EECE7105 2022 Spring Final Project

Raytracing through Simple Gradient–index Materials

Runzhi Wang
May 4 2022

Introduction

The idea of ray tracing comes from as early as the 16th century when it was described by Albrecht Dürer, who is credited for its invention (Georg,1990). Ray tracing can be used in the design of lenses and optical systems, such as cameras, microscopes, telescopes, and binoculars, and its use in this field dates back to the 1900s. Geometric ray tracing is used to describe the propagation of light through a lens system or optical instrument so that the imaging properties of the system can be modeled (Spenser,1962).

The following effects can be integrated into a ray tracer in a straightforward fashion:

- Dispersion leads to chromatic aberration
- Polarization
- Crystal optics
- Fresnel equations
- Laser light effects
- Thin film interference (optical coating, soap bubble) can be used to calculate the reflectivity of a surface.

Atmospheric refraction is the deviation of light or other electromagnetic waves from a straight line as they travel through the atmosphere due to changes in air density with height (Thomas&Joseph,1996). This refraction is due to the fact that the speed of light through air decreases with increasing density (increase in refractive index). Atmospheric refraction near the ground creates a mirage. This refraction can also raise or lower or stretch or shorten the image of distant objects without involving a mirage. Turbulent air can make distant objects appear to flicker or shimmer. The term also applies to the refraction of sound. Atmospheric refraction is taken into account when measuring the positions of celestial and terrestrial objects.

Refraction near the horizon is highly variable, mainly because of the variability in temperature gradients near the Earth's surface and the geometric sensitivity of nearly horizontal rays to this variability. As early as 1830, Friedrich Bessel discovered that even after all temperature and pressure (but not temperature gradients) corrections for the observer, highly accurate measurements of refraction are 2 degrees above the horizon and ±0.19' 0.50' half a degree above the horizon (Feltcher,1952).

This situation creates a gradient index of refraction. This gradient can be used to produce lenses with flat surfaces, or lenses without the aberrations typical of traditional spherical lenses. Gradient index lenses can have spherical, axial or radial refractive gradients.

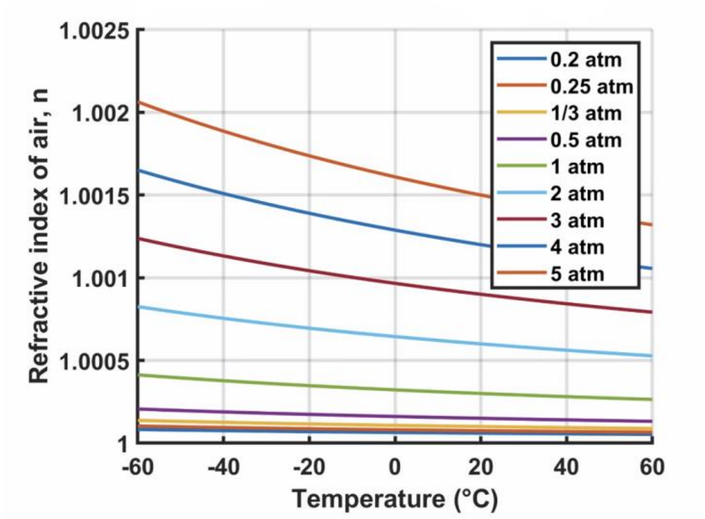
Another example of gradient-index optics in nature is the common mirage that appears in pools of water on roads in hot weather. The pool is actually an image of the sky, clearly on the road because the light is refracting (bending) from its normal straight path. This is due to the difference in refractive index between the hotter, less dense air on the road surface and the denser, cooler air above. Changes in air temperature (and thus density) cause a gradient in its refractive index, causing it to increase with altitude(Dheyab et al,2014). This refractive index gradient causes light rays from the sky (at a shallower angle to the road) to refract, bending them into the observer's eye, where their apparent location is the road surface.

Therefore, one can use ray tracing to simulate this phenomenon by treating air as a graded-index material.

Method

The refractive index of air is affected by temperature, so the refractive index can be judged by the temperature of air.

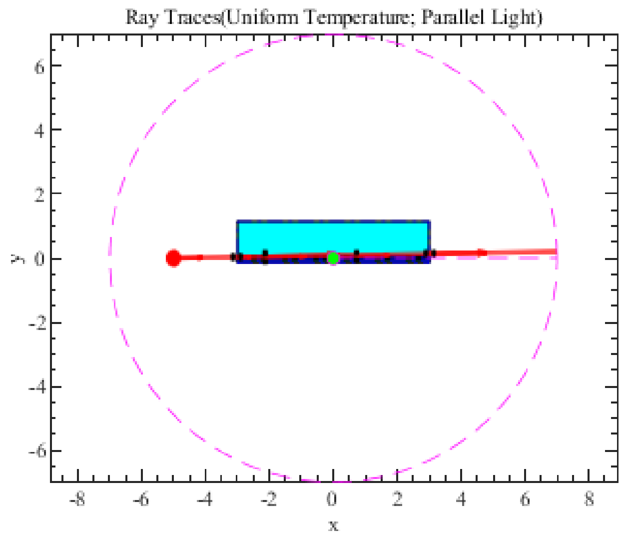
$$n(P,T) = 1 + 0.000293 * \frac{P}{P_0} * \frac{T_0}{T}$$

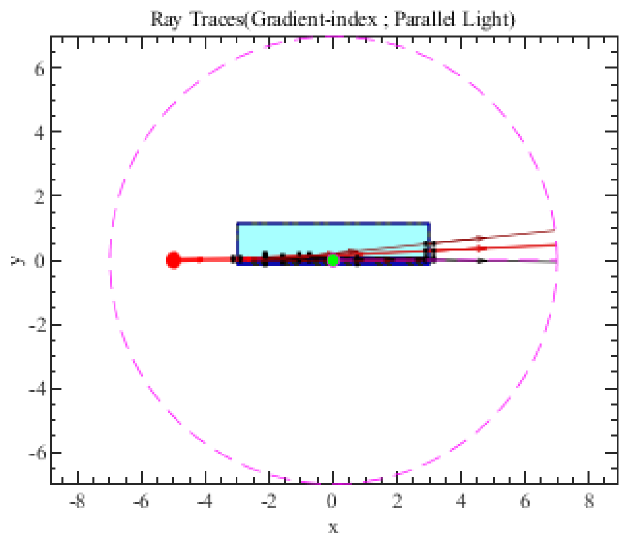


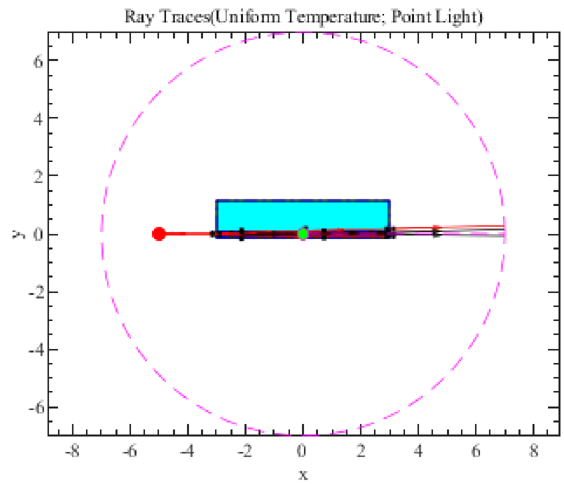
A measured temperature close to the ground is higher, and the refractive index of the air is lower; the relative farther away from the ground, the higher the refractive index of the air. Ray tracing is then performed based on the Snell formula.

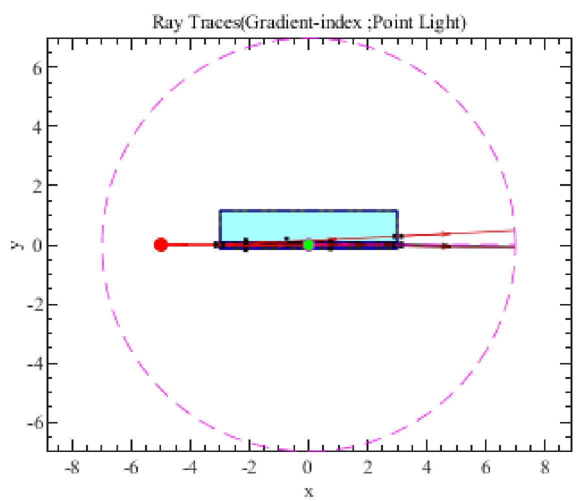
Result

A parallel light with an optical axis of 1° and a horizontal point light are selected for simulation.


