

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:

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

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

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)

- $L_i L_j L_k$ = 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 .

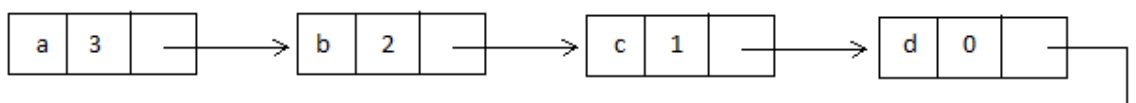
To make a class to represent a transfer function in C++, we need a class to represent polynomials.

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;
    .....
    .....
```

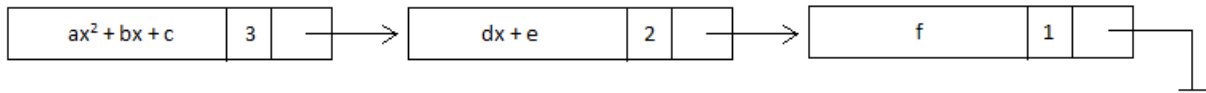
Operations:

```
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:

Values:

```
/** 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;
    .....
    .....
```

Operations:

```
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 inmalt(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:

Values:

```
/
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;
};
```

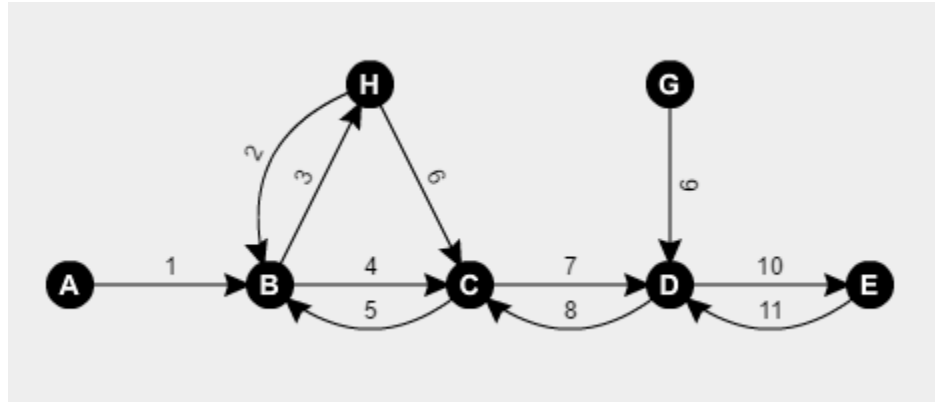
Operations:

```
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
};
```

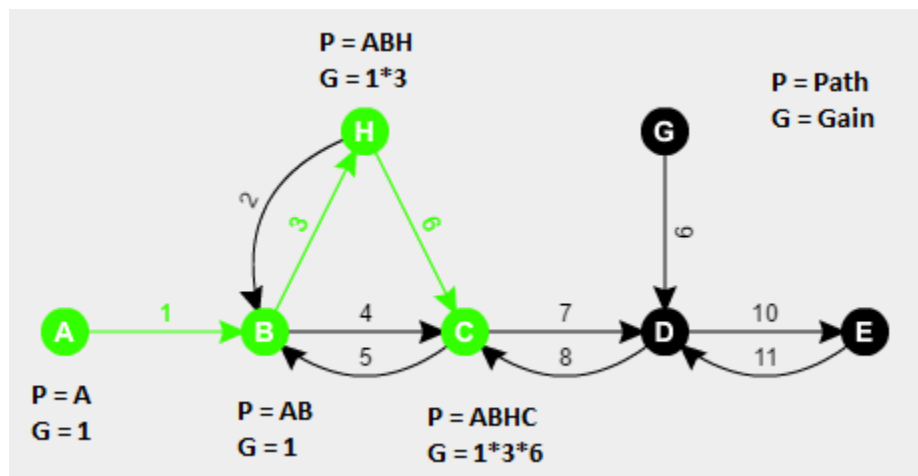
III. SIGNAL FLOW GRAPH ALGORITHMS

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

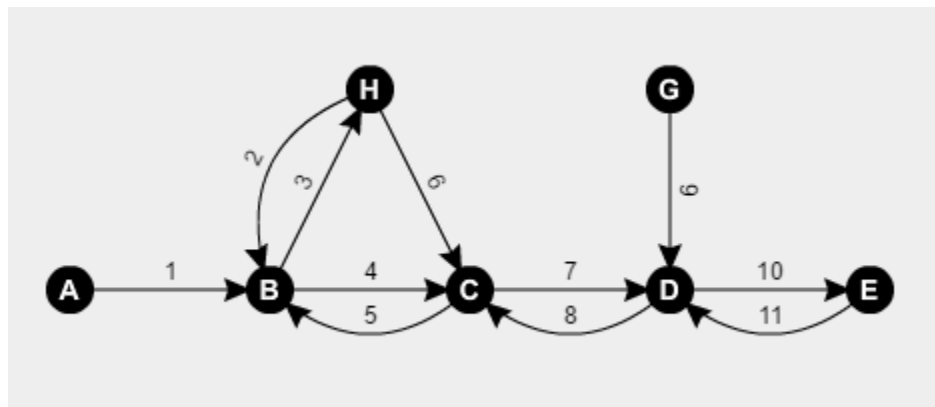

```

    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 w
            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:

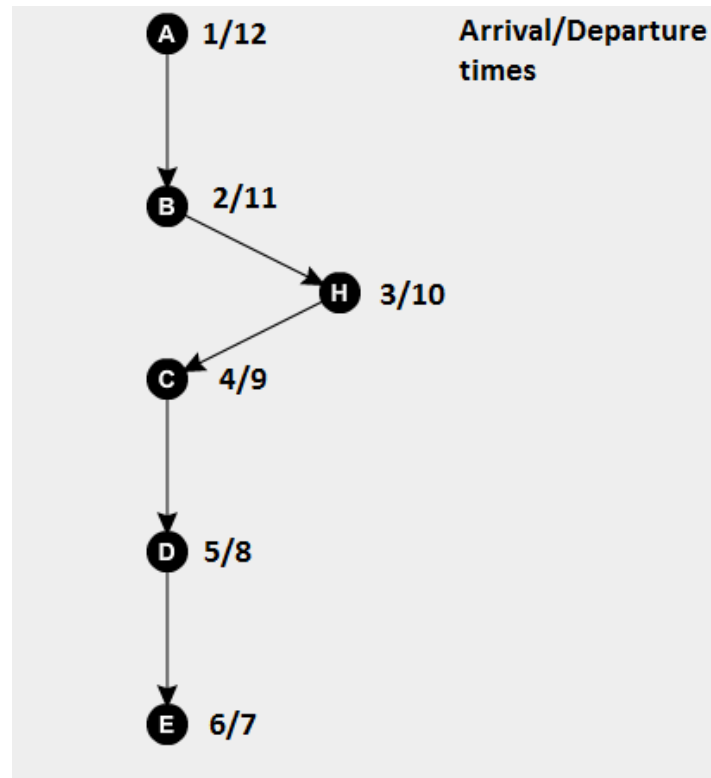


To detect a loop, we first perform a Depth First Search (DFS) on this graph and mark the *arrival times* and the *departure times* for every node in the DFS tree obtain on performing DFS.

