A multiple input, multiple output model generator

Fredrik Bengtsson, Torsten Wik

Abstract

When new methods of design and analysis are introduced in the control engineering field, it is scientifically important to compare the new results with existing methods. Often this requires application of the methods on examples, and for this purpose benchmark processes are introduced. However, in many areas of control engineering research the number of examples is limited to a relatively few examples, and in particular so when multi-input multi-output (MIMO) systems are considered. For a thorough assessment of a method, however, as large number of relevant models as possible should be used. As a remedy we propose a framework for generating linear multiple input, multiple output models based on predefined system properties, such as model type, size, stability, time constants, delays etc.

1 Introduction

When analyzing new methods, it is common to demonstrate their benefits on various example systems. While this is a useful way to demonstrate a new method, it does not easily allow one to draw general conclusions of the strengths and limitations of the new method. To do this one would like to implement the method on a large number of systems, with varying properties. For single-input single-output system a batch of process models have been collected for such evaluations [3]. However to our knowledge, no such extensive batch is available for MIMO systems today. Here we present a MIMO model generator, coded in Matlab, which generates a multiple-input, multiple-output (MIMO) model according to user-defined parameters. This generator can be used to generate a large number of MIMO systems, which can be used to thoroughly test new methods, as well as to conduct statistical analysis on them.

2 MIMO model generator

The MIMO generator is implemented in the Matlab file `mimogen.m`. Within this file the user specifies the parameters of the MIMO system he or she wishes to generate, and then runs the file. A random MIMO system will be generated in the form of a transfer function matrix (TFM) according to the users instructions. The properties the user can specify are described below, and are presented along with their Matlab notations in Table 1.

2.1 Number of inputs and outputs

Here the user specifies the dimensions of the TFM. Both square and non-square TFMs can be generated.

2.2 Number of inputs affecting each output

This is a measure of the sparsity of the TFM. The user specifies the maximum and minimum number of inputs that can affect each output. This is equivalent to specifying the maximum and minimum number of non-zero elements in each row of the TFM. For each row the generator will randomly determine how many non-zero elements there are using a uniform probability distribution. The generator will ensure that the diagonal of the TFM always contains non-zero elements (to ensure that each input affects at least one output), while the remaining non-zero elements will be placed randomly on each row.

2.3 Transfer function order

Here the user specifies the minimum and maximum transfer function order. Each non-zero element in the TFM will be assigned an order between these two values with a uniform probability distribution. The user also specifies the minimum and maximum relative degree of the transfer functions. A number of zeros will then be assigned to each transfer function in the TFM so that they have a relative degree within the prescribed values. Also here a uniform probability distribution is used.

2.4 Properties of the transfer functions

The user can specify the span of the number of unstable poles, non-minimum phase zeros and purely imaginary pole pairs the system can have. The number of each is generated from their respective spans using a uniform probability function. The TFM will then place these poles and zeros randomly among the non-zero elements in the TFM. The user can also specify the number of elements in the TFM that has integration, double integration or differentiation. These will also be distributed randomly among the non-zero elements of the TFM, with steps being taken such that the same element does not have both integration and differentiation. After this, stable zeros and poles will be added to each transfer function in the TFM such that they get the order and relative degree as previously determined by the generator. For the poles the user specifies what percentage of them should be complex, with separate percentages for stable and unstable poles.

2.5 Static Gain

The user can specify the maximum static gain of the transfer functions. The minimum gain will always be set to 1. Each transfer function is assigned a static gain between these values using the following function:

$$K = K_{max}^{\zeta}$$

where K_{max} is the maximum gain. ζ is a uniformly distributed random variable between 0 and 1, generated individually for each transfer function. This probability function is chosen to ensure that low gains occur with some frequency, as with a uniform distribution very few gains close to the minimum gain would occur. For instance with a gain of 1-1000, statistically 90% of gains would occur between 100-1000 if a uniform distribution was chosen. With this distributions, statistically an equal amount of gains will occur between 100-1000, 10-100, and between 1-10. For integrators and derivatives, which always have a static gain of infinity or zero respectively, K is implemented as the static gain the transfer function would have without any integrators or derivatives.

2.6 Time constants

The user specifies a maximum and a minimum value of the time constants in the transfer functions. For non-complex poles the time constants are derived similarly to the gain, i.e

$$T = 10^{\log_{10}(T_{min})+\zeta(\log_{10}(T_{max})-\log_{10}(T_{min}))}$$

where T_{min} and T_{max} are minimum and maximum values of the time constants assigned by the user and ζ is a uniformly distributed random variable between 0 and 1.