Ray Traces(Uniform Temperature; Parallel Light)
y
x

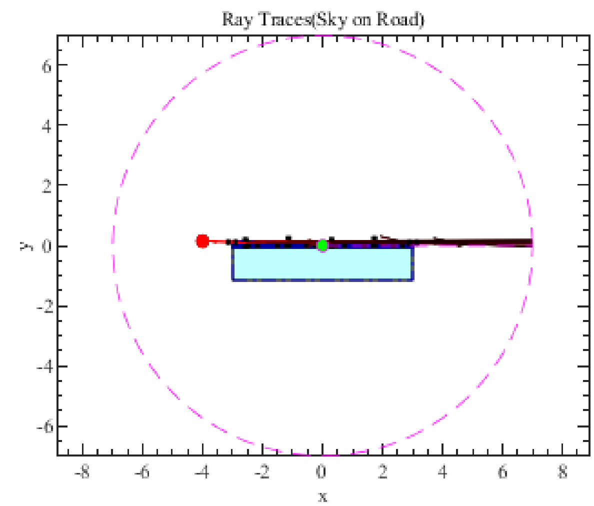

Ray Traces(Gradient-index ; Parallel Light)
y
x

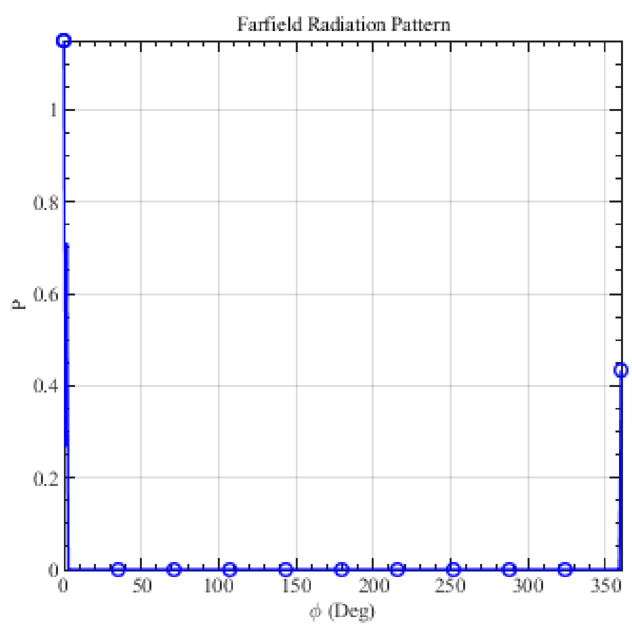




Apparently having a gradient index material bends the light image.

In order to simulate the common mirage, we choose -5° (355°) parallel light for simulation.





Apparently, there is light being bent upwards, which also causes people to see "pools" on the way.

Summary

Apparently, the atmospheric phenomenon of mirage is caused by the gradient index of refraction. That is to say, if the temperature difference reaches a certain level, it will cause a mirage phenomenon. This simulation only simulates the refractive index in the vertical direction. In reality, due to the different surface materials, different refractive indices will be generated in the horizontal direction at the same height. Additionally, there may be some fugitive dust in the air causing scattering. Temperature-induced changes in atmospheric pressure are also worth considering. These can be done in future simulations.

Reference

Dheyab, A. B., Hammod, H. Y., Abdullah, G. H., Muayad, M. W., & Hassan, A. (2014). Gradient-index lenses in imaging applications using Zemax program. Int. J. Innov. Res. Sci. Eng. Technol., 3, 16834-16839.

Fletcher, A. (1952). Astronomical refraction at low altitudes in marine navigation. The Journal of Navigation, 5(4), 307-330.

Georg Rainer Hofmann (1990). "Who invented ray tracing?". The Visual Computer. 6 (3): 120–124. doi:10.1007/BF01911003. S2CID 26348610..

Spencer, G. H., & Murty, M. V. R. K. (1962). General ray-tracing procedure. JOSA, 52(6), 672-678.

Thomas, M. E., & Joseph, R. I. (1996). Astronomical refraction. Johns Hopkins apl technical digest, 17(3), 279.

Appendix 1: Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
%EECE7105 Spring  2022
%Final Project
%Raytracing through Simple Gradient¨Cindex Materials
%This code is tested with MATLAB R2021a
%Reference of this code were derived from the following sources
%github.com/DCC-Lab/RayTracing
%github.com/damienBloch/inkscape-raytracing
%github.com/MansourM61/OpticalRayTracer
%github.com/scottprahl/pygrin
%github.com/GNiendorf/tracepy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
%% Cleaning Environment
clc;
clear all;
close all;

%% Parameters
% Change the parameters in this section to define geometries and
sources,
% etc.

% Change this value according to the background material in the space
n_b = 1.00;  % background refractive index%


% Define all closed geometries in this part.
% Each geometry is composed of several pieces. Each piece is a
parametric
% function handle. The geometry pieces must be defined clockwise.
% Geometry 1
geometry_1 = { @(t) [t, 0.15], [-3, 3];%up bondary
              @(t) [+3, t], [0.15, 0.10];%right bondary
              @(t) [t, 0.10], [+3, -3];%down bondary
              @(t) [-3, t], [0.10, 0.15];%left bondary
            };  % geometry 1 definiton
n_g_1 = 1.0006;  % geometry 1 refractive index

% Geometry 2
geometry_2 = { @(t) [t, 0.10], [-3, +3];
              @(t) [+3, t], [0.10, 0.05];
              @(t) [t, 0.05], [+3, -3];
              @(t) [-3, t], [0.05, 0.10];
            };  % geometry 2 definiton
n_g_2 = 1.0005;  % geometry 2 refractive index

% Geometry 3
geometry_3 = { @(t) [t, 0.05], [-3, +3];
              @(t) [+3, t], [0.05, 0];
              @(t) [t, 0.00], [+3, -3];
              @(t) [-3, t], [0, 0.05];
            };  % geometry 3 definiton
```

```
n_g_3 = 1.0004;  % geometry 3 refractive index
% Geometry 4
geometry_4 = { @(t) [t, 0.00], [-3, +3];
               @(t) [+3, t], [0, -0.05];
               @(t) [t, -0.05], [+3, -3];
               @(t) [-3, t], [-0.05, 0];
             };  % geometry 4 definiton
n_g_4 = 1.0003;  % geometry 4 refractive index
% Geometry 5
geometry_5 = { @(t) [t, -0.05], [-3, +3];
               @(t) [+3, t], [-0.05, -0.10];
               @(t) [t, -0.10], [+3, -3];
               @(t) [-3, t], [-0.10, -0.05];
             };  % geometry 5 definiton
n_g_5 = 1.0002;  % geometry 5 refractive index
% Geometry 6
geometry_6 = { @(t) [t, -0.10], [-3, +3];
               @(t) [+3, t], [-0.10, -1.15];
               @(t) [t, -1.15], [+3, -3];
               @(t) [-3, t], [-1.15, -0.10];
             };  % geometry 6 definiton
n_g_6 = 1.0001;  % geometry 6 refractive index

% Define all sources in this section.
% Each source has a cartesdian location as well as propagation angle
% reletive to +x axis. Each source shoots a ray based on the given
% properties carrying a power < 1.0
% Source 1
x_s_1 = -4;  % source 1 x position
y_s_1 = 0.15;  % source 1 y position
t_s_1 = 358;  % source 1 propagation angle (Deg)
p_s_1 = 1.0;  % source 1 power

% Source 2
x_s_2 = -4;  % source 2 x position
y_s_2 = 0.15;  % source 2 y position
t_s_2 = 358;  % source 2 propagation angle (Deg)
p_s_2 = 1.0;  % source 2 power

% Source 3
x_s_3 = -4;  % source 3 x position
y_s_3 = 0.15;  % source 3 y position
t_s_3 = 358;  % source 3 propagation angle (Deg)
p_s_3 = 1.0;  % source 3 power


% ray tracing parameters
dx = 0.1;  % x resolution for changing the geometry into line pieces
dy = 0.1;  % y resolution for changing the geometry into line pieces
coeff = 0.9;  % for changing the geometry into line pieces, if smaller
line is required, this value specifies the line size reduction
norm_len = 0.2;  % normal line length at the ray-boundary incindence
location
TOL = 10*eps;  % numerical tolerance for calculation errors
max_rt_d = 10;  % if the ray is bounced more, the tracing stops
valid_ratio = 0.01;  % if the bounced power to ray power is less, the
```

```
tracing stops


% fafield parameters
% The parameters in this part define a circle to be the boundary of
% farfield. The radiation pattern is calculated from the rays
approaching
% this boundary.
x_f = 0;  % farfield x centre
y_f = 0;  % farfield y centre
R_f = 7;  % farfield radius
Res = 360;  % farfield resolution; the number of angles from 0 to 360
Deg
t_ref = 0;  % farfield reference angle (Deg); reference for farfield
angles


%% Initialisation
% In this section the required arrays for ray tracing algorithm is
% generated. Change the corresponding values according to the defined
% geometries or sources.

% Add or remove geometries and corresponding refractive index to be
% included in the ray tracing.
Geometry = {geometry_1, geometry_2, geometry_3, geometry_4,
geometry_5,geometry_6};  % array of all geometries
n_g = [n_g_1, n_g_2, n_g_3,n_g_4,n_g_5,n_g_6];  % geometry refractive
index


% Add or remove position, angle and power of sources to be included in
the
% ray tracing.
x_s = [x_s_1, x_s_2,x_s_3];  % source x position
y_s = [y_s_1, y_s_2,y_s_3];  % source y position
t_s = [t_s_1, t_s_2,t_s_3];  % source propagation angle (Deg)
p_s = [p_s_1, p_s_2,p_s_3];  % source power

Farfield = [x_f, y_f, R_f, Res, t_ref];  % farfield parameters
delta = [dx, dy];  % resolution in x and y direction
NoG = length(Geometry);  % number of geometries



%% Procesing:
%% Step 1: Create a sample model
ShapePoints = cell(1, NoG);  % array of shape points

for Index_G = 1:NoG  % go through the geometries
    ShapePoints{Index_G} = RT_GeometryQuantizer(Geometry{Index_G},
delta, coeff);  % quantise the geometry
    NoP = size(ShapePoints, 1);  % number of points
end
```

```
%% Step 2: Create light source

NoS = length(x_s);  % number of sources

n_s = zeros(1, NoS);  % source refractive index array

for Index_S = 1:NoS  % go through the sources

    point = [x_s(Index_S), y_s(Index_S)];  % source coordinate

    flag = false;  % initial flag

    for Index_G = 1:NoG  % go through the geometries
        flag = RT_InsideShape(point, ShapePoints{Index_G}, TOL);  %
check if the point is inside the shape
        if flag == true
            break;  % stop scanning more geometries
        end
    end

    if(~flag)  % if the source is outside the shape
        n_s(Index_S) = n_b;  % update the source refractive index with
background medium
    else   % if the source is inside the shape
        n_s(Index_S) = n_g(Index_G);  % update the source refractive
index with geometry medium
    end

end


%% Step 3: Ray Tracing
RT_Array = cell(1, NoS);  % array of rays

for Index_S = 1:NoS  % go through the sources
    point = [x_s(Index_S), y_s(Index_S)];  % source coordinate
    source = [point, t_s(Index_S), p_s(Index_S)];  % source property

    rt_param = [norm_len, max_rt_d, valid_ratio];  % ray tracing
parameters

    RT_Array{Index_S} = RT_RayTracer(source, ShapePoints, n_b, n_g,
rt_param, TOL);  % perform ray tracing for given source and shape
end


