);
        }
        D.push_back(I);
    }
    int n,f,t;
    transferfunction w;
    cout << "enter no. of edges:" << endl;
    cin >> n;
    for(int i=1; i<=n; i++)
    {
        sparse s;
        cout << "enter source node:" << endl;
        cin >> f;
        cout << "enter destination node:" << endl;
        cin >> t;
        cout << "ENTER CORRESPONDING WEIGHT:" << endl;
        (s.weight).declare();
        s.from = f;
        s.to = t;
        W.push_back(s);
    }
}
void digraph:: Paths(int a, int b)
{

    transferfunction q;
    q.equalto(p);
    vector<int> y;
    y.assign(x.begin(),x.end());
    if(a==b)
    {
        Gp temp;
        (temp.gain).equalto(p);
        (temp.pth).assign(x.begin(),x.end());
        paths.push_back(temp);
        return;
    }
    visited[a] = 1;
    for(int i=0; i<D[a].size(); i++)
    {
        if(visited[D[a][i]] == 0)
        {
            for(vector<sparse>:: iterator j=W.begin(); j!=W.end(); j++)
            {
                if((j->from) == a && (j->to) == D[a][i])
                {
                    p.inmult(j->weight);
                    x.push_back(D[a][i]);
```

```
                    break;
                }
            }
            Paths(D[a][i],b);
            p.equalto(q);
            x.assign(y.begin(),y.end());
        }
    }
    visited[a] = 0;
}
void digraph:: getpath()
{
    visited.assign(D.size(),0);
    int a,b;
    cout << "from?" << endl;
    cin >> a;
    cout << "to?" << endl;
    cin >> b;
    x.push_back(a);
    Paths(a,b);
    for(vector<Gp>:: iterator i=paths.begin(); i!=paths.end(); i++)
    {
        (i->gain).display();
        cout << "    ";
        for(int j=0; j<(i->pth).size(); j++)
        {
            cout << (i->pth)[j];
        }
        cout << endl;
    }

}
void digraph:: Paths2(int a, int b)
{

    transferfunction q;
    q.equalto(p);
    vector<int> y;
    y.assign(x.begin(),x.end());
    if(a==b)
    {
        Gp temp;
        (temp.gain).equalto(p);
        (temp.pth).assign(x.begin(),x.end());
        loops.push_back(temp);
        return;
    }
    visited[a] = 1;
    for(int i=0; i<D[a].size(); i++)
    {
        if(visited[D[a][i]] == 0)
```

```
        {
            for(vector<sparse>:: iterator j=W.begin(); j!=W.end(); j++)
            {
                if((j->from) == a && (j->to) == D[a][i])
                {
                    p.inmult(j->weight);
                    x.push_back(D[a][i]);
                    break;
                }
            }
            Paths2(D[a][i],b);
            p.equalto(q);
            x.assign(y.begin(),y.end());
        }
    }
    visited[a] = 0;
}
void digraph:: DFS(int a)
{
    t++;
    visited[a] = 1;
    for(int i=0; i<D[a].size(); i++)
    {
        if(visited[D[a][i]] == 0)
            DFS(D[a][i]);
    }
    t++;
    deptm[a] = t;
}

void digraph:: getloops()
{
    visited.clear();
    visited.assign(D.size(),0);
    deptm.assign(D.size(),0);
    for(int i=0; i<D.size(); i++)
    {
        if(visited[i] == 0)
            DFS(i);
    }
    int n=0;
   for(int j=0; j<W.size(); j++)
   {
       if(deptm[W[j].to]>=deptm[W[j].from])
       {
           visited.clear();
           visited.assign(D.size(),0);
           p.cleanup();
           x.clear();
           x.push_back(W[j].to);
           Paths2(W[j].to,W[j].from);
```

```
            for(int k=n; k<loops.size(); k++)
            {
                (loops[k].gain).inmult(W[j].weight);
                (loops[k].pth).push_back(W[j].to);
            }
        }
        n=loops.size();
    }
    vector<Gp>:: iterator i=loops.begin();
    vector<Gp>:: iterator j;
    while(i!=loops.end()){
     j=i+1;
     int flag=0;
     while(j!=loops.end()){

         if((i->pth).size()==(j->pth).size()){
             vector<int> a,b;
             a.assign((i->pth).begin(),(i->pth).end()-1);
             b.assign((j->pth).begin(),(j->pth).end()-1);
             if(isperm(a,b)){
                 loops.erase(i);
                 flag=1;
                 break;
             }
     }
         j++;
    }
    if(flag!=1)
    i++;
    }
    if(!loops.empty())
    {
    for(vector<Gp>:: iterator i=loops.begin(); i!=loops.end(); i++)
     {
         (i->gain).display();
         cout << "    ";
         for(int j=0; j<(i->pth).size(); j++)
         {
             cout << (i->pth)[j];
         }
         cout << endl;
     }
    }
    else
     cout << "no loops" << endl;