The time constants of the zeros are also determined from stochastic distributions. Here, there are two options for the user. In the first they are decided with the same method as for the poles, with the user specifying the maximum and minimum values of the time constants. However determining the time constant of the zeros independently of the poles may cause issues as the impact of a zero in a transfer function is closely linked to the poles of the transfer function. Zeros which are considerably faster than the dominant pole generally have very little impact, while zeros which are considerably slower will cause extreme under- or overshoots [2]. Therefore, there is an second option to instead determine the time constant of the zeros based on the undershoot or overshoot they may cause. Here the user specifies a maximum undershoot and a maximum overshoot. Using only the stable components of each transfer function the time constant which causes the specified under- or overshoot is found using a golden section search [1] with the user defining the tolerance of a completed search (the maximum size of the interval which the time constant is found to be within). This time constant (T_{Zmax}) is used to determine the time constant of the zeros using the following expression:

$$T_{zero} = 10^{\log_{10}(\frac{T_{Zmax}}{Z_{minfact}})+\zeta(\log_{10}(T_{Zmax})-\log_{10}(\frac{T_{Zmax}}{Z_{minfact}}))}$$

where $Z_{minfact}$ is a factor specified by the user and ζ is a uniformly distributed random variable between 0 and 1. For transfer functions which lack stable dynamics, the time constants of the zeros will be determined in the same way as the time constants of the poles.

For complex poles the damping is generated by a uniform probability function, while the undamped natural frequency is determined with the same distribution as the gain and time constants, with minimum and maximum corresponding to the pole's time constants.

In many real MIMO-systems, the same time constant appears in many transfer functions in the TFM because they share dynamics. With this in mind, we added an option of "distinct time constants" in the MIMO model generator to achieve this characteristic. If this is enabled, the generator randomly creates a set of time constants using the methods described previously. Whenever a time constant is needed, the generator will randomly pick a time constant from this set. Separate sets are used for stable poles, unstable poles, complex stable poles, complex unstable poles, minimum phase zeros and non-minimum phase zeros. The size of each set is generated with the user specifying a percentage, this percentage is then multiplied with the total number of poles or zeros of the specific type to get the size of the set. So for instance if there are a total of 17 stable poles in the system and the user specifies 60 percent, then the size of the set for stable poles will be $\lceil 17 \cdot 0.6 \rceil = 11$ (note that we always round up when calculating the size of the set). In this case the zeros are always generated by the user specifying the maximum and minimum values of the time constants and not based on the specified under- and overshoot.

2.7 Delay

The user specifies the percentage of the elements in the TFM that contains delays. The MIMO model generator then randomly adds delays to elements in the TFM until the specified percentage is reached. The size of the delay is generated separately for each delay element with a uniform probability function, whose bounds are specified by the user. The user may also chose if the delay should be expressed with a Padé approximation or not. If this is the case the user also decides the degree of the Padé approximation.

Recommended use

The MIMO model generator allows for the construction of large amounts of linear MIMO systems. Hence it can be used to conduct statistical analysis on methods that are used on such systems. Moreover, it can be used to evaluate different methods of control system design to determine which types of TFM they work well for. This can be expanded to establish a benchmark for which new methods can be tested.

References

[1] Lars-Christer Böiers. *Mathematical Methods of Optimization*. Studentlitteratur AB, 2010. Lärobok i matematik.

[2] Graham C. Goodwin, Stefan F. Graebe, and Mario E. Salgado. *Control System Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[3] K.J. Åström and T. Hägglund. Benchmark systems for pid control. *IFAC Proceedings Volumes*, 33(4):165 – 166, 2000. IFAC Workshop on Digital Control: Past, Present and Future of PID Control, Terrassa, Spain, 5-7 April 2000.