%% Step 4: Calculate Farfield
Farfield_Rays = cell(1, NoS);  % array of rays

Angle = linspace(0, 360, Res);  % full space angle
RadPat = zeros(1, Res);  % initial radiation pattern
```

```
for Index_S = 1:NoS  % go through the sources
    Farfield_Rays{Index_S} = RT_EstimateFarfield(RT_Array{Index_S},
Farfield, TOL);  % calculate the farfield

    if isempty(Farfield_Rays{Index_S})  % if there is no farfield
        continue;  % skip the loop
    end

    Ang_f = Farfield_Rays{Index_S}.index;  % index of the farfield
angles
    Power_f = Farfield_Rays{Index_S}.power;  % index of the farfield
powers

    RadPat(Ang_f) = RadPat(Ang_f) + Power_f;  % accumulate the power

end


%% Step 5: Ray Tracing Presentation
F_RT_H = figure;  % create a new figure
hold on;  % hold the drawings
box on; % create the box around the figure

RT_RayPlotter(gca, [], ShapePoints, n_g, [], [], 'Geometry', TOL);  %
plot the geometries

for Index_S = 1:NoS  % go through the sources
    source = [x_s(Index_S), y_s(Index_S)];  % source coordinate

    [Ray_H, Arrow_H, Normal_H] = RT_RayPlotter(gca, source, [], [],...
    RT_Array{Index_S}, Farfield, 'Ray', TOL);  % plot the ray tracing
end

source = [x_s(Index_S), y_s(Index_S)];  % source coordinate

RT_RayPlotter(gca, [], [], [], [], Farfield, 'Farfield', TOL);  % plot
the farfield

xlabel('x');  % x axis label
ylabel('y');  % y axis label
title('Ray Traces');  % set the title of figure
axis equal;  % set the aspect ratio of figure to 1

MakeitPretty(F_RT_H, [10, 9], ['L', 'L'], [12, 0.5, 5, 10],
'Ray_Tracing');  % save ray tracing plot


%% Step 6: Farfield Presentation
F_FF_H = figure;  % create a new figure
hold on;  % hold the drawings
box on; % create the box around the figure

MarkerStyle = {'', 'o', '*', 's', '^', 'h', 'x', '+', 'd', 'v', '<',
'>', 'p'};  % define marker styles
```

```
MarkerPlot(Angle, RadPat, 'b', '-', MarkerStyle{2}, 10);

xlabel('\phi (Deg)');  % x axis label
ylabel('P');  % y axis label
title('Farfield Radiation Pattern');  % set the title of figure
axis([0, 360, 0, max(RadPat)]);  % set the axis limits
grid on;  % switch on the grids

MakeitPretty(F_FF_H, [10, 9], ['L', 'L'], [12, 1, 5, 10],
'Farfield_RadPat');  % save farfield radiation pattern plot
%% Function
%%
function ShapePoints = RT_GeometryQuantizer(Geometries, Delta, Coeff)

MAX_MEM = 1000;  % maximum length of temporary memory

dx = Delta(1);  % x resolution
dy = Delta(2);  % y resolution

NoS = size(Geometries, 1);  % number of embedded shapes

Temp = zeros(MAX_MEM, 2); % temperary memory

Mem_Index = 1;  % memory index

for Index = 1:NoS
    t_s = Geometries{Index, 2}(1);  % start point of the shape
    t_e = Geometries{Index, 2}(2);  % end point of the shape

    if(t_s < t_e)  % if t parameter is increasing
        dt_0 = +min(dx, dy);  % initial delta t
    else % if t parameter is decreasing
        dt_0 = -min(dx, dy);  % initial delta t
    end

    dt = dt_0;  % intialize delta t

    t = t_s;  % start from the beginning
    p_s = Geometries{Index, 1}(t);  % start point
    Temp(Mem_Index, :) = p_s;  % insert the first point

    t_1 = t;  % previous t parameter

    if(t_s < t_e)  % if t parameter is increasing
        Cond = t < t_e;  % loop condition
    else % if t parameter is decreasing
        Cond = t > t_e;  % loop condition
    end

    while( Cond )  % while the piece is not finished
        t = t_1 + dt;  % calculate next t
        p_s = Geometries{Index, 1}(t);  % next point
```

```
        while( (abs(p_s(1) - Temp(Mem_Index, 1)) > dx) || (abs(p_s(2) -
Temp(Mem_Index, 2)) > dy) )
            dt = dt*Coeff;  % set a new delta t
            t = t_1 + dt;  % calculate next t
            p_s = Geometries{Index, 1}(t);  % next point
        end

        Mem_Index = Mem_Index + 1;  % update memory index
        t_1 = t;  % update previous t parameter
        dt = dt_0;  % intialize delta t
        Temp(Mem_Index, :) = p_s;  % insert the point

        if(t_s < t_e)  % if t parameter is increasing
            Cond = t < t_e;  % loop condition
        else % if t parameter is decreasing
            Cond = t > t_e;  % loop condition
        end

    end
end

ShapePoints = Temp([1:Mem_Index - 1, 1], :);  % generate quantised
shape matrix
end
%%
function flag = RT_InsideShape(Point, Shape, Tol)

NoP = size(Shape, 1);  % number of points

NoH = 0;  % number of hits

for Index = 1:(NoP - 1)  % go through the points
    x0 = Shape(Index, 1);  % start x point
    y0 = Shape(Index, 2);  % start y point

    x1 = Shape(Index + 1, 1);  % end x point
    y1 = Shape(Index + 1, 2);  % end y point

    line = [x0, y0, x1, y1];  % line geometry

    [~, ~, ~, ~, flag_int] = RT_Intersection([Point, 0], line, Tol);  %
find the intersection point

    dirFlag = flag_int(1);  % direction of propagation
    hitFlag = flag_int(2);  % hit/miss

    if(dirFlag && hitFlag)  % check if insection is valid
        NoH  = NoH + 1;  % increase the number of impact
    end

end

if(mod(NoH, 2) == 0)  % if the source is outside the shape
    flag = false;  % update the return flag
```

```
else   % if the source is inside the shape
    flag = true;  % update the return flag
end
end
%%
function MakeitPretty(FigureHandle, FigureProperties, AxisProperties,
PlotProperties, OutputFile)
%MAKEITPRETTY Graphical Modification Function
%
% Make it Pretty function
% Written by Mojtaba Mansour Abadi
%
% Using this MATLAB function, shit gets in, a butterfly goes out!
% If you don't believe, it's OK, no one cares.
%
% To use the function, plot your figure and type:
% MakeitPretty(FH, [FW, FL], [XS, YS], [FS, LW, MS, AS], FN)
%
% FH = Figure handle, you can enter 'gcf' if you are dealing with one
figure only.
%
% FW = desired figure width.
% FL = desired figure height.
%
% XS = desired X axis style: N = linear, G = logarithmic
% YS = desired Y axis style: N = linear, G = logarithmic
%
% FS = desired Text font size.
% LW = desired line width.
% MS = desired marker size.
% AS = desired axes label size
%
% FN = desired file name
%
% Example:
% X = 1:0.1*pi:2*pi;
% hold on;
% plot(X, sin(X), 'b-');
% plot(X, cos(X), 'k-*');
% xlabel('t');
% ylabel('f(t)');
% legend('sin', 'cos');
% MakeitPretty(gcf, [300, 400], 'LNLG', [16, 2.5, 16, 10], 'Figure');
%
% That's it.

Const_TS = 0.75;
Const_TL = [2, 1]*0.01;

Def_FS = 16.0;
Def_AS = 10.0;
Def_XS = 'linear';
Def_YS = 'log';
Def_LW = 1.5;
Def_MS = 5.0;
Def_FW = 10.0;
```

```
    Def_FL = 9.0;
    Def_FN = 'Figure';

    switch nargin
        case 0
            FH = gcf;
            FS = Def_FS;
            LW = Def_LW;
            MS = Def_MS;
            AS = Def_AS;
            FW = Def_FW;
            FL = Def_FL;
            XS = Def_XS;
            YS = Def_YS;
            FN = Def_FN;
        case 1
            FH = FigureHandle;
            FS = Def_FS;
            LW = Def_LW;
            MS = Def_MS;
            AS = Def_AS;
            FW = Def_FW;
            FL = Def_FL;
            XS = Def_XS;
            YS = Def_YS;
            FN = Def_FN;
        case 2
            FH = FigureHandle;
            T1 = FigureProperties;
            FW = T1(1);
            FL = T1(2);
            XS = Def_XS;
            YS = Def_YS;
            FS = Def_FS;
            LW = Def_LW;
            MS = Def_MS;
            AS = Def_AS;
            FN = Def_FN;
        case 3
            FH = FigureHandle;
            T1 = FigureProperties;
            FW = T1(1);
            FL = T1(2);
            T2 = AxisProperties;
            if T2(1) == 'G'
                XS = 'log';
            else
                XS = 'linear';
            end
            if T2(2) == 'G'
                YS = 'log';
            else
                YS = 'linear';
            end
            FS = Def_FS;
            LW = Def_LW;
            MS = Def_MS;
```

