Appendix A (indoor VLC system model)

```
disp('indoor VLC LOS channel witZh RSS &Drms& SNR parameters')
lx=8; ly=8; lz=3;% office dimension in meter
h=2.2; %the distance between source and receiver plane
Rb = 1e6;                    % Data rate of system
I2 = 0.562;                  % Noise Bandwidth Factor
Iamp = 5e-12;                % Amplifier Current
Bn = 50e6;                   % Noise Bandwidth
R = 1;                       % Responsivity of Photodiode
q = 1.6e-19;                 % Electron Charge
[XT,YT]=meshgrid([-lx/4 lx/4],[-ly/4 ly/4]);
x=linspace(-lx/2,lx/2,lx*5);
y=linspace(-ly/2,ly/2,ly*5);
[XR,YR]=meshgrid(x,y);
P_LED=4e-3; %transmitted optical power by individual LED
nLED=16; % number of LED array nLED*nLED
P_total=nLED*nLED*p_LED; %Total transmitted power
theta = 60 ;  % semi-angle at half power
ml=-log10(2)/log10(cosd(theta)); %Lambertian order of emission
Adet=.4; %detector physical area of a PD
Ts=1;%gain of an optical filter
index=1.5;%refractive index of a lens at a PD
FOV=70;%FOV of a receiver
G_Con=(index^2)/(sind(FOV).^2);%gain of an optical concentrator
D1=sqrt((XR-XT(1,1)).^2+(YR-YT(1,1)).^2+h^2);% distance vector from
%source 1
cosphi_A1=h./D1;% angle vector
receiver_angle=acosd(cosphi_A1);
H_E1=(ml+1)*Adet.*cosphi_A1.^(ml+1)./(2*pi.*D1.^2);% channel DC
%gainforsource 1
P_rec_E1=P_total.*H_E1.*Ts.*G_Con;% received power from source 1;
P_rec_E1(abs(receiver_angle)>FOV)=0;
P_rec_E2=fliplr(P_rec_E1);
P_rec_E3=flipud(P_rec_E1);
P_rec_E4=fliplr(P_rec_E3);
P_rec_total=P_rec_E1+P_rec_E2+P_rec_E3+P_rec_E4;
P_rec_dBm=pow2db(P_rec_total);
surfc(x,y,P_rec_dBm)
title('Resived Signal Strength RSS')
xlabel('Length of Room');
ylabel('Width of Room');
zlabel('RSS in dB');
figure
contour(x,y,P_rec_dBm)
title('2D CONTUR ')
hold on
mesh(x,y,P_rec_dBm)
figure
% Calculate Noise in System
Bs = Rb*I2;
Pn = Iamp/Rb;
Ptotal = P_rec_total+Pn;
new_shot = 2*q*Ptotal*Bs;
```

```
new_amp = Iamp^2*Bn;
% Calculate SNR
new_total = new_shot + new_amp;
SNRl = (R.*P_rec_total).^2./ new_total;
SNRdb = 10*log10(SNRl);
index = index + 1;
%Plot Graph %
mesh(x,y,SNRdb);
title('SNR Distribution in Room');
xlabel('Length of Room');
ylabel('Width of Room');
zlabel('SNR in dB');
figure
C=3e8*1e-9;% time will be measured in ns in the program
rho=0.8;% reflection coefficient
m=-log10(2)/log10(cosd(theta));
Nx=lx*3; Ny=ly*3; Nz=round(lz*3);% number of grid in each surface
dA=lz*ly/(Ny*Nz);
% calculation grid area
x=-lx/2:lx/Nx:lx/2;
y=-ly/2:ly/Ny:ly/2;
z=-lz/2:lz/Nz:lz/2;
% first transmitter calculation
TP1=[0 0 lz/2];% transmitter position
TPV=[0 0 -.9];% transmitter position vector
RPV=[0 0 .9];% receiver position vector
WPV1=[.9 0 0];
WPV2=[0 .9 0];
WPV3=[-.9 0 0];
WPV4=[0 -.9 0];
% wall vectors
delta_t=1/2;% time resolution in ns, use in the form of 1/2^m
 for ii=1:Nx+1
 for jj=1:Ny+1
 RP=[x(ii) y(jj) -lz/2];
