ARL-TR-9059 ● SEP 2020

DEVCOM ARMY RESEARCH LABORATORY

Simulation of Acoustic Propagation of Elevated Sources to a Microphone Array

by Geoffrey H Goldman

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9059 ● SEP 2020

Simulation of Acoustic Propagation of Elevated Sources to a Microphone Array

Geoffrey H Goldman
Sensors and Electron Devices Directorate, CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2020	Technical Report	1 October 2019–30 September 2020

4. TITLE AND SUBTITLE
Simulation of Acoustic Propagation of Elevated Sources to a Microphone Array

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Geoffrey H Goldman

5d. PROJECT NUMBER
WR.0033636.1.1

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLS-SA
Adelphi, MD 20783-1138

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-9059

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
A MATLAB code is developed to propagate harmonic signals from elevated sources in the lower audio band (30–500 Hz) to a small microphone array near the ground using standard models to simulate propagation speed profiles, wind speed and direction, ground reflection, and turbulence to predict the phase and relative amplitude of a received narrowband signal. Wind speed and temperature profiles are modeled using Monin–Obukhov similarity theory. Signal propagation from a source is simulated using ray tracing through a stratified atmosphere (from Pierce) and a finite-difference, time-domain method. The ground reflection is modeled using either a one-parameter model (by Delany and Bazley) or a two-parameter ground impedance model (by Attenborough et al.). Spatial coherence of the signal at the array is modeled using a statistical approach by Kozick et al. The coherence of signals at different microphones is reduced based upon the range of the source and distance between microphones. Using these models, propagation time from the source to the microphone array for direct and multipath signals is calculated using custom 3D interpolation algorithms. The errors associated with these algorithms are analyzed by testing with a homogeneous atmosphere. This code is a building block for simulating more complex signals and a resource for testing beamforming algorithms.

15. SUBJECT TERMS
Monin–Obukhov similarity theory, signal propagation, ray tracing, spatial coherence, 3D interpolation, beamforming

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Geoffrey H Goldman
Unclassified	Unclassified	Unclassified	UU	66	19b. TELEPHONE NUMBER (Include area code) (301) 394-0882

Contents

List of Figures

List of Tables

Acknowledgments

I would thank Vladimir Ostashev for the code to simulate sound speed and wind speed profiles using Monin–Obukhov similarity theory, Kirk Alberts for his help on modeling reflections from the ground, and Brian Sadler for his help on modeling turbulence.

1. Introduction

Acoustic systems on the battlefield are an important sensor for the Soldier. Typically, passive acoustic systems use either lower frequency with sound in the infrasound spectrum (below 20 Hz) or higher-frequency sound in the lower end of the audio spectrum (20–5000 Hz). Simulating lower-frequency systems is usually more complicated due to lower signal attenuation and longer detection ranges. However, the same phenomenology is present at higher frequencies, but often ignored due to its small effect on results. For example, most classical array signal processing algorithms assume the signals propagate in straight lines. For most high-frequency systems employing small microphone arrays, this is a reasonable approximation. However, for low-frequency systems, the sound propagation models usually produce curved paths. As greater accuracy is required for higher-frequency systems, the signal processing algorithms will also need to incorporate effects such as wind and temperature gradients, ground reflection, and turbulence that are normally present in low-frequency systems.

Classical acoustic array signal processing algorithms are based on signal models with a constant amplitude and spatially varying time delays or phase shifts calculated from the geometry of the microphone array and the location of the source. The microphones are typically located in free space and in the far field. The amplitude of the signal is usually unknown, but given the direction of arrival (DOA) of the emitted signals, the relative phase relationships of the received signals at an array of microphones can be calculated. There are errors associated with using this model, but using a more complex model is often not practical, particularly for sources and arrays near the ground. For this scenario, an accurate simulation requires a detailed description of the terrain and environmental conditions between the source and the array. However, for elevated acoustic targets at tactical ranges, the interaction of the source signal with the terrain is limited, since most of the propagation is well above the ground. This allows the propagation to be modeled using simpler techniques with fewer initial conditions. In addition, large temperature and wind gradients with respect to height near the ground can affect signals from elevated sources more than smaller gradients with respect to the ground coordinates for sources near grazing angles. Using better models for scenarios with elevated targets should improve the evaluation accuracy for DOA, detection, and classification algorithms.

2. Simulation Overview

The code described in this report simulates a narrowband signal received at a small microphone array emitted from an elevated source at a given frequency and direction of departure. The input to the code includes the frequency, amplitude, phase offset, 3D location of the emitted signal, and the geometry for an array of microphones. The software also requires input describing the environmental conditions that will be used to model the temperature and wind profiles, ground reflection, and turbulence. The output of the code is the simulated received signal at each microphone location. It is calculated by adding the signals from direct and indirect ray paths generated by the source and received by the microphones. This is illustrated later in the report and is discussed in more detail in the next section. The signals are modeled as sinusoidal functions with a constant amplitude, phase, and frequency. Sources with signatures that contain multiple frequencies can be simulated by adding together multiple single-frequency waveforms. For the raytracing and ground reflection models selected, the ray paths are independent of frequency. However, the effect of turbulence is highly dependent on frequency. The time for a signal to propagate from the source to the microphone array is calculated using custom 3D interpolation algorithms.

The absolute location of the microphone array is determined by other input parameters and is not an assignable field. For some applications, such as simulating the performance of beamforming algorithms, this is acceptable. For other applications, a preprocessing direction-of-departure estimation algorithm will need to be developed. The code does not calculate atmospheric, spherical spreading, or ray divergence losses. For simulating beamforming performance, these losses are often not needed. The critical information is usually the signal-to-interference-plus-noise ratio (SINR), which can be simulated with additive noise and interference models.

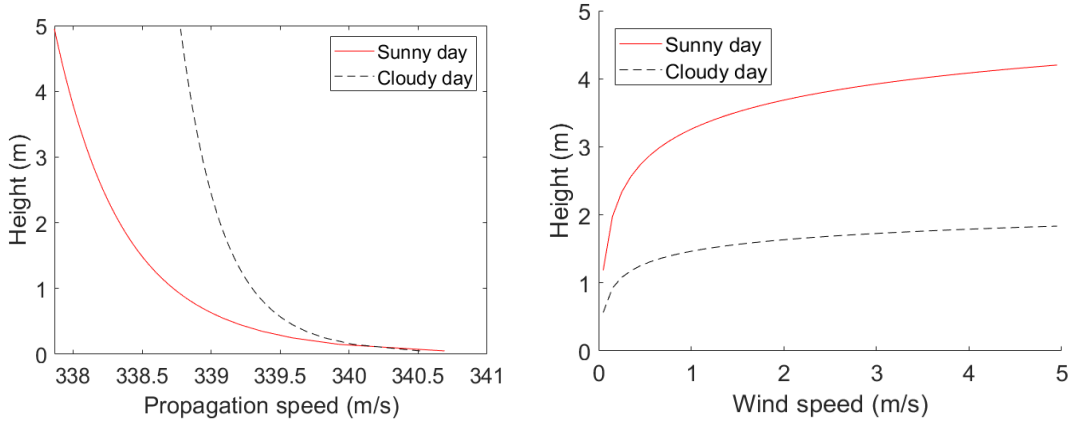
The code uses several standard models to achieve a new simulation capability. It uses models to simulate propagation speed profiles, wind speed and direction profiles, ground reflection, and turbulence to predict the relative phase and amplitude of a received sinusoidal signal. The wind speed and temperature profiles are modeled using Monin–Obukhov (M-O) similarity theory (Monin and Yaglom 1979). The propagation of the signal from a source is simulated using a ray tracing approach through a stratified atmosphere as described by Pierce (1991) and implemented using finite-difference, time-domain (FDTD) method. The ground reflection is modeled using either a one-parameter model developed by Delany and Bazley (1970) or a two-parameter ground impedance model developed by Attenborough et al. (2011). The spatial coherence of the signal at the array is

modeled using a statistical approach developed by Kozick et al. (Kozick and Sadler 2004; Sadler et al. 2004).