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:

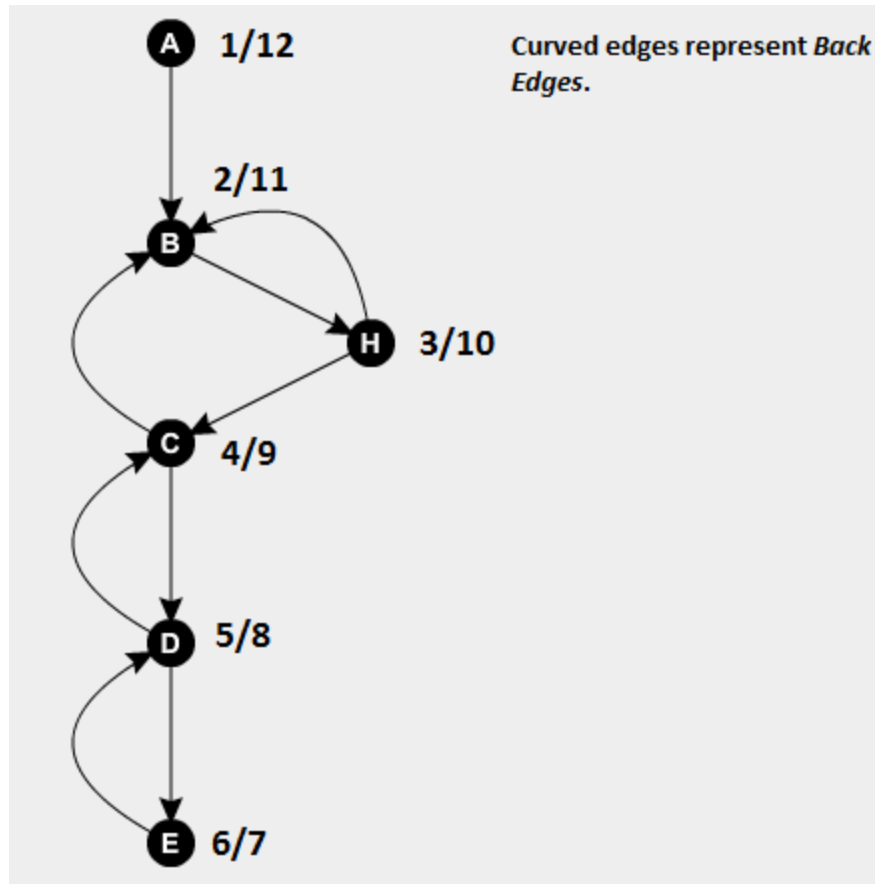
Tree Edge: Edges through which we reached (discovered) a vertex for the first time, i.e., edges leading to an unexplored vertex. These are the edges of the DFS tree.

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]:

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

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 .

$L_i(V) \cap L_j(V)$ = Set of all common vertices in $L_i(V)$ and $L_j(V)$.

$L_i(V) \cup L_j(V)$ = Union of the set of vertices from vector $L_i(V)$ and $L_j(V)$.

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

$L_i \wedge L_j = L_k$ which is a loop structure with $L_k(G) = L_i(G) \cdot L_j(G)$ and $L_k(V) = L_i(V) \cup L_j(V)$.

Pseudo code:

```

Set global k <- 1
Set global D <- 1 -  $\sum L_j$ 

procedure delta()
    Set S <-  $\{L_1, L_2, L_3, \dots, L_n\}$ 
    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  $L_{i+1}$  to  $L_n$  do
        if  $L_i(V) \cap L_j(V) = \emptyset$ 
            set D <- D +  $(-1)^k \cdot L_i(G) \cdot L_j(G)$ 
            set q <- k
            set  $s_2 \leftarrow \{L_1, L_2, \dots, L_i, \dots, L_{j-1}, L_i \wedge L_j, L_{j+1}, \dots, L_n\}$ 
            recursively call del(j,  $s_2$ )
            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

```
1  #ifndef POLYNOMIAL_H
2  #define POLYNOMIAL_H
3  #include <iostream>
4  #include <list>
5  #include <cmath>
6  using namespace std;
7  struct poly
8  {
9      float coeff;
10     float exp;
11 };
12 class polynomial
13 {
14 private:
15     list<poly> P;
16 public:
17     void declare(void);
18     void display(void);
19     void add(polynomial, polynomial);
20     void sub(polynomial, polynomial);
21     void mult(polynomial, polynomial);
22     int siz(void);
23     bool isequal(polynomial);
24     void equalto(polynomial);
25     void inmult(polynomial);
26     void unit(void);
27     bool isconst();
28     bool isunit();
29     float value(void);
30     void assignconst(float);
31 };
32 #endif // POLYNOMIAL_H
```

POLYNOMIAL CPP FILE

```
1  #include "polynomial.h"
2  #include <iostream>
3  #include <list>
4  #include <cmath>
5  using namespace std;
6
7  void polynomial::declare()
8  {
9      float c,e;
10     poly f;
11     int n;
```