```
            AS = Def_AS;
            FN = Def_FN;
        case 4
            FH = FigureHandle;
            T1 = FigureProperties;
            FW = T1(1);
            FL = T1(2);
            T2 = AxisProperties;
            if T2(1) == 'G'
                XS = 'log';
            else
                XS = 'linear';
            end
            if T2(2) == 'G'
                YS = 'log';
            else
                YS = 'linear';
            end
            T3 = PlotProperties;
            FS = T3(1);
            LW = T3(2);
            MS = T3(3);
            AS = T3(4);
            FN = Def_FN;
        case 5
            FH = FigureHandle;
            T1 = FigureProperties;
            FW = T1(1);
            FL = T1(2);
            T2 = AxisProperties;
            if T2(1) == 'G'
                XS = 'log';
            else
                XS = 'linear';
            end
            if T2(2) == 'G'
                YS = 'log';
            else
                YS = 'linear';
            end
            T3 = PlotProperties;
            FS = T3(1);
            LW = T3(2);
            MS = T3(3);
            AS = T3(4);
            FN = OutputFile;
    end

    set(FH, 'Color', [1, 1, 1])
    % set(FH, 'Resize', 'off');
    set(FH, 'RendererMode', 'auto');
    set(FH, 'Renderer', 'painters');

    OBJ = findobj(FH, 'type', 'axes');

    for Index = 1:length(OBJ)
```

```
        AX1 = OBJ(Index);
        % AX1 = get(FH, 'CurrentAxes');

        set(AX1, 'XMinorTick', 'on');
        set(AX1, 'YMinorTick', 'on');
        set(AX1, 'ZMinorTick', 'on');

        set(AX1, 'FontName','Times New Roman');
        set(AX1, 'FontSize', FS);
        set(AX1, 'Box', 'on');

        set(AX1, 'XScale', XS);
        set(AX1, 'YScale', YS);

        set(AX1, 'LineWidth', Const_TS);
        set(AX1, 'TickLength', Const_TL);

        XL = get(AX1, 'XLabel');
        set(XL, 'FontName','Times New Roman');
        set(XL, 'FontSize', FS);

        YL = get(AX1, 'YLabel');
        set(YL, 'FontName','Times New Roman');
        set(YL, 'FontSize', FS);

        ZL = get(AX1, 'ZLabel');
        set(ZL, 'FontName','Times New Roman');
        set(ZL, 'FontSize', FS);
        set(AX1, 'FontSize', AS);

    end

    root = get(FH, 'Parent');

    % OBJ = get(AX, 'Children');
    OBJ = findobj(FH, 'type', 'line');

    for child = OBJ
        set(child, 'LineWidth', LW);
        set(child, 'MarkerSize', MS);
    end

    set(root, 'ShowHiddenHandles', 'on');

    ANN = findobj(FH, 'type', 'hggroup');

    for annot = ANN
        if(~isprop(annot, 'LineWidth'))
            continue;
        end
        set(annot, 'LineWidth', LW);
    end

    for annot = ANN
```

```
    if(~isprop(annot, 'FontName'))
        continue;
    end
    set(annot, 'FontName','Times New Roman');
    set(annot, 'FontSize', FS);
end

OBJ = findobj(FH, 'type', 'text');

for child = OBJ
    set(child, 'FontName','Times New Roman');
    set(child, 'FontSize', FS);
end

set(root, 'ShowHiddenHandles', 'off');

set(FH, 'paperunits', 'centimeters', 'paperposition', [0, 0, FW, FL]);

print(FH, '-djpeg', '-r600', FN);

return;
end
%%
function Handle = MarkerPlot(X, Y, Color, Style, Marker, NOM)

% Num_X = length(X);

% Step = floor(Num_X/NOM);

DelX = (X(end) - X(1))/NOM;

X_M = zeros(1, NOM + 1);
Y_M = zeros(1, NOM + 1);

X_M(1) = X(1);
Y_M(1) = Y(1);

for Index = 2:NOM
    X_c = (Index - 1)*DelX + X(1);
    Diff = X - X_c;

    Index_M = find(Diff(1:(end - 1)) <= 0);

    X_M(Index) = X(Index_M(end));
    Y_M(Index) = Y(Index_M(end));
end

X_M(end) = X(end);
Y_M(end) = Y(end);

HL = ishold;

plot(X, Y, [Color, Style]);
```

```
hold on;
if (strcmp(Marker, '') == 0)
%     Handle = plot(X(1:Step:end), Y(1:Step:end), [Color, Marker]);
    plot(X_M, Y_M, [Color, Marker]);
end

if(HL == 0)
    hold off;
end

Handle = plot(X_M(1), Y_M(1), [Color, Style, Marker]);
return;
end
%%
function Farfield_Rays = RT_EstimateFarfield(RT_Array, Farfield, Tol)

x_f = Farfield(1);  % farfield x centre
y_f = Farfield(2);  % farfield y centre
R_f = Farfield(3);  % farfield radius
Res = Farfield(4);  % farfield resolution
t_ref = Farfield(5);  % farfield reference angle (Deg)

R_f_def = 2;  % default farfield radius

NoR = length(RT_Array);  % number of rays

Farfield_Flag = zeros(1, NoR);  % farfield flag array

for Index = 1:NoR  % go through all the rays
    if (RT_Array(Index).t_index == -1) && (RT_Array(Index).r_index == -
1)  % if node is farfield
        Farfield_Flag(Index) = 1;  % set the farfield flag
    end
end

if (sum(Farfield_Flag) < 1)  % no farfield ray exists
    Farfield_Rays = [];  % set the return matrix to empty
    return;  % return to the caller function
end

Farfield_Array = RT_Array(find(Farfield_Flag == 1));  % create farfield
array

NoF = length(Farfield_Array);  % number of farfield array

Angle = linspace(0, 360, Res);  % full space angle

Farfield_index = zeros(1, NoF);  % farfield index array
Farfield_angle = zeros(1, NoF);  % farfield angle array
Farfield_distance = zeros(1, NoF);  % farfield distance array
Farfield_power = zeros(1, NoF);  % farfield distance array

for Index = 1:NoF  % go though the farfield arrayfun
```

```
    x_p =  Farfield_Array(Index).pos(1);  % x position
    y_p =  Farfield_Array(Index).pos(2);  % y position
    t_p =  Farfield_Array(Index).ang;  % line angle

    cx_1 = (x_p - x_f)*cosd(t_p);  % dummy param x 1
    cy_1 = (y_p - y_f)*sind(t_p);  % dummy param y 1

    cx_2 = (x_p - x_f)^2;  % dummy param x 2
    cy_2 = (y_p - y_f)^2;  % dummy param y 2

    a_p = 1;  % quadratic equation a parameter
    b_p = 2*(cx_1 + cy_1);  % quadratic equation b parameter
    c_p = cx_2 + cy_2 - R_f^2;  % quadratic equation c parameter

    R_f_ray = (-b_p + sqrt(b_p^2 - 4*a_p*c_p))/(2*a_p);  % farfield of
ray

    if (abs(imag(R_f_ray)) > Tol) || (real(R_f_ray) < 0)  % if the
calculated radius is invalid
        R_f_ray = R_f_def;  % set the radius to default farfield radius
    end

    x_r_f = x_p + R_f_ray*cosd(t_p);  % ray x at farfield
    y_r_f = y_p + R_f_ray*sind(t_p);  % ray y at farfield

    V_p_x = x_p - x_f;  % centre to point x vector
    V_p_y = y_p - y_f;  % centre to point y vector

    V_f_x = x_r_f - x_p;  % point to farfield x vector
    V_f_y = y_r_f - y_p;  % point to farfield y vector

    V_r_f_x = V_p_x + V_f_x;  % centre to farfield x vector
    V_r_f_y = V_p_y + V_f_y;  % centre to farfield y vector

    t_r_f = atan2d(V_r_f_y, V_r_f_x);  % point to farfield angle
    t_r_f = t_r_f - t_ref;  %  set the angle based on the reference
    if (t_r_f < 0)  % if the angle is from -180 to 0
        t_r_f = t_r_f + 360;  % make the angle between 180 to 360
    end

    [~, Index_min] = min(abs(Angle - t_r_f));  % find the closest angle
    Farfield_index(Index) = Index_min;  % update farfield index array

    angle_f = Angle(Index_min);  % pick the  closest angle
    Farfield_angle(Index) = angle_f;  % update farfield angle array

    dist_f = norm([V_r_f_x, V_r_f_y]);  % point to farfield distance
    Farfield_distance(Index) = dist_f;  % update farfield distance
array

    power_f = Farfield_Array(Index).power;  % farfield power
    Farfield_power(Index) = power_f;  % update farfield distance array
```

```
end

Farfield_Rays = struct('index', Farfield_index, 'angle',
Farfield_angle,...
    'distance', Farfield_distance, 'power', Farfield_power);  % create
the farfield measurements array
end
%%
function [Index_geo, Index_int] = RT_FindClosestIntersection(Ray,
Shapes, TochBound, Tol)

Index_int = [];  % intersection index
Index_geo = [];  % intersection index
d_min = [];  % minimum distance

NoG = length(Shapes);  % number of embedded shapes

for Index_G = 1:NoG  % go though each geometry
    Points = Shapes{Index_G};  % pick the geometry