 t_vector=0:25/delta_t; % time vector in
 h_vector=zeros(1,length(t_vector));% receiver position vector
 % LOS channel gain
 D1=sqrt(dot(TP1-RP,TP1-RP));
 cosphi=lz/D1;
 tau0=D1/C;
 index=find(round(tau0/delta_t)==t_vector);
 if abs(acosd(cosphi))<=FOV
  h_vector(index)=h_vector(index)+(m+1)*Adet.*cosphi.^(m+1)./(2*pi.*D1.^ 2);
 end
 %reflection from first wall
 count=1;
 for kk=1:Ny+1
 for ll=1:Nz+1
WP1=[-lx/2 y(kk) z(ll)];
D1=sqrt(dot(TP1-WP1,TP1-WP1));
cos_phi=abs(WP1(3)-TP1(3))/D1;
cos_alpha=abs(TP1(1)-WP1(1))/D1;
D2=sqrt(dot(WP1-RP,WP1-RP));
cos_beta=abs(WP1(1)-RP(1))/D2;
cos_psi=abs(WP1(3)-RP(3))/D2;
tau1=(D1+D2)/C;
```

```
index=find(round(tau1/delta_t)==t_vector);
if abs(acosd(cos_psi))<=FOV

h_vector(index)=h_vector(index)+(m+1)*Adet*rho*dA*cos_phi^m*cos_alpha*cos_bet
a*cos_psi/(2*pi^2*D1^2*D2^2);
end
  count=count+1;
 end
 end
 %% Reflection from second wall
 count=1;
 for kk=1:Nx+1
 for ll=1:Nz+1
 WP2=[x(kk) -ly/2 z(ll)];
 D1=sqrt(dot(TP1-WP2,TP1-WP2));
 cos_phi= abs(WP2(3)-TP1(3))/D1;
 cos_alpha=abs(TP1(2)-WP2(2))/D1;
 D2=sqrt(dot(WP2-RP,WP2-RP));
 cos_beta=abs(WP2(2)-RP(2))/D2;
 cos_psi=abs(WP2(3)-RP(3))/D2;
 tau2=(D1+D2)/C;
 index=find(round(tau2/delta_t)==t_vector);
 if abs(acosd(cos_psi))<=FOV

h_vector(index)=h_vector(index)+(m+1)*Adet*rho*dA*cos_phi^m*cos_alpha*cos_bet
a*cos_psi/(2*pi^2*D1^2*D2^2);
 end
 count=count+1;
 end
end
 % Reflection from third wall
 count=1;
for kk=1:Ny+1
for ll=1:Nz+1
WP3=[lx/2 y(kk) z(ll)];
D1=sqrt(dot(TP1-WP3,TP1-WP3));
cos_phi= abs(WP3(3)-TP1(3))/D1;
cos_alpha=abs(TP1(1)-WP3(1))/D1;
D2=sqrt(dot(WP3-RP,WP3-RP));
cos_beta=abs(WP3(1)-RP(1))/D2;
cos_psi=abs(WP3(3)-RP(3))/D2;
tau3=(D1+D2)/C;
index=find(round(tau3/delta_t)==t_vector);
if abs(acosd(cos_psi))<=FOV

h_vector(index)=h_vector(index)+(m+1)*Adet*rho*dA*cos_phi^m*cos_alpha*cos_bet
a*cos_psi/(2*pi^2*D1^2*D2^2);
end
 count=count+1;
end
end
 % Reflection from fourth wall
 count=1;
 for kk=1:Nx+1
  for ll=1:Nz+1
 WP4=[x(kk) ly/2 z(ll)];
 D1=sqrt(dot(TP1-WP4,TP1-WP4));
```

```
 cos_phi= abs(WP4(3)-TP1(3))/D1;
 cos_alpha=abs(TP1(2)-WP4(2))/D1;
 D2=sqrt(dot(WP4-RP,WP4-RP));
 cos_beta=abs(WP4(2)-RP(2))/D2;
 cos_psi=abs(WP4(3)-RP(3))/D2;
 tau4=(D1+D2)/C;
 index=find(round(tau4/delta_t)==t_vector);
    if abs(acosd(cos_psi))<=FOV

h_vector(index)=h_vector(index)+(m+1)*Adet*rho*dA*cos_phi^m*cos_alpha*cos_bet
a*cos_psi/(2*pi^2*D1^2*D2^2);
    end
  count=count+1;
   end
  end
  t_vector=t_vector*delta_t;
  mean_delay(ii,jj)=sum((h_vector).^2.*t_vector)/sum(h_vector.^2);
  Drms(ii,jj)=sqrt(sum((t_vector-
mean_delay(ii,jj)).^2.*h_vector.^2)/sum(h_vector.^2));
  end
  end
  surf(x,y, Drms)
 % titel('diffuse reflection')
  xlabel('Length of Room');
  ylabel('Width of Room');
  zlabel('Drms');
```

Appendix B (outdoor VLC system model code)

```
% Description: This file is used to define all parameters.