Table 1: User defined parameters to the MIMO generator and their default values

Parameter	Default value	Mathematical notation	Matlab notation
Size			
Number of inputs	5	n_u	n_u
Number of outputs	5	n_y	n_y
Minimum number of inputs affecting each output	$\lceil 1/4 \times n_u \rceil$	m_{min}	m_min
Maximum number of inputs affecting each output	$\lceil 3/4 \times n_u \rceil$	m_{max}	m_max
Minimum transfer function order	1	n_{min}	n_min
Maximum transfer function order	2	n_{max}	n_max
Minimum relative degree	1	r_{min}	r_min
Maximum relative degree	n_{max}	r_{max}	r_max
Poles and Zeros in the entire system			
Maximum number of unstable poles	0	$N_{USP_{max}}$	N_USP_max
Minimum number of unstable poles	0	$N_{USP_{min}}$	N_USP_min
Maximum number of purely imaginary pole pairs	0	$N_{MSP_{max}}$	N_MSP_max
Minimum number of purely imaginary pole pairs	0	$N_{MSP_{min}}$	N_MSP_min
Percentage of unstable poles which are complex	0	γ_{CUSP}	gamma_CUSP
Percentage of stable poles which are complex	20	γ_{CSP}	gamma_CSP
Percentage of transfer functions with single integrators	0	γ_I	gamma_I
Percentage of transfer functions with double integrators	0	γ_{2I}	gamma_2I
Percentage of transfer functions with derivatives	0	γ_D	gamma_D
Maximum number of non-minimum phase zeros	2	$N_{NMPZ_{max}}$	N_NMPZ_max
Minimum number of non-minimum phase zeros	0	$N_{NMPZ_{min}}$	N_NMPZ_min
Dynamics			
Maximum static gain	10	K_{max}	K_max
Minimum pole time constant	1	T_{min}	T_min
Maximum pole time constant	10	T_{max}	T_max
Minimum damping for complex poles	0.1	ζ_{min}	zeta_min
Minimum zero time constant	T_{min}	T_{zmin}	Tz_min
Maximum zero time constant	T_{max}	T_{zmax}	Tz_max
Distinct time constants	false	Dis_T	Dis_T
Percentage used to determine number of distinct stable poles	60	$P_{T_{SP}}$	P_T_SP
Percentage used to determine number of distinct unstable poles	60	$P_{T_{USP}}$	P_T_USP
Percentage used to determine number of distinct purely imaginary pole pairs	60	$P_{T_{MSP}}$	P_T_MSP
Percentage used to determine number of distinct complex stable pole pairs	60	$P_{T_{CSP}}$	P_T_CSP
Percentage used to determine number of distinct complex unstable pole pairs	60	$P_{T_{CUSP}}$	P_T_CUSP
Percentage used to determine number of distinct minimum phase zeros	60	$P_{T_{MPZ}}$	P_T_MPZ
Percentage used to determine number of distinct non-minimum phase zeros	60	$P_{T_{NMPZ}}$	P_T_NMPZ
Basing zeros' time constants on poles when possible	true	$Z_{TFbased}$	Z_TFbased
Maximum overshoot percentage	20	OS_{max}	OS_max
Maximum undershoot percentage	30	US_{max}	US_max
Tolerance when determining overshoot/undershoot	0.01	Tol	Tol
Factor used to determine minimum time constant	100	$Z_{Tminfact}$	Z_Tminfact
Delay			
Percentage of transfer functions with delay	20	γ_L	gamma_L
Minimum Delay	0	L_{min}	L_min
Maximum Delay	T_{max}	L_{max}	L_max
Padé approximation order	2	$padeOrder$	padeOrder

