REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE May 15, 2000	3. REPORT TYPE AND DATES COVERED Final report 8/16/99 - 6/30/2000

4. TITLE AND SUBTITLE A Complete Physics-Based Channel Parameter Simulation for Wave Propagation in a Forest Environment	5. FUNDING NUMBERS DAAD19-99-1-0325
6. AUTHOR(S) Kamal Sarabandi and Il-Suek Koh	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Michigan, Radiation Laboratory 1301 Beal Avenue ,Ann Arbor MI. 48109-2122	8. PERFORMING ORGANIZATION REPORT NUMBER F001961-F
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U. S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709-2211	10. SPONSORING / MONITORING AGENCY REPORT NUMBER ARO 40363.1-EL-II

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12 a. DISTRIBUTION / AVAILABILITY STATEMENT	12 b. DISTRIBUTION CODE
Approved for public release; distribution unlimited.	

13. ABSTRACT (Maximum 200 words)

At HF through UHF frequencies, wave propagation in a forest environment is mainly attributed to a lateral wave which propagates at the canopy-air interface. Due to the existence of tree trunks, significant multiple scattering also occurs which is the dominant source of field fluctuations. Basically the current induced in the tree trunks by the source and the lateral wave re-radiate and generate higher order lateral waves and direct scattered waves. Using a full-wave analysis based on the method of moments in conjunction with Monte Carlo simulations, the effect of multiple scattering among a very large number of tree trunks is studied. It is shown that only scatterers near the source and the observation points contribute to the field fluctuations significantly. This result drastically simplifies the numerical complexity of the problem. Keeping about 200 tree trunks in the vicinity of the transmitter dipole and the receiver point, a Monte Carlo simulation is used to evaluate the statistics of the spatial and spectral behavior of the field at the receiver. Using a wideband simulation, the temporal behavior (impulse response) is also studied as is performance of antenna arrays and the effects of different spatial diversity combining schemes in such a multi-path environment.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Electromagnetics, Wave propagation, Forest environment	89
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION ON THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500
89)
Std. 239-18

Standard Form 298 (Rev.2-
Prescribed by ANSI
Z98-102

20000628061

A Complete Physics-Based Channel Parameter Simulation for Wave Propagation in a Forest Environment

Kamal Sarabandi, Il-Suek Koh
Radiation Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
Tel :(734) 936-1575, Fax:(734) 647-2106
email: saraband@eecs.umich.edu

Abstract

At HF through UHF frequencies, wave propagation in a forest environment is mainly attributed to a lateral wave which propagates at the canopy-air interface. Due to the existence of tree trunks, significant multiple scattering also occurs which is the dominant source of field fluctuations. Basically the current induced in the tree trunks by the source and the lateral wave re-radiate and generate higher order lateral waves and direct scattered waves. Using a full-wave analysis based on the method of moments in conjunction with Monte Carlo simulations, the effect of multiple scattering among a very large number of tree trunks is studied. It is shown that only scatterers near the source and the observation points contribute to the field fluctuations significantly. This result drastically simplifies the numerical complexity of the problem. Keeping about 200 tree trunks in the vicinity of the transmitter dipole and the receiver point, a Monte Carlo simulation is used to evaluate the statistics of the spatial and spectral behavior of the field at the receiver. Using a wideband simulation, the temporal behavior (impulse response) is also studied as is performance of antenna arrays and the effects of different spatial diversity combining schemes in such a multi-path environment.

1 Introduction

Accurate prediction of radio wave propagation in a communications channel is essential in the development and design of an efficient, and low-cost wireless system. High fidelity models are necessary in order to treat the tradeoff between radiated power and signal processing by addressing such issues as coherency, field variations, multi-path, and dispersive (path delay) effects. Current channel models are heuristic

DTIC QUALITY INSPECTED 4

in nature and have limited applicability. A physics-based approach to channel characterization gives insight into the mechanisms of radio wave propagation and inherently provides highly accurate results. With this as a motivation, a physics-based modeling approach is being pursued at the University of Michigan for which advanced and efficient electromagnetic diffraction models are being developed.

Due to relatively high attenuation rates, direct wave propagation in a forest environment is not possible over large distances at high frequencies. In the HF-UHF range where both the transmitter and receiver are embedded in the foliage, radio signals can propagate over relatively large distances. This peculiar behavior is explained by certain type of surface waves known as the lateral wave [1].

Mathematically speaking, the lateral wave is the contribution of the branch cut from the integral of the spectral representation of a dipole field inside a half-space dielectric. This formulation provides an expression for the mean-field assuming the canopy-air interface is smooth. For predicting the mean-field more accurately, in a recent study [2], the effect of canopy-air interface roughness on the propagation of lateral waves was studied and it was shown that the canopy-air interface roughness reduces the mean-field. As mentioned earlier path-loss only partially characterizes the channel and other scattering mechanisms must be accounted for to complete the model. At UHF and lower frequencies, the dimensions of leaves and branches are small compared to wavelength and do not cause significant scattering which is the source of signal fluctuations, multi-path, and dispersion. The effect of tree trunks which are electrically large create a highly scattering environment.

In this paper the effect of tree trunks is accounted for by computing their interaction with the source and the lateral waves. A numerical solution based on the method of moments (MoM) in conjunction with Monte-Carlo simulation is proposed to evaluate the scattering effects of tree trunks. However, considering the number of tree trunks between the transmitter and receiver, it is quite obvious that a brute-force application of MoM is not possible due to the exorbitant memory and computation time requirements. To make the solution tractable while maintaining the model fidelity, three techniques are proposed: 1) simplification of the MoM formulation noting that tree trunks are sparsely distributed, 2) simplification base on physical insight by noting that the scattered fields from tree trunks between the transmitter and receiver, but distant from them, are almost in-phase, and 3) simplification of field computation using the reciprocity theorem. In what follows, the forest model is described first and then the simplified MoM formulation is presented. This model is used to demonstrate that only scatterers near the source and observation points significantly contribute to the field fluctuations. Next the near-field interaction of tree trunks with the source field and lateral waves are computed by application of reciprocity and using the MoM formulation. Finally, numerical simulations are presented, where the spatial decorrelations in a forest environment are examined and the performance of the antenna arrays and spatial diversity schemes evaluated.

2 Forest Model

The model presented in this section is suitable for predicting the statistics of the field radiated by an elementary antenna embedded in a forest environment and is valid for frequencies up to UHF. As mentioned earlier, since the dimensions of leaves and branches are small compared to the wavelength, the forest canopy can be modeled by a homogeneous dielectric . However, the trunks whose dimensions are comparable to or larger than a wavelength must be treated separately. Figure 1 shows the geometry of the wave propagation problem where both the source and observation points are within a dielectric slab with effective permittivity ε_{eff}. This dielectric slab is assumed to have a smooth lower interface with a dielectric half-space (representing the ground) and a rough upper interface with air. Tree trunks are modeled by dielectric cylinders perpendicular to the ground plane having a relatively large height-to-diameter ratio. Ignoring the scattering from tree trunks, the mean field at the receiver is composed of the following components as depicted, in Fig. 2 :

1. geometric optics terms which include the direct propagation between the transmitter and receiver and reflected fields from the upper and lower interfaces. The mean reflected field from the upper interface is reduced exponentially where the exponent is proportional to the rms height of the canopy-air interface roughness [3]. These terms are only important when the receiver is relatively close to the transmitter, otherwise due to the lossy nature of the effective dielectric constant of the foliage these components do not contribute much. Basically the geometric optics terms exponentially decay with distance between the transmitter and the receiver.

2. Lateral wave (a diffraction term) which propagates along the canopy-air interface and decays proportionally to the reciprocal of the radial distance squared($\sim 1/\rho^2$). The path loss associated with the lateral wave propagation increases with increasing foliage density (effective dielectric constant) and decreases by bringing the source or observation points closer to the canopy-air interface. The path loss increases as the canopy-air interface roughness (rms height) relative to wavelength increases. Close examination of the expression representing the lateral wave contribution reveals that the contribution can be attributed to a

ray that emanates from the source along the critical angle, propagates along the canopy-air interface, and arrives at the receiver along the critical angle as shown in Fig. 2. Other higher order lateral wave contributions also exist, of which two are significant. One corresponds to the ray emitted from the source which reflects from the ground boundary, propagates along the critical angle and then travels along the interface (E_{GL}). The other one is the reflected lateral wave from the ground which arrives at the receiver (E_{LG}). These two contributions are also shown in Fig. 2

The expression for the electric field resulted from the asymptotic expansion of the spectral integral around the branch cut (Lateral wave) for a half-space dielectric with a smooth boundary is given by [2]

$$\vec{\mathbf{E}}_L = \frac{jIlZ_0}{2\pi(1-\kappa)^{1/4}} \frac{e^{-jk_0\rho}e^{-jk_1\sqrt{1-\kappa}(h+h')}}{\left[(h+h')\sqrt{\kappa}-\rho\sqrt{1-\kappa}\right]^{3/2}\rho^{1/2}} \bar{\bar{\mathbf{A}}}\cdot\hat{l} \tag{1}$$

where ρ is the radial distance between the transmitter and receiver, h and h' are the depth of transmitter and receiver below the canopy-air interface, Il is the current moment of the dipole, and k_0 and Z_0 are, respectively, the propagation constant and the characteristic impedance of free-space. In (1) the unit vector, $\hat{l}$, denotes the dipole orientation $\kappa = 1/\varepsilon_{eff}$ where ε_{eff} is the effective dielectric constant of foliage, and $\bar{\bar{\mathbf{A}}}$ is a symmetric dyad given by

$$\bar{\bar{\mathbf{A}}} = \begin{bmatrix} \frac{\cos^2\phi-\kappa}{\kappa} & \frac{\cos\phi\sin\phi}{\kappa} & \sqrt{\frac{1-\kappa}{\kappa}}\cos\phi \\ \frac{\cos\phi\sin\phi}{\kappa} & \frac{\sin^2\phi-\kappa}{\kappa} & \sqrt{\frac{1-\kappa}{\kappa}}\sin\phi \\ \sqrt{\frac{1-\kappa}{\kappa}}\cos\phi & \sqrt{\frac{1-\kappa}{\kappa}}\sin\phi & 1 \end{bmatrix} \tag{2}$$

Here ϕ is a cylindrical coordinate angle representing the location of observation point with respect to the transmitter. Assuming the canopy height is H, the lateral wave from the image of the source in the ground plane (E_{GL}) is given by

$$\vec{\mathbf{E}}_{GL} = \frac{jIlZ_0}{2\pi(1-\kappa)^{1/4}} \frac{e^{-jk_0\rho}e^{-jk_1\sqrt{1-\kappa}(2H-h+h')}}{\left[(2H-h+h')\sqrt{\kappa}-\rho\sqrt{1-\kappa}\right]^{3/2}\rho^{1/2}} \bar{\bar{\mathbf{A}}}\cdot\bar{\bar{\mathbf{R}}}\cdot\hat{l} \tag{3}$$

where $\bar{\bar{\mathbf{R}}}$ is the reflectivity matrix given by

$$\bar{\bar{\mathbf{R}}} = \begin{bmatrix} \sin^2\phi R_\perp - \cos^2\phi R_{||} & -\sin\phi\cos\phi(R_\perp + R_{||}) & 0 \\ -\sin\phi\cos\phi(R_\perp + R_{||}) & \cos^2\phi R_\perp - \sin^2\phi R_{||} & 0 \\ 0 & 0 & R_{||} \end{bmatrix} \tag{4}$$

and $R_\perp$ and $R_\parallel$ are the Fresnel reflection coefficients of the ground evaluated at the critical angle $\theta_c = \sin^{-1}\sqrt{\kappa}$. Similarly E_{LG} is given by

$$\vec{\mathbf{E}}_{LG} = \frac{jIlZ_0}{2\pi(1-\kappa)^{1/4}} \frac{e^{-jk_0\rho}e^{-jk_1\sqrt{1-\kappa}(2H+h-h')}}{\left[(2H+h-h')\sqrt{\kappa}-\rho\sqrt{1-\kappa}\right]^{3/2}\rho^{1/2}} \bar{\bar{\mathbf{R}}}\cdot\bar{\bar{\mathbf{A}}}\cdot\hat{l} \tag{5}$$

In (3) and (5) the effects of the ground surface wave is ignored. The phase term of equation (1) indicates that the lateral wave is locally a plane wave propagating along the direction $\vec{k}_1 = k_0\left[\cos\phi\hat{x} + \sin\phi\hat{y} - \sqrt{\frac{1-\kappa}{\kappa}}\hat{z}\right]$ and also noting $\vec{\mathbf{E}}_L \cdot \vec{k}_1 = 0$. The expression for the lateral wave contribution for rough canopy-air interface is more complicated than (1), but the mean-field still represents a plane wave locally [2]. To include the effects of scattering from tree trunks consider the geometry of the problem as shown in Fig. 3. The source and its image excite polarization currents in the dielectric cylinders which in turn re-radiate and produce multiple scattering among the tree trunks and secondary lateral waves (lateral waves generated by scattering from the tree trunks) that arrive at the receiver. The sum of all lateral and the secondary lateral waves will be referred to as the total incident wave in the vicinity of all receiver. The total incident wave excites polarization currents within the dielectric cylinders (tree trunks) near the receiver, which re-radiate and together with the total incident fields constitute the total field at the receiver. A formal solution to the problem can be obtained rather easily by casting the formulation in terms of an integral equation for the polarization currents induced inside the dielectric cylinders. However, the brute-force solution of the integral equation using the method of moments is not tractable because of the large number of unknowns and the complex nature of the dyadic Green's function of the problem. To make the problem tractable an accurate approximate solution is sought. An accurate approximate solution, considering the physics of the problem, can be obtained as will be shown in the following section.

3 Reduced Problem

As mentioned before, estimation of field fluctuations in a forest environment requires the computation of multiple scattering effects, as well as the interaction of lateral waves with large number of dielectric cylinders. It is expected that the induced polarization currents in cylinders near the source be much stronger than those in cylinders distant from the source. Also the contribution to the received fields from cylinders in the vicinity of the receiver is expected to be high. Although the contribution to the scattered fields from individual cylinders between the transmitter and receiver which are not in the close proximity of the either is relatively small, one may argue that there are many such cylinders and the overall contribution may be significant. Considering the path-length between the transmitter to a cylinder and from the cylinder to the receiver, and noting that the scattering is strongest in near forward direction, the scattered fields from these cylinders arrive almost in-phase. Therefore the scattered field from cylinders that are not close to the receiver and transmitter are not expected to contribute to the field fluctuations significantly. The effect of these scatterers may be accounted for by replacing them with an effective dielectric constant. To examine the validity of these postulations, a 2-D medium consisting of infinite cylinders, excited by line source, is considered. Figure 4(a) shows the geometry of the problem where a line source capable of supporting z-directed current (TM case) or ρ-directed current (TE case) is used as the transmitter. Figure 4(b) shows the geometry of the reduced problem where the scatterers that are not close to the receiver and transmitter are replaced with a effective dielectric constant. Our objective here is to show that the mean and variance of the field in the original problem and the reduced problem are the same. The method of moments is used to solve these problems for a given arrangement of cylinders. Then using Monte-Carlo simulation, the desired field statistics are computed. In the implementation of the Monte-Carlo simulation, the boundaries of the medium must be varied randomly by a few wavelengths to avoid coherence effects [4]. Position of cylinders are determined by a random number generator. In this filling process the distance of a new cylinder is measured from the previous ones to ensure a minimum distance between the tree trunks. The filling process for each medium realization is continued until a desired number density is reached.

3.1 The Method of Moments for M-body Sparse Scatterers

In this section a numerical technique based on the method of moments appropriate for a relatively large number of sparsely distributed dielectric cylinders, illuminated at oblique incidence is described. The integral equation formulation for the induced polarization current for infinite cylinders whose axes are parallel to the z-axis is given by

$$\frac{1}{\varepsilon_1 - 1}\vec{\mathbf{J}}(\vec{\rho}) = -jk_0Y_0\vec{\mathbf{E}}^i + \sum_{n=1}^{N}\int_{s'_n} \bar{\bar{G}}(|\vec{\rho} - \vec{\rho}'|) \cdot \vec{\mathbf{J}}(\vec{\rho}')ds' \qquad (6)$$

where $\vec{\mathbf{E}}^i$ represents the incident field having a propagation constant k_0, ε_r is the relative dielectric constant of the cylinders and s'_n is the cross section of the nth cylinder. The explicit expression for the dyadic Green's function is given by

$$\bar{\bar{G}}(|\vec{\rho}-\vec{\rho}'|)=\frac{j}{4}\begin{bmatrix} k_0^2+\frac{\partial^2}{\partial x^2} & \frac{\partial^2}{\partial x\partial y} & -jk_z\frac{\partial}{\partial x} \\ \frac{\partial^2}{\partial x\partial y} & k_0^2+\frac{\partial^2}{\partial y^2} & -jk_z\frac{\partial}{\partial y} \\ -jk_z\frac{\partial}{\partial x} & -jk_z\frac{\partial}{\partial y} & k_\rho^2 \end{bmatrix} H_0^{(1)}(k_\rho|\vec{\rho}-\vec{\rho}'|) \tag{7}$$

where k_z is the propagation constant of the incident wave along the z-axis, and $k_\rho = \sqrt{k_0^2-k_z^2}$. Following the standard procedure of the MoM, the cross section of the cylinders are descretized into small cells over which the current distribution may be considered a constant vector. Using point matching, the integral equation(6) can be cast in terms of a matrix equation of the form

$$\bar{\bar{Z}}\cdot\vec{\mathbf{J}}=\vec{\mathbf{V}} \tag{8}$$

where $\bar{\bar{Z}}$ is known as the impedance matrix. The size of the impedance matrix is proportional to the total number of cells for all N cylinders and is a limiting factor with regard to computer memory. Noting that in a fully-grown forest, the tree trunks are sparsely distributed, storage of all elements of $\bar{\bar{Z}}$ can be avoided. In this case, the impedance matrix of the individual cylinders (block diagonal elements) are computed and stored. This can be limited to a few matrices corresponding a small number of cylinder, with different radii that represents the variability in tree trunk diameters. Next the impedance matrix elements or the pairwise interaction between the cells at the center of the cylinders are computed and stored. The interaction between other elements are computed in an approximate manner as needed in the program and are not stored. For further clarification, consider the ith and the jth cylinders in the global coordinate system whose centers are respectively located at (X_i,Y_i) and (X_j,Y_j) as shown in Fig. 5. Suppose the interaction between two cells, whose local coordinate in the ith and jth system are respectively given by (x_i',y_i') and (x_j',y_j'), are needed. In this case, using the Taylor series expansion,

$$|\vec{\rho}_i-\vec{\rho}_j|\approx\rho_{ij}+\frac{(X_i-X_j)(x_i'-x_j')}{\rho_{ij}}+\frac{(Y_i-Y_j)(y_i'-y_j')}{\rho_{ij}} \tag{9}$$

where $\rho_{ij}=\sqrt{(X_i-X_j)^2+(Y_i-Y_j)^2}$ is the distance between the centers of the cylinders. The approximation in (9) can be used in the evaluation of integrals needed for the computation of matrix elements. For example,

$$\begin{aligned} I_{i'j'} &= \int_{x_j'-\triangle/2}^{x_j'+\triangle/2}\int_{y_j'-\triangle/2}^{y_j'+\triangle/2} H_0^{(1)}(k_\rho|\vec{\rho}_i-\vec{\rho}_j|)dx'dy' \\ &\approx \alpha_n H_0^{(1)}(k_\rho\rho_{ij})\cdot e^{i\Phi_{i'j'}} = I_{ij}\cdot e^{i\Phi_{i'j'}} \end{aligned}$$

where $\Phi_{i'j'} = k_\rho \left[(X_i - X_j)(x'_i - x'_j) + (Y_i - Y_j)(y'_i - y'_j)\right] / \rho_{ij}$, and $\alpha_n = \triangle^2 \left[1 - \frac{(k_\rho \triangle)^2}{24}\right]$. As mentioned earlier I_{ij} is stored and $I_{i'j'}$ is computed as needed . Using a sparse matrix storage scheme [5] and storing a small number of terms like, $(X_i - X_j)/\rho_{ij}$, $(Y_i - Y_j)/\rho_{ij}$, $(x'_i - x'_j)$ and $(y'_i - y'_j)$, the needed memory size is reduced. Similarly for other terms of the dyadic Green's function that require spatial derivatives, we can use the following approximations:

$$\left(\frac{\partial^2}{\partial u^2} + k_\rho^2\right) I_{i'j'} = \frac{\alpha_n k_\rho^2}{2} \left\{H_0^{(1)}(k_\rho \rho_{ij}) \pm H_2^{(1)}(k_\rho \rho_{ij})[\cos^2\theta_{i'j'} - \sin^2\theta_{i'j'}]\right\} \cdot e^{\Phi_{i'j'}} \quad (10)$$

where the positive and negative signs are used for $u = x$ for $u = y$, respectively. Also in (10)

$$\cos^2\theta_{i'j'} - \sin^2\theta_{i'j'} \approx \frac{\cos^2\theta_{ij} - \sin^2\theta_{ij} + (a_1 - a_2)/\rho_{ij}}{1 + (a_1 + a_2)/\rho_{ij}^2},$$

$$a_1 = 2(X_i - X_j)(x'_i - x'_j), \quad a_2 = 2(Y_i - Y_j)(y'_i - y'_j) \quad \theta_{ij} = \tan^{-1}\left[(Y_i - Y_j)/(X_i - X_i)\right]$$

Moreover

$$\frac{\partial I_{i'j'}}{\partial u} \approx \frac{\alpha_n k_\rho^2}{2} \left[H_0^{(1)}(k_\rho \rho_{ij}) + H_2^{(1)}(k_\rho \rho_{ij})\right] (u_{i'} - u_{j'}) \cdot e^{\Phi_{i'j'}}$$

for $u = x$ or y. Finally for the $\frac{\partial^2 I_n}{\partial x \partial y}$ term, we can use

$$\frac{\partial^2 I_{i'j'}}{\partial x \partial y} = \frac{(X_i - X_j)(Y_i - Y_j) + (X_i - X_j)(y'_i - y'_i) + (Y_i - Y_j)(x'_i - x'_j)}{\rho_{ij}^2 + (a_1 + a_2)} \cdot k_\rho^2 H_2^{(1)}(k_\rho \rho_{ij}) \cdot e^{\Phi_{i'j'}}$$

Table 1 shows the sub-matrices and arrays that are stored in this scheme for M identical cylinder, each descretized into N pixels.

3.2 Statistically Equivalent 2-D Medium

In this section, a complete MoM solution for a large number of cylinders is used to verify the conjectures mentioned for the reduced problem. However before presenting these results, the accuracy of the approximate MoM solution for the M-body, sparse scatterers problem is evaluated. An array of 50 equally spaced dielectric cylinders with $\varepsilon_r = 5 + j$ and radius 0.3m are arranged along Y-axis with a spacing of 2m. This array is illuminated by a 50MHz plane wave at an oblique incident angle $\theta = 60^o$ and TE polarization. The scattered field computed by the exact MoM and the approximate MoM are compared along the curve, $y = x\ln x$. Small discrepancies are observed between the two solutions which are plotted in Fig. 7 as relative percentage error. Many other cases were also examined and it was found that when the average ratio of cylinder spacing to cylinder radius is larger than 5, the approximate MoM is capable of producing very accurate results. Having confidence in the approximate MoM algorithm, simulation of wave propagation in a sparse random medium is carried out. A large number of dielectric cylinders (2000) confined in a rectangular box as shown in Fig. 4, are considered. The box is chosen to have average dimensions of 50m $\times$ 800m. As mentioned earlier, to eliminate the coherence effect of finite box size (see [4]) the dimension of the box is varied by at least one wavelength in the Monte Carlo simulation. A line source is used as the transmitter, at location (x = 20, y = 25) and the field is calculated at a point located at (x = 780, y = 25). Many sample media are generated and the MoM solution for each sample is used to construct approximate statistics of the field at the receiving point. In specifics, the mean and standard deviation (SDV) of the field is monitored and the simulation is continued up until a convergence is reached. The contribution to the total mean-field and standard deviation from scatterers in range bins of 20m wide are stored separately and are plotted in Fig. 8, for three frequencies of 10, 30, and 50MHz. It is interesting to note that the contribution to the total standard deviation (field fluctuations) are mainly dominated by the scatterers near the source and observation points. This indicates that scatterers between, but not close to, the transmitter and receiver are not significant sources of field fluctuations and can be replaced with an effective dielectric constant. The geometry of the reduced problem is shown in Fig. 4(b) where 600 cylinders near the source and 250 cylinders near the observation point are kept and the rest are replaced with an effect dielectric medium. In this simulation a cylinder density $0.05/\text{m}^2$ is used and the comparison of standard deviation for the complete and reduced problems for the TM case is shown in Fig. 9. As can be seen in Fig. 9 the reduced and complete problems produce effectively the same field variations. To examine the field variance only, far less scatterers produce the same result. Figures 10(a), 10(b),and 10(c) show respectively the field variance for TE (10(a), and 10(b)) and TM (10(c)) excitation using 200 cylinders near the source and 150 cylinders near the receiver. For these simulations cylinder density $0.01/\text{m}^2$ is used. For the reduced problem, the effective medium is anisotropic and its dielectric constant is a tensor given by [4],

$$\begin{aligned}\varepsilon_{eff} &= \varepsilon_h + (\varepsilon_i - \varepsilon_h) f \qquad \text{for TM} \\ \varepsilon_{eff} &= \varepsilon_h + f(\varepsilon_i - \varepsilon_h) \frac{2}{\varepsilon_i + \varepsilon_h} \qquad \text{for TE}\end{aligned}$$

4 Fast Field Calculation Based on Reciprocity

As our goal in this investigation is the computation of the fields of a dipole in a forest environment, computation of polarization currents in tree trunks when illuminated by the dipole is needed. Even with the simplification mentioned in the previous section, a 3-D scattering problem which includes more than 200 cylinders is not tractable numerically. In this section, we demonstrate a procedure where, with the help of reciprocity theorem, the 3-D scattering problem is first reduced to an equivalent 2-D problem, which is then solved by the method of moments. To demonstrate this procedure, let us first consider a short dipole near a single cylinder as shown in Fig. 6. The field at the receiver is the sum of the field of the dipole and the radiated field from the polarization current induced in the dielectric cylinder embedded in the canopy above the ground. Since the observation point is in the far-field of the cylinder and the dipole, reciprocity can be applied to simplify the problem. According to the reciprocity theorem [6], the vertical component of the received field for a dipole excitation with orientation $\hat{l}$ is equal to the $\hat{l}$ component of the field near the cylinder of the same dipole oriented vertically and located at the observation point. According to (1), the field of the dipole (in the modified problem) illuminating the dielectric cylinder is locally plane wave. Also noting that the induced polarization currents in a finite, long dielectric cylinder is approximately the same as those of an infinite cylinder having the same radius and dielectric constant [7, 8], the MoM solution for 2-D problems can be used to find the induced polarization currents. Once this polarization current is obtained, the near field can easily be computed and the expression for it given in the Appendix. The same procedure is applied to find the contribution of all cylinders in the vicinity of the transmitter, at the receiver. To compute the effect of the scattered field of cylinders near the receiver, the fields of the dipole and all cylinders in its vicinity are computed at each cylinder location near the receiver. Again these fields are locally plane waves illuminating tree trunks near the receiver at an oblique incidence equal to the critical angle. MoM is used to find the induced polarization current in cylinders near the receiver from which the scattered field is computed. Hence the contribution from all cylinders near the transmitter and receiver are included. To account for the effect of the forest floor, the geometric optics images of the dipole and lateral waves on the ground plane are considered and their contributions are evaluated using a similar procedure. It is worth mentioning that in this case, the number of cylinders which need to be kept in the vicinity of the transmitter and receiver for the reduced problem is expected to be smaller than what was obtained for the 2-D problem. Basically for the dipole excitation where the finite tree trunks are illuminated by the resulting plane waves at oblique incidence, fewer number of cylinders can interact with each other.

5 Numerical Simulation

Based on the rigorous electromagnetic model described in the previous section, a very accurate propagation model which provides a complete channel characterization of forest media is possible through Monte Carlo simulations. To examine the effect of tree trunks on the field of a transmitter in a forest, we first consider a single tree trunk in the near-field of a short dipole. For this example a dielectric cylinder with permittivity $\varepsilon = 5 + j$, radius a = 35cm , and height h_c = 15m is considered in a forest with a canopy having $\varepsilon_{eff} = 1.03 + j0.036$ and height H = 20m. A vertical dipole transmitting at 90MHz is placed at a distance of 10cm from the surface of the cylinder and is moved up and down, and around the cylinder and its field is computed at an observation point 1km away from the cylinder and 5m above the ground plane. Figure 11 shows the normalized z-component of the field at the receiver as a function of dipole height above the ground and for three azimuthal angles around the cylinder. The normalization here is with respect to the field of the dipole in the absence of the cylinder. It is shown that the field at the receiving point fluctuate as the dipole is moved along the cylinder axis. This fluctuation is a result of constructive and destructive interference of the direct field, scattered field from the cylinder, and their images on the ground plane. Since the cylinder radius is small compared to the wavelength a gentle variation is observed as the dipole is moved around the cylinder. It is interesting to note that for most dipole locations, existence of the cylinder in the near-field region of the dipole enhances the field at the receiver since the tree trunk acts as a passive radiator. Next, computation of path-loss and field standard deviation is considered for the forest of the previous example. In this case tree trunks having an average height 15m and height standard deviation of 1m with number density $0.05/\text{m}^2$ and dielectric constant of $\varepsilon = 5 + j$ are also included. As mentioned before, the number of cylinders we need to keep for the reduced problem is expected be smaller than those for the 2-D problem. Here we kept 200 cylinders near the transmitter located at the origin 3m above the ground and 50 cylinders near the receiver located 1km away from the transmitter and 5m above the ground. Table 2 shows the path-loss (with respect to the free-space) of the normal component of the wave and its standard deviation at 50MHz for a vertical dipole. Also shown in Table 2 are the path-loss and standard deviation if 390 cylinders (250 near the source and 140 near the receiver) are kept. The Monte-Carlo simulation converges after about 30 realizations and is shown that the results based on 200 cylinders is very close to those based on 390 cylinders. This convergence test indicates that a moderate number of scatterers (about 200) is sufficient for accurate characterization of field variance. The results also indicate a standard-to-mean ratio deviation of almost unity for the random variable E_z. Using different forest parameters and frequencies (at HF band) random variable E_z showed a similar behavior. That is the statistics of the channel follows Ricean statistics with a standard deviation-to-mean ratio ranging from 0.5 - 2. This is different than Rayleigh statistics which is commonly assumed for communication channels. Significant non-zero mean-field is a direct result of contributions from the lateral wave.

For communication channels with significant multi-path, antenna arrays are usually used to mitigate fading. In this case, because of the existence of a considerable coherent mean-field, antenna arrays may provide coherent gain. To investigate the performance of antenna arrays in a forest environment two basic configurations, namely, broadside and end-fire are considered. For the broadside configuration four vertical dipoles are arranged along a line perpendicular to the line between the transmitter and receiver points with a spacing of $\lambda/2$. The four elements of the end-fire array are placed along the line between the transmitter and receiver separated by $7\lambda/16$ and having a progressive phase factor of $-7\pi/8$ [9]. The field of these two arrays are evaluated and the path-loss and SDV-to-mean ratio is reported in Table 2. It is found that the broadside array has about 11dB less path-loss than the end-fire array. This is very close to the 12-dB gain of a 4-element array and indicates that the field of all four elements of the broadside array arrive at the receivers almost coherently whereas those of the end-fire array are incoherent. This behavior can be understood by considering the spatial correlation function in the forest environment. The spatial correlation coefficient between two points $\vec{r}_1$ and $\vec{r}_2$ is defined as [11]

$$C_s(\vec{r}_1, \vec{r}_2) = \frac{\text{Cov}\left[\vec{E}(\vec{r}_1), \vec{E}(\vec{r}_2)\right]}{\sigma_{E_1}\sigma_{E_2}}$$

where σ_{E_1} and σ_{E_2} are the variance of the field at $\vec{r}_1$ and $\vec{r}_2$ respectively. A grid of nine points along x-axis (direction between the transmitter and receiver) and nine points along y-axis separated by $\lambda/2$ that is 1km from the transmitter is generated and components of the electric field are calculated many times for different cylinder arrangements in order to compute the correlation coefficient. Figures 12(a) and 12(b) show the magnitude and phase of $C_s(\vec{r}_1, \vec{r}_2)$ along the x- and y-axis respectively. It is shown that the signal along y-axis remains coherent over much larger distance than along x-axis. This result is in agreement with the observed behavior of broadside and end-fire antenna arrays. Sampling theorem [10, 11, 12] is used to generate the smooth curves that goes in between the discrete points in Figs. 12.

To study the path-delay effects, the impulse response of the channel must be characterized. The impulse response cannot be directly determined. However, by characterizing frequency response over a wide bandwidth the impulse response can be determined by applying the Fourier transformation to generate time domain (range) information. Direct application of FFT is not computationally efficient considering the number of frequency points which are needed in order to avoid aliasing. For example, with a receiver at a distance of 1.5km from the transmitter a maximum frequency spacing of 200 KHz is required. A pulse width of 12.5 nsec corresponds to 80MHz bandwidth which can be used to resolve path delays 3.75m apart. In this case for each forest realization 400 frequency points must be simulated. To circumvent this difficulty we used a non-uniform frequency sampling scheme. Gauss quadrature integration [12] is used to evaluate the Fourier transform. The order of Legendre polynomial should be chosen so that the minimum distance between the zeros of the Legendre polynomial is smaller than the minimum frequency spacing required to avoid aliasing (200 KHz for the above example). Impulse response of the forest considered in the previous example is simulated at HF using a bandwidth of 10 - 90MHz. A receiver at a distance of 1km from the transmitter is considered. It is expected that the dispersion (pulse spreading) will be observed due to multiple scattering among tree trunks and the frequency dependent path-loss of lateral waves. Figure 13 shows the frequency response of the received field in the absence of the tree trunks and the ground plane for two cases: 1) smooth canopy-air interface and 2) rough canopy-air interface with a rms height 2m. The transmitter and receiver are 17m and 15 m below the interface and the effective dielectric constant of the canopy is $\varepsilon_{eff} = 1.03 + j0.036$, as before. It is shown that the canopy-air interface roughness increase the dispersion. Figure 14 shows the impulse response of the forest channel including tree trunks and the underlying ground plane. For the calculation of the impulse response, Gauss quadrature method with 40 points is used over the frequency range of 10 - 90MHz. A Gaussian pulse is assumed to be transmitted which is also plotted in Fig. 14 for comparison. The amplitude of the transmitted pulse is multiplied by the path-loss and delayed by the free-space distance between the transmitter and receiver. As can be seen in Figure 14, pulse spreading and ringing are observed which are the result of dispersion and multipath. The last simulation demonstrates performance of different spatial diversity schemes. Three antennae 1.5λ apart are aligned x-axis placed 1km away from a transmitter operating at 50MHz in the forest. Three diversity schemes are examined: 1) selective diversity (SD), 2) equal gain combining (EG), and 3) maximal ratio (MR) combining [13]. In SD scheme the detector simply chooses the output of the receiver with the highest receiver power. In EG combining method the detector is provided with the averaged detected signal from each receiver (a weighting factor of $w_i = 1/3$ is used in the combining). In the diversity scheme based on maximal ratio combining a weighting factor proportional to the signal power is used to combine the signals from the receiver ($w_i = \frac{P_i}{\sum_{j=1}^{3} P_j}$). To assess the performance of the above-mentioned diversity combining methods, cumulative distribution function(CDF) of fading depth for each diversity scheme is calculated using a Monte-Carlo simulations. The fading depth is defined as power level at the output of each combiner (SD, EG, or MR) above the average power in dB ($10\log(|\vec{E}|^2/\langle|\vec{E}|^2\rangle)$). Figure 15 shows the CDF of fading depth for SD, EG, and MR combining methods. Also shown is the CDF of one of the channels. It is shown that the performance of all three diversity scheme is approximately the same for this problem.

6 Concluding Remarks

A complete wave propagation model capable of predicting path-loss, time delay response, and coherent frequency response in a forest environment is developed. The model is based on a hybrid analytical and numerical approach which allows for efficient computation of important channel properties without compromising accuracy. The model accounts for multiple scattering among tree trunks and includes the effects of the forest ground plane. Branches and leaves are represented by an effective dielectric constant which limits the range of validity of the model up to UHF. Except for the path-loss calculation other statistical channel characteristics are determined using a Monte-Carlo simulation. In general, propagation simulations are slower at higher frequencies which depending on the computer platform also constitutes a limit for the highest frequency. The model is used to simulate performance of antenna arrays and different diversity schemes. It is shown that because of a significant non-zero mean-field, spatial diversity only improves the channel fading moderately and there is not a significant difference between different diversity combining schemes. The model presented in this paper shows the possibility of constructing a very accurate physics-based propagation model capable of end-to-end channel simulations.