%% simulation
Sim_Par = 'POW';  % simulation parameter; 'POW' = average optical power is
varid, 'EXR' = extention ratio is varied.
Avg_Opt_Pow_start = 0.1e-3;  % optical average power - start value (W)
Avg_Opt_Pow_end = 5e-3;  % optical average power - stop value (W)
Avg_Opt_Pow_step = 5;  % optical average power - step value (W)
Avg_Opt_Pow_const = 100e-3;  % onstant optical average power (W)
ext_ratio_start = 2;  % extinction ratio = P_max/P_min - start value
ext_ratio_end = 200;  % extinction ratio = P_max/P_min - stop value
ext_ratio_step = 10;  % extinction ratio = P_max/P_min - step value
ext_ratio_const = 20;  % constant extinction ratio = P_max/P_min
RepOrd = 2;  % repeatation Order


%% bit operations
Sync_EN = false;  % if synchronisation is needed
NoB = 5e6;  % no of bits for each burst transmission
BR = 1e6;  % Data rate (bps)
NoS = 5;  % No of samples per bit


%% baseband modulation
lambda = 1550e-9; % laser wavelength (m)////////////
Sig_Step = 5e-3;  % signal step for each signal level (A)
BW = BR*1.25;  % bandwidth of baseband signal


%% laser
Las_Eff = 0.5;  % laser internal modulation efficiency (W/A)///////
Sig_Level = 0.5;  % a calibration constant - the DC level of optical signal
for NRZ-OOK modulation


%% background light
P_background = 0.01e-3;  % background light power (W); radiation from
diffused from sun and sky
```

```
%% geometrical loss
GL_EN = false;  % if the geometrical loss is taken into account
Prop_Mod = 'UNI';  % beam propagation/illumination model; 'UNI' = uniform
propagation/illumination, 'GUS' = Gaussian propagation/illumination


%% lossy channel
MiscLoss = 0;  % miscellaneous channel loss (dB)
Link_Len = 500;  % link Length (m)
Fading_Add = 'M2';  % fading effect; 'M1' = fading affects average power,
'M2' = fading affects signal, 'M3' = fading affects average power + signal


%% fog/smoke channel
FS_EN =true;  % if the effect of fog/smoke is present
Vis_FS = 2.25;  % fog/smoke visibility (km)
lambda_0 = 1550e-9;  % reference wavelength (green light) (m)
T_th_FS = 2/100;  % contrast threshold (typical value = 2%)


%% turbulence channel
Turb_EN = false;  % if the effect of turbulence is present
Cn2 = 4e-13;  % the refractive index structure coefficient (m^-2/3)
F_t = 500;  % turbulence maximum frequency (Hz); inverse of tepmoral
coherence(1 - 10 mS)
Resamp_Turb = 'RECT';  % method of resampling the turbulence samples; 'RES' =
uses 'resample' function, 'RECT' = uses 'rectpulse' function
Turb_Mod = 'GG';  % turbulence model; 'LN' = Log-Normal model, 'GG' = Gamma-
Gamma model


%% pointing error channel
PE_EN = false;  % if the effect of pointing errors is present
sig_j_PE = 0.5;  % pointing errors horizontal jitter (m)
mu_h_PE = 0;  % horizontal displacement
mu_v_PE = 0;  % vertical displacement
F_p = 500;  % pointing errors maximum frequency (Hz); inverse of tepmoral
coherence(1 - 10 mS)
Resamp_PE = 'RECT';  % method of resampling the pointing errors samples;
'RES' = uses 'resample' function, 'RECT' = uses 'rectpulse' function


%% transmitter optics
TX_AP_Type = 'ANG';  % transmitter beam parameter type; 'ANG' = divergence
angle is given, 'DIA' = Tx aperture diameter is given
Prop_Type = 'GUS';  % laser propagation model; 'GUS' = Gaussain propagaion
model, 'UNI' = uniform propagation model