3. Wind and Temperature Profiles

A standard approach for estimating temperature and wind speed profiles is to use M-O similarity theory (Monin and Obukhov 1954). It characterizes variations in the surface layer of the atmosphere as a function of a normalized length parameter. It uses the wind speed and temperature near the ground level to estimate profiles up to approximately a height of 100 m (Monin and Yaglom 1979). The implementation of the model is described by Attenborough et al. (2007). The propagation speed is calculated from the temperature profiles using the ideal-gas law. The propagation speed is proportional to the square root of the temperature measured in kelvin. The wind direction is simulated using a uniform distribution over the interval $[0,2\pi]$ radians. A slightly more complex model that was considered but not used is to make the wind direction a Gauss–Markov random process. Its time-constant could be selected based upon the length parameter in M-O similarity theory. This model may be added in the future to the simulation.

The code used to implement the M-O model was written by Ostashev et al. (2008). Typical values for the input parameters to the model are described in another publication by Ostashev et al. (2012). Two realizations of the simulations are shown in Fig. 1. Their input parameters are reference temperature = 288 K, roughness height of the ground surface = 0.01 m, friction velocity = 0.3 m/s, and reference surface sensible heat flux = 200 for sunny or 40 for cloudy conditions, respectively. The results show the largest changes occur near the ground. Monte Carlo simulations can be run by randomly perturbing these input parameters. These profiles are input to a FDTD-based ray tracing code.



a. Simulated propagation speed. b. Simulated wind speed.

Fig. 1 Typical profiles calculated using M-O similarity theory calculated for sunny and cloudy conditions

4. FDTD Simulation

There are many techniques to propagate acoustic signal through a moving atmosphere. For a layered atmosphere, propagating waves can be modeled using the inhomogeneous Helmholtz equation. If atmosphere is axisymmetric with respect to the source (Solomon), the resulting differential equations can be solved using a generalized fast field program (FFP). If there is no axisymmetry, a less-restrictive model can be calculated using the Eikonal equation, which assumes the propagation speed is an arbitrary function of the geometry (Officer 1958). It is a high-frequency approximation that is valid if the fractional change in the velocity gradient is much less than the wave frequency of the source. The resulting equations are described by Pierce as shown:

$$\frac{dx}{dt} = \frac{c^2 s}{\Omega} + \mathrm{v} \tag{1}$$

$$\frac{ds}{dt} = -\frac{\Omega}{c}\nabla c - s\,\mathbf{x}\,(\nabla\,\mathbf{x}\,\mathrm{v}) - (s\cdot\nabla)\mathrm{v} \tag{2}$$

$$\Omega = 1 - s\cdot\mathrm{v} \tag{3}$$

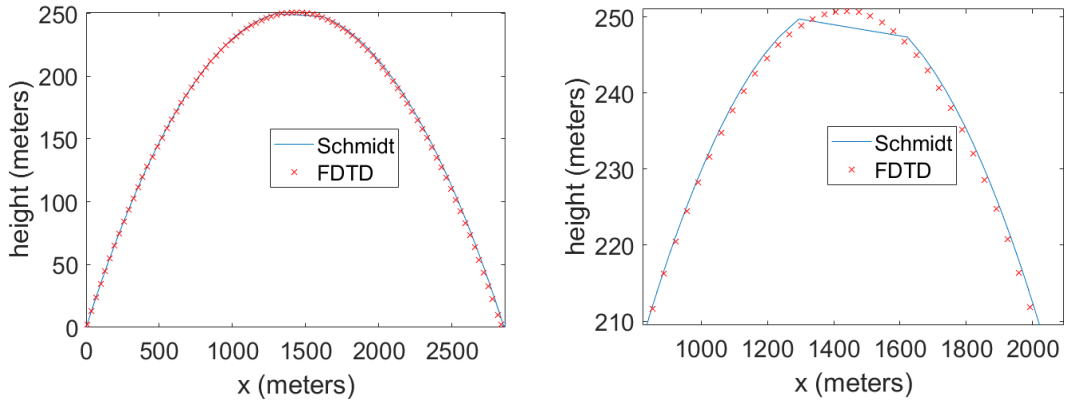
where x is a moving point that lies on the wave front of the pressure wave; c is the propagation speed that depends on height; v is velocity of the wind; s is the wave-slowness vector, which is parallel to the wave front normal vector; the operator **x** denotes cross product; and the operator ∇ denotes the del or nabla operator used in vector calculus (Pierce 1991). For a layered atmosphere, the wind and propagation speed are assumed to be constant within a layer, so its derivatives with respect to the x and y coordinates are zero. This simplifies Eq. 2 to

$$\frac{ds}{dt} = -\frac{\Omega}{c}\frac{dx}{dz} - s_x\frac{dv_x}{dz} - s_y\frac{dv_y}{dz} \ . \tag{4}$$

The solution to these equations is coded in MATLAB using a backward Euler FDTD method as shown in Appendix A. The main input parameters are the initial angle of departure of the ray, wind and temperature profiles, the FDTD time step size, and two heights where the time step size is modified. The changing time step is a desirable feature because of the larger potential errors caused by large temperature and wind changes near the surface. The FDTD simulation assumes the ground is flat and located at z = 0.

The output of the FDTD simulation is three arrays indexed by the time iteration number in the simulation. The arrays contain the 3D location of a ray, slowness vector, and time step size. The relative phase of a sinusoidal signal along the ray path is equal to $\Omega = \omega t(z)$, where ω is the frequency and *t(z)* is the total time for the ray at height z. The propagation of the rays is independent of frequency and no amplitude information is calculated. The DOA for a ray is usually slightly different from the direction normal to the propagating wavefront.

The output of the FDTD code is compared to the results for a ray tracing code written by Val Schmidt from the Center for Coastal and Ocean Mapping/Joint Hydrographic Center at the University of New Hampshire in 2009. The code propagates a ray through a layered median with different propagation speeds. It is written in MATLAB and available for download at the Mathworks Central File Exchange website (Schmidt 2020). The default values for the parameters in this code were used to simulate rays that are compared to the results from the FDTD code. The input to both of the codes is an initial elevation angle of 20°, azimuth angle of 90°, and a linear sound speed profile starting at a speed of 330 m/s at z = 0 m and ending at a speed of 360 m/s at z = 356.6 m. Figure 2 shows the path of a ray calculated using the two codes. Note the elevation angle appears to be larger than the simulated value due to different scaling factors on the x- and y-axis.



a. Full trajectory b. Zoomed in trajectory

Fig. 2 Simulated trajectories of a ray. The red "x" shows outputs from the FDTD code and the blue curve shows outputs from Schimdt's code

These results indicate that the codes generate similar output. The zoomed-in results indicate that there are some small differences, probably due to the implementation of Schmidt's code, which steps through layers, not time, and therefore has higher errors when the ray is nearly parallel to the xy-plane. No attempt is made to reduce this error by modifying the parameters in the code. Overall, the results are very similar, even though two different techniques are used to generate the results.

The FDTD code also simulates the effect of wind on the propagation of rays. To check the code for errors, a wind profile is generated with a direction that is initially perpendicular to the direction of propagation of a ray with a constant velocity as a function of height and a linear propagation speed profile. The wind should push the ray in the x-direction by approximately the wind speed multiplied by the time for the ray to intersect the ground (shift = velocity × time). The results of the simulation are shown in Fig. 3 for a constant wind speed profile of 1 m/s in the direction of the y-axis. Both the analytically predicted shift and the simulated shift on the x-axis are 5.56 m. Identical magnitudes were obtained for a wind direction along the x-axis. These results indicates there is good agreement between the predicted and simulated result for a constant wind profile.