```

12     cout << "enter no. of terms in the polynomial" << endl;
13     cin >> n;
14     cout << "enter successive coefficients and exponents" << endl;
15     for(int i=0; i<n; i++)
16     {
17
18         cin >> c;
19         cin >> e;
20         f.coeff = c;
21         f.exp = e;
22         P.push_back(f);
23
24     }
25 }
26 void polynomial:: display()
27 {
28     if(P.empty())
29     {
30         cout << "no polynomial entered." << endl;
31         return;
32     }
33     list<poly>::iterator i;
34     for(i=P.begin(); i!=P.end(); i++)
35     {
36         if(i->exp == 0)
37         {
38             if(i == P.begin())
39             {
40                 cout << i->coeff;
41             }
42             else
43             {
44                 if(i->coeff < 0)
45                     cout << i->coeff;
46                 else
47                     cout << "+" << i->coeff;
48             }
49         }
50         else if(i->exp == 1)
51         {
52             if(i->coeff != 1)
53             {
54                 if(i == P.begin())
55                 {
56
57                     if(i->coeff!=1)
58                         cout << i->coeff << "s";
59                     else
60                         cout << "-s";
61                 }
62                 else
63                 {

```



```

64         if(i->coeff > 0)
65             cout << "+" << i->coeff << "s";
66         else if(i->coeff != -1)
67             cout << i->coeff << "s";
68         else
69             cout << "-s";
70     }
71 }
72 else
73 {
74     if(i == P.begin())
75         cout << "s";
76     else
77     {
78         if(i->coeff > 0)
79             cout << "+" << "s";
80         else
81             cout << "s";
82     }
83 }
84 }
85 else
86 {
87     if(i->coeff != 1)
88     {
89         if(i == P.begin())
90         {
91             if(i->coeff != -1)
92                 cout << i->coeff << "s^" << i->exp;
93             else
94                 cout << "-s^" << i->exp;
95         }
96         else
97         {
98             if(i->coeff < 0)
99             {
100
101                 if(i->coeff != -1)
102                     cout << i->coeff << "s^" << i->exp;
103                 else
104                     cout << "-s^" << i->exp;
105             }
106             else
107                 cout << "+" << i->coeff << "s^" << i->exp;
108         }
109     }
110 else
111 {
112     if(i == P.begin())
113     {
114         cout << "s^" << i->exp;
115     }

```

```

116         else
117         {
118             if(i->coeff < 0)
119                 cout << "s^" << i->exp;
120             else
121                 cout << "+" << "s^" << i->exp;
122         }
123     }
124 }
125
126 }
127 }
128 void polynomial:: add(polynomial p1, polynomial p2)
129 {
130     if((p1.P).empty() && (p2.P).empty())
131     {
132         cout << "both polynomials empty" << endl;
133         return;
134     }
135     else if((p1.P).empty())
136     {
137         P.assign((p2.P).begin(), (p2.P).end());
138         return;
139     }
140     else if((p2.P).empty())
141     {
142         P.assign((p1.P).begin(), (p1.P).end());
143         return;
144     }
145     list<poly>:: iterator i = (p1.P).begin();
146     list<poly>:: iterator j = (p2.P).begin();
147     while(i!=(p1.P).end() && j!=(p2.P).end())
148     {
149         if(i->exp > j->exp)
150         {
151             P.push_back((*i));
152             ++i;
153         }
154         else if(j->exp > i->exp)
155         {
156             P.push_back((*j));
157             ++j;
158         }
159         else
160         {
161             poly f;
162             f.coeff = i->coeff + j->coeff;
163             f.exp = i->exp;
164             P.push_back(f);
165             ++i;
166             ++j;
167         }

```

```

168     }
169     if(i!=(p1.P).end())
170     {
171         P.insert(P.end(),i,(p1.P).end());
172     }
173     else if(j!=(p2.P).end())
174     {
175         P.insert(P.end(),j,(p2.P).end());
176     }
177 }
178 void polynomial:: sub(polynomial p1, polynomial p2)
179 {
180     for(list<poly>:: iterator i=(p2.P).begin(); i!=(p2.P).end();
181 ++i)
182     {
183         i->coeff = (-1)*(i->coeff);
184     }
185     polynomial p3;
186     p3.add(p1,p2);
187     list<poly>:: iterator j=(p3.P).begin();
188     int flag;
189     while(j!=(p3.P).end())
190     {
191         flag = 0;
192         if(j->coeff == 0)
193         {
194             j = (p3.P).erase(j);
195             flag = 1;
196         }
197         if(flag!=1)
198             j++;
199     }
200     P.assign((p3.P).begin(),(p3.P).end());
201 }
202 void polynomial:: mult(polynomial p1, polynomial p2)
203 {
204     for(list<poly>:: iterator i=(p1.P).begin(); i!=(p1.P).end();
205 ++i)
206     {
207         for(list<poly>:: iterator j=(p2.P).begin();
208 j!=(p2.P).end(); ++j)
209         {
210             poly f;
211             f.coeff = (i->coeff)*(j->coeff);
212             f.exp = (i->exp) + (j->exp);
213             int flag = 0;
214             for(list<poly>:: iterator k=P.begin(); k!=P.end();
215 ++k)
216             {
217                 if(k->exp == f.exp)
218                 {
219                     k->coeff += f.coeff;

```

```

216             flag = 1;
217             break;
218         }
219     }
220     if(flag==0)
221         P.push_back(f);
222     }
223 }
224 list<poly>:: iterator k=P.begin();
225 while(k!=P.end())
226 {
227     if(k->coeff == 0)
228         k = P.erase(k);
229     else
230         ++k;
231 }
232 }
233 int polynomial:: siz()
234 {
235     return (P.size());
236 }
237 bool polynomial:: isequal(polynomial p2)
238 {
239     list<poly>:: iterator i = P.begin();
240     list<poly>:: iterator j = (p2.P).begin();
241     int flag = 0;
242     while(i!=P.end() || j!=(p2.P).end())
243     {
244         if(((i->coeff) != (j->coeff) || ((i->exp) != (j)-
245 >exp))
246         {
247             flag = 1;
248             break;
249         }
250         i++;
251         j++;
252     }
253     if(i!=P.end() || j!=(p2.P).end())
254         return false;
255     else if(flag == 1)
256         return false;
257     else
258         return true;
259 }
260 void polynomial:: equalto(polynomial p)
261 {
262     for(list<poly>:: iterator i = (p.P).begin(); i!=(p.P).end();
263 i++)
264         P.push_back((*i));
265 }
266 void polynomial:: inmult(polynomial p)
267 {

```