theta_d_v = 10;  % full vertical divergence angle (Deg)
theta_d_h = 10;  % full horizontal divergence angle (Deg)
w_tx_v = 5e-3;  % vertical beam size (m)
w_tx_h = 5e-3;  % horizontal beam size (m)
Tx_Pos = [0; 0; 0];  % laser source cartesian position in the global
coordinate (m, m, m)
```

```
Tx_Dir = [cosd(90); sind(90); 0];  % direction of the source propagation;
cannot be 0
Tx_Ori = 0;  % orientation of the source (Deg); around local z axis


%% receiver optics
Rx_Ap_dia = 0.005;  % receiver aperture diameter (m)
Rx_Pos = [sqrt(Link_Len^2 - 2^2); 2; 0];  % receiver aperture cartesian
position in the global coordinate (m, m, m)
Rx_Dir = [cosd(45 + 180); sind(45 + 180); 0];  % direction of the receiver
aperture facing; normal to the detector face; cannot be 0
Rx_Ori = 0;  % orientation of the aperture (Deg); around local z axis
Rx_Trn = 85;  % receiver aperture transmittance (%)
AFOV = 1;  % receiver aperture full-angle angular field-of-view (Deg)


%% photodetector
PD_Resp = 0.5;  % responsivity of photodiode (A/W)
PD_Gain = 1e1;  % transimpedance amplifier gain (V/A)
PD_NEP = 1e-12;  % noise equivalent power (NEP) of optical receiver
(W/sqrt(Hz))
PD_RL = 50;  % receiver load impedance (Ohms)
BW_BR_r = 1.25;  % bandwidth to bit rate ratio (Hz/bps)
q_ch = 1.60217662e-19;  % electron charge (C)


%% detection
Thresh_Len_Coeff = 1;  % this coefficient is used to shorten/widen the length
of adaptive threshold estimation



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clc;  % clean the command console
clearvars;  % clean all variables in the workspace. Use 'clear all' for older
versions of MATLAB
close all;  % close all open figures, windows, etc.


%% global parameters
run('.\\GlobalParameters.m');  % load the simulation parameters


%% initialisation
run('.\\Codes\\InitialisingParameters.m');  % initialise the simulation
parameters


SNR_dB_Sim = zeros(1, Loop_Order);  % SNR array (dB)
BER_Sim = zeros(1, Loop_Order);  % BER array
Q_Sim = zeros(1, Loop_Order);  % Q-factor array
NoE_Sim = zeros(1, Loop_Order);  % no of erronous bits at each simulation
step
```

```
Threshold_Sim = zeros(NoT*Thresh_Len_Coeff + 1, Loop_Order);  % threshold
array; including values for adaptive thresholding


%% Main Loop
% calculate BER for each given parameter
for Index = 1:Loop_Order


    % ----------------------------------------------------------------------
    % assign the simulation parameters
    Avg_Opt_Pow = Avg_Opt_Pow_Array(Index);  % set the average optical power
    ext_ratio = ext_ratio_Array(Index);  % set the extinction ratio

    % ----------------------------------------------------------------------
    % Section 2
    % laser modulation part 1
    % calculating power levels for each SNR value
    P_Sim_avg = Avg_Opt_Pow;    % calculate average optical power
    P_Sim_0 = 2*P_Sim_avg/(1 + ext_ratio);  % output power for bit 0
    P_Sim_1 = 2*P_Sim_avg/(1 + 1/ext_ratio);  % output power for bit 1
    DelP_Sim = P_Sim_1 - P_Sim_0;  % amplityde of optical signal


    % to Increase the accurary, repeat each BER step "RepOrd" times
    for Index_Rep = 1:RepOrd

        % print out information regarding current iteration
        fprintf(['Index = %d out of %d\nAverage optical power\t= %5.3f dBm,\t
%5.3f mW',...