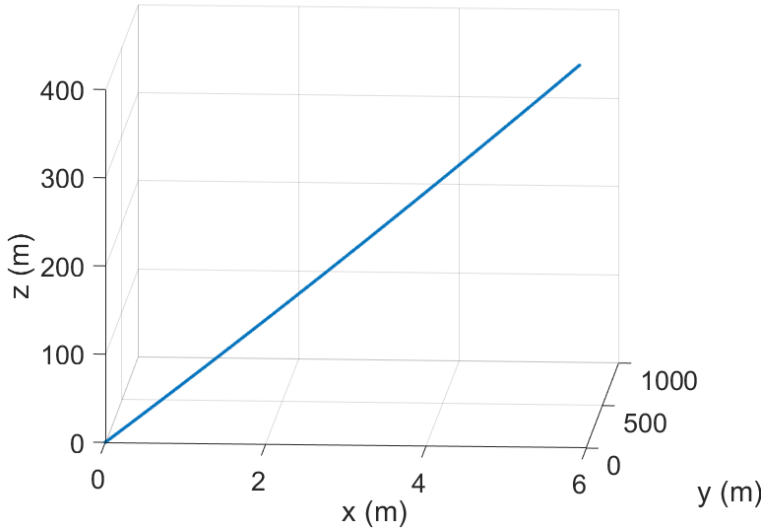


Fig. 3 Simulated ray with constant wind speed that is initially perpendicular to the ray. The expected and simulated offset on the x-axis is 5.56 m.

5. Ground Bounce

The acoustic signal reflected by the ground is modeled as the superposition of a direct path and an indirect path with a single ground reflection. For most scenarios, the differential spherical attenuation lost between the direct and indirect path is small and can be ignored (Salomons 2001). The ground reflection coefficients are calculated using locally reactive ground impedance models. For these models, the ground impedance is independent of aspect angle. In general, models with a higher number of parameter result in better agreement with experimental measurements. However, the values of the parameters are often unknown and can be difficult to estimate in an uncontrolled environment. For this reason, only 1- and 2-parameter models are implemented in the simulation.

Delany and Bazley (1970) developed a 1-parameter model and Attenborough et al. (2011) developed a 2-parameter model. The code for these models is listed in Appendix B. The input to Delany and Bazley model is frequency and flow resistivity with typical values for grass between 100 and 300 KPasm^{-2}. This model is valid for frequency/flow resistivity ratios of 10^{-4} to 10^{-1} m^{3}kg^{-1} (Salomons 2001). The input to the 2-parameter model is flow resistivity and the effective rate of change of porosity with respect to depth. The output of both of these models is normalized acoustic impedance that is independent of frequency and aspect angle. The complex reflection coefficient can now be calculated using

$$\rho(\theta) = \frac{Z\cos(\theta)-1}{Z\cos(\theta)+1} \tag{5}$$

where Z is the normalized acoustic impedance and θ is the aspect angle of a ray that approaches $\pi/2$ at grazing angle. At grazing angles, $\rho(\theta)$ approaches –1 and the indirect signal cancels the direct signal. Measurements indicate that there is some reduction in signal strength, but less than the predicted level. This model is not valid near grazing angles (Salomons 2001).

For a source in the far field, the pressure at a microphone for a narrowband signal received near the ground over a homogenous surface is modeled using

$$P(\theta) = real\left(p_c(\theta)\left(1 + \rho e^{j\varphi}\right)\right) \tag{6}$$

where $real(\cdot)$ denotes the real value operator, $p_c(\theta)$ is the complex pressure from the direct path signal, φ is the additional phase due to the path difference between the direct path and indirect path, and j is a complex number (Salomons 2001).

The additional phase shift term in Eq. 6 can be estimated using the propagation time output by the ray tracing simulations. Since the propagation and wind speed profiles do not change as a function of the x and y coordinate, there is symmetry between the downward path and the upward path for a propagating ray, as shown by the blue curves in Fig. 4. To simplify the processing of the ground bounce signal, the method of images can be used to represent the ground bounce path using a virtual microphone, as illustrated by the red dot in Fig. 4. The simulated raytracing paths can be extended to negative z values using symmetry, as illustrated by the red line in Fig. 4. Now, the propagation time to a virtual microphone can be estimated using interpolation with a bundle of closely spaced rays.

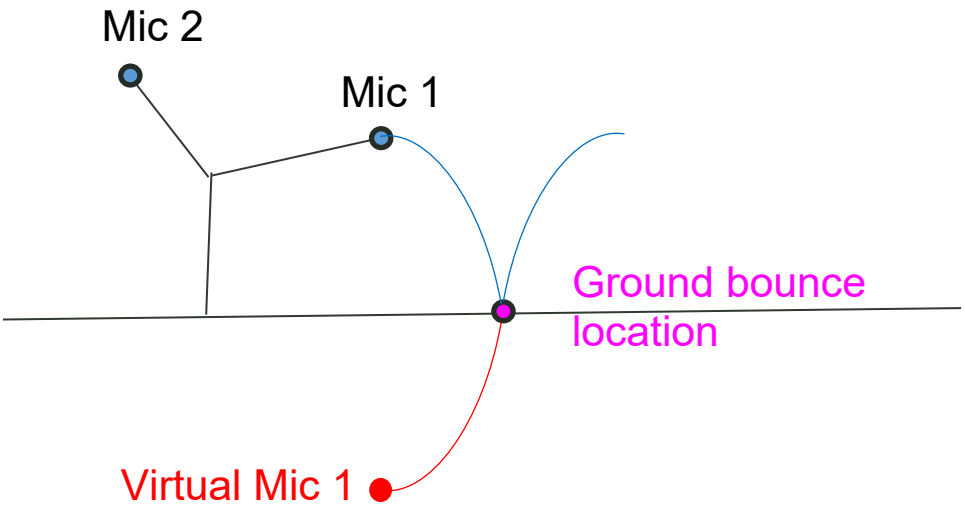
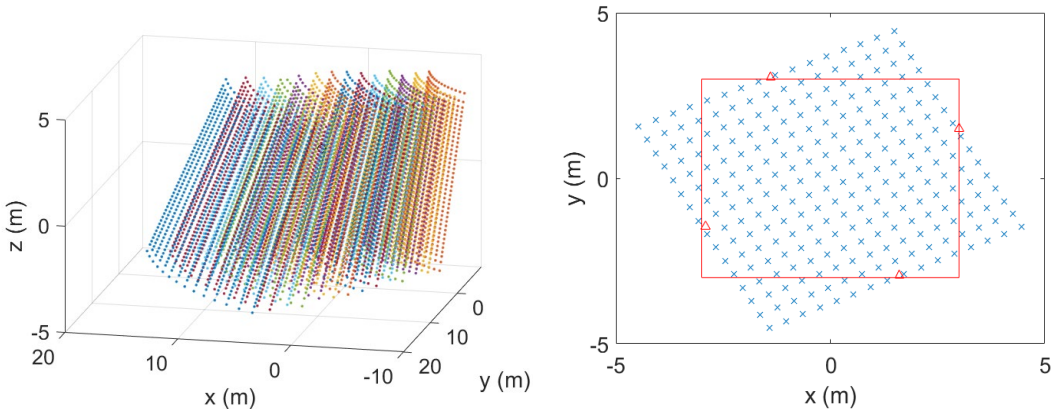


Fig. 4 Indirect path of a ray and its mirror image for microphone 1

Figure 5a shows a bundle of closely spaced rays that has been extended to negative values of z and Fig. 5b shows their intersection with the xy-plane at z = 0. To interpolate the propagation time, the real or virtual microphones should be located within the interior of the bundle of rays. The ray bundles are generated by launching rays from a grid of launch angles with values between ($\varphi_{center} \pm \Delta\varphi$, $\theta_{center} \pm \Delta\theta$), where $\Delta\varphi$ and $\Delta\theta$ describe the maximum change in the polar angles with respect to the center angles. The grid is constructed with an equal number of angles in φ and θ with linear spacing. For a tightly spaced grid, the difference in adjacent ray paths is small. However, the intersection of the rays on the xy-plane is not on a grid with equal spacing, as shown in Fig. 5b.



a. Cluster of real (z > 0) and virtual (z < 0) rays.

b. Intersection of the rays with the ground with respect to the center ray at z = 0.