Appendix 1-Matlab code

Listing 1: Matlab code for MIMO generator

```
function [ TFMfinal ] = mimogen()
%%%%%%%%%%%%Insert parameter values here%%%%%%%%%%%%%%%%%
n_u = 5; % # inputs
n_y = 5; % # outputs

m_min = ceil(n_u /4);    % Min # inputs affecting each output
m_max = ceil(3*n_u /4);  % Max # inputs affecting each output
n_min = 1;               % Min transfer function order
n_max = 2;               % Max transfer function order
r_min = 1;               % Min relative degree
r_max = n_max;           % Max relative degree




% Dynamics
K_max=100; %maximum static gain

T_min = 1;               % Shortest time constant
T_max = 10;              % Largest time constant
zeta_min=0.1;            % Minimum damping for complex poles

%Zero time constants:

TZ_min=1;
TZ_max=10;



Dis_T=false;             % Distinct time constants

% Number of distinct time constants for each type(as a percentage of the
% total number of time constants of each type)

P_T_SP=60;          % Stable Poles
P_T_USP=60;         % Unstable poles
P_T_MSP=60;         % Purely imaginary poles(number of pairs)
P_T_CSP=60;         % Stable complex poles(number of pairs)
P_T_CUSP=60;        % Unstable complex poles(number of pairs)
P_T_MPZ=60;         % Minimum phase zeros
P_T_NMPZ=60;        % Nonminimum phase zeros

Z_TFbased=true;     % Base zeros time constants on poles
                    % over-/undershoot when possible

OS_max=10;          % Maximum overshoot
US_max=25;          % Maximum undershoot
Tol=0.01;           % Precision when determining overshoot/undershoot

Z_Tminfact=100;     %Factor used to determine minimum time constant
```

```
% Poles and zeros
N_USP_min=0;        % Maximum number of unstable poles
N_USP_max=0;        % Minimum number of unstable poles

N_MSP_min=0;        % Maximum number of purely imaginary pole pairs
N_MSP_max=0;        % Minimum number of purely imaginary pole pairs

gamma_CUSP=0;       % Percentage of unstable poles which are complex
gamma_CSP=20;       % Percentage of stable poles which are complex

gamma_I  = 0;       % Percentage of TFs which have integration
gamma_2I = 0;       % Percentage of TFs which have double integration
gamma_D  = 0;       % Percentage of TFs which have differentiation


N_NMPZ_max=2;       % Maximum number of non-minimum phase zeros
N_NMPZ_min=0;       % Minimum number of non-minimum phase zeros


%Delays
gamma_L=20;         % Percentage of Tfs having time delays
L_max=0.5;          % Max delay
L_min=0;            % Min delay
padeOrder=2;        % Order of Padé approximation of system,
                    % values less then one mean no pade approximation
                    % is done



%%%%%%%%%%%%%%%%other parameters automatically determined here

% Randomising number of purely imaginary poles
N_MSP=randi([N_MSP_min N_MSP_max]);
N_MSP=N_MSP*2;                      %Pairs of poles

% Randomising number of unstable poles
N_USP=randi([N_USP_min N_USP_max]);

% Randomising number of non minimum phase zeros
N_NMPZ=randi([N_NMPZ_min N_NMPZ_max]);

omega_Umin=1/T_max;       % Natural frequency unstable complex poles
omega_Umax=1/T_min;


omega_Smin=1/T_max;      % Natural frequency stable complex poles
omega_Smax=1/T_min;


%-------------------------------------------------
if gamma_I+gamma_2I+gamma_D>100
    error('gamma_I+gamma_2I+gamma_D must be less than 100%!')
end
```

```
%% Inputs are placed here

jNotOK = 1;              % There are unused inputs
while jNotOK == 1
    % Calculations
    I = zeros(n_y,n_u);      % Index matrix
    for i=1:n_y
        % number of inputs affecting output j:
        m(i)   = m_min-1+randi(m_max-m_min+1,1); %m \in U(m_min,m_max)
        % random choise of inputs affecting output j:
        Irand  = randperm(n_u-1);
        Ii     = Irand(1:m(i)-1);
        Ii1row = zeros(1,n_u-1);
        Ii1row(Ii)= 1;
        if i==1
            I(1,1)     = 1;              % Non-zero UL diagonal entries
            I(i,2:end) = Ii1row;