            '\nExtinction ratio\t\t= %5.2f\n'],...
            Index, Loop_Order, 10*log10(Avg_Opt_Pow) + 30, Avg_Opt_Pow*1e3,
ext_ratio);  % print out the SNR index
        fprintf('Repetition\t\t\t\t= %d out of %d\n', Index_Rep, RepOrd);  %
print out the repetition index
        fprintf([repmat('*\t', 1, 18), '\n']);  % print out the separater


        % ------------------------------------------------------------------
        % Section 1
        % Section 2
        % pseudorandom binary sequence (PRBS) generation and NRZ-OOK bits
        Raw_Bit_Sim = randi([0, 1], 1, NoB);  % generate random bits
        Signal_Sim_Out = rectpulse(Raw_Bit_Sim, NoS);  % resample bits


        % ------------------------------------------------------------------
        % Section 2
        % Section 3
        % laser modulation part 2
        % generating output optical signal based on generated OOK signal
        % and calculated power levels
        Laser_Sim_Out = DelP_Sim*(Signal_Sim_Out - Sig_Level);  % generate
laser output (W)
```

```
        Sig_Sim_Power = var(Laser_Sim_Out);  % measure the signal power
        clear Signal_Sim_Out;  % relesse the memory


        % -----------------------------------------------------------------
        % finding the 1st rising edge of signal based on transmit signal
        % finding rising edge
        run('.\\Codes\\FindFirstRisingEdge.m');  % find the first rising edge


        % -----------------------------------------------------------------
        % Section 11
        % geometrical loss effect
        if (GL_EN == true)
            run('.\\Codes\\GeometricalLoss.m');  % apply the geometrical loss
        else
            Geo_Loss_Sim = 1;  % geometrical loss is ignored
        end


        % -----------------------------------------------------------------
        % Section 11
        % fog/smoke channel effect
        if (FS_EN == true)
            run('.\\Codes\\AtmosphericAttenuation.m');  % apply the
atmospheric atteaution effect
        else
            Atm_Att_Sim = 1;  % atmospheric attenuationm effet is ignored
        end


        % -----------------------------------------------------------------
        % Section 9
        % Section 10
        % turbulence channel effect
        if (Turb_EN == true)
            run('.\\Codes\\TurbulenceEffect.m');  % apply the turbulence
effect
        else
            Turb_Sim = 1;  % turbulence effet is ignored
        end


        % -----------------------------------------------------------------
        % Section 9
        % Section 10
        % pointing error channel effect
        if (PE_EN == true)
            run('.\\Codes\\PointingErrosEffect.m');  % apply the pointing
errors effect
        else
            PE_Sim = 1;  % turbulence effet is ignored
        end
```

```
        % ---------------------------------------------------------------
        % Section 4
        % FSO channel effect
        Misc_Loss = 10^(-MiscLoss/10);  % total FSO channel loss
        Fading_Sim = Misc_Loss*Geo_Loss_Sim*Atm_Att_Sim*Turb_Sim.*PE_Sim;  %
channel fading coefficient
        Mean_Fading_Sim = mean(Fading_Sim);  % mean value of the simulated
fading coefficient
        Const_Loss = Atm_Att_Sim*Geo_Loss_Sim*Misc_Loss;  % constant loss due
to atmosphere, propagation, misc loss; this loss only uses onstant
attenuation rather than varying fading
        Var_Fading_Sim = var(Fading_Sim);  % variance value of the simulated
fading coefficient
        clear Turb_Sim;  % release the memory
        clear PE_Sim;  % release the memory
        if(strcmp(Fading_Add, 'M1'))
            Rec_Sim_Opt = Fading_Sim.*P_Sim_avg + Laser_Sim_Out;  % received
optical signal after loss and turbulence effect (Method 1)
        elseif(strcmp(Fading_Add, 'M2'))
            Rec_Sim_Opt = Fading_Sim.*Laser_Sim_Out + P_Sim_avg;  % received
optical signal after loss and turbulence effect (Method 2)
        elseif(strcmp(Fading_Add, 'M3'))
            Rec_Sim_Opt = Fading_Sim.*(Laser_Sim_Out + P_Sim_avg);  %
received optical signal after loss and turbulence effect (Method 3)
        else
            error('pick a fading implementation method');  % print out an
error message and exit
        end
        fprintf('Channel Coefficient:\tMean Value\t\t= %.3f\n',...
