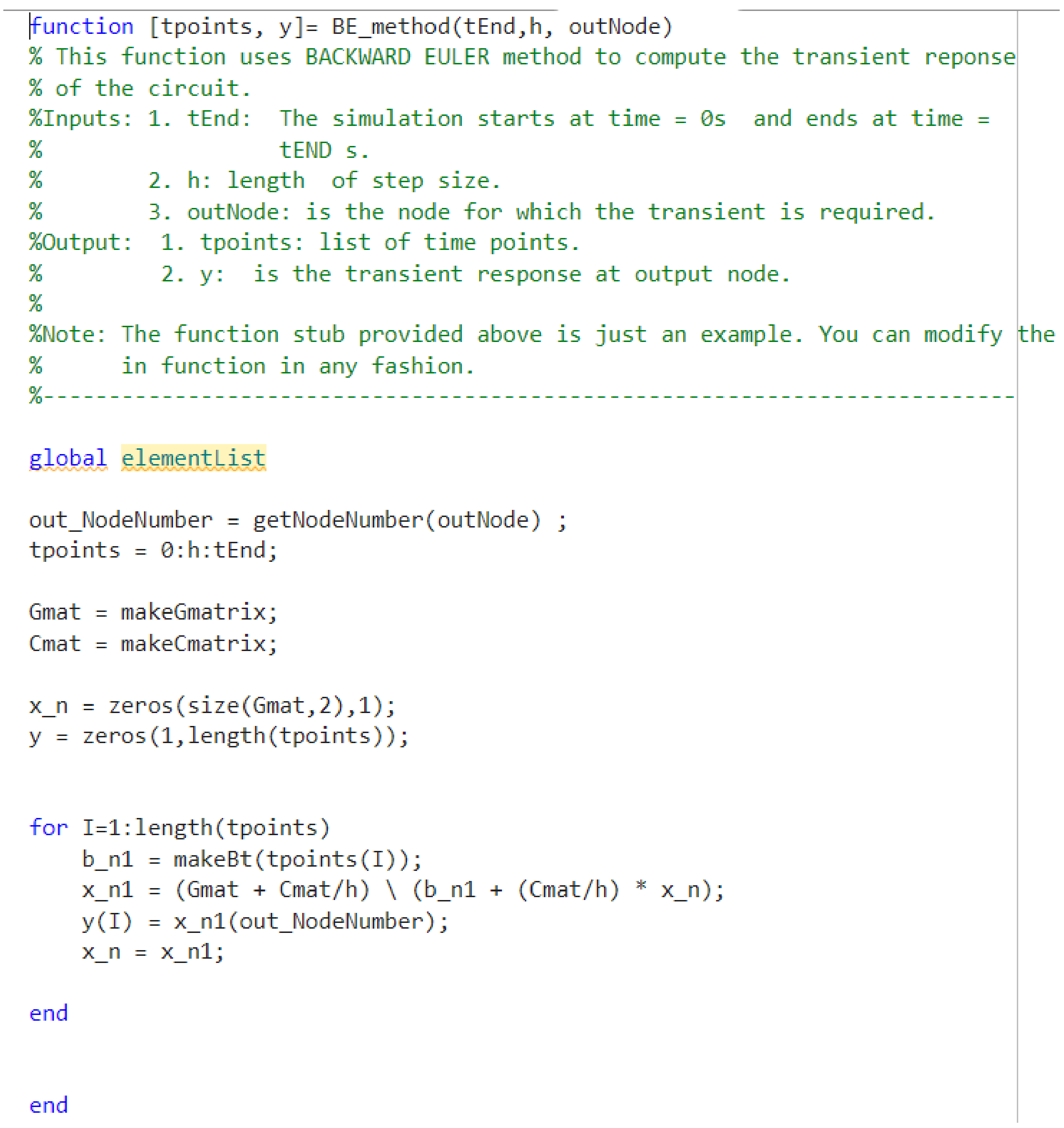
ECSE 597 Assignment 3

Yicheng Song 260763294

Q1.



```
function [tpoints, y]= BE_method(tEnd,h, outNode)
% This function uses BACKWARD EULER method to compute the transient reponse
% of the circuit.
%Inputs: 1. tEnd:  The simulation starts at time = 0s  and ends at time =
%                  tEND s.
%        2. h: length  of step size.
%        3. outNode: is the node for which the transient is required.
%Output:  1. tpoints: list of time points.
%         2. y:  is the transient response at output node.
%
%Note: The function stub provided above is just an example. You can modify the
%      in function in any fashion.
%----------------------------------------------------------------------------

global elementList

out_NodeNumber = getNodeNumber(outNode) ;
tpoints = 0:h:tEnd;

Gmat = makeGmatrix;
Cmat = makeCmatrix;

x_n = zeros(size(Gmat,2),1);
y = zeros(1,length(tpoints));


for I=1:length(tpoints)
    b_n1 = makeBt(tpoints(I));
    x_n1 = (Gmat + Cmat/h) \ (b_n1 + (Cmat/h) * x_n);
    y(I) = x_n1(out_NodeNumber);
    x_n = x_n1;

end


end
```

Fig 1. Code of BE method.

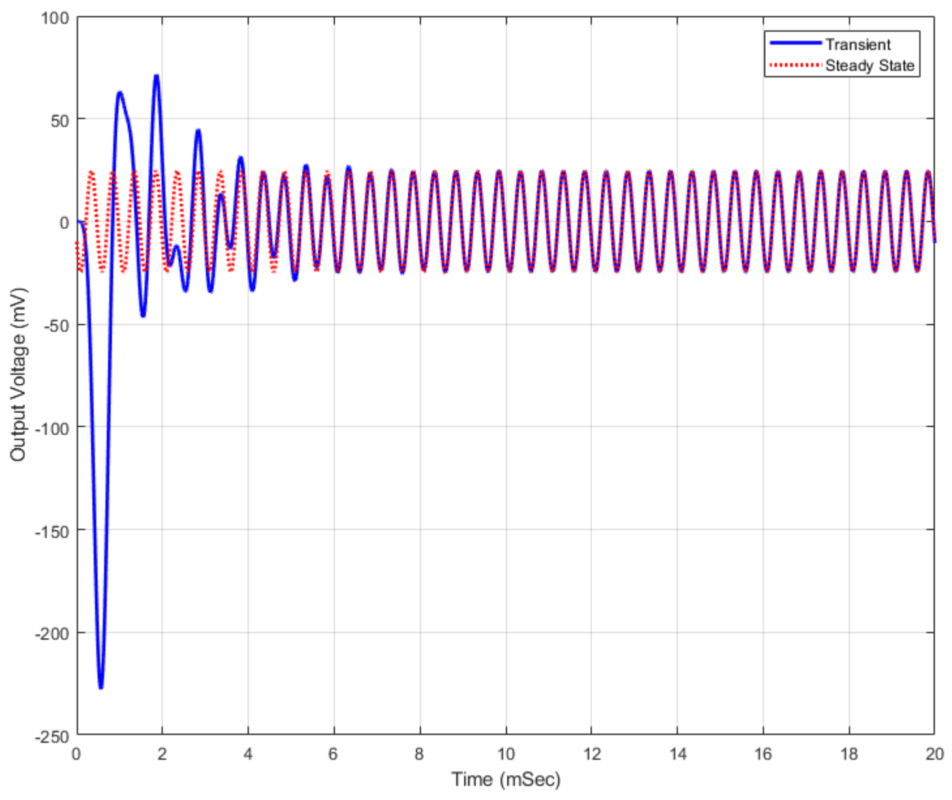
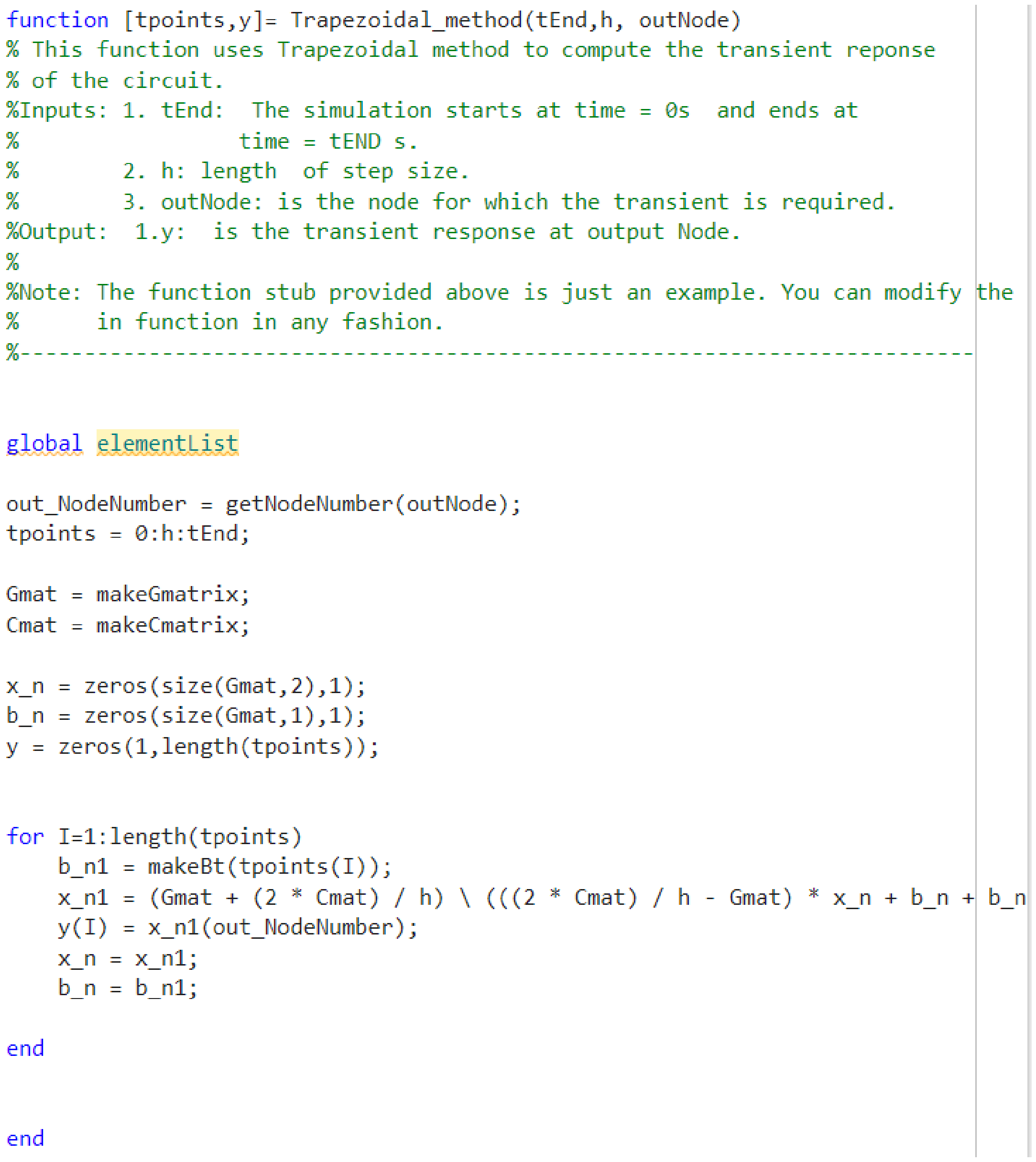