```

266     polynomial p1,p2;
267     (p2.P).assign(P.begin(),P.end());
268     p1.mult(p2,p);
269     P.assign((p1.P).begin(),(p1.P).end());
270 }
271 void polynomial:: unit()
272 {
273     poly p;
274     p.coeff = 1;
275     p.exp = 0;
276     P.push_back(p);
277 }
278 bool polynomial:: isconst()
279 {
280     if(P.size() == 1)
281     {
282         if((P.begin())->exp == 0)
283             return true;
284         else
285             return false;
286     }
287     else
288         return false;
289 }
290 bool polynomial:: isunit()
291 {
292     if(P.size() == 1)
293     {
294         if((P.begin())->exp == 0 && (P.begin())->coeff == 1)
295             return true;
296         else
297             return false;
298     }
299     else
300         return false;
301 }
302 float polynomial:: value()
303 {
304     return ((P.begin())->coeff);
305 }
306 void polynomial:: assignconst(float n)
307 {
308     P.clear();
309     poly p;
310     p.coeff = n;
311     p.exp = 0;
312     P.push_back(p);
313 }
314

```

FACTORIZED POLYNOMIAL HEADER FILE

```
1  #ifndef POLYNOMIAL_H
2  #define POLYNOMIAL_H
3  #include <iostream>
4  #include <list>
5  #include <cmath>
6  #include "polynomial.h"
7  using namespace std;
8  struct foly
9  {
10     polynomial p;
11     int exp;
12 };
13 class folynomial
14 {
15 private:
16     list<foly> F;
17     void sim(void);
18 public:
19     void declare(void);
20     void display(void);
21     void add(folynomial, folynomial);
22     void sub(folynomial, folynomial);
23     polynomial expand(void);
24     void equalto(folynomial);
25     void mult(folynomial, folynomial);
26     void inmult(folynomial);
27     void num(folynomial, folynomial);
28     void den(folynomial, folynomial);
29     void clean(void);
30     void unit(void);
31     bool isunit(void);
32 };
33
34 #endif // POLYNOMIAL_H
```

FACTORIZED POLYNOMIAL CPP FILE

```
1  #include "folynomial.h"
2  #include <iostream>
3  #include <list>
4  #include <cmath>
5  #include "polynomial.h"
6  using namespace std;
7
8  void folynomial::declare()
9  {
10     int m,n;
11     cout << "enter number of factors in the polynomial:" << endl;
12     cin >> m;
13     for(int i=1; i<=m; i++)
```

```

14     {
15         foly f;
16         cout << "enter factor no." << i << ":" << endl;
17         (f.p).declare();
18         cout << "enter power of this factor:" << endl;
19         cin >> n;
20         f.exp = n;
21         F.push_back(f);
22     }
23 }
24 void polynomial:: display()
25 {
26     if(!isunit())
27         sim();
28     if(F.empty())
29     {
30         cout << "no polynomial entered" << endl;
31         return;
32     }
33     list<foly>:: iterator i;
34     for(i=F.begin(); i!=F.end(); i++)
35     {
36         if((i->p).siz() != 1)
37         {
38             cout << "(";
39             (i->p).display();
40             if(i->exp == 1)
41                 cout << ")";
42             else
43                 cout << ")^" << i->exp;
44         }
45         else
46         {
47             (i->p).display();
48             if(i->exp != 1 && !(i->p).isconst())
49                 cout << "^" << i->exp;
50         }
51     }
52 }
53 void polynomial:: add(polynomial f1, polynomial f2)
54 {
55     list<foly>:: iterator i = (f1.F).begin();
56     list<foly>:: iterator j;
57     int flag;
58     while(i!=(f1.F).end())
59     {
60         j = (f2.F).begin();
61         flag = 0;
62         while(j!=(f2.F).end())
63         {
64             if((i->p).isequal((j->p)))
65                 {

```

```

66         foly f;
67         if(i->exp > j->exp)
68         {
69             (f.p).equalto((j->p));
70             f.exp = j->exp;
71             F.push_back(f);
72
73             (i->exp) = (i->exp) - (j->exp);
74             j = (f2.F).erase(j);
75             goto label;
76         }
77         else if(i->exp < j->exp)
78         {
79             (f.p).equalto((j->p));
80             f.exp = i->exp;
81             F.push_back(f);
82
83             (j->exp) = (j->exp) - (i->exp);
84             flag = 1;
85             i = (f1.F).erase(i);
86             goto label;
87         }
88         else
89         {
90             (f.p).equalto((j->p));
91             f.exp = j->exp;
92             F.push_back(f);
93             i = (f1.F).erase(i);
94             j = (f2.F).erase(j);
95             flag = 1;
96             goto label;
97         }
98     }
99     else
100         ++j;
101 }
102 label:
103     if(flag!=1)
104     {
105         ++i;
106     }
107
108 }
109 for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
110 {
111     if((i->p).isconst())
112     {
113         for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
114         {
115             if((j->p).isconst())
116             {
117                 if((i->p).value()<(j->p).value())

```



```

118             {
119                 (j->p).assignconst(((j->p).value())/((i-
120 >p).value())));
121                 F.push_front((*i));
122                 i = (f1.F).erase(i);
123                 break;
124             }
125             else if((i->p).value()>(j->p).value())
126             {
127                 (i->p).assignconst(((i->p).value())/((j-
128 >p).value())));
129                 F.push_front((*j));
130                 j = (f2.F).erase(j);
131                 break;
132             }
133             else
134             {
135                 F.push_front((*i));
136                 i = (f1.F).erase(i);
137                 j = (f2.F).erase(j);
138             }
139         }
140     }
141
142     polynomial p1,p2,p3;
143     p1 = f1.expand();
144     p2 = f2.expand();
145     p3.add(p1,p2);
146     folly f;
147     (f.p).equalto(p3);
148     f.exp = 1;
149     F.push_back(f);
150 }
151 void folynomial:: sub(folynomial f1, folynomial f2)
152 {
153     list<folly>:: iterator i = (f1.F).begin();
154     list<folly>:: iterator j;
155     int flag;
156     while(i!=(f1.F).end())
157     {
158         j = (f2.F).begin();
159         flag = 0;
160         while(j!=(f2.F).end())
161         {
162             if((i->p).isequal((j->p)))
163             {
164                 folly f;
165                 if(i->exp > j->exp)
166                 {
167                     (f.p).equalto((j->p));

```