Fig. 5 Simulated rays from a source at a range of 500 m

An algorithm was developed to calculate the grid angles associated with bundle of rays. The input to the algorithm is the center launch angle, the number of elements in the grid, and the desired xy-extent of the rays denoted by Δx and Δy at z = 0. The output of the software is $\Delta\varphi$ and $\Delta\theta$, the maximum angles of departure for the cluster of rays with respect to the center ray. For good interpolation results, the xy-extent of the rays should be larger than the xy-extent of the microphone array. The optimal xy-extent is not obvious, but the user can always make the grid larger so that the microphones are contained within the ray bundle. The code for the algorithm is listed in Appendix C. A block diagram describing the algorithm is shown in Fig. 6.

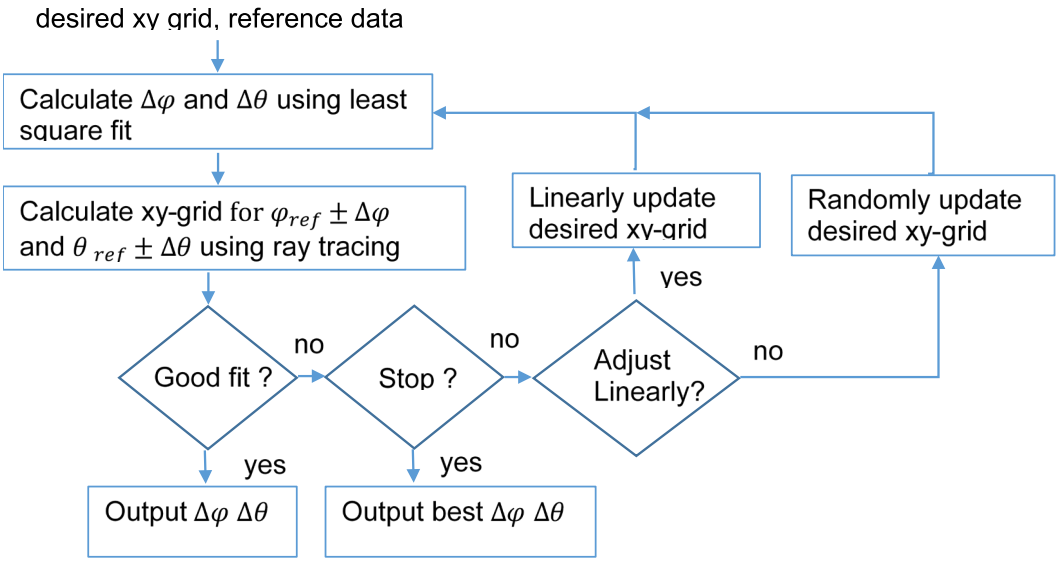


Fig. 6 Block diagram for calculating the cluster of rays

The algorithm calculates the maximum angle of departure for a cluster of rays using a deterministic or a random search. The main inputs to the algorithm are the polar angles of the center ray at z = 0, its range, and the desired xy-extent at z = 0. First, a least-squares fit is performed to estimate the maximum change in the polar angles, $\Delta\varphi$ and $\Delta\theta$, of the departure angles of the rays. The relationship between the Cartesian and spherical coordinates (Wikipedia 2020b) is given by $x = Rsin(\theta)cos(\varphi)$ and $y = Rsin(\theta)sin(\varphi)$, where R is the range. These equations can be linearized by performing a first-order Taylor series approximation evaluated at the center angle of the ray evaluated at z=0 as shown:

$$\begin{bmatrix}\Delta x\\ \Delta y\end{bmatrix} = \begin{bmatrix}\begin{vmatrix}\frac{dx}{d\varphi} & \frac{dx}{d\theta}\\ \frac{dy}{d\varphi} & \frac{dy}{d\theta}\end{vmatrix}\end{bmatrix}\begin{bmatrix}\Delta\varphi\\ \Delta\theta\end{bmatrix} \tag{7}$$

where Δx and Δy are the Cartesian coordinates of the desired maximum deviation of cluster of rays intersecting the ground, the matrix represents the linearized equations, and $\Delta\varphi$ and $\Delta\theta$ are the estimated differential angles of departure. There are known errors associated with this approximation. The angles $\Delta\varphi$ and $\Delta\theta$ are estimated at the source location while Δx and Δy are specified on the ground. The following procedure addresses this issue.

The solution obtained from Eq. 7 is checked by performing ray tracing at the four departure angles given by ($\varphi_{center} \pm \Delta\varphi$, $\theta_{center} \pm \Delta\theta$) and evaluating the error. The red triangles in Fig. 5b show an examples of the position of the rays at z = 0. The desired xy-extent of the ray is shown by the red rectangle in Fig. 5b. The

function that quantifies the error between the desired and actual location of these rays is

$$Q=\left|\left|\frac{\max(x_i)-\min(x_i)}{2\Delta x}\right|-1\right|+\left|\left|\frac{\max(y_i)-\min(y_i)}{2\Delta y}\right|-1\right| \tag{8}$$

where x_i and y_i are arrays containing four points indexed by i. If the error value computed using Eq. 8 is above a threshold, then the value of Δx and Δy used in Eq. 7 is updated using

$$\Delta\acute{x} = \Delta x\frac{\max(x_i)-\min(x_i)}{2\Delta x} \tag{9}$$

$$\Delta\acute{y} = \Delta y\frac{\max(y_i)-\min(y_i)}{2\Delta y} \tag{10}$$

and $\Delta\varphi$ and $\Delta\theta$ is recalculated. This sequence of steps is performed until the error is below a threshold or the number of iterations exceeds a number. If the maximum number of iterations is exceeded and the error is still above a threshold, then a search with random perturbations is initiated.

A random search may provide better solutions when the Taylor series approximation is inaccurate. Random Gaussian perturbations of Δx and Δy centered around their initial value are performed, then $\Delta\varphi$ and $\Delta\theta$ are recalculated and the error function in Eq. 8 is evaluated. This sequence of steps is performed until the error is below a threshold or the number of iterations exceeds a number. If the error is still above the threshold, then the $\Delta\varphi$ and $\Delta\theta$ values that resulted in the smallest error value are selected.

6. 3D Interpolation

The propagation time of the source signal to the microphones can be calculated with 3D interpolation using the cluster of ray paths and their associated propagation times. A block diagram of the algorithm is shown in Fig. 7. The algorithm is based upon 1D linear interpolation along the z-axis followed by 2D interpolation in the xy-plane. First, each ray and its associated propagation time array is resampled at indices that correspond to fixed values of z using the interp1 function in MATLAB. This is possible since it is required that all the rays hit the ground. Next, each ray and its associated propagation time array is interpolated along the z-axis at the z-coordinate of the selected microphone using linear 1D interpolation. The output for each ray is an xy-point and a propagation time. Next, 2D interpolation is performed.