        else
            I(i,1:i-1) = Ii1row(1:i-1);
            I(i,i)   = 1;                % Non-zero UL diagonal entries
            I(i,i+1:end) = Ii1row(i:end);
        end
    end
    % Check that all inputs are used
    jNotOK = 0;
    for j=1:n_u
        if sum(I(:,j))==0
            jNotOK = 1;      % There are unused inputs
        end
    end
end                      % All inputs are used

% Index matrix -> TFM (Transfer Function Matrix):
TFM{1} = I;
% Pole order n (random between n_min & n_max -> TFM
TFM{2} = I.*(n_min-1 + randi(n_max-n_min+1,n_y,n_u)); % denominator
                                                      % order

% Zero polynomial order
TFM{3}  = zeros(n_y,n_u);

for i=1:n_y
    for j=1:n_u
        if I(i,j) == 1;
            % Relative degree:
            if TFM{2}(i,j)>r_min
                r_order = I(i,j)*(r_min-1+randi(min(r_max,TFM{2}(i,j))-r_min
                   +1,1));
                TFM{3}(i,j) = TFM{2}(i,j)-r_order;   % numerator order
            else
                TFM{3}(i,j) = 0;
            end
```

```
        end
    end
end
% Integrations
N_I  = floor(0.01*gamma_I*sum(sum(I)));
N_2I = floor(0.01*gamma_2I*sum(sum(I)));
N_D  = floor(0.01*gamma_D*sum(sum(I)));


% Total number of non zero poles

N_Ptot = sum(sum(TFM{2}))-N_I-N_2I*2;

% Total number of non zero zeros

N_Ztot=sum(sum(TFM{3}))-N_D;
%% Adding double integrators
% Transfer functions which can have a double integrator:
p2I=TFM{2}>1;

Nump2I=sum(sum(p2I));%Number of possible places for double integrators

if Nump2I<N_2I
    disp('Insufficient second order systems to add double integrators')
    N_2I=Nump2I;
end

p2Iperm= randperm(Nump2I); % Randomise place order for
                           % double integrators
p2s=p2Iperm(1:N_2I);
p2Index=find(TFM{2}>1);
I2s= zeros(n_y,n_u);
for k=1:length(p2s)
    I2s(p2Index(p2s(k)))=1;
end




% Remove used poles:
TFM{2}=TFM{2}-I2s*2;

%% Placing complex stable poles

%Number of stable poles
N_SP=N_Ptot-N_USP-N_MSP; %Number of stable poles
N_CSP=floor(N_SP*gamma_CSP/100/2); %Number of complex stable poles

[CSPs,err]=TFplaceGen(TFM{2},N_CSP,2); %Places the poles
TFM{2}=TFM{2}-CSPs*2;
if err
    disp('Insufficient second order systems to add the requested complex stable
        poles, added maximum amount')
```

```
end




%% Placing purely imaginary poles:

[MSPs,err]=TFplaceGen(TFM{2},N_MSP/2,2);
TFM{2}=TFM{2}-MSPs*2;
if err
    disp('Insufficient second order systems to add the requested marginally
       stable poles, added maximum amount')
end




%% Placing Unstable complex poles


N_UCP=floor(N_USP*gamma_CUSP/100/2); %Number of complex unstable pole pairs
[UCPs,err]=TFplaceGen(TFM{2},N_UCP,2); %places the complex unstable pole pairs
TFM{2}=TFM{2}-UCPs*2; %Removes used poles
if err
    disp('Insufficient second order systems to add the requested unstable
       complex poles, added maximum amount')
end

%% Derivatives

p1D=(TFM{3}>0)-(TFM{3}>0).*I2s; % Ensure that double integrators and
                                % derivatives are not added to the
                                % same TF
Nump1D=sum(sum(p1D));
if Nump1D<N_D
    disp('Insufficient first order numerators to add D parts')
    N_D=Nump1D;
end
p1Dperm= randperm(Nump1D);
p1s=p1Dperm(1:N_D);
p1Dndex=find(p1D);
D1s= zeros(n_y,n_u);
for k=1:length(p1s)
    D1s(p1Dndex(p1s(k)))=1;
end

% remove used zeros:
TFM{3}=TFM{3}-D1s;

%% Single integrator
% Transfer functions which can have a single integrator:
p1I=(TFM{2}>0)-(TFM{2}>0).*I2s-(TFM{2}>0).*D1s;
```

```
Nump1I=sum(sum(p1I));
if Nump1I<N_I
    disp('Insufficient first order systems to add integrators')
    N_I=Nump1I;
end
p1Iperm= randperm(Nump1I);
p1s=p1Iperm(1:N_I);
p1Index=find(p1I);
I1s= zeros(n_y,n_u);
for k=1:length(p1s)
    I1s(p1Index(p1s(k)))=1;
end

% remove used poles
TFM{2}=TFM{2}-I1s;

%% Non mimimum phase zeros:

[NMPZs,err]=TFplaceGen(TFM{3},N_NMPZ,1);
TFM{3}=TFM{3}-NMPZs*1;%Removing used zeros