Appendix: MoM Formulation

In this appendix a general method of moments formulation for 2-D dielectric objects with possible dielectric inhomogeneity is provided. The axis of a cylinder with an arbitrary cross section is assumed to be parallel to the z axis and the surrounding medium is assumed to be free-space. Let the relative permittivity of the cylinder be $\epsilon_r(x, y)$ and its relative permeability be unity ($\mu_r = 1$). The cylinder perturbs the incident field and the difference between the perturbed (total) and incident field is known as the scattered field; thus,

$$\mathbf{E}^t(\bar{\rho}) = \mathbf{E}^i(\bar{\rho}) + \mathbf{E}^s(\bar{\rho}), \tag{A.1}$$

where $\bar{\rho}$ is the position vector in cylindrical coordinates. From Maxwell's equations it can be shown that a volumetric current density of the form

$$\mathbf{J}_e(\bar{\rho}) = -ik_0Y_0[\epsilon_r(\bar{\rho}) - 1]\mathbf{E}^t(\bar{\rho}), \qquad \bar{\rho} \in S, \tag{A.2}$$

known as the polarization current, in free space, can replace the cylinder to reproduce the scattered field. Therefore the scattered field, can be obtained from:

$$\mathbf{E}^s(\bar{\rho}) = -ik_0Z_0 \int_s \mathbf{J}_e(\bar{\rho'}) \cdot \overline{\overline{G}}(\bar{\rho}, \bar{\rho'}) \mathrm{d}s', \tag{A.3}$$

where $\overline{\overline{G}}(\bar{\rho}, \bar{\rho'})$ is the two-dimensional dyadic Green's function given by ([7]). Using (A.1), (A.2),and (A.3) an integral equation for the unknown polarization current can be obtained,

$$\mathbf{J}_e(\bar{\rho}) = -ik_0Y_0[\epsilon_r(\bar{\rho}) - 1]\{\mathbf{E}^i(\bar{\rho}) - ik_0Z_0 \int_s \mathbf{J}_e(\bar{\rho'}) \cdot \overline{\overline{G}}(\bar{\rho}, \bar{\rho'}) \mathrm{d}s'\}.$$

Assuming the incident field is illuminating the cylinder at an oblique incident angle θ_i ($k_z^i = k_0 \cos\theta_i$), in order to satisfy the phase matching condition, all field and polarization current components have a z-dependance of the form $e^{-k_z z}$. If the incident field is normal incident $k_z^i = 0$, then the problem may be decoupled into TM and TE problems. For the TM case the incident, scattered, and hence, the polarization current have only z components and the integral equation is simplified to

$$\begin{aligned} J_z(x, y) = \; & -ik_0Y_0[\epsilon_r(x, y) - 1]E_z^i(x, y) + i\tfrac{k_0^2}{4}[\epsilon_r(x, y) - 1] \\ & \int_s J_z(x', y')H_0^{(1)}(k_\rho \mid \vec{\rho} - \vec{\rho'} \mid)\mathrm{d}x'\mathrm{d}y'\}. \end{aligned} \tag{A.4}$$

In the TE case both x and y components of the polarization current are induced and they satisfy the following coupled integro-differential equations

$$\begin{aligned} J_x(x,y) = \; & -ik_0Y_0[\epsilon_r(x,y)-1]E_x^i(x,y)+\tfrac{i}{4}[\epsilon_r(x,y)-1] \\ & \cdot\{(\tfrac{\partial^2}{\partial x^2}+k_0^2)\int_s J_x(x',y')H_0^{(1)}(k_\rho \mid \vec{\rho}-\vec{\rho'} \mid)\mathrm{d}x'\mathrm{d}y' \\ & +\tfrac{\partial^2}{\partial x\partial y}\int_s J_y(x',y')H_0^{(1)}(k_\rho \mid \vec{\rho}-\vec{\rho'} \mid)\mathrm{d}x'\mathrm{d}y'\} \\ J_y(x,y) = \; & -ik_0Y_0[\epsilon_r(x,y)-1]E_y^i(x,y)+\tfrac{i}{4}[\epsilon_r(x,y)-1] \\ & \cdot\{\tfrac{\partial^2}{\partial x\partial y}\int_s J_x(x',y')H_0^{(1)}(k_\rho \mid \vec{\rho}-\vec{\rho'} \mid)\mathrm{d}x'\mathrm{d}y' \\ & +(\tfrac{\partial^2}{\partial y^2}+k_0^2)\int_s J_y(x',y')H_0^{(1)}(k_\rho \mid \vec{\rho}-\vec{\rho'} \mid)\mathrm{d}x'\mathrm{d}y'\} \end{aligned} \tag{A.5}$$

The resultant integro-differential operators obtained for the polarization current do not impose any restriction on the functional form of the current, in particular, the pulse function is in the domain of the operators. It should be noted here that the kernel of the integral equation (Green's function) for the TE case is more singular $(\frac{1}{\rho^2})$ than for the TM case $(\ln\rho)$. There is no known solution for these integral equations in general, but their forms are amenable to numerical solution. An approximate numerical solution for the integral equations can be obtained using the method of moment in conjunction with the pulse expansion function and the point-matching technique. The cross section of the scatterer is divided into N rectangular cells that are small enough so that the polarization current and relative permittivity can be assumed to be constant. The unknown current can be approximated by

$$J_p(x,y)=\sum_{m=1}^{N} J_mP(x-x_m,y-y_m), \qquad p=x,y,\text{ or } z \tag{A.6}$$

where J_m are the unknown coefficients to be determined and $P(x-x_m,y-y_m)$ is the pulse function defined by

$$P(x-x_m,y-y_m)=\begin{cases} 1 & \mid x-x_m \mid < \frac{\Delta X_m}{2}, \quad \mid y-y_m \mid < \frac{\Delta Y_m}{2} \\ 0 & \text{otherwise} \end{cases} \tag{A.7}$$

By inserting the current as expanded in (A.6) into the integral equations (A.4) and (A.5) and then setting the observation point at the center of the m$^{\text{th}}$ cell, a linear set of equations is formed. In matrix notation, these linear equations can be represented by

$$\mathcal{Z}^{TM}\mathcal{J}_z=\mathcal{E}_z \tag{A.8}$$

for the TM case, where $\mathcal{Z}^{TM}$ is the impedance matrix, $\mathcal{J}_z$ is the unknown vector, and $[\mathcal{E}_z]$ is the excitation vector. Similarly for the TE case the coupled integral equation

(A.5) in matrix form becomes

$$\begin{aligned} \mathcal{Z}_1^{TE}\mathcal{J}_x + \mathcal{Z}_2^{TE}\mathcal{J}_y &= \mathcal{E}_x \\ \mathcal{Z}_3^{TE}\mathcal{J}_x + \mathcal{Z}_4^{TE}\mathcal{J}_y &= \mathcal{E}_y \end{aligned} \tag{A.9}$$

where as before $\mathcal{Z}_1^{TE}, \cdots, \mathcal{Z}_4^{TE}$ are $N \times N$ impedance matrices and $\mathcal{E}_x$ and $\mathcal{E}_y$ are the excitation vectors. The above coupled matrix equation can be represented by a $2N \times 2N$ matrix equation similar to (A.8). For the general case of obliquely incident, all three current components are coupled, and the final matrix becomes

$$\begin{aligned} \mathcal{Z}_1^{TE}\mathcal{J}_x + \mathcal{Z}_2^{TE}\mathcal{J}_y + \mathcal{Z}^x\mathcal{J}_z &= \mathcal{E}_x \\ \mathcal{Z}_3^{TE}\mathcal{J}_x + \mathcal{Z}_4^{TE}\mathcal{J}_y + \mathcal{Z}^y\mathcal{J}_z &= \mathcal{E}_y \\ \mathcal{Z}^x\mathcal{J}_x + \mathcal{Z}^y\mathcal{J}_y + k_\rho^2\mathcal{Z}^{TM}\mathcal{J}_z &= \mathcal{E}_z \end{aligned} \tag{A.10}$$

Although the variation of the polarization current and dielectric constant over each cell is ignored, this cannot be done for the Green's function. Actually for cells close to the observation point, the Green's function varies rapidly and its contribution must be evaluated more precisely. Let us denote the function representing the Green's function contribution by

$$I_n(x,y) = \int_{x_n-\frac{\Delta X_n}{2}}^{x_n+\frac{\Delta X_n}{2}} \int_{y_n-\frac{\Delta Y_n}{2}}^{y_n+\frac{\Delta Y_n}{2}} H_0^{(1)}(k_\rho\sqrt{(x-x')^2+(y-y')^2})\mathrm{d}x'\mathrm{d}y' \tag{A.11}$$

where $1 \le n \le N$. If the observation point (x,y) is different from (x_n, y_n) , then the integrand in (A.11) is not singular and since $\mid x' - x_n \mid \le \frac{\Delta X_n}{2}$ and $\mid y' - y_n \mid \le \frac{\Delta Y_n}{2}$, its Taylor series expansion may be substituted. By retaining the terms up to the cubic order in the expansion of the Hankel function, $I_n(x,y)$ is found to be:

$$\begin{aligned} I_n(x,y) = \ & \Delta X_n \Delta Y_n\{H_0^{(1)}(k_\rho\sqrt{(x-x_n)^2+(y-y_n)^2}) + \tfrac{(k_\rho \Delta X_n)^2}{24}A(x-x_n, y-y_n) + \\ & \tfrac{(k_\rho \Delta Y_n)^2}{24}B(x-x_n, y-y_n)\}, \end{aligned} \tag{A.12}$$

where

$$A(x-x_n, y-y_n) = A(r_n, \theta_n) = -H_0^{(1)}(k_\rho r_n)\cos^2\theta_n + \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n), \tag{A.13}$$

$$B(x-x_n, y-y_n) = B(r_n, \theta_n) = -H_0^{(1)}(k_\rho r_n)\sin^2\theta_n + \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n}(\sin^2\theta_n - \cos^2\theta_n) \tag{A.14}$$

with the following definition for a pair of local polar coordinates:

$$\begin{aligned} r_n = & \sqrt{(x-x_n)^2+(y-y_n)^2} \\ \theta_n = & \arctan\left(\frac{x_n-x}{y_n-y}\right). \end{aligned} \tag{A.15}$$

The first order derivatives of $I_n(x,y)$ are also easily calculated and are given by

$$\begin{aligned} \frac{\partial I_n(x,y)}{\partial x} = & \Delta X_n \Delta Y_n \{-H_1^{(1)}(k_\rho \sqrt{(x-x_n)^2+(y-y_n)^2}) \cos\theta_n + \\ & \frac{(k_\rho \Delta X_n)^2}{24} \frac{\partial}{\partial x} A(x-x_n, y-y_n) + \frac{(k_\rho \Delta Y_n)^2}{24} \frac{\partial}{\partial x} B(x-x_n, y-y_n)\} \end{aligned} \tag{A.16}$$

$$\begin{aligned} \frac{\partial I_n(x,y)}{\partial y} = & \Delta X_n \Delta Y_n \{-H_1^{(1)}(k_\rho \sqrt{(x-x_n)^2+(y-y_n)^2}) \sin\theta_n + \\ & \frac{(k_\rho \Delta X_n)^2}{24} \frac{\partial}{\partial y} A(x-x_n, y-y_n) + \frac{(k_\rho \Delta Y_n)^2}{24} \frac{\partial}{\partial y} B(x-x_n, y-y_n)\} \end{aligned} \tag{A.17}$$

where

$$\begin{aligned} \frac{\partial}{\partial x} A(r_n,\theta_n) = & H_1^{(1)}(k_\rho r_n)\cos^3\theta_n - H_0^{(1)}(k_\rho r_n)\frac{2\cos\theta_n \sin^2\theta_n}{r_n} + \\ & \frac{\cos\theta_n}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n)\left[k_\rho H_1^{(1)'}(k_\rho r_n) - \frac{H_1^{(1)}(k_\rho r_n)}{r_n}\right] + \\ & \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n} \cdot \frac{4\cos\theta_n \sin^2\theta_n}{r_n} \end{aligned} \tag{A.18}$$

$$\begin{aligned} \frac{\partial}{\partial x} B(r_n,\theta_n) = & H_1^{(1)}(k_\rho r_n)\cos^3\theta_n - H_0^{(1)}(k_\rho r_n)\frac{2\cos\theta_n \sin^2\theta_n}{r_n} - \\ & \frac{\cos\theta_n}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n)\left[k_\rho H_1^{(1)'}(k_\rho r_n) - \frac{H_1^{(1)}(k_\rho r_n)}{r_n}\right] - \\ & \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n} \cdot \frac{4\cos\theta_n \sin^2\theta_n}{r_n} \end{aligned} \tag{A.19}$$

$$\begin{aligned} \frac{\partial}{\partial y} A(r_n,\theta_n) = & H_1^{(1)}(k_\rho r_n)\cos^2\theta_n \sin\theta_n + H_0^{(1)}(k_\rho r_n)\frac{2\cos^2\theta_n \sin\theta_n}{r_n} + \\ & \frac{\sin\theta_n}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n)\left[k_\rho H_1^{(1)'}(k_\rho r_n) - \frac{H_1^{(1)}(k_\rho r_n)}{r_n}\right] + \\ & \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n} \cdot \frac{4\cos^2\theta_n \sin\theta_n}{r_n} \end{aligned} \tag{A.20}$$

$$\begin{aligned} \frac{\partial}{\partial y} B(r_n,\theta_n) = & H_1^{(1)}(k_\rho r_n)\cos^2\theta_n \sin\theta_n + H_0^{(1)}(k_\rho r_n)\frac{2\cos^2\theta_n \sin\theta_n}{r_n} - \\ & \frac{\sin\theta_n}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n)\left[k_\rho H_1^{(1)'}(k_\rho r_n) - \frac{H_1^{(1)}(k_\rho r_n)}{r_n}\right] - \\ & \frac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n} \cdot \frac{4\cos^2\theta_n \sin\theta_n}{r_n} \end{aligned} \tag{A.21}$$

The second order derivatives of $I_n(x,y)$ are also needed for calculation of the impedance matrix elements for the TE case, and are given by

$$
\begin{aligned}
(\tfrac{\partial^2}{\partial x^2}+k_0^2)I_n(x,y) = \; & \Delta X_n \Delta Y_n k_\rho^2 \{H_0^{(1)}(k_\rho r_n)\sin^2\theta_n + \tfrac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n}(\cos^2\theta_n - \sin^2\theta_n) \\
& + \tfrac{\Delta X^2}{24}(\tfrac{\partial^2}{\partial x^2}+k_0^2)A(r_n,\theta_n) + \tfrac{\Delta Y^2}{24}(\tfrac{\partial^2}{\partial x^2}+k_0^2)B(r_n,\theta_n)\} + \\
& \Delta X_n \Delta Y_n k_z^2 H_0^{(1)}(k_\rho r_n)\sin^2\theta_n \\
(\tfrac{\partial^2}{\partial y^2}+k_0^2)I_n(x,y) = \; & \Delta X_n \Delta Y_n k_\rho^2 \{H_0^{(1)}(k_\rho r_n)\cos^2\theta_n + \tfrac{H_1^{(1)}(k_\rho r_n)}{k_\rho r_n}(\sin^2\theta_n - \cos^2\theta_n) \\
& \tfrac{\Delta X^2}{24}(\tfrac{\partial^2}{\partial y^2}+k_0^2)A(r_n,\theta_n) + \tfrac{\Delta Y^2}{24}(\tfrac{\partial^2}{\partial y^2}+k_0^2)B(r_n,\theta_n)\} + \\
& \Delta X_n \Delta Y_n k_z^2 H_0^{(1)}(k_\rho r_n)\cos^2\theta_n
\end{aligned}
\tag{A.22}
$$

where

$$
\begin{aligned}
\tfrac{\partial^2}{\partial x^2}A(r_n,\theta_n) = \; & k_\rho^2\{H_0^{(1)}(k_\rho r_n)[\cos^2\theta_n(\tfrac{3}{8}\cos^2\theta_n + \tfrac{1}{8}\sin^2\theta_n) \\
& + \tfrac{2\sin^2\theta_n}{(k_\rho r_n)^2}(3\cos^2\theta_n - \sin^2\theta_n)] \\
& + H_1^{(1)}(k_\rho r_n^q)[\tfrac{5}{k_\rho r_n^q}\cos^2\theta_n^q \sin^2\theta_n^q - \tfrac{4\sin^2\theta_n}{(k_\rho r_n)^3}(3\cos^2\theta_n - \sin^2\theta_n)] \\
& + H_2^{(1)}(k_\rho r_n)[-\tfrac{1}{2}\cos^4\theta_n + \tfrac{\sin^2\theta_n}{(k_\rho r_n)^2}(-9\cos^2\theta_n + \sin^2\theta_n)] \\
& + H_4^{(1)}(k_\rho r_n)[\tfrac{1}{8}\cos^2\theta_n(\cos^2\theta_n - \sin^2\theta_n)]\},
\end{aligned}
\tag{A.23}
$$

$$
\begin{aligned}
\tfrac{\partial^2}{\partial z^2}A(r_n,\theta_n) = \; & k_\rho^2\{H_0^{(1)}(k_\rho r_n)[\sin^2\theta_n(\tfrac{3}{8}\cos^2\theta_n + \tfrac{1}{8}\sin^2\theta_n) \\
& + \tfrac{2\cos^2\theta_n}{(k_\rho r_n)^2}(\cos^2\theta_n - 3\sin^2\theta_n)] \\
& + H_1^{(1)}(k_\rho r_n)[-\tfrac{4}{k_\rho r_n}\cos^2\theta_n \sin^2\theta_n + \tfrac{4\cos^2\theta_n}{(k_\rho r_n)^3}(3\sin^2\theta_n - \cos^2\theta_n)] \\
& + H_2^{(1)}(k_\rho r_n)[-\tfrac{1}{2}\sin^2\theta_n\cos^2\theta_n + \tfrac{\cos^2\theta_n}{(k_\rho r_n)^2}(9\sin^2\theta_n - \cos^2\theta_n)] \\
& + H_4^{(1)}(k_\rho r_n)[\tfrac{1}{8}\sin^2\theta_n(\cos^2\theta_n - \sin^2\theta_n)]\},
\end{aligned}
\tag{A.24}
$$

$$
\begin{aligned}
\tfrac{\partial^2}{\partial x^2}B(r_n,\theta_n) = \; & k_\rho^2\{H_0^{(1)}(k_\rho r_n)[\cos^2\theta_n(\tfrac{3}{8}\sin^2\theta_n + \tfrac{1}{8}\cos^2\theta_n) \\
& + \tfrac{2\sin^2\theta_n}{(k_\rho r_n)^2}(\sin^2\theta_n - 3\cos^2\theta_n)] \\
& + H_1^{(1)}(k_\rho r_n)[\tfrac{\sin^2\theta_n}{k_\rho r_n}(-4\cos^2\theta_n + \sin^2\theta_n) + \tfrac{4\sin^2\theta_n}{(k_\rho r_n)^3}(3\cos^2\theta_n - \sin^2\theta_n)] \\
& + H_2^{(1)}(k_\rho r_n)[-\tfrac{1}{2}\sin^2\theta_n\cos^2\theta_n + \tfrac{\sin^2\theta_n}{(k_\rho r_n)^2}(9\cos^2\theta_n - \sin^2\theta_n)] \\
& + H_4^{(1)}(k_\rho r_n)[\tfrac{1}{8}\cos^2\theta_n(\sin^2\theta_n - \cos^2\theta_n)]\},
\end{aligned}
\tag{A.25}
$$

$$\begin{aligned}\frac{\partial^2}{\partial z^2}B(r_n,\theta_n) = & \; k_\rho^2\{H_0^{(1)}(k_\rho r_n)[\sin^2\theta_n(\tfrac{3}{8}\sin^2\theta_n + \tfrac{1}{8}\cos^2\theta_n) \\ & + \tfrac{2\cos^2\theta_n}{(k_\rho r_n)^2}(3\sin^2\theta_n - \cos^2\theta_n)] \\ & + H_1^{(1)}(k_\rho r_n)[\tfrac{5}{k_\rho r_n}\sin^2\theta_n\cos^2\theta_n - \tfrac{4\cos^2\theta_n}{(k_\rho r_n)^3}(3\sin^2\theta_n - \cos^2\theta_n)] \\ & + H_2^{(1)}(k_\rho r_n)[-\tfrac{1}{2}\sin^4\theta_n - \tfrac{\cos^2\theta_n}{(k_\rho r_n)^2}(9\sin^2\theta_n - \cos^2\theta_n)] \\ & + H_4^{(1)}(k_\rho r_n)[\tfrac{1}{8}\sin^2\theta_n(\sin^2\theta_n - \cos^2\theta_n)]\}.\end{aligned} \tag{A.26}$$

We also note that an exact analytical expression for $\frac{\partial^2}{\partial x \partial y}I_n(x,y)$ can be obtained without using the Taylor expansion and is given by

$$\begin{aligned}\frac{\partial^2}{\partial x \partial y}I_n(x,y) = & \; H_0^{(1)}(k_\rho\sqrt{(x - x_n - \tfrac{\Delta X_n}{2})^2 + (y - y_n - \tfrac{\Delta Y_n}{2})^2}) - \\ & H_0^{(1)}(k_\rho\sqrt{(x - x_n - \tfrac{\Delta X_n}{2})^2 + (y - y_n + \tfrac{\Delta Y_n}{2})^2}) - \\ & H_0^{(1)}(k_\rho\sqrt{(x - x_n + \tfrac{\Delta X_n}{2})^2 + (y - y_n - \tfrac{\Delta Y_n}{2})^2}) + \\ & H_0^{(1)}(k_\rho\sqrt{(x - x_n + \tfrac{\Delta X_n}{2})^2 + (y - y_n + \tfrac{\Delta Y_n}{2})^2})\end{aligned} \tag{A.27}$$

When the observation point is in the center of the cell itself, the Taylor series expansion cannot be used. In this case we can employ the small argument expansion of the Hankel function, i.e.

$$H_0^{(1)}(x) \approx 1 + \frac{2i}{\pi}(\ln\frac{x}{2} + \gamma) \tag{A.28}$$

Then at the center of the cell (self-cell contribution), we have

$$\begin{aligned}I_n(x_n,y_n) = & \; \tfrac{i4}{\pi}\{\tfrac{k_\rho^2\Delta X_n\Delta Y_n}{2}[\gamma - \tfrac{i\pi+3}{2} + \ln(\tfrac{k_\rho\sqrt{(\Delta X_n)^2+(\Delta Y_n)^2}}{2})] \\ & + (\tfrac{k_\rho\Delta X_n}{2})^2\arctan(\tfrac{\Delta Y_n}{\Delta X_n}) + (\tfrac{k_\rho\Delta Y_n}{2})^2(\tfrac{\pi}{2} - \arctan(\tfrac{\Delta Y_n}{\Delta X_n}))\}.\end{aligned} \tag{A.29}$$

Using the same expansion we can also get

$$\begin{aligned}(\tfrac{\partial^2}{\partial x^2} + k_0^2)I_n(x_n,y_n) = & \; \tfrac{i4k_0^2}{\pi}\{\tfrac{k_\rho^2\Delta X_n\Delta Y_n}{4}[\gamma - \tfrac{i\pi+3}{2} + \ln(\tfrac{k_\rho\sqrt{(\Delta X_n)^2+(\Delta Y_n)^2}}{2})] \\ & + (\tfrac{k_\rho\Delta X_n}{2})^2\arctan(\tfrac{\Delta Y_n}{\Delta X_n}) + (\tfrac{k_\rho\Delta Y_n}{2})^2(\tfrac{\pi}{2} - \arctan(\tfrac{\Delta Y_n}{\Delta X_n}))\},\end{aligned} \tag{A.30}$$

$$\begin{aligned}(\tfrac{\partial^2}{\partial y^2} + k_0^2)I_n(x_n,y_n) = & \; \tfrac{i4k_0^2}{\pi}\{\tfrac{k_\rho^2\Delta X_n\Delta Y_n}{4}[\gamma - \tfrac{i\pi+3}{2} + \ln(\tfrac{k_\rho\sqrt{(\Delta X_n)^2+(\Delta Y_n)^2}}{2})] \\ & + (\tfrac{k_\rho\Delta Y_n}{2})^2(\tfrac{\pi}{2} - \arctan(\tfrac{\Delta Y_n}{\Delta X_n})) + (\tfrac{k_\rho\Delta X_n}{2})^2\arctan(\tfrac{\Delta Y_n}{\Delta X_n})\}.\end{aligned} \tag{A.31}$$

The evaluation of the second order derivatives of $I_n(x,y)$ (expressions in (A.22)) gives accurate results when $r_n \geq \lambda/60$. For smaller values of r_n the small argument expression for the Hankel function can be used. In such cases we have

$$\begin{aligned}(\frac{\partial^2}{\partial x^2}+k_0^2)I_n(x,y) = &\ F_1(x,y)-F_2(x,y)\\ (\frac{\partial^2}{\partial y^2}+k_0^2)I_n(x,y) = &\ G_1(x,y)-G_2(x,y)\end{aligned} \tag{A.32}$$

where

$$\begin{aligned}F_j(x,y) = &\ k_0^2\frac{\Delta X_n}{2}b_{nj}(\frac{3i}{\pi}-\frac{2i\gamma}{\pi}-1)+(\tan\frac{a_{n2}}{b_{nj}}-\tan\frac{a_{n1}}{b_{nj}})(\frac{2i}{\pi}-\frac{ik_\rho^2 b_{nj}}{\pi})\\ &-\frac{ib_{nj}k_\rho^2}{\pi}[a_{n2}\ln\frac{\sqrt{a_{n2}^2+b_{nj}^2}}{2}-a_{n1}\ln\frac{\sqrt{a_{n1}^2+b_{nj}^2}}{2}]\\ G_j(x,y) = &\ k_0^2\frac{\Delta Y_n}{2}a_{nj}(\frac{3i}{\pi}-\frac{2i\gamma}{\pi}-1)+(\tan\frac{b_{n2}}{a_{nj}}-\tan\frac{b_{n1}}{a_{nj}})(\frac{2i}{\pi}-\frac{ik_\rho^2 a_{nj}}{\pi})\\ &-\frac{ia_{nj}k_\rho^2}{\pi}[b_{n2}\ln\frac{\sqrt{b_{n2}^2+a_{nj}^2}}{2}-b_{n1}\ln\frac{\sqrt{b_{n1}^2+a_{nj}^2}}{2}]\end{aligned} \tag{A.33}$$

with

$$a_{nj} = \begin{cases} x-x_n-\frac{\Delta X_n}{2} & i=1\\ x-x_n+\frac{\Delta X_n}{2} & i=2\end{cases} \tag{A.34}$$

$$b_{nj} = \begin{cases} y-y_n-\frac{\Delta Y_n}{2} & i=1\\ y-y_n+\frac{\Delta Y_n}{2} & i=2\end{cases} \tag{A.35}$$

Now we are in a position to express the impedance matrix elements in terms of $I_n(x,y)$. The off-diagonal entries of the impedance matrix for the TM case are given by

$$Z_{mn}^{TM} = \frac{ik_0^2}{4}[\epsilon_r(x_m,y_m)-1]I_n(x_m,y_m) \tag{A.36}$$

and the diagonal entries are

$$Z_{nn}^{TM} = \frac{ik_0^2}{4}[\epsilon_r(x_n,y_n)-1]I_n(x_n,y_n)-1 \tag{A.37}$$

For TE polarization, where the impedance matrix is composed of four sub-impedance matrices, the off-diagonal elements of each matrix are

$$\begin{aligned}Z_{1mn}^{TE} = &\ \frac{i}{4}[\epsilon_r(x_m,y_m)-1][(\frac{\partial^2}{\partial x^2}+k_0^2)I_n(x_m,y_m)]\\ Z_{2mn}^{TE} = &\ \frac{i}{4}[\epsilon_r(x_m,y_m)-1][\frac{\partial^2}{\partial x\partial y}I_n(x_m,y_m)]\\ Z_{3mn}^{TE} = &\ Z_{2mn}^{TE}\\ Z_{4mn}^{TE} = &\ \frac{i}{4}[\epsilon_r(x_m,y_m)-1][(\frac{\partial^2}{\partial y^2}+k_0^2)I_n(x_m,y_m)]\end{aligned} \tag{A.38}$$

and the diagonal elements are given by

$$\begin{aligned}Z_{1nn}^{TE} = &\ \frac{i}{4}[\epsilon_r(x_n,y_n)-1][(\frac{\partial^2}{\partial x^2}+k_0^2)I_n(x_n,y_n)]-1\\ Z_{2nn}^{TE} = &\ 0\\ Z_{3nn}^{TE} = &\ 0\\ Z_{4nn}^{TE} = &\ \frac{i}{4}[\epsilon_r(x_n,y_n)-1][(\frac{\partial^2}{\partial y^2}+k_0^2)I_n(x_n,y_n)]-1\end{aligned} \tag{A.39}$$

Finally the terms, Z^x, and Z^y are given by

$$\begin{aligned} Z^x_{mn} &= \tfrac{k_z}{4}[\epsilon_r(x_m, y_m) - 1]\tfrac{\partial}{\partial x} I_n(x_m, y_m) \\ Z^y_{mn} &= \tfrac{k_z}{4}[\epsilon_r(x_m, y_m) - 1]\tfrac{\partial}{\partial y} I_n(x_m, y_m) \end{aligned} \tag{A.40}$$

for off-diagonal elements, and

$$Z^x_{nn} = Z^y_{nn} = 0 \qquad m \neq n \tag{A.41}$$

for diagonal elements. The excitation vector elements for the TM and TE incidence cases, respectively, are given by

$$\mathcal{E}_x = ik_0 Y_0[\epsilon_r(x_m, y_m) - 1]E^i_x(x_m, y_m), \qquad \text{TE} \tag{A.42}$$

$$\mathcal{E}_y = ik_0 Y_0[\epsilon_r(x_m, y_m) - 1]E^i_y(x_m, y_m), \qquad \text{TE} \tag{A.43}$$

$$\mathcal{E}_z = ik_0 Y_0[\epsilon_r(x_m, y_m) - 1]E^i_z(x_m, y_m) \qquad \text{TM} \tag{A.44}$$

Appendix B: Near Field Calculation

In this section an efficient formulation for the calculation of scattered fields from dielectric cylinders is provided. With the help of the Hertz potential, electric field can be easily computed from a known current distribution, $\vec{\mathbf{J}}$, by using $\vec{\mathbf{E}} = \nabla(\nabla \cdot \vec{\Pi}_e) + k_1^2 \vec{\Pi}_e$. The electrical Hertz vector is represented by

$$\vec{\Pi}_e = -\sum_n \frac{1}{4\pi j w \varepsilon} \int_{v'} \vec{J}_n e^{-jk'_z z'} \frac{e^{jk|\vec{r}-\vec{r}'|}}{|\vec{r}-\vec{r}'|} dv' \tag{B.1}$$

where $\vec{J}_n$ is a constant vector. Let $I = \int_{s'_n} \frac{e^{jk|\vec{r}-\vec{r}'|}}{|\vec{r}-\vec{r}'|} e^{-jk'_z z'} dx'dy' = \int_{s'_n} f(|\vec{r}-\vec{r}'|) e^{-jk'_z z'} dx'dy'$. where s'_n is a cell area. For small pixel area and using mid-point integration

$$\int_{s'_n} \approx \triangle^2 \frac{e^{jkr_n}}{r_n}$$

where $r_n = \sqrt{(x-x_n)^2 + (y-y_n)^2 + (z-z')^2}$. Based on the above result, other integrals can be evaluated as

$$
\begin{aligned}
I_{xx} &= \frac{\partial^2}{\partial x^2} I \approx \triangle^2 \frac{e^{jkr_n}}{r_n}(jkr_n - 1)\left[\frac{1}{r_n} - \frac{3\cos^2\theta_n}{r_n} + jk\cos^2\theta_n\right] \\
I_{yy} &= \frac{\partial^2}{\partial y^2} I \approx \triangle^2 \frac{e^{jkr_n}}{r_n}(jkr_n - 1)\left[\frac{1}{r_n} - \frac{3\sin^2\theta_n}{r_n} + jk\sin^2\theta_n\right] \\
I_{zz} &= \frac{\partial^2}{\partial z^2} I \approx \triangle^2 \frac{e^{jkr_n}}{r_n}(jkr_n - 1)\left[\frac{1}{r_n} - \frac{3\cos^2\theta_n}{r_n} + jk\cos^2\theta_n\right] \\
I_u &= \frac{\partial}{\partial u} I \approx \triangle^2 \frac{jkr_n - 1}{r_n^2} e^{jkr_n} \begin{cases} \cos\theta_n & u = x \\ \sin\theta_n & u = y \end{cases}
\end{aligned}
$$

The rest terms, I_{xy}, I_{xz} and I_{yz}, can be evaluated analytically as

$$
\begin{aligned}
I_{xy} &= \triangle^2 \left[\frac{e^{jkr_n^{--}}}{r_n^{--}} - \frac{e^{jkr_n^{-+}}}{r_n^{-+}} - \frac{e^{jkr_n^{+-}}}{r_n^{+-}} + \frac{e^{jkr_n^{++}}}{r_n^{++}}\right] \\
\int_0^L I_{xz} dz' &= \frac{\partial^2}{\partial x \partial z} \int_0^L I dz' = \triangle^2 \frac{e^{jkr_{n0}}}{r_{n0}}(jkr_{n0} - 1)\cos\theta_{n0} - \triangle^2 \frac{e^{jkr_{nl}}}{r_{n1l}}(jkr_{nl} - 1)\cos\theta_{nl} e^{-jk_z' L} \\
\int_0^L I_{yz} dz' &= \frac{\partial^2}{\partial y \partial z} \int_0^L I dz' = \triangle^2 \frac{e^{jkr_{n0}}}{r_{n0}}(jkr_{n0} - 1)\sin\theta_{n0} - \triangle^2 \frac{e^{jkr_{nl}}}{r_{nl}}(jkr_{nl} - 1)\sin\theta_{nl} e^{-jk_z' L}
\end{aligned}
$$

where

$$
\begin{aligned}
r_n^{\pm\pm} &= \sqrt{(x - x_n \pm \triangle/2)^2 + (y - y_n \pm \triangle/2)^2 + (z-z')^2} \\
r_{n0} &= \sqrt{(x-x_n)^2 + (y-y_n)^2 + z^2} \\
r_{nl} &= \sqrt{(x-x_n)^2 + (y-y_n)^2 + (z-L)^2} \\
\cos_{\theta_{n0}} &= z/r_{n0}, \quad \cos_{\theta_{nl}} = (z-L)/r_{n0}
\end{aligned}
$$

Therefore final expression for the electrical field that includes the effect of ground is given by

$$
\begin{aligned}
\vec{\mathbf{E}} \approx -\frac{1}{4\pi jw\varepsilon} \sum_n \int_0^L [& \left\{(R_h+1)(I_{xx} + k_1^2 I)l_x + (R_h+1)I_{xy}l_y + (R_v+1)I_{xz}l_z\right\}\hat{x} + \\
& \left\{(R_h+1)I_{xy}l_x + (R_h+1)(I_{yy} + k_1^2 I)l_y + (R_v+1)I_{yz}l_z\right\}\hat{y} + \\
& \left\{(R_h+1)I_{xz}l_x + (R_h+1)I_{yz}l_y + (R_v+1)(I_{zz} + k_1^2 I)l_z\right\} + \\
& \hat{R}_h(I_{xz}\hat{x} + I_{yz}\hat{y} + I_{zz}\hat{z})(\cos\varphi l_x + \sin\varphi l_y)] \, dz'
\end{aligned}
\tag{B.2}
$$

where the summation is over all cells in all cylinders and R_h, and R_v are the ground Fresnel reflection coefficient and also $\hat{R}_h = \sin 2\theta \frac{\cos\theta - \sqrt{\kappa - \sin^2\theta}}{\kappa\cos\theta + \sqrt{\kappa - \sin^2\theta}}$ [14].

References

[1] Theodor Tamir, "On Radio-Wave Propagation in Forest Environments," *IEEE Trans. Antennas Propagat.*, vol. AP-15 pp.806-817, Nov. 1967.

[2] K. Sarabandi, and I. Koh, "Effect of Canopy-Air Interface Roughness on HF-UHF Wave Propagation in Forest," *IEEE Trans. Antennas Propagat.* to be submitted for publication.

[3] Ulaby, F.T., R.K. Moore, and A.K. Fung, *Microwave Remote Sensing Active and Passive.* Norwood: Arthech House, 1982.

[4] Sarabandi K., P. Siquera, "Numerical Scattering Analysis for two Dimensional Dense Random Media: Characterization of Effective Permittivity," *IEEE Trans. Antennas Propagat.*, vol.45, No.5 April. 1997.

[5] G.F. Carey and J.T. Oden, *Finite Elements.*, New Jersey: Prentice-Hall, 1984.

[6] F. Harrington *Time-Harmonic Electromagnetic Fields.* New York: McGRAW-HILL, 1961.

[7] Sarabandi K., "Electromagnetic Scattering from Vegetation Canopies," Ph.D Thesis, The University of Michigan, 1989

[8] K. Sarabandi, P.F. Polatin, and F.T. Ulaby, "Monte Carlo Simulation of Scattering from a Layer of Vertical Cylinder," *IEEE Trans. Antennas Propagat.*, vol.41, No.4 pp.465-475, April. 1993.

[9] W.L.Stutzman, and G.A. Thiele, *Antenna Theory and Design.* New York: John Wiley & Sons, 1981.

[10] J.G. Proakis, *Digital Communications.* McGRAW-HILL, 1989.

[11] A. Papoulis, *Probability, Random Variables, and Stochastic Process.* McGRAW-HILL, 1984.

[12] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C.*, Cambridge: Cambrige University, 1988.

[13] William C. Jakes *Microwave Mobile Communications.* New York: IEEE Press, 1994.

[14] K.A. Michalski, "On the Effective Evaluation of Integrals Arising in the Sommerfeld Half-Space Problem," *Proc. IEE*, Vol. 132H, pp.312-318, Aug. 1985.

Stored Components	Size
Imp. Matrix of individual cylinder	$M \times M$: TM, $2M \times 2M$: TE
$k_\rho(X_i - X_j)/\rho_{ij}, k_\rho(Y_i - Y_j)/\rho_{ij}$	M(M - 1) array
$\Phi_{i'j'}$	M(M - 1) array
$\cos^2\theta_{ij} - \sin^2\theta_{ij}$	M(M - 1)/2 array

Table 1: Stored matrices and arrays for the implementation of M-body sparse scatterers.

Antenna Configuration	Path-loss(dB)		SDV/Mean(dB)	
	200 cyl.	390cyl.	200 cyl.	390 cyl.
single dipole	47.5	48.16	1.42	1.16
4-element Broadside	48		3	
4-element End-fire	59		2.45	

Table 2: Table 2: Path-loss and standard deviation-to-mean ratio of the field of a vertical dipole and 4-element dipole arrays in the forest with $\varepsilon_{eff} = 1.03 + j0.036$, canopy height H = 20m, tree density 0.05/m^2, and tree trunk height of h = 15m.

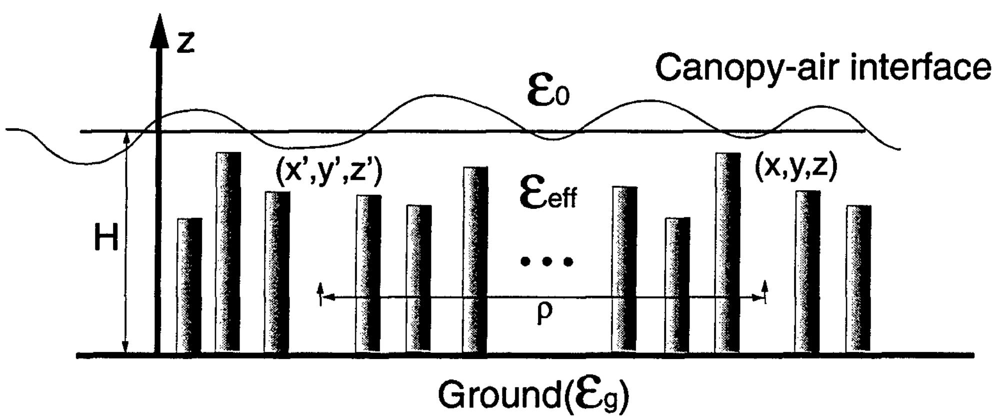


Figure 1: Geometry of a forest model for characterization of wave propagation in a forest environment.

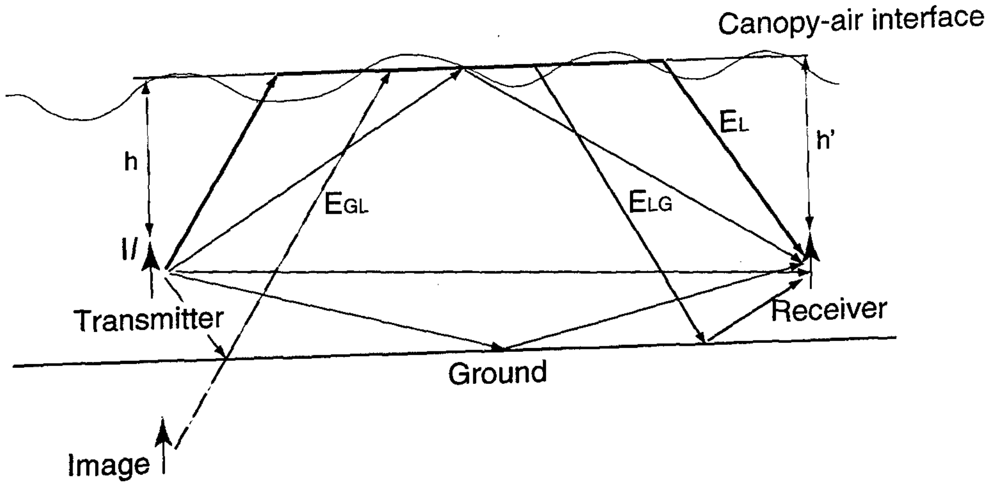


Figure 2: Wave propagation mechanisms contributing to the mean field without tree trunks in a forest environment.

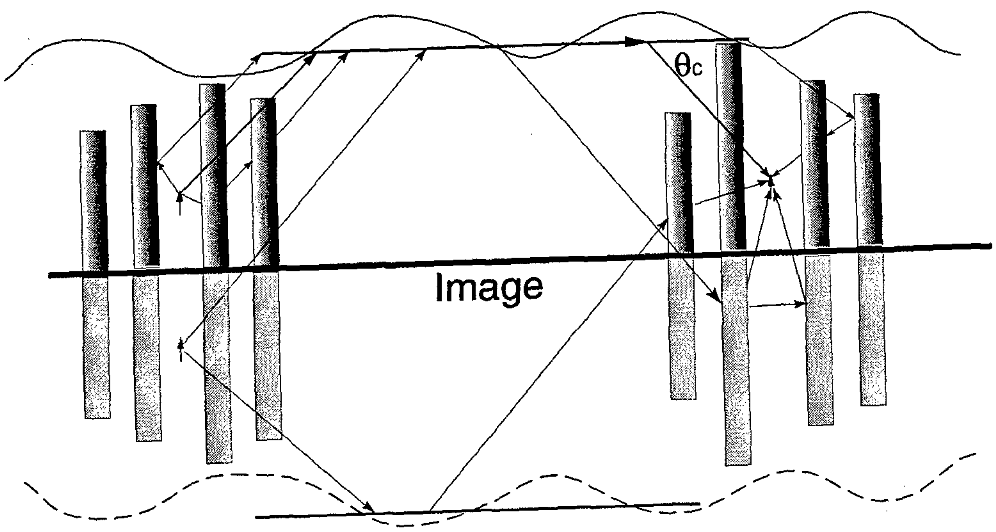
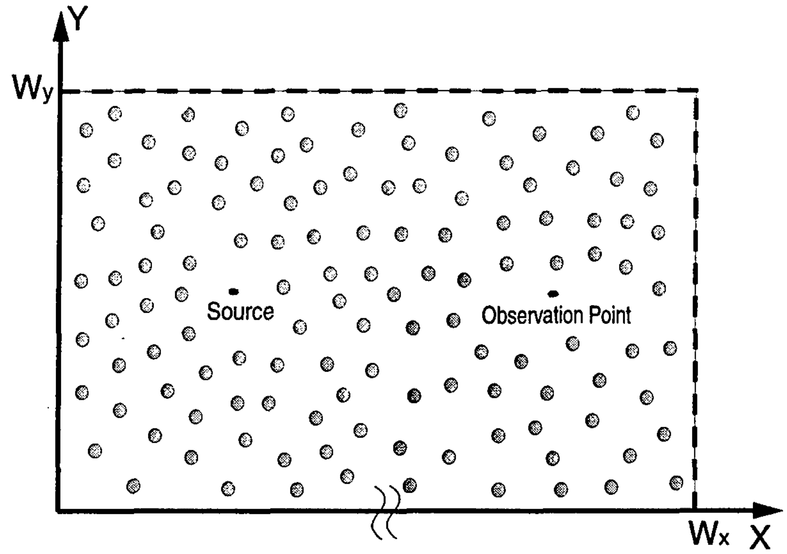
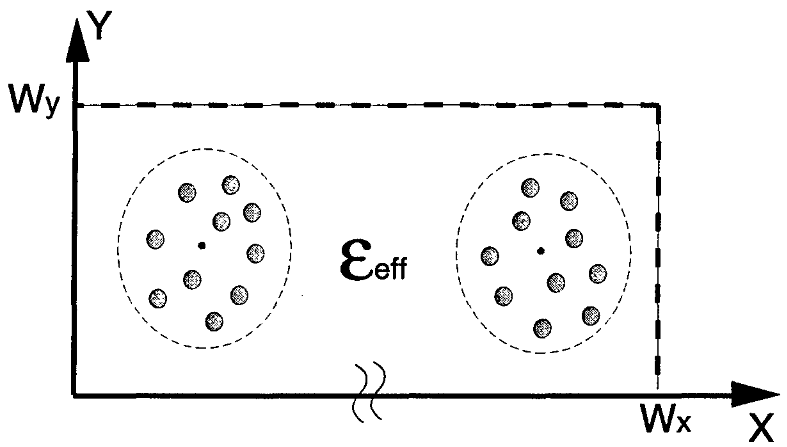


Figure 3: Scattering mechanisms resulted from wave interaction with tree trunks in a forest environment.



(a) Complete Problem



(b) Reduced Problem

Figure 4: A 2-D random medium consisting of cylinders (a) and its statistically equivalent model (b). Also shown is a fictitious boundary used for the Monte Carlo simulation.

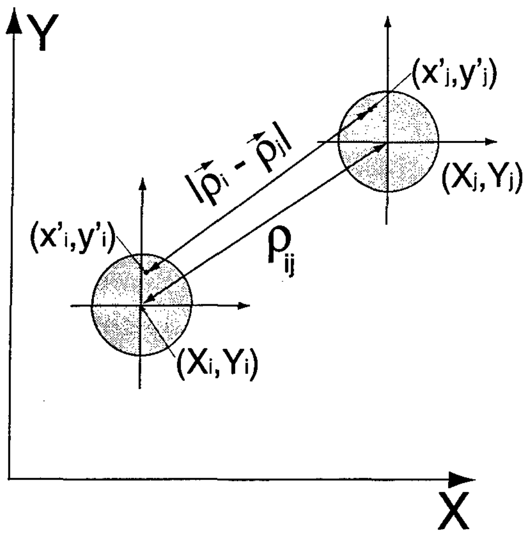


Figure 5: Global and local coordinate systems used in the MoM formulation of sparse scatterers.