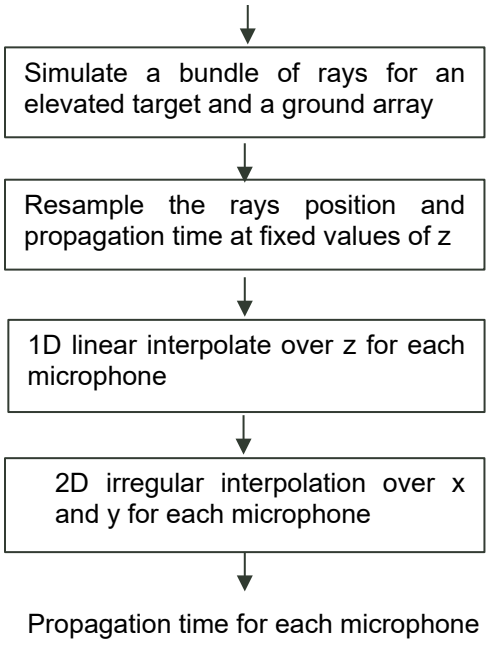


Fig. 7 Block diagram of the 3D interpolation algorithm

The 2D interpolation algorithm is based on either bilinear interpolation using four points (Appendix D) or triangular-based interpolation using three points (Appendix E). The block diagram for the algorithm is shown in Fig. 8. The algorithm uses four points for interpolation if the desired location for interpolation is in the center of the points. Otherwise, it uses three points. The assumption is four points are better than three points for interpolation, which is not always true. The input to the algorithm is the results from the previously described 1D interpolation. First, a grid point that is close to the xy-coordinate of the selected microphone is determined using a Euclidean metric. Initially, the closest grid point is selected. If multiple iterations of the algorithm are required, then the next closest grid point is selected. Next, the four closest xy-grid points that surround the desired xy-location are determined by increment and decrement in the angle indices relative to the indices of the selected grid point. If these four grid points have exactly two x values and two y values that are above and below the desired microphone position, then bilinear interpolation is performed; otherwise, three points are used. The bilinear interpolation algorithms available in MATLAB require that the values are on a uniformly spaced grid, but this requirement is ignored. The points used in the bilinear interpolation are ordered so that the initial linear interpolation in the x-coordinate corresponds to the pair of points with the largest change in x values.

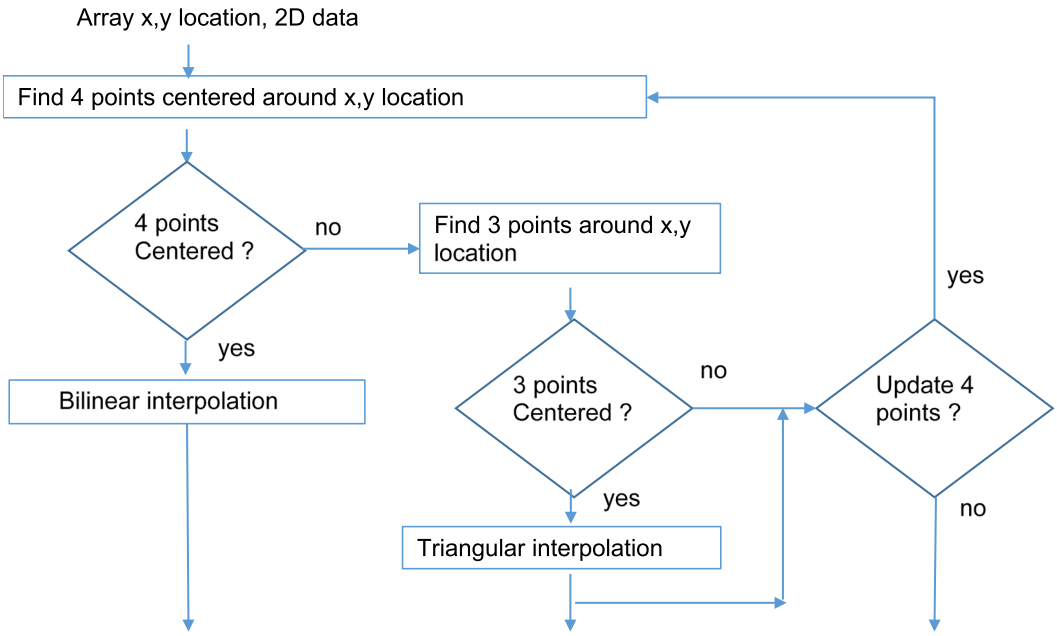


Fig. 8 Block diagram for 3D interpolation

Interpolation with three points is performed if bilinear interpolation is not possible due to a poor distribution of the location of the four selected grid points. Figure 9 illustrates interpolation using three points. First, the same initial grid point used in bilinear interpolation is selected, as shown by the red triangle labeled ray 2 position. Next, a line is draw from initial grid point to the position of the microphone, as illustrated by the blue line. Then, the line between the two remaining ray positions is computed, as illustrated by the red line. Now, the propagation time at the intersection of the two lines, as shown by the blue diamond is computed using linear interpolation. Next, the propagation time at the microphone xy-position, illustrated by the blue circle, is computed using linear interpolation between the intersection of the lines shown by the blue diamond and triangle labeled ray 2 position.

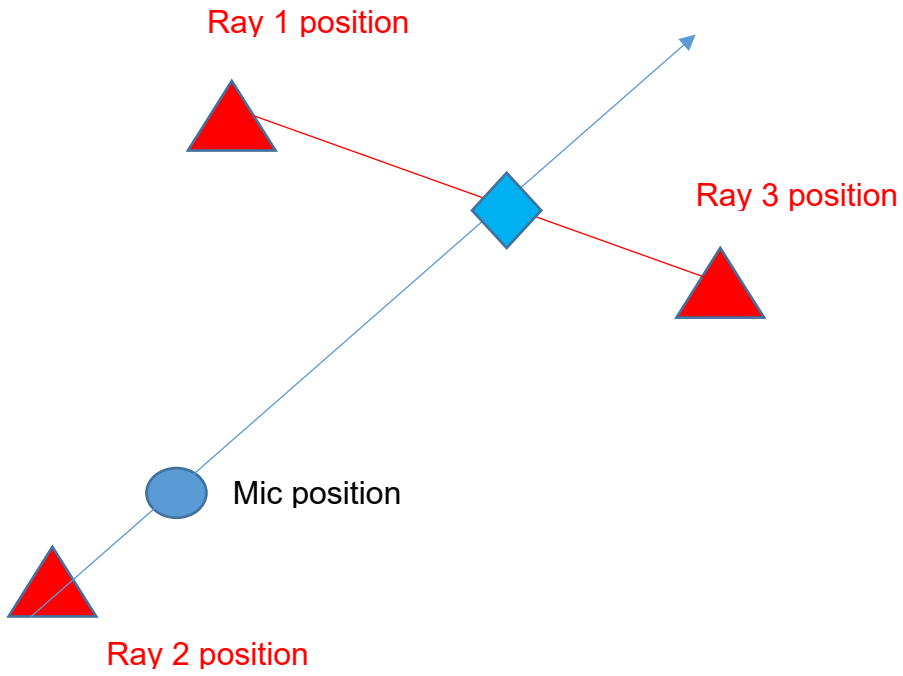


Fig. 1 Interpolation of the propagation time using three points

7. Turbulence

Turbulence can cause the received signal at a microphone array to spatially decorrelate. Many models exist for simulating the effects of turbulence on acoustic signals. This simulation uses a simple model developed by Kozick et al. (2003) for describing turbulence. Basically, the model assumes the spatial correlation between the microphones is dependent upon their relative distance and the frequency of the signal. More precisely, the complex envelope of a narrowband signal measured by two microphones can be modeled with a complex Gaussian distribution that decorrelates as a function of range and cross range using

$$Z' \sim CN(e^{j\chi}\sqrt{1-\Omega}a, \Omega S\Gamma \circ (aa^H) + \sigma^2 I) \tag{11}$$

where CN denotes complex normal or Gaussian distribution, χ is the phase associated with a source signal and range, $\circ$ denotes element-wise product, S is the average power of the source, σ^2 is the average power of the noise, $\Omega \in [0,1]$ represents saturation ($\Omega >> 0$ strong scattering), σ^2 is the power of additive noise, and I is the identity matrix,

$$a = \begin{bmatrix} 1 \\ e^{j\phi} \end{bmatrix},$$

$$\Gamma = \begin{bmatrix} 1 & \gamma \\ \gamma & 1 \end{bmatrix},$$

$\gamma \epsilon\ [0,1]$ is the spatial coherence between two microphones and Φ is an angle related to the DOA of the signal. The remained terms in Eq. 11 are

$$\Omega = 1 - e^{-2ud_0},$$

$\underline{u}$ is a constant and d_0 is the range from the source to the microphone,

$$\gamma = \frac{e^{-vd_0} - (1 - \Omega)}{\Omega},$$

$$v = \kappa(weather)\omega^2 \rho^{\frac{5}{3}},$$

$$\kappa = \frac{0.137}{c_0^2}\left(\frac{C_T^2}{T_0^2} + \frac{22}{3}\frac{C_v^2}{c_0^2}\right)$$

ω is frequency and ρ is the spacing between the sensor (Kozick et al. 2003; Sadler et al. 2004).