```

168         f.exp = j->exp;
169         F.push_back(f);
170
171         (i->exp) = (i->exp) - (j->exp);
172         j = (f2.F).erase(j);
173         goto label;
174     }
175     else if(i->exp < j->exp)
176     {
177         (f.p).equalto((j->p));
178         f.exp = i->exp;
179         F.push_back(f);
180
181         (j->exp) = (j->exp) - (i->exp);
182         flag = 1;
183         i = (f1.F).erase(i);
184         goto label;
185     }
186     else
187     {
188         (f.p).equalto((j->p));
189         f.exp = j->exp;
190         F.push_back(f);
191         i = (f1.F).erase(i);
192         j = (f2.F).erase(j);
193         flag = 1;
194         goto label;
195     }
196 }
197 else
198     ++j;
199 }
200 label:
201     if(flag!=1)
202     {
203         ++i;
204     }
205
206 }
207 for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
208 {
209     if((i->p).isconst())
210     {
211         for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
212         {
213             if((j->p).isconst())
214             {
215                 if((i->p).value()<(j->p).value())
216                 {
217                     (j->p).assignconst(((j->p).value())/((i-
218 >p).value()));
219                     F.push_front((*i));

```

```

219         i = (f1.F).erase(i);
220         break;
221     }
222     else if((i->p).value() > (j->p).value())
223     {
224         (i->p).assignconst(((i->p).value()) / ((j->
225 >p).value()));
226         F.push_front((*j));
227         j = (f2.F).erase(j);
228         break;
229     }
230     else
231     {
232         F.push_front((*i));
233         i = (f1.F).erase(i);
234         j = (f2.F).erase(j);
235     }
236 }
237 }
238 }
239
240 polynomial p1,p2,p3;
241 p1 = f1.expand();
242
243 p2 = f2.expand();
244 p3.sub(p1,p2);
245 foly f;
246 (f.p).equalto(p3);
247 f.exp = 1;
248 F.push_back(f);
249 }
250
251 polynomial folynomial:: expand()
252 {
253     polynomial q;
254     q.unit();
255     for(list<foly>:: iterator i = F.begin(); i!=F.end(); ++i)
256     {
257         for(int j=1; j<=(i->exp); j++)
258             q.inmult((i->p));
259     }
260     return q;
261 }
262 void folynomial:: equalto(folynomial f)
263 {
264     F.clear();
265     F.assign((f.F).begin(), (f.F).end());
266 }
267 void folynomial:: mult(folynomial f1, folynomial f2)
268 {
269     if((f1.F).size()==1 && (f2.F).size()==1) {

```

```

270         if(((f1.F).begin())->p).isunit() && ((f2.F).begin())-
271 >p).isunit()){
272             foly f;
273             (f.p).unit();
274             f.exp = 1;
275             F.push_back(f);
276             return;
277         }
278     list<foly>:: iterator i = (f1.F).begin();
279     list<foly>:: iterator j;
280     int flag;
281     while(i!=(f1.F).end())
282     {
283         flag = 0;
284         j = (f2.F).begin();
285         while(j!=(f2.F).end())
286         {
287             if((i->p).isequal(j->p))
288             {
289                 foly f;
290                 (f.p).equalto(i->p);
291                 f.exp = (i->exp) + (j->exp);
292                 F.push_back(f);
293                 i = (f1.F).erase(i);
294                 j = (f2.F).erase(j);
295                 flag = 1;
296                 break;
297             }
298             else
299                 ++j;
300         }
301         if(flag == 0)
302             ++i;
303     }
304     for(i=(f1.F).begin(); i!=(f1.F).end(); ++i)
305         F.push_back((*i));
306     for(j=(f2.F).begin(); j!=(f2.F).end(); ++j)
307         F.push_back((*j));
308 }
309 void folynomial:: inmult(folynomial f)
310 {
311     folynomial f1,f2;
312     (f2.F).assign(F.begin(),F.end());
313     f1.mult(f2,f);
314     F.assign((f1.F).begin(),(f1.F).end());
315 }
316 void folynomial:: num(folynomial f1,folynomial f2)
317 {
318     list<foly>:: iterator i = (f1.F).begin();
319     list<foly>:: iterator j;
320     int flag;

```

```

321 while (i!=(f1.F).end())
322 {
323     j = (f2.F).begin();
324     flag = 0;
325     while (j!=(f2.F).end())
326     {
327         if ((i->p).isequal((j->p)))
328         {
329             if (i->exp > j->exp)
330             {
331                 (i->exp) = (i->exp) - (j->exp);
332                 j = (f2.F).erase(j);
333                 goto label;
334             }
335             else if (i->exp < j->exp)
336             {
337                 (j->exp) = (j->exp) - (i->exp);
338                 flag = 1;
339                 i = (f1.F).erase(i);
340                 goto label;
341             }
342             else
343             {
344                 i = (f1.F).erase(i);
345                 j = (f2.F).erase(j);
346                 flag = 1;
347                 goto label;
348             }
349         }
350         else
351             ++j;
352     }
353     label:
354     if (flag!=1)
355     {
356         ++i;
357     }
358 }
359 }
360 for (i=(f1.F).begin(); i!=(f1.F).end(); i++)
361 {
362     if ((i->p).isconst())
363     {
364         for (j=(f2.F).begin(); j!=(f2.F).end(); j++)
365         {
366             if ((j->p).isconst())
367             {
368                 if ((i->p).value() < (j->p).value())
369                 {
370                     (j->p).assignconst(((j->p).value()) / ((i-
371 >p).value()));
372                     i = (f1.F).erase(i);

```

```

372                 break;
373             }
374             else if((i->p).value()>(j->p).value())
375             {
376                 (i->p).assignconst(((i->p).value())/((j->
>p).value()));
377                 j = (f2.F).erase(j);
378                 break;
379             }
380             else
381             {
382                 i = (f1.F).erase(i);
383                 j = (f2.F).erase(j);
384             }
385         }
386     }
387 }
388 }
389 if((f1.F).empty())
390 {
391     folyp f;
392     (f.p).unit();
393     f.exp = 0;
394     (f1.F).push_back(f);
395 }
396 if((f2.F).empty())
397 {
398     folyp f;
399     (f.p).unit();
400     f.exp = 0;
401     (f2.F).push_back(f);
402 }
403
404 F.assign((f1.F).begin(), (f1.F).end());
405 }
406 void polynomial:: den(polynomial f1, polynomial f2)
407 {
408     list<folyp>:: iterator i = (f1.F).begin();
409     list<folyp>:: iterator j;
410     int flag;
411     while(i!=(f1.F).end())
412     {
413         j = (f2.F).begin();
414         flag = 0;
415         while(j!=(f2.F).end())
416         {
417             if((i->p).isequal((j->p)))
418             {
419                 if(i->exp > j->exp)
420                 {
421                     (i->exp) = (i->exp) - (j->exp);
422                     j = (f2.F).erase(j);

```