Fig 2. Result of testbench q1.

The result of testbench q1 show that the transient response is consistent with steady state response when time is larger than 6 ms, which is a sine wave.

Q2.



```
function [tpoints,y]= Trapezoidal_method(tEnd,h, outNode)
% This function uses Trapezoidal method to compute the transient reponse
% of the circuit.
%Inputs: 1. tEnd:  The simulation starts at time = 0s  and ends at
%                  time = tEND s.
%        2. h: length  of step size.
%        3. outNode: is the node for which the transient is required.
%Output:  1.y:  is the transient response at output Node.
%
%Note: The function stub provided above is just an example. You can modify the
%      in function in any fashion.
%---------------------------------------------------------------------------


global elementList

out_NodeNumber = getNodeNumber(outNode);
tpoints = 0:h:tEnd;

Gmat = makeGmatrix;
Cmat = makeCmatrix;

x_n = zeros(size(Gmat,2),1);
b_n = zeros(size(Gmat,1),1);
y = zeros(1,length(tpoints));


for I=1:length(tpoints)
    b_n1 = makeBt(tpoints(I));
    x_n1 = (Gmat + (2 * Cmat) / h) \ (((2 * Cmat) / h - Gmat) * x_n + b_n + b_n
    y(I) = x_n1(out_NodeNumber);
    x_n = x_n1;
    b_n = b_n1;

end


end
```

Fig 3. Code of TR method.

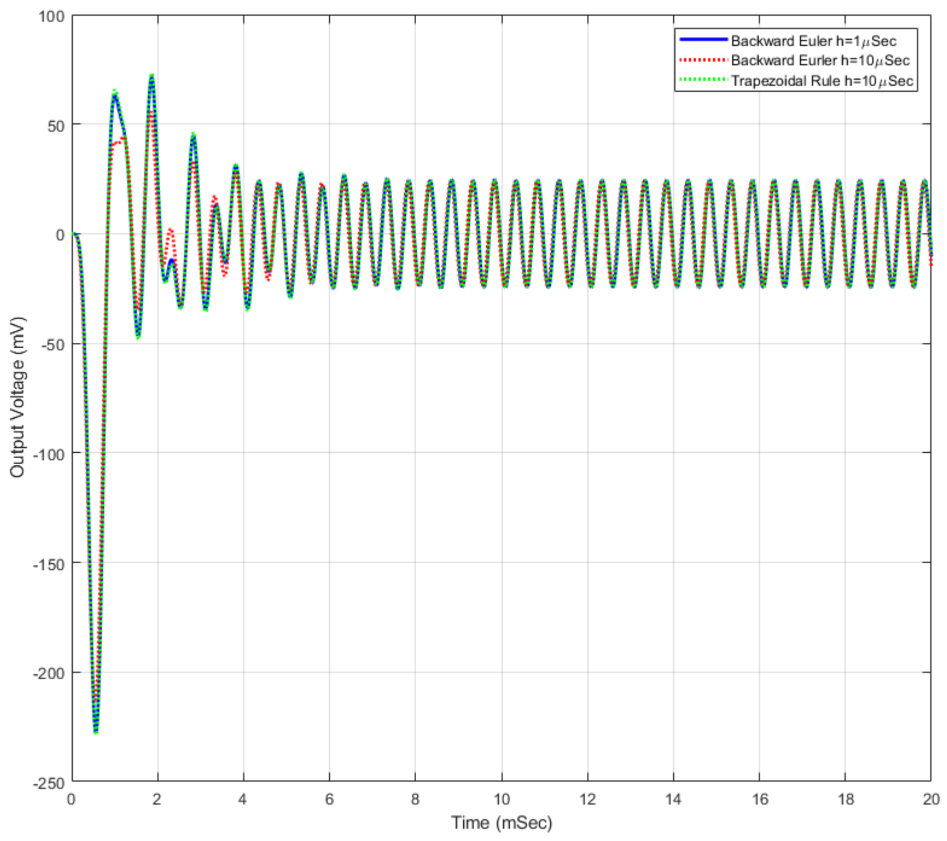
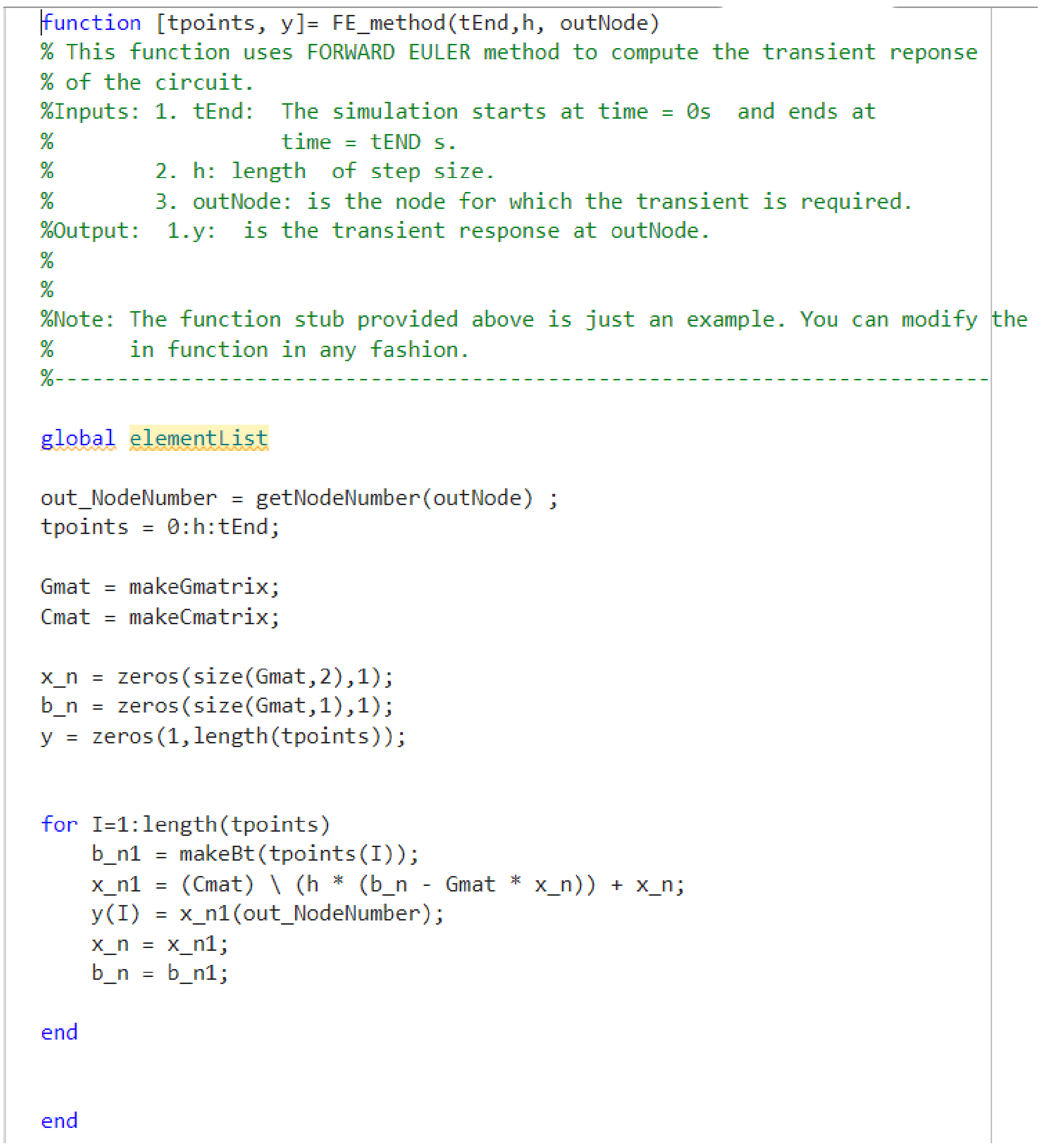