When the scattering due to turbulence is weak, Ω << 1, the spatial coherence is approximately equal to $e^{-\kappa\omega^2\rho^{\frac{5}{3}}d_0}$. The frequency term in this expression is squared, while the microphone separation term is to the power of 5/3. This indicates that the frequency has a slightly larger impact than separation distance in calculating the spatial coherence.

The model can be extended to use more real and virtual microphones by using a larger mean vector and covariance matrix in Eq. 11. The range to the source, denoted by d_0, for the indirect ray is slightly different from the range of the direct ray, but these differences are ignored in the simulation and only the average range from the direct path is used in the turbulence calculations.

The ray tracing software does not calculate the absolute amplitude of the simulated signal. The divergence of the rays is not estimated and atmospheric loses are not calculated. The motivation for this decision is that the signal-to-noise ratio (SNR) and SINR are more important than the absolute signal level for many signal processing applications. In addition, often the signal strength of the source is not known. This information provides motivation for normalizing the amplitude of the mean value in Eq. 11 to one, then adjusting the covariance matrix and temporarily removing the contribution due to the independent and identically distributed (iid) noise as shown:

$$CN\left(e^{j\chi}a, \frac{\Omega S}{1-\Omega}\Gamma \circ (aa^H)\right) \tag{12}$$

For passive acoustic signal processing algorithms, the absolute phase of the signal is almost never known, only the relative phase. If Z is circularly symmetric Gaussian distribution, then

$$E(Z) = E(Z)e^{j\phi} \tag{13}$$

where E denotes expected value and ϕ is a random variable with values between [-π,π]. This is consistent with the assumptions made in most acoustic applications. If the distribution is a circularly symmetric complex Gaussian (Wikipedia 2020a), then the mean is zero. This requirement can be achieved by subtracting the mean from Z_n in Eq. 12 and define a new random variable Z as shown:

$$\mathrm{Z}{\sim}CN\left(0, \frac{\Omega S}{1-\Omega}\Gamma \circ (aa^H)\right). \tag{14}$$

The distance parameters in the turbulence model need to be determined for arbitrary geometries with ground reflection. The distance between the direct and indirect rays used for estimating the effect of turbulence are illustrated in Fig. 10. The red curves show the paths for the direct and indirect rays intersecting microphone 1 and the direct ray intersection the center of the microphone array, shown by the green triangle. The blue line shows a tangent line to the DOA at the center of the array. The distance between the direct and indirect ray, shown by the dotted purple line for microphone 1, is calculated by finding the intersection on a line starting at microphone 1 with a direction that is perpendicular to the line tangent to the DOA with the xy-plane at z = 0. This is not an exact estimate, but many approximations have already been made in the turbulence model, so additional approximations with small errors should not have a material impact. For many scenarios, this approximation is not appropriate for estimating the propagation time for the source signal. For high-frequency signals, small time or phase errors can have a big impact on DOA algorithms.

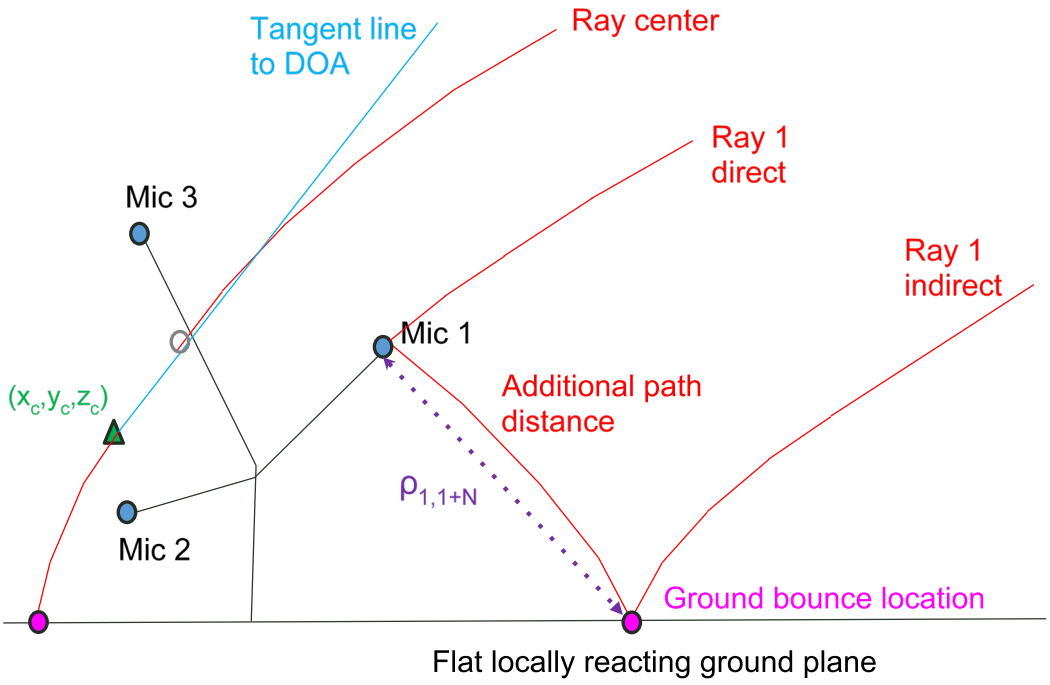


Fig. 10 Geometry for calculating the distance between the direct and indirect rays

Once the position of the ground reflection is estimated, the microphone and ground bounce positions are rotated by θ, the polar angle of arrival about the DOA-axis calculated at the center of the microphone array. This rotation is illustrated in Fig. 11. The distance used for turbulence calculations relative to the first microphone is shown by the horizontal dotted purple lines. The Euclidean xy-distance between the i^{th} and j^{th} microphones or ground bounce location is calculated in the new coordinate system using