if err
    disp('Insufficient zeros to add the requested non minimum phase zeros,
       added maximum amount')
end

% Minimum phase zeros
N_MPZ=N_Ztot-N_NMPZ;
MPZs=TFM{3}; %All remaining zeros are minimum phase

%% Unstable real poles
N_USPR=N_USP-N_UCP*2;

[USPs,err]=TFplaceGen(TFM{2},N_USPR,1);
TFM{2}=TFM{2}-USPs*1;
if err
    disp('Insufficient poles to add the requested unstable poles, added maximum
        amount')
end

% Remaining poles are stable:
SPs=TFM{2};

%% Placing time delays:

N_TD  = floor(0.01*gamma_L*sum(sum(I)));

p1TD=TFM{1};

Nump1TD=sum(sum(p1TD));
p1Dperm= randperm(Nump1TD);
p1s=p1Dperm(1:N_TD);
p1Index=find(p1TD);
TDs= zeros(n_y,n_u);
```

```
for k=1:length(p1s)
    TDs(p1Index(p1s(k)))=1;
end


%% GENERATING TRANSFER FUNCTIONS

s=tf('s');


%Implementing gain, using a logarithmic probability function

TFMfinal=TFM{1}.*K_max.^rand(n_y,n_u);


%Generating distinct time constants



%For poles
Tsak_max=log10(T_max);
Tsak_min=log10(T_min);

%Complex poles:
Omegasak_max=log10(omega_Smax);
Omegasak_min=log10(omega_Smin);

OmegaUsak_max=log10(omega_Umax);
OmegaUsak_min=log10(omega_Umin);

%Zeros
TZsak_min=log10(TZ_min);
TZsak_max=log10(TZ_max);

%calculating the number of each type
Num_T_SP=ceil(N_SP*P_T_SP/100);        % Stable Poles
Num_T_USPR=ceil(N_USPR*P_T_USP/100);   % Unstable real poles
Num_T_MSP=ceil(N_MSP*P_T_MSP/100);     % Purely imaginary poles
                                       % (number of pairs)
Num_T_CSP=ceil(N_CSP*P_T_CSP/100);     % Stable complex poles
                                       % (number of pairs)
Num_T_CUSP=ceil(N_UCP*P_T_CUSP/100);   % Unstable complex poles
                                       % (number of pairs)
Num_T_MPZ=ceil(N_MPZ*P_T_MPZ/100);     % Minimum phase zeros
Num_T_NMPZ=ceil(N_NMPZ*P_T_NMPZ/100); % Nonminimum phase zeros

if Dis_T
    %Sets for distinct time constants

    T_DisS=zeros(Num_T_SP,1); %Stable poles
    T_DisU=zeros(Num_T_USPR,1); %Unstable poles
    W_DisCS=zeros(Num_T_CSP,1);%Stable complex poles natural frequency
    z_DisCS=zeros(Num_T_CSP,1);%Stable complex poles dambing
    W_DisMS=zeros(Num_T_MSP,1);%Purely imaginary poles natural frequency
```

```
    W_DisCU=zeros(Num_T_CUSP,1);%Unstable complex poles natural frequency
    z_DisCU=zeros(Num_T_CUSP,1);%Unstable complex poles damping

    Zmp_Dis=zeros(Num_T_MPZ,1); %Minimum phase zeros
    Znmp_Dis=zeros(Num_T_MPZ,1);%Non-Minimum phase zeros

    for k=1:Num_T_SP
        TCs=10.^(Tsak_min+(Tsak_max-Tsak_min)*rand(1));
        T_DisS(k)=TCs;
    end

    for k=1:Num_T_USPR
        TCs=10.^(Tsak_min+(Tsak_max-Tsak_min)*rand(1));
        T_DisU(k)=TCs;
    end

    for k=1:Num_T_CSP
        Ws=10.^(Omegasak_min+(Omegasak_max-Omegasak_min)*rand(1));
        Zetas=zeta_min+(1-zeta_min)*rand(1);
        W_DisCS(k)=Ws;
        z_DisCS(k)=Zetas;
    end

    for k=1:Num_T_MSP
        Ws=10.^(Omegasak_min+(Omegasak_max-Omegasak_min)*rand(1));
        W_DisMS(k)=Ws;
    end

    for k=1:Num_T_CUSP
        Ws=10.^(OmegaUsak_min+(OmegaUsak_max-OmegaUsak_min)*rand(1));
        Zetas=zeta_min+(1-zeta_min)*rand(1);
        W_DisCU(k)=Ws;
        z_DisCU(k)=Zetas;
    end

    for k=1:Num_T_MPZ
        Zmp_Dis(k)=10.^(TZsak_min+(TZsak_max-TZsak_min).*rand(1));
    end

    for k=1:Num_T_NMPZ
        Znmp_Dis(k)=10.^(TZsak_min+(TZsak_max-TZsak_min).*rand(1));
    end
end



%Generating first order stable poles:
if sum(sum(SPs))
    go=1;
else
    go=0;
end
SPstemp=SPs;
```

```
while(go)
    poleM=SPstemp>0;
    SPstemp=SPstemp-poleM;
    %Generating time constants:
    if Dis_T
        for i=1:n_y
            for j=1:n_u
                TCs(i,j)=T_DisS(randi(length(T_DisS)));
            end
        end
    else
        TCs=10.^(Tsak_min+(Tsak_max-Tsak_min)*rand(n_y,n_u));
    end

    TFMtemp=poleM.*TCs.*s+1;
    TFMfinal=TFMfinal.*tfinverter(TFMtemp);
    if sum(sum(SPstemp))==0
        go=0;
    end
end

%Generating complex stable poles

go=1;