            Mean_Fading_Sim);  % print out the mean value
        fprintf('Channel Coefficient:\tVar Value\t\t= %.3f\n',...
            Var_Fading_Sim);  % print out the mean value
        fprintf('Channel Coefficient:\tConstant Loss\t= %.3f dB\n',...
            -10*log10(Const_Loss));  % print out the mean value
        clear Laser_Sim_Out  % release the memory
        clear Fading_Sim;  % release the memory


        % ---------------------------------------------------------------
        % Section 5
        % optical to electical coversion (photodetector)
        Rec_Sim_Sig = PD_Resp*PD_Gain*Rec_Sim_Opt;  % PD conversion
        clear Rec_Sim_Opt;  % release the memory


        % ---------------------------------------------------------------
        % Section 6
        % applying SNR to the received signal
        P_noise_SHN =
(PD_Gain^2)*(2*q_ch*PD_Resp*Avg_Opt_Pow*Const_Loss*BW)/PD_RL;  % total power
of noise (W) due to shot noise
        P_noise = P_noise_NEP + P_noise_SHN + P_noise_BGD;  % total power of
noise (W)
```

```
        Rec_Sim_Sig_Power =
((Const_Loss*PD_Resp*PD_Gain)^2*Sig_Sim_Power)/PD_RL;  % received signal
power (W); normalised to 50 Ohms
        SNR = Rec_Sim_Sig_Power/P_noise;  % signal-to-noise ratio (SNR)
        SNR_dB_Sim(Index) = 10*log10(SNR);  % update SNR array with SNR (dB)
        AdditiveNoise_Sim = randn(1, NoP)*sqrt(P_noise*PD_RL);  % generating
white Gaussian noise
        Det_Sim_Sig = Rec_Sim_Sig + AdditiveNoise_Sim;  % adding white
Gaussian noise to the detected signal
        clear Rec_Sim_Sig;  % release the memory
        clear AdditiveNoise_Sim;  % release the memory


        % -------------------------------------------------------------------
        % performing signall processing on detected electical signal
        LPF_RX_Sim_1 = Det_Sim_Sig - mean(Det_Sim_Sig);  % remove DC level
        P2P_RX_Sim = std(LPF_RX_Sim_1)*2;  % calculate peak-to-peak of
received signal
        LPF_RX_Sim_2 = LPF_RX_Sim_1/P2P_RX_Sim;  % normalise received signal
        clear Det_Sim_Sig;  % release the memory
        RX_Sim_OOK = LPF_RX_Sim_2((Index_RE + round(NoS/2)):NoS:end);  %
sample and hold the detected signal
        clear LPF_RX_Sim_1;  % release the memory
        clear LPF_RX_Sim_2;  % release the memory


        % -------------------------------------------------------------------
        % Section 8
        % analysing detected signal
        % calculating Q-factor parameters
        SigLength = length(RX_Sim_OOK);  % sampled signal length
        run('.\\Codes\\QFactorExtractor.m');  % calculate Q-factor


        % -------------------------------------------------------------------
        % Section 7
        % thresholding and extract the bits
        % generating transmit bits matrix
        run('.\\Codes\\Quantisation.m');  % extract bits from the received
signal


        % -------------------------------------------------------------------
        % Section 8
        % BER calculation
        [NoE_Sim_Temp, BER_Sim_Temp] = biterr(TX_OOK_bit, RX_Sim_OOK_bit);  %
calculate BER and no of erronous bits
        clear TX_OOK_bit;  % release the memory
        clear RX_Sim_OOK_bit;  % release the memory
        BER_Sim(Index) = BER_Sim(Index) + BER_Sim_Temp;  % update the BER
array for each iteration
        NoE_Sim(Index) = NoE_Sim(Index) + NoE_Sim_Temp;  % update the no of
errors array for each iteration
        Q_Sim(Index) = Q_Sim(Index) + Q_Fac;  % update the Q factor array for
each iteration
```

```
        % ----------------------------------------------------------------------
        % signal detection presentaion for each iteration
        fprintf(['SNR\t\t\t= %4.2f dB,\tBER\t\t\t= %5.3g\n',....
            'Threshold\t= %5.3f,\tQ-factor\t= %5.3f\n\n'],...
            SNR_dB_Sim(Index), BER_Sim_Temp, Threshold_Sim_Temp, Q_Fac);  %
print out the received signal properties for each iteration

    end

    % ----------------------------------------------------------------------
    % BER calculation for each SNR
    BER_Sim(Index) = BER_Sim(Index)/RepOrd;  % update the BER array
    Threshold_Sim(Index) = Threshold_Sim(Index)/RepOrd;  % update the
threshold array
    Q_Sim(Index) = Q_Sim(Index)/RepOrd;  % update the Q-factor array

    % ----------------------------------------------------------------------
    % signal detection presentaion for each SNR
    fprintf('SNR\t\t\t\t\t= %4.2f\nAverage BER\t\t\t= %5.3g\nAverage
Threshold\t= %5.3f\nAverage Q-factor\t= %5.3f\n',...
        SNR_dB_Sim(Index), BER_Sim(Index), Threshold_Sim(1, Index),
Q_Sim(Index));  % print out the received signal properties for each SNR
    fprintf([repmat('-', 1, 70), '\n']);  % print out the separater

end


%% analytical initialisation
eta = 1;  % responsivity for theoritical BER
N0 = 1;  % noise spectral density for theoritical BER
k_ord_GH = 400;  % no of Gaussï¿½Hermite quadrature series expansion for
theoritical BER
k_ord_GL = 50;  % no of Gaussï¿½Laguerre quadrature series expansion for
theoritical BER
SNR_dB_Anl = linspace(SNR_dB_Sim(1), SNR_dB_Sim(end), 100);  % SNR (dB) array
for theoritical BER
SNR_Anl = 10.^(SNR_dB_Anl/10);  % SNR array for theoritical BER
I0 = sqrt(N0*2*SNR_Anl)/eta;  % intensity for theoritical BER


%% theoritical BER calculation
% theoritical analysis - pointing errors
run('.\\Codes\\PointingErrorsBER.m');  % calculating analytical BER based on
Log-Normal model

% theoritical analysis - weak turbulence
run('.\\Codes\\LogNormalBER.m');  % calculating analytical BER based on Log-
Normal model

% theoritical analysis - strong turbulence
run('.\\Codes\\GammaGammaBER.m');  % calculating analytical BER based on
Gamma-Gamma model
```

```
% theoritical analysis - pointing errors + weak turbulence
run('.\\Codes\\PointingErrorsLogNormalBER.m');  % calculating analytical BER
based on Gamma-Gamma model

% theoritical analysis - pointing errors + strong turbulence
run('.\\Codes\\PointingErrorsGammaGammaBER.m');  % calculating analytical BER
based on Gamma-Gamma model

% theoritical analysis - clear channel
run('.\\Codes\\ClearChannelBER.m');  % calculating analytical BER over clear
channel

% theoritical analysis - fog/smoke channel
run('.\\Codes\\FSChannelBER.m');  % calculating analytical BER over clear
channel


%% plotting Results
Style = {'', 'o', '*', 's', '^', 'h', 'x', '+', 'd', 'v', '<', '>', 'p'};

figure;  % create and empty figure window
hold on;  % hold all plots
box on;  % make the box around the axis visible
Sim_Type = '';  % simulation type; used for changing the name of the output
graph

% generate the proper plot title for each simulation scenario
if((Turb_EN == false) && (PE_EN == false) && (FS_EN == false))  % clear
channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_cl, 'b', '-', Style{1}, 10);
    str_par = 'clear channel';
    Sim_Type = '-Clear_Channel';  % adjust simulation type variable
elseif((Turb_EN == false) && (PE_EN == false) && (FS_EN == true))  %
fog/smoke channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_fs, 'b', '-', Style{1}, 10);
    str_par = 'fog/smoke channel';
    Sim_Type = '-Fog_Channel';  % adjust simulation type variable
elseif((Turb_EN == false) && (PE_EN == true))  % pointing errors channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_PE, 'b', '-', Style{1}, 10);
    str_par = sprintf('pointing error channel with \\sigma_{j} = %4.2f',
sig_j_PE);
    Sim_Type = '-PE_Channel';  % adjust simulation type variable
elseif((Turb_EN == true) && (PE_EN == false) && strcmp(Turb_Mod, 'LN'))  %
log-normal turbulence channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_LN_turb, 'b', '-', Style{1}, 10);
    str_par = sprintf('turbulence channel (Log-Normal model) with
\\sigma_{R}^2 = %4.2f', sig2_R_Sim);
    Sim_Type = '-LN_Channel';  % adjust simulation type variable
elseif((Turb_EN == true) && (PE_EN == false) && strcmp(Turb_Mod, 'GG'))  %
gamma-gamma turbulence channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_GG_turb, 'b', '-', Style{1}, 10);
    str_par = sprintf('turbulence channel (Gamma-Gamma model) with
\\sigma_{R}^2 = %4.2f', sig2_R_Sim);
    Sim_Type = '-GG_Channel';  % adjust simulation type variable
```

```
elseif((Turb_EN == true) && (PE_EN == true) && strcmp(Turb_Mod, 'LN'))  %
log-normal turbulence + pointing errors channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_PE_LN_turb, 'b', '-', Style{1}, 10);
    str_par = sprintf(['turbulence channel (Log-Normal model) with
\\sigma_{R}^2 = %4.2f\n'...
        'and pointing error channel with \\sigma_{j} = %4.2f'], sig2_R_Sim,
sig_j_PE);
    Sim_Type = '-LN_PE_Channel';  % adjust simulation type variable
elseif((Turb_EN == true) && (PE_EN == true) && strcmp(Turb_Mod, 'GG'))  %
gamma-gamma turbulence + pointing errors channel case
    MarkerPlot(SNR_dB_Anl, BER_Anl_PE_GG_turb, 'b', '-', Style{1}, 10);
    str_par = sprintf(['turbulence channel (Gamma-Gamma model) with
\\sigma_{R}^2 = %4.2f\n'...
        'and pointing error channel with \\sigma_{j} = %4.2f'], sig2_R_Sim,
sig_j_PE);
    Sim_Type = '-GG_PE_Channel';  % adjust simulation type variable
end

MarkerPlot(SNR_dB_Sim, BER_Sim, 'r', '--', Style{2}, 10);  % plot BER vs. SNR
curve
Dummy = BER_Sim(BER_Sim > 0);  % isolate non-zero BER values
axis([SNR_dB_Sim(1), SNR_dB_Sim(end), min(Dummy), 1]);  % auto adjust the
plot axis
xlabel('SNR (dB)');  % label x axis
ylabel('BER');  % label y axis

str_title = sprintf('BER vs. SNR for %s', str_par);  % forma the title text
title(str_title);  % create the title

MakeitPretty(gcf, [10, 9], ['L', 'G'], [12, 1.5, 5, 10],
['.\\Graphs\\BER_SNR', Sim_Type]);  % format the plot
```