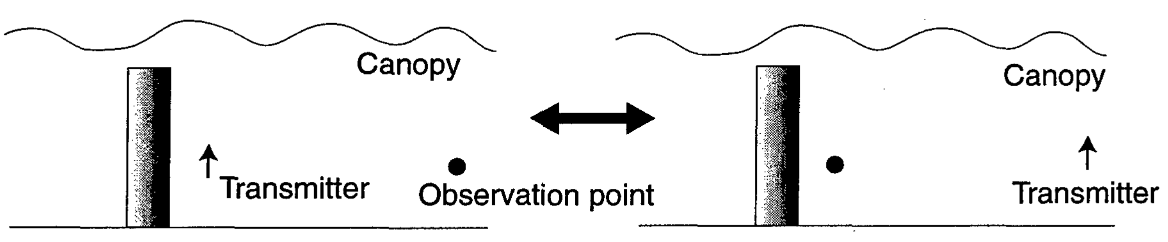
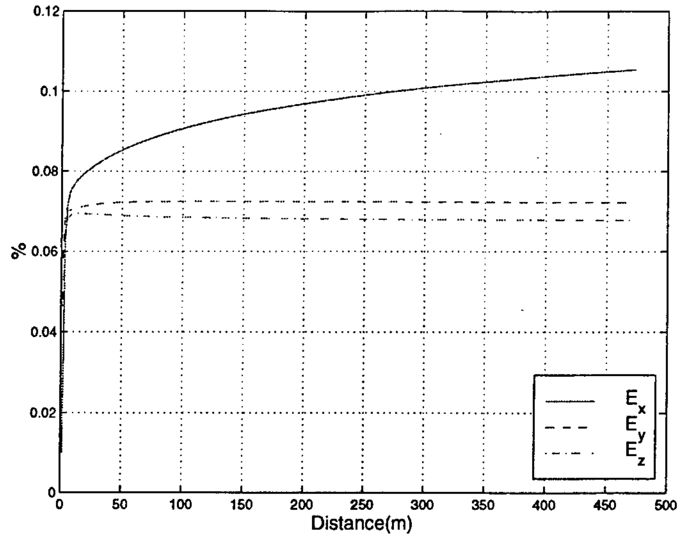
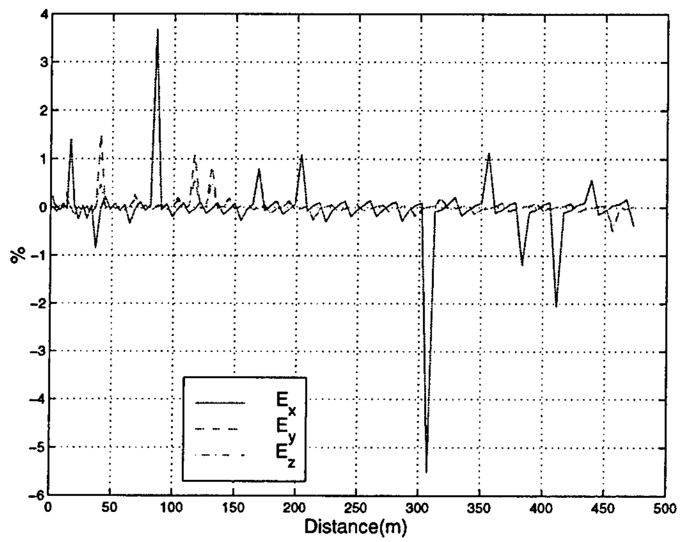


Figure 6: Application of reciprocity theorem for the computation of scattered field of cylinder from a nearby dipole embedded in a dielectric slab above ground plane.

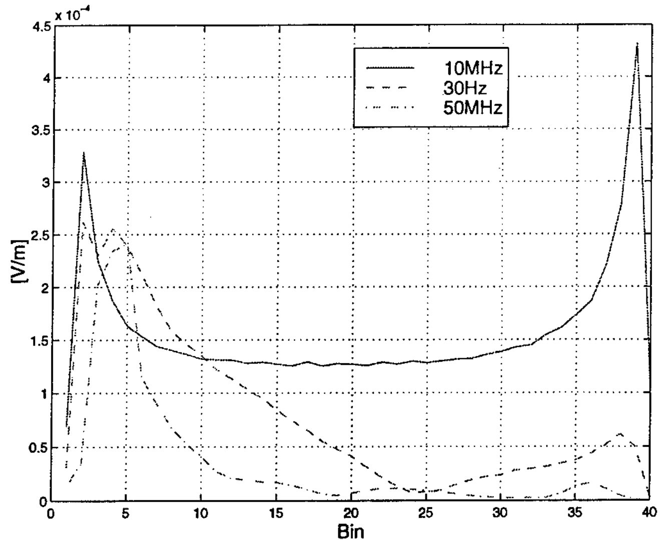


(a)

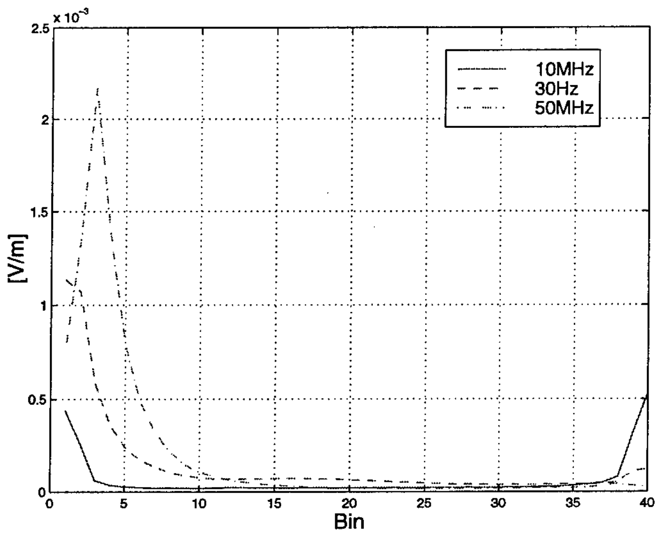


(b)

Figure 7: Relative percentage error in magnitude (a) and phase (b) of scattered electric field from 50 infinite dielectric cylinders that are located along y-axis with 2m separation when a TE plane wave is obliquely incident with $\theta = 60^o$ at 50 MHz.

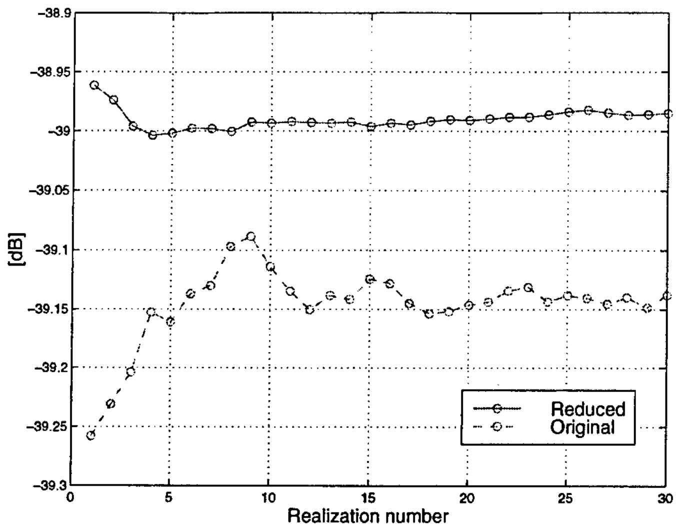


(a)

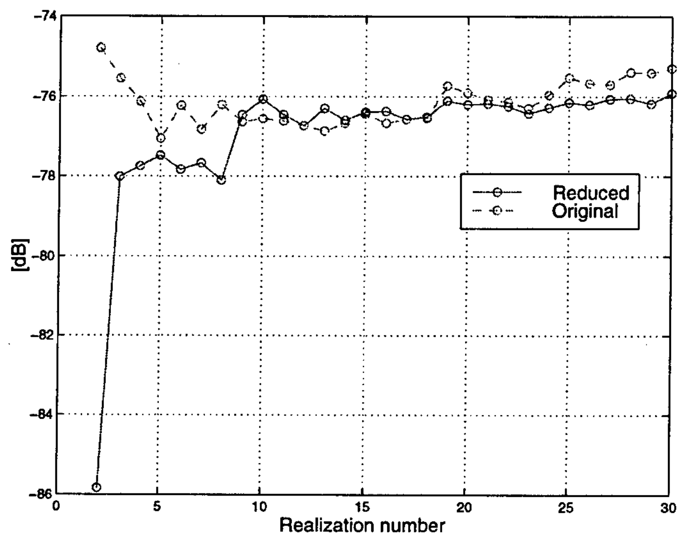


(b)

Figure 8: Mean and standard deviation of scattered electric field generated by scatterers in each 20m bin with TM source(x = 20, y = 25) and an observation point(x = 780, y = 25). Three different frequencies are used, 10, 30, and 50MHz and total number of scatterers is 2000. (a) Magnitude of mean (b) Magnitude of standard deviation.

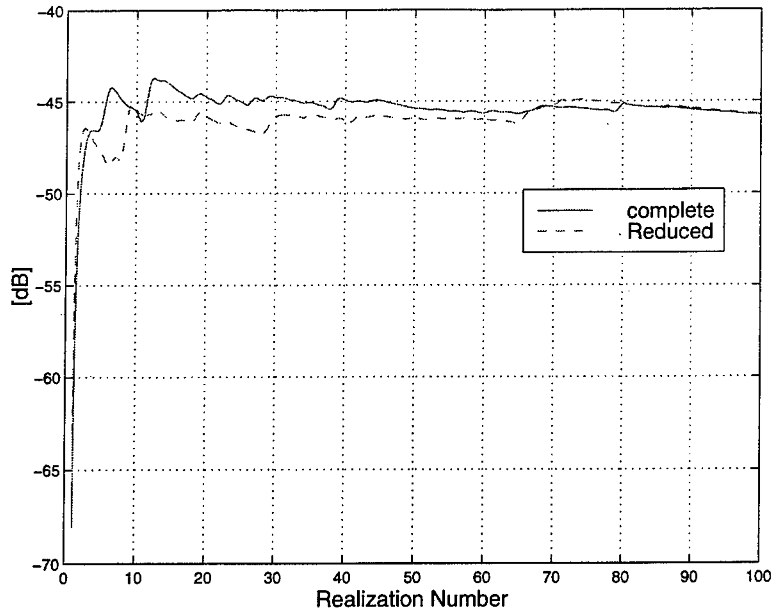


(a)



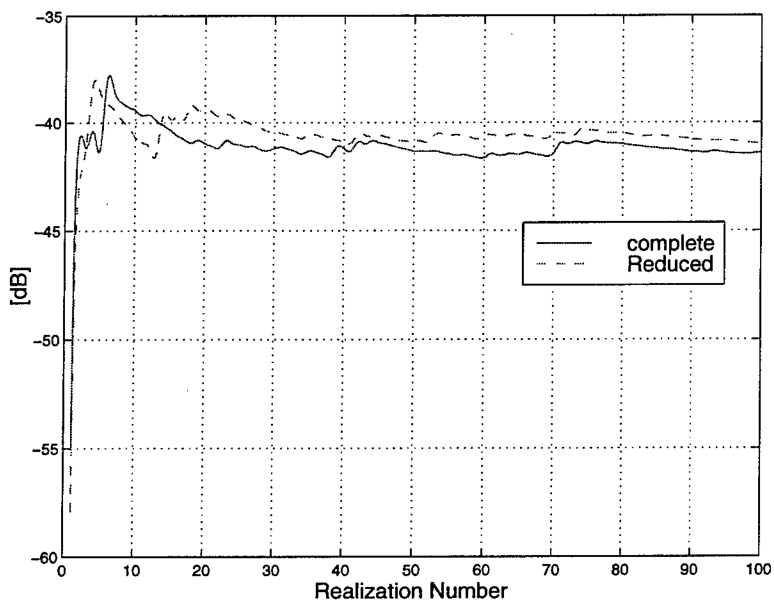
(b)

Figure 9: Comparison of the mean field (a) and standard deviation (b) of E_z of the reduced problem that keeps 600 scatterers near the source and 250 scatterers near the observation point with those of the complete problem that contains 2000 scatterers, with TM line source excitation.



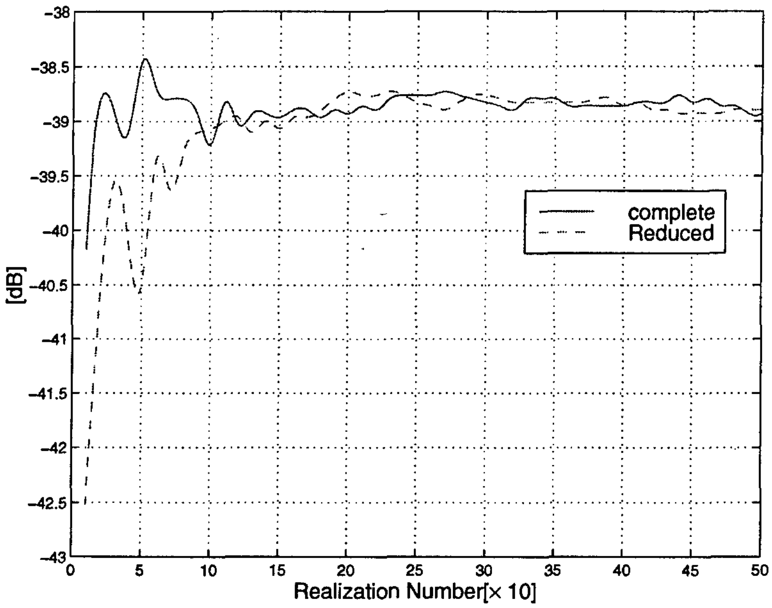
-40
-45
-50
-55
-60
-65
-70
[dB]
0
10
20
30
40
50
60
70
80
90
100
Realization Number
complete
Reduced

(a)



-35
-40
-45
-50
-55
-60
[dB]
0
10
20
30
40
50
60
70
80
90
100
Realization Number
complete
Reduced

(b)



(c)

Figure 10: Comparison of the standard deviation for TE ((a) and (b)) and TM ((c)) excitations of the complete and reduced problems that keeps 200 (TE) or 150 (TM) scatterers near the source and 150 scatterers near the observation point with those of the complete problem that contains 500 scatterers, with TE line source excitation.

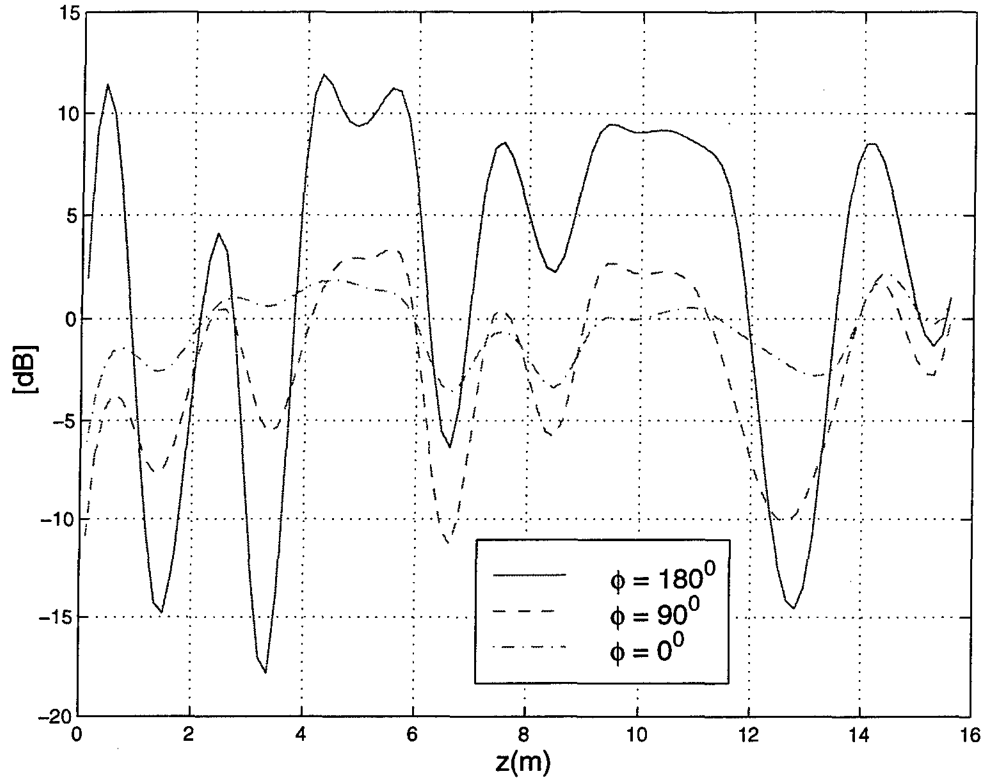
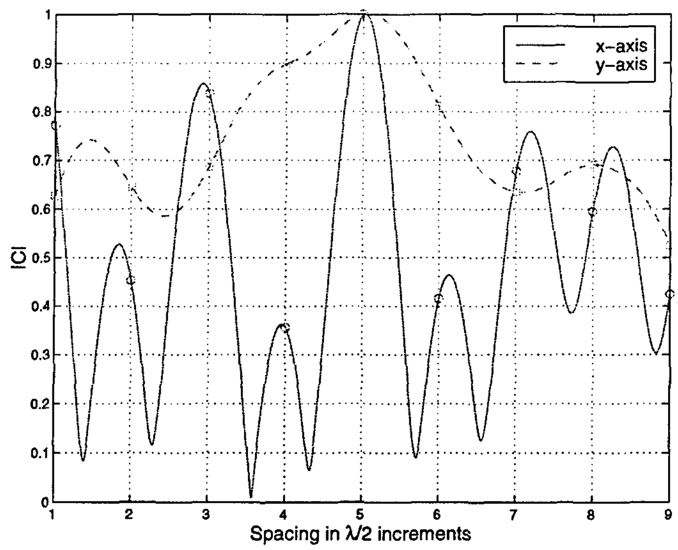
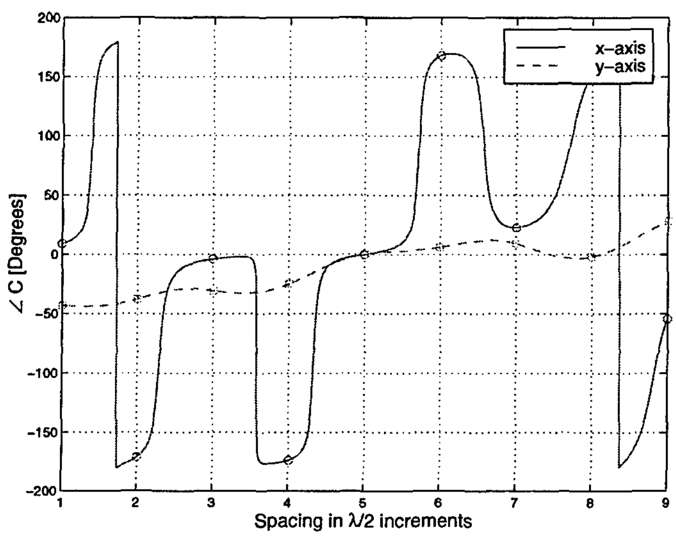


Figure 11: The ratio of the fields of a dipole with and without a tree trunk as a function of dipole height (z) and azimuthal angle around the cylinder. The dipole is vertical and 10cm away from the cylinder surface and observation point is 1km away.



(a)



(b)

Figure 12: Spatial correlation of magnitude (a) and phase (b) of field of a vertical dipole (E_z) along x-axis and y-axis.

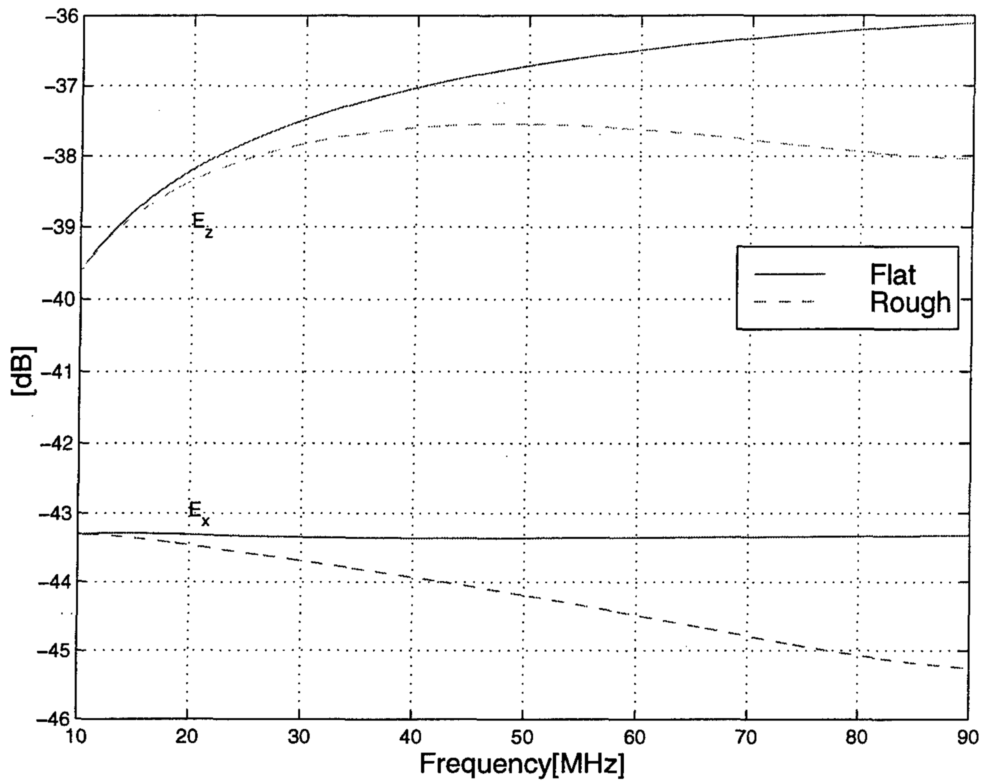


Figure 13: Magnitude of mean field as function of frequency in the absence of tree trunks and the ground plane. RMS height is 2m and $\varepsilon_{eff} = 1.03 + j0.0036$ and the transmitter and receiver are 17m and 15m below the canopy-air interface, respectively.

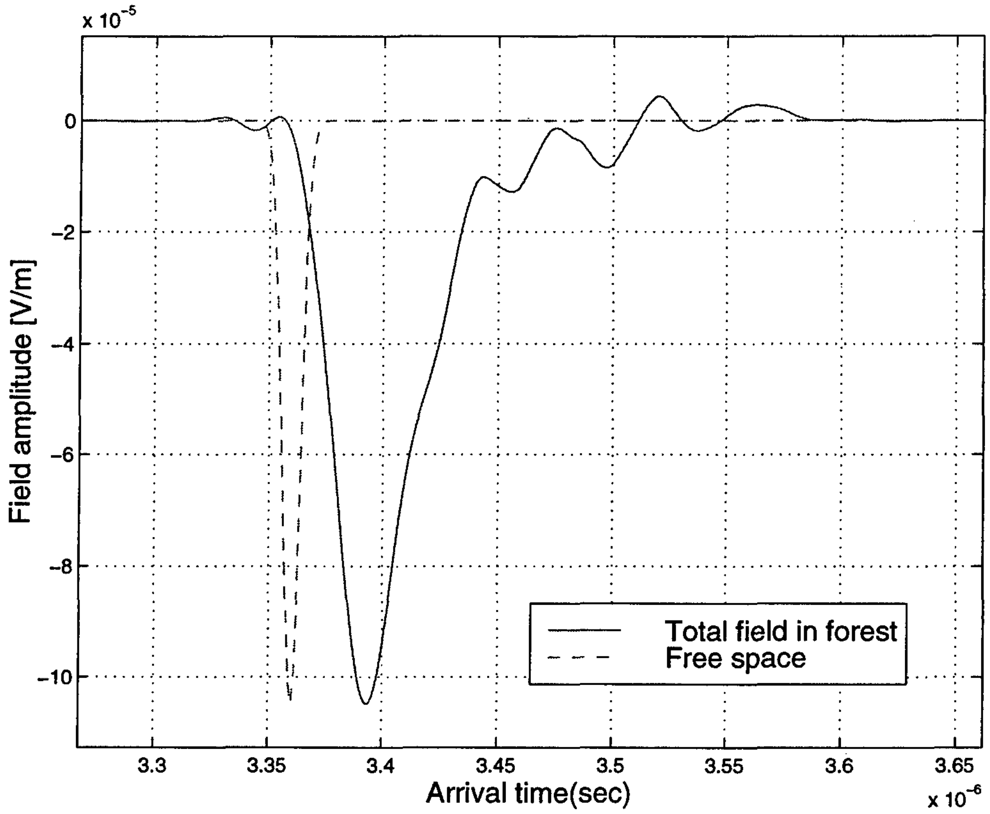


Figure 14: Impulse response with Gaussian pulse excitation on the $\hat{z}$ directed dipole. 40 non-uniformly spaced sampling points are used. Dot line is free-space response multiplied by the path-loss evaluated at 50MHz (see Table 2).

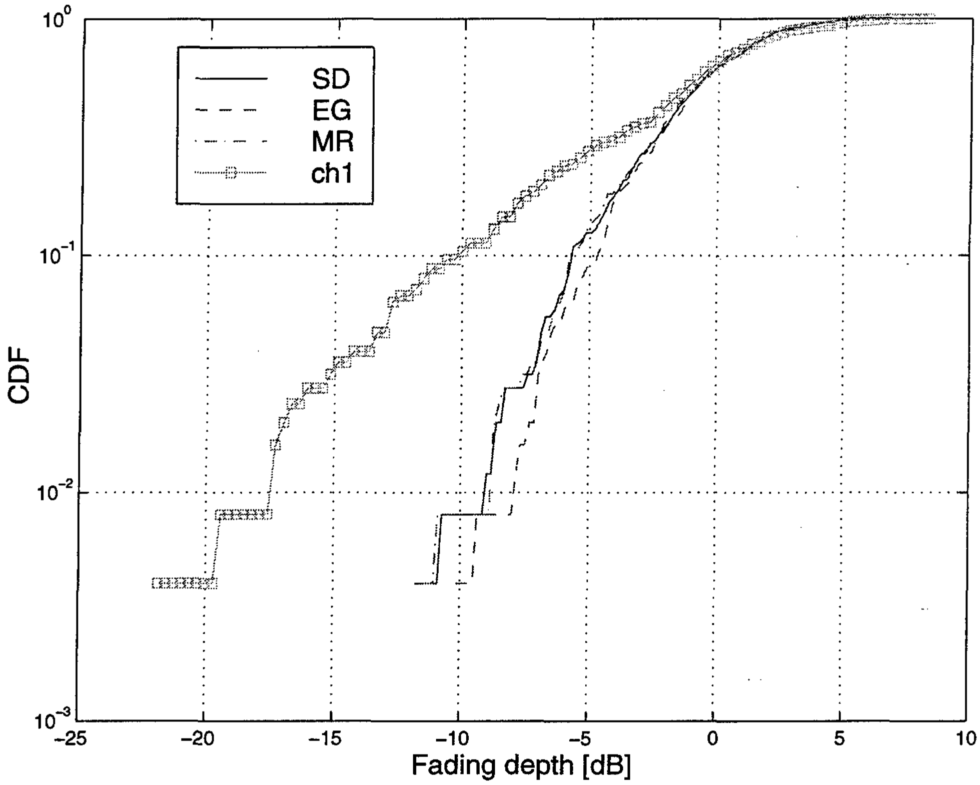


Figure 15: Cumulative distribution function(CDF) of fading depth for each diversity scheme when antenna arrays are long x-axis.

Appendix c: Software Description

The document provides a short summary of different programs that are needed for simulation of wave propagation in a forest environment. There are 32different programs and input files. That different simulations can easily be carried out by combining appropriate modules. Of 32 files 17 are header files that contain different related mathematical routines and are called automatically from the main programs. Table 1 includes the names and functions of these header files. All header files are specified by a suffix ".h".

1 - Description of Program

Table 1: Header Files

File Name	Description
CG_solver.h	Contains routines for CG type matrix solver
build_and_solve_system.h	Contains routines for building and solving matrix equation for approximate MoM.
calculus.h	Contains routines for Gauss quadrature integration
complex.h	Contains routines for complex arithmetic
constant_em.h	Contains physical and mathematical constants
em_tool.h	Contains routines for calculation of Fresnel reflection coefficient
file_handler.h	Contains a routine for opening files
incident_field.h	Contains a routine for the calculation of lateral wave without tree trunks
mesh_tool1.h	Contains a routine for generating tree locations and discretizing each cylinder
mom_tool.h	Contains a routine for the calculation of impedance matrix.
my_malloc.h	Contains a routine using "malloc" function
near_field.h	Contains a routine for the calculation of near field from finite cylinder
random_tool.h	Contains random number generator routine
read_conf.h	Contains a routine for reading input files
reflected_wave.h	Contains a routine for the direct and reflected scattered field from the ground plane using near_field.h
sparse_matrix.h	Contains a function for indexing sparse matrices
specialfun.h	Contains special functions routines

There are also 5 programs that handle pre- and post-processing. These program files are denoted by a suffix ".c" and listed in Table 2.

Table 2: Pre- and Post-processing Files

File Name	Description
mesh_s_corr.c	Generates 50 different realization of tree trunk positions for spatial correlation simulation
mesh_time.c	Generates a single realization of tree trunk position for time domain simulation
file_merger_s_corr.c	Performs the calculation of output results (mean, standard deviation, and correlation coefficient) for spatial correlation simulation
file_merger_time.c	Merges output files of time_domain.c (see Table 3) into a single file used by time_post.c
time_post.c	Calculates time domain response from the frequency domain results

There are 3 main programs that simulate wave propagation in a forest environment. These programs are specified by a suffix ".c" and listed in Table3.

Table 3: Main Programs

File Name	**Description**
array.c	Calculates field at one observation point with one dipole, or array excitation
s_corr.c	Calculates spatial correlation along x-axis or y-axis with one dipole, or array excitation
time_domain.c	Calculates spectral domain solution for time domain response with one dipole, or array excitation

There are 3 different input files that are needed to run the main program and are summarized in Table 4.

Table 4: Input Files

File Name	**Description**
array.conf	Used for array.c and s_corr.c, and time_domain.c

Table 4: Input Files

File Name	Description
array_info.conf	Contains locations, orientation, and phase difference of the array elements
time_domain.conf	Used for time_domain.c

There are 4 m-files that plot the results of simulations. Table 5 includes names and their functions.

Table 5: m-Files

File Name	Description
array_m.m	Plots the resulting electric field of "array.c" as function of realization
s_corr_m.m	Plots the results of "s_corr.c" using "sampling_m.m"
sampling_m.m	Matlab function for interpolation using sampling theorem
time_domain_m.m	Plots the results of "time_domain.c".

2 - File Format of Input Files

1) array.conf

The input values should be inserted under the variable name in the order shown below.

Table 6: array.conf

Variable name	Description	Float or Interger
frequency:	frequency [Hz]	Float
obs_x, obs_y, obs_z	x, y, and z axis of Observation point [m]	Float
radius	radius of cylinder [m]	Float
min_gap	minimum distance (min_gap*wavelength) between cylinders	Float
density	density of cylinders [$/m^2$]	Float
forest_H	height of forest [m]	Float

Table 6: array.conf

Variable name	Description	Float or Interger
cyl_h	Mean height of cylinder [m]	Float
cyl_v	Variation of height of cylinder [m]	Float
error_c	error criterion for iteration solver	Float
er_eff_r	Real part of effective dielectric constant of the canopy	Float
er_eff_i	Image part of effective dielectric constant of the canopy	Float
eg_r	Real part of dielectric constant for ground	Float
eg_i	Image part of dielectric constant for ground	Float
eg_sigma	Conductivity of dielectric constant for ground	Float
#of_antenna	Number of antennae in the array under consideration	Integer
#of_mesh	Number of discretization points along a cylinder diameter used in the method of moments.	Integer
#of_scatterer1	Number of scatterers near source	Integer
#of_scatterer2	Number of scatterers near receiver	Integer
#of_iteration	Number of iteration number	Integer
#of_points	Number of segments along z-axis of cylinders for numerical integration. ($> L/0.15\lambda$)	Integer
Maxit	number of iteration for iterative solver. (typically around 3 times the matrix size)	Integer

Note: Float type variables should be input like **1.0** or **1.**.

2)array_info.conf

Table 7: array_info.conf

Variable name	Description	Float or Interger
ant_x, ant_y, ant_z	Coordinate of the transmitter antenna [m]	Float
ori_x, ori_x, ori_x	Orientation of transmitter	Float

Table 7: array_info.conf

Variable name	Description	Float or Interger
phase_diff	Phase difference with respect to the first antenna [Degrees].	Float

Note: The coordinate, orientation and phase-difference for each antenna in the array should appear in separate successive lines.

Example - The input file for three z directed antennas in an array with a progressive phase 180 degrees.

```
ant_x  ant_y  ant_z  ori_x  ori_y  ori_z  phase_dif
 20.    25.    3.     0.     0.     1.      0.
 20.    30.    3.     0.     0.     1.     180.
 20.    35.    3.     0.     0.     1.     360.
```

3) time_domain.conf

Table 8: time_domain.conf

Variable name	Description	Float or Interger
starting_f	Starting frequency [Hz]	Float
end_f	Stop frequency [Hz]	Float
sN	Total number of samples between starting_f and end_f	Integer
dis_ro	Distance between transmitter and receiver [m]	Float

3 - Running the Programs

For "array.c", simply compile and run.

For "s_corr.c" and "time_domain.c", follow the steps shown below.

1st step: To generate tree locations use "mesh_s_corr.c" for spatial correlation calculation and "mesh_time.c" for time domain calculation. The "mesh_s_corr.c" generates 50 samples of tree arrangements for Monte Carlo simulation. "mesh_time.c" generates only one sample of tree arrangement. The "mesh_time.c" receives input argument as "mesh_time n" where n means nth realization.

2nd step: Run main program, "s_corr.c" or "time_domain.c".
"s_corr.c" computes fields at 9 different equally-spaced points along a line(specified in "mesh_s_corr.c"). Separation between two adjacent points is $\lambda/2$ where λ is wavelength.
To make the program conducive for parallel computation, s_corr is run using three parameters. To run, type "s_corr n i j" with n, i, j being integer and positive numbers. n specifies the realization number $n \in \{0, N-1\}$, i specifies the observation point $i \in \{1, 9\}$, and $j \in \{1, 2\}$ is a parameter specifying computation for scatterers near the source(j = 0) and near the observation point(j = 1). To compile, type "gcc -o s_corr s_corr.c -lm", which generates an executable file called s_corr. For Monte-Carlo simulation the following procedure may be followed:

```
for n = 0:N-1
  for i = 1:9,
    for j = 1:2,
      s_corr n i j
```

where N is the total number of realization.
Note: You can change the direction of observation point by changing "const char direction = 'x';" in the "mesh_s_corr.c" that means observation point moves along x-axis. If you change this sentence to "const char direction = 'y';" this means observation point moves along y-axis.

The program, "time_domain.c" calculates fields at sN different frequency points which is specified in "time_domian.conf" that are determined by the Gaussian quadrature procedure. To run this program, type "time_domain n" with n being interger and positive number.
$n \in \{0, sN-1\}$ is a index pointing a frequency point.

3rd step: After finishing the 2nd step, run "file_merger_s_corr.c" for spatial correlation or "file_merger_time.c and then time_post.c" for time domain analysis to obtain the final results.

For Monte-Carlo simulation the procedure shown below may be followed:

```
for n = 1:N,
  mesh_time n
  for i = 0:39,
    time_domain i
  end
  file_merger_time
  time_post
 move output files to other file name to prevent overwriting those at next simulation
end
```

4 - Output Files

1) array.c
There are 3 output files, dipole_x_p_my.dat, dipole_y_p_my.dat, and dipole_z_p_my.dat.

dipole_x_p_my.dat includes the x-component of scattered electric field
dipole_y_p_my.dat includes the y-component of scattered electric field
dipole_z_p_my.dat includes the z-component of scattered electric field

Each line of the output files include the
real part and the imaginary part of the resulting field respectively.
....

2) s_corr.c

There are 3 output files for this program as well. These are s_corr_mean.dat, s_corr_var.dat, and s_corr.dat.

s_corr_mean: Mean field at the observation points
s_corr_var.dat: Standard deviation of field at the observation points
s_corr.dat: Spatial correlation coefficient with respect to the center point

The format of data in files, s_corr_mean.dat and s_corr.dat is as follows.
Re[Ex] Im[Ex] Re[Ey] Im[Ey] Re[Ez] Im[Ez]
....

The format of data in file, s_corr_var.dat is of the following form.
Var[Ex] Var[Ey] Var[Ez]
....

Note: The program "s_corr_m.m" is an m-file which plots spatial correlation after interpolating data using the sampling theorem.

3) time_domain.c

There are 6 output files for this program.
ifx_TD_inh.dat: x-component of lateral wave
ify_TD_inh.dat: y-component of lateral wave
ifz_TD_inh.dat: z-component of lateral wave
sfx_TD_inh.dat: x-component of scattered wave
sfy_TD_inh.dat: y-component of scattered wave
sfz_TD_inh.dat: z-component of scattered wave

The format of data in these files are as follows:
real part of the resulting field   imaginary part of the resulting field
....