CSPstemp=CSPs;

while(go)
    poleM=CSPstemp>0;
    CSPstemp=CSPstemp-poleM;
    if Dis_T
        for i=1:n_y
            for j=1:n_u
                k=randi(length(W_DisCS));
                TCs(i,j)=W_DisCS(k);
                Zetas(i,j)=z_DisCS(k);
            end
        end
    else
        TCs=10.^(Omegasak_min+(Omegasak_max-Omegasak_min)*rand(n_y,n_u));
        Zetas=zeta_min+(1-zeta_min)*rand(n_y,n_u);
    end

    TFMtemp=poleM./(TCs).^2.*s^2+2*Zetas./TCs.*poleM.*s+1;
    TFMfinal=TFMfinal.*tfinverter(TFMtemp);
    if sum(sum(CSPstemp))==0
        go=0;
    end
end

%adding minimumphase zeros
go=1;
MPZstemp=MPZs;
TFMtemp=TFMfinal;
if Dis_T==false&&Z_TFbased==true
```

```
    MPZtemp2=MPZstemp.*((SPs+CSPs)>0);
    MPZstemp=MPZstemp-MPZtemp2;


    while(go)
        poleM=MPZtemp2>0;
        MPZtemp2=MPZtemp2-poleM;
        %Generating time constants:
        Tzeromax=ones(n_y,n_u);
        for i=1:n_y
            for j=1:n_u
                if poleM(i,j)>0
                    Tzeromax(i,j)=shootfinder(TFMtemp(i,j),OS_max,1,Tol);
                end
            end
        end
        Tzeromin=log10(Tzeromax/Z_Tminfact);
        Tzeromax=log10(Tzeromax);
        TCs=10.^(Tzeromin+(Tzeromax-Tzeromin).*rand(n_y,n_u));
        TFMfinal=TFMfinal.*(poleM.*TCs.*s+1);
        if sum(sum(MPZtemp2))==0
            go=0;
        end
    end
else

    if sum(sum(MPZs))
        go=1;
    else
        go=0;
    end

    while(go)
        poleM=MPZstemp>0;
        MPZstemp=MPZstemp-poleM;
        %Generating time constants:

        if Dis_T
            for i=1:n_y
                for j=1:n_u
                    TCs(i,j)=Zmp_Dis(randi(length(Zmp_Dis)));
                end
            end
        else
            TCs=10.^(TZsak_min+(TZsak_max-TZsak_min).*rand(n_y,n_u));
        end
        TFMfinal=TFMfinal.*(poleM.*TCs.*s+1);
        if sum(sum(MPZstemp))==0
            go=0;
        end
    end
end
```

```
%Generating nonminimum phase zeros:
go=1;
NMPZstemp=NMPZs;
if Dis_T==false&&Z_TFbased==true
    NMPZtemp2=NMPZstemp.*((SPs+CSPs)>0);
    NMPZstemp=NMPZstemp-NMPZtemp2;

    while(go)
        poleM=NMPZtemp2>0;
        NMPZtemp2=NMPZtemp2-poleM;
        %Generating time constants:
        Tzeromax=ones(n_y,n_u);
        for i=1:n_y
            for j=1:n_u
                if poleM(i,j)>0
                    Tzeromax(i,j)=shootfinder(TFMtemp(i,j),US_max,-1,Tol);
                end
            end
        end
        Tzeromin=log10(Tzeromax/Z_Tminfact);
        Tzeromax=log10(Tzeromax);
        TCs=10.^(Tzeromin+(Tzeromax-Tzeromin).*rand(n_y,n_u));
        TFMfinal=TFMfinal.*(-poleM.*TCs.*s+1);
        if sum(sum(NMPZtemp2))==0
            go=0;
        end
    end
else
    if sum(sum(NMPZs))
        go=1;
    else
        go=0;
    end
    while(go)
        poleM=NMPZstemp>0;
        NMPZstemp=NMPZstemp-poleM;
        %Generating time constants:

        if Dis_T
            for i=1:n_y
                for j=1:n_u
                    TCs(i,j)=Znmp_Dis(randi(length(Znmp_Dis)));
                end
            end
        else
            TCs=10.^(TZsak_min+(TZsak_max-TZsak_min)*rand(n_y,n_u));
        end
        TFMfinal=TFMfinal.*(-poleM.*TCs.*s+1);
        if sum(sum(NMPZstemp))==0
            go=0;
        end
    end
end
```

```
%Generating first order unstable poles:

if sum(sum(USPs))
    go=1;
else
    go=0;
end
USPstemp=USPs;

while(go)
    poleM=USPstemp>0;
    USPstemp=USPstemp-poleM;
    if Dis_T
        for i=1:n_y
            for j=1:n_u
                TCs(i,j)=T_DisU(randi(length(T_DisU)));
            end
        end
    else
        TCs=10.^(Tsak_min+(Tsak_max-Tsak_min)*rand(n_y,n_u));
    end
    TFMtemp=-poleM.*TCs.*s+1;
    TFMfinal=TFMfinal.*tfinverter(TFMtemp);
    if sum(sum(USPstemp))==0
        go=0;
    end
end

%Adding first and second order integrators:
TFMfinal=TFMfinal.*((ones(n_y,n_u)-I1s)+I1s/s).*((ones(n_y,n_u)-I2s)+I2s/s/s);


%Generating purely imaginary poles