$$\rho_{i,j} = \left\| (x'_i, y'_i) - (x'_j, y'_j) \right\| \tag{15}$$

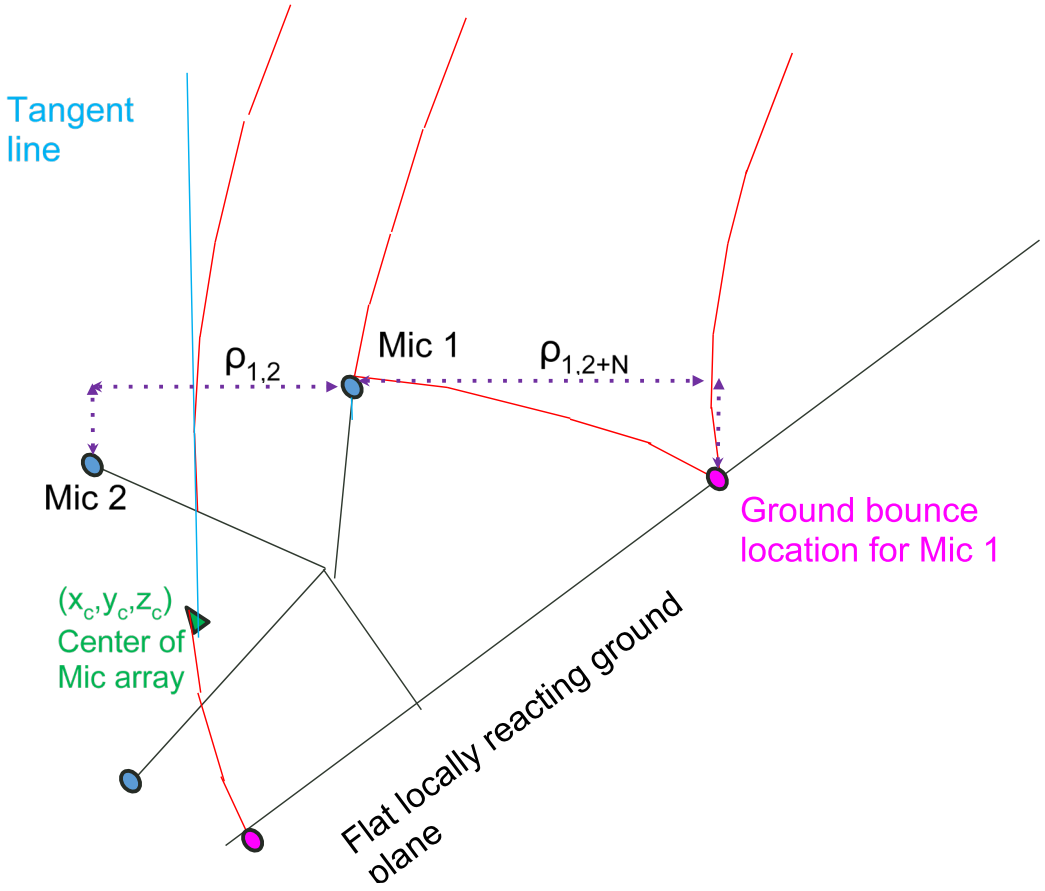


Fig. 11 Geometry for rays and microphone array rotated about the DOA at the center of the array and the resulting distance used to estimate turbulence

A standard approach is used to generate random complex samples to simulate the effect of turbulence. A complex multivariate Gaussian distribution can be represented by

$$Z \sim AW \tag{16}$$

where $W \sim CN(0, I_m)$, is a matrix that represents a linear transformation and I_m is the identity matrix with dimension m (Wikipedia 2020c). The covariance matrix associated with Z is given by

$$C_z = AA^H. \tag{17}$$

Now, the linear transformation, A, can be calculated using the Cholesky decomposition

$$R_z^H R_z = Chol(C_z) \tag{18}$$

where R_z is a complex upper triangular matrix. Now, a circular symmetric complex Gaussian distribution can be simulated using the results from Eqs. 16 and 18 and a standard Gaussian random number generator.

For a circular symmetric complex Gaussian distribution, the complex components can also be described using only real values with

$$\begin{bmatrix} Z_{real} \\ Z_{imag} \end{bmatrix} = \begin{bmatrix} Real(A) & -Imag(A) \\ Imag(A) & Real(A) \end{bmatrix} \begin{bmatrix} W_{real} \\ W_{imag} \end{bmatrix} = A_{big} W_{big} \qquad (19)$$

where the subscript big denotes real variables that are constructed by concatenating together the real and imaginary components (Gallager 2008), where samples for W_{big} are generated using samples from a standard 1D Gaussian distribution. Now, realizations of the complex random variables can be calculated using

$$z = e^{j\chi} a + z_{real} + \mathrm{j} z_{imag} \, . \qquad (20)$$

Next, the signal at each microphone is computed by adding the signals from the real and virtual microphone pairs as shown:

$$Z_\Lambda = \Lambda \mathrm{Z} \qquad (21)$$

where Λ is a $N \, x \, 2N$ matrix, Z is $2N \, x \, 1$ vector, and N is the number of microphones in the array. For a two-element microphone array, Λ is given by

$$\Lambda = \begin{bmatrix} 1 & 0 & \rho & 0 \\ 0 & 1 & 0 & \rho \end{bmatrix} \qquad (22)$$

The estimated covariance matrix of the measured signal is calculated by adding Gaussian white noise as shown:

$$C_\chi = \Lambda C_z \Lambda^T + \sigma I_N \qquad (23)$$

where σ is a scalar factor and I_N denotes an NxN identity matrix.

One potential issue for simulating turbulence is the Cholesky decomposition requires that the input matrix is symmetric positive semi-definite. This assumption can be violated due to symmetries in the locations of the microphones. Singularities in the covariance matrix can be eliminated using a standard approach described by Rebonato and Jäckel (2000).

The MATLAB function cholcov corrects for singular covariance matrices, then performs a Cholesky decomposition. First, the eigenvalues and eigenvectors of the covariance matrix are calculated. Then, the negative and small eigenvalues that are below a threshold are eliminated. This method does not preserve the trace of the

covariance matrix, but it should have a minimal impact on signal processing algorithms if the threshold is much smaller than the average eigenvalue.

8. Test Results

All of the models in the simulation, except the interpolation routines, have been previously tested and validated. So, only the interpolation code was testing for performance, while the other models were tested for functionality. The interpolation algorithms were tested by comparing theoretical propagation time values to interpolated values calculated in a homogenous atmosphere. Rays in a homogenous environment travel in a straight line, which enable simple calculations to determine the theoretical propagation times.

Two simulations were run to test the accuracy of the interpolation algorithms and the consistency of the ray tracing and ray bundling algorithms. The first test simulated rays on a predetermined grid of launch angles and the second test calculated the angles for the bundle of rays. The simulated time step in the FDTD simulation is dependent on the height on the ray. At heights above 4 m, the step size is 0.006 s and propagates approximately 2 m per increment. At heights between 2.2 and 4 m, the step size is 0.002 s and, at heights below 2.2 m, the step size is 0.001 s. Testing is performed on a predetermined grid and for an estimated grid of launch angles for the bundle of rays.

The first test simulates a bundle of rays radiating from a source at a location of (0, 0, 500) m with launch angles between $\varphi = [0,2]$ degrees and $\theta = [0,2]$ degrees with varying levels of angular resolution propagating toward an array of microphones. Figure 5 shows an example of a bundle of rays for z values between [-2,2]. The simulated location of the microphones that are shown in Table 1. The x-coordinate of the first four microphones are incremented while the z-coordinate of the last three microphones are incremented. Figure 12 shows the intersection of the rays with the xy-plane at z = 0 shown by the blue dots and the position of the first four microphones in Table 1 are shown by the red x's.

Table 1 Simulated location of seven microphones

Microphone position (m)
(-4.21, -0.0601, 0)
(-7.61, -0.0601, 0)
(-11.81, -0.0601, 0)
(-17.11, -0.0601, 0)
(-4.21, -0.0601, 0.41)
(-4.21, -0.0601, 0.62)
(-4.21, -0.0601, 1.0)

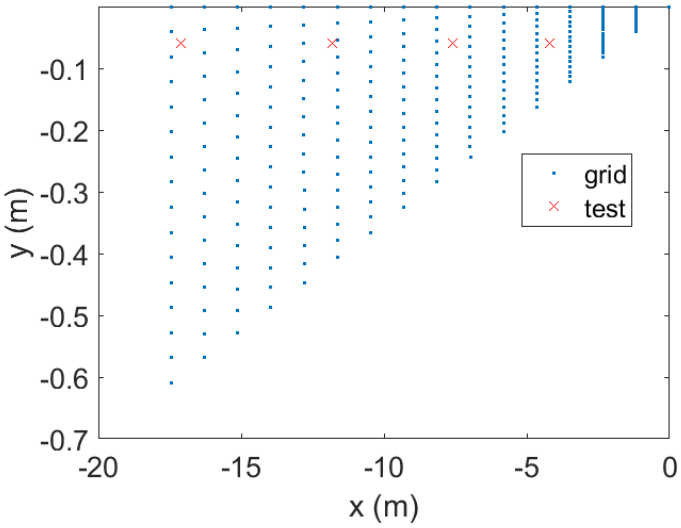


Fig. 12 Cluster of rays intersecting the xy-plane at z = 0 and several test points representing the microphone positions

The results are tested by comparing the propagation time from the source to the microphones using theoretical values based on the geometry and estimated values based on interpolation as shown in Fig. 13. The number of elements used to generate the cluster of rays ranged from 5^2 to 36^2, as shown on the x-axis and the difference between the estimated time and exact time is shown on the y-axis. The numbers in the legend correspond to the microphone number in Table 1. Due to biases in the estimation algorithm, the time error was always positive. To better understand these biases, the differential propagation time as a function of the ray intersection location with the xy-plane at z = 0 is plotted in Fig. 14. It shows that the values have both a linear and nonlinear dependence on the x and y variables. This graph indicates the estimates should be biased, since the simulation is based on linear interpolation and the function being estimated is not linear. The graph also shows that the estimates for the last four microphones with varying z-values were highly correlated.