5) Description of main variables

Table 9: Main Variables

Variable	Description	Type
N	Total number of elements in a cylinder	Integer

Table 9: Main Variables

Variable	Description	Type
mN	Total size of matrix	Integer
N1	Number of discretization points along a cylinder diameter used in the method of moments	Integer
i_N	Number of iteration	Integer
S_N	Total number of scatterers = S_N1 + S_N2	Integer
S_N1	Number of scatterers near transmitter	Integer
S_N2	Number of scatterers near receiver	Integer
maxiter	Max. number of iteration for iterative solver	Integer
int_N	Number of segments along z-axis of cylinders for numerical integration	Integer
a_N	Number of antennae	Integer
kr	k_ρ	Float
k0	Propagation constant in free space	Float
y_0	Free space admittance	Float
z0	Free space impedance	Float
lambda	Wavelength	Float
ww	Weight factor for Gauss quadrature	Float array
xx	Abscissas point for Gauss quadrature	Float array
er_r	Real part of effective dielectric constant of the canopy	Float
er_i	Image part of effective dielectric constant of the canopy	Float
er_r	Real part of dielectric constant for ground	Float
eg_i	Image part of dielectric constant for ground	Float
sigma2	Conductivity of dielectric constant for ground	Float
f0	Frequency	Float
w0	$2\pi f0$	Float
j_error	Error criterion for iterative solver	Float
ra	Radius of cylinder	Float

Table 9: Main Variables

Variable	Description	Type
min_gap	minimum distance (min_gap*wavelength) between cylinders	Float
density	Density of cylinders	Float
f_L	Height of forest	Float
s_L	Mean height of cylinder	Float
s_v	Variance of height of cylinders	Float
x_c, y_c, z_x	Coordinate of the receiver	Float
cx, cy, cz	Center coordinate of each cylinder	Float array
array_posi	Location of each array antenna	Float array
array_ori	Orientation of each array antenna	Float array
array_pha	Phase difference of each array antenna	Float array
height_s	Height of each cylinder	Float array
center	Coordinate of each element	Float 3-D array
delta_x, delta_y	Size of each element in x-, and y- axis	Float
Z1	Impedance of effective dielectric	Complex
im_field[3]	Scattered field from scatterers near the source. 0: x-comp. 1: y-comp. 2: z-comp.	Complex array
e1	Dielectric constant of the effective media	Complex
eg	Dielectric constant of the ground	Complex
k1	Wave number of the effective media	Complex
kg	Wave number of the ground	Complex
kz	k_z	Complex
pre_const1, pre_constxy, pre_constxy1	Pre-calculated pre-factor for calculating approximate MoM matrix	Complex