Fig 4. Result of testbench q2.

The result of testbench q2 shows that all 3 outputs are consistent when time is large. For BE method, h=1 is more accurate than h=10 compared to the plot of TR. Moreover, when step size is fixed (for example, h=10 in the plot above), TR provides a more accurate result than BE.

Q3.



```
function [tpoints, y]= FE_method(tEnd,h, outNode)
% This function uses FORWARD EULER method to compute the transient reponse
% of the circuit.
%Inputs: 1. tEnd:  The simulation starts at time = 0s  and ends at
%                  time = tEND s.
%        2. h: length  of step size.
%        3. outNode: is the node for which the transient is required.
%Output:  1.y:  is the transient response at outNode.
%
%
%Note: The function stub provided above is just an example. You can modify the
%      in function in any fashion.
%---------------------------------------------------------------------------

global elementList

out_NodeNumber = getNodeNumber(outNode) ;
tpoints = 0:h:tEnd;

Gmat = makeGmatrix;
Cmat = makeCmatrix;

x_n = zeros(size(Gmat,2),1);
b_n = zeros(size(Gmat,1),1);
y = zeros(1,length(tpoints));


for I=1:length(tpoints)
    b_n1 = makeBt(tpoints(I));
    x_n1 = (Cmat) \ (h * (b_n - Gmat * x_n)) + x_n;
    y(I) = x_n1(out_NodeNumber);
    x_n = x_n1;
    b_n = b_n1;

end


end
```

Fig 5. Code of FE method.

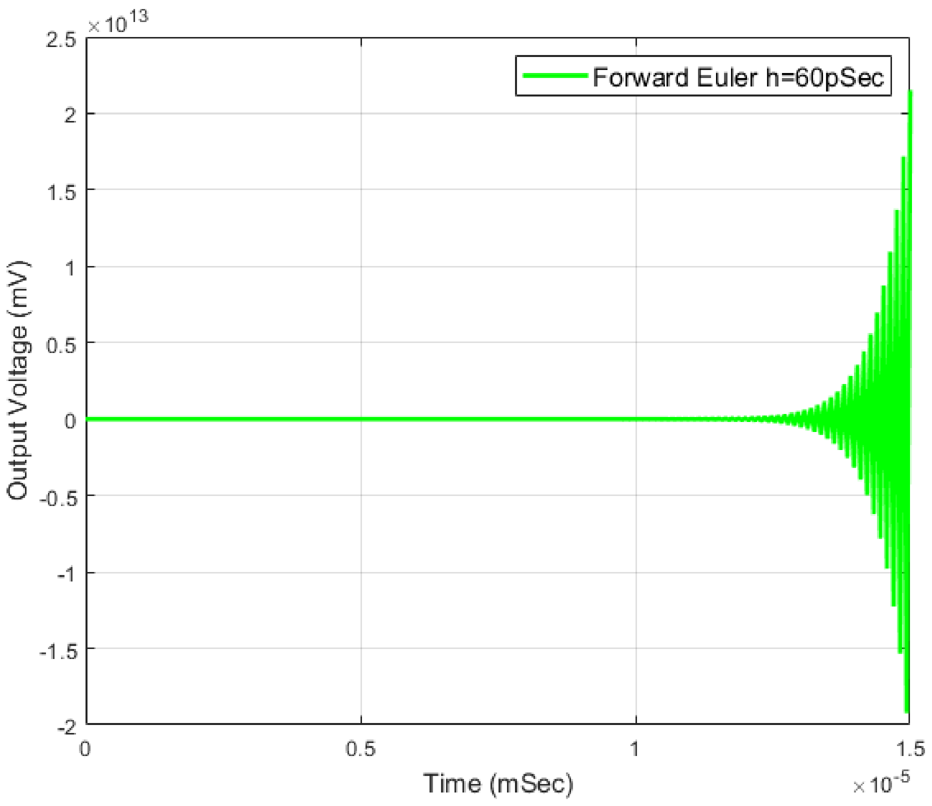


Fig 6. Result of FE when step size h=60 ps.

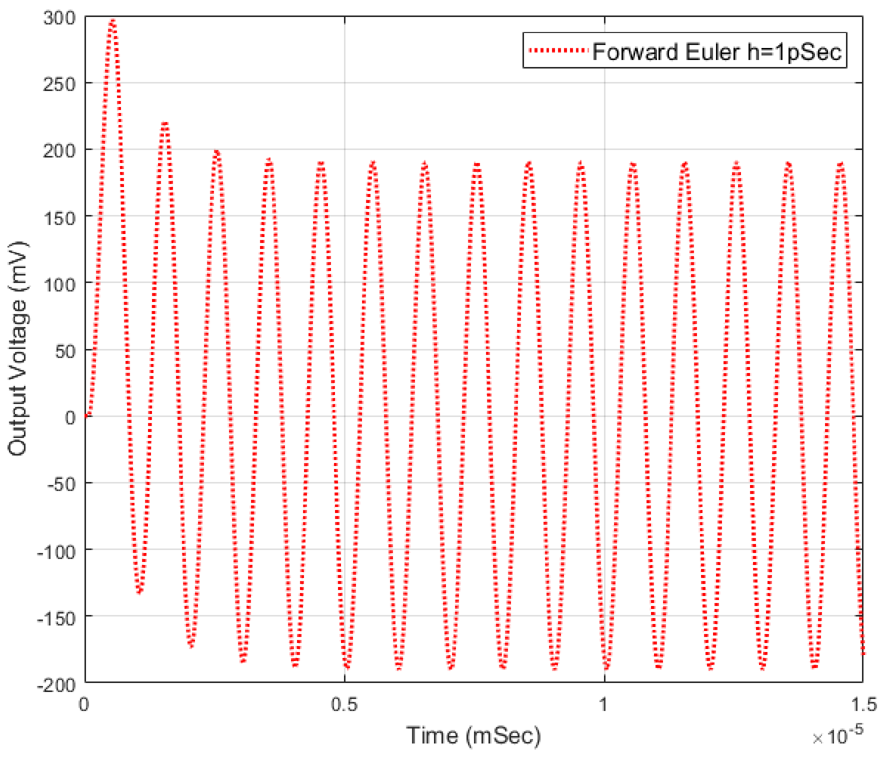


Fig 7. Result of FE when step size h=1 ps.