```

423         goto label;
424     }
425     else if(i->exp < j->exp)
426     {
427         (j->exp) = (j->exp) - (i->exp);
428         flag = 1;
429         i = (f1.F).erase(i);
430         goto label;
431     }
432     else
433     {
434         i = (f1.F).erase(i);
435         j = (f2.F).erase(j);
436         flag = 1;
437         goto label;
438     }
439 }
440 else
441     ++j;
442 }
443 label:
444     if(flag!=1)
445     {
446         ++i;
447     }
448
449 }
450 for(i=(f1.F).begin(); i!=(f1.F).end(); i++)
451 {
452     if((i->p).isconst())
453     {
454         for(j=(f2.F).begin(); j!=(f2.F).end(); j++)
455         {
456             if((j->p).isconst())
457             {
458                 if((i->p).value()<(j->p).value())
459                 {
460                     (j->p).assignconst(((j->p).value())/((i->
461 >p).value()));
462                     i = (f1.F).erase(i);
463                     break;
464                 }
465                 else if((i->p).value()>(j->p).value())
466                 {
467                     (i->p).assignconst(((i->p).value())/((j->
468 >p).value()));
469                     j = (f2.F).erase(j);
470                     break;
471                 }
472             }
473             else
474             {
475                 i = (f1.F).erase(i);

```

```

473             j = (f2.F).erase(j);
474         }
475     }
476 }
477 }
478 }
479 if((f1.F).empty())
480 {
481     foly f;
482     (f.p).unit();
483     f.exp = 0;
484     (f1.F).push_back(f);
485 }
486 if((f2.F).empty())
487 {
488     foly f;
489     (f.p).unit();
490     f.exp = 0;
491     (f2.F).push_back(f);
492 }
493
494 F.assign((f2.F).begin(), (f2.F).end());
495 }
496 void polynomial:: clean()
497 {
498     F.clear();
499 }
500 void polynomial:: sim()
501 {
502     int flag = 0;
503     polynomial p1,p2;
504     p1.unit();
505     p2.unit();
506     list<folyp>:: iterator i = F.begin();
507     while(i!=F.end())
508     {
509         flag = 0;
510         if((i->p).isconst())
511         {
512             p1.inmult(i->p);
513             i = F.erase(i);
514             flag = 1;
515         }
516         if(flag != 1)
517             ++i;
518     }
519     if(!p1.isequal(p2))
520     {
521         foly f;
522         (f.p).equalto(p1);
523         f.exp = 0;
524         F.push_front(f);

```



```

525     }
526     i = F.begin();
527     list<folly>:: iterator j;
528     while(i!=F.end())
529     {
530         j = ++i;
531         --i;
532         while(j!=F.end())
533         {
534             flag = 0;
535             if((i->p).isequal(j->p))
536             {
537                 (i->exp)++;
538                 j = F.erase(j);
539                 flag = 1;
540             }
541             if(flag==0)
542                 ++j;
543         }
544         ++i;
545     }
546 }
547 void folynomial:: unit()
548 {
549     folly f;
550     (f.p).unit();
551     f.exp = 1;
552     F.push_back(f);
553 }
554 bool folynomial:: isunit()
555 {
556     if(F.size()==1)
557     {
558         if((F.begin())->p).isunit())
559             return true;
560     }
561     return false;
562 }
563
564

```

TRANSFER FUNCTION HEADER FILE

```

1  #ifndef TRANSFERFUNCTION_H
2  #define TRANSFERFUNCTION_H
3  #include <iostream>
4  #include <list>
5  #include <cmath>
6  #include "polynomial.h"
7  #include "folynomial.h"
8  using namespace std;
9  class transferfunction

```

```

10 {
11 private:
12     folynomial N;
13     folynomial D;
14 public:
15     void declare(void);
16     void display(void);
17     void add(transferfunction, transferfunction);
18     void sub(transferfunction, transferfunction);
19     void mult(transferfunction, transferfunction);
20     void div(transferfunction, transferfunction);
21     void inmult(transferfunction);
22     void inadd(transferfunction);
23     void insub(transferfunction);
24     void equalto(transferfunction);
25     void cleanup(void);
26     void unit(void);
27 };
28
29 #endif // TRANSFERFUNCTION_H

```

TRANSFER FUNCTION CPP FILE

```

1  #include "transferfunction.h"
2  #include <iostream>
3  #include <list>
4  #include <cmath>
5  #include "folynomial.h"
6  #include "polynomial.h"
7  using namespace std;
8
9  void transferfunction::declare()
10 {
11     cout << "enter numerator:" << endl;
12     N.declare();
13     cout << "enter denominator:" << endl;
14     D.declare();
15 }
16 void transferfunction::display()
17 {
18     N.display();
19     cout << "/";
20     D.display();
21 }
22 void transferfunction::add(transferfunction t1, transferfunction
t2)
23 {
24     folynomial n, n1, n2;
25     folynomial d;
26     d.mult(t1.D, t2.D);
27     n1.mult(t1.N, t2.D);
28     n2.mult(t1.D, t2.N);

```

```

29     n.add(n1,n2);
30     N.num(n,d);
31     D.den(n,d);
32 }
33 void transferfunction:: sub(transferfunction t1, transferfunction
t2)
34 {
35     folynomial n,n1,n2;
36     folynomial d;
37     d.mult(t1.D,t2.D);
38     n1.mult(t1.N,t2.D);
39     n2.mult(t1.D,t2.N);
40     n.sub(n1,n2);
41     N.num(n,d);
42     D.den(n,d);
43 }
44 void transferfunction:: mult(transferfunction t1,
transferfunction t2)
45 {
46     folynomial n,d;
47     n.mult((t1.N),(t2.N));
48     d.mult((t1.D),(t2.D));
49     N.num(n,d);
50     D.den(n,d);
51 }
52 void transferfunction:: div(transferfunction t1, transferfunction
t2)
53 {
54     folynomial n,d;
55     n.mult((t1.N),(t2.D));
56     d.mult((t1.D),(t2.N));
57     N.num(n,d);
58     D.den(n,d);
59 }
60 void transferfunction:: inmult(transferfunction t)
61 {
62     transferfunction t1,t2;
63     (t1.N).equalto(N);
64     (t1.D).equalto(D);
65     t2.mult(t,t1);
66     N.equalto(t2.N);
67     D.equalto(t2.D);
68 }
69 void transferfunction:: inadd(transferfunction t)
70 {
71     transferfunction t1,t2;
72     (t1.N).equalto(N);
73     (t1.D).equalto(D);
74     t2.add(t,t1);
75     N.equalto(t2.N);
76     D.equalto(t2.D);
77 }

```