Table 9: Main Variables

Variable	Description	Type
mat	Diagonal block matrix	Complex array
F	Current Vector	Complex array
J_dis	Current distribution, resulting vector	Complex array
P, PP	Needed memory for iterative solver	Complex array or 2-D array

```c
// CG type solver
// matrx must be returned through function

#ifndef zero
  #define zero Complex(0.,0.)
#endif

int error_check(n,norm_F,vec,j_error,maxiter,mN)

/* n : iteration number, norm_F : initial norm of F vector,
   j_error : tolerance, maxiter : Max. number of iteration,
   mN : size of matrix
*/

double norm_F; fcomplex *vec; float j_error;
{
  unsigned int i,flag;
  fcomplex sum;
  double norm_F1;

  flag = 0;
  sum = zero;
  for(i=0;i<mN;i++) sum = Cadd(sum,Cmul(vec[i],vec[i]));
  norm_F1 = Cabs(Csqrt(sum));

  if(norm_F == 0.) {if(norm_F1 > j_error) flag = 1;}
  else if(norm_F1/norm_F > j_error) flag = 1;

  if(norm_F == 0.) printf("Iteration number = %d error = %9.8lf\n",n,norm_F1);
  else printf("Iteration number = %d error = %9.8lf\n",n,norm_F1/norm_F);

  if(n > maxiter) {flag = 0; printf("Too many iteration...\n");}
  return flag;
}

// Born approximation type initializer

void Initial_guess(vec_,f_vec_,N__)
fcomplex *vec_,*f_vec_;
{
  int i;

  for(i=0;i<N__;i++) f_vec_[i] = vec_[i];
}

// Old version BCG reference : J.J.Min Finite element Method

void BCG_old(func,vec_,f_vec_,P_,N__,j_error_,maxiter_)
fcomplex (*func)(); fcomplex *vec_,*f_vec_,*P_; double j_error_;
{
  int i,jj,ii,flag,n=0;
  fcomplex sum,alpha,beta;
  double norm_F;

  sum = zero;
  for(i=0;i<N__;i++) sum = Cadd(sum,Cmul(vec_[i],vec_[i]));
  norm_F = Cabs(Csqrt(sum));

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),f_vec_[ii]));
      vec_[i] = Csub(vec_[i],sum);
  }

  alpha = zero;
  for(i=0;i<N__;i++)
    alpha = Cadd(alpha,Cmul(vec_[i],vec_[i]));

  for(i=0;i<N__;i++)
    P_[i] = Cdiv(vec_[i],alpha);

  do
  {
    flag = 0; n++;

    alpha = zero;
    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),P_[ii]));
      alpha = Cadd(alpha,Cmul(sum,P_[i]));
    }
    alpha = Cdiv(Complex(1.,0.),alpha);

    for(i=0;i<N__;i++)
      f_vec_[i] = Cadd(f_vec_[i],Cmul(alpha,P_[i]));

    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),P_[ii]));
      vec_[i] = Csub(vec_[i],Cmul(alpha,sum));
    }

    flag = error_check(n,norm_F,vec_,j_error_,maxiter_,N__);

    if(flag != 0)
    {
      beta = zero;
      for(i=0;i<N__;i++)
        beta = Cadd(beta,Cmul(vec_[i],vec_[i]));
      beta = Cdiv(Complex(1.,0.),beta);

      for(i=0;i<N__;i++)
        P_[i] = Cadd(P_[i],Cmul(beta,vec_[i]));
    }

  } while(flag != 0);
}

void BCG(func,vec_,f_vec_,P_,N__,j_error_,maxiter_,pre_)
fcomplex (*func)(); fcomplex *vec_,*f_vec_,*P_;
double j_error_;
{

  // Solver for Ax=y based BCG method for symmetric matrix
  // if pre_ = 0 without pre-conditioner
  // else with  Jocobi type pre-conditioner
  // func is function that returns A value.
  // vec_ is y vector. f_vec_ has final solution.
  // P_ is a vector needed for calculation.
  // j_error is tolerance. and maxiter_ is max. iteration number.

  int i,jj,ii,flag,n=0;
```

```
  fcomplex sum,sum1,alpha,beta,pre_roh,roh;
  double norm_F;

  sum = zero;
  for(i=0;i<N__;i++) sum = Cadd(sum,Cmul(vec_[i],vec_[i]));
  norm_F = Cabs(Csqrt(sum));

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),f_vec_[ii]));
    vec_[i] = Csub(vec_[i],sum);
  }

  if(pre_ != 0)
  {
    flag = 0;
    for(i=0;i<N__;i++)
      if(Cabs((*func)(i,i)) < 1.e-20) flag = 1;
  }

  if(flag != 0)
  {
    printf("There is zero diagonal term in the matrix...\n");
    printf("I ignore the pre-conditioner option...\n");
    pre_ = 0;
  }

  do
  {
    flag = 0; n++;

    if(pre_ == 1) for(i=0;i<N__;i++) vec_[i] = Cdiv(vec_[i],(*func)(i,i));

    roh = zero;
    for(i=0;i<N__;i++)
    if(pre_ == 1) roh = Cadd(roh,Cmul(vec_[i],Cmul(vec_[i],(*func)(i,i))));
    else roh = Cadd(roh,Cmul(vec_[i],vec_[i]));

    if(Cabs(roh) == 0.) { printf("BCG fails!!!\n"); exit(0);}
    if(n == 1) { for(i=0;i<N__;i++) P_[i] = vec_[i]; pre_roh = roh;}
    else
    {
      beta = Cdiv(roh,pre_roh);
      for(i=0;i<N__;i++) P_[i] = Cadd(vec_[i],Cmul(beta,P_[i]));
      pre_roh = roh;
    }
    if(pre_ == 1) for(i=0;i<N__;i++) vec_[i] = Cmul(vec_[i],(*func)(i,i));

    alpha = zero;
    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),P_[ii]));
      alpha = Cadd(alpha,Cmul(sum,P_[i]));
    }
    alpha = Cdiv(roh,alpha);

    for(i=0;i<N__;i++) f_vec_[i] = Cadd(f_vec_[i],Cmul(alpha,P_[i]));
    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),P_[ii]));
      vec_[i] = Csub(vec_[i],Cmul(sum,alpha));
    }

    flag = error_check(n,norm_F,vec_,j_error_,maxiter_,N__);

  } while(flag != 0);
}

void CGS(func,vec_,f_vec_,PP_,N__,j_error_,maxiter_,pre_)
fcomplex (*func)(); fcomplex *vec_,*f_vec_,**PP_;
double j_error_;
{

  // Solver for Ax=y based CGS method
  // if pre_ = 0 without pre-conditioner
  // else with  Jocobi type pre-conditioner
  // func is function that returns A value.
  // vec_ is y vector. f_vec_ has final solution.
  // PP_ is a memory needed for calculation.
  // j_error is tolerance. and maxiter_ is max. iteration number.

  int i,jj,ii,flag,n=0;
  fcomplex sum,sum1,sum2,temp,alpha,beta,pre_roh,roh;
  double norm_F;

  sum = zero;
  for(i=0;i<N__;i++) sum = Cadd(sum,Cmul(vec_[i],vec_[i]));
  norm_F = Cabs(Csqrt(sum));

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),f_vec_[ii]));
    vec_[i] = Csub(vec_[i],sum);
  }

  flag = 0;
  for(i=0;i<N__;i++)
  {
    if(pre_ != 0) if(Cabs((*func)(i,i)) < 1.e-20) flag = 1;
    PP_[3][i] = vec_[i];
  }

  if(flag != 0)
  {
    printf("There is zero diagonal term in the matrix...\n");
    printf("I ignore the pre-conditioner option...\n");
    pre_ = 0;
  }

  do
  {
    flag = 0; n++;

    roh = zero;
    for(i=0;i<N__;i++) roh = Cadd(roh,Cmul(vec_[i],PP_[3][i]));

    if(Cabs(roh) == 0.) { printf("CGS fails!!!\n"); exit(0);}
    if(n == 1)
    { for(i=0;i<N__;i++) PP_[1][i] = PP_[0][i] = vec_[i]; pre_roh = roh;}
    else
    {
      beta = Cdiv(roh,pre_roh);
```

```
    for(i=0;i<N__;i++)
    {
      PP_[1][i] = Cadd(vec_[i],Cmul(beta,PP_[2][i]));
   PP_[0][i] = Cadd(PP_[1][i],Cmul(beta,Cadd(PP_[2][i],Cmul(beta,PP_[0][i]))));
    }

    pre_roh = roh;
  }

  if(pre_ == 1) for(i=0;i<N__;i++) PP_[0][i] = Cdiv(PP_[0][i],(*func)(i,i));

  alpha = zero;
  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),PP_[0][ii]));
    alpha = Cadd(alpha,Cmul(sum,PP_[3][i]));
  }
  alpha = Cdiv(roh,alpha);

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),PP_[0][ii]));
    PP_[2][i] = Csub(PP_[1][i],Cmul(sum,alpha));
  }

  for(i=0;i<N__;i++)
  {
    if(pre_ == 1)
      sum = Cmul(alpha,Cdiv(Cadd(PP_[1][i],PP_[2][i]),(*func)(i,i)));
    else sum = Cmul(alpha,Cadd(PP_[1][i],PP_[2][i]));
   f_vec_[i] = Cadd(f_vec_[i],sum);
 }

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      if(pre_ == 1)
      {
        temp = Cadd(PP_[1][ii],PP_[2][ii]);
        sum = Cadd(sum,Cmul((*func)(i,ii),Cdiv(temp,(*func)(ii,ii))));
      }
      else sum = Cadd(sum,Cmul((*func)(i,ii),Cadd(PP_[1][ii],PP_[2][ii])));
    vec_[i] = Csub(vec_[i],Cmul(sum,alpha));
  }

  flag = error_check(n,norm_F,vec_,j_error_,maxiter_,N__);
  if((flag == 1) && (pre_ == 1))
    for(i=0;i<N__;i++) PP_[0][i] = Cmul(PP_[0][i],(*func)(i,i));
 } while(flag != 0);
}

void BICGSTAB(func,vec_,f_vec_,PP_,N__,j_error_,maxiter_,pre_)
fcomplex (*func)(); fcomplex *vec_,*f_vec_,**PP_;
double j_error_;
{

  // Solver for Ax=y based BCG stabilized method that needs more
  // iteration for reducing memory requirement
  // if pre_ = 0 without pre-conditioner
  // else with  Jocobi type pre-conditioner

  // func is function that returns A value.
  // vec_ is y vector. f_vec_ has final solution.
  // PP_ is a memory needed for calculation.
  // j_error is tolerance. and maxiter_ is max. iteration number.

  int i,jj,ii,flag,n=0;
  fcomplex sum,sum1,sum2,alpha,beta,w_,pre_roh,roh;
  double norm_F;

  sum = zero;
  for(i=0;i<N__;i++) sum = Cadd(sum,Cmul(vec_[i],vec_[i]));
  norm_F = Cabs(Csqrt(sum));

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),f_vec_[ii]));
    vec_[i] = Csub(vec_[i],sum);
  }

  flag = 0;
  for(i=0;i<N__;i++)
  {
    if(pre_ != 0) if(Cabs((*func)(i,i)) < 1.e-20) flag = 1;
    PP_[3][i] = vec_[i];
  }

  if(flag != 0)
  {
    printf("There is zero diagonal term in the matrix...\n");
    printf("I ignore the pre-conditioner option...\n");
    pre_ = 0;
  }

  do
  {
    flag = 0; n++;

    roh = zero;
    for(i=0;i<N__;i++) roh = Cadd(roh,Cmul(vec_[i],PP_[3][i]));

    if(Cabs(roh) == 0.) { printf("BICGSTAB fails!!!\n"); exit(0);}
    if(n == 1) {for(i=0;i<N__;i++) PP_[0][i] = vec_[i]; pre_roh = roh;}
    else
    {
      beta = Cmul(Cdiv(roh,pre_roh),Cdiv(alpha,w_));
      for(i=0;i<N__;i++)
        PP_[0][i] = Cadd(vec_[i],Cmul(beta,Csub(PP_[0][i],Cmul(w_,PP_[2][i]))));
      pre_roh = roh;
    }

    if(pre_ == 1) for(i=0;i<N__;i++) PP_[0][i] = Cdiv(PP_[0][i],(*func)(i,i));

    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),PP_[0][ii]));
      PP_[2][i] = sum;
    }

    alpha = zero;
    for(i=0;i<N__;i++)
      alpha = Cadd(alpha,Cmul(PP_[2][i],PP_[3][i]));
```

```
    alpha = Cdiv(roh,alpha);

    for(i=0;i<N__;i++)
      PP_[1][i] = Csub(vec_[i],Cmul(alpha,PP_[2][i]));

    flag = error_check(n,norm_F,PP_[1],j_error_,maxiter_,N__);
    if(flag == 0)
    {
      for(i=0;i<N__;i++)
        f_vec_[i] = Cadd(f_vec_[i],Cmul(alpha,PP_[0][i]));
    }
    else
    {
      if(pre_ == 1) for(i=0;i<N__;i++) PP_[1][i] = Cdiv(PP_[1][i],(*func)(i,i));

      sum1 = sum2 = zero;
      for(i=0;i<N__;i++)
      {
        sum = zero;
        for(ii=0;ii<N__;ii++)
          sum = Cadd(sum,Cmul((*func)(i,ii),PP_[1][ii]));
        if(pre_ == 1) sum1 = Cadd(sum1,Cmul(sum,Cmul(PP_[1][i],(*func)(i,i))));
        else sum1 = Cadd(sum1,Cmul(sum,PP_[1][i]));
        sum2 = Cadd(sum2,Cmul(sum,sum));
      }
      w_ = Cdiv(sum1,sum2);

      if(Cabs(w_) < 1.e-20) {printf("BICGSTAB fails...\n"); exit(0);}

      for(i=0;i<N__;i++)
      {
        sum = Cadd(Cmul(alpha,PP_[0][i]),Cmul(w_,PP_[1][i]));
        f_vec_[i] = Cadd(f_vec_[i],sum);
      }

      for(i=0;i<N__;i++)
      {
        sum = zero;
        for(ii=0;ii<N__;ii++)
          sum = Cadd(sum,Cmul((*func)(i,ii),PP_[1][ii]));
        if(pre_ == 1) vec_[i] = Csub(Cmul((*func)(i,i),PP_[1][i]),Cmul(w_,sum));
        else vec_[i] = Csub(PP_[1][i],Cmul(w_,sum));
      }

      flag = error_check(n,norm_F,vec_,j_error_,maxiter_,N__);
      if((flag == 1) && (pre_ == 1))
        for(i=0;i<N__;i++)
        {
          PP_[0][i] = Cmul(PP_[0][i],(*func)(i,i));
          PP_[1][i] = Cmul(PP_[1][i],(*func)(i,i));
        }
    }
  } while(flag != 0);
}

void BICGSTAB_s(func,vec_,f_vec_,PP_,N__,j_error_,maxiter_,pre_)
fcomplex (*func)(); fcomplex *vec_,*f_vec_,**PP_;
double j_error_;
{

  // Solver for Ax=y based BCG stabilized method using all needed memory
  // to speed up
  // if pre_ = 0 without pre-conditioner
  // else with  Jocobi type pre-conditioner
  // func is function that returns A value.

  // vec_ is y vector. f_vec_ has final solution.
  // PP_ is a memory needed for calculation.
  // j_error is tolerance. and maxiter_ is max. iteration number.

  int i,jj,ii,flag,n=0;
  fcomplex sum,sum1,sum2,alpha,beta,w_,pre_roh,roh;
  double norm_F;

  sum = zero;
  for(i=0;i<N__;i++) sum = Cadd(sum,Cmul(vec_[i],vec_[i]));
  norm_F = Cabs(Csqrt(sum));

  for(i=0;i<N__;i++)
  {
    sum = zero;
    for(ii=0;ii<N__;ii++)
      sum = Cadd(sum,Cmul((*func)(i,ii),f_vec_[ii]));
    vec_[i] = Csub(vec_[i],sum);
  }

  flag = 0;
  for(i=0;i<N__;i++)
  {
    if(pre_ != 0) if(Cabs((*func)(i,i)) < 1.e-20) flag = 1;
    PP_[6][i] = vec_[i];
  }

  if(flag != 0)
  {
    printf("There is zero diagonal term in the matrix...\n");
    printf("I ignore the pre-conditioner option...\n");
    pre_ = 0;
  }

  do
  {
    flag = 0; n++;

    roh = zero;
    for(i=0;i<N__;i++) roh = Cadd(roh,Cmul(vec_[i],PP_[6][i]));

    if(Cabs(roh) == 0.) { printf("BICGSTAB fails!!!\n"); exit(0);}
    if(n == 1) {for(i=0;i<N__;i++) PP_[0][i] = vec_[i]; pre_roh = roh;}
    else
    {
      beta = Cmul(Cdiv(roh,pre_roh),Cdiv(alpha,w_));
      for(i=0;i<N__;i++)
        PP_[0][i] = Cadd(vec_[i],Cmul(beta,Csub(PP_[0][i],Cmul(w_,PP_[3][i]))));

      pre_roh = roh;
    }

    for(i=0;i<N__;i++)
      if(pre_ == 1) PP_[2][i] = Cdiv(PP_[0][i],(*func)(i,i));
      else PP_[2][i] = PP_[0][i];

    for(i=0;i<N__;i++)
    {
      sum = zero;
      for(ii=0;ii<N__;ii++)
        sum = Cadd(sum,Cmul((*func)(i,ii),PP_[2][ii]));
      PP_[3][i] = sum;
    }

    alpha = zero;
```

```
    for(i=0;i<N__;i++)
      alpha = Cadd(alpha,Cmul(PP_[3][i],PP_[6][i]));
    alpha = Cdiv(roh,alpha);

    for(i=0;i<N__;i++)
      PP_[1][i] = Csub(vec_[i],Cmul(alpha,PP_[3][i]));

    flag = error_check(n,norm_F,PP_[1],j_error_,maxiter_,N__);
    if(flag == 0)
    {
      for(i=0;i<N__;i++)
        f_vec_[i] = Cadd(f_vec_[i],Cmul(alpha,PP_[2][i]));
    }
    else
    {
      for(i=0;i<N__;i++)
        if(pre_ == 1) PP_[4][i] = Cdiv(PP_[1][i],(*func)(i,i));
        else PP_[4][i] = PP_[1][i];

      for(i=0;i<N__;i++)
      {
        sum = zero;
        for(ii=0;ii<N__;ii++)
          sum = Cadd(sum,Cmul((*func)(i,ii),PP_[4][ii]));
        PP_[5][i] = sum;
      }

      sum1 = sum2 = zero;
      for(i=0;i<N__;i++)
      {
        sum1 = Cadd(sum1,Cmul(PP_[5][i],PP_[1][i]));
        sum2 = Cadd(sum2,Cmul(PP_[5][i],PP_[5][i]));
      }
      w_ = Cdiv(sum1,sum2);

      if(Cabs(w_) < 1.e-20) {printf("BICGSTAB_s fails...\n"); exit(0);}

      for(i=0;i<N__;i++)
      {
        sum = Cadd(Cmul(alpha,PP_[2][i]),Cmul(w_,PP_[4][i]));
        f_vec_[i] = Cadd(f_vec_[i],sum);
      }

      for(i=0;i<N__;i++)
        vec_[i] = Csub(PP_[1][i],Cmul(w_,PP_[5][i]));

      flag = error_check(n,norm_F,vec_,j_error_,maxiter_,N__);
    }
  } while(flag != 0);
}
```

```
// Functions for building matrix & vector
// Functions needed to solve system eq.
// This header file is for *_my.c

//Generating impedance matrix for sparse matrix scheme

void build_matrix_my(kr,kz,n1)
double kr; fcomplex kz;
{
  unsigned int i,ii,jj,a_,b,r_i,r_ii;
  unsigned int a=0,n;
  fcomplex temp,er_m,t1;
  double x,y,xn,yn,d_x,d_y,nN,dis;

  temp = Complex(0.,1./4.);
  d_x = delta_x;
  d_y = delta_y;

  for(i=0;i<3*N;i++)
  {
    x = center[i][0];
    y = center[i][1];
    er_m = Csub(er_ft(x,y),Complex(1.,0.));
    er_m = Cmul(temp,er_m);
    a_ = (int)((double)i/((double)N*3.));
    r_i = i - a_*3*N;

    for(ii=i;ii<3*N;ii++)
    {
      xn = center[ii][0];
      yn = center[ii][1];
      b = (int)((double)ii/((double)N*3.));
      r_ii = ii - b*3*N;

      if(r_i < N) {
        if(r_ii < N) {
          if(r_i == r_ii)
            mat[a] = Csub(Cmul(er_m,Inn_xx(x,y,xn,yn,d_x,d_y,kr,kz)),Cone);
          else mat[a] = Cmul(er_m,In_xx(x,y,xn,yn,d_x,d_y,kr,kz));
                     }
        else if(r_ii < 2*N) {
          if(r_i == (r_ii - N)) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
             }
        else {
          if(r_i == (r_ii - 2*N)) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
             }
      }
      else if(r_i < 2*N) {
        if(r_ii < N) {
          if((r_i - N) == r_ii) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
                     }
        else if(r_ii < 2*N) {
          if((r_i - N) == (r_ii - N))
            mat[a] = Csub(Cmul(er_m,Inn_yy(x,y,xn,yn,d_x,d_y,kr,kz)),Cone);
          else mat[a] = Cmul(er_m,In_yy(x,y,xn,yn,d_x,d_y,kr,kz));
             }
        else {
          if((r_i -N) == (r_ii - 2*N)) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
             }
      }
      else {
        if(r_ii < N) {
          if((r_i - 2*N) == r_ii) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
                     }
        else if(r_ii < 2*N) {
          if((r_i - 2*N) == (r_ii - N)) mat[a] = zero;
          else mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
             }
        else {
          if((r_i - 2*N) == (r_ii - 2*N))
            mat[a] = Csub(Cmul(er_m,Inn(x,y,xn,yn,d_x,d_y,kr)),Cone);
          else mat[a] = Cmul(er_m,In(x,y,xn,yn,d_x,d_y,kr));
             }
       }
    a++;
    }
  }

  a = 0;
  for(i=0;i<S_N;i++)
  {
    n = IA[i+1] - IA[i];
    x = cx[i+n1*S_N1];
    y = cy[i+n1*S_N1];
    er_m = Csub(er_ft(x,y),Complex(1.,0.));
    er_m = Cmul(temp,er_m);

    for(ii=0;ii<n;ii++)
    {
      xn = cx[JA[ii + IA[i]]+n1*S_N1];
      yn = cy[JA[ii + IA[i]]+n1*S_N1];

      dis = sqrt(sqr(x - xn) + sqr(y - yn));
      off_d[a] = Cmul(er_m,In(x,y,xn,yn,d_x,d_y,kr));
      off_d0[a] = Cmul(pre_const,H1(0,kr*dis));
      off_d1[a] = Cmul(pre_const,H1(2,kr*dis));
      off_d_xy[a] = Cmul(pre_const_xy,H1(2,kr*dis));
      a++;
    }
  }
}

void build_vector(n,n1)
{
  fcomplex er_m,temp,temp1;
  int jj,ii;
  double x,y;
  fcomplex Ex,Ey,Ez;

  temp = RCmul(k0*y_0,Cmul(e1,j));

  for(jj=0;jj<S_N;jj++)
    for(ii=0;ii<N;ii++)
        {
      x = center[ii + 3*jj*N + 3*N*n1*S_N1][0];
      y = center[ii + 3*jj*N + 3*N*n1*S_N1][1];
      er_m = Csub(er_ft(x,y),Complex(1.,0.));
      er_m = Cmul(temp,er_m);
      if(n1 == 0)
      {Ex = i_field_x(x,y); Ey = i_field_y(x,y); Ez = i_field_z(x,y);}
      else {
        Ex = i_field_x1(x,y); Ey = i_field_y1(x,y); Ez = i_field_z1(x,y);
          }
```

```
      F[ii+3*jj*N] = Cmul(er_m,Ex);
      F[ii+N+3*jj*N] = Cmul(er_m,Ey);
      F[ii+2*N+3*jj*N] = Cmul(er_m,Ez);
        }
}

// Indexing impedance matrix stored with sparse matrix scheme

fcomplex Z_my(i,k)
{
  int a,b,a1,b1,n,im,km;
  fcomplex temp,temp1,temp2,compensation_factor;
  float t,t1,t2;
  double c_s,si;

  a = (int)((double)i/((double)N*3.));
  b = (int)((double)k/((double)N*3.));
  im = i - a*3*N; km = k - b*3*N;

  if((a*3*N <= k) && ((a+1)*3*N > k))
  {
    // Diagonal block diagram

    a1 = (int)((double)(i - a*3*N)/((double)N));
    b1 = (int)((double)(k - b*3*N)/((double)N));
    if(k >= i) n = 3*N*im - (im-1)*im/2 + km - im;
    else n = 3*N*km - (km-1)*km/2 + im - km;

    if(k >= i) return mat[n];
    else {
    if(a1 != 2) return mat[n];
    else if(b1 == 2) return mat[n];
         else return RCmul(-1.,mat[n]);
        }
  }
  else {
    // Off-diagonal block diagram

    if(k > i) { n = array_position(a,b);
                t = mx[im][km]*dis_ij_x[n] + my[im][km]*dis_ij_y[n];
                t1 = mx[im][km]*dis_2_ij_x[n] + my[im][km]*dis_2_ij_y[n];
                t2 = mx[im][km]*dis_2_ij_x[n] - my[im][km]*dis_2_ij_y[n];}
    else { n = array_position(b,a);
           t = mx[km][im]*dis_ij_x[n] + my[km][im]*dis_ij_y[n];
           t1 = mx[km][im]*dis_2_ij_x[n] + my[km][im]*dis_2_ij_y[n];
           t2 = mx[km][im]*dis_2_ij_x[n] - my[km][im]*dis_2_ij_y[n];}

    compensation_factor = Complex(cos(t),sin(t));
    temp = Cmul(off_d0[n],compensation_factor);
    temp1 = Cmul(off_d1[n],compensation_factor);
    c_s = (cos_sin[n] + t2);
    c_s /= (1. + t1);

    if(im < N) {
        if(km < N) {
          temp2 = RCmul(sqr(1./kr),Cmul(kz,kz));
          temp2 = Cmul(temp2,Cmul(off_d[n],compensation_factor));
          return Cadd(Cadd(temp,RCmul(c_s,temp1)),temp2);
                 }
        else if(km < 2*N) {
           if(k > i) t2 = my[im][km]*dis_2_ij_x[n] + mx[im][km]*dis_2_ij_y[n];
           else t2 = my[km][im]*dis_2_ij_x[n] + mx[km][im]*dis_2_ij_y[n];
           si = (t2/2. + dis_ij_x[n]*dis_ij_y[n]/sqr(kr));
           si /= (1. + t1);
          return RCmul(si,Cmul(off_d_xy[n],compensation_factor));
                                 }
        else {
          temp2 = RCmul(sqr(kr),Cmul(kz,Cmul(j,Cadd(temp,temp1))));
          return RCmul(center[i][0]-center[k][0],temp2);
             }
      }
      else if(im < 2*N) {
        if(km < N)
        {
          if(k > i) t2 = my[im][km]*dis_2_ij_x[n] + mx[im][km]*dis_2_ij_y[n];
          else t2 = my[km][im]*dis_2_ij_x[n] + mx[km][im]*dis_2_ij_y[n];
          si = (t2/2. + dis_ij_x[n]*dis_ij_y[n]/sqr(kr));
          si /= (1. + t1);
          return RCmul(si,Cmul(off_d_xy[n],compensation_factor));
        }
        else if(km < 2*N) {
          temp2 = RCmul(sqr(1./kr),Cmul(kz,kz));
          temp2 = Cmul(temp2,Cmul(off_d[n],compensation_factor));
          return Cadd(Csub(temp,RCmul(c_s,temp1)),temp2);
                           }
        else {
          temp2 = RCmul(sqr(kr),Cmul(kz,Cmul(j,Cadd(temp,temp1))));
          return RCmul(center[i][1]-center[k][1],temp2);
                           }
         }
      else {
        if(km < N)
        {
          temp2 = RCmul(sqr(kr),Cmul(kz,Cmul(j,Cadd(temp,temp1))));
          return RCmul(center[i][0]-center[k][0],temp2);
        }
        else if(km < 2*N) {
          temp2 = RCmul(sqr(kr),Cmul(kz,Cmul(j,Cadd(temp,temp1))));
          return RCmul(center[i][1]-center[k][1],temp2);
                           }
        else {
          return Cmul(off_d[n],compensation_factor);
             }
          }
      }
}

// storing needed data for calculating impedance matrix

void pre_processing(kr_,n1)
double kr_;
{
  int i,ii,n=0,a;
  float dis,x,y,x1,y1;

  for(i=0;i<3*N;i++)
    for(ii=0;ii<3*N;ii++)
    {
      mx[i][ii] = ((center[i][0] - cx[0]) - (center[ii][0] - cx[0]));
      my[i][ii] = ((center[i][1] - cy[0]) - (center[ii][1] - cy[0]));
    }

  for(i=0;i<S_N;i++)
  {
    a = IA[i+1] - IA[i];
    for(ii=0;ii<a;ii++)
    {
      x = cx[i+n1*S_N1]; x1 = cx[JA[ii + IA[i]]+n1*S_N1];
      y = cy[i+n1*S_N1]; y1 = cy[JA[ii + IA[i]]+n1*S_N1];
      dis = sqrt(sqr(x - x1)+ sqr(y - y1));
```

```
      dis_ij_x[n] = kr_*(x - x1)/dis;
      dis_ij_y[n] = kr_*(y - y1)/dis;
      dis_2_ij_x[n] = 2.*(x - x1)/sqr(dis);
      dis_2_ij_y[n] = 2.*(y - y1)/sqr(dis);
      cos_sin[n] = (sqr(x-x1) - sqr(y-y1))/sqr(dis);

      n++;
    }
  }
}

// stoing index for sparse matrix scheme (see Orden)

int sparse_matrix_processing(nn,nn1)
{
  int i,ii,n=0;
  float temp;

  for(i=nn*S_N1;i<S_N+nn*S_N1;i++)
  {
    for(ii=(i+1);ii<S_N+nn*S_N1;ii++)
      if(sqrt(sqr(cx[i]-cy[i])+sqr(cx[ii]-cy[ii])) - max_rho < zero_error) n++;
    IA[i+1-nn*S_N1] = n;
  }

  if(nn == 0)
  {
    if(nn1 == 0)
    {
      JA = (int *)malloc(sizeof(int)*n);
      if(JA == NULL) {printf("malloc error : JA\n"); exit(0);}
    }
    else
    {
      JA = (int *)realloc(JA,sizeof(int)*n);
      if(JA == NULL) {printf("n = %d, realloc error : JA\n",n); exit(0);}
    }
  }

  n = 0;
  for(i=nn*S_N1;i<S_N+nn*S_N1;i++)
  {
    for(ii=(i+1);ii<S_N+nn*S_N1;ii++)
    {
//      temp = sqrt(sqr(cx[i+nn*S_N1]-cy[i+nn*S_N1])+sqr(cx[ii+nn*S_N1]-cy[ii+nn*S_N1]))
;
//    if((temp - max_rho) < zero_error) n++;
      if(sqrt(sqr(cx[i]-cy[i])+sqr(cx[ii]-cy[ii])) - max_rho < zero_error)
      {JA[n] = ii-nn*S_N1; n++;}
    }
  }
  return n;
}

void obtain_F_vector(func)
fcomplex (*func)();
{
  Initial_guess(F,J_dis,mN);

  if(strncmp(solver_type,"BCG",10) == 0)
    BCG(func,F,J_dis,P,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"CGS",10) == 0)
    CGS(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB",10) == 0)
    BICGSTAB(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB_s",10) == 0)
    BICGSTAB_s(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else {printf("Invaild solver type...\n"); exit(0);}
}
```

```c
#ifndef EPS1
  #define EPS1 3.e-11
#endif

void gauleg(x1,x2,xx,ww,n)
double x1,x2; int n; double xx[],ww[];
{
  int m,jj,i;
  double z1,z,xm,xl,pp,p3,p2,p1;

  m = (n+1)/2;
  xm = 0.5*(x2+x1);
  xl = 0.5*(x2-x1);
  for(i=1;i<=m;i++)
  {
    z = cos(3.141592654*(i-0.25)/(n+0.5));
    do{
      p1 = 1.0;
      p2 = 0.;
      for(jj=1;jj<=n;jj++)
      {
        p3 = p2;
        p2 = p1;
        p1 = ((2.*jj-1.)*z*p2-(jj-1.)*p3)/jj;
      }
      pp = n*(z*p1-p2)/(z*z-1.);
      z1 = z;
      z = z1 - p1/pp;
    } while (fabs(z-z1) > EPS1);
    xx[i] = xm - xl*z;
    xx[n+1 -i] = xm+xl*z;
    ww[i] = 2.*xl/((1.-z*z)*pp*pp);
    ww[n+1-i] = ww[i];
  }
}
```

```
#ifndef pi
  #define pi M_PI
#endif

#define Ec 0.57721566490
#define e0 (8.85e-12)
#define u0 (pi*4.e-7)

#define ITMAX 100
#define EPS 3.e-7

#define ACC 40.
#define BIGNO 1.e10
#define BIGNI 1.e-10
```

```
// containing the routine needed for EM calculation

fcomplex R_v(e1_,kz1_,e2_,kz2_)
fcomplex e1_,kz1_,e2_,kz2_;
{
  fcomplex temp_;

  temp_ = Csub(Cmul(e2_,kz1_),Cmul(e1_,kz2_));
  return Cdiv(temp_,Cadd(Cmul(e2_,kz1_),Cmul(e1_,kz2_)));
}

fcomplex R_h(e1_,kz1_,e2_,kz2_)
fcomplex e1_,kz1_,e2_,kz2_;
{
  return Cdiv(Csub(kz1_,kz2_),Cadd(kz1_,kz2_));
}
```

```
FILE *file_open(name_,w_r_)
char name_[],w_r_[];
{
  FILE *temp;

  if((temp = fopen(name_,w_r_)) == NULL)
  {printf("File open error : %s\n",name_); exit(0);}
  return temp;
}
```

```
// Lateral wave type incident field due to abitrary oriented dipole
// Must insert this part in middle of program

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

fcomplex i_E_x_(sin_2v,cos_2v,sin_x,cos_x,rho,f2)
double sin_2v,cos_2v,sin_x,cos_x,rho; fcomplex f2;
{
  double a1,a2,o_z,h_;
  fcomplex ans,ans1;
  fcomplex temp,temp1,temp2,exp_t;

  o_z = h_ = 0.;
  a1 = (1. - cos_2v)*lx - sin_2v*ly; a2 = -(1. + cos_2v)*lx - sin_2v*ly;

  temp = Cdiv(sin_wb,Csqrt(cos_wb));
  temp2 = Csub(RCmul(o_z+h_,sin_wb),RCmul(rho,cos_wb));
  temp1 = Cdiv(temp,Cmul(Csqrt(temp2),temp2));
  temp = Cmul(temp1,Cadd(Complex(a1,0.),RCmul(a2,Cdiv(Cmul(cos_wb,cos_wb),k))));
  exp_t = Cexp(Cmul(Cmul(j,k1),Cadd(RCmul((o_z+h_),cos_wb),RCmul(rho,sin_wb))));
  temp = Cmul(Cmul(j,f2),Cmul(exp_t,temp));
  ans1 = RCmul(1./sqrt(rho),temp);

  // due to z-comp. of dipole
  temp = Cmul(Cmul(sin_wb,Csqrt(cos_wb)),Cdiv(exp_t,Cmul(Csqrt(temp2),temp2)));
  temp = Cmul(j,Cmul(f2,Cdiv(Cmul(sin_wb,temp),k)));
  return ans = Cadd(ans1,RCmul(lz*2.*cos_x/sqrt(rho),temp));
}

fcomplex i_E_y_(sin_2v,cos_2v,sin_x,cos_x,rho,f2)
double sin_2v,cos_2v,sin_x,cos_x,rho; fcomplex f2;
{
  double a1,a2,o_z,h_;
  fcomplex ans,ans1;
  fcomplex temp,temp1,temp2,exp_t;

  o_z = h_ = 0.;
  a1 = (1. + cos_2v)*ly - sin_2v*lx; a2 = -(1. - cos_2v)*ly - sin_2v*lx;

  temp = Cdiv(sin_wb,Csqrt(cos_wb));
  temp2 = Csub(RCmul(o_z+h_,sin_wb),RCmul(rho,cos_wb));
  temp1 = Cdiv(temp,Cmul(Csqrt(temp2),temp2));
  temp = Cmul(temp1,Cadd(Complex(a1,0.),RCmul(a2,Cdiv(Cmul(cos_wb,cos_wb),k))));
  exp_t = Cexp(Cmul(Cmul(j,k1),Cadd(RCmul((o_z+h_),cos_wb),RCmul(rho,sin_wb))));
  temp = Cmul(Cmul(j,f2),Cmul(exp_t,temp));
  ans1 = RCmul(1./sqrt(rho),temp);

  // due to z-comp. of dipole
  temp = Cmul(Cmul(sin_wb,Csqrt(cos_wb)),Cdiv(exp_t,Cmul(Csqrt(temp2),temp2)));
  temp = Cmul(j,Cmul(f2,Cdiv(Cmul(sin_wb,temp),k)));
  return ans = Cadd(ans1,RCmul(lz*2.*sin_x/sqrt(rho),temp));
}

fcomplex i_E_z_(sin_x,cos_x,rho,f2)
double sin_x,cos_x,rho; fcomplex f2;
{
  double o_z,h_;
  fcomplex ans,ans1;
  fcomplex temp,temp1,temp2,exp_t;

  o_z = h_ = 0.;

  temp = Cdiv(Cmul(Cmul(sin_wb,sin_wb),sin_wb),Cmul(k,Csqrt(cos_wb)));
  temp2 = Csub(RCmul(o_z+h_,sin_wb),RCmul(rho,cos_wb));
  temp = Cdiv(temp,Cmul(Csqrt(temp2),temp2));
  exp_t = Cexp(Cmul(Cmul(j,k1),Cadd(RCmul((o_z+h_),cos_wb),RCmul(rho,sin_wb))));
  temp = Cmul(Cmul(j,f2),Cmul(exp_t,temp));
  ans1 = RCmul(-lz*2./sqrt(rho),temp);

  // due to x,y-comp. of dipole
  temp = Cmul(Cmul(sin_wb,Csqrt(cos_wb)),Cdiv(exp_t,Cmul(Csqrt(temp2),temp2)));
  temp = Cmul(j,Cmul(f2,Cdiv(Cmul(sin_wb,temp),k)));
  return ans1 = Cadd(ans1,RCmul(2.*(cos_x*lx+sin_x*ly)/sqrt(rho),temp));
}

fcomplex i_field_x(x,y)
double x,y;
{
  fcomplex temp,factor,factor1,exp_t;
  double sin_2v,cos_2v,ang_xy,sin_x,cos_x,ang,rho;

  rho = sqrt(sqr(x - a_x) + sqr(y - a_y));
  ang_xy = phase(Complex(x - a_x,y - a_y));
  sin_2v = sin(2.*ang_xy); cos_2v = cos(2.*ang_xy);
  sin_x = sin(ang_xy); cos_x = cos(ang_xy);

  factor1 = RCmul(1./(4.*pi),Z1);

  temp = i_E_x_(sin_2v,cos_2v,sin_x,cos_x,rho,factor1);
  exp_t = Cexp(RCmul((2.*f_L - a_z),Cmul(j,kz)));
  return Cmul(temp,exp_t);
}

fcomplex i_field_y(x,y)
double x,y;
{
  fcomplex temp,factor,factor1,exp_t;
  double sin_2v,cos_2v,ang_xy,sin_x,cos_x,ang,rho;

  rho = sqrt(sqr(x - a_x) + sqr(y - a_y));
  ang_xy = phase(Complex(x - a_x,y - a_y));
  sin_2v = sin(2.*ang_xy); cos_2v = cos(2.*ang_xy);
  sin_x = sin(ang_xy); cos_x = cos(ang_xy);

  factor1 = RCmul(1./(4.*pi),Z1);

  temp = i_E_y_(sin_2v,cos_2v,sin_x,cos_x,rho,factor1);
  exp_t = Cexp(RCmul((2.*f_L - a_z),Cmul(j,kz)));
  return Cmul(temp,exp_t);
}

fcomplex i_field_z(x,y)
double x,y;
{
  fcomplex temp,factor,factor1,exp_t;
  double sin_2v,cos_2v,ang_xy,sin_x,cos_x,ang,rho;

  rho = sqrt(sqr(x - a_x) + sqr(y - a_y));
  ang_xy = phase(Complex(x - a_x,y - a_y));
  sin_x = sin(ang_xy); cos_x = cos(ang_xy);

  factor1 = RCmul(1./(4.*pi),Z1);
```

```
  temp = i_E_z_(sin_x,cos_x,rho,factor1);
  exp_t = Cexp(RCmul((2.*f_L - a_z),Cmul(j,kz)));
  return Cmul(temp,exp_t);
}

fcomplex i_field_x1(x,y)
double x,y;
{
  int i;
  fcomplex exp_t,i_f;
  double x1,y1,t_a_x,t_a_y;

  t_a_x = a_x; t_a_y = a_y;
  a_x = x; a_y = y;

  i_f = zero;
  for(i=0;i<a_N;i++)
  {
    x1 = array_posi[i][0]; y1 = array_posi[i][1];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    i_f = Cadd(i_f,Cmul(i_field_x(x1,y1),Cexp(Complex(0.,array_pha[i]))));
  }

  exp_t = Cexp(Cadd(RCmul(-k0*(a_x - x),j),RCmul(a_z,Cmul(j,kz))));
  a_x = t_a_x; a_y = t_a_y;
  return Cadd(Cmul(im_field[0],exp_t),i_f);
}

fcomplex i_field_y1(x,y)
double x,y;
{
  int i;
  fcomplex exp_t,i_f;
  double x1,y1,t_a_x,t_a_y;

  t_a_x = a_x; t_a_y = a_y;
  a_x = x; a_y = y;

  i_f = zero;
  for(i=0;i<a_N;i++)
  {
    x1 = array_posi[i][0]; y1 = array_posi[i][1];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    i_f = Cadd(i_f,Cmul(i_field_y(x1,y1),Cexp(Complex(0.,array_pha[i]))));
  }

  exp_t = Cexp(Cadd(RCmul(-k0*(a_x - x),j),RCmul(a_z,Cmul(j,kz))));
  a_x = t_a_x; a_y = t_a_y;
  return Cadd(Cmul(im_field[1],exp_t),i_f);
}

fcomplex i_field_z1(x,y)
double x,y;
{
  int i;
  fcomplex exp_t,i_f;
  double x1,y1,t_a_x,t_a_y;

  t_a_x = a_x; t_a_y = a_y;
  a_x = x; a_y = y;

  i_f = zero;
  for(i=0;i<a_N;i++)
  {
    x1 = array_posi[i][0]; y1 = array_posi[i][1];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    i_f = Cadd(i_f,Cmul(i_field_z(x1,y1),Cexp(Complex(0.,array_pha[i]))));
  }

  exp_t = Cexp(Cadd(RCmul(-k0*(a_x - x),j),RCmul(a_z,Cmul(j,kz))));
  a_x = t_a_x; a_y = t_a_y;
  return Cadd(Cmul(im_field[2],exp_t),i_f);
}

fcomplex i_field_x2(x,y)
double x,y;
{
  fcomplex exp_t;

  exp_t = Cexp(RCmul(-k0*(a_x - x),j));
  return Cmul(im_field[0],exp_t);
}

fcomplex i_field_y2(x,y)
double x,y;
{
  fcomplex exp_t;

  exp_t = Cexp(RCmul(-k0*(a_x - x),j));
  return Cmul(im_field[1],exp_t);
}

fcomplex i_field_z2(x,y)
double x,y;
{
  fcomplex exp_t;

  exp_t = Cexp(RCmul(-k0*(a_x - x),j));
  return Cmul(im_field[2],exp_t);
}
```

```
// Functions needed for mesh

int determine_N()
{
  unsigned short int i,ii,n = 0;
  double ang = 0.;
  double pre_x,pre_y,x,y,c_x,c_y;

  for(i=0;i<=N1;i++)
  {
    x = ra - 2.*ra*i/N1;
    pre_x = ra - 2.*ra*(i-1)/N1;
    c_x = (x + pre_x)/2.;
    if(i != 0 )
    for(ii=0;ii<=N1;ii++)
      {
        y = ra - 2.*ra*ii/N1;
        pre_y = ra - 2.*ra*(ii-1)/N1;
        c_y = (y + pre_y)/2.;
        if((sqr(c_x) +sqr(c_y)) <= sqr(ra))
          n++;
      }
  }
  return n;
}

void mesh(xc,yc,jj,n1)
float xc,yc;
{
  int i,ii,n = 0;
  float ang = 0.;
  float pre_x,pre_y,x,y,c_x,c_y;

  for(i=0;i<=N1;i++)
  {
    x = ra - 2.*ra*i/N1;
    pre_x = ra - 2.*ra*(i-1)/N1;
    c_x = (x + pre_x)/2.;
    if(i != 0 )
    for(ii=0;ii<=N1;ii++)
      {
        y = ra - 2.*ra*ii/N1;
        pre_y = ra - 2.*ra*(ii-1)/N1;
        c_y = (y + pre_y)/2.;
        if((sqr(c_x) +sqr(c_y)) <= sqr(ra))
        {
          center[n + jj*3*N + n1*3*N*S_N1][0] = c_x + xc;
          center[n + jj*3*N + n1*3*N*S_N1][1] = c_y + yc;
          center[n + jj*3*N + N + n1*3*N*S_N1][0] = c_x + xc;
          center[n + jj*3*N + N + n1*3*N*S_N1][1] = c_y + yc;
          center[n + jj*3*N + 2*N + n1*3*N*S_N1][0] = c_x + xc;
          center[n + jj*3*N + 2*N + n1*3*N*S_N1][1] = c_y + yc;
          if(jj == 0 && n == 0) {
            delta_x = fabs(x - pre_x);
            delta_y = fabs(y - pre_y);