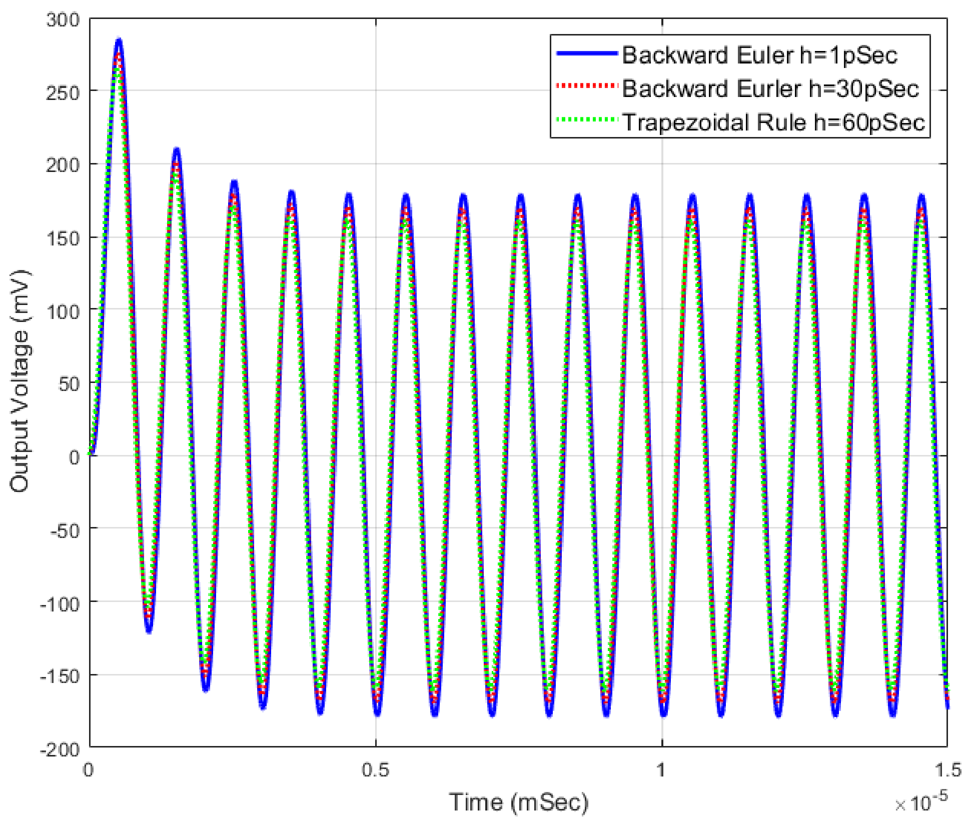
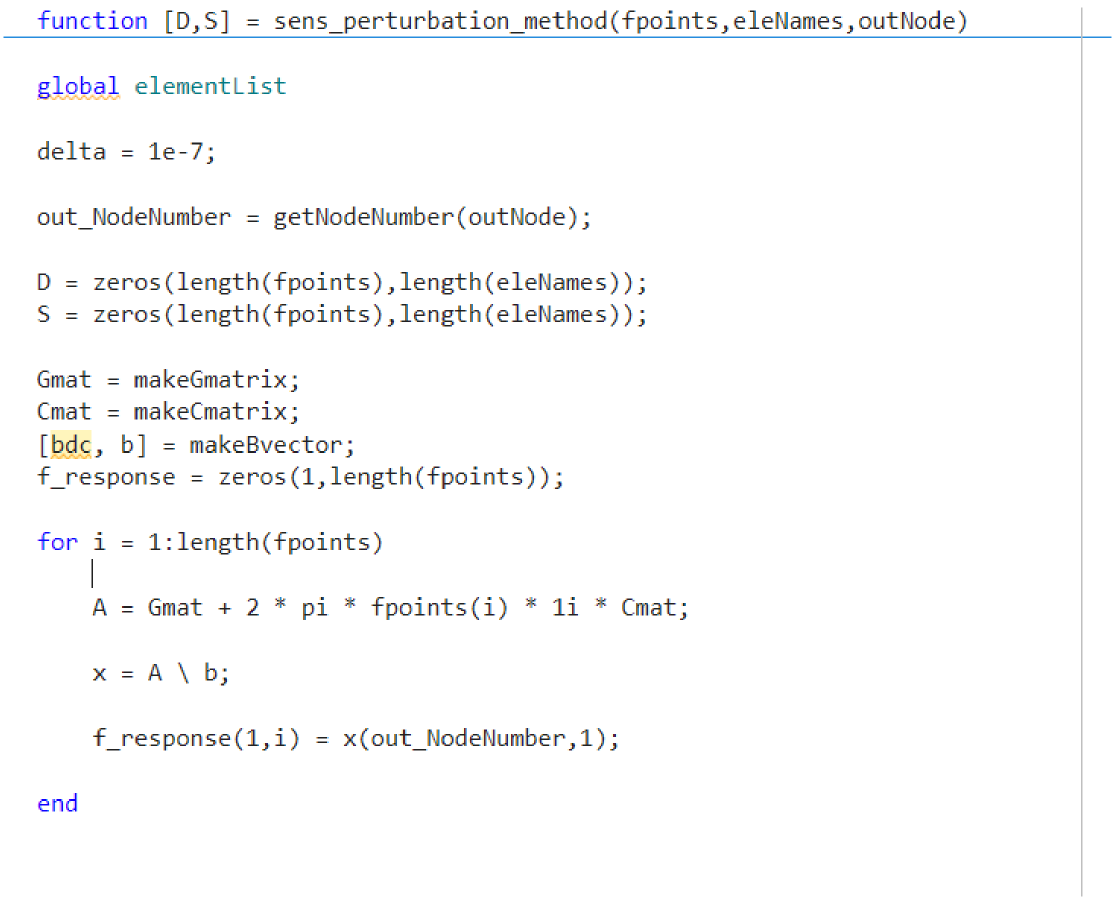


Fig 8. Other results of testbench q3.

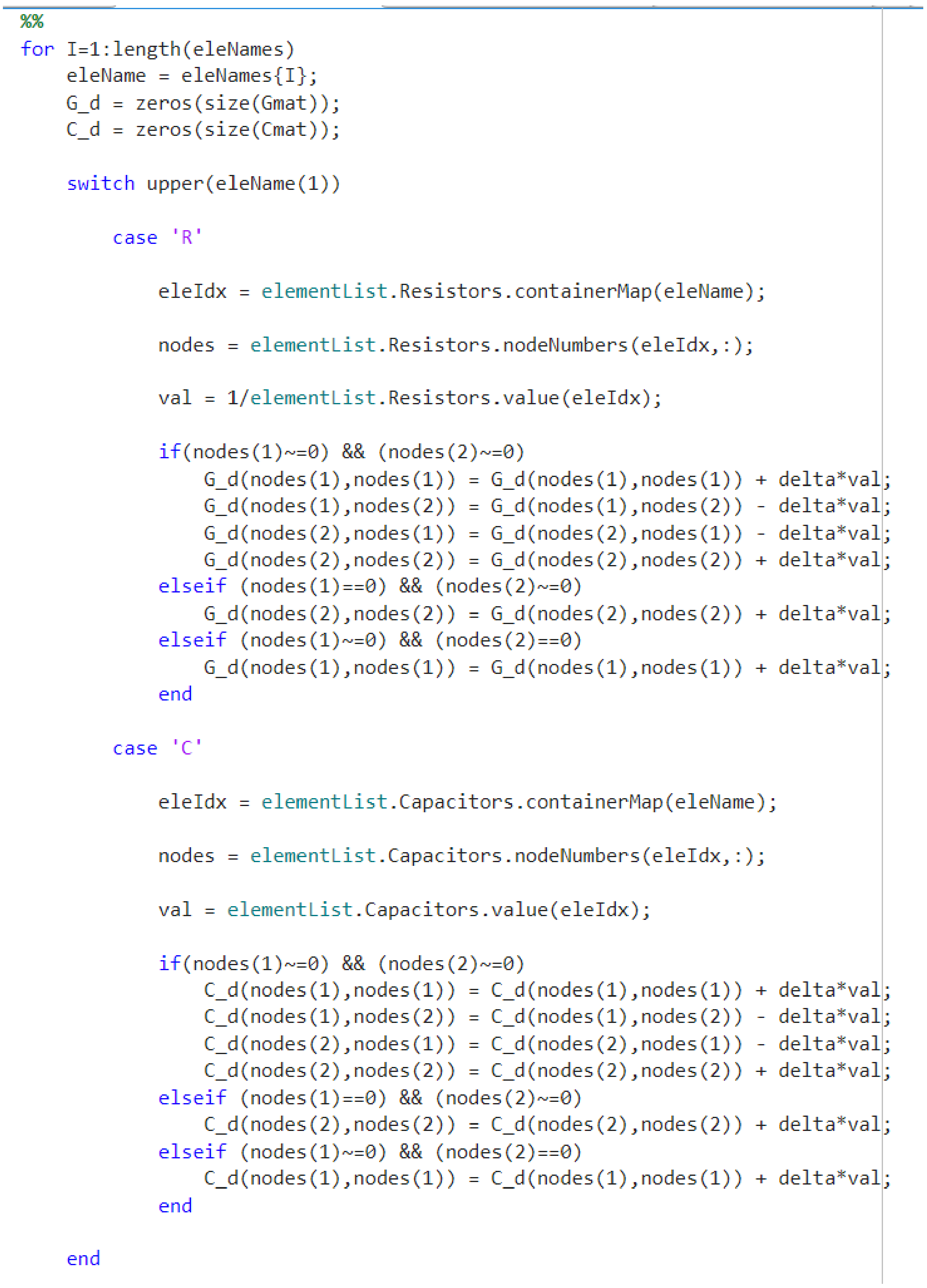
Fig 6 and fig 7 shows that for FE, when step size is too large, the output will diverge. Fig 8 shows that for BE and TR, convergence is not a problem to worry about: the output will always converge no matter how large the step size is.