```

78 void transferfunction:: insub(transferfunction t)
79 {
80     transferfunction t1,t2;
81     (t1.N).equalto(N);
82     (t1.D).equalto(D);
83     t2.sub(t1,t);
84     N.equalto(t2.N);
85     D.equalto(t2.D);
86 }
87 void transferfunction:: equalto(transferfunction t)
88 {
89     N.equalto(t.N);
90     D.equalto(t.D);
91 }
92 void transferfunction:: cleanup()
93 {
94     N.clean();
95     D.clean();
96 }
97 void transferfunction:: unit()
98 {
99     N.unit();
100    D.unit();
101 }

```

DIGRAPH HEADER FILE

```

1  #ifndef DIGRAPH_H
2  #define DIGRAPH_H
3  #include <iostream>
4  #include <list>
5  #include <cmath>
6  #include <vector>
7  #include "polynomial.h"
8  #include "folynomial.h"
9  #include "transferfunction.h"
10 using namespace std;
11 struct sparse
12 {
13     int from;
14     int to;
15     transferfunction weight;
16 };
17 struct Gp
18 {
19     transferfunction gain;
20     vector<int> pth;
21 };
22 class digraph
23 {
24     private:
25         vector< vector<int> > D;

```

```

26     vector<sparse> W;
27     vector<Gp> paths;
28     vector<Gp> loops;
29     transferfunction delta;
30
31 public:
32     void declare(void);
33     void Paths(int,int);
34     void getpath(void);
35     void DFS(int);
36     void Paths2(int,int);
37     void getloops(void);
38     bool disjoint(vector<int>, vector<int>);
39     void del(int,vector<Gp>);
40     void getdelta(void);
41     void gettf(void);
42     vector< vector<int> > update_D(vector<int>);
43     vector<sparse> update_W(vector<int>);
44     bool isperm(vector<int>, vector<int>);
45 };
46
47 #endif // DIGRAPH_H

```

DIGRAPH CPP FILE

```

1  #include "digraph.h"
2  #include <iostream>
3  #include <list>
4  #include <vector>
5  #include <cmath>
6  #include "folynomial.h"
7  #include "polynomial.h"
8  #include "transferfunction.h"
9  using namespace std;
10     transferfunction p;
11     vector<int> x;
12     vector<int> visited;
13     vector<int> deptm;
14     int t=0;
15 void digraph:: declare()
16 {
17     int x,k,m;
18     cout << "enter no. of nodes in the graph:" << endl;
19     cin >> x;
20     for(int i=0; i<x; i++)
21     {
22
23         vector<int> I;
24         cout << "enter no. of neighbours of node no." << i <<
endl;
25         cin >> k;
26         for(int j=1; j<=k; j++)

```

```

27         {
28             cout << "enter neighbour no." << j << endl;
29             cin >> m;
30             I.push_back(m);
31         }
32         D.push_back(I);
33     }
34     int n,f,t;
35     transferfunction w;
36     cout << "enter no. of edges:" << endl;
37     cin >> n;
38     for(int i=1; i<=n; i++)
39     {
40         sparse s;
41         cout << "enter source node:" << endl;
42         cin >> f;
43         cout << "enter destination node:" << endl;
44         cin >> t;
45         cout << "ENTER CORRESPONDING WEIGHT:" << endl;
46         (s.weight).declare();
47         s.from = f;
48         s.to = t;
49         W.push_back(s);
50     }
51 }
52 void digraph:: Paths(int a, int b)
53 {
54
55     transferfunction q;
56     q.equalto(p);
57     vector<int> y;
58     y.assign(x.begin(),x.end());
59     if(a==b)
60     {
61         Gp temp;
62         (temp.gain).equalto(p);
63         (temp.pth).assign(x.begin(),x.end());
64         paths.push_back(temp);
65         return;
66     }
67     visited[a] = 1;
68     for(int i=0; i<D[a].size(); i++)
69     {
70         if(visited[D[a][i]] == 0)
71         {
72             for(vector<sparse>:: iterator j=W.begin();
j!=W.end(); j++)
73             {
74                 if((j->from) == a && (j->to) == D[a][i])
75                 {
76                     p.inmult(j->weight);
77                     x.push_back(D[a][i]);

```

```

78             break;
79         }
80     }
81     Paths(D[a][i],b);
82     p.equalto(q);
83     x.assign(y.begin(),y.end());
84 }
85 }
86 visited[a] = 0;
87 }
88 void digraph:: getpath()
89 {
90     visited.assign(D.size(),0);
91     int a,b;
92     cout << "from?" << endl;
93     cin >> a;
94     cout << "to?" << endl;
95     cin >> b;
96     x.push_back(a);
97     Paths(a,b);
98     for(vector<Gp>:: iterator i=paths.begin(); i!=paths.end();
i++)
99     {
100         (i->gain).display();
101         cout << "    ";
102         for(int j=0; j<(i->pth).size(); j++)
103         {
104             cout << (i->pth)[j];
105         }
106         cout << endl;
107     }
108 }
109 }
110 void digraph:: Paths2(int a, int b)
111 {
112     transferfunction q;
113     q.equalto(p);
114     vector<int> y;
115     y.assign(x.begin(),x.end());
116     if(a==b)
117     {
118         Gp temp;
119         (temp.gain).equalto(p);
120         (temp.pth).assign(x.begin(),x.end());
121         loops.push_back(temp);
122         return;
123     }
124     visited[a] = 1;
125     for(int i=0; i<D[a].size(); i++)
126     {
127         if(visited[D[a][i]] == 0)

```