          }
          n++;
        }
      }
  }
}

void generate_mesh_array_cir(n,n1,n2,x_center,y_center,radius)
float x_center,y_center,radius;

// Generating scatterers locations within circle around given point
// For array simulation

{
  unsigned int i,jj,flag,n_sc;
  float xc,yc,tmp,min_gap1;
  float delta_y1;
  float idum = (-1.);
  static float pre_x_length;
  fcomplex temp;

  if(min_gap < 1.) min_gap1 = 1.;
  else min_gap1 = min_gap;

  if(n != 0)
    for(i=0;i<n;i++)
    {
      xc = rand0(&idum);
      yc = rand0(&idum);
    }

  x_length = radius;
  delta_y1 = (float)rand0(&idum)*lambda*dy;
  for(jj=0;jj<S_N;jj++)
  {
    n_sc = (int)(density*2.*pi*(sqr(x_length) - sqr(x_length - radius)));
    if(n_sc == 0) n_sc++;
    if((jj % n_sc) == 0 && jj != 0) x_length += radius;
    do{
       flag = 0;
       if(x_length > 2.*radius) xc = rand0(&idum)*2*radius+x_length -2.*radius;
       else xc = rand0(&idum)*radius;
       yc = 2.*pi*rand0(&idum);
       temp = RCmul(xc,Cexp(Complex(0.,yc)));
       xc = temp.r; yc = temp.i;
       xc += x_center; yc += y_center;

       for(i=0;i<jj;i++)
         if((sqrt(sqr(cx[i+n1*S_N1]-xc)+sqr(cy[i+n1*S_N1]-yc))<lambda*min_gap))
           flag = 1;
       if(flag == 0)
       {
         for(i=0;i<n2;i++)
         {
           tmp = sqrt(sqr(xc - array_posi[i][0]) + sqr(yc - array_posi[i][1]));
           if(tmp < lambda*min_gap) { flag = 2; break;}
         }
         if(sqrt(sqr(xc - x_c) + sqr(yc - y_c)) < lambda*min_gap1) flag = 3;
       }
     } while(flag != 0);

    cx[jj+n1*S_N1] = xc; cy[jj+n1*S_N1] = yc;
  }
  if(n == 0) pre_x_length = x_length;
}

void generate_mesh_array_rec(n,n1,n2,x_center,y_center,x_L,y_L)
float x_center,y_center,x_L,y_L;

// Generating scatterers locations within rectangular
// with given width and length
// For array simulation
```

```
{
  unsigned int i,jj,flag,n_sc;
  float xc,yc,tmp,min_gap1;
  float idum = (-1.);
  fcomplex temp;

  if(min_gap < 1.) min_gap1 = 1.;
  else min_gap1 = min_gap;

  if(n != 0)
    for(i=0;i<n;i++)
    {
      xc = rand0(&idum);
      yc = rand0(&idum);
    }

  x_length = x_L; y_length = y_L;
  for(jj=0;jj<S_N;jj++)
  {
    n_sc = (int)(density*(x_length*y_length-(x_length-x_L)*(y_length-y_L)));
    if(n_sc == 0) n_sc++;

    if((jj % n_sc) == 0 && jj != 0) {x_length += x_L; y_length += y_L;}
    do{
     flag = 0;
     if(x_length > x_L) xc = rand0(&idum)*2*x_L+x_length-2.*x_L;
     else xc = rand0(&idum)*x_L;

     if(y_length > y_L) yc = rand0(&idum)*2*y_L+y_length-2.*y_L;
     else yc = rand0(&idum)*y_L;
     xc += x_center - x_length/2.; yc += y_center - y_length/2.;

     for(i=0;i<jj;i++)
       if((sqrt(sqr(cx[i+n1*S_N1]-xc)+sqr(cy[i+n1*S_N1]-yc))<lambda*min_gap))
         flag = 1;
     if(flag == 0)
     {
       for(i=0;i<n2;i++)
       {
         tmp = sqrt(sqr(xc - array_posi[i][0]) + sqr(yc - array_posi[i][1]));
         if(tmp < lambda*min_gap) { flag = 2; break;}
       }
       if(sqrt(sqr(xc - x_c) + sqr(yc - y_c)) < lambda*min_gap1) flag = 3;
     }
    } while(flag != 0);

    cx[jj+n1*S_N1] = xc; cy[jj+n1*S_N1] = yc;
  }
}
```

```c
// Containing the procedue for calculation of mom matrix for 2D problem

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

// TM case with normal incident
// These are valid for non-homogeneous mesh

fcomplex In(x_,y_,xn_,yn_,d_x_,d_y_,k_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_;
{
  fcomplex h,an,bn;
  double dis,ang,temp,k0_dis;

  dis = sqrt(sqr(x_ - xn_) + sqr(y_ - yn_));
  ang = phase(Complex(xn_ - x_,yn_ - y_));
  k0_dis = k_*dis;
  temp = (sqr(cos(ang)) - sqr(sin(ang)))/(k0_dis);

  an = RCmul(-1.,H1(0,k0_dis));
  bn = Csub(RCmul(sqr(sin(ang)),an),RCmul(temp,H1(1,k0_dis)));
  an = Cadd(RCmul(sqr(cos(ang)),an),RCmul(temp,H1(1,k0_dis)));

  h = Cadd(H1(0,k0_dis),RCmul(sqr(k_*d_x_)/24.,an));
  h = Cadd(h,RCmul(sqr(k_*d_y_)/24.,bn));
  h = RCmul(d_x_*d_y_,h);

  return RCmul(sqr(k_),h);
}

fcomplex Inn(x_,y_,xn_,yn_,d_x_,d_y_,k_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_;
{
  fcomplex temp;
  double d1,d2;

  d1 = log(k_*sqrt(sqr(d_x_/2.) + sqr(d_y_/2.))/2.);
  d2 = atan(d_y_/d_x_);
  temp = RCmul((d_x_*d_y_/2.),Complex((Ec + d1 - 3./2.),-1.*pi/2.));
  d1 = sqr(d_x_/2.)*d2 + sqr(d_y_/2.)*(pi/2. - d2);
  temp = Cmul(Cadd(temp,Complex(d1,0.)),RCmul(4./pi,j));

  return RCmul(sqr(k_),temp);
}

// For small cell

fcomplex F_(n_,x_,y_,xn_,yn_,d_x_,d_y_,k_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_;
{
  fcomplex temp,ans;
  double an_,bn_,an1_,an2_,bn1_,bn2_,temp_d;

  an1_ = x_ - xn_ - d_x_/2.;
  an2_ = x_ - xn_ + d_x_/2.;
  bn1_ = y_ - yn_ - d_y_/2.;
  bn2_ = y_ - yn_ + d_y_/2.;

  an_ = (n_ == 1) ? an1_ : an2_;
  bn_ = (n_ == 1) ? bn1_ : bn2_;

  temp_d = an2_*log(sqrt(sqr(an2_) + sqr(bn_))/2.);
  temp_d -= an1_*log(sqrt(sqr(an1_) + sqr(bn_))/2.);
  ans = Complex(0.,-bn_*sqr(k_)/pi*temp_d);
  temp_d = atan(an2_/bn_) - atan(an1_/bn_);
  temp = Complex(0.,(2. - sqr(k_)*bn_)/pi*temp_d);
  temp_d = sqr(k_)*d_x_*bn_/2.;
  ans = Cadd(ans,temp);
  temp = RCmul(temp_d,Complex(-1.,(3. - 2.*Ec)/pi));
  return Cadd(ans,temp);
}

fcomplex G_(n_,x_,y_,xn_,yn_,d_x_,d_y_,k_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_;
{
  fcomplex temp,ans;
  double an_,bn_,an1_,an2_,bn1_,bn2_,temp_d;

  an1_ = x_ - xn_ - d_x_/2.;
  an2_ = x_ - xn_ + d_x_/2.;
  bn1_ = y_ - yn_ - d_y_/2.;
  bn2_ = y_ - yn_ + d_y_/2.;

  an_ = (n_ == 1) ? an1_ : an2_;
  bn_ = (n_ == 1) ? bn1_ : bn2_;

  temp_d = bn2_*log(sqrt(sqr(bn2_) + sqr(an_))/2.);
  temp_d -= bn1_*log(sqrt(sqr(bn1_) + sqr(an_))/2.);
  ans = Complex(0.,-an_*sqr(k_)/pi*temp_d);
  temp_d = atan(bn2_/an_) - atan(bn1_/an_);
  temp = Complex(0.,(2. - sqr(k_)*an_)/pi*temp_d);
  temp_d = sqr(k_)*d_y_*an_/2.;
  ans = Cadd(ans,temp);
  temp = RCmul(temp_d,Complex(-1.,(3. - 2.*Ec)/pi));
  return Cadd(ans,temp);
}

// TE case with normal incident
// These are valid for only homogeneous mesh

fcomplex In_xx(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex h,h1;
  double dis,ang,temp,k0_dis;

  dis = sqrt(sqr(x_ - xn_) + sqr(y_ - yn_));

  if((dis - pi/(30.*k_)) < 1.e-10)
    h = Csub(F_(1,x_,y_,xn_,yn_,d_x_,d_y_,k_),F_(2,x_,y_,xn_,yn_,d_x_,d_y_,k_));
  else {
    ang = phase(Complex(xn_ - x_,yn_ - y_));
    k0_dis = k_*dis;
    temp = (sqr(cos(ang)) - sqr(sin(ang)))/(k0_dis);

    h = Cadd(RCmul(sqr(sin(ang)),H1(0,k0_dis)),RCmul(temp,H1(1,k0_dis)));
    temp = sqr(d_x_)*(1. - sqr(k_*d_x_)/24.);
    h = RCmul(temp*k_*k_,h);
      }
  if(Cabs(kz_) > 1.e-15) {
    h1 = In(x_,y_,xn_,yn_,d_x_,d_y_,k_);
```

```
    h1 = RCmul(sqr(1./k_),Cmul(Cmul(kz_,kz_),h1));
    return Cadd(h1,h);
                              }
  else return h;
}

fcomplex In_yy(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex h,h1;
  double dis,ang,temp,k0_dis;

  dis = sqrt(sqr(x_ - xn_) + sqr(y_ - yn_));

  if((dis - pi/(30.*k_)) < 1.e-10)
    h = Csub(G_(1,x_,y_,xn_,yn_,d_x_,d_y_,k_),G_(2,x_,y_,xn_,yn_,d_x_,d_y_,k_));
  else {
    ang = phase(Complex(xn_ - x_,yn_ - y_));
    k0_dis = k_*dis;
    temp = (-sqr(cos(ang)) + sqr(sin(ang)))/k0_dis;

    h = Cadd(RCmul(sqr(cos(ang)),H1(0,k0_dis)),RCmul(temp,H1(1,k0_dis)));
    temp = sqr(d_x_)*(1. - sqr(k_*d_x_)/24.);
    h = RCmul(temp*k_*k_,h);
       }
  if(Cabs(kz_) > 1.e-15) {
    h1 = In(x_,y_,xn_,yn_,d_x_,d_y_,k_);
    h1 = RCmul(sqr(1./k_),Cmul(Cmul(kz_,kz_),h1));
    return Cadd(h1,h);
                              }
  else return h;
}

fcomplex In_xy(x_,y_,xn_,yn_,d_x_,d_y_,k_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_;
{
  fcomplex temp;
  double t_x,t_y;

  t_x = sqr(x_ - xn_ - d_x_/2.); t_y = sqr(y_ - yn_ - d_y_/2.);
  temp = H1(0,k_*sqrt(t_x + t_y));
  t_x = sqr(x_ - xn_ - d_x_/2.); t_y = sqr(y_ - yn_ + d_y_/2.);
  temp = Csub(temp,H1(0,k_*sqrt(t_x + t_y)));
  t_x = sqr(x_ - xn_ + d_x_/2.); t_y = sqr(y_ - yn_ - d_y_/2.);
  temp = Csub(temp,H1(0,k_*sqrt(t_x + t_y)));
  t_x = sqr(x_ - xn_ + d_x_/2.); t_y = sqr(y_ - yn_ + d_y_/2.);
  temp = Cadd(temp,H1(0,k_*sqrt(t_x + t_y)));
  return temp;
}

fcomplex Inn_xx(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex temp,h1;
  double d1,d2;

  d1 = log(k_*sqrt(sqr(d_x_/2.) + sqr(d_y_/2.))/2.);
  d2 = atan(d_y_/d_x_);
  temp = RCmul((d_x_*d_y_*sqr(k_)/2.),Complex((Ec + d1 - 3./2.),-pi/2.));
  d1 = 2.*d2 + sqr(d_y_*k_/2.)*(pi/2. - d2);
  temp = Cmul(Cadd(temp,Complex(d1,0.)),RCmul(4./pi,j));

  if(Cabs(kz_) > 1.e-15) {
    h1 = Inn(x_,y_,xn_,yn_,d_x_,d_y_,k_);
    h1 = RCmul(sqr(1./k_),Cmul(Cmul(kz_,kz_),h1));
                              }
  else h1 = zero;
  return Cadd(temp,h1);
}

fcomplex Inn_yy(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex temp,h1;
  double d1,d2;

  d1 = log(k_*sqrt(sqr(d_x_/2.) + sqr(d_y_/2.))/2.);
  d2 = atan(d_y_/d_x_);
  temp = RCmul((d_x_*d_y_*sqr(k_)/2.),Complex((Ec + d1 - 3./2.),-pi/2.));
  d1 = 2.*(pi/2. - d2) + sqr(d_x_*k_/2.)*d2;
  temp = Cmul(Cadd(temp,Complex(d1,0.)),RCmul(4./pi,j));

  if(Cabs(kz_) > 1.e-15) {
    h1 = Inn(x_,y_,xn_,yn_,d_x_,d_y_,k_);
    h1 = RCmul(sqr(1./k_),Cmul(Cmul(kz_,kz_),h1));
                              }
  else h1 = zero;

  return Cadd(temp,h1);
}

// Oblique incident case
// Homogeneous mesh

fcomplex In_xz(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex h;
  double dis,alpha_n,ang_;

  dis = sqrt(sqr(x_ - xn_) + sqr(y_ - yn_));
  alpha_n = d_x_*d_y_*(1. - sqr(k_*d_x_)/24.);
//  ang_ = phase(Complex(x_-xn_,y_-yn_));

  h = RCmul(alpha_n,H1(1,k_*dis));
//  return RCmul(k_*cos(ang_),Cmul(kz_,Cmul(h,j)));
  return RCmul(k_*(x_ - xn_)/dis,Cmul(kz_,Cmul(h,j)));
}

fcomplex In_yz(x_,y_,xn_,yn_,d_x_,d_y_,k_,kz_)
double x_,y_,xn_,yn_,d_x_,d_y_,k_; fcomplex kz_;
{
  fcomplex h;
  double dis,alpha_n,ang_;

  dis = sqrt(sqr(x_ - xn_) + sqr(y_ - yn_));
  alpha_n = d_x_*d_y_*(1. - sqr(k_*d_x_)/24.);
//  ang_ = phase(Complex(x_-xn_,y_-yn_));

  h = RCmul(alpha_n,H1(1,k_*dis));
//  return RCmul(k_*sin(ang_),Cmul(kz_,Cmul(h,j)));
  return RCmul(k_*(y_ - yn_)/dis,Cmul(kz_,Cmul(h,j)));
}
```

```
float *malloc_f_1(n)
{
  float *v;

  v = (float *)malloc(sizeof(float)*n);
  if(v == NULL) {printf("malloc error : malloc_f_1 \n"); exit(0);}

  return v;
}

double *malloc_d_1(n)
{
  double *v;

  v = (double *)malloc(sizeof(double )*n);
  if(v == NULL) {printf("malloc error : malloc_d_1 \n"); exit(0);}

  return v;
}

fcomplex *malloc_c_1(n)
{
  fcomplex *vec;

  vec = (fcomplex *)malloc(sizeof(fcomplex)*n);
  if( vec == NULL) {printf("Malloc error....\n"); exit(0);}
  return vec;
}

fcomplex **malloc_c_2(n,n1)
{
  fcomplex **vec;
  int i_;

  vec = (fcomplex **)malloc(sizeof(fcomplex *)*n);
  if( vec == NULL) {printf("Malloc error....\n"); exit(0);}
  for(i_=0;i_<n;i_++)
  {
    vec[i_] = (fcomplex *)malloc(sizeof(fcomplex)*n1);
    if(vec[i_] == NULL) {printf("Malloc error....\n"); exit(0);}
  }
  return vec;
}

void free_d_2(i,m)
int i;double **m;
{
  register int l;

  for(l=i-1;l>=0;l--)
        free((double *)m[l]);
  free((double **)m);
}

void free_c_2(i,m)
int i; fcomplex **m;
{
  register int l;

  for(l=i-1;l>=0;l--)
        free((fcomplex *)m[l]);
  free((fcomplex **)m);
}
```

```
// Near field calculation of finite cylinder with the same current as
// infinite cylinder
// Mesh must be homogeneous and small enough to use mid-point rule
// for integration with respective to x,y


#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef distance
  #define distance(x,x1,y,y1,z,z1) sqrt(sqr(x - x1)+sqr(y - y1)+sqr(z - z1))
#endif

fcomplex q_gaus(func,aa,bb,x_,y_,z_,xn_,yn_,dx_,k_,kz_,n__,xx_,ww_)
fcomplex (*func)(); double aa,bb,x_,y_,z_,xn_,yn_,dx_; fcomplex k_,kz_;
double *xx_,*ww_;
{
  int i,m;
  double xr,xm,dx;
  fcomplex s,temp_;

  xm=0.5*(bb+aa);
  xr=0.5*(bb-aa);
  s=zero;
  m = (n__+1)/2;
  for (i=1;i<=n__;i++) {
    dx = xx_[i]*xr;
    temp_ = (*func)(x_,y_,z_,xn_,yn_,xm+dx,dx_,k_,kz_);
    temp_ = Cadd(temp_,(*func)(x_,y_,z_,xn_,yn_,xm-dx,dx_,k_,kz_));
    s = Cadd(s,RCmul(ww_[i],temp_));
  }
  return s = RCmul(xr,s);
}

fcomplex I_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_;
{
  double dis;

  dis = distance(x_,xn_,y_,yn_,z_,z1_);
  return RCmul(sqr(dx_)/dis,Cexp(RCmul(dis,Cmul(j,k_))));
}

fcomplex Ixx_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_;
{
  double dis,temp;
  fcomplex ans;

  dis = distance(x_,xn_,y_,yn_,z_,z1_); temp = (x_- xn_)/dis; temp *= temp;
  ans = RCmul(temp,Cmul(j,k_));
  ans.r += (1. - 3.*temp)/dis;
  ans = Cmul(ans,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  return Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
}

fcomplex Iyy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_;
{
  double dis,temp;
  fcomplex ans;

  dis = distance(x_,xn_,y_,yn_,z_,z1_); temp = (y_ - yn_)/dis; temp *= temp;
  ans = RCmul(temp,Cmul(j,k_));
  ans.r += (1. - 3.*temp)/dis;
  ans = Cmul(ans,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  return Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
}

fcomplex Izz_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_;
{
  double dis,temp;
  fcomplex ans;

  dis = distance(x_,xn_,y_,yn_,z_,z1_); temp = (z_ - z1_)/dis; temp *= temp;
  ans = RCmul(temp,Cmul(j,k_));
  ans.r += (1. - 3.*temp)/dis;
  ans = Cmul(ans,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  return Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
}

fcomplex Ixy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_;
{
  double dis,temp;
  fcomplex ans;

  dis = distance(x_,(xn_+dx_/2.),y_,(yn_+dx_/2.),z_,z1_);
  ans = RCmul(1./dis,Cexp(RCmul(dis,Cmul(j,k_))));
  dis = distance(x_,(xn_+dx_/2.),y_,(yn_-dx_/2.),z_,z1_);
  ans = Csub(ans,RCmul(1./dis,Cexp(RCmul(dis,Cmul(j,k_)))));
  dis = distance(x_,(xn_-dx_/2.),y_,(yn_+dx_/2.),z_,z1_);
  ans = Csub(ans,RCmul(1./dis,Cexp(RCmul(dis,Cmul(j,k_)))));
  dis = distance(x_,(xn_-dx_/2.),y_,(yn_-dx_/2.),z_,z1_);
  return RCmul(sqr(dx_),Cadd(ans,RCmul(1./dis,Cexp(RCmul(dis,Cmul(j,k_))))));
}

fcomplex Ixz_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp;
  fcomplex ans,ans1;

  dis = distance(x_,xn_,y_,yn_,z_,0.); temp = (x_ - xn_)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);

  dis = distance(x_,xn_,y_,yn_,z_,z1_); temp = (x_ - xn_)/dis;
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
```

```
  ans1 = Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(ans1,Cexp(RCmul(-z1_,Cmul(j,kz_))));

  return Csub(ans,ans1);
}

fcomplex Iyz_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp;
  fcomplex ans,ans1;

  dis = distance(x_,xn_,y_,yn_,z_,0.); temp = (y_ - yn_)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);

  dis = distance(x_,xn_,y_,yn_,z_,z1_); temp = (y_ - yn_)/dis;
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans1 = Cmul(RCmul(sqr(dx_/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(ans1,Cexp(RCmul(-z1_,Cmul(j,kz_))));

  return Csub(ans,ans1);
}

fcomplex integrand_xx_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
{
  fcomplex ans;

  ans = Ixx_(x_,y_,z_,xn_,yn_,z1_,dx_,k_);
  ans = Cadd(ans,Cmul(Cmul(k_,k_),I_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)));
  return Cmul(ans,Cexp(RCmul(-z1_,Cmul(j,kz_))));
}

fcomplex integrand_yy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
{
  fcomplex ans;

  ans = Iyy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_);
  ans = Cadd(ans,Cmul(Cmul(k_,k_),I_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)));
  return Cmul(ans,Cexp(RCmul(-z1_,Cmul(j,kz_))));
}

fcomplex integrand_zz_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
{
  fcomplex ans;

  ans = Izz_(x_,y_,z_,xn_,yn_,z1_,dx_,k_);
  ans = Cadd(ans,Cmul(Cmul(k_,k_),I_(x_,y_,z_,xn_,yn_,z1_,dx_,k_)));
  return Cmul(ans,Cexp(RCmul(-z1_,Cmul(j,kz_))));
}

fcomplex integrand_xy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_,kz_)
double x_,y_,z_,xn_,yn_,z1_,dx_; fcomplex k_,kz_;
{
  fcomplex ans;

  ans = Ixy_(x_,y_,z_,xn_,yn_,z1_,dx_,k_);
  return Cmul(ans,Cexp(RCmul(-z1_,Cmul(j,kz_))));
}

fcomplex Exx_(x_,y_,z_,xn_,yn_,dx_,L_,k_,kz_,int_N_,xx_,ww_)
double x_,y_,z_,xn_,yn_,dx_,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(integrand_xx_,0.,L_,x_,y_,z_,xn_,yn_,dx_,k_,kz_,int_N_,xx_,ww_);
}

fcomplex Exy_(x_,y_,z_,xn_,yn_,dx_,L_,k_,kz_,int_N_,xx_,ww_)
double x_,y_,z_,xn_,yn_,dx_,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(integrand_xy_,0.,L_,x_,y_,z_,xn_,yn_,dx_,k_,kz_,int_N_,xx_,ww_);
}

fcomplex Eyy_(x_,y_,z_,xn_,yn_,dx_,L_,k_,kz_,int_N_,xx_,ww_)
double x_,y_,z_,xn_,yn_,dx_,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(integrand_yy_,0.,L_,x_,y_,z_,xn_,yn_,dx_,k_,kz_,int_N_,xx_,ww_);
}

fcomplex Ezz_(x_,y_,z_,xn_,yn_,dx_,L_,k_,kz_,int_N_,xx_,ww_)
double x_,y_,z_,xn_,yn_,dx_,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(integrand_zz_,0.,L_,x_,y_,z_,xn_,yn_,dx_,k_,kz_,int_N_,xx_,ww_);
}
```

```
#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1./M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1./M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

float rand0(idum)
int *idum;
{
  static long ix1,ix2,ix3;
  static float r[98];
  float temp;
  static int iff = 0;
  int jj;

  if(*idum < 0 || iff == 0) {
    iff = 1;
    ix1 = (IC1 - (*idum)) % M1;
    ix1 = (IA1*ix1 + IC1) % M1;
    ix2 = ix1 % M2;
    ix1 = (IA1*ix1 + IC1) % M1;
    ix3 = ix1 % M3;
    for(jj=1;jj<=97;jj++) {
      ix1 = (IA1*ix1 + IC1) % M1;
      ix2 = (IA2*ix2 + IC2) % M2;
      r[jj] = (ix1+ix2*RM2)*RM1;
    }
    *idum = 1;
  }
  ix1 = (IA1*ix1 + IC1) % M1;
  ix2 = (IA2*ix2 + IC2) % M2;
  ix3 = (IA3*ix3 + IC3) % M3;
  jj = 1 + ((97*ix3)/M3);
  if(jj > 97 || jj <1) { printf("Rand0 : This cannot happen\n"); exit(0);}
  temp = r[jj];
  r[jj] = (ix1+ix2*RM2)*RM1;
  return temp;
}

float gasdev(idum)
int *idum;
{
  static int iset = 0;
  static float gset;
  float fac,r,v1,v2;

  if(iset == 0) {
    do {
      v1 = 2.*rand0(idum) - 1.;
      v2 = 2.*rand0(idum) - 1.;
      r = v1*v1 + v2*v2;
    } while (r >= 1.);
    fac = sqrt(-2*log(r)/r);
    gset = v1*fac;
    iset = 1;
    return v2*fac;
  } else {
      iset = 0;
      return gset;
  }
}
```

```c
void read_conf_array()
{
  double ssss;
  FILE *f1;
  char temp[200];

  f1 = file_open("array.conf","r");

  fscanf(f1,"%s %s %s %s\n",temp,temp,temp,temp);
  fscanf(f1,"%f %f %f %f\n",&f0,&x_c,&y_c,&z_c);

  fscanf(f1,"\n%s %s %s %s %s %s %s\n",temp,temp,temp,temp,temp,temp,temp);
  fscanf(f1,"%f %f %f ",&ra,&min_gap,&density);
  fscanf(f1,"%f %f %f %f\n",&f_L,&s_L,&s_v,&j_error);

  fscanf(f1,"\n%s %s %s %s %s\n",temp,temp,temp,temp,temp);
  fscanf(f1,"%lf %lf ",&er_r,&er_i);
  fscanf(f1,"%lf %lf %lf\n",&eg_r,&eg_i,&sigma2);

  fscanf(f1,"\n%s %s %s %s %s %s %s\n",temp,temp,temp,temp,temp,temp,temp);
  fscanf(f1,"%d %d %d %d %d ",&a_N,&N1,&S_N1,&S_N2,&i_N);
  fscanf(f1,"%d %d\n",&int_N,&maxiter);

  fclose(f1);

  y_length = 50.; dy = 1.; error_it = 5.; delta = 1.e-8; pre_cond = 0; sigma1 = 0.;
  strcpy(solver_type,"BICGSTAB_s");
}

void real_array_info()
{
  int i;
  char temp[50];
  FILE *f1;

  f1 = file_open("array_info.conf","r");

  for(i=0;i<a_N;i++)
  {
   if(i == 0) {
     fscanf(f1,"%s %s %s %s %s %s %s\n",temp,temp,temp,temp,temp,temp,temp);
   }
   fscanf(f1,"%f %f %f ",&array_posi[i][0],&array_posi[i][1],&array_posi[i][2]);
   fscanf(f1,"%f %f %f ",&array_ori[i][0],&array_ori[i][1],&array_ori[i][2]);
   fscanf(f1,"%f\n",&array_pha[i]);
  }
  fclose(f1);
}
```

```
// Calculation of reflected wave

fcomplex R_h1(sin_v,cos_v)
double sin_v,cos_v;
{
  fcomplex ans,temp;

  ans = Csqrt(Csub(k,Complex(sqr(sin_v),0.)));
  temp = Complex(cos_v,0.);
  ans = Cdiv(Csub(temp,ans),Cadd(RCmul(cos_v,k),ans));
  return RCmul(2.*cos_v*sin_v,ans);
}

fcomplex r_integral_xx(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
{
  fcomplex rh,ans,kz1,kgz;
  double dis,tmp;

  dis = sqrt(sqr(x - xn) + sqr(y - yn) + sqr(z -zn));
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h(e1,kz1,eg,kgz);

  return Cmul(rh,integrand_xx_(x,y,z,xn,yn,zn,dx,k_,kz_));
}

fcomplex r_integral_yy(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
{
  fcomplex rh,ans,kz1,kgz;
  double dis,tmp;

  dis = sqrt(sqr(x - xn) + sqr(y - yn) + sqr(z -zn));
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h(e1,kz1,eg,kgz);

  return Cmul(rh,integrand_yy_(x,y,z,xn,yn,zn,dx,k_,kz_));
}

fcomplex r_integral_zz(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
{
  fcomplex rv,ans,kz1,kgz;
  double dis,tmp;

  dis = sqrt(sqr(x - xn) + sqr(y - yn) + sqr(z -zn));
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rv = R_v(e1,kz1,eg,kgz);

  return Cmul(rv,integrand_zz_(x,y,z,xn,yn,zn,dx,k_,kz_));
}

fcomplex r_integral_zz1(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
{
  fcomplex rh,ans,kz1,kgz;
  double dis,tmp;

  dis = sqrt(sqr(x - xn) + sqr(y - yn) + sqr(z -zn));
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = Cmul(R_h1(tmp,(z - zn)/dis),Cexp(RCmul(-zn,Cmul(j,kz_))));

  return Cmul(rh,Izz_(x,y,z,xn,yn,zn,dx,k_));
}

fcomplex r_integral_xy(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
{
  fcomplex rh,ans,kz1,kgz;
  double dis,tmp;

  dis = sqrt(sqr(x - xn) + sqr(y - yn) + sqr(z -zn));
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h(e1,kz1,eg,kgz);

  return Cmul(rh,integrand_xy_(x,y,z,xn,yn,zn,dx,k_,kz_));
}

fcomplex r_Ixz(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp,tmp;
  fcomplex ans,ans1,rv,kz1,kgz;

  dis = distance(x,xn,y,yn,z,0.);
  kz1 = RCmul(z/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rv = R_v(e1,kz1,eg,kgz);

  temp = (x - xn)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
  ans = Cmul(ans,rv);

  dis = distance(x,xn,y,yn,z,zn); temp = (x - xn)/dis;
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rv = R_v(e1,kz1,eg,kgz);
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans1 = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(Cmul(ans1,Cexp(RCmul(-zn,Cmul(j,kz_)))),rv);

  return Csub(ans,ans1);
}

fcomplex r_Ixz1(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp,tmp;
  fcomplex ans,ans1,rh,kz1,kgz;

  dis = distance(x,xn,y,yn,z,0.);
  kz1 = RCmul(z/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h1(tmp,z/dis);

  temp = (x - xn)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
  ans = Cmul(ans,rh);
```

```
  dis = distance(x,xn,y,yn,z,zn); temp = (x - xn)/dis;
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h1(tmp,(z - zn)/dis);
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans1 = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(Cmul(ans1,Cexp(RCmul(-zn,Cmul(j,kz_)))),rh);

  return Csub(ans,ans1);
}

fcomplex r_Iyz(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp,tmp;
  fcomplex ans,ans1,rv,kz1,kgz;

  dis = distance(x,xn,y,yn,z,0.);
  kz1 = RCmul(z/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn));
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rv = R_v(e1,kz1,eg,kgz);

  temp = (y - yn)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
  ans = Cmul(ans,rv);

  dis = distance(x,xn,y,yn,z,zn); temp = (y - yn)/dis;
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn));
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rv = R_v(e1,kz1,eg,kgz);
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans1 = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(Cmul(ans1,Cexp(RCmul(-zn,Cmul(j,kz_)))),rv);

  return Csub(ans,ans1);
}

fcomplex r_Iyz1(x,y,z,xn,yn,zn,dx,k_,kz_)
double x,y,z,xn,yn,zn,dx; fcomplex k_,kz_;
// z1_ is cylinder height.
{
  double dis,temp,tmp;
  fcomplex ans,ans1,rh,kz1,kgz;

  dis = distance(x,xn,y,yn,z,0.);
  kz1 = RCmul(z/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h1(tmp,z/dis);

  temp = (y - yn)/dis;
  ans = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans);
  ans = Cmul(ans,rh);

  dis = distance(x,xn,y,yn,z,zn); temp = (y - yn)/dis;
  kz1 = RCmul((z - zn)/dis,k1); tmp = sqrt(sqr(x-xn) + sqr(y-yn))/dis;
  kgz = Csqrt(Csub(Cmul(kg,kg),RCmul(sqr(tmp),Cmul(k1,k1))));
  rh = R_h1(tmp,(z - zn)/dis);
  ans1 = RCmul(temp,Csub(RCmul(dis,Cmul(j,k_)),Cone));
  ans1 = Cmul(RCmul(sqr(dx/dis),Cexp(RCmul(dis,Cmul(j,k_)))),ans1);
  ans1 = Cmul(Cmul(ans1,Cexp(RCmul(-zn,Cmul(j,kz_)))),rh);

  return Csub(ans,ans1);
}

fcomplex r_Exx(x,y,z,xn,yn,dx,L_,k_,kz_,int_N_,xx_,ww_)
double x,y,z,xn,yn,dx,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(r_integral_xx,-L_,0.,x,y,z,xn,yn,dx,k_,kz_,int_N_,xx_,ww_);
}

fcomplex r_Exy(x,y,z,xn,yn,dx,L_,k_,kz_,int_N_,xx_,ww_)
double x,y,z,xn,yn,dx,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(r_integral_xy,-L_,0.,x,y,z,xn,yn,dx,k_,kz_,int_N_,xx_,ww_);
}

fcomplex r_Eyy(x,y,z,xn,yn,dx,L_,k_,kz_,int_N_,xx_,ww_)
double x,y,z,xn,yn,dx,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(r_integral_yy,-L_,0.,x,y,z,xn,yn,dx,k_,kz_,int_N_,xx_,ww_);
}

fcomplex r_Ezz(x,y,z,xn,yn,dx,L_,k_,kz_,int_N_,xx_,ww_)
double x,y,z,xn,yn,dx,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(r_integral_zz,-L_,0.,x,y,z,xn,yn,dx,k_,kz_,int_N_,xx_,ww_);
}

fcomplex r_Ezz1(x,y,z,xn,yn,dx,L_,k_,kz_,int_N_,xx_,ww_)
double x,y,z,xn,yn,dx,L_; fcomplex k_,kz_; double *xx_,*ww_;
{
  return q_gaus(r_integral_zz1,-L_,0.,x,y,z,xn,yn,dx,k_,kz_,int_N_,xx_,ww_);
}

fcomplex scattered_field_x_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;

// due to Finite cylinder

{
  double d_x,d_y,xn,yn;
  fcomplex temp,temp1,temp2,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];

    temp = Cmul(Exx_(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + 3*n*N]);
    temp1 = Cmul(Exy_(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + N +3*n*N]);

    temp2 = Cmul(Ixz_(x,y,z,xn,yn,L,d_x,k,kz),J_dis[i + 2*N +3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
  return Cmul(coef,sum);
}

fcomplex r_scattered_field_x_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;
```

```
// due to Finite cylinder & reflected wave

{
  double d_x,d_y,xn,yn,dis,cos_v,sin_v;
  fcomplex temp,temp1,temp2,temp3,temp4,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];
    dis = sqrt(sqr(x - xn) + sqr(y - yn));
    cos_v = (x - xn)/dis; sin_v = (y - yn)/dis;

    temp3 = r_Ixz1(x,y,z,xn,yn,-L,d_x,k,kz);
    temp = Cmul(r_Exx(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + 3*n*N]);
    temp4 = Cmul(temp3,J_dis[i + 3*n*N]);
    temp = Cadd(temp,RCmul(cos_v,temp4));
    temp1 = Cmul(r_Exy(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i+N+3*n*N]);
    temp4 = Cmul(temp3,J_dis[i+N+3*n*N]);
    temp1 = Cadd(temp1,RCmul(sin_v,temp4));
    temp2 = Cmul(r_Ixz(x,y,z,xn,yn,-L,d_x,k,kz),J_dis[i + 2*N +3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
  return Cmul(coef,sum);
}

fcomplex scattered_field_y_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;

// due to Finite cylinder

{
  double d_x,d_y,xn,yn;
  fcomplex temp,temp1,temp2,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];

    temp = Cmul(Exy_(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + 3*n*N]);
    temp1 = Cmul(Eyy_(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + N +3*n*N]);

    temp2 = Cmul(Iyz_(x,y,z,xn,yn,L,d_x,k,kz),J_dis[i + 2*N +3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
  return Cmul(coef,sum);
}

fcomplex r_scattered_field_y_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;

// due to Finite cylinder & reflected wave

{
  double d_x,d_y,xn,yn,dis,cos_v,sin_v;
  fcomplex temp,temp1,temp2,temp3,temp4,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];
    dis = sqrt(sqr(x - xn) + sqr(y - yn));
    cos_v = (x - xn)/dis; sin_v = (y - yn)/dis;

    temp3 = r_Iyz1(x,y,z,xn,yn,-L,d_x,k,kz);
    temp = Cmul(r_Exy(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i + 3*n*N]);
    temp4 = Cmul(temp3,J_dis[i + 3*n*N]);
    temp = Cadd(temp,RCmul(cos_v,temp4));
    temp1 = Cmul(r_Eyy(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i+N+3*n*N]);
    temp4 = Cmul(temp3,J_dis[i + N +3*n*N]);
    temp1 = Cadd(temp1,RCmul(sin_v,temp4));
    temp2 = Cmul(r_Iyz(x,y,z,xn,yn,-L,d_x,k,kz),J_dis[i + 2*N +3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
  return Cmul(coef,sum);
}

fcomplex scattered_field_z_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;

// due to Finite cylinder & reflected wave

{
  double d_x,d_y,xn,yn;
  fcomplex temp,temp1,temp2,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];

    temp = Cmul(Ixz_(x,y,z,xn,yn,L,d_x,k,kz),J_dis[i + 3*n*N]);
    temp1 = Cmul(Iyz_(x,y,z,xn,yn,L,d_x,k,kz),J_dis[i + N +3*n*N]);
    temp2 = Cmul(Ezz_(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i+2*N+3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
```

```
  return Cmul(coef,sum);
}

fcomplex r_scattered_field_z_f(x,y,z,k,kz,n,n1,L)
double x,y,z,L; fcomplex k,kz;

// due to Finite cylinder & reflected wave

{
  double d_x,d_y,xn,yn,dis,sin_v,cos_v;
  fcomplex temp,temp1,temp2,temp3,temp4,sum,coef;
  unsigned short int i;

  sum = zero;
  d_x = delta_x;
  d_y = delta_y;
  coef = RCmul(z0/(4.*pi*k0),Cdiv(j,e1));

  for(i=0;i<N;i++)
  {
    xn = center[i + 3*n*N + 3*N*n1*S_N1][0];
    yn = center[i + 3*n*N + 3*N*n1*S_N1][1];
    dis = sqrt(sqr(x - xn) + sqr(y - yn));
    cos_v = (x - xn)/dis; sin_v = (y - yn)/dis;

    temp3 = r_Ezz1(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww);
    temp = Cmul(r_Ixz(x,y,z,xn,yn,-L,d_x,k,kz),J_dis[i + 3*n*N]);
    temp4 = Cmul(temp3,J_dis[i+3*n*N]);
    temp = Cadd(temp,RCmul(cos_v,temp4));
    temp1 = Cmul(r_Iyz(x,y,z,xn,yn,-L,d_x,k,kz),J_dis[i + N +3*n*N]);
    temp4 = Cmul(temp3,J_dis[i+N+3*n*N]);
    temp1 = Cadd(temp1,RCmul(sin_v,temp4));
    temp2 = Cmul(r_Ezz(x,y,z,xn,yn,d_x,L,k,kz,int_N,xx,ww),J_dis[i+2*N+3*n*N]);
    temp = Cadd(temp,Cadd(temp1,temp2));

    sum = Cadd(sum,temp);
  }
  return Cmul(coef,sum);
}
```

```
// Functions needed for indexing of sparse matrix

int array_position(a,b)
{
  int n1,ii,ref;

  n1 = IA[a+1] - IA[a]; ref = b-a-1;
  if(JA[IA[a]+ref] == b) return IA[a]+ref;
  for(ii=(n1-1);ii>=0;ii--) if(JA[IA[a]+ii] == b) return (IA[a] + ii);
  printf("Error in array_position routine!!!\n"); exit(0);
}
```

```
#undef j

double sinc(x)
double x;
{
  if(x == 0.) return 1.;
  else return sin(x)/x;
}

double bessj0(x1)
double x1;
{
  double ax,z;
  double xx,y1,ans,ans1,ans2;

  if((ax = fabs(x1)) < 8.)
  {
    y1 = x1*x1;
    ans1 = 57568490574.+y1*(-13362590354.+y1*(651619640.7+y1*(-11214424.18
           +y1*(77392.33017+y1*(-184.9052456)))));
    ans2 = 57568490411.+y1*(1029532985.+y1*(9494680.718+y1*(59272.64853
           +y1*(267.8532712+y1*1.))));
    ans = ans1/ans2;
  }
  else
  {
    z = 8./ax;
    y1=z*z;
    xx = ax - 0.785398164;
    ans1=1.+y1*(-0.1098628627e-2+y1*(0.2734510407e-4
         +y1*(-0.2073370639e-5+y1*0.2093887211e-6)));
    ans2=-0.1562499995e-1+y1*(0.1430488765e-3+y1*(-0.6911147651e-5
         +y1*(0.7621095161e-6-y1*0.934935152e-7)));
    ans = sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
  }
  return ans;
}

double bessy0(x1)
double x1;
{
  double z,xx,y1,ans,ans1,ans2;

  if(x1 < 8.)
  {
    y1 = x1*x1;
    ans1 = -2957821389.+y1*(7062834065.+y1*(-512359803.6
           +y1*(10879881.29+y1*(-86327.92757+y1*228.4622733))));
    ans2 = 40076544269.+y1*(745249964.8+y1*(7189466.438
           +y1*(47447.26470+y1*(226.1030244+y1*1.))));
    ans = ans1/ans2 + 0.636619772*bessj0(x1)*log(x1);
  }
  else
  {
    z = 8./x1;
    y1 = z*z;
    xx = x1 - 0.785398164;
    ans1 = 1. + y1*(-0.1098628627e-2+y1*(0.2734510407e-4+
           y1*(-0.2073370639e-5+y1*0.2093887211e-6)));
    ans2 = -0.1562499995e-1+y1*(0.1430488765e-3+y1*(-0.6911147651e-5
           +y1*(0.7621095161e-6+y1*(-0.934945152e-7))));
    ans = sqrt(0.636619772/x1)*(sin(xx)*ans1+z*cos(xx)*ans2);
  }
  return ans;
}

double bessi0(x_)
double x_;
{
  double ax,ans,y;

  if((ax = fabs(x_)) < 3.75) {
    y = x_/3.75;
    y *= y;
    ans = 1. + y*(3.5156229 + y*(3.0899424 + y*(1.2067492
          + y*(0.2659732 + y*(0.360768e-1 + y*0.45813e-2)))));
  } else {
      y = 3.75/ax;
      ans = (exp(x_)/sqrt(ax)) *(0.39894228 + y*(0.1328592e-1
            + y*(0.225319e-2 + y*(-0.157565e-2 + y*(0.916281e-2
            + y*(-0.2057706e-1 + y*(0.2635537e-1 + y*(-0.1647633e-1
            +y*0.392377e-2))))))));
  }
  return ans;
}

double bessj1(x1)
double x1;
{
  double ax,z,xx,y1,ans,ans1,ans2;