    NoP = size(Points, 1);  % number of points

    for Index = 1:(NoP - 1)  % go through all the points and find the
closest intersection

        if (Index_G == TochBound(1)) && (Index == TochBound(2))
            continue;
        end

        x0 = Points(Index, 1);  % start x point
        y0 = Points(Index, 2);  % start y point

        x1 = Points(Index + 1, 1);  % end x point
        y1 = Points(Index + 1, 2);  % end y point

        line = [x0, y0, x1, y1];  % line geometry

        [~, ~, ~, dist, flag_int] = RT_Intersection(Ray, line, Tol);  %
find the intersection point

        dirFlag = flag_int(1);  % direction of propagation
        hitFlag = flag_int(2);  % hit/miss

        if(dirFlag && hitFlag)  % check if insection is valid
            if(isempty(d_min))  % if this is the first intersection?
                d_min = dist;  % update the minimum distance
                Index_geo = Index_G;  % update the index of geometry
                Index_int = Index;  % update the index of minimum
intersection
            else  % if this is not the first intersection?
                if(dist < d_min)  % if the new intersection is closer?
                    d_min = dist;  % update the minimum distance
                    Index_geo = Index_G;  % update the index of
geometry
```

```
                    Index_int = Index;  % update the index of minimum
intersection
                end
            end
        end
    end
end
end
%%
function [Ray, Normal] = RT_RayPlotPreparation(RT_Array, Farfield, Tol)

RT_Array_len = length(RT_Array);  % ray tracing array length

x_f = Farfield(1);  % farfield x centre
y_f = Farfield(2);  % farfield y centre
R_f = Farfield(3);  % farfield radius

R_f_def = 2;  % default farfield radius

X_ray_temp = zeros(1,2*RT_Array_len);  % temporary ray x position
Y_ray_temp = zeros(1,2*RT_Array_len);  % temporary ray y position
U_ray_temp = zeros(1,2*RT_Array_len);  % temporary ray x direction
vector
V_ray_temp = zeros(1,2*RT_Array_len);  % temporary ray y direction
vector
C_ray_temp = zeros(1,2*RT_Array_len);  % temporary ray color
coefficient

Index_ray = 1;  % current index of rays

for Index = 1:RT_Array_len  % go though the nodes to draw arrows
    Index_t = RT_Array(Index).t_index;  % next transmit element
    Index_r = RT_Array(Index).r_index;  % next reflect element

    x_0 =  RT_Array(Index).pos(1);  % x position
    y_0 =  RT_Array(Index).pos(2);  % y position
    t_0 =  RT_Array(Index).ang;  % line angle

    if( (Index_t == 0) || (Index_t <= -2) ) &&...
            ( (Index_r == 0) || (Index_r <= -2) ) % it is a termination
or invalid node or tir
        % do nothing and pass
    elseif( (Index_t == 0) || (Index_t <= -2))  % it is a termination
or invalid node
        x_1 =  RT_Array(Index_r).pos(1);  % x end
        y_1 =  RT_Array(Index_r).pos(2);  % y end
        u = (x_1 - x_0);  % x direction vector
        v = (y_1 - y_0);  % y direction vector

        X_ray_temp(Index_ray) = x_0;  % update the arrow x
        Y_ray_temp(Index_ray) = y_0;  % update the arrow y
        U_ray_temp(Index_ray) = v;  % update the arrow x direction
vector
        V_ray_temp(Index_ray) = u;  % update the arrow y direction
vector
```

```
        C_ray_temp(Index_ray) = RT_Array(Index).power;  % update the
power coefficeint

        Index_ray = Index_ray + 1;  % update the current index of rays

    elseif (Index_t == -1)  % it is a farfield

        cx_1 = (x_0 - x_f)*cosd(t_0);  % dummy param x 1
        cy_1 = (y_0 - y_f)*sind(t_0);  % dummy param y 1

        cx_2 = (x_0 - x_f)^2;  % dummy param x 2
        cy_2 = (y_0 - y_f)^2;  % dummy param y 2

        a_p = 1;  % quadratic equation a parameter
        b_p = 2*(cx_1 + cy_1);  % quadratic equation b parameter
        c_p = cx_2 + cy_2 - R_f^2;  % quadratic equation c parameter


        R_f_ray = (-b_p + sqrt(b_p^2 - 4*a_p*c_p))/(2*a_p);  % farfield
of ray

        if (abs(imag(R_f_ray)) > Tol) || (real(R_f_ray) < 0)  % if the
calculated radius is invalid
            R_f_ray = R_f_def;  % set the radius to default farfield
radius
        end

        x_1 = x_0 + R_f_ray*cosd(t_0);  % x end at farfield
        y_1 = y_0 + R_f_ray*sind(t_0);  % y end at farfield

        u = (x_1 - x_0);  % x direction vector
        v = (y_1 - y_0);  % y direction vector

        X_ray_temp(Index_ray) = x_0;  % update the arrow x
        Y_ray_temp(Index_ray) = y_0;  % update the arrow y
        U_ray_temp(Index_ray) = v;  % update the arrow x direction
vector
        V_ray_temp(Index_ray) = u;  % update the arrow y direction
vector

        C_ray_temp(Index_ray) = RT_Array(Index).power;  % update the
power coefficeint

        Index_ray = Index_ray + 1;  % update the current index of rays

    else  % it is a ray
        x_1 =  RT_Array(Index_t).pos(1);  % x end
        y_1 =  RT_Array(Index_t).pos(2);  % y end
        u = (x_1 - x_0);  % x direction vector
        v = (y_1 - y_0);  % y direction vector

        X_ray_temp(Index_ray) = x_0;  % update the arrow x
        Y_ray_temp(Index_ray) = y_0;  % update the arrow y
```

```
        U_ray_temp(Index_ray) = v;  % update the arrow x direction
vector
        V_ray_temp(Index_ray) = u;  % update the arrow y direction
vector

        C_ray_temp(Index_ray) = RT_Array(Index).power;  % update the
power coefficeint

        Index_ray = Index_ray + 1;  % update the current index of rays

    end
end

Index_ray = Index_ray - 1;  % remove the last item

X_ray = X_ray_temp(1:Index_ray);  % cut the extra x locations
Y_ray = Y_ray_temp(1:Index_ray);  % cut the extra y locations
U_ray = U_ray_temp(1:Index_ray);  % cut the extra x direction vectors
V_ray = V_ray_temp(1:Index_ray);  % cut the extra y direction vectors
C_ray = C_ray_temp(1:Index_ray);  % cut the extra color coeefcients

Ray = struct('x', X_ray, 'y', Y_ray, 'u', U_ray, 'v', V_ray, 'c',
C_ray);  % create Ray struct

X_norm_temp = zeros(1,2*RT_Array_len);  % temporary normal x position
Y_norm_temp = zeros(1,2*RT_Array_len);  % temporary normal y position
U_norm_temp = zeros(1,2*RT_Array_len);  % temporary normal x direction
vector
V_norm_temp = zeros(1,2*RT_Array_len);  % temporary normal y direction
vector

Index_normal = 1;  % current index of normals

for Index = 2:RT_Array_len  % go through the normals to draw normal
lines
    normal = RT_Array(Index).normal;  % normal line

    X_norm_temp(Index_normal) = normal(1);  % nomal x start
    Y_norm_temp(Index_normal) = normal(2);  % nomal y start
    U_norm_temp(Index_normal) = normal(3) - normal(1);  % nomal x
direction vector
    V_norm_temp(Index_normal) = normal(4) - normal(2);  % nomal y
direction vector
    Index_normal = Index_normal + 1;  % update the current index of
normals

end

Index_normal = Index_normal - 1;  % remove the last item

X_norm = X_norm_temp(1:Index_normal);  % cut the extra x locations
Y_norm = Y_norm_temp(1:Index_normal);  % cut the extra y locations
U_norm = U_norm_temp(1:Index_normal);  % cut the extra x direction
vectors
```

```
V_norm = V_norm_temp(1:Index_normal);  % cut the extra y direction
vectors

Normal = struct('x', X_norm, 'y', Y_norm, 'u', U_norm, 'v', V_norm);  %
create Normal struct
end
%%
function [Ray_H, Arrow_H, Normal_H] = RT_RayPlotter(Axe_H, Source,
Geometries, n_Geometry, RT_Array, Farfield, drawFlag, Tol)

DEBUG = false;  % no debugging message

ArrowLen = 0.5;  % arrow vector length
HeadSize = 0.5;  % arrow head size
NormalSize = 2;  % normal line size
RayBaseColor = [1, 0, 0];  % ray base color
FarfieldBaseColor = [1, 0, 1];  % farfield base color
GeometryBaseColor = [0, 1, 1];  % Geometry base color


axes(Axe_H);  % se teh ecurrent axes

if strcmp(drawFlag, 'Geometry')  % if function is called for drawing
geometries

    NoG = length(Geometries);  % number of geometries

    n_max = max(n_Geometry);  % maximum refractive index
    n_min = min(n_Geometry);  % minimum refractive index

    BaseColor = rgb2hsv(GeometryBaseColor);  % map the RGB color to HSV
space

    H_value = BaseColor(1);  % hue value of HSV color
    V_Value = BaseColor(3);  % value value of HSV color

    Sat_min = 0.25;  % minimum saturation
    Sat_max = 1.00;  % maximum saturation

    if abs(n_max - n_min) > Tol  % if materials are not the same
        m_color = (Sat_max - Sat_min)/(n_max - n_min + eps);  % linear
gradient of saturation with respect to refractive index
    else  % if materials are the same
        m_color = 0;  % set the saturation gadient to zero
    end

    for Index = 1:NoG

        S_value = n_Geometry(Index)*m_color + Sat_max -
n_max*m_color;  % calculate S value

        Color_hsv = [H_value, S_value, V_Value];  % geometry color
        Color = hsv2rgb(Color_hsv);  % map the HSV color to RGB space
```