```

129         {
130             for(vector<sparse>:: iterator j=W.begin();
j!=W.end(); j++)
131             {
132                 if((j->from) == a && (j->to) == D[a][i])
133                 {
134                     p.inmult(j->weight);
135                     x.push_back(D[a][i]);
136                     break;
137                 }
138             }
139             Paths2(D[a][i],b);
140             p.equalto(q);
141             x.assign(y.begin(),y.end());
142         }
143     }
144     visited[a] = 0;
145 }
146 void digraph:: DFS(int a)
147 {
148     t++;
149     visited[a] = 1;
150     for(int i=0; i<D[a].size(); i++)
151     {
152         if(visited[D[a][i]] == 0)
153             DFS(D[a][i]);
154     }
155     t++;
156     deptm[a] = t;
157 }
158
159 void digraph:: getloops()
160 {
161     visited.clear();
162     visited.assign(D.size(),0);
163     deptm.assign(D.size(),0);
164     for(int i=0; i<D.size(); i++)
165     {
166         if(visited[i] == 0)
167             DFS(i);
168     }
169     int n=0;
170     for(int j=0; j<W.size(); j++)
171     {
172         if(deptm[W[j].to]>=deptm[W[j].from])
173         {
174             visited.clear();
175             visited.assign(D.size(),0);
176             p.cleanup();
177             x.clear();
178             x.push_back(W[j].to);
179             Paths2(W[j].to,W[j].from);

```



```

180         for(int k=n; k<loops.size(); k++)
181         {
182             (loops[k].gain).inmult(W[j].weight);
183             (loops[k].pth).push_back(W[j].to);
184         }
185     }
186     n=loops.size();
187 }
188 vector<Gp>:: iterator i=loops.begin();
189 vector<Gp>:: iterator j;
190 while(i!=loops.end()){
191     j=i+1;
192     int flag=0;
193     while(j!=loops.end()){
194
195         if((i->pth).size()==(j->pth).size()){
196             vector<int> a,b;
197             a.assign((i->pth).begin(),(i->pth).end()-1);
198             b.assign((j->pth).begin(),(j->pth).end()-1);
199             if(isperm(a,b)){
200                 loops.erase(i);
201                 flag=1;
202                 break;
203             }
204         }
205         j++;
206     }
207     if(flag!=1)
208         i++;
209 }
210 if(!loops.empty())
211 {
212     for(vector<Gp>:: iterator i=loops.begin(); i!=loops.end();
213 i++)
214     {
215         (i->gain).display();
216         cout << "      ";
217         for(int j=0; j<(i->pth).size(); j++)
218         {
219             cout << (i->pth)[j];
220         }
221         cout << endl;
222     }
223     else
224         cout << "no loops" << endl;
225 }
226
227 }
228 bool digraph:: disjoint(vector<int> L1, vector<int> L2)
229 {
230     for(int i=0; i<L1.size(); i++)

```

```

231     {
232         for(int j=0; j<L2.size(); j++)
233         {
234             if(L1[i]==L2[j])
235                 return false;
236         }
237     }
238     return true;
239 }
240 void digraph:: del(int v, vector<Gp> loops1)
241 {
242     t=t+1;
243     vector<Gp> loops2;
244     loops2.assign(loops1.begin(), loops1.end());
245     for(int i=v+1; i<loops1.size(); i++)
246     {
247         if(disjoint(loops1[v].pth, loops1[i].pth))
248         {
249             if(t%2==0) {
250                 transferfunction r;
251                 r.mult(loops1[v].gain, loops1[i].gain);
252                 delta.inadd(r);
253             }
254             else{
255                 transferfunction r;
256                 r.mult(loops1[v].gain, loops1[i].gain);
257                 delta.insub(r);
258             }
259             int q=t;
260             for(int j=0; j<(loops2[v].pth).size(); j++)
261                 (loops2[i].pth).push_back((loops2[v].pth)[j]);
262             (loops2[i].gain).inmult(loops2[v].gain);
263             del(i, loops2);
264             t=q;
265         }
266     }
267 }
268 void digraph:: getdelta()
269 {
270     delta.unit();
271     transferfunction x;
272     for(int i=0; i<loops.size(); i++)
273     {
274         x.inadd(loops[i].gain);
275     }
276     delta.insub(x);
277     for(int i=0; i<loops.size(); i++){
278         t=1;
279         del(i, loops);
280     }
281     transferfunction y;
282     y.unit();

```

```

283 delta.inadd(y);
284 }
285 vector< vector<int> > digraph:: update_D(vector<int> x)
286 {
287     vector< vector<int> > D_temp;
288     D_temp.assign(D.begin(),D.end());
289     for(int i=0; i<x.size(); i++)
290     {
291         D_temp[x[i]].clear();
292         for(int j=0; j<D_temp.size(); j++)
293         {
294             vector<int>:: iterator k=D_temp[j].begin();
295             while(k!=D_temp[j].end())
296             {
297                 if(*k==x[i])
298                 {
299                     D_temp[j].erase(k);
300                     break;
301                 }
302                 k++;
303             }
304         }
305     }
306     return D_temp;
307 }
308 vector<sparse> digraph:: update_W(vector<int> x)
309 {
310     vector<sparse> W_temp;
311     W_temp.assign(W.begin(),W.end());
312     for(int i=0; i<x.size(); i++)
313     {
314         vector<sparse>:: iterator j=W_temp.begin();
315         while(j!=W_temp.end())
316         {
317             int flag=0;
318             if(j->from==x[i]||j->to==x[i])
319             {
320                 W_temp.erase(j);
321                 flag=1;
322             }
323             if(flag==0)
324                 ++j;
325         }
326     }
327     return W_temp;
328 }
329 void digraph:: gettf()
330 {
331     getpath();
332     getloops();
333     getdelta();
334     transferfunction tf1,tf3,tf4;

```

```

335     for(int i=0; i<paths.size(); i++)
336     {
337         digraph d_temp;
338         vector< vector<int> > D_temp=update_D(paths[i].pth);
339         vector<sparse> W_temp=update_W(paths[i].pth);
340         (d_temp.D).assign(D_temp.begin(),D_temp.end());
341         (d_temp.W).assign(W_temp.begin(),W_temp.end());
342         d_temp.getloops();
343         d_temp.getdelta();
344         transferfunction tf2;
345         tf2.mult(paths[i].gain,d_temp.delta);
346         tf1.inadd(tf2);
347     }
348     tf1.insub(tf4);
349     tf3.div(tf1,delta);
350     tf3.display();
351 }
352 bool digraph:: isperm(vector<int> A, vector<int> B)
353 {
354     vector<int>:: iterator it1=A.begin();
355     vector<int>:: iterator it2=B.begin();
356     while(it2!=B.end()){
357         if(*it1!=*it2)
358             it2++;
359         else{
360             A.erase(it1);
361             B.erase(it2);
362             it2=B.begin();
363         }
364     }
365     if(A.empty() && B.empty())
366         return true;
367     else
368         return false;
369 }

```