  if((ax=fabs(x1)) < 8.)
  {
    y1 = x1*x1;
    ans1 = x1*(72362614232.+y1*(-7895059235.+y1*(242396853.1
           +y1*(-2972611.439+y1*(15704.48260+y1*(-30.16036606))))));
    ans2 = 144725228442.+y1*(2300535178.+y1*(18583304.74
           +y1*(99447.43394+y1*(376.9991397+y1*1.))));
    ans = ans1/ans2;
  }
  else
  {
    z = 8./ax;
    y1 = z*z;
    xx = ax - 2.356194491;
    ans1 = 1.+y1*(0.183105e-2+y1*(-0.3516396496e-4
           +y1*(0.2457520174e-5+y1*(-0.240337019e-6))));
    ans2 = 0.04687499995+y1*(-0.2002690873e-3
           +y1*(0.8449199096e-5+y1*(-0.88228987e-6+y1*0.105787412e-6)));
    ans = sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    if(x1<0.) ans = -ans;
  }
  return ans;
}

double bessy1(x1)
double x1;
{
  double z,xx,y1,ans,ans1,ans2;

  if(x1 < 8.)
  {
    y1 = x1*x1;
    ans1 = x1*(-0.4900604943e13+y1*(0.1275274390e13+y1*(-0.5153438139e11
           +y1*(0.7349264551e9+y1*(-0.4237922726e7+y1*0.8511937935e4)))));
    ans2 = 0.2499580570e14+y1*(0.4244419664e12+y1*(0.3733650367e10
           +y1*(0.2245904002e8+y1*(0.1020426050e6+y1*(0.3549632885e3+y1)))));
    ans = ans1/ans2 + 0.636619772*(bessj1(x1)*log(x1)-1./x1);
  }
```

```
  else
  {
    z = 8./x1;
    y1 = z*z;
    xx = x1 - 2.356194491;
    ans1 = 1.+y1*(0.183105e-2+y1*(-0.3516396496e-4+y1*(0.2457520174e-5
           +y1*(-0.240337019e-6))));
    ans2 = 0.04687499995+y1*(-0.2002690873e-3+y1*(0.8449199096e-5
           +y1*(-0.88228987e-6+y1*0.105787412e-6)));
    ans = sqrt(0.636619772/x1)*(sin(xx)*ans1+z*cos(xx)*ans2);
  }
  return ans;
}

double bessy_(n,x1)
int n; double x1;
{
  int i;
  double by,bym,byp,tox;

  if(n > 1)
  {
    tox = 2./x1;
    by = bessy1(x1);
    bym = bessy0(x1);
    for(i=1;i<n;i++)
    {
      byp = i*tox*by - bym;
      bym = by;
      by = byp;
    }
  }
  return by;
}

double bessj_(n,x1)
int n; double x1;
{
  int i,jsum,m;
  double ax,bj,bjm,bjp,sum,tox,ans;

  ax = fabs(x1);
  if(ax == 0.) return 0.;
  else if(ax > (double)n)
  {
    tox = 2./ax;
    bjm = bessj0(ax);
    bj = bessj1(ax);
    for(i=1;i<n;i++)
    {
      bjp = i*tox*bj - bjm;
      bjm = bj;
      bj = bjp;
    }
    ans = bj;
  }
  else
  {
    tox = 2./ax;
    m = 2*((n+(int)sqrt(ACC*n))/2);
    jsum = 0;
    bjp = 0.; sum = 0.; ans = 0.;
    bj = 1.;
    for(i=m;i>0;i--)
    {
      bjm = i*tox*bj - bjp;
      bjp = bj;
      bj = bjm;
      if(fabs(bj) > BIGNO)
      {
        bj *= BIGNI;
        bjp *= BIGNI;
        ans *= BIGNI;
        sum *= BIGNI;
      }
      if(jsum) sum += bj;
      jsum =! jsum;
      if(i == n) ans = bjp;
    }
    sum = 2.*sum -bj;
    ans /= sum;
  }
  return x1<0. && n%2 == 1 ? -ans : ans;
}

double bessj(n,x1)
double x1;
{
  double temp;
  float t;

  if(n == 0) return bessj0(x1);
  if(abs(n) == 1) { if(n > 0) return bessj1(x1); else return -bessj1(x1);}

  temp = bessj_(abs(n),x1); if(n >= 0) return temp;
  if(abs(n) % 2 == 0) t = 1.; else t = -1.;
  return t*temp;
}

double bessy(n,x1)
double x1;
{
  double temp;
  float t;

  if(n == 0) return bessy0(x1);
  if(abs(n) == 1) { if(n > 0) return bessy1(x1); else return -bessy1(x1);}

  temp = bessy_(abs(n),x1); if(n >= 0) return temp;
  if(abs(n) % 2 == 0) t = 1.; else t = -1.;
  return t*temp;
}

fcomplex H1(n,x1)
double x1;
{
  fcomplex temp;

  return Complex(bessj(n,x1),bessy(n,x1));
//  temp = Complex(bessj(abs(n),x1),bessy(abs(n),x1));
//  if(n >= 0) return temp;
//  else return Cmul(Cexp(Complex(0.,abs(n)*pi)),temp);
}

fcomplex H2(n,x1)
double x1;
{
  fcomplex temp;

  return Complex(bessj(n,x1),-bessy(n,x1));
```

```
//  temp = Complex(bessj(abs(n),x1),-bessy(abs(n),x1));
//  if(n >= 0) return temp;
//  else return Cmul(Cexp(Complex(0.,-abs(n)*pi)),temp);
}

float gammln(xx)
float xx;
{
  float x1,tmp,ser;
  static float cof[6] = {76.18009173,-86.50532033,24.01409822,
                         -1.231739516,0.120858003e-2,-0.536382e-5};
  int i;

  x1 = xx-1.;
  tmp = x1+5.5;
  tmp -= (x1+0.5)*log(tmp);
  ser = 1.;
  for(i=0;i<=5;i++)
  {
    x1 += 1.;
    ser += cof[i]/x1;
  }
  return -tmp+log(2.50662827465*ser);
}

double factrl(n)
{
  static int ntop = 4;
  static double a1[33] = {1.,1.,2.,6.,24.};
  int j;

  if(n < 0) { printf("factorial error : n < 0\n"); exit(0); }
  if(n > 32) return exp(gammln(n+1.));
  while(ntop < n)
  {
    j = ntop++;
    a1[ntop] = a1[j]*ntop;
  }
  return a1[n];
}

void gser(gamser,a,x,gln)
float a,x,*gamser,*gln;
{
  int n;
  float sum,del,ap;

  *gln=gammln(a);
  if(x <= 0.) {
    if(x < 0.) {printf("error1\n"); exit(0);}
    *gamser = 0.;
    return;
  } else {
      ap = a;
      del = sum = 1./a;
      for(n=1;n<=ITMAX;n++) {
        ap += 1.;
        del *= x/ap;
        sum += del;
        if(fabs(del) < fabs(sum)*EPS) {
          *gamser = sum*exp(-x+a*log(x)-(*gln));
          return;
        }
    }
  printf("a too large, ITMAX too small\n");
  return;
  }
}

void gcf(gammcf,a,x,gln)
float a,x,*gammcf,*gln;
{
  int n;
  float gold=0.,g,fac=1.,b1=1.;
  float b0=0.,anf,ana,an,a1,a0=1.;
  float gammln();

  *gln = gammln(a);
  a1 = x;
  for(n=1;n<=ITMAX;n++) {
    an = (float)n;
    ana = an - a;
    a0 = (a1+a0*ana)*fac;
    b0 = (b1+b0*ana)*fac;
    anf = an*fac;
    a1 = x*a0+anf*a1;
    b1 = x*b0+anf*b1;
    if(a1) {
      fac = 1./a1;
      g = b1*fac;
      if(fabs((g-gold)/g) < EPS) {
        *gammcf = exp(-x+a*log(x) - (*gln))*g;
        return;
      }
      gold = g;
    }
  }
  printf("a too large, ITMAX too small\n"); exit(0);
}

float gammq(a,x)
float a,x;
{
  float gamser,gammcf,gln;
  void gcf(),gser();

  if(x < 0. || a <= 0.) {printf("error2\n"); return 0.;}
  if(x < (a+1.)) {
    gser(&gamser,a,x,&gln);
    return 1. - gamser;
  } else {
    gcf(&gammcf,a,x,&gln);
    return gammcf;
  }
}

float gammp(a,x)
float a,x;
{
  float gamser,gammcf,gln;

  if(x < 0. || a <= 0.) {printf("error3\n"); exit(0);}
  if(x < (a+1.)) {
    gser(&gamser,a,x,&gln);
    return gamser;
  } else {
    gcf(&gammcf,a,x,&gln);
    return 1. - gammcf;
  }
}
```

```
double erf(x_)
double x_;
{
  return x_ < 0. ? -gammp(0.5,x_*x_) : gammp(0.5,x_*x_);
}

double erfcc(x_)  // Complementary error function
double x_;
{
  double t_,z_,ans_;

  z_ = fabs(x_);
  t_ = 1./(1.+0.5*z_);
  ans_ = t_*exp(-z_*z_-1.26551223+t_*(1.00002368+t_*(0.37409196+t_*(0.09678418+
             t_*(-0.18628808+t_*(0.27886807+t_*(-1.13520398+t_*(1.48851587+
             t_*(-0.82215223+t_*0.17087277)))))))));
  return x_ >= 0. ? ans_ : 2. - ans_;
}

#define j Complex(0.,1.)
```

```
// One dipole or array simulation in forest enviroment


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

//Including routines for complex arithmetic
#include "complex.h"

// Containg constant
#include "constant_em.h"

// Routine for spacial function
#include "specialfun.h"

// Routine for integration
#include "calculus.h"

// Routine for CG type matrix solver
#include "CG_solver.h"

// Routine for reflection coefficient
#include "em_tool.h"

// Routine for random number reneration
#include "random_tool.h"

// Routine for opening file
#include "file_handler.h"

// Routine for malloc application
#include "my_malloc.h"

// Subfunction for calculation of impedance matrix element (2-D case)
#include "mom_tool.h"

// Subroutine for calculation of electric field generated by cylinder
#include "near_field.h"

#define dim 2
#define zero_error 1.e-20

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif


// Complex zero
#ifndef zero
  #define zero Complex(0.,0.)
#endif


// complex number of j
#ifndef j
  #define j Complex(0.,1.)
#endif


// Complex number of one
#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

FILE *f2,*f3,*f4;
char solver_type[10];
unsigned short int N;
unsigned int mN,N1,i_N,S_N,S_N1,S_N2,maxiter,pre_cond,int_N,a_N;
double kr,k0,y_0,z0,lambda,i_ang;
double *ww,*xx;
double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
float f0,w0,in_a,j_error,ra,max_rho,delta,a_x,a_y,a_z,min_gap,y_length;
float x_length,density,dy,f_L,s_L,s_v,x_c,y_c,z_c,lx,ly,lz,error_it;
float *cx,*cy,**array_posi,**array_ori,*array_pha;
float *height_s,**center,delta_x,delta_y;
fcomplex sin_wb,cos_wb,Z1,im_field[3];
fcomplex k,k1,kg,kz,e1,eg,pre_const,pre_const1,pre_const_xy,pre_const_xy1;
fcomplex *mat,*F,**m;
fcomplex *J_dis,*P,**PP;

// The below header file needs some of global variable


// Calculation of incident field
#include "incident_field.h"

// Calculation of reflected electric field generated by cylinder from ground
#include "reflected_wave.h"

// Generating cylinder location & mesh one cylinder
#include "mesh_tool1.h"

// Routine for reading input files
#include "read_conf.h"


// Dielectric constant for cylinder
fcomplex er_ft(x,y)
double x,y;
{
  return Complex(5.,1.);
}


// allow memory for location cylinder and height
void allow_mesh_memory()
{
  unsigned int i,n;

  height_s = (float *)malloc(sizeof(float)*S_N1);
  if(height_s == NULL) {printf("malloc error : height_s\n"); exit(0);}

  cx = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cx == NULL) {printf("malloc error : cx\n"); exit(0);}

  cy = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cy == NULL) {printf("malloc error : cy\n"); exit(0);}

  n = 3*N*(S_N1+S_N2);
  center = (float **)malloc(sizeof(float *)*n);
  if(center == NULL) {printf("malloc error : center\n"); exit(0);}
  for(i=0;i<n;i++)
    center[i] = malloc_f_1(dim);
}
```

```
// allow main memory
void allow_memory()
{
  unsigned int i,ii,nn;
  int n;

  allow_mesh_memory();

  n = mN*(mN+1)/2;

  mat = (fcomplex *)malloc(sizeof(fcomplex)*n);
  if(mat == NULL) {printf("malloc error : mat\n"); exit(0);}

  F = malloc_c_1(mN);
  J_dis = malloc_c_1(mN);

  if(strncmp(solver_type,"BCG",3) == 0) P = malloc_c_1(mN);
  else
  {
    if(strncmp(solver_type,"BICGSTAB_s",10) == 0) nn = 7;
    else nn = 4;

    PP = (fcomplex **)malloc(sizeof(fcomplex *)*nn);
    if(PP == NULL) {printf("malloc error : PP\n"); exit(0);}
    for(i=0;i<nn;i++)
      PP[i] = malloc_c_1(mN);
  }
}


// generating impedance matrix
void build_matrix(kr,kz,n)
double kr; fcomplex kz;
{
  unsigned int i,ii,jj,a_,b,r_i,r_ii;
  unsigned int a=0;
  fcomplex temp,er_m,t1;
  double x,y,xn,yn,d_x,d_y,nN;

  temp = Complex(0.,1./4.);
  d_x = delta_x;
  d_y = delta_y;

  for(i=0;i<mN;i++)
  {
    x = center[i + 3*N*n*S_N1][0];
    y = center[i + 3*N*n*S_N1][1];
    er_m = Csub(er_ft(x,y),Complex(1.,0.));
    er_m = Cmul(temp,er_m);
    a_ = (int)((double)i/((double)N*3.));
    r_i = i - a_*3*N;

    for(ii=i;ii<mN;ii++)
    {
      xn = center[ii + 3*N*n*S_N1][0];
      yn = center[ii + 3*N*n*S_N1][1];
      b = (int)((double)ii/((double)N*3.));
      r_ii = ii - b*3*N;

      if(a_ == b)
      {
        if(r_i < N) {
          if(r_ii < N) {
            if(r_i == r_ii)
              mat[a] = Csub(Cmul(er_m,Inn_xx(x,y,xn,yn,d_x,d_y,kr,kz)),Cone);
            else mat[a] = Cmul(er_m,In_xx(x,y,xn,yn,d_x,d_y,kr,kz));
                }
          else if(r_ii < 2*N) {
            if(r_i == (r_ii - N)) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
              }
          else {
            if(r_i == (r_ii - 2*N)) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
              }
        }
        else if(r_i < 2*N) {
          if(r_ii < N) {
            if((r_i - N) == r_ii) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
                }
          else if(r_ii < 2*N) {
            if((r_i - N) == (r_ii - N))
              mat[a] = Csub(Cmul(er_m,Inn_yy(x,y,xn,yn,d_x,d_y,kr,kz)),Cone);
            else mat[a] = Cmul(er_m,In_yy(x,y,xn,yn,d_x,d_y,kr,kz));
              }
          else {
            if((r_i -N) == (r_ii - 2*N)) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
              }
        }
        else {
          if(r_ii < N) {
            if((r_i - 2*N) == r_ii) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
                }
          else if(r_ii < 2*N) {
            if((r_i - 2*N) == (r_ii - N)) mat[a] = zero;
            else mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
              }
          else {
            if((r_i - 2*N) == (r_ii - 2*N))
              mat[a] = Csub(Cmul(er_m,Inn(x,y,xn,yn,d_x,d_y,kr)),Cone);
            else mat[a] = Cmul(er_m,In(x,y,xn,yn,d_x,d_y,kr));
              }
           }
      }
      else
      {
        if(r_i < N) {
          if(r_ii < N)
            mat[a] = Cmul(er_m,In_xx(x,y,xn,yn,d_x,d_y,kr,kz));
          else if(r_ii < 2*N)
            mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
          else mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
        }
        else if(r_i < 2*N) {
          if(r_ii < N)
            mat[a] = Cmul(er_m,In_xy(x,y,xn,yn,d_x,d_y,kr));
          else if(r_ii < 2*N)
            mat[a] = Cmul(er_m,In_yy(x,y,xn,yn,d_x,d_y,kr,kz));
          else mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
        }
        else {
          if(r_ii < N)
            mat[a] = Cmul(er_m,In_xz(x,y,xn,yn,d_x,d_y,kr,kz));
          else if(r_ii < 2*N)
            mat[a] = Cmul(er_m,In_yz(x,y,xn,yn,d_x,d_y,kr,kz));
          else mat[a] = Cmul(er_m,In(x,y,xn,yn,d_x,d_y,kr));
```

```
          }
      }
        a++;
    }
  }
}


// generating current vector
void build_vector(n,n1)
{
  fcomplex er_m,temp,temp1;
  int jj,ii;
  double x,y;
  fcomplex Ex,Ey,Ez;

  temp = RCmul(k0*y_0,Cmul(e1,j));

  for(jj=0;jj<S_N;jj++)
    for(ii=0;ii<N;ii++)
       {
      x = center[ii + 3*jj*N + 3*N*n1*S_N1][0];
      y = center[ii + 3*jj*N + 3*N*n1*S_N1][1];
      er_m = Csub(er_ft(x,y),Complex(1.,0.));
      er_m = Cmul(temp,er_m);
      if(n1 == 0)
      {Ex = i_field_x(x,y); Ey = i_field_y(x,y); Ez = i_field_z(x,y);}
      else {
        Ex = i_field_x1(x,y); Ey = i_field_y1(x,y); Ez = i_field_z1(x,y);
         }
      F[ii+3*jj*N] = Cmul(er_m,Ex);
      F[ii+N+3*jj*N] = Cmul(er_m,Ey);
      F[ii+2*N+3*jj*N] = Cmul(er_m,Ez);
        }
}


fcomplex Z(i,k)

// i & k are index for impedance matrix
// return element of impedance matrix at (i,k)

{
  int n,a_,a1_,b_,b1_;

  if(k >= i) n = mN*i - (i-1)*i/2 + k - i;
  else n = mN*k - (k-1)*k/2 + i - k;

  a_ = (int)((double)i/((double)N*3.));
  b_ = (int)((double)k/((double)N*3.));
  a1_ = (int)((double)(i - a_*3*N)/((double)N));
  b1_ = (int)((double)(k - b_*3*N)/((double)N));

  if(k >= i) return mat[n];
  else {
    if(a1_ != 2) {
      if(b1_!= 2) return mat[n];
          else return RCmul(-1.,mat[n]);
        }
    else if(b1_ == 2) return mat[n];
          else return RCmul(-1.,mat[n]);
  }
}
```

```
// Solving matrix equation using iterative method
void obtain_F_vector()
{
  Initial_guess(F,J_dis,mN);

  if(strncmp(solver_type,"BCG",3) == 0)
    BCG(Z,F,J_dis,P,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"CGS",3) == 0)
    CGS(Z,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB",20) == 0)
    BICGSTAB(Z,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB_s",20) == 0)
    BICGSTAB_s(Z,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else {printf("Invaild solver type...\n"); exit(0);}
}


// allow memory for location of array & orientaion of array
void allow_memory_for_array(n)
{
  int i;

  // position of array
  array_posi = (float **)malloc(sizeof(float *)*n);
  if(array_posi == NULL) {printf("malloc error : array_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    array_posi[i] = malloc_f_1(3);

  //orientation of array
  array_ori = (float **)malloc(sizeof(float *)*n);
  if(array_ori == NULL) {printf("malloc error : array_ori\n"); exit(0);}
  for(i=0;i<n;i++)
    array_ori[i] = malloc_f_1(3);

  // Phase differnce of each antenna with respect to the first antenna
  array_pha = malloc_f_1(n);
}


// read input files
int read_conf()
{
  FILE *f1;
  unsigned short int flag = 0;
  unsigned int i;
  double norm_1;
  double tmp1,tmp2,tmp3;
  fcomplex temp;
  char *file_name,*w_r;


  read_conf_array();

  allow_memory_for_array(a_N);

  real_array_info();

  if(S_N1 < S_N2)
  {
    printf("The # of scatterer around source must ");
    printf("be lager than the receiver!!\n");
    exit(0);
  }

  for(i=0;i<a_N;i++)
```

```
  {
    flag = 0;
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];
    norm_l = sqrt(sqr(lx) + sqr(ly) + sqr(lz));
    lx /= norm_l; ly /= norm_l; lz /= norm_l;
    array_ori[i][0] = lx; array_ori[i][1] = ly; array_ori[i][2] = lz;

    if(lx !=0) if(ly !=0 || lz != 0) flag = 1;
    if(ly !=0) if(lx !=0 || lz != 0) flag = 1;
    if(lz !=0) if(lx !=0 || ly != 0) flag = 1;
    if(flag != 0) {printf("Dipole must have one component!!\n"); exit(0);}

  }

  printf("er_r = %lf\n",er_r);

  w0 = 2.*pi*f0;
  k0 = w0*sqrt(e0*u0); kg = RCmul(k0,Csqrt(eg));
  y_0 = sqrt(e0/u0); z0 = 1./y_0;
  e1 = Complex(er_r,er_i+sigma1/(w0*e0));
  eg = Complex(eg_r,eg_i+sigma2/(w0*e0));
  k1 = RCmul(k0,Csqrt(e1));
  k = Cdiv(Cone,e1);
  Z1 = Cdiv(Complex(z0,0.),Csqrt(e1));
  lambda = 2.*pi/k0;

  s_v = sqrt(s_v);     // transform variance to standard deviation

  temp = er_ft(0.,0.);
  printf("Frequency = %f[MHz], radius of trunk = %f[m]  ",f0/1.e6,ra);
  printf("Dielectric constant of trunk = %lf + j %lf\n",temp.r,temp.i);
  printf("Forest heigh = %f & ",f_L);
  printf("effective dielectric constant = %lf + j%lf\n",e1.r,e1.i);
  printf("Dielectric constant of ground = %lf + j%lf\n",eg.r,eg.i);
  printf("ObsErvation point : (%f,%f,%f)\n",x_c,y_c,z_c);

  printf("There are %d antennas.\n",a_N);
  printf("Antenna Position : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_posi[i][0],array_posi[i][1],array_posi[i][2]);

  printf("Antenna orientation : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_ori[i][0],array_ori[i][1],array_ori[i][2]);

  printf("Phase differnce with reference to the 1st antenna : \n");
  for(i=0;i<a_N;i++)
  {
    printf("%f[Degrees]\n",array_pha[i]);
    array_pha[i] *= pi/180.;
  }

  if(pre_cond == 0)
  printf("%s without pre-conditioner is used for matrix solver.\n",solver_type);
  else
    printf("%s with pre-conditioner is used for matrix solver.\n",solver_type);
  printf("Integral point for z-axis is %d\n",int_N);

  S_N = S_N1;
  a_x = x_c; a_y = y_c; a_z = z_c;
  // Exchage Tx location and Rx for applying the reciprocity thm.
  return a_N;
}
```

```
// allow memory for coefficient for Gauss quadrature
void first_memory()
{
  xx = (double *)malloc(sizeof(double)*(int_N+1));
  if(xx == NULL) {printf("malloc error : xx\n"); exit(0);}

  ww = (double *)malloc(sizeof(double)*(int_N+1));
  if(ww == NULL) {printf("malloc error : ww\n"); exit(0);}
}


// Initialize some data
int initialize_data()
{
  a_N = read_conf();
  first_memory();
  gauleg(0.,1.,xx,ww,int_N);

  sin_wb = Csqrt(k); cos_wb = Csqrt(Csub(Cone,k));
  return a_N;
}


// Calculation of electric field near observation point & save
void save_result(n,n1,n2)
{
  unsigned int i,ii;
  float idum = (-1.);
  double x,y,z,L,r_ang;
  fcomplex msf_s_x,msf_s_y,msf_s_z,i_mag_x,i_mag_y,i_mag_z,temp,temp1;

//  x = x_c; y = y_c; z = z_c;
  x = a_x; y = a_y; z = a_z;
  if(n1 != 0) for(ii=0;ii<n1*S_N2;ii++) L = gasdev(&idum);
  else for(ii=0;ii<S_N1;ii++) L = gasdev(&idum);

  msf_s_x = zero; msf_s_y = zero; msf_s_z = zero;

  for(ii=0;ii<n;ii++)  // Summation over all elements
  {
    L = gasdev(&idum);  // Gauss Deviates with zero mean & 1. variance of 1
    L = s_v*L + s_L;    // Transformation

    temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);    // Direct scattered field
    temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L); // Reflected scattered field
    msf_s_x = Cadd(msf_s_x,Cadd(temp,temp1));

    temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
    temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
    msf_s_y = Cadd(msf_s_y,Cadd(temp,temp1));

    temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    msf_s_z = Cadd(msf_s_z,Cadd(temp,temp1));
  }

  msf_s_x = Cadd(msf_s_x,im_field[0]);
  msf_s_y = Cadd(msf_s_y,im_field[1]);
  msf_s_z = Cadd(msf_s_z,im_field[2]);

  fprintf(f2,"%30.28lf %30.28lf\n",msf_s_x.r,msf_s_x.i);
  fprintf(f3,"%30.28lf %30.28lf\n",msf_s_y.r,msf_s_y.i);
  fprintf(f4,"%30.28lf %30.28lf\n",msf_s_z.r,msf_s_z.i);
}
```

```c
// Calculation of electric field from cylinders near source
fcomplex first_field(n,n1,n2,n3)
{
  unsigned int ii,i;
  float idum = (-1.);
  double L,r_ang,x,y,z;
  fcomplex ans,sum,temp,temp1;

  if(n1 != 0) for(ii=0;ii<n1*S_N1;ii++) L = gasdev(&idum);

  ans = zero;

  for(i=0;i<n3;i++)
  {
    sum = zero;
    x = array_posi[i][0]; y = array_posi[i][1]; z = array_posi[i][2];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    for(ii=0;ii<S_N;ii++)
    {
      if((n == 0) && (i == 0))
      {
        L = gasdev(&idum); // Gauss Deviates with zero mean & variance of 1
        L = s_v*L + s_L;// Transformation to deviates with L mean & s_v variance
        height_s[ii] = L;
      }
      else L = height_s[ii];

      if(lx != 0.)
      {
        temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(ly != 0.)
      {
        temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(lz != 0.)
      {
        temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
      }
      sum = Cadd(sum,Cadd(temp,temp1));
    }
    ans = Cadd(ans,Cmul(sum,Cexp(Complex(0.,array_pha[i]))));
  }
  return ans;
}

void main()
{
  unsigned short int flag_x,flag_y,flag_z;
  unsigned int i,ii,iii,a,a1,n = 0;
  double temp;
  float a_c_x = 0,a_c_y = 0.,m_r_x = 0.,m_r_y = 0.;
  char *file_name,*w_r,*st;
  char fn[20];
  FILE *f1,*f6;

  printf("The program starts!!!\n");

  // output files
  f2 = file_open("dipole_x_p_my.dat","w");
  f3 = file_open("dipole_y_p_my.dat","w");
  f4 = file_open("dipole_z_p_my.dat","w");

  a_N = initialize_data(); S_N = S_N1;
  printf("Initializing data is completed....\n");
  printf("# of scatterers is %d\n",S_N1+S_N2);
  N = determine_N(); printf("Number of element in a cylinder  = %d\n",N);
  mN = 3*N*S_N;
  kz = Cmul(k1,cos_wb); kr = k0; i_ang = acos(kz.r/k1.r);
  printf("k0 = %lf kr = %lf kz = %lf + j%lf\n",k0,kr,kz.r,kz.i);

  for(i=0;i<a_N;i++)
  { a_c_x += array_posi[i][0]; a_c_y += array_posi[i][1];}
  a_c_x /= (float)a_N; a_c_y /= (float)a_N;

  for(i=0;i<a_N;i++)
  {
    temp = fabs(a_c_x - array_posi[i][0]);
    m_r_x = (temp > m_r_x) ? temp : m_r_x;

    temp = fabs(a_c_y - array_posi[i][1]);
    m_r_y = (temp > m_r_y) ? temp : m_r_y;
  }

  if(a_N > 1)
  {
    if(m_r_x != 0.) m_r_x *= 2.;
    else if(m_r_y != 0.) m_r_x = m_r_y/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
    if(m_r_y != 0.) m_r_y *= 2.;
    else if(m_r_x != 0.) m_r_y = m_r_x/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
  }

  allow_memory(); printf("Memory is allowed...\n");

  for(ii=0;ii<i_N;ii++)
  {
    printf("Iteration Number : %d \n",ii);
    for(a=0;a<2;a++)
    {
      // a == 0: calculation of electric field from cylinders near source
      // a == 1: calculation of electric field from cylinders near receiver

      S_N = (a == 0) ? S_N1 : S_N2;
      mN = 3*N*S_N;  // total number of elements

      // generating cylinder location
      if(a_N == 1)
      {
        if(a == 0) generate_mesh_array_cir(2*ii,a,a_N,a_c_x,a_c_y,2.*lambda);
        else generate_mesh_array_cir(2*ii+a,a,a_N,a_x,a_y,2.*lambda);
      }
      else
        if(a == 0) generate_mesh_array_cir(2*ii,a,a_N,a_c_x,a_c_y,2.*lambda);
        else generate_mesh_array_cir(2*ii+a,a,a_N,a_x,a_y,2.*lambda);

      printf("generating mesh!!!");

      // decretizing each cylinder
      for(i=0;i<S_N;i++)
        mesh(cx[i+a*S_N1],cy[i+a*S_N1],i,a);

      if((ii == 0) && (a == 0))
```

```
    {
      if(fabs(delta_x-delta_y) < 1.e-6) {printf("Homogeneous Mesh...\n");}
      else { printf("Error : delta(x) != delta(y)...\n");
             printf("delta_x = %f delta_y = %f\n",delta_x,delta_y); exit(0);}

      temp = sqr(delta_x*kr)*(1. - sqr(kr*delta_x)/24.);
      pre_const = RCmul(temp,Csub(er_ft(0.,0.),Complex(1.,0.)));
      pre_const = Cmul(pre_const,Complex(0.,1./4.));
      pre_const1 = Cmul(RCmul(sqrt(1./(pi*kr)),pre_const),Complex(1.,-1.));
      pre_const = RCmul(0.5,pre_const);

      pre_const_xy = Csub(er_ft(0.,0.),Complex(1.,0.));
      pre_const_xy = Cmul(pre_const_xy,Complex(0.,sqr(kr*delta_x)/4.));
      pre_const_xy1 = RCmul(-sqrt(1./(pi*kr)),pre_const_xy);
      pre_const_xy1 = Cmul(pre_const_xy1,Complex(1.,-1.));
    }

    build_matrix(kr,kz,a);

    if(a == 0)
      for(i=0;i<3;i++)
        {

        // i = 0: Calculation of x-comp. of electric field
        // i = 1: Calculation of y-comp. of electric field
                // i = 2: Calculation of z-comp. of electric field

        if(i == 0) {lx = 1.; ly = 0.; lz = 0.;}
        else if(i == 1) {lx = 0.; ly = 1.; lz = 0.;}
             else {lx = 0.; ly = 0.; lz = 1.;}

        build_vector(S_N*N,a);

        if(i == 0) printf("%dth iteration : x-component\n",ii);
        else if(i == 1) printf("%dth iteration : y-component\n",ii);
             else printf("%dth iteration : z-component\n",ii);

        obtain_F_vector();
        im_field[i] = first_field(i,ii,a,a_N);
              }
    else {
     printf("Considering the near scatterers...\n");
     build_vector(S_N*N,a);
     obtain_F_vector();
     save_result(S_N,ii,a);
     printf("Save is completed...\n");
         }
  }
}

fclose(f2); fclose(f3); fclose(f4);
printf("The program ends!!!\n");

}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

#include "complex.h"
#include "my_malloc.h"
#include "file_handler.h"

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

void main()
{
  const int N = 50,l_n = 9,rN = 4;
  unsigned short int flag[N];
  int i,ii,tN;
  char *file_name,*w_r,st[10];
  char fn[20],test_c[100];
  FILE *f1,*f2,*f3,*f4,*f5;
  double *var_x,*var_y,*var_z;
  fcomplex *mean_x,*mean_y,*mean_z;
  fcomplex *ans_x,*ans_y,*ans_z;
  fcomplex **data_x,**data_y,**data_z;

  f5 = file_open("field_z.dat","w");

  for(i=0;i<N;i++) flag[i] = 0;

  for(i=0;i<N;i++)
  {
    for(ii=1;ii<=l_n;ii++)
    {
      strcpy(fn,"s_corr"); sprintf(st,"%d",i);
      strcat(fn,st); sprintf(st,"%d",ii); strcat(fn,st);
      strcat(fn,".dat");
      if((f1 = fopen(fn,"r")) == NULL)
      {
        flag[i] = 1;  fclose(f1); break;
      }
      fclose(f1);
    }
  }

//  for(i=0;i<N;i++) printf("flag[%d] = %d\n",i,flag[i]);

  for(i=0;i<N;i++)
  {
    if(flag[i] == 0)
    {
      for(ii=1;ii<=l_n;ii++)
      {
        strcpy(fn,"s_corr"); sprintf(st,"%d",i);
        strcat(fn,st); sprintf(st,"%d",ii); strcat(fn,st);
        strcat(fn,".dat");
        f1 = fopen(fn,"r");
        fscanf(f1,"%s\n",&test_c);
        fclose(f1);
      }
//      printf("test_c = %s\n",test_c);
        if(strncmp(test_c,"NaN",10) == 0) flag[i] = 2;
    }
  }

  tN = 0;
  for(i=0;i<N;i++) if(flag[i] == 0) tN++;
  printf("tN = %d\n",tN);

  var_x = malloc_d_1(l_n+1);
  var_y = malloc_d_1(l_n+1);
  var_z = malloc_d_1(l_n+1);
  mean_x = malloc_c_1(l_n+1);
  mean_y = malloc_c_1(l_n+1);
  mean_z = malloc_c_1(l_n+1);
  ans_x = malloc_c_1(l_n+1);
  ans_y = malloc_c_1(l_n+1);
  ans_z = malloc_c_1(l_n+1);
  data_x = malloc_c_2(tN,l_n+1);
  data_y = malloc_c_2(tN,l_n+1);
  data_z = malloc_c_2(tN,l_n+1);

  tN = 0;
  for(i=0;i<N;i++)
  {
    if(flag[i] == 0)
    {
      for(ii=1;ii<=l_n;ii++)
      {
        strcpy(fn,"s_corr"); sprintf(st,"%d",i);
        strcat(fn,st); sprintf(st,"%d",ii); strcat(fn,st);
        strcat(fn,".dat");
        f1 = fopen(fn,"r");
        fscanf(f1,"%lf %lf\n",&data_x[tN][ii-1].r,&data_x[tN][ii-1].i);
        fscanf(f1,"%lf %lf\n",&data_y[tN][ii-1].r,&data_y[tN][ii-1].i);
        fscanf(f1,"%lf %lf\n",&data_z[tN][ii-1].r,&data_z[tN][ii-1].i);
        fclose(f1);

                fprintf(f5,"%lf %lf ",data_z[tN][ii-1].r,data_z[tN][ii-1].i);
      }
      tN++;
          fprintf(f5,"\n");
    }
  }

//  tN--;
  printf("File read is complete!!!!: tN = %d\n",tN);

  for(i=0;i<l_n;i++)
  {
    var_x[i] = var_y[i] = 0.; var_z[i] = 0.;
    mean_x[i] = zero; mean_y[i] = zero; mean_z[i] = zero;
    ans_x[i] = zero; ans_y[i] = zero; ans_z[i] = zero;
  }

  for(ii=0;ii<l_n;ii++)
    for(i=0;i<tN;i++)
    {
      mean_x[ii] = Cadd(mean_x[ii],data_x[i][ii]);
      mean_y[ii] = Cadd(mean_y[ii],data_y[i][ii]);
      mean_z[ii] = Cadd(mean_z[ii],data_z[i][ii]);

      var_x[ii] += sqr(Cabs(data_x[i][ii]));
      var_y[ii] += sqr(Cabs(data_y[i][ii]));
      var_z[ii] += sqr(Cabs(data_z[i][ii]));
```

```
    ans_x[ii] = Cadd(ans_x[ii],Cmul(data_x[i][ii],Conj(data_x[i][rN])));
    ans_y[ii] = Cadd(ans_y[ii],Cmul(data_y[i][ii],Conj(data_y[i][rN])));
    ans_z[ii] = Cadd(ans_z[ii],Cmul(data_z[i][ii],Conj(data_z[i][rN])));
   }

  printf("Mean & S.D are calculated!!!!\n");

  for(i=0;i<l_n;i++)
  {
    mean_x[i] = RCmul(1./(double)tN,mean_x[i]);
    mean_y[i] = RCmul(1./(double)tN,mean_y[i]);
    mean_z[i] = RCmul(1./(double)tN,mean_z[i]);

    var_x[i] /= (double)tN; var_y[i] /= (double)tN;
    var_z[i] /= (double)tN;

    var_x[i] -= sqr(Cabs(mean_x[i]));
    var_y[i] -= sqr(Cabs(mean_y[i]));
    var_z[i] -= sqr(Cabs(mean_z[i]));

    ans_x[i] = RCmul(1./(double)tN,ans_x[i]);
    ans_y[i] = RCmul(1./(double)tN,ans_y[i]);
    ans_z[i] = RCmul(1./(double)tN,ans_z[i]);
  }

  for(i=0;i<l_n;i++)
  {
    ans_x[i] = Csub(ans_x[i],Cmul(mean_x[i],Conj(mean_x[rN])));
    ans_x[i] = RCmul(1./sqrt(var_x[i]*var_x[rN]),ans_x[i]);

    ans_y[i] = Csub(ans_y[i],Cmul(mean_y[i],Conj(mean_y[rN])));
    ans_y[i] = RCmul(1./sqrt(var_y[i]*var_y[rN]),ans_y[i]);

    ans_z[i] = Csub(ans_z[i],Cmul(mean_z[i],Conj(mean_z[rN])));
    ans_z[i] = RCmul(1./sqrt(var_z[i]*var_z[rN]),ans_z[i]);
  }

  f2 = file_open("s_corr_mean.dat","w");
  f3 = file_open("s_corr_var.dat","w");
  f4 = file_open("s_corr.dat","w");

  for(i=0;i<l_n;i++)
  {
    fprintf(f2,"%20.18lf %20.18lf ",mean_x[i].r,mean_x[i].i);
    fprintf(f2,"%20.18lf %20.18lf ",mean_y[i].r,mean_y[i].i);
    fprintf(f2,"%20.18lf %20.18lf\n",mean_z[i].r,mean_z[i].i);

    fprintf(f3,"%20.18lf %20.18lf %20.18lf\n",var_x[i],var_y[i],var_z[i]);

    fprintf(f4,"%20.18lf %20.18lf ",ans_x[i].r,ans_x[i].i);
    fprintf(f4,"%20.18lf %20.18lf ",ans_y[i].r,ans_y[i].i);
    fprintf(f4,"%20.18lf %20.18lf\n",ans_z[i].r,ans_z[i].i);
  }
}
```

```c
//  Merging resulting files into one file for time domain analysis


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "file_handler.h"

void main()
{
  int i,n=0;
  double r_data,i_data;
  FILE *f1,*f2;
  char *file_name,*w_r,st[30];
  char fn[50];
  char c;

  printf("Program starts!!!\n");

  strcpy(fn,"frequency.dat"); w_r = "w";
  f2 =  file_open(fn,w_r);

  for(i=0;i<50;i++)
  {
    strcpy(fn,"frequency.dat");
    sprintf(st,"%d",i); strcat(fn,st); w_r = "r";

    if((f1 = fopen(fn,w_r)) != NULL)
    {
      if((c = getc(f1)) == EOF) printf("Incomplete: frequency.dat%d\n",i);
      else putc(c,f2);
      while((c = getc(f1)) != EOF)
        putc(c,f2);

      fclose(f1);
    }
  }
  fclose(f2);

  strcpy(fn,"fre_inh1.dat"); w_r = "w";
  f2 =  file_open(fn,w_r);

  for(i=0;i<50;i++)
  {
    strcpy(fn,"fre_inh1.dat");
    sprintf(st,"%d",i); strcat(fn,st); w_r = "r";

    if((f1 = fopen(fn,w_r)) != NULL)
    {
      if((c = getc(f1)) == EOF) printf("Incomplete: fre_inh1.dat%d\n",i);
      else putc(c,f2);
      while((c = getc(f1)) != EOF)
        putc(c,f2);
      fclose(f1);
    }
  }
  fclose(f2);

  strcpy(fn,"fre_inh2.dat"); w_r = "w";
  f2 =  file_open(fn,w_r);

  for(i=0;i<50;i++)
  {
    strcpy(fn,"fre_inh2.dat");
    sprintf(st,"%d",i); strcat(fn,st); w_r = "r";

    if((f1 = fopen(fn,w_r)) != NULL)
    {
      if((c = getc(f1)) == EOF) printf("Incomplete: fre_inh2.dat%d\n",i);
      else putc(c,f2);
      while((c = getc(f1)) != EOF)
        putc(c,f2);
      fclose(f1);
    }
  }
  fclose(f2);

  strcpy(fn,"incident_field_inh.dat"); w_r = "w";
  f2 =  file_open(fn,w_r);

  for(i=0;i<50;i++)
  {
    strcpy(fn,"incident_field_inh.dat");
    sprintf(st,"%d",i); strcat(fn,st); w_r = "r";

    if((f1 = fopen(fn,w_r)) != NULL)
    {
      if((c = getc(f1)) == EOF) printf("Incomplete: incident_field_inh.dat%d\n",i);
      else putc(c,f2);
      while((c = getc(f1)) != EOF)
        putc(c,f2);
      fclose(f1);
    }
  }
  fclose(f2);
}
```