By using MATLAB, the eigenvalues found are: -3.532e10, -2.3473e10, -1e10 and -0.1206e10. Since the stability condition for FE is $|1+h*\lambda|<1$, we could conclude that the stability condition for this circuit netlist is $0<h<5.66e\text{-}11$. This would explain the plots shown in fig 6 and fig 7, where 6e-11 is out of the stability boundary.

Q4.



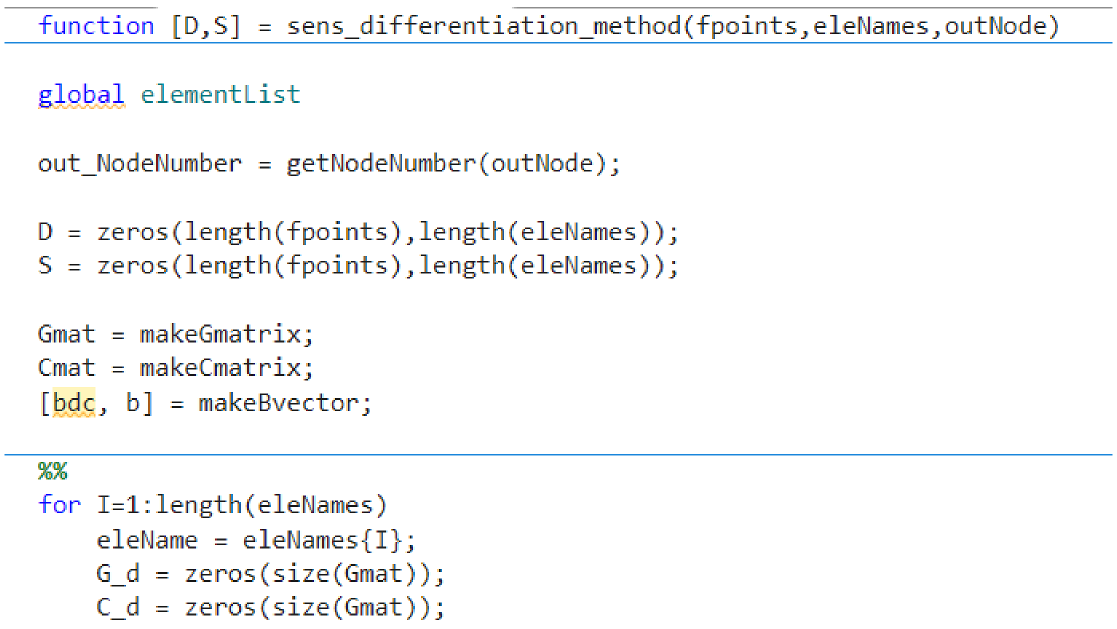
```
function [D,S] = sens_perturbation_method(fpoints,eleNames,outNode)

global elementList

delta = 1e-7;

out_NodeNumber = getNodeNumber(outNode);

D = zeros(length(fpoints),length(eleNames));
S = zeros(length(fpoints),length(eleNames));

Gmat = makeGmatrix;
Cmat = makeCmatrix;
[bdc, b] = makeBvector;
f_response = zeros(1,length(fpoints));

for i = 1:length(fpoints)
    
    A = Gmat + 2 * pi * fpoints(i) * 1i * Cmat;
    
    x = A \ b;
    
    f_response(1,i) = x(out_NodeNumber,1);
    
end
```



```
%%
for I=1:length(eleNames)
    eleName = eleNames{I};
    G_d = zeros(size(Gmat));
    C_d = zeros(size(Cmat));

    switch upper(eleName(1))

        case 'R'

            eleIdx = elementList.Resistors.containerMap(eleName);

            nodes = elementList.Resistors.nodeNumbers(eleIdx,:);

            val = 1/elementList.Resistors.value(eleIdx);

            if(nodes(1)~=0) && (nodes(2)~=0)
                G_d(nodes(1),nodes(1)) = G_d(nodes(1),nodes(1)) + delta*val;
                G_d(nodes(1),nodes(2)) = G_d(nodes(1),nodes(2)) - delta*val;
                G_d(nodes(2),nodes(1)) = G_d(nodes(2),nodes(1)) - delta*val;
                G_d(nodes(2),nodes(2)) = G_d(nodes(2),nodes(2)) + delta*val;
            elseif (nodes(1)==0) && (nodes(2)~=0)
                G_d(nodes(2),nodes(2)) = G_d(nodes(2),nodes(2)) + delta*val;
            elseif (nodes(1)~=0) && (nodes(2)==0)
                G_d(nodes(1),nodes(1)) = G_d(nodes(1),nodes(1)) + delta*val;
            end

        case 'C'

            eleIdx = elementList.Capacitors.containerMap(eleName);

            nodes = elementList.Capacitors.nodeNumbers(eleIdx,:);

            val = elementList.Capacitors.value(eleIdx);

            if(nodes(1)~=0) && (nodes(2)~=0)
                C_d(nodes(1),nodes(1)) = C_d(nodes(1),nodes(1)) + delta*val;
                C_d(nodes(1),nodes(2)) = C_d(nodes(1),nodes(2)) - delta*val;
                C_d(nodes(2),nodes(1)) = C_d(nodes(2),nodes(1)) - delta*val;
                C_d(nodes(2),nodes(2)) = C_d(nodes(2),nodes(2)) + delta*val;
            elseif (nodes(1)==0) && (nodes(2)~=0)
                C_d(nodes(2),nodes(2)) = C_d(nodes(2),nodes(2)) + delta*val;
            elseif (nodes(1)~=0) && (nodes(2)==0)
                C_d(nodes(1),nodes(1)) = C_d(nodes(1),nodes(1)) + delta*val;
            end

    end
    for i = 1:length(fpoints)

        A_new = (Gmat + G_d) + 2 * pi * fpoints(i) * 1i * (Cmat + C_d);

        x_new = A_new \ b;

        D(i,I) = (x_new(out_NodeNumber,1)-f_response(1,i))/(delta*val);
        S(i,I) = val/f_response(1,i)*D(i,I);

    end

end

end
```

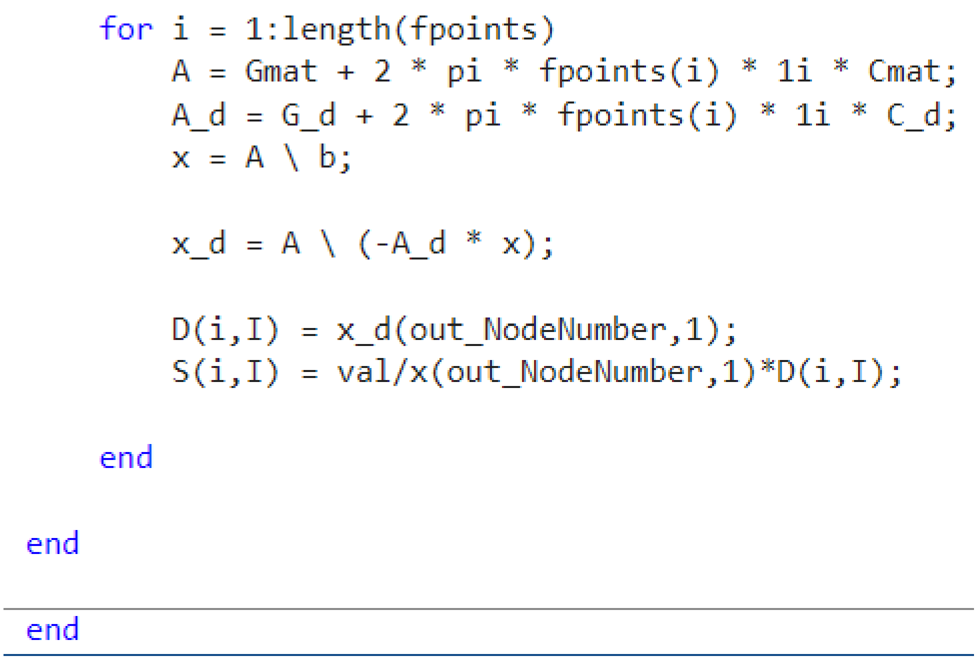
Fig 9. Code of sensitivity analysis using perturbation method.