if sum(sum(MSPs))
    go=1;
else
    go=0;
end
MSPstemp=MSPs;

while(go)
    poleM=MSPstemp>0;
    MSPstemp=MSPstemp-poleM;
    %Generating time constants:
    if Dis_T
        for i=1:n_y
            for j=1:n_u
                TCs(i,j)=W_DisMS(randi(length(W_DisMS)));
            end
        end
    else
        TCs=10.^(Omegasak_min+(Omegasak_max-Omegasak_min)*rand(n_y,n_u));
    end
```

```
    TFMtemp=poleM./(TCs).^2*s*s+1;
    TFMfinal=TFMfinal.*tfinverter(TFMtemp);
    if sum(sum(MSPstemp))==0
        go=0;
    end
end



%Generating complex unstable poles
if sum(sum(UCPs))
    go=1;
else
    go=0;
end
UCPstemp=UCPs;

while(go)
    poleM=UCPstemp>0;
    UCPstemp=UCPstemp-poleM;
    %Generating time constants:
    if Dis_T
        for i=1:n_y
            for j=1:n_u
                k=randi(length(W_DisCU));
                TCs(i,j)=W_DisCU(k);
                Zetas(i,j)=z_DisCU(k);
            end
        end
    else

        TCs=10.^(OmegaUsak_min+(OmegaUsak_max-OmegaUsak_min)*rand(n_y,n_u));
        Zetas=zeta_min+(1-zeta_min)*rand(n_y,n_u);

    end


    TFMtemp=poleM./(TCs).^2.*s^2-2*Zetas./TCs.*poleM.*s+1;
    TFMfinal=TFMfinal.*tfinverter(TFMtemp);
    if sum(sum(UCPstemp))==0
        go=0;
    end
end

%Adding derivatives
TFMfinal=TFMfinal.*(1-D1s+D1s*s);


%Time delays

L=-TDs.*(L_min+rand(n_y,n_u)*(L_max-L_min))*s;
TFMfinal=TFMfinal.*(exp(L));
```

```
if padeOrder>0
    TFMfinal=pade(TFMfinal,padeOrder);
end

end

function [ tfen ] = tfinverter( tfsak )
%%Inverts each element in a TFM

[a, b]=size(tfsak);
tfen=tfsak;
for k=1:a
    for i=1:b
        tfen(k,i)=inv(tfsak(k,i));
    end
end


end


function [ Ps,err ] = TFplaceGen( TFM, NBR,req  )

% Randomly places the elements

% TFM Matrices of where there are TF orders available to be placed
% Number of elements to be placed
% req Order of element(2 for complex poles, otherwise
% 1)

% Outputs Ps-matrix with location of placed poles and zeros
% err-returns 1 if unable to place the required number of elements
% (it will then place as many as it can)
Ps= zeros(size(TFM));
p2Index=[];
err=0;
while sum(sum(TFM>(req-1)))
    p2P=TFM>(req-1);
    p2Index=[p2Index;find(p2P)]; %#ok<AGROW>
    TFM=TFM-req;
end

if length(p2Index)<NBR
    err=1;
    NBR=length(p2Index);
end
p2perm= randperm(length(p2Index));
p2s=p2perm(1:NBR);
for k=1:length(p2s)
    Ps(p2Index(p2s(k)))=Ps(p2Index(p2s(k)))+1;
end
end
```

```
function [T]= shootfinder(G,Smax,Type,prec)
% Determines the time constant of a zero which causes a specified
% over/undershoot of the inputed transfer function

% G transferfunction
% Smax-maximum over/undershoot
% Type-Less then or equal to zero for determining undershoot-
% else determines overshoot
% prec-Tolerance of search for time constant
s=tf('s');
gr=(sqrt(5)+1)/2;
a=0;
b=10;
if Type<=0
    while(true)
        Gtmp1=G*(s*-b+1);
        S1=stepinfo(Gtmp1);
        if S1.Undershoot<Smax
            b=b*10;
        else
            break
        end
    end

    c=b-(b-a)/gr;
    d=a+(b-a)/gr;
    while(abs(c-d)>prec)

        Gtmp1=G*(s*-c+1);
        Gtmp2=G*(s*-d+1);
        S1=stepinfo(Gtmp1);
        S2=stepinfo(Gtmp2);
        fc=abs(S1.Undershoot-Smax);
        fd=abs(S2.Undershoot-Smax);
        if fc<fd
            b=d;
        else
            a=c;
        end
        c=b-(b-a)/gr;
        d=a+(b-a)/gr;
    end
    T=(c+d)/2;
end

if Type>0
    while(true)
        Gtmp1=G*(s*b+1);
        S1=stepinfo(Gtmp1);
        if S1.Overshoot<Smax
            b=b*10;
        else
```

```
            break
        end
    end

    c=b-(b-a)/gr;
    d=a+(b-a)/gr;
    while(abs(c-d)>prec)

        Gtmp1=G*(s*c+1);
        Gtmp2=G*(s*d+1);
        S1=stepinfo(Gtmp1);
        S2=stepinfo(Gtmp2);
        fc=abs(S1.Overshoot-Smax);
        fd=abs(S2.Overshoot-Smax);
        if fc<fd
            b=d;
        else
            a=c;
        end
        c=b-(b-a)/gr;
        d=a+(b-a)/gr;
    end
    T=(c+d)/2;
end


end
```