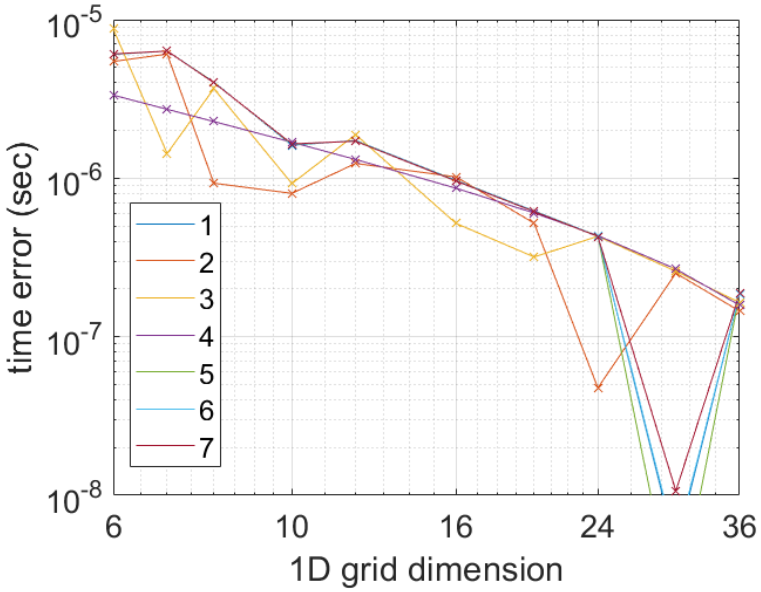


Fig. 13 Error between actual and simulated propagation times

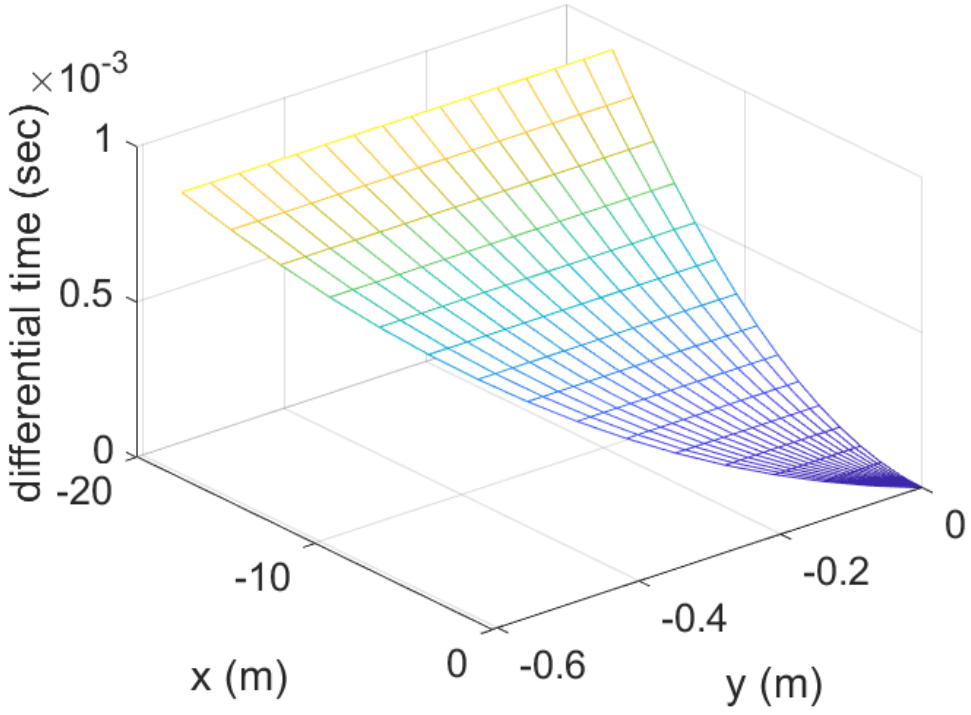


Fig. 14 Simulated differential propagation time in the xy-plane

The results in Fig. 13 are summarized by plotting standard deviation of the time error versus the interpolation grid size (Fig. 15). As expected, the results indicate that interpolation using more closely spaced rays reduces the error.

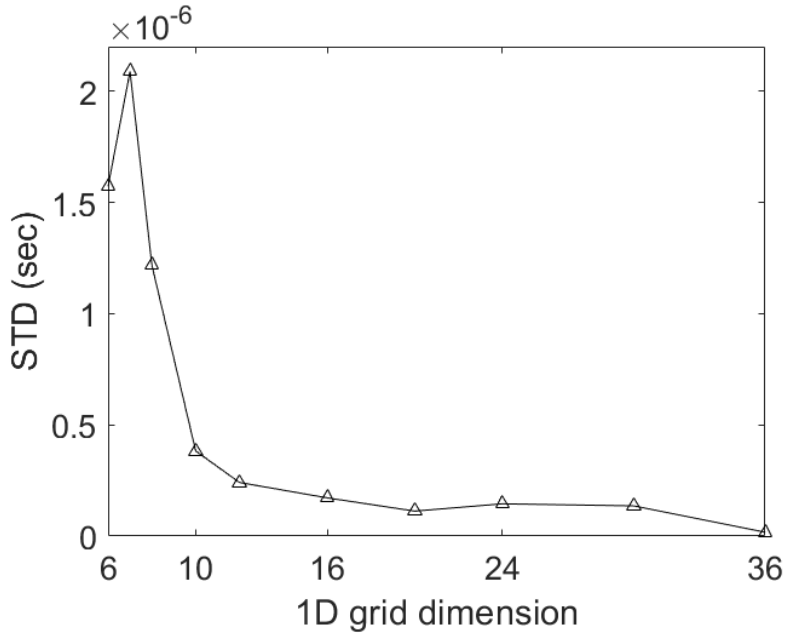


Fig. 15 Standard deviation of the propagation time error

The magnitude of the differential propagation time error is plotted in Fig. 16. For most passive acoustic applications, the absolute time delay of a signal from a source and its initial phase are not important. Beamforming and time difference of arrival (TDOA) algorithms use the differential time or phase to estimate the amplitude and the angular direction of the source. Figure 16 shows the absolute value of the difference between adjacent microphones in the x-coordinate, which is shown by circles, and the z-coordinate, which is shown by an "x". The results show that for this geometry, there is less differential error in interpolating in the z-direction than the x-direction.

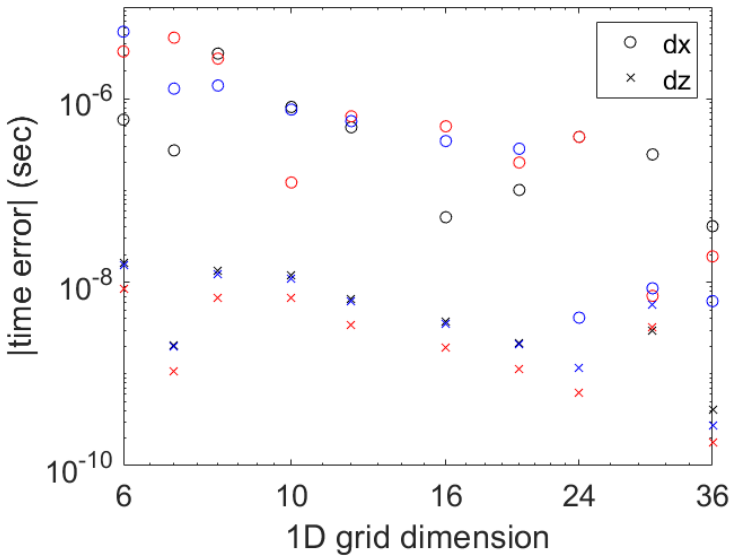


Fig. 16 Absolute value of propagation time error between adjacent microphones