}
bool digraph:: disjoint(vector<int> L1, vector<int> L2)
{
    for(int i=0; i<L1.size(); i++)
```

```
    {
        for(int j=0; j<L2.size(); j++)
        {
            if(L1[i]==L2[j])
                return false;
        }
    }
    return true;
}
void digraph:: del(int v, vector<Gp> loops1)
{
    t=t+1;
    vector<Gp> loops2;
    loops2.assign(loops1.begin(),loops1.end());
    for(int i=v+1; i<loops1.size(); i++)
    {
        if(disjoint(loops1[v].pth, loops1[i].pth))
        {
            if(t%2==0){
                transferfunction r;
                r.mult(loops1[v].gain,loops1[i].gain);
                delta.inadd(r);
            }
            else{
                transferfunction r;
                r.mult(loops1[v].gain,loops1[i].gain);
                delta.insub(r);
            }
            int q=t;
            for(int j=0; j<(loops2[v].pth).size(); j++)
                (loops2[i].pth).push_back((loops2[v].pth)[j]);
            (loops2[i].gain).inmult(loops2[v].gain);
            del(i,loops2);
            t=q;
        }
    }
}
void digraph:: getdelta()
{
delta.unit();
transferfunction x;
for(int i=0; i<loops.size(); i++)
{
    x.inadd(loops[i].gain);
}
delta.insub(x);
for(int i=0; i<loops.size(); i++){
        t=1;
    del(i,loops);
}
transferfunction y;
y.unit();
```

```
delta.inadd(y);
}
vector< vector<int> > digraph:: update_D(vector<int> x)
{
    vector< vector<int> > D_temp;
    D_temp.assign(D.begin(),D.end());
    for(int i=0; i<x.size(); i++)
    {
        D_temp[x[i]].clear();
        for(int j=0; j<D_temp.size(); j++)
        {
            vector<int>:: iterator k=D_temp[j].begin();
            while(k!=D_temp[j].end())
            {
                if(*k==x[i])
                {
                    D_temp[j].erase(k);
                    break;
                }
                k++;
            }
        }
    }
    return D_temp;
}
vector<sparse> digraph:: update_W(vector<int> x)
{
    vector<sparse> W_temp;
    W_temp.assign(W.begin(),W.end());
    for(int i=0; i<x.size(); i++)
    {
        vector<sparse>:: iterator j=W_temp.begin();
        while(j!=W_temp.end())
        {
            int flag=0;
            if(j->from==x[i]||j->to==x[i])
            {
                W_temp.erase(j);
                flag=1;
            }
            if(flag==0)
                ++j;
        }
    }
    return W_temp;
}
void digraph:: gettf()
{
    getpath();
    getloops();
    getdelta();
    transferfunction tf1,tf3,tf4;
```

```
    for(int i=0; i<paths.size(); i++)
    {
        digraph d_temp;
        vector< vector<int> > D_temp=update_D(paths[i].pth);
        vector<sparse> W_temp=update_W(paths[i].pth);
        (d_temp.D).assign(D_temp.begin(),D_temp.end());
        (d_temp.W).assign(W_temp.begin(),W_temp.end());
        d_temp.getloops();
        d_temp.getdelta();
        transferfunction tf2;
        tf2.mult(paths[i].gain,d_temp.delta);
        tf1.inadd(tf2);
    }
    tf1.insub(tf4);
    tf3.div(tf1,delta);
    tf3.display();
}
bool digraph:: isperm(vector<int> A, vector<int> B)
{
    vector<int>:: iterator it1=A.begin();
    vector<int>:: iterator it2=B.begin();
    while(it2!=B.end()){
        if(*it1!=*it2)
            it2++;
        else{
            A.erase(it1);
            B.erase(it2);
            it2=B.begin();
        }
    }
    if(A.empty()&&B.empty())
        return true;
    else
        return false;
}
```