```
        fill(Geometries{Index}(:, 1), Geometries{Index}(:, 2),
Color);  % draw the shape fill
        plot(Geometries{Index}(:, 1), Geometries{Index}(:, 2), 'k-',
'LineWidth', 2);   % draw the shape boundary

        x_b_0 = min(Geometries{Index}(:, 1));  % x0 bounding box
        x_b_1 = max(Geometries{Index}(:, 1));  % x1 bounding box
        y_b_0 = min(Geometries{Index}(:, 2));  % y0 bounding box
        y_b_1 = max(Geometries{Index}(:, 2));  % y1 bounding box

        plot([x_b_0, x_b_1, x_b_1, x_b_0, x_b_0], [y_b_0, y_b_0, y_b_1,
y_b_1, y_b_0], 'b-.');   % draw the shape bounding box
    end

elseif strcmp(drawFlag, 'Ray')  % if function is called for drawing
rays

    x_s = Source(1);  % source x position
    y_s = Source(2);  % source y position

    plot(x_s, y_s, 'Color', RayBaseColor, 'Marker', 'o', 'LineStyle',
'-', 'MarkerFaceColor', 'r', 'MarkerSize', 10);  % draw the source

    [Ray, Normal] = RT_RayPlotPreparation(RT_Array, Farfield(1:3),
Tol);  % create the rays and normals arrays

    X_ray = Ray.x;  % ray x locations
    Y_ray = Ray.y;  % ray y locations
    U_ray = Ray.u;  % ray x direction vectors
    V_ray = Ray.v;  % ray y direction vectors
    C_ray = Ray.c;  % ray colors

    Ray_H = zeros(1, length(X_ray));  % rays handle
    Arrow_H = zeros(1, length(X_ray));  % arrows handle

    for Index = 1:length(X_ray)  % go through the nodes
        Color = C_ray(Index)*RayBaseColor;  % update the color

        Ray_H(Index) = quiver(X_ray(Index), Y_ray(Index),...
            V_ray(Index), U_ray(Index), 'Color', Color,...
            'AutoScale', 'off', 'ShowArrowHead', 'off');  % plot the
rays
        Arrow_H(Index) = quiver(X_ray(Index), Y_ray(Index),...
            V_ray(Index)*ArrowLen, U_ray(Index)*ArrowLen,...
            'Color', Color, 'MaxHeadSize', HeadSize);  % plot the
arrows
    end

    X_norm = Normal.x;  % raw normal x locations
    Y_norm = Normal.y;  % raw normal y locations
    U_norm = Normal.u;  % raw normal x direction vectors
    V_norm = Normal.v;  % raw normal y direction vectors

    [X_sorted, I_sorted] = sort(X_norm);  % sort x locations
```

```
    Y_sorted = Y_norm(I_sorted);  % sort y locations
    U_sorted = U_norm(I_sorted);  % sort x direction vectors
    V_sorted = V_norm(I_sorted);  % sort y direction vectors

    dx = diff(X_sorted);  % calculate x difference
    dy = diff(Y_sorted);  % calculate y difference
    du = diff(U_sorted);  % calculate u difference
    dv = diff(V_sorted);  % calculate v difference

    I_dup = find((abs(dx) < Tol) & (abs(dy) < Tol) & (abs(du) < Tol) &
(abs(dv) < Tol));  % find similar normals

    X_norm(I_dup) = [];  % remove x location duplicates
    Y_norm(I_dup) = [];  % remove y location duplicates
    U_norm(I_dup) = [];  % remove x direction vector duplicates
    V_norm(I_dup) = [];  % remove y direction vector duplicates

    Normal_H = zeros(1, length(X_norm));  % normal handle

    for Index = 1:length(X_norm)  % go through the normals
        Normal_H(Index) = quiver(X_norm(Index), Y_norm(Index),...
            U_norm(Index), V_norm(Index),...
            'k:', 'AutoScale', 'off', 'ShowArrowHead', 'off',
'LineWidth', NormalSize);  % plot the normals
    end

elseif strcmp(drawFlag, 'Farfield')  % if function is called for
drawing farfield


    x_f = Farfield(1);  % farfield x centre
    y_f = Farfield(2);  % farfield y centre
    R_f = Farfield(3);  % farfield radius
    Res = Farfield(4);  % farfield resolution
    t_ref = Farfield(5);  % farfield reference angle (Deg)

    t = linspace(0, 360, Res);  % theta angle (Deg)

    x = x_f + R_f*cosd(t);  % x locus of farfield
    y = y_f + R_f*sind(t);  % y locus of farfield

    plot(x, y, 'Color', FarfieldBaseColor, 'LineStyle', '--');  % plot
farfield circle

    x_ref = x_f + [0, R_f*cosd(t_ref)];  % reference x locus of
farfield
    y_ref = y_f + [0, R_f*sind(t_ref)];  % reference y locus of
farfield

    plot(x_ref, y_ref, 'Color', FarfieldBaseColor, 'LineStyle', '--
');  % plot reference angle

    plot(x_f, y_f, 'Color', FarfieldBaseColor, 'Marker', 'o',
'MarkerFaceColor', 'g', 'MarkerSize', 10);  % plot farfield centre
```

```
else  % in case of unknown state
    if(DEBUG == true)  % if debugging is enabled
        print('Unknown state!');  % print the error
    end
end
end
%%
function [V_t, V_r, R_s, R_p, P_n, TIRFlag] = RT_SnellsLaw(V_s, Source,
V_b, Boundary, Pos_i, Norm_l, Tol)

DEBUG = false;  % no debugging message

x_s = Source(1);  % source x position (m)
y_s = Source(2);  % source y position (m)
t_s = Source(3);  % propagation angle (deg)
n_s = Source(4);  % source refractive index

x_b_0 = Boundary(1);  % boundary x start (m) - CW direction
y_b_0 = Boundary(2);  % boundary y start (m)
x_b_1 = Boundary(3);  % boundary x stop (m)
y_b_1 = Boundary(4);  % boundary y stop (m)
n_b = Boundary(5); % boundary refractive index

X_i = Pos_i(1);  % intersection x position
Y_i = Pos_i(2);  % intersection x position

ti_ = acosd( dot(V_s, V_b) / ( norm(V_s) * norm(V_b) ) );  % incident
angle (deg)
td = acosd( dot([1, 0], V_b) / ( norm(V_b) ) );  % direction of the
boundary

if td < 90  % the choice of (x0, y0) and (x1, y1) is correct
    dFlipBoundary = false;  % boundary vector is straight
    tb = atan2d( +(y_b_1 - y_b_0), +(x_b_1 - x_b_0) );  % boundary
angle (deg)
    x_p = x_b_0;  % pivot x coordinate
    y_p = y_b_0;  % pivot x coordinate

else  % the choice of (x0, y0) and (x1, y1) is wrong
    dFlipBoundary = true;  % boundary vector is flipped
    tb = atan2d( -(y_b_1 - y_b_0), -(x_b_1 - x_b_0) );  % boundary
angle (deg)
    x_p = x_b_1;  % pivot x coordinate
    y_p = y_b_1;  % pivot x coordinate

end

if ti_ < 90  % check if the incident angle is in range
    ti = ti_;  % set the angle
else  % check if the incident angle is out of range
    ti = 180 - ti_;  % set the angle
end

r00 = +cosd(-tb); % rotation coefficients
```

```
r01 = -sind(-tb); % from
http://www.euclideanspace.com/maths/geometry/affine/aroundPoint/matrix2
d/
r10 = +sind(-tb);
r11 = +cosd(-tb);

xs_ = r00*x_s + r01*y_s + x_p - r00*x_p - r01*y_p;  % rotated x
position of source
ys_ = r10*x_s + r11*y_s + y_p - r10*x_p - r11*y_p;  % rotated y
position of source

xi_ = r00*X_i + r01*Y_i + x_p - r00*x_p - r01*y_p;  % rotated x
position of incidence
yi_ = r10*X_i + r11*Y_i + y_p - r10*x_p - r11*y_p;  % rotated y
position of incidence

ax_i_ = xi_ - xs_;  % intersection x vector - local coordinate
ay_i_ = yi_ - ys_;  % intersection y vector - local coordinate

if ax_i_ < 0  % check for positive x propagation - local coordinate
    ti_x_lc_dir = -1;  % positive propagation
else  % check for negative x propagation
    ti_x_lc_dir = +1;  % negative propagation
end
if ay_i_ < 0  % check for positive y propagation - local coordinate
    ti_y_lc_dir = -1;  % positive propagation
else  % check for negative y propagation
    ti_y_lc_dir = +1;  % negative propagation
end

t1 = 90 - ti;  % angle in medium 1

sin_t2 = (n_s/n_b) * sind(t1);  % sin (angle) in medium 2
t2 = asind(sin_t2);  % transmit angle in medium 2
t2_ = 90 - t2;  % transmit angle in medium 2

if(ti_x_lc_dir >= 0) && (ti_y_lc_dir >= 0) && (dFlipBoundary ==
false) % moving to +x, +y, Quarter 1, no flipping
    ax_t_ = +cosd(t2_); % transmit x vector - local coordinate
    ay_t_ = +sind(t2_); % transmit y vector - local coordinate

    ax_r_ = +cosd(ti); % reflection x vector - local coordinate
    ay_r_ = -sind(ti); % reflection y vector - local coordinate

elseif(ti_x_lc_dir < 0) && (ti_y_lc_dir >= 0) && (dFlipBoundary ==
false) % moving to -x, +y, Quarter 2, no flipping
    ax_t_ = -cosd(t2_); % transmit x vector - local coordinate
    ay_t_ = +sind(t2_); % transmit y vector - local coordinate

    ax_r_ = -cosd(ti); % reflection x vector - local coordinate
    ay_r_ = -sind(ti); % reflection y vector - local coordinate

elseif(ti_x_lc_dir < 0) && (ti_y_lc_dir < 0) && (dFlipBoundary ==
false) % moving to -x, -y, Quarter 3, no flipping
    ax_t_ = -cosd(t2_); % transmit x vector - local coordinate
```