mesh_s_corr.c


```c
// Generating 50 different tree locations for spatial correlation simulation

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

#include "complex.h"
#include "constant_em.h"
#include "random_tool.h"
#include "file_handler.h"
#include "my_malloc.h"

#define range 2.             // Normalized by lambda
#define spacing (1./2.)     // Normalized by lambda
#define dim 2
#define zero_error 1.e-20

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

const char direction = 'y';

FILE *f2,*f3,*f4;
char solver_type[10];
unsigned short int N;
unsigned int mN,N1,i_N,S_N,S_N1,S_N2,maxiter,pre_cond,int_N,a_N;
double kr,k0,y_0,z0,lambda,i_ang;
double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
float f0,w0,in_a,j_error,ra,max_rho,delta,a_x,a_y,a_z,min_gap,y_length;
float x_length,density,dy,f_L,s_L,s_v,x_c,y_c,z_c,lx,ly,lz,error_it;
float *cx,*cy,**array_posi,**array_ori,*array_pha,**obs_posi;
float *height_s,**center,delta_x,delta_y;

#include "read_conf.h"

void allow_mesh_memory()
{
  cx = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cx == NULL) {printf("malloc error : cx\n"); exit(0);}

  cy = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cy == NULL) {printf("malloc error : cy\n"); exit(0);}
}

void allow_memory_for_array(n)
{
  int i;

  array_posi = (float **)malloc(sizeof(float *)*n);
  if(array_posi == NULL) {printf("malloc error : array_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    array_posi[i] = malloc_f_1(3);

  array_ori = (float **)malloc(sizeof(float *)*n);
  if(array_ori == NULL) {printf("malloc error : array_ori\n"); exit(0);}
  for(i=0;i<n;i++)
    array_ori[i] = malloc_f_1(3);

  array_pha = malloc_f_1(n);
}

void read_conf()
{
  FILE *f1;
  unsigned short int flag = 0;
  unsigned int i;
  double norm_1;
  double tmp1,tmp2,tmp3;
  fcomplex temp;
  char *file_name,*w_r;

  read_conf_array();
  allow_memory_for_array(a_N); allow_mesh_memory();
  real_array_info();

  if(S_N1 < S_N2)
  {
    printf("The # of scatterer around source must ");
    printf("be lager than the receiver!!\n");
    exit(0);
  }

  w0 = 2.*pi*f0;
  k0 = w0*sqrt(e0*u0);
  lambda = 2.*pi/k0;

  //return a_N;
}

void initialize_data(n)
{
  int n1,i,ii;
  FILE *m_x,*m_y;

  read_conf();

  obs_posi = (float **)malloc(sizeof(float *)*n);
  if(obs_posi == NULL) {printf("malloc error : obs_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    obs_posi[i] = malloc_f_1(dim);

  if(direction == 'x')
  {
    for(i=0;i<n;i++)
    {
       obs_posi[i][0] = x_c + range*lambda - i*spacing*lambda;
       obs_posi[i][1] = y_c;
    }
  }
  else if(direction == 'y')
  {
    for(i=0;i<n;i++)
    {
       obs_posi[i][0] = x_c;
```

```
      obs_posi[i][1] = y_c + range*lambda - i*spacing*lambda;
    }
  }
  else {printf("Wrong Direction for observation points!!!\n"); exit(0);}

  m_x = file_open("mesh_x.dat","w");
  m_y = file_open("mesh_y.dat","w");
  for(i=0;i<n;i++)
  {
    fprintf(m_x,"%f\n",obs_posi[i][0]);
    fprintf(m_y,"%f\n",obs_posi[i][1]);
  }
  fclose(m_x); fclose(m_y);

//  return a_N;
}

void generate_mesh_array_cir(n,n1,n2,n3,x_center,y_center,radius)
float x_center,y_center,radius;
{
  unsigned int i,jj,flag,n_sc,c_n;
  float xc,yc,tmp,min_dis,min_dis1;
  float idum = (-1.);
  static float pre_x_length,s_radius;
  fcomplex temp;

  c_n = n1;
  n1 = (n1 <= 1) ? n1 : 1;

  if(n != 0)
    for(i=0;i<n;i++)
    {
      xc = rand0(&idum);
      yc = rand0(&idum);
    }
  if(c_n > 1) radius = s_radius;

  min_dis = (min_gap*lambda < ra) ? 2.*ra: min_gap*lambda;
  x_length = radius;
  for(jj=0;jj<S_N;jj++)
  {
    if(c_n <= 1) {
      n_sc = (int)(density*2.*pi*(sqr(x_length) - sqr(x_length - radius)));
        if(n_sc == 0) n_sc++;
      if((jj % n_sc) == 0 && jj != 0) x_length += radius;
    }

    if(c_n > 1) {
      if(sqrt(sqr(x_center-cx[jj+n1*S_N1]) + sqr(y_center-cy[jj+n1*S_N1]))>radius)
        flag = 1;
      }
    else flag = 1;

      if(flag == 1) {
      do{
        flag = 0;
        if(x_length > 2.*radius) xc = (rand0(&idum) - 1.)*2.*radius+x_length;
        else xc = rand0(&idum)*radius;
        yc = 2.*pi*rand0(&idum);
        temp = RCmul(xc,Cexp(Complex(0.,yc)));
        xc = temp.r; yc = temp.i;
        xc += x_center; yc += y_center;

        for(i=0;i<jj;i++)
          if((sqrt(sqr(cx[i+n1*S_N1]-xc)+sqr(cy[i+n1*S_N1]-yc))< min_dis))
            flag = 1;
        if(flag == 0)
        {
          for(i=0;i<n2;i++)
          {
            tmp = sqrt(sqr(xc - array_posi[i][0]) + sqr(yc - array_posi[i][1]));
            if(tmp < min_dis) { flag = 2; break;}
          }
          for(i=0;i<n3;i++)
          {
            tmp = sqrt(sqr(xc - obs_posi[i][0]) + sqr(yc - obs_posi[i][1]));
            if(tmp < min_dis) { flag = 3; break;}
          }
        }
      } while(flag != 0);

      cx[jj+n1*S_N1] = xc; cy[jj+n1*S_N1] = yc;
      }
  }
  if(n == 0) pre_x_length = x_length;
  if(c_n == 1) s_radius = x_length;
}

void main()
{
  unsigned short int flag_x,flag_y,flag_z;
  unsigned int s_n,l_n,i,i1,ii,ii1,iii,a,a1,a2,a3,n = 0,n1;
  double e_s,e_v;
  float x,y,a_c_x = 0,a_c_y = 0.,temp;
  char *file_name,*w_r,*st;
  char fn[20],app[10];
  FILE *f1,*f5,*f6;

  printf("The program starts!!!\n");

  n1 = 2*(int)(range/spacing); n1++;
  printf("n1 = %d\n",n1);

  initialize_data(n1); S_N = S_N1;

  for(i=0;i<a_N;i++)
  {a_c_x += array_posi[i][0]; a_c_y += array_posi[i][1];}
  a_c_x /= (float)a_N; a_c_y /= (float)a_N;

  for(ii=0;ii<50;ii++)
  {
    generate_mesh_array_cir(ii,0,a_N,n1,a_c_x,a_c_y,2*lambda);

    for(a=1;a<=n1;a++)
       {
      a2 = (a <= 1) ? a : 1;
      S_N = (a == 0) ? S_N1 : S_N2;

      x_c = obs_posi[a-1][0]; y_c = obs_posi[a-1][1];

      generate_mesh_array_cir(ii,a,a_N,n1,x_c,y_c,2*lambda);

      sprintf(app,"%d",ii);
      strcpy(fn,"mesh_s_corr");
      strcat(fn,app); sprintf(app,"%d",a); strcat(fn,app);
      strcat(fn,".dat");
      f1 = file_open(fn,"w");

      for(i=0;i<S_N1+S_N2;i++)
        fprintf(f1,"%f %f\n",cx[i],cy[i]);
```

```
      }
  }

  printf("The program ends!!!\n");

}
```

```c
/*
  Time domain analysis with inhomogeneous sampling points
  in frequency domain
  Calculation of Fourier Transform with Gaussian Quadrature alogorithm
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

#include "complex.h"
#include "constant_em.h"
#include "specialfun.h"
#include "calculus.h"
#include "CG_solver.h"
#include "em_tool.h"
#include "random_tool.h"
#include "file_handler.h"
#include "my_malloc.h"
#include "mom_tool.h"
#include "near_field.h"

#define dim 2
#define zero_error 1.e-20

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef distance
  #define distance(x,x1,y,y1,z,z1) sqrt(sqr(x - x1)+sqr(y - y1)+sqr(z - z1))
#endif

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

char solver_type[20];
unsigned short int N;
unsigned int mN,N1,i_N,S_N,S_N1,S_N2,maxiter,pre_cond,int_N,a_N;
double kr,k0,y_0,z0,lambda,i_ang;
double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
float f0,w0,in_a,j_error,ra,max_rho,delta,a_x,a_y,a_z,min_gap,y_length;
float x_length,density,dy,f_L,s_L,s_v,x_c,y_c,z_c,lx,ly,lz,error_it,sigma;
float *cx,*cy,*dis_ij_x,*dis_ij_y,**mx,**my,**array_posi,**array_ori,*array_pha;
float *dis_2_ij_x,*dis_2_ij_y;
float *height_s,**center,delta_x,delta_y;
fcomplex a_field_x,a_field_y,a_field_z,im_field[3];
fcomplex k,k1,kg,kz,e1,eg,pre_const,pre_const1,pre_const_xy,pre_const_xy1;
fcomplex sin_wb,cos_wb,Z1;

// The below header file needs some of global variable

#include "mesh_tool1.h"
#include "read_conf.h"

void allow_mesh_memory()
{
  cx = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cx == NULL) {printf("malloc error : cx\n"); exit(0);}

  cy = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cy == NULL) {printf("malloc error : cy\n"); exit(0);}
}

void allow_memory_for_array(n)
{
  int i;

  array_posi = (float **)malloc(sizeof(float *)*n);
  if(array_posi == NULL) {printf("malloc error : array_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    array_posi[i] = malloc_f_1(3);

  array_ori = (float **)malloc(sizeof(float *)*n);
  if(array_ori == NULL) {printf("malloc error : array_ori\n"); exit(0);}
  for(i=0;i<n;i++)
    array_ori[i] = malloc_f_1(3);

  array_pha = malloc_f_1(n);
}

void set_constant()
{
  w0 = 2.*pi*f0;
  k0 = w0*sqrt(e0*u0); kg = RCmul(k0,Csqrt(eg));
  y_0 = sqrt(e0/u0); z0 = 1./y_0;
  e1 = Complex(er_r,er_i+sigma1/(w0*e0));
  eg = Complex(eg_r,eg_i+sigma2/(w0*e0));
  k1 = RCmul(k0,Csqrt(e1));
  k = Cdiv(Cone,e1);
  Z1 = Cdiv(Complex(z0,0.),Csqrt(e1));
  lambda = 2.*pi/k0;

  sin_wb = Csqrt(k); cos_wb = Csqrt(Csub(Cone,k));
  kz = Cmul(k1,cos_wb); kr = k0;
}

int read_conf(x1,x2,x3,y1,y2,y3)
double *x1,*x2,*x3,*y1,*y2,*y3;
{
  FILE *f1;
  unsigned short int flag = 0;
  unsigned int i;
  double w,norm_1;
  double tmp1,tmp2,tmp3;
  fcomplex temp;
  char *file_name,*w_r;

  read_conf_array();
  allow_mesh_memory();
  allow_memory_for_array(a_N);
  real_array_info();


  if(S_N1 < S_N2)
  {
    printf("The # of scatterer around source must ");
    printf("be lager than the receiver!!\n");
```

```
    exit(0);
  }

  for(i=0;i<a_N;i++)
  {
    flag = 0;
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];
    norm_l = sqrt(sqr(lx) + sqr(ly) + sqr(lz));
    lx /= norm_l; ly /= norm_l; lz /= norm_l;
    array_ori[i][0] = lx; array_ori[i][1] = ly; array_ori[i][2] = lz;

    if(lx !=0) if(ly !=0 || lz != 0) flag = 1;
    if(ly !=0) if(lx !=0 || lz != 0) flag = 1;
    if(lz !=0) if(lx !=0 || ly != 0) flag = 1;
    if(flag != 0) {printf("Dipole must have one component!!\n"); exit(0);}

  }

  set_constant();
  s_v = sqrt(s_v);     // transform variance to standard deviation

  printf("There are %d antennas.\n",a_N);
  printf("Antenna Position : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_posi[i][0],array_posi[i][1],array_posi[i][2]);

  printf("Antenna orientation : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_ori[i][0],array_ori[i][1],array_ori[i][2]);

  printf("Phase differnce with reference to the 1st antenna : \n");
  for(i=0;i<a_N;i++)
  {
    printf("%f[Degrees]\n",array_pha[i]);
    array_pha[i] *= pi/180.;
  }

  if(pre_cond == 0)
  printf("%s without pre-conditioner is used for matrix solver.\n",solver_type);
  else
    printf("%s with pre-conditioner is used for matrix solver.\n",solver_type);
  printf("Integral point for z-axis is %d\n",int_N);

  S_N = S_N1;
  tmp1 = x_c; tmp2 = y_c; tmp3 = z_c;
  x_c = a_x; y_c = a_y; z_c = a_z;
  a_x = tmp1; a_y = tmp2; a_z = tmp3;

  return a_N;
}

int initialize_data(x1,x2,x3,y1,y2,y3)
double *x1,*x2,*x3,*y1,*y2,*y3;
{
  double temp;

  a_N = read_conf(x1,x2,x3,y1,y2,y3);

  return a_N;
}

void delay_time(n,f1_)
FILE *f1_;
{
  int i,ii;
  double d,lc;

  lc = 1./sqrt(u0*e0);

  if(n == 0) {
    for(i=0;i<a_N;i++)
       {
      d = sqrt(sqr(a_x - array_posi[i][0]) + sqr(a_y - array_posi[i][1]));
          fprintf(f1_,"%30.281f\n",d/lc);
       }

    for(i=0;i<a_N;i++)
      for(ii=0;ii<S_N1;ii++)
      {
        d = sqrt(sqr(cx[ii]-array_posi[i][0])+sqr(cy[ii]-array_posi[i][1]));
                d += sqrt(sqr(cx[ii]-a_x)+sqr(cy[ii]-a_y));
            fprintf(f1_,"%30.281f\n",d/lc);
      }
  }
  else {
    for(i=0;i<a_N;i++)
      for(ii=0;ii<S_N2;ii++)
      {
        d = sqrt(sqr(cx[ii+S_N1]-array_posi[i][0])+sqr(cy[ii+S_N1]-array_posi[i][1]));
                d += sqrt(sqr(cx[ii+S_N1]-a_x)+sqr(cy[ii+S_N1]-a_y));
            fprintf(f1_,"%30.281f\n",d/lc);
      }
  }
}

void main()
{
  int sN;
  float starting_f,stop_f,m_f,d_f;
  unsigned short int flag_x,flag_y,flag_z;
  unsigned int s_n,l_n,i,ii,p_ii,iii,aii,a,a1,n = 0,m_ii;
  float s_f0,a_c_x = 0,a_c_y = 0.,m_r_x = 0.,m_r_y = 0.;
  double temp,er_r,er_i,sigma1,eg_r,eg_i,sigma2;
  double *lww,*lxx;
  char *file_name,*w_r,*st;
  char fn[200];
  FILE *f2,*f4,*f6;

  f2 = file_open("time_domain.conf","r");

  fscanf(f2,"%s %s\n",fn,fn);
  fscanf(f2,"%f %f\n",&starting_f,&stop_f);
  fscanf(f2,"\n%s\n",fn);
  fscanf(f2,"%d\n",&sN);

  fclose(f2);
  m_f = (starting_f + stop_f)/2.; d_f = -(starting_f - stop_f)/2.;

  lww = malloc_d_1(sN+1); lxx = malloc_d_1(sN+1); gauleg(0.,1.,lxx,lww,sN);

  f2 = file_open("pre_factor.dat","w");
  f4 = file_open("delay_time.dat","w");
  f6 = file_open("mesh_time.dat","w");

  for(i=1;i<=sN;i++) fprintf(f2,"%20.181f %f\n",lww[i],d_f);

  a_N = initialize_data(&er_r,&er_i,&sigma1,&eg_r,&eg_i,&sigma2);
  printf("Initializing data is completed....\n");
  printf("# of scatterers is %d\n",S_N1+S_N2);
  N = determine_N(); printf("Number of element in a cylinder  = %d\n",N);
```

```
  mN = 3*N*S_N;

  for(i=0;i<a_N;i++)
  { a_c_x += array_posi[i][0]; a_c_y += array_posi[i][1];}
  a_c_x /= (float)a_N; a_c_y /= (float)a_N;

  for(i=0;i<a_N;i++)
  {
    temp = fabs(a_c_x - array_posi[i][0]);
    m_r_x = (temp > m_r_x) ? temp : m_r_x;

    temp = fabs(a_c_y - array_posi[i][1]);
    m_r_y = (temp > m_r_y) ? temp : m_r_y;
  }

  if(a_N > 1)
  {
    if(m_r_x != 0.) m_r_x *= 2.;
    else if(m_r_y != 0.) m_r_x = m_r_y/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
    if(m_r_y != 0.) m_r_y *= 2.;
    else if(m_r_x != 0.) m_r_y = m_r_x/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
  }

  printf("Frequency = %f[MHz]\n",f0/1.e6);
  set_constant();

  for(a=0;a<2;a++)
  {
    S_N = (a == 0) ? S_N1 : S_N2;
    mN = 3*N*S_N;

    if(a_N == 1)
        {
      if(a == 0) generate_mesh_array_cir(2,a,a_N,a_c_x,a_c_y,lambda);
      else generate_mesh_array_cir(2+a,a,a_N,a_x,a_y,lambda);
        }
    else
      if(a == 0) generate_mesh_array_rec(2,a,a_N,a_c_x,a_c_y,m_r_x,m_r_y);
      else generate_mesh_array_cir(a+2,a,a_N,a_x,a_y,lambda);

    delay_time(a,f4);
    for(aii=0;aii<S_N;aii++)
      fprintf(f6,"%lf %lf\n",cx[aii+a*S_N1],cy[aii+a*S_N1]);
  }

  fclose(f4); fclose(f6);
  printf("The program ends!!!\n");
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

#include "complex.h"
#include "constant_em.h"
#include "specialfun.h"
#include "calculus.h"
#include "CG_solver.h"
#include "em_tool.h"
#include "random_tool.h"
#include "file_handler.h"
#include "my_malloc.h"
#include "mom_tool.h"
#include "near_field.h"

#define range 2.            // Normalized by lambda
#define spacing (1./2.)     // Normalized by lambda
#define dim 2
#define zero_error 1.e-20

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef distance
  #define distance(x,x1,y,y1,z,z1) sqrt(sqr(x - x1)+sqr(y - y1)+sqr(z - z1))
#endif

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

FILE *f2,*f3,*f4;
char solver_type[10];
unsigned short int N;
unsigned int mN,N1,i_N,S_N,S_N1,S_N2,maxiter,pre_cond,int_N,a_N;
unsigned int *IA,*JA;
double kr,k0,y_0,z0,lambda,i_ang;
double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
double *ww,*xx;
float f0,w0,in_a,j_error,ra,max_rho,delta,a_x,a_y,a_z,min_gap,y_length;
float x_length,density,dy,f_L,s_L,s_v,x_c,y_c,z_c,lx,ly,lz,error_it;
float *cx,*cy,*dis_ij_x,*dis_ij_y,**mx,**my,**array_posi,**array_ori,*array_pha,**obs_po
si;
float *dis_2_ij_x,*dis_2_ij_y;
float *height_s,**center,delta_x,delta_y;
double *cos_sin;
fcomplex sin_wb,cos_wb,Z1,im_field[3],**im_field1;
fcomplex k,k1,kg,kz,e1,eg,pre_const,pre_const1,pre_const_xy,pre_const_xy1;
fcomplex *mat,*F,**m;
fcomplex *J_dis,*P,**PP,*off_d,*off_d0,*off_d1,*off_d_xy;

// The below header file needs some of global variable

#include "incident_field.h"
#include "reflected_wave.h"
#include "mesh_tool1.h"
#include "sparse_matrix.h"
#include "read_conf.h"

fcomplex er_ft(x,y)
double x,y;
{
  return Complex(5.,1.);
}


// Containing routines for generating impedance matrix & current vector
// & function for indexing impedance matrix with sparse matrix scheme
#include "build_and_solve_system.h"

void allow_mesh_memory()
{
  unsigned int i,n;

  height_s = (float *)malloc(sizeof(float)*S_N1);
  if(height_s == NULL) {printf("malloc error : height_s\n"); exit(0);}

  cx = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cx == NULL) {printf("malloc error : cx\n"); exit(0);}

  cy = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cy == NULL) {printf("malloc error : cy\n"); exit(0);}

  n = 3*N*(S_N1+S_N2);
  center = (float **)malloc(sizeof(float *)*n);
  if(center == NULL) {printf("malloc error : center\n"); exit(0);}
  for(i=0;i<n;i++)
    center[i] = malloc_f_1(dim);
}

void allow_varying_size_memory(size_off_d)
{
  dis_ij_x = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_ij_x == NULL) {printf("malloc error : dis_ij\n"); exit(0);}

  dis_ij_y = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_ij_y == NULL) {printf("malloc error : dis_ij\n"); exit(0);}

  dis_2_ij_x = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_2_ij_x == NULL) {printf("malloc error : dis_2_ij\n"); exit(0);}

  dis_2_ij_y = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_2_ij_y == NULL) {printf("malloc error : dis_2_ij\n"); exit(0);}

  cos_sin = (double *)malloc(sizeof(double)*size_off_d);
  if(cos_sin == NULL) {printf("malloc error : cos_sin\n"); exit(0);}

  off_d = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d == NULL) {printf("malloc error : off_d0\n"); exit(0);}

  off_d0 = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d0 == NULL) {printf("malloc error : off_d0\n"); exit(0);}

  off_d1 = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d1 == NULL) {printf("malloc error : off_d1\n"); exit(0);}
```

```c
  off_d_xy = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d_xy == NULL) {printf("malloc error : off_d_xy\n"); exit(0);}
}

// resize allowed memory

void re_allow_varying_size_memory(size_off_d)
{
  int size_memory;

  dis_ij_x = (float *)realloc(dis_ij_x,sizeof(float)*size_off_d);
  if(dis_ij_x == NULL) {printf("realloc error : dis_ij\n"); exit(0);}

  dis_ij_y = (float *)realloc(dis_ij_y,sizeof(float)*size_off_d);
  if(dis_ij_y == NULL) {printf("realloc error : dis_ij\n"); exit(0);}

  dis_2_ij_x = (float *)realloc(dis_2_ij_x,sizeof(float)*size_off_d);
  if(dis_2_ij_x == NULL) {printf("realloc error : dis_2_ij\n"); exit(0);}

  dis_2_ij_y = (float *)realloc(dis_2_ij_y,sizeof(float)*size_off_d);
  if(dis_2_ij_y == NULL) {printf("realloc error : dis_2_ij\n"); exit(0);}

  cos_sin = (double *)realloc(cos_sin,sizeof(double)*size_off_d);
  if(cos_sin == NULL) {printf("realloc error : cos_sin\n"); exit(0);}

  off_d = (fcomplex *)realloc(off_d,sizeof(fcomplex)*size_off_d);
  if(off_d == NULL) {printf("realloc error : off_d0\n"); exit(0);}

  off_d0 = (fcomplex *)realloc(off_d0,sizeof(fcomplex)*size_off_d);
  if(off_d0 == NULL) {printf("realloc error : off_d0\n"); exit(0);}

  off_d1 = (fcomplex *)realloc(off_d1,sizeof(fcomplex)*size_off_d);
  if(off_d1 == NULL) {printf("realloc error : off_d1\n"); exit(0);}

  off_d_xy = (fcomplex *)realloc(off_d_xy,sizeof(fcomplex)*size_off_d);
  if(off_d_xy == NULL) {printf("realloc error : off_d_xy\n"); exit(0);}
}

void allow_memory()
{
  unsigned int i,ii,nn;
  int n;

  allow_mesh_memory();

  mx = (float **)malloc(sizeof(float *)*3*N);
  if(mx == NULL) {printf("malloc error : mx\n"); exit(0);}
  for(i=0;i<3*N;i++)
  {
    mx[i] = (float *)malloc(sizeof(float)*3*N);
    if(mx[i] == NULL) {printf("malloc error : mx\n"); exit(0);}
  }

  my = (float **)malloc(sizeof(float *)*3*N);
  if(my == NULL) {printf("malloc error : mx\n"); exit(0);}
  for(i=0;i<3*N;i++)
  {
    my[i] = (float *)malloc(sizeof(float)*3*N);
    if(my[i] == NULL) {printf("malloc error : my\n"); exit(0);}
  }

  n = 3*N*(3*N+1)/2;

  mat = (fcomplex *)malloc(sizeof(fcomplex)*n);
  if(mat == NULL) {printf("malloc error : mat\n"); exit(0);}

  F = malloc_c_1(mN);
  J_dis = malloc_c_1(mN);

  if(strncmp(solver_type,"BCG",3) == 0) P = malloc_c_1(mN);
  else
  {
    if(strncmp(solver_type,"BICGSTAB_s",10) == 0) nn = 7;
    else nn = 4;

    PP = (fcomplex **)malloc(sizeof(fcomplex *)*nn);
    if(PP == NULL) {printf("malloc error : PP\n"); exit(0);}
    for(i=0;i<nn;i++)
      PP[i] = malloc_c_1(mN);
  }
}

void allow_memory_for_array(n)
{
  int i;

  array_posi = (float **)malloc(sizeof(float *)*n);
  if(array_posi == NULL) {printf("malloc error : array_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    array_posi[i] = malloc_f_1(3);

  array_ori = (float **)malloc(sizeof(float *)*n);
  if(array_ori == NULL) {printf("malloc error : array_ori\n"); exit(0);}
  for(i=0;i<n;i++)
    array_ori[i] = malloc_f_1(3);

  array_pha = malloc_f_1(n);
}

int read_conf()
{
  FILE *f1;
  unsigned short int flag = 0;
  unsigned int i;
  double norm_l;
  double tmp1,tmp2,tmp3;
  fcomplex temp;
  char *file_name,*w_r;

  read_conf_array();
  allow_memory_for_array(a_N);
  real_array_info();

  if(S_N1 < S_N2)
  {
    printf("The # of scatterer around source must ");
    printf("be lager than the receiver!!\n");
    exit(0);
  }

  for(i=0;i<a_N;i++)
  {
    flag = 0;
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];
    norm_l = sqrt(sqr(lx) + sqr(ly) + sqr(lz));
    lx /= norm_l; ly /= norm_l; lz /= norm_l;
    array_ori[i][0] = lx; array_ori[i][1] = ly; array_ori[i][2] = lz;

    if(lx !=0) if(ly !=0 || lz != 0) flag = 1;
```

```
    if(ly !=0) if(lx !=0 || lz != 0) flag = 1;
    if(lz !=0) if(lx !=0 || ly != 0) flag = 1;
    if(flag != 0) {printf("Dipole must have one component!!\n"); exit(0);}

  }

  w0 = 2.*pi*f0;
  k0 = w0*sqrt(e0*u0); kg = RCmul(k0,Csqrt(eg));
  y_0 = sqrt(e0/u0); z0 = 1./y_0;
  e1 = Complex(er_r,er_i+sigma1/(w0*e0));
  eg = Complex(eg_r,eg_i+sigma2/(w0*e0));
  k1 = RCmul(k0,Csqrt(e1));
  k = Cdiv(Cone,e1);
  Z1 = Cdiv(Complex(z0,0.),Csqrt(e1));
  lambda = 2.*pi/k0;

  s_v = sqrt(s_v);      // transform variance to standard deviation

  temp = er_ft(0.,0.);
  printf("Frequency = %f[MHz], radius of trunk = %f[m]  ",f0/1.e6,ra);
  printf("Dielectric constant of trunk = %lf + j %lf\n",temp.r,temp.i);
  printf("Forest heigh = %f & ",f_L);
  printf("effective dielectric constant = %lf + j%lf\n",e1.r,e1.i);
  printf("Dielectric constant of ground = %lf + j%lf\n",eg.r,eg.i);
  printf("ObsErvation point : (%f,%f,%f)\n",x_c,y_c,z_c);

  printf("There are %d antennas.\n",a_N);
  printf("Antenna Position : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_posi[i][0],array_posi[i][1],array_posi[i][2]);

  printf("Antenna orientation : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_ori[i][0],array_ori[i][1],array_ori[i][2]);

  printf("Phase differnce with reference to the 1st antenna : \n");
  for(i=0;i<a_N;i++)
  {
    printf("%f[Degrees]\n",array_pha[i]);
    array_pha[i] *= pi/180.;
  }

  if(pre_cond == 0)
  printf("%s without pre-conditioner is used for matrix solver.\n",solver_type);
  else
    printf("%s with pre-conditioner is used for matrix solver.\n",solver_type);
  printf("Integral point for z-axis is %d\n",int_N);

  return a_N;
}

void first_memory()
{
  IA = (int *)malloc(sizeof(int)*(S_N+1));
  if(IA == NULL) {printf("malloc error : IA\n"); exit(0);}

  xx = (double *)malloc(sizeof(double)*(int_N+1));
  if(xx == NULL) {printf("malloc error : xx\n"); exit(0);}

  ww = (double *)malloc(sizeof(double)*(int_N+1));
  if(ww == NULL) {printf("malloc error : ww\n"); exit(0);}
}

int initialize_data(n)
{
  int n1,i,ii;
  double temp;
  FILE *m_x,*m_y;

  a_N = read_conf();
  first_memory();
  gauleg(0.,1.,xx,ww,int_N);
  sin_wb = Csqrt(k); cos_wb = Csqrt(Csub(Cone,k));

  obs_posi = (float **)malloc(sizeof(float *)*n*n);
  if(obs_posi == NULL) {printf("malloc error : obs_posi\n"); exit(0);}
  for(i=0;i<n*n;i++)
    obs_posi[i] = malloc_f_1(dim);

  im_field1 = (fcomplex **)malloc(sizeof(fcomplex *)*n*n);
  if(im_field1 == NULL) {printf("malloc error : im_field1\n"); exit(0);}
  for(i=0;i<n*n;i++)
    im_field1[i] = malloc_c_1(3);

  m_x = file_open("mesh_x.dat","r");
  m_y = file_open("mesh_y.dat","r");
  for(i=0;i<n;i++)
  {
    fscanf(m_x,"%f\n",&obs_posi[i][0]);
    fscanf(m_y,"%f\n",&obs_posi[i][1]);
  }
  fclose(m_x); fclose(m_y);
  IA[0] = 0;
  temp = sqr(sqr(1. - delta));
  max_rho = sqrt(temp/(1. - temp))*ra; //printf("Max. bound = %f\n",max_rho);

  return a_N;
}

void save_result(n,n1,n2,n3)
{
  unsigned int i,ii;
  float idum = (-1.);
  double x,y,z,L,r_ang;
  fcomplex msf_s_x,msf_s_y,msf_s_z,i_mag_x,i_mag_y,i_mag_z,temp,temp1;

//  x = obs_posi[n3-1][0]; y = obs_posi[n3-1][1]; z = obs_posi[n3-1][2];
  x = x_c; y = y_c; z = z_c;

  if(n1 != 0) for(ii=0;ii<n3*S_N2;ii++) L = gasdev(&idum);
  else for(ii=0;ii<n3*S_N1;ii++) L = gasdev(&idum);

  msf_s_x = zero; msf_s_y = zero; msf_s_z = zero;

  for(ii=0;ii<n;ii++)
  {
        L = gasdev(&idum);  // Gauss Deviates with zero mean & 1. variance of 1
    L = s_v*L + s_L;    // Transformation

    temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
    temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
    msf_s_x = Cadd(msf_s_x,Cadd(temp,temp1));

    temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
    temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
    msf_s_y = Cadd(msf_s_y,Cadd(temp,temp1));

    temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    msf_s_z = Cadd(msf_s_z,Cadd(temp,temp1));
```

```c
  }

  im_field[0] = Cadd(msf_s_x,im_field[0]);
  im_field[1] = Cadd(msf_s_y,im_field[1]);
  im_field[2] = Cadd(msf_s_z,im_field[2]);
}

fcomplex first_field(n,n1,n2,n3)
{
  unsigned int ii,i;
  float idum = (-1.);
  double L,r_ang,x,y,z;
  fcomplex ans,sum,temp,temp1;

  if(n1 != 0) for(ii=0;ii<n1*S_N1;ii++) L = gasdev(&idum);

  ans = zero;

  for(i=0;i<n3;i++)
  {
    sum = zero;
    x = array_posi[i][0]; y = array_posi[i][1]; z = array_posi[i][2];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    for(ii=0;ii<S_N;ii++)
    {
      if((n == 0) && (i == 0))
      {
        L = gasdev(&idum); // Gauss Deviates with zero mean & variance of 1
        L = s_v*L + s_L;// Transformation to deviates with L mean & s_v variance
        height_s[ii] = L;
      }
      else L = height_s[ii];

      if(lx != 0.)
      {
        temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(ly != 0.)
      {
        temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(lz != 0.)
      {
        temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
      }
      sum = Cadd(sum,Cadd(temp,temp1));
    }
    ans = Cadd(ans,Cmul(sum,Cexp(Complex(0.,array_pha[i]))));
  }
  return ans;
}

void main(int argc, char *argv[])

// First argument: Iteration number
// Second argument: Observation Point
// Third argument: simulation step => 1: for scatterers near antenna
//                                    2: for scatterers near observation point

//void main()
{
  unsigned short int flag_x,flag_y,flag_z;
  unsigned int s_n,l_n,ss,i,i1,ii,ii1,iii,a,a1,a2,a3,n = 0,n1;
  double e_s,e_v;
  double temp;
  float *obs_po_x,*obs_po_y;
  float x,y;
  fcomplex temp_c;
  char *file_name,*w_r,st[10];
  char fn[20];
  FILE *f1,*f5,*f6;

  if(argc < 3) {printf("Need more argument!!\n"); exit(0);}
  s_n = atoi(argv[1]);
  l_n = atoi(argv[2]);
  ss = atoi(argv[3]);
  if(ss > 2) {printf("Wrong argument for simulation step!!\n"); exit(0);}

  printf("aaaaa = %d %d %d\n",s_n,l_n,ss);

  printf("The program starts!!!\n");

  n1 = 2*(int)(range/spacing); n1++;

  if(l_n > n1)
    {printf("%d is too big for observation point!!!\n",l_n); exit(0);}

  a_N = initialize_data(n1); S_N = S_N1;
  printf("Initializing data is completed....\n");
  printf("# of scatterers is %d\n",S_N1+S_N2);
  N = determine_N(); printf("Number of element in a cylinder  = %d\n",N);
  mN = 3*N*S_N;
  kz = Cmul(k1,cos_wb); kr = k0; i_ang = acos(kz.r/k1.r);
  printf("k0 = %lf kr = %lf kz = %lf + j%lf\n",k0,kr,kz.r,kz.i);

  allow_memory(); printf("Memory is allowed...\n");

  strcpy(fn,"mesh_s_corr");
  sprintf(st,"%d",s_n);
  strcat(fn,st); sprintf(st,"%d",l_n); strcat(fn,st);
  strcat(fn,".dat");

  printf("fn = %s\n",fn);

  f2 = file_open(fn,"r");

  for(i=0;i<S_N1+S_N2;i++) fscanf(f2,"%f %f\n",&cx[i],&cy[i]);
  fclose(f2);

  printf("Complete read mesh data....\n");

  obs_po_x = malloc_f_1(n1); obs_po_y = malloc_f_1(n1);

  f2 = file_open("mesh_x.dat","r"); f3 = file_open("mesh_y.dat","r");

  for(i=0;i<n1;i++)
  {
    fscanf(f2,"%f\n",&obs_po_x[i]); fscanf(f3,"%f\n",&obs_po_y[i]);
  }

  fclose(f2); fclose(f3);

//  x_c = a_x; y_c = a_y;
  a_x = obs_po_x[l_n]; a_y = obs_po_y[l_n]; a_z = z_c;

  a2 = ss - 1;
```

```c
S_N = (ss == 1) ? S_N1 : S_N2;
mN = 3*N*S_N;

for(i=0;i<S_N1;i++)
  mesh(cx[i],cy[i],i,0);

for(i=0;i<S_N2;i++)
  mesh(cx[i+a2*S_N1],cy[i+a2*S_N1],i,1);

S_N = S_N1;
a1 = sparse_matrix_processing(0,0);
allow_varying_size_memory(a1);
S_N = (ss == 1) ? S_N1 : S_N2;
a1 = sparse_matrix_processing(a2,s_n);

if(fabs(delta_x-delta_y) < 1.e-6) {printf("Homogeneous Mesh...\n");}
else { printf("Error : delta(x) != delta(y)...\n");
       printf("delta_x = %f delta_y = %f\n",delta_x,delta_y); exit(0);}

temp = sqr(delta_x*kr)*(1. - sqr(kr*delta_x)/24.);
pre_const = RCmul(temp,Csub(er_ft(0.,0.),Complex(1.,0.)));
pre_const = Cmul(pre_const,Complex(0.,1./4.));
pre_const1 = Cmul(RCmul(sqrt(1./(pi*kr)),pre_const),Complex(1.,-1.));
pre_const = RCmul(0.5,pre_const);

pre_const_xy = Csub(er_ft(0.,0.),Complex(1.,0.));
pre_const_xy = Cmul(pre_const_xy,Complex(0.,sqr(kr*delta_x)/4.));
pre_const_xy1 = RCmul(-sqrt(1./(pi*kr)),pre_const_xy);
pre_const_xy1 = Cmul(pre_const_xy1,Complex(1.,-1.));

pre_processing(kr,a2);
build_matrix_my(kr,kz,a2);

if(a2 == 0) {
  for(i=0;i<3;i++)
      {
    if(i == 0) {lx = 1.; ly = 0.; lz = 0.;}
    else if(i == 1) {lx = 0.; ly = 1.; lz = 0.;}
         else {lx = 0.; ly = 0.; lz = 1.;}
    build_vector(S_N*N,a2);

    if(i == 0) printf("%dth iteration : x-component\n",s_n);
    else if(i == 1) printf("%dth iteration : y-component\n",s_n);
         else printf("%dth iteration : z-component\n",s_n);

        obtain_F_vector(Z_my);
    im_field[i] = first_field(i,s_n,a2,a_N);
      }
  strcpy(fn,"im_field"); sprintf(st,"%d",s_n);
  strcat(fn,st); sprintf(st,"%d",l_n); strcat(fn,st);
  strcat(fn,".dat");
  f2 = file_open(fn,"w");

  for(ii=0;ii<3;ii++)
    fprintf(f2,"%20.18lf %20.18lf\n",im_field[ii].r,im_field[ii].i);
}
else {
  strcpy(fn,"im_field"); sprintf(st,"%d",s_n);
  strcat(fn,st); sprintf(st,"%d",l_n); strcat(fn,st);
  strcat(fn,".dat");
  f2 = file_open(fn,"r");

  for(ii=0;ii<3;ii++)
    fscanf(f2,"%lf %lf\n",&im_field[ii].r,&im_field[ii].i);
      fclose(f2);

    printf("Considering the near scatterers...\n");
        build_vector(S_N*N,a2);
    obtain_F_vector(Z_my);
    save_result(S_N,s_n,a2,a2);

    strcpy(fn,"s_corr"); sprintf(st,"%d",s_n);
    strcat(fn,st); sprintf(st,"%d",l_n); strcat(fn,st);
    strcat(fn,".dat");
    f2 = file_open(fn,"w");

    for(ii=0;ii<3;ii++)
      fprintf(f2,"%20.18lf %20.18lf\n",im_field[ii].r,im_field[ii].i);

    printf("Save is completed...\n");
        }

  printf("The program ends!!!\n");

}
```

```c
/*
  Time domain analysis with inhomogeneous sampling points
  in frequency domain
  Calculation of Fourier Transform with Gaussian Quadrature alogorithm
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>

#include "complex.h"
#include "constant_em.h"
#include "specialfun.h"
#include "calculus.h"
#include "CG_solver.h"
#include "em_tool.h"
#include "random_tool.h"
#include "file_handler.h"
#include "my_malloc.h"
#include "mom_tool.h"
#include "near_field.h"

#define dim 2
#define zero_error 1.e-20

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

#ifndef distance
  #define distance(x,x1,y,y1,z,z1) sqrt(sqr(x - x1)+sqr(y - y1)+sqr(z - z1))
#endif

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

FILE *f2,*f3,*f4;
char solver_type[20];
unsigned short int N;
unsigned int mN,N1,i_N,S_N,S_N1,S_N2,maxiter,pre_cond,int_N,a_N;
unsigned int *IA,*JA;
double kr,k0,y_0,z0,lambda,i_ang;
double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
double *ww,*xx;
float f0,w0,in_a,j_error,ra,max_rho,delta,a_x,a_y,a_z,min_gap,y_length;
float x_length,density,dy,f_L,s_L,s_v,x_c,y_c,z_c,lx,ly,lz,error_it,sigma;
float *cx,*cy,*dis_ij_x,*dis_ij_y,**mx,**my,**array_posi,**array_ori,*array_pha;
float *dis_2_ij_x,*dis_2_ij_y;
float *height_s,**center,delta_x,delta_y;
double vari_x,vari_y,vari_z;
double *cos_sin;
fcomplex a_field_x,a_field_y,a_field_z,im_field[3];
fcomplex k,k1,kg,kz,e1,eg,pre_const,pre_const1,pre_const_xy,pre_const_xy1;
fcomplex sin_wb,cos_wb,Z1;
fcomplex *mat,*F,**m,**o_m;
fcomplex *J_dis,*P,**PP,*off_d,*off_d0,*off_d1,*off_d_xy;

// The below header file needs some of global variable

#include "incident_field.h"
#include "reflected_wave.h"
#include "mesh_tool1.h"
#include "sparse_matrix.h"
#include "read_conf.h"

fcomplex er_ft(x,y)
double x,y;
{
  return Complex(5.,1.);
}

#include "build_and_solve_system.h"

void allow_mesh_memory()
{
  unsigned int i,n;

  height_s = (float *)malloc(sizeof(float)*S_N1);
  if(height_s == NULL) {printf("malloc error : height_s\n"); exit(0);}

  cx = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cx == NULL) {printf("malloc error : cx\n"); exit(0);}

  cy = (float *)malloc(sizeof(float)*(S_N1+S_N2));
  if(cy == NULL) {printf("malloc error : cy\n"); exit(0);}

  n = 3*N*(S_N1+S_N2);
  center = (float **)malloc(sizeof(float *)*n);
  if(center == NULL) {printf("malloc error : center\n"); exit(0);}
  for(i=0;i<n;i++)
    center[i] = malloc_f_1(dim);
}

void allow_varying_size_memory(size_off_d)
{
  dis_ij_x = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_ij_x == NULL) {printf("malloc error : dis_ij\n"); exit(0);}

  dis_ij_y = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_ij_y == NULL) {printf("malloc error : dis_ij\n"); exit(0);}

  dis_2_ij_x = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_2_ij_x == NULL) {printf("malloc error : dis_2_ij\n"); exit(0);}

  dis_2_ij_y = (float *)malloc(sizeof(float)*size_off_d);
  if(dis_2_ij_y == NULL) {printf("malloc error : dis_2_ij\n"); exit(0);}

  cos_sin = (double *)malloc(sizeof(double)*size_off_d);
  if(cos_sin == NULL) {printf("malloc error : cos_sin\n"); exit(0);}

  off_d = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d == NULL) {printf("malloc error : off_d0\n"); exit(0);}

  off_d0 = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d0 == NULL) {printf("malloc error : off_d0\n"); exit(0);}

  off_d1 = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d1 == NULL) {printf("malloc error : off_d1\n"); exit(0);}
```

```c
  off_d_xy = (fcomplex *)malloc(sizeof(fcomplex)*size_off_d);
  if(off_d_xy == NULL) {printf("malloc error : off_d_xy\n"); exit(0);}
}

void re_allow_varying_size_memory(size_off_d)
{
  int size_memory;

  dis_ij_x = (float *)realloc(dis_ij_x,sizeof(float)*size_off_d);
  if(dis_ij_x == NULL) {printf("realloc error : dis_ij\n"); exit(0);}

  dis_ij_y = (float *)realloc(dis_ij_y,sizeof(float)*size_off_d);
  if(dis_ij_y == NULL) {printf("realloc error : dis_ij\n"); exit(0);}

  dis_2_ij_x = (float *)realloc(dis_2_ij_x,sizeof(float)*size_off_d);
  if(dis_2_ij_x == NULL) {printf("realloc error : dis_2_ij\n"); exit(0);}

  dis_2_ij_y = (float *)realloc(dis_2_ij_y,sizeof(float)*size_off_d);
  if(dis_2_ij_y == NULL) {printf("realloc error : dis_2_ij\n"); exit(0);}

  cos_sin = (double *)realloc(cos_sin,sizeof(double)*size_off_d);
  if(cos_sin == NULL) {printf("realloc error : cos_sin\n"); exit(0);}

  off_d = (fcomplex *)realloc(off_d,sizeof(fcomplex)*size_off_d);
  if(off_d == NULL) {printf("realloc error : off_d0\n"); exit(0);}

  off_d0 = (fcomplex *)realloc(off_d0,sizeof(fcomplex)*size_off_d);
  if(off_d0 == NULL) {printf("realloc error : off_d0\n"); exit(0);}

  off_d1 = (fcomplex *)realloc(off_d1,sizeof(fcomplex)*size_off_d);
  if(off_d1 == NULL) {printf("realloc error : off_d1\n"); exit(0);}

  off_d_xy = (fcomplex *)realloc(off_d_xy,sizeof(fcomplex)*size_off_d);
  if(off_d_xy == NULL) {printf("realloc error : off_d_xy\n"); exit(0);}
}

void allow_memory()
{
  unsigned int i,ii,nn;
  int n;

  allow_mesh_memory();

  mx = (float **)malloc(sizeof(float *)*3*N);
  if(mx == NULL) {printf("malloc error : mx\n"); exit(0);}
  for(i=0;i<3*N;i++)
  {
    mx[i] = (float *)malloc(sizeof(float)*3*N);
    if(mx[i] == NULL) {printf("malloc error : mx\n"); exit(0);}
  }

  my = (float **)malloc(sizeof(float *)*3*N);
  if(my == NULL) {printf("malloc error : mx\n"); exit(0);}
  for(i=0;i<3*N;i++)
  {
    my[i] = (float *)malloc(sizeof(float)*3*N);
    if(my[i] == NULL) {printf("malloc error : my\n"); exit(0);}
  }

  n = 3*N*(3*N+1)/2;

  mat = (fcomplex *)malloc(sizeof(fcomplex)*n);
  if(mat == NULL) {printf("malloc error : mat\n"); exit(0);}
```

```c
  F = malloc_c_1(mN);
  J_dis = malloc_c_1(mN);

  if(strncmp(solver_type,"BCG",3) == 0) P = malloc_c_1(mN);
  else
  {
    if(strncmp(solver_type,"BICGSTAB_s",10) == 0) nn = 7;
     else nn = 4;

    PP = (fcomplex **)malloc(sizeof(fcomplex *)*nn);
    if(PP == NULL) {printf("malloc error : PP\n"); exit(0);}
    for(i=0;i<nn;i++)
      PP[i] = malloc_c_1(mN);
  }
}

void allow_memory_for_array(n)
{
  int i;

  array_posi = (float **)malloc(sizeof(float *)*n);
  if(array_posi == NULL) {printf("malloc error : array_posi\n"); exit(0);}
  for(i=0;i<n;i++)
    array_posi[i] = malloc_f_1(3);

  array_ori = (float **)malloc(sizeof(float *)*n);
  if(array_ori == NULL) {printf("malloc error : array_ori\n"); exit(0);}
  for(i=0;i<n;i++)
    array_ori[i] = malloc_f_1(3);

  array_pha = malloc_f_1(n);
}

void set_constant()
{
  w0 = 2.*pi*f0;
  k0 = w0*sqrt(e0*u0); kg = RCmul(k0,Csqrt(eg));
  y_0 = sqrt(e0/u0); z0 = 1./y_0;
  e1 = Complex(er_r,er_i+sigma1/(w0*e0));
  eg = Complex(eg_r,eg_i+sigma2/(w0*e0));
  k1 = RCmul(k0,Csqrt(e1));
  k = Cdiv(Cone,e1);
  Z1 = Cdiv(Complex(z0,0.),Csqrt(e1));
  lambda = 2.*pi/k0;

  sin_wb = Csqrt(k); cos_wb = Csqrt(Csub(Cone,k));
  kz = Cmul(k1,cos_wb); kr = k0;
}

int read_conf()
{
  FILE *f1;
  unsigned short int flag = 0;
  unsigned int i;
  double w,norm_1;
  double tmp1,tmp2,tmp3;
  fcomplex temp;
  char *file_name,*w_r;

  read_conf_array();
  allow_memory_for_array(a_N);

  real_array_info();

  if(S_N1 < S_N2)
```

```c
  {
    printf("The # of scatterer around source must ");
    printf("be lager than the receiver!!\n");
    exit(0);
  }

  for(i=0;i<a_N;i++)
  {
    flag = 0;
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];
    norm_l = sqrt(sqr(lx) + sqr(ly) + sqr(lz));
    lx /= norm_l; ly /= norm_l; lz /= norm_l;
    array_ori[i][0] = lx; array_ori[i][1] = ly; array_ori[i][2] = lz;

    if(lx !=0) if(ly !=0 || lz != 0) flag = 1;
    if(ly !=0) if(lx !=0 || lz != 0) flag = 1;
    if(lz !=0) if(lx !=0 || ly != 0) flag = 1;
    if(flag != 0) {printf("Dipole must have one component!!\n"); exit(0);}

  }

  set_constant();
  s_v = sqrt(s_v);     // transform variance to standard deviation

  temp = er_ft(0.,0.);
  printf("Radius of trunk = %f[m]  ",ra);
  printf("Dielectric constant of trunk = %lf + j %lf\n",temp.r,temp.i);
  printf("Forest heigh = %f & ",f_L);
  printf("effective dielectric constant = %lf + j%lf\n",e1.r,e1.i);
  printf("Dielectric constant of ground = %lf + j%lf\n",eg.r,eg.i);
  printf("Observation point : (%f,%f,%f)\n",x_c,y_c,z_c);

  printf("There are %d antennas.\n",a_N);
  printf("Antenna Position : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_posi[i][0],array_posi[i][1],array_posi[i][2]);

  printf("Antenna orientation : \n");
  for(i=0;i<a_N;i++)
    printf("(%f,%f,%f)\n",array_ori[i][0],array_ori[i][1],array_ori[i][2]);

  printf("Phase differnce with reference to the 1st antenna : \n");
  for(i=0;i<a_N;i++)
  {
    printf("%f[Degrees]\n",array_pha[i]);
    array_pha[i] *= pi/180.;
  }

  if(pre_cond == 0)
  printf("%s without pre-conditioner is used for matrix solver.\n",solver_type);
  else
    printf("%s with pre-conditioner is used for matrix solver.\n",solver_type);
  printf("Integral point for z-axis is %d\n",int_N);

  S_N = S_N1;
  tmp1 = x_c; tmp2 = y_c; tmp3 = z_c;
  x_c = a_x; y_c = a_y; z_c = a_z;
  a_x = tmp1; a_y = tmp2; a_z = tmp3;

  return a_N;
}

void first_memory()
{
  IA = (int *)malloc(sizeof(int)*(S_N+1));
  if(IA == NULL) {printf("malloc error : IA\n"); exit(0);}

  xx = (double *)malloc(sizeof(double)*(int_N+1));
  if(xx == NULL) {printf("malloc error : xx\n"); exit(0);}

  ww = (double *)malloc(sizeof(double)*(int_N+1));
  if(ww == NULL) {printf("malloc error : ww\n"); exit(0);}
}

int initialize_data()
{
  double temp;

  a_N = read_conf();
  first_memory();

  temp = sqr(sqr(1. - delta));
  max_rho = sqrt(temp/(1. - temp))*ra; printf("Max. bound = %f\n",max_rho);
  IA[0] = 0; x_length = 10.;
  gauleg(0.,1.,xx,ww,int_N);

//  sin_wb = Csqrt(k); cos_wb = Csqrt(Csub(Cone,k));
  return a_N;
}

void obtain_F_vector_up(func)
fcomplex (*func)();
{
  Initial_guess(F,J_dis,mN);

  if(strncmp(solver_type,"BCG",3) == 0)
    BCG(func,F,J_dis,P,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"CGS",3) == 0)
    CGS(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB",20) == 0)
    BICGSTAB(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else if(strncmp(solver_type,"BICGSTAB_s",20) == 0)
    BICGSTAB_s(func,F,J_dis,PP,mN,j_error,maxiter,pre_cond);
  else {printf("Invaild solver type...\n"); exit(0);}
}

void save_result(n,n1,n2)
{
  unsigned int i,ii;
  float idum = (-1.);
  double x,y,z,L,r_ang;
  fcomplex msf_s_x,msf_s_y,msf_s_z,i_mag_x,i_mag_y,i_mag_z,temp,temp1;

  x = a_x; y = a_y; z = a_z;
  if(n1 != 0) for(ii=0;ii<n1*S_N2;ii++) L = gasdev(&idum);
  else for(ii=0;ii<S_N1;ii++) L = gasdev(&idum);

  msf_s_x = zero; msf_s_y = zero; msf_s_z = zero;

  for(ii=0;ii<n;ii++)
  {
    L = gasdev(&idum);  // Gauss Deviates with zero mean & 1. variance of 1
    L = s_v*L + s_L;    // Transformation

    temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
    fprintf(f3,"%30.28lf %30.28lf ",temp.r,temp.i);
    temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
    fprintf(f3,"%30.28lf %30.28lf ",temp1.r,temp1.i);

    temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
```

```c
    fprintf(f3,"%30.28lf %30.28lf ",temp.r,temp.i);
    temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
    fprintf(f3,"%30.28lf %30.28lf ",temp1.r,temp1.i);

    temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    fprintf(f3,"%30.28lf %30.28lf ",temp.r,temp.i);
    temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
    fprintf(f3,"%30.28lf %30.28lf\n",temp1.r,temp1.i);
  }
}

fcomplex first_field(n,n1,n2,n3,f1_)
FILE *f1_;
{
  unsigned int ii,i;
  float idum = (-1.);
  double L,r_ang,x,y,z;
  fcomplex ans,sum,temp,temp1;

  if(n1 != 0) for(ii=0;ii<n1*S_N1;ii++) L = gasdev(&idum);

  ans = zero;

  for(i=0;i<n3;i++)
  {
    sum = zero;
    x = array_posi[i][0]; y = array_posi[i][1]; z = array_posi[i][2];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    for(ii=0;ii<S_N;ii++)
    {
      if((n == 0) && (i == 0))
      {
        L = gasdev(&idum); // Gauss Deviates with zero mean & variance of 1
        L = s_v*L + s_L;// Transformation to deviates with L mean & s_v variance
        height_s[ii] = L;
      }
      else L = height_s[ii];

      if(lx != 0.)
      {
        temp = scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_x_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(ly != 0.)
      {
        temp = scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_y_f(x,y,z,k1,kz,ii,n2,L);
      }
      else if(lz != 0.)
      {
        temp = scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
        temp1 = r_scattered_field_z_f(x,y,z,k1,kz,ii,n2,L);
      }
      fprintf(f1_,"%20.18lf %20.18lf ",temp.r,temp.i);
      fprintf(f1_,"%20.18lf %20.18lf ",temp1.r,temp1.i);
      temp = Cmul(Cadd(temp,temp1),Cexp(Complex(0.,array_pha[i])));
//      fprintf(f1_,"%20.18lf %20.18lf ",temp.r,temp.i);
      sum = Cadd(sum,temp);
    }
    ans = Cadd(ans,sum);
  }
  fprintf(f1_,"\n");
  return ans;
}

fcomplex incident_wave(n,n3,f1_)
FILE *f1_;
{
  unsigned int ii,i;
  double t_a_x,t_a_y,x,y,z;
  fcomplex ans,temp;

  t_a_x = a_x; t_a_y = a_y;
  ans = zero;

  for(i=0;i<n3;i++)
  {
    x = array_posi[i][0]; y = array_posi[i][1]; z = array_posi[i][2];
    lx = array_ori[i][0]; ly = array_ori[i][1]; lz = array_ori[i][2];

    a_x = x; a_y = y;

    if(n == 0) temp = i_field_x(t_a_x,t_a_y);
      else if(n == 1) temp = i_field_y(t_a_x,t_a_y);
        else if(n == 2) temp = i_field_z(t_a_x,t_a_y);
              else {printf("Wrong argument for incident field\n"); exit(0);}
    temp = Cmul(temp,Cexp(Cadd(Complex(0.,array_pha[i]),RCmul(-z,Cmul(j,kz)))));
    fprintf(f1_,"%20.18lf %20.18lf  ",temp.r,temp.i);
    ans = Cadd(ans,temp);
  }
  fprintf(f1_,"\n");
  a_x = t_a_x; a_y = t_a_y;
  return ans;
}

void delay_time(n,f1_)
FILE *f1_;
{
  int i,ii;
  double d,lc;

  lc = 1./sqrt(u0*e0);

  if(n == 0) {
    for(i=0;i<a_N;i++)
       {
      d = sqrt(sqr(a_x - array_posi[i][0]) + sqr(a_y - array_posi[i][1]));
          fprintf(f1_,"%30.28lf\n",d/lc);
       }

    for(i=0;i<a_N;i++)
      for(ii=0;ii<S_N1;ii++)
      {
        d = sqrt(sqr(cx[ii]-array_posi[i][0])+sqr(cy[ii]-array_posi[i][1]));
            d += sqrt(sqr(cx[ii]-a_x)+sqr(cy[ii]-a_y));
          fprintf(f1_,"%30.28lf\n",d/lc);
      }
  }
  else {
    for(i=0;i<a_N;i++)
      for(ii=0;ii<S_N2;ii++)
      {
        d = sqrt(sqr(cx[ii+S_N1]-array_posi[i][0])+sqr(cy[ii+S_N1]-array_posi[i][1]));
            d += sqrt(sqr(cx[ii+S_N1]-a_x)+sqr(cy[ii+S_N1]-a_y));
          fprintf(f1_,"%30.28lf\n",d/lc);
      }
  }
}
```

```
void main(int argc, char *argv[])
{
  int sN;
  float starting_f,stop_f,m_f,d_f;
  unsigned short int flag_x,flag_y,flag_z;
  unsigned int s_n,l_n,i,ii,p_ii,iii,aii,a,a1,n = 0,m_ii,flag=0;
  int *indx;
  double e_s,e_v;
  double temp,variance_x,variance_y,variance_z;
  double er_r,er_i,sigma1,eg_r,eg_i,sigma2;
  double *lww,*lxx;
  float s_f0,a_c_x = 0,a_c_y = 0.,m_r_x = 0.,m_r_y = 0.;
  fcomplex d,*col,tmp;
  char *file_name,*w_r,*st;
  char fn[200];
  FILE *f1,*f5,*f6;

  if(argc == 1) {printf("Need more argument!!\n"); exit(0);}
  s_n = atoi(argv[1]);
  if(argc == 2) l_n = s_n + 1;
  else l_n = atoi(argv[2]);
  st = argv[1];

  f1 = file_open("time_domain.conf","r");

  fscanf(f1,"%s %s\n",fn,fn);
  fscanf(f1,"%f %f\n",&starting_f,&stop_f);
  fscanf(f1,"\n%s\n",fn);
  fscanf(f1,"%d\n",&sN);

  fclose(f1);
  m_f = (starting_f + stop_f)/2.; d_f = -(starting_f - stop_f)/2.;

  printf("sN = %d\n",sN);
  printf("%f %f\n",starting_f,stop_f);

  if(s_n > 2*sN - 1) {printf("Too large argument!!!\n"); exit(0);}

  printf("The program starts!!!\n");

  a_N = initialize_data();
  printf("Initializing data is completed....\n");
  printf("# of scatterers is %d\n",S_N1+S_N2);
  N = determine_N(); printf("Number of element in a cylinder  = %d\n",N);
  mN = 3*N*S_N;

  lww = malloc_d_1(sN+1); lxx = malloc_d_1(sN+1); gauleg(0.,1.,lxx,lww,sN);

  strcpy(fn,"frequency.dat");
  strcat(fn,st); w_r = "w";
  f1 = file_open(fn,w_r);

  strcpy(fn,"fre_inh1.dat");
  strcat(fn,st); w_r = "w";
  f2 = file_open(fn,w_r);

  strcpy(fn,"fre_inh2.dat");
  strcat(fn,st); w_r = "w";
  f3 = file_open(fn,w_r);

  strcpy(fn,"incident_field_inh.dat");
  strcat(fn,st); w_r = "w";
  f5 = file_open(fn,w_r);

  f4 = file_open("mesh_time.dat","r");

  for(i=0;i<a_N;i++)
  { a_c_x += array_posi[i][0]; a_c_y += array_posi[i][1];}
  a_c_x /= (float)a_N; a_c_y /= (float)a_N;

  for(i=0;i<a_N;i++)
  {
    temp = fabs(a_c_x - array_posi[i][0]);
    m_r_x = (temp > m_r_x) ? temp : m_r_x;

    temp = fabs(a_c_y - array_posi[i][1]);
    m_r_y = (temp > m_r_y) ? temp : m_r_y;
  }

  if(a_N > 1)
  {
    if(m_r_x != 0.) m_r_x *= 2.;
    else if(m_r_y != 0.) m_r_x = m_r_y/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
    if(m_r_y != 0.) m_r_y *= 2.;
    else if(m_r_x != 0.) m_r_y = m_r_x/(float)a_N;
    else {printf("Something wrong with array location!\n"); exit(0);}
  }

  allow_memory(); printf("Memory is allowed...\n");
  s_f0 = f0;

  for(i=0;i<S_N1+S_N2;i++) fscanf(f4,"%f %f\n",&cx[i],&cy[i]);

  for(iii=0;iii<i_N;iii++)
  {
    // for 3*sigma
    if(s_n < sN) f0 = s_f0 + d_f*lxx[s_n + 1];
        else f0 = s_f0 - d_f*lxx[s_n - sN + 1];

    printf("Frequency = %f[MHz]\n",f0/1.e6);
    set_constant();
    printf("Iteration Number : %d \n",s_n);
    fprintf(f1,"%f\n",f0);

    for(a=0;a<3;a++)
      im_field[a] = incident_wave(a,a_N,f5);

    for(a=0;a<2;a++)
    {
      S_N = (a == 0) ? S_N1 : S_N2;
      mN = 3*N*S_N;

          for(i=0;i<S_N;i++)
          mesh(cx[i+a*S_N1],cy[i+a*S_N1],i,a);

      a1 = sparse_matrix_processing(a,iii);

      if(a == 0) {
        if(iii == 0) allow_varying_size_memory(a1);
        else re_allow_varying_size_memory(a1);
               }
      if((iii == 0) && (a == 0))
      {
        if(fabs(delta_x-delta_y) < 1.e-6) {printf("Homogeneous Mesh...\n");}
        else { printf("Error : delta(x) != delta(y)...\n");
               printf("delta_x = %f delta_y = %f\n",delta_x,delta_y); exit(0);}

        temp = sqr(delta_x*kr)*(1. - sqr(kr*delta_x)/24.);
        pre_const = RCmul(temp,Csub(er_ft(0.,0.),Complex(1.,0.)));
```

```
      pre_const = Cmul(pre_const,Complex(0.,1./4.));
      pre_const1 = Cmul(RCmul(sqrt(1./(pi*kr)),pre_const),Complex(1.,-1.));
      pre_const = RCmul(0.5,pre_const);

      pre_const_xy = Csub(er_ft(0.,0.),Complex(1.,0.));
      pre_const_xy = Cmul(pre_const_xy,Complex(0.,sqr(kr*delta_x)/4.));
      pre_const_xy1 = RCmul(-sqrt(1./(pi*kr)),pre_const_xy);
      pre_const_xy1 = Cmul(pre_const_xy1,Complex(1.,-1.));
     }

     pre_processing(kr,a);
     build_matrix_my(kr,kz,a);
     if(a == 0)
       for(i=0;i<3;i++)
           {
         if(i == 0) {lx = 1.; ly = 0.; lz = 0.;}
         else if(i == 1) {lx = 0.; ly = 1.; lz = 0.;}
              else {lx = 0.; ly = 0.; lz = 1.;}
         build_vector(S_N*N,a);

         if(i == 0) printf("%dth iteration : x-component\n",s_n);
         else if(i == 1) printf("%dth iteration : y-component\n",s_n);
              else printf("%dth iteration : z-component\n",s_n);

         obtain_F_vector_up(Z_my);
         im_field[i] = first_field(i,s_n,a,a_N,f2);
           }
     else {
       if(S_N != 0) {
         printf("Considering the near scatterers...\n");
         build_vector(S_N*N,a);
         obtain_F_vector_up(Z_my);
       }
       save_result(S_N,s_n,a);
       printf("Save is completed...\n");
         }
    }
  }

  fclose(f2); fclose(f3); fclose(f5);
  printf("The program ends!!!\n");
}
```

```c
/*
  Converting frequency domain into time domain
  using gaussian quatrature numerical integration of Fourier Transform
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "complex.h"
#include "constant_em.h"
#include "my_malloc.h"
#include "file_handler.h"

#ifndef zero
  #define zero Complex(0.,0.)
#endif

#ifndef j
  #define j Complex(0.,1.)
#endif

#ifndef Cone
  #define Cone Complex(1.,0.)
#endif

#ifndef sqr
  #define sqr(x) ((x)*(x))
#endif

double fc,BW;
double sigma,t0;

int N,a_N,S_N1,S_N2;
double d_f,mean_f,delta_t,exact_delay;
double *f,*ww,*delay_time;
fcomplex **ifx,**ify,**ifz,**sfx,**sfy,**sfz;
fcomplex **sfx1,**sfy1,**sfz1,**sfx2,**sfy2,**sfz2;

int file_length(name_)
char name_[];
{
  int sum_ = 0;
  char c_;
  FILE *f_;

  if(f_ = file_open(name_,"r"))
  {
    while((c_ = getc(f_)) != EOF)
      if(c_ == '\n') sum_++;
    fclose(f_);
  }

  return sum_;
}

double mean_f_(n)
{
  double sum = 0.e100;
  int i;

  for(i=0;i<n-1;i++)
    sum += fabs(f[i] - f[i+1]);

  return sum/(double)n;
}

void read_data()
{
  int i,ii,n,s_i,s_ii;
  float tmp;
  double temp,temp1;
  char ttemp[50];
  fcomplex gaussian;
  FILE *input;

  sigma = 3./(2.*pi*BW); t0 = 3.*sigma;

  N = file_length("frequency.dat");

  if(N % 2 != 0) {printf("Must be even sampling points!!\n"); exit(0);}

  input = file_open("array.conf","r");

  fscanf(input,"%s %s %s %s\n",ttemp,ttemp,ttemp,ttemp);
  fscanf(input,"%f %f %f %f\n",&tmp,&tmp,&tmp,&tmp);

  fscanf(input,"\n%s %s %s %s %s %s %s\n",ttemp,ttemp,ttemp,ttemp,ttemp,ttemp,ttemp);
  fscanf(input,"%f %f %f ",&tmp,&tmp,&tmp);
  fscanf(input,"%f %f %f %f\n",&tmp,&tmp,&tmp,&tmp);

  fscanf(input,"\n%s %s %s %s %s %s\n",ttemp,ttemp,ttemp,ttemp,ttemp,ttemp);
  fscanf(input,"%lf %lf ",&temp,&temp);
  fscanf(input,"%lf %lf %lf\n",&temp,&temp,&temp);

  fscanf(input,"\n%s %s %s %s %s %s %s\n",ttemp,ttemp,ttemp,ttemp,ttemp,ttemp,ttemp);
  fscanf(input,"%d %d %d %d %d ",&a_N,&i,&S_N1,&S_N2,&i);

  fclose(input);

  printf("Number of Antennas = %d\n",a_N);
  printf("Sample size = %d\n",N);

  f = malloc_d_1(N);
  ifx = malloc_c_2(a_N,N); ify = malloc_c_2(a_N,N); ifz = malloc_c_2(a_N,N);
  sfx = malloc_c_2(2*a_N*S_N1,N);
  sfy = malloc_c_2(2*a_N*S_N1,N);
  sfz = malloc_c_2(2*a_N*S_N1,N);
  sfx1 = malloc_c_2(S_N2,N);
  sfy1 = malloc_c_2(S_N2,N);
  sfz1 = malloc_c_2(S_N2,N);
  sfx2 = malloc_c_2(S_N2,N);
  sfy2 = malloc_c_2(S_N2,N);
  sfz2 = malloc_c_2(S_N2,N);
  delay_time = malloc_d_1(S_N1+S_N2+2);

  input = file_open("delay_time.dat","r");
  for(i=0;i<S_N1+S_N2+2;i++) fscanf(input,"%lf\n",&delay_time[i]);
  fclose(input);

  input = file_open("frequency.dat","r");
  for(i=0;i<N;i++) fscanf(input,"%lf\n",&f[i]);
  fclose(input);

  mean_f = mean_f_(N/2)*(2.*pi);

  input = file_open("incident_field_inh.dat","r");
```

```
for(i=0;i<3*N;i++)
{
  n = (int)((double)i/3.);
  for(ii=0;ii<a_N;ii++)
  {
    if(ii == (a_N - 1)) {
      if(i % 3 == 0) fscanf(input,"%lf %lf\n",&ifx[ii][n].r,&ifx[ii][n].i);
      if(i % 3 == 1) fscanf(input,"%lf %lf\n",&ify[ii][n].r,&ify[ii][n].i);
      if(i % 3 == 2) fscanf(input,"%lf %lf\n",&ifz[ii][n].r,&ifz[ii][n].i);
            }
    else {
      if(i % 3 == 0) fscanf(input,"%lf %lf ",&ifx[ii][n].r,&ifx[ii][n].i);
      if(i % 3 == 1) fscanf(input,"%lf %lf ",&ify[ii][n].r,&ify[ii][n].i);
      if(i % 3 == 2) fscanf(input,"%lf %lf ",&ifz[ii][n].r,&ifz[ii][n].i);
        }
      }
}

fclose(input);

input = file_open("fre_inh1.dat","r");

for(i=0;i<3*N;i++)
{
  n = (int)((double)i/3.);
  for(ii=0;ii<2*a_N*S_N1;ii++)
  {
    if(ii == (2*a_N*S_N1 - 1)) {
      if(i % 3 == 0) fscanf(input,"%lf %lf\n",&sfx[ii][n].r,&sfx[ii][n].i);
      if(i % 3 == 1) fscanf(input,"%lf %lf\n",&sfy[ii][n].r,&sfy[ii][n].i);
      if(i % 3 == 2) fscanf(input,"%lf %lf\n",&sfz[ii][n].r,&sfz[ii][n].i);
            }
    else {
      if(i % 3 == 0) fscanf(input,"%lf %lf ",&sfx[ii][n].r,&sfx[ii][n].i);
      if(i % 3 == 1) fscanf(input,"%lf %lf ",&sfy[ii][n].r,&sfy[ii][n].i);
      if(i % 3 == 2) fscanf(input,"%lf %lf ",&sfz[ii][n].r,&sfz[ii][n].i);
        }
  }
}

fclose(input);

input = file_open("fre_inh2.dat","r");

for(i=0;i<N;i++)
  for(ii=0;ii<S_N2;ii++)
  {
    fscanf(input,"%lf %lf ",&sfx1[ii][i].r,&sfx1[ii][i].i);
    fscanf(input,"%lf %lf ",&sfx2[ii][i].r,&sfx2[ii][i].i);
    fscanf(input,"%lf %lf ",&sfy1[ii][i].r,&sfy1[ii][i].i);
    fscanf(input,"%lf %lf ",&sfy2[ii][i].r,&sfy2[ii][i].i);
    fscanf(input,"%lf %lf ",&sfz1[ii][i].r,&sfz1[ii][i].i);
    fscanf(input,"%lf %lf\n",&sfz2[ii][i].r,&sfz2[ii][i].i);
  }

fclose(input);

for(ii=0;ii<S_N2;ii++)
  for(i=0;i<N;i++)
  {
    temp = exp(-sqr(sigma*2.*pi)*sqr(f[i] - fc)/2.);
    gaussian = Complex(temp,0.);

    sfx1[ii][i] = Cmul(gaussian,sfx1[ii][i]);
    sfy1[ii][i] = Cmul(gaussian,sfy1[ii][i]);
    sfz1[ii][i] = Cmul(gaussian,sfz1[ii][i]);

    sfx2[ii][i] = Cmul(gaussian,sfx2[ii][i]);
    sfy2[ii][i] = Cmul(gaussian,sfy2[ii][i]);
    sfz2[ii][i] = Cmul(gaussian,sfz2[ii][i]);
  }

  for(i=0;i<N;i++)
    for(ii=0;ii<a_N;ii++)
    {
      temp = exp(-sqr(sigma*2.*pi)*sqr(f[i] - fc)/2.);
      gaussian = Complex(temp,0.);
      ifx[ii][i] = Cmul(gaussian,ifx[ii][i]);
      ify[ii][i] = Cmul(gaussian,ify[ii][i]);
      ifz[ii][i] = Cmul(gaussian,ifz[ii][i]);
    }

  for(i=0;i<N;i++)
    for(ii=0;ii<2*a_N*S_N1;ii++)
    {
      temp = exp(-sqr(sigma*2.*pi)*sqr(f[i] - fc)/2.);
      gaussian = Complex(temp,0.);
      sfx[ii][i] = Cmul(gaussian,sfx[ii][i]);
      sfy[ii][i] = Cmul(gaussian,sfy[ii][i]);
      sfz[ii][i] = Cmul(gaussian,sfz[ii][i]);
    }

//  for(i=0;i<N;i++) f[i] -= (fc - BW);
  for(i=0;i<N;i++) f[i] -= fc;

  N = file_length("pre_factor.dat");
  printf("N = %d\n",N);

  ww = malloc_d_1(N);
  for(i=0;i<N;i++) ww[i] = 0.;

  input = file_open("pre_factor.dat","r");
  for(i=0;i<N;i++)
    fscanf(input,"%lf %lf\n",&ww[i],&d_f);
  fclose(input);
}

fcomplex fourier(N_,fre_,in_data_,t_,ww_)
fcomplex *in_data_; double *fre_,*ww_,t_;
{
  fcomplex temp_,ans_ = zero;
  double tmp_;
  int i_;

  tmp_ = -2.*pi*t_;
  for(i_=0;i_<N_;i_++)
  {
    temp_ = Cmul(in_data_[i_+N_],Cexp(Complex(0.,fre_[i_+N_]*tmp_)));
        temp_ = Cadd(temp_,Cmul(in_data_[i_],Cexp(Complex(0.,fre_[i_]*tmp_))));
        ans_ = Cadd(ans_,RCmul(ww_[i_],temp_));
  }
  return RCmul(d_f,ans_);
}


double find_delay(n1,n,data,t,t0)
fcomplex *data; double *t; double t0;
{
  int i,save_t=0;
  double s_time,e_time,mag = 0.;
```

```
  s_time = delay_time[n1] - 1./mean_f; e_time = delay_time[n1] + 2./mean_f;
  for(i=0;i<n;i++)
    if((t[i] > s_time) && (t[i] < e_time)) {
          if(mag <= fabs(data[i].r)) {save_t = i; mag = fabs(data[i].r);}
    }

  return t[save_t];
}

void filtering_field(n,n1,data,t)
fcomplex *data; double *t;
{
  const double alpha1 = 0.3,alpha2 = 0.1;
  int i,ii=0;
  double t_sigma1,t_sigma2,s_time,e_time,t1;

  printf("exact_delay = %e delay_time = %e\n",exact_delay,delay_time[n]);

  s_time = exact_delay - 1./mean_f; e_time = exact_delay + 1./mean_f;
  t_sigma1 = (alpha1*2./mean_f)/3.; t_sigma2 = (alpha2*2./mean_f)/3.;
  for(i=0;i<n1;i++)
    if((t[i] < s_time) || (t[i] > e_time))
      data[i] = zero;
    else {
          if(t[i] > (t1=(e_time - alpha2*2./mean_f)))
            data[i] = RCmul(exp(-sqr(t[i] - t1)/(2.*sqr(t_sigma2))),data[i]);
      else if(t[i] < (t1=(s_time + alpha1*2./mean_f)))
             data[i] = RCmul(exp(-sqr(t[i] - t1)/(2.*sqr(t_sigma1))),data[i]);
        }
}

void main()
{
  int i,ii,jj,i1=0,o_N,t_N;
  double *t,df,s_t,t_sigma,n1,n2;
  fcomplex temp,*ffz;
  fcomplex *o_data_ix,*o_data_iy,*o_data_iz;
  fcomplex *o_data_sx,*o_data_sy,*o_data_sz;
  fcomplex *im_data_x,*im_data_y,*im_data_z,*im_o_data;
  float sf,ef,dis_ro,range_con;
  char ttemp[50];
  FILE *o_ifx,*o_ify,*o_ifz,*o_sfx,*o_sfy,*o_sfz,*f1;

  f1 = file_open("time_domain.conf","r");

  fscanf(f1,"%s %s\n",ttemp,ttemp);
  fscanf(f1,"%f %f\n",&sf,&ef);
  fscanf(f1,"%s\n",ttemp);
  fscanf(f1,"%d\n",i);
  fscanf(f1,"\n%s\n",ttemp);
  fscanf(f1,"%f\n",&dis_ro);

  fclose(f1);

  fc = (sf + ef)/2.; BW = (ef - sf)/2.;
  range_con = 200.;

  read_data();
  o_N = 2*N*30;

  df = f[1] - f[0];

  t = malloc_d_1(o_N);
  o_data_ix = malloc_c_1(o_N);
  o_data_iy = malloc_c_1(o_N);
  o_data_iz = malloc_c_1(o_N);
  o_data_sx = malloc_c_1(o_N);
  o_data_sy = malloc_c_1(o_N);
  o_data_sz = malloc_c_1(o_N);
  im_o_data = malloc_c_1(o_N);

  im_data_x = malloc_c_1(2*N);
  im_data_y = malloc_c_1(2*N);
  im_data_z = malloc_c_1(2*N);

  ffz = malloc_c_1(2*N);

//  t[0] = s_t = 1000./3.e8 - 200./3.e8 + t0;
//  for(i=1;i<o_N;i++) t[i] = t[i-1] + 400./3.e8/(o_N - 1);
  t[0] = s_t = dis_ro/3.e8 - range_con/3.e8 + t0;
  for(i=1;i<o_N;i++) t[i] = t[i-1] + 2*range_con/3.e8/(o_N - 1);

  exact_delay = t[o_N/2];
  for(i=0;i<o_N;i++) im_o_data[i] = Complex(1.,0.);
  filtering_field(0,o_N,im_o_data,t);

  o_ifx = file_open("filter.dat","w");

  for(i=0;i<o_N;i++) fprintf(o_ifx,"%lf\n",im_o_data[i].r);

  fclose(o_ifx);

  for(i=0;i<o_N;i++)
  {
    o_data_ix[i] = zero; o_data_iy[i] = zero; o_data_iz[i] = zero;
    o_data_sx[i] = zero; o_data_sy[i] = zero; o_data_sz[i] = zero;
  }

  printf("Pulse width = %e, 1/<f> = %e\n",6.*sigma,1./mean_f);

  o_ifx = file_open("ifx_TD_inh.dat","w");
  o_ify = file_open("ify_TD_inh.dat","w");
  o_ifz = file_open("ifz_TD_inh.dat","w");

  o_sfx = file_open("sfx_TD_inh.dat","w");
  o_sfy = file_open("sfy_TD_inh.dat","w");
  o_sfz = file_open("sfz_TD_inh.dat","w");

  for(ii=0;ii<a_N;ii++)
  {
    for(jj=0;jj<2*N;jj++)
    {
      im_data_x[jj] = ifx[ii][jj];
      im_data_y[jj] = ify[ii][jj];
      im_data_z[jj] = ifz[ii][jj];
        }

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_x,t[i],ww);

    exact_delay = find_delay(ii,o_N,im_o_data,t,t0);
    filtering_field(ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_ix[i] = Cadd(o_data_ix[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_y,t[i],ww);
```

```
    exact_delay = find_delay(ii,o_N,im_o_data,t,t0);
    filtering_field(ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_iy[i] = Cadd(o_data_iy[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_z,t[i],ww);

    exact_delay = find_delay(ii,o_N,im_o_data,t,t0);
    filtering_field(ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_iz[i] = Cadd(o_data_iz[i],im_o_data[i]);
  }

//  t = s_t = 1./(2.*df);

  for(ii=0;ii<2*a_N*S_N1;ii++)
  {
    for(jj=0;jj<2*N;jj++)
    {
      im_data_x[jj] = sfx[ii][jj];
      im_data_y[jj] = sfy[ii][jj];
      im_data_z[jj] = sfz[ii][jj];
    }

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_x,t[i],ww);

    exact_delay = find_delay(i1+a_N,o_N,im_o_data,t,t0);
    filtering_field(i1+a_N,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sx[i] = Cadd(o_data_sx[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_y,t[i],ww);

    exact_delay = find_delay(i1+a_N,o_N,im_o_data,t,t0);
    filtering_field(i1+a_N,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sy[i] = Cadd(o_data_sy[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_z,t[i],ww);

    exact_delay = find_delay(i1+a_N,o_N,im_o_data,t,t0);
    filtering_field(i1+a_N,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sz[i] = Cadd(o_data_sz[i],im_o_data[i]);

    if((ii % 2 == 1) && (ii > a_N)) i1++;
  }

  for(ii=0;ii<S_N2;ii++)
  {
    for(jj=0;jj<2*N;jj++)
    {
      im_data_x[jj] = sfx1[ii][jj];
      im_data_y[jj] = sfy1[ii][jj];
      im_data_z[jj] = sfz1[ii][jj];
    }

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_x,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sx[i] = Cadd(o_data_sx[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_y,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sy[i] = Cadd(o_data_sy[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_z,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+ii,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sz[i] = Cadd(o_data_sz[i],im_o_data[i]);

    for(jj=0;jj<2*N;jj++)
    {
      im_data_x[jj] = sfx2[ii][jj];
      im_data_y[jj] = sfy2[ii][jj];
      im_data_z[jj] = sfz2[ii][jj];
    }

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_x,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+1,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sx[i] = Cadd(o_data_sx[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_y,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+1,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sy[i] = Cadd(o_data_sy[i],im_o_data[i]);

    for(i=0;i<o_N;i++)
      im_o_data[i] = fourier(N,f,im_data_z,t[i],ww);

    exact_delay = find_delay(ii+a_N+S_N1,o_N,im_o_data,t,t0);
    filtering_field(S_N1+a_N+1,o_N,im_o_data,t);

    for(i=0;i<o_N;i++)
      o_data_sz[i] = Cadd(o_data_sz[i],im_o_data[i]);
  }

  for(i=0;i<o_N;i++)
  {
```

```
    fprintf(o_sfx,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_sx[i].r,o_data_sx[i].i);

    fprintf(o_sfy,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_sy[i].r,o_data_sy[i].i);

        fprintf(o_sfz,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_sz[i].r,o_data_sz[i]
.i);

    fprintf(o_ifx,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_ix[i].r,o_data_ix[i].i);

    fprintf(o_ify,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_iy[i].r,o_data_iy[i].i);

        fprintf(o_ifz,"%20.18lf %20.18lf %20.18lf\n",t[i]+t0,o_data_iz[i].r,o_data_iz[i]
.i);
  }
}
```