Q5.



```
function [D,S] = sens_differentiation_method(fpoints,eleNames,outNode)

global elementList

out_NodeNumber = getNodeNumber(outNode);

D = zeros(length(fpoints),length(eleNames));
S = zeros(length(fpoints),length(eleNames));

Gmat = makeGmatrix;
Cmat = makeCmatrix;
[bdc, b] = makeBvector;

%%
for I=1:length(eleNames)
    eleName = eleNames{I};
    G_d = zeros(size(Gmat));
    C_d = zeros(size(Gmat));
```

```
switch upper(eleName(1))

    case 'R'
        eleIdx = elementList.Resistors.containerMap(eleName);

        nodes = elementList.Resistors.nodeNumbers(eleIdx,:);

        val = 1/elementList.Resistors.value(eleIdx);

        if(nodes(1)~=0) && (nodes(2)~=0)
            G_d(nodes(1),nodes(1)) = 1;
            G_d(nodes(1),nodes(2)) = -1;
            G_d(nodes(2),nodes(1)) = -1;
            G_d(nodes(2),nodes(2)) = 1;
        elseif (nodes(1)==0) && (nodes(2)~=0)
            G_d(nodes(2),nodes(2)) = 1;
        elseif (nodes(1)~=0) && (nodes(2)==0)
            G_d(nodes(1),nodes(1)) = 1;
        end

    case 'C'

        eleIdx = elementList.Capacitors.containerMap(eleName);

        nodes = elementList.Capacitors.nodeNumbers(eleIdx,:);

        val = elementList.Capacitors.value(eleIdx);

        if(nodes(1)~=0) && (nodes(2)~=0)
            C_d(nodes(1),nodes(1)) = 1;
            C_d(nodes(1),nodes(2)) = -1;
            C_d(nodes(2),nodes(1)) = -1;
            C_d(nodes(2),nodes(2)) = 1;
        elseif (nodes(1)==0) && (nodes(2)~=0)
            C_d(nodes(2),nodes(2)) = 1;
        elseif (nodes(1)~=0) && (nodes(2)==0)
            C_d(nodes(1),nodes(1)) = 1;
        end

end
```



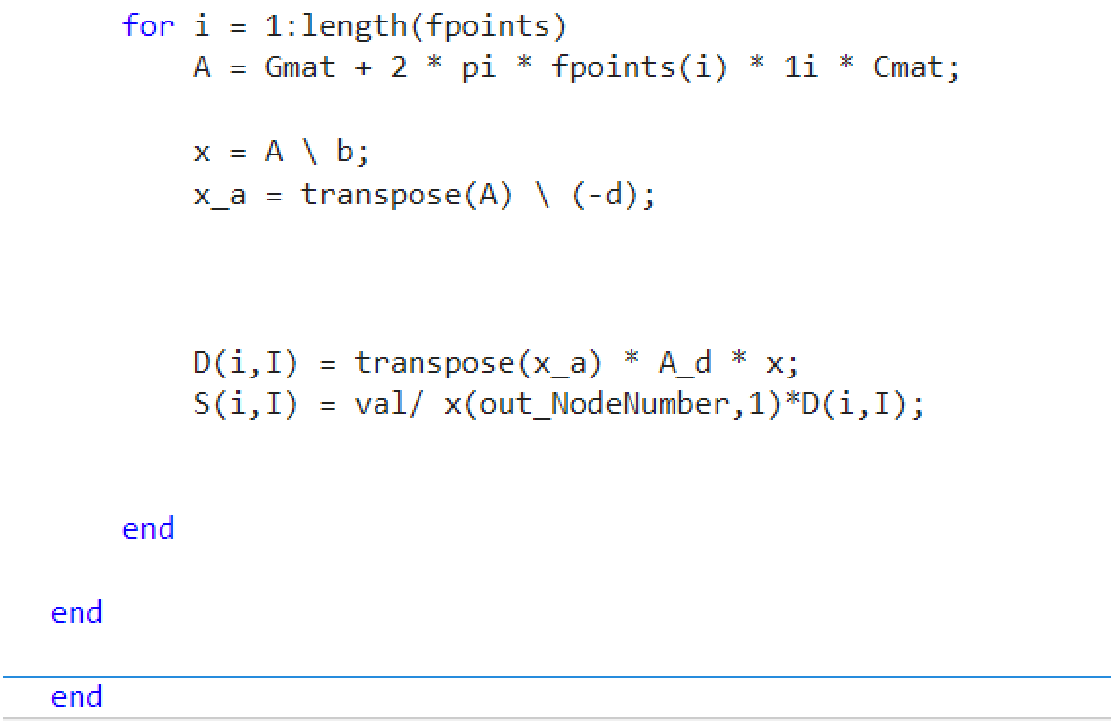
```
        for i = 1:length(fpoints)
            A = Gmat + 2 * pi * fpoints(i) * 1i * Cmat;
            A_d = G_d + 2 * pi * fpoints(i) * 1i * C_d;
            x = A \ b;

            x_d = A \ (-A_d * x);

            D(i,I) = x_d(out_NodeNumber,1);
            S(i,I) = val/x(out_NodeNumber,1)*D(i,I);

        end

    end

    end
```

Fig 10. Code of sensitivity analysis using differentiation method.

Q6.

```
function [D,S] = adjoint_sensitivity(fpoints,eleNames,outNode)

global elementList

%%
out_NodeNumber = getNodeNumber(outNode);

D = zeros(length(fpoints),length(eleNames));
S = zeros(length(fpoints),length(eleNames));

Gmat = makeGmatrix;
Cmat = makeCmatrix;
[bdc, b] = makeBvector;

d = zeros(size(Gmat,2),1);
d(out_NodeNumber,1) = 1;
```

```
%%
for I=1:length(eleNames)
    eleName = eleNames{I};
    A_d = zeros(size(Gmat));

    switch upper(eleName(1))

        case 'R'
            eleIdx = elementList.Resistors.containerMap(eleName);

            nodes = elementList.Resistors.nodeNumbers(eleIdx,:);

            val = elementList.Resistors.value(eleIdx);

            if(nodes(1)~=0) && (nodes(2)~=0)
                A_d(nodes(1),nodes(1)) = 1;
                A_d(nodes(1),nodes(2)) = -1;
                A_d(nodes(2),nodes(1)) = -1;
                A_d(nodes(2),nodes(2)) = 1;
            elseif (nodes(1)==0) && (nodes(2)~=0)
                A_d(nodes(2),nodes(2)) = 1;
            elseif (nodes(1)~=0) && (nodes(2)==0)
                A_d(nodes(1),nodes(1)) = 1;
            end

        case 'C'

            eleIdx = elementList.Capacitors.containerMap(eleName);

            nodes = elementList.Capacitors.nodeNumbers(eleIdx,:);

            val = elementList.Capacitors.value(eleIdx);

            if(nodes(1)~=0) && (nodes(2)~=0)
                A_d(nodes(1),nodes(1)) = 1;
                A_d(nodes(1),nodes(2)) = -1;
                A_d(nodes(2),nodes(1)) = -1;
                A_d(nodes(2),nodes(2)) = 1;
            elseif (nodes(1)==0) && (nodes(2)~=0)
                A_d(nodes(2),nodes(2)) = 1;
            elseif (nodes(1)~=0) && (nodes(2)==0)
                A_d(nodes(1),nodes(1)) = 1;
            end

    end
```



```
    for i = 1:length(fpoints)
        A = Gmat + 2 * pi * fpoints(i) * 1i * Cmat;

        x = A \ b;
        x_a = transpose(A) \ (-d);



        D(i,I) = transpose(x_a) * A_d * x;
        S(i,I) = val/ x(out_NodeNumber,1)*D(i,I);


    end

end

end
```

Fig 11. Code of sensitivity analysis using adjoint method.

Q7.

The absolute sensitivity of R1 and C1 are shown below in fig 12, 13, 14 and 15.

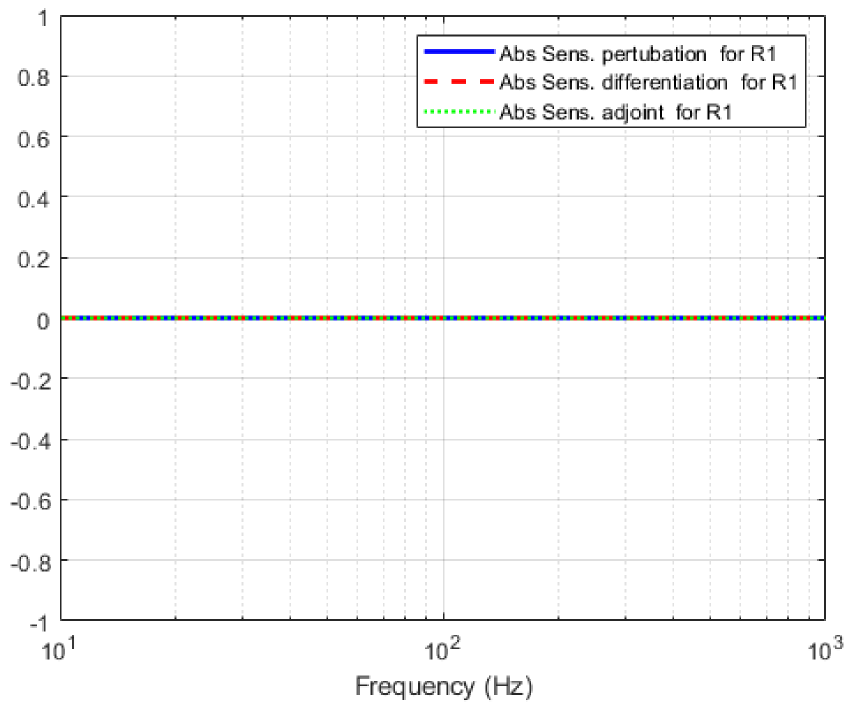


Fig 12. Absolute sensitivity of R1.

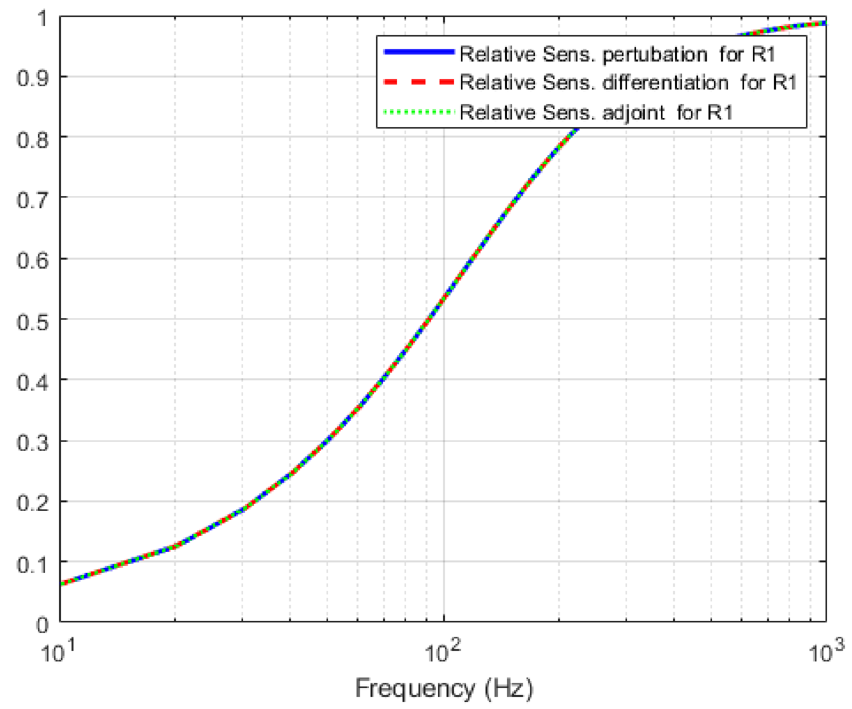


Fig 13. Relative sensitivity of R1.

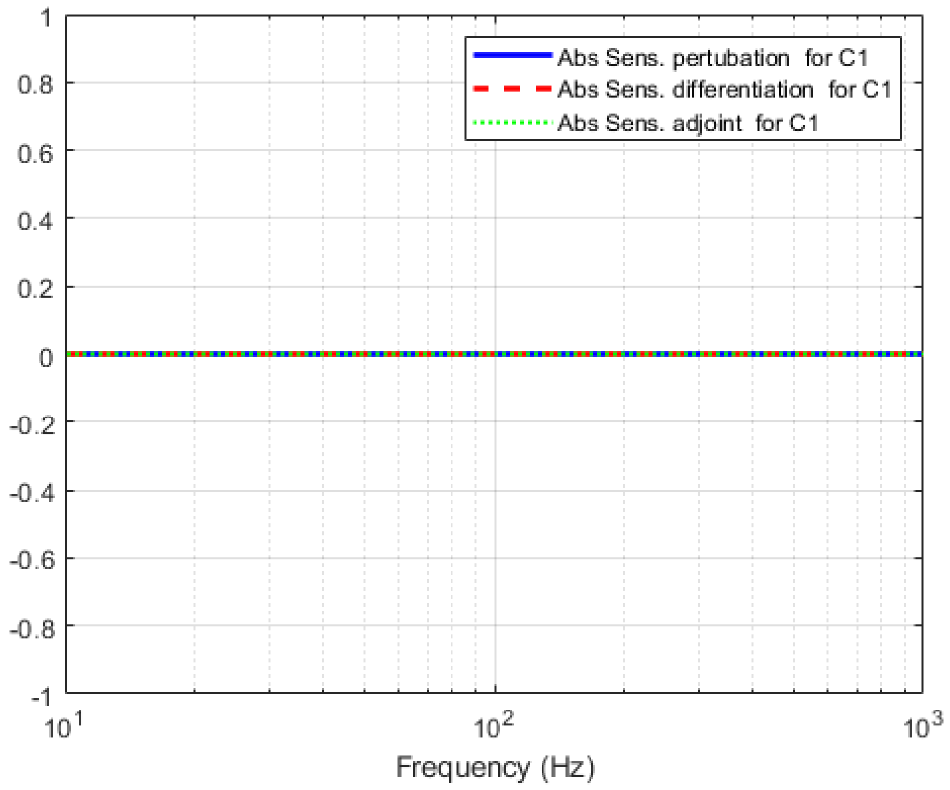


Fig 14. Absolute sensitivity of C1.

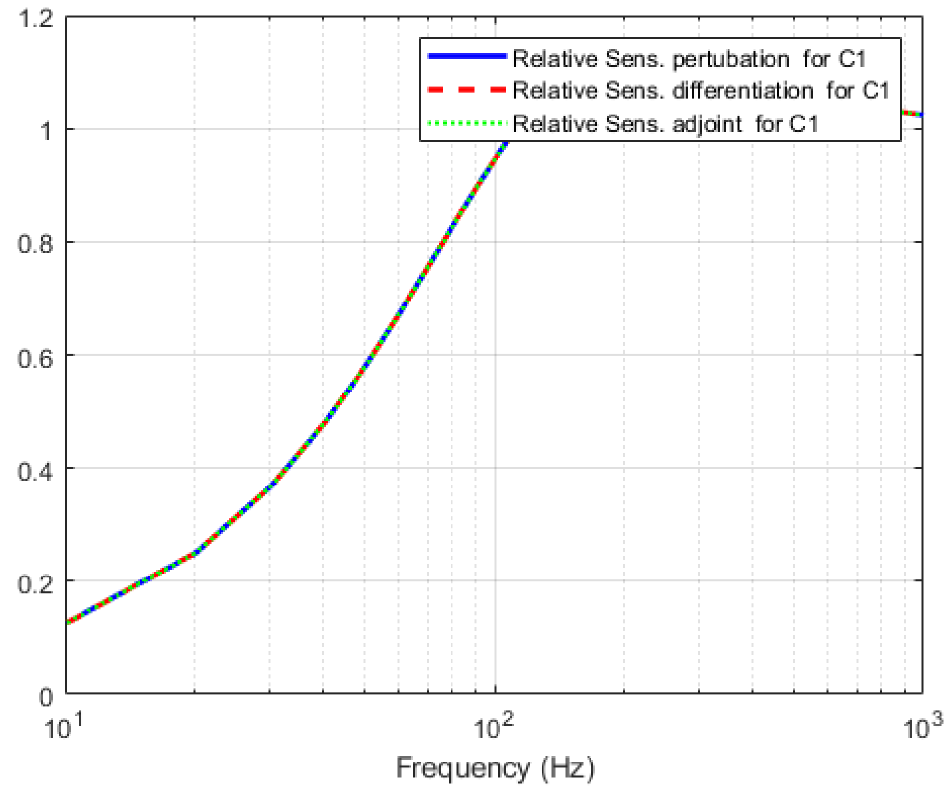


Fig 15. Relative sensitivity of C1.

The relative sensitivity of all other elements computed in 3 ways are shown below in fig 16, 17, 18 and 19.

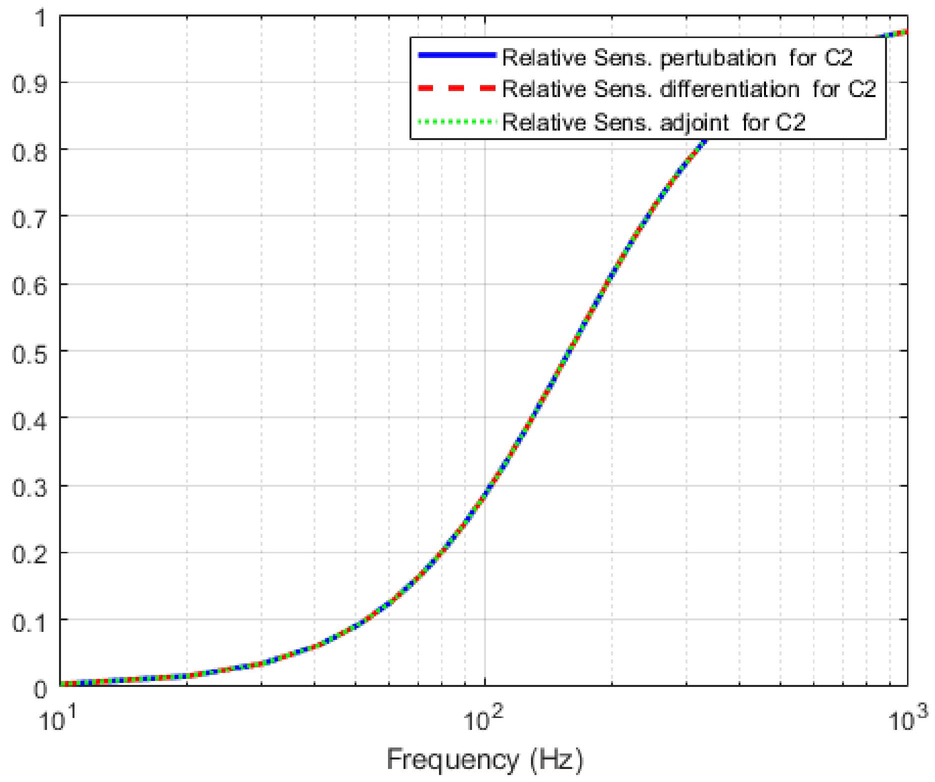


Fig 16. Relative sensitivity of C2.

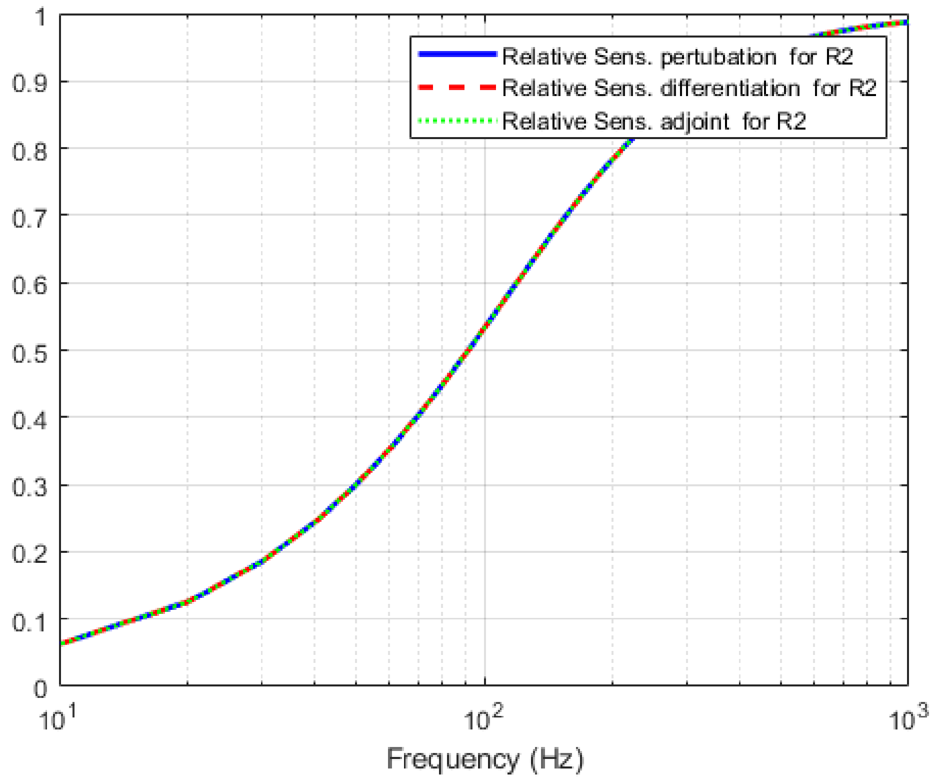


Fig 17. Relative sensitivity of R2.

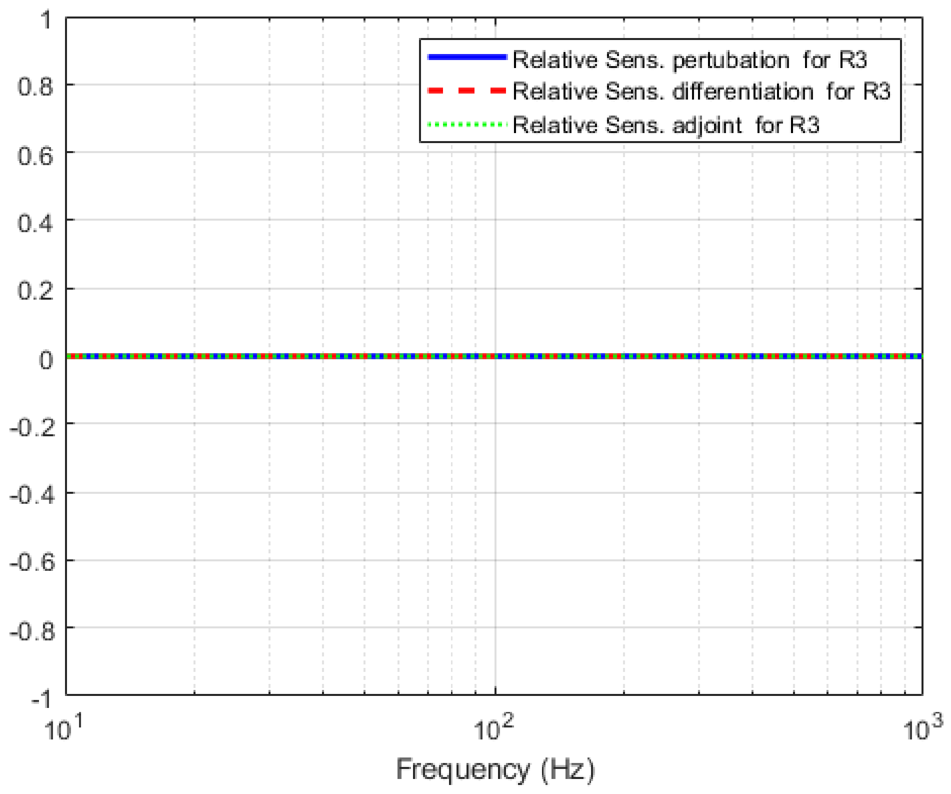


Fig 18. Relative sensitivity of R3.

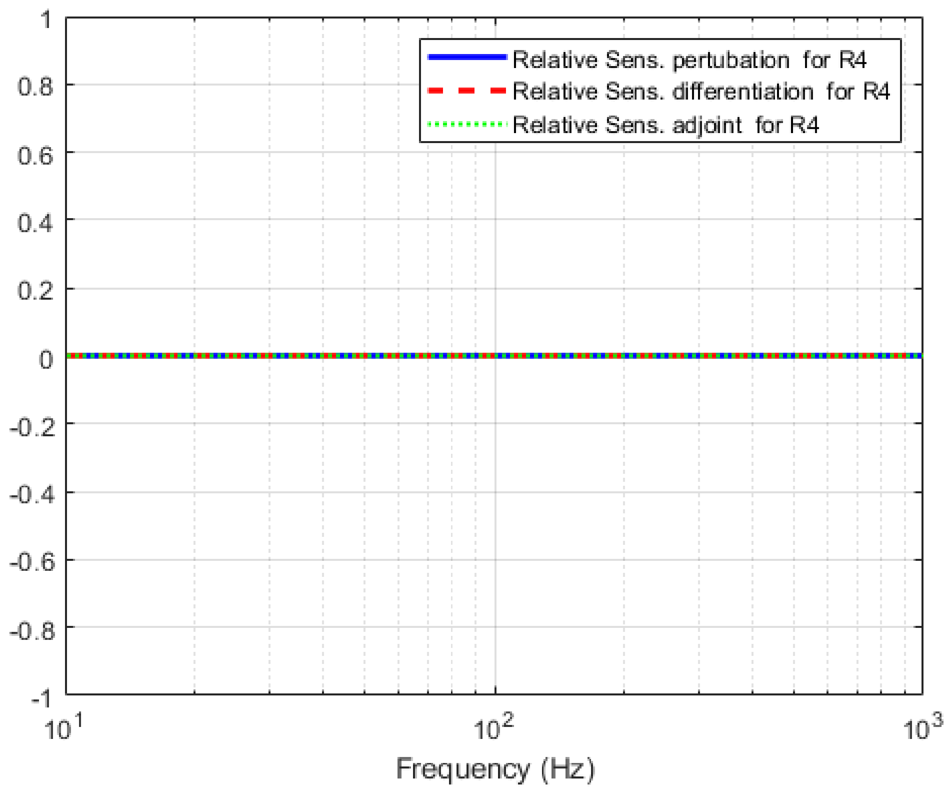


Fig 19. Relative sensitivity of R4.