```
        ay_t_ = -sind(t2_); % transmit y vector - local coordinate

        ax_r_ = -cosd(ti); % reflection x vector - local coordinate
        ay_r_ = +sind(ti); % reflection y vector - local coordinate

    elseif(ti_x_lc_dir >= 0) && (ti_y_lc_dir < 0) && (dFlipBoundary ==
    false) % moving to +x, -y, Quarter 4, no flipping
        ax_t_ = +cosd(t2_); % transmit x vector - local coordinate
        ay_t_ = -sind(t2_); % transmit y vector - local coordinate

        ax_r_ = +cosd(ti); % reflection x vector - local coordinate
        ay_r_ = +sind(ti); % reflection y vector - local coordinate

    elseif(ti_x_lc_dir >= 0) && (ti_y_lc_dir >= 0) && (dFlipBoundary ==
    true) % moving to +x, +y, Quarter 1, flipping
        ax_t_ = +cosd(t2_); % transmit x vector - local coordinate
        ay_t_ = +sind(t2_); % transmit y vector - local coordinate

        ax_r_ = +cosd(ti); % reflection x vector - local coordinate
        ay_r_ = -sind(ti); % reflection y vector - local coordinate

    elseif(ti_x_lc_dir < 0) && (ti_y_lc_dir >= 0) && (dFlipBoundary ==
    true) % moving to -x, +y, Quarter 2, flipping
        ax_t_ = -cosd(t2_); % transmit x vector - local coordinate
        ay_t_ = +sind(t2_); % transmit y vector - local coordinate

        ax_r_ = -cosd(ti); % reflection x vector - local coordinate
        ay_r_ = -sind(ti); % reflection y vector - local coordinate

    elseif(ti_x_lc_dir < 0) && (ti_y_lc_dir < 0) && (dFlipBoundary ==
    true) % moving to -x, -y, Quarter 3, flipping
        ax_t_ = -cosd(t2_); % transmit x vector - local coordinate
        ay_t_ = -sind(t2_); % transmit y vector - local coordinate

        ax_r_ = -cosd(ti); % reflection x vector - local coordinate
        ay_r_ = +sind(ti); % reflection y vector - local coordinate

    elseif(ti_x_lc_dir >= 0) && (ti_y_lc_dir < 0) && (dFlipBoundary ==
    true) % moving to +x, -y, Quarter 4, flipping
        ax_t_ = +cosd(t2_); % transmit x vector - local coordinate
        ay_t_ = -sind(t2_); % transmit y vector - local coordinate

        ax_r_ = +cosd(ti); % reflection x vector - local coordinate
        ay_r_ = +sind(ti); % reflection y vector - local coordinate
    end

    if((sin_t2 - 1) < Tol)  % no total internal reflection
        TIRFlag = false;  % no TIR
        if(DEBUG == true)
            disp('no total internal reflection!');  % print the message
        end
    else  % total intertnal reflection
        TIRFlag = true;  % TIR
        if(DEBUG == true)
```

```
        disp('total internal reflection!');  % print the message
    end
end

xt_ = xi_ + ax_t_;  % transmit x position
yt_ = yi_ + ay_t_;  % transmit y position

xr_ = xi_ + ax_r_;  % reflection x position
yr_ = yi_ + ay_r_;  % reflection y position

r00 = +cosd(tb); % rotation coefficients
r01 = -sind(tb); % from
http://www.euclideanspace.com/maths/geometry/affine/aroundPoint/matrix2
d/
r10 = +sind(tb);
r11 = +cosd(tb);

xt = r00*xt_ + r01*yt_ + x_p - r00*x_p - r01*y_p;  % rotated x position
of transmittance
yt = r10*xt_ + r11*yt_ + y_p - r10*x_p - r11*y_p;  % rotated y position
of transmittance

xr = r00*xr_ + r01*yr_ + x_p - r00*x_p - r01*y_p;  % rotated x position
of reflectance
yr = r10*xr_ + r11*yr_ + y_p - r10*x_p - r11*y_p;  % rotated y position
of reflectance

xn_0 = r00*xi_ + r01*(yi_ - Norm_l) + x_p - r00*x_p - r01*y_p;  %
rotated x position of normal start
yn_0 = r10*xi_ + r11*(yi_ - Norm_l) + y_p - r10*x_p - r11*y_p;  %
rotated y position of normal start

xn_1 = r00*xi_ + r01*(yi_ + Norm_l) + x_p - r00*x_p - r01*y_p;  %
rotated x position of normal start
yn_1 = r10*xi_ + r11*(yi_ + Norm_l) + y_p - r10*x_p - r11*y_p;  %
rotated y position of normal start

P_n = [xn_0, yn_0, xn_1, yn_1];  % normal geometry

ax_t = xt - X_i; % transmit x vector - global coordinate
ay_t = yt - Y_i; % transmit y vector - global coordinate

V_t = [ax_t, ay_t];  % transmittance vector

ax_r = xr - X_i; % reflection x vector - global coordinate
ay_r = yr - Y_i; % reflection y vector - global coordinate

V_r = [ax_r, ay_r];  % reflectance vector

R_s = ( abs(n_s*cosd(t1) - n_b*cosd(t2)) / abs(n_s*cosd(t1) +
n_b*cosd(t2)) )^2;  % reflectance = reflectivity = power reflection
coefficient, s polarization
R_p = ( abs(n_s*cosd(t2) - n_b*cosd(t1)) / abs(n_s*cosd(t2) +
n_b*cosd(t1)) )^2;  % reflectance = reflectivity = power reflection
```

```
coefficient, p polarization
end
%%
function [V_s, V_b, P_i, Dist, Flags] = RT_Intersection(Ray, Line, Tol)

DEBUG = false;  % no debugging message

x_ray = Ray(1);  % incoming ray x position (m)
y_ray = Ray(2);  % incoming ray y position (m)
t_ray = Ray(3);  % propagation angle (deg)

x0 = Line(1);  % line x start (m) - CW direction
y0 = Line(2);  % line y start (m)
x1 = Line(3);  % line x stop (m)
y1 = Line(4);  % line y stop (m)

ms = tand(t_ray); % source ray line slope
cnt_s = y_ray - tand(t_ray)*x_ray;  % source ray line constant
ax_s = cosd(t_ray);  % source x vector
ay_s = sind(t_ray);  % source y vector
V_s = [ax_s, ay_s];  % source vector

mb = (y1 - y0)/(x1 - x0 + eps);  % boundary slope
cnt_b = y1 - mb*x1;  % boundary line constant
ax_b = x1 - x0;  % boundary x vector
ay_b = y1 - y0;  % boundary y vector
V_b = [ax_b, ay_b];  % boundary vector

xi = (cnt_b - cnt_s)/(ms - mb + eps);  % intersection x coordinate
yi1 = ms*xi + cnt_s;  % yi #1 calculation
yi2 = mb*xi + cnt_b;  % yi #2 calculation
if ( ~isinf(yi1) && ~isnan(yi1) )  % check if yi1 is valid
    yi = yi1;  % yi1 is valid
else  % check if yi1 in not valid
    yi = yi2;  % yi2 is assumed to be valid
end

P_i = [xi, yi];  % intersect point

Dist = sqrt( (x_ray - xi)^2 + (y_ray - yi)^2 );  % source to insertion
point distance

ax_i = xi - x_ray;  % intersection x vector
ay_i = yi - y_ray;  % intersection y vector
v_i = [ax_i, ay_i];  % intersection vector

cos_t_ray_i = dot(V_s, v_i) / ( norm(V_s) * norm(v_i) + eps);  % cos of
angle between the ray and incident vector

if(abs(cos_t_ray_i - 1.0) < Tol)
    dirFlag = true;  % going towards the boundary
    if(DEBUG == true)  % if debugging is enabled
        disp('going towards the boundary!');  % print the message
    end
else
```

```
        dirFlag = false;  % coming from the boundary
        if(DEBUG == true)  % if debugging is enabled
            disp('getting away from the boundary!');  % print the message
        end
    end

    x_i_f = ((max(x0, x1) + Tol) >= xi) && ((min(x0, x1) - Tol) <= xi);   %
    check if x_i is whithin the boundary range
    y_i_f = ((max(y0, y1) + Tol) >= yi) && ((min(y0, y1) - Tol) <= yi);   %
    check if y_i is whithin the boundary range

    if((x_i_f == true) && (y_i_f == true))
        hitFlag = true;  % ray hitting boundary
        if(DEBUG == true)  % if debugging is enabled
            disp('hitting the boundary!');  % print the message
        end
    else
        hitFlag = false;  % ray missing boundary
        if(DEBUG == true)  % if debugging is enabled
            disp('missing the boundary!');  % print the message
        end
    end

    v_d = [V_b(2), -V_b(1)];  % centrifugal vector

    cos_d_i = dot(v_d, v_i) / ( norm(v_d) * norm(v_i) + eps);  % cos of
    angle between the ray and centrifugal vector

    if(cos_d_i <= 0)
        sideFlag = true;  % the ray is inside the boundary
        if(DEBUG == true)  % if debugging is enabled
            disp('inside the boundary!');  % print the message
        end
    else
        sideFlag = false;  % the ray is outside the boundary
        if(DEBUG == true)  % if debugging is enabled
            disp('outside the boundary!');  % print the message
        end
    end

    Flags = [dirFlag, hitFlag, sideFlag];  % intersection flags

    end
    %%
    function [RT_Array] = RT_RayTracer(Source, Geometries, n_background,
    n_shape, RT_Param, Tol)

    DEBUG = false;  % no debugging message

    NoL = 100;  % binary-tree ray array size

    x_s = Source(1);  % source x point
    y_s = Source(2);  % source y point
    t_s = Source(3);  % source angle (Deg)
    p_s = Source(4);  % source power
```

```
norm_len = RT_Param(1);  % normal line length
max_rt_d = RT_Param(2);  % maximum ray tracing depth
valid_ratio = RT_Param(3);  % minimum valid power ratio

BT(NoL) = struct('pos', [0, 0], 'ang', 0, 'power', 0,...
                 't_index', 0, 'r_index', 0, 'p_index', 0,...
                 'b_index', [0, 0], 'normal', [0, 0, 0, 0]);  % binary-
tree ray array

for Index = 1:NoL  % go through all the elements in the tree
    BT(Index).pos = [0, 0];  % set all the nodes position to [0, 0]
    BT(Index).ang = 0;  % set all the nodes angle to 0 Deg
    BT(Index).power = 0;  % set all the nodes power to 0
    BT(Index).t_index = -2;  % set all the nodes to not been decided
    BT(Index).r_index = -2;  % set all the nodes to not been decided
    BT(Index).p_index = -2;  % set all the nodes to not been decided
    BT(Index).b_index = [0, 0];  % set all the nodes to not on boundary
    BT(Index).normal = [0, 0, 0, 0];  % set all the nodes to not on
boundary
end

TR_depth = 1;  % initial ray tracing depth

Index_c = 1;  % index of current element
Index_e = 2;  % index of next available element
Index_b = [0, 0];  % index of boundary; no boundary is touched

BT(Index_c).pos = [x_s, y_s];  % update current node position
BT(Index_c).ang = t_s;  % update current node angle
BT(Index_c).power = p_s;  % update current node power
BT(Index_c).p_index = 0;  % update current node power

% p_index (= 0 source, <> 0 others)
% r/l_index (= 0 termination, = -1 farfield, = -2 not been decided)

while(true)  % loop throught the traces

    Index_t = BT(Index_c).t_index;  % get the transmit index
    Index_r = BT(Index_c).r_index;  % get the receive index
    Index_p = max(0, BT(Index_c).p_index);  % get the parent index

    if( (Index_t ~= -2) && (Index_r ~= -2) && (Index_p == 0) )  % if
current node is the source and both sides are decided or reflection is
decided and transmit is tir
        if(DEBUG == true)
            disp('end of tracing');  % end the loop
        end
        break;  % leave the loop
    end

    if (Index_p == 0)  % if current node is root
        if ( BT(Index_c).t_index == -2 ) % if transmit side is not
decided
            calcFlag = 'T';  % set the flag to calculate the transmit
```

```
        elseif ( BT(Index_c).r_index == -2 )  % if receive side is not
decided
            calcFlag = 'R';  % set the flag to calculate the receive
        else  % default state at root
            Index_c = 1;  % set the current element to the root
            Index_b = [0, 0];  % set boundary collision for root
            TR_depth = 1;  % reset the ray tracing depth
        end

    elseif (Index_p ~= 0) && (BT(Index_p).t_index ~= 0)  % if current
node is not root and there is no tir
        if ( BT(Index_c).t_index == -2 ) % if transmit side is not
decided
            calcFlag = 'T';  % set the flag to calculate the transmit
        elseif ( BT(Index_c).r_index == -2 )  % if receive side is not
decided
            calcFlag = 'R';  % set the flag to calculate the receive
        else  % if both sides are decided
            Index_c = BT(Index_c).p_index;  % set the current element
to the parent
            Index_b = BT(Index_p).b_index;  % update the touched
boundary
            TR_depth = TR_depth - 1;  % update the depth of ray tracing
            continue;
        end

    elseif (Index_p ~= 0) && (BT(Index_p).t_index == 0)  % if current
node is not root and there is tir
        if ( BT(Index_c).r_index == -2 )  % if receive side is not
decided
            calcFlag = 'R';  % set the flag to calculate the receive
        else  % if both sides are decided
            Index_c = BT(Index_c).p_index;  % set the current element
to the parent
            Index_b = BT(Index_p).b_index;  % update the touched
boundary
            TR_depth = TR_depth - 1;  % update the depth of ray tracing
            continue;
        end

    else  % to avoid any unknown state
        if(DEBUG == true)  % if debugging is enabled
            print('Unknown state!');  % print the error
        end
    end

    ray = [BT(Index_c).pos, BT(Index_c).ang];  % ray geometry

    [Index_Geo, Index_int] = RT_FindClosestIntersection(ray,
Geometries, Index_b, Tol);  % find closest intersection

    if ( isempty(Index_int) )  % if there is no intersection
        BT(Index_c).t_index = -1;  % set all the nodes to farfield
        BT(Index_c).r_index = -1;  % set all the nodes to farfield
```

```
        if (Index_p ~= 0)  % if the current node is not the root
            Index_c = BT(Index_c).p_index;  % update the parent index
to the node above
            TR_depth = TR_depth - 1;  % update the depth of ray tracing
        end

        %         Index_p = max(1, BT(Index_c).p_index);  % get the
parent index
        Index_b = BT(Index_c).b_index;  % update the touched boundary

        continue;  % start over the loop
    end

    Index_b = [Index_Geo, Index_int];  % update the touched boundary

    x0 = Geometries{Index_Geo}(Index_int, 1);  % intersected boundary x
point
    y0 = Geometries{Index_Geo}(Index_int, 2);  % intersected boundary y
point
    x1 = Geometries{Index_Geo}(Index_int + 1, 1);  % intersected
boundary x point
    y1 = Geometries{Index_Geo}(Index_int + 1, 2);  % intersected
boundary y point

    src = ray;  % source geometry
    line = [x0, y0, x1, y1];  % boundary geometry

    [v_s, v_b, p_i, ~, flags] = RT_Intersection(src, line, Tol);  %
find the intersection

    insideFlag = flags(3);  % inside/outside flag

    if insideFlag  % if point is inside the shape
        n_ray = n_shape(Index_Geo);  % set the ray refractive index to
geometry
        n_bnd = n_background;  % set the boundary refractive index to
geometry
    else  % if point is outside the shape
        n_ray = n_background;  % set the ray refractive index to
geometry
        n_bnd = n_shape(Index_Geo);  % set the boundary refractive
index to geometry
    end

    source = [src, n_ray];  % ray geometry
    boundary = [line, n_bnd];  % boundary geometry

    [v_t, v_r, Rs, Rp, pn, tir] = RT_SnellsLaw(v_s, source, v_b,
boundary, p_i, norm_len, Tol);  % calculate transmittance/reflectance

    Ts = 1 - Rs;  % transmittance = tranmsissivity = power transmission
coefficient, s polarization
    Tp = 1 - Rp;  % transmittance = tranmsissivity = power transmission
coefficient, p polarization
```

```
    R_eff = (Rs + Rp)/2;  % effective reflectance
    T_eff = (Ts + Tp)/2;  % effective transmittance

    if ( calcFlag == 'T' )  % if transmit is to be calculated

        if tir == true  % if there is tir
            BT(Index_c).t_index = 0;  % set the transmit node to
termination
            Index_p = max(1, BT(Index_c).p_index);  % get the parent
index
            Index_b = BT(Index_c).b_index;  % update the touched
boundary

        else
            BT(Index_c).t_index = Index_e;  % set the transmit index to
the next available element

            Index_p = Index_c;  % set parent index to current element
            Index_c = Index_e;  % set current index to new element

            Ang = atan2d(v_t(2), v_t(1));  % transmit ray angle
            p_t = BT(Index_p).power*T_eff;  % transmitted power

            BT(Index_c).pos = p_i;  % update current node position with
intersection
            BT(Index_c).ang = Ang;  % update current node angle with
calculated angle
            BT(Index_c).power = p_t;  % update current node power
            BT(Index_c).p_index = Index_p;  % update current node power
            BT(Index_c).b_index = Index_b;  % update the boundary
collision
            BT(Index_c).normal = pn;  % update the normal line

            Index_e = Index_e + 1;  % update the next available element
index

            TR_depth = TR_depth + 1;  % update the depth of ray tracing

        end

    elseif ( calcFlag == 'R' )  % if reflect is to be calculated

        BT(Index_c).r_index = Index_e;  % set the transmit index to the
next available element

        Index_p = Index_c;  % set parent index to current element
        Index_c = Index_e;  % set current index to new element

        Ang = atan2d(v_r(2), v_r(1));  % transmit ray angle
        p_r = BT(Index_p).power*R_eff;  % transmitted power

        BT(Index_c).pos = p_i;  % update current node position with
intersection
```

```
        BT(Index_c).ang = Ang;  % update current node angle with
calculated angle
        BT(Index_c).power = p_r;  % update current node power
        BT(Index_c).p_index = Index_p;  % update current node power
        BT(Index_c).b_index = Index_b;  % update the boundary collision
        BT(Index_c).normal = pn;  % update the normal line

        Index_e = Index_e + 1;  % update the next available element
index

        TR_depth = TR_depth + 1;  % update the depth of ray tracing

    else  % in case of unknown state
        if(DEBUG == true)  % if debugging is enabled
            disp('undefined state!');  % print the error
        end
    end

    if(TR_depth > max_rt_d)  || (BT(Index_c).power/p_s <
valid_ratio)  % check for maximum depth of ray tracing or minimum ratio
of power
        BT(Index_c).t_index = 0;  % set all the nodes to termination
        BT(Index_c).r_index = 0;  % set all the nodes to termination
        BT(Index_c).p_index = Index_p;  % update current node power

        if (Index_p ~= 0)  % if the current node is not the root
            Index_c = BT(Index_c).p_index;  % update the parent index
to the node above
            TR_depth = TR_depth - 1;  % update the depth of ray tracing
        end

%          Index_p = max(1, BT(Index_c).p_index);  % get the parent
index
        Index_b = BT(Index_c).b_index;  % update the touched boundary
    end

end

RT_Array = BT(1:(Index_e - 1));  % extract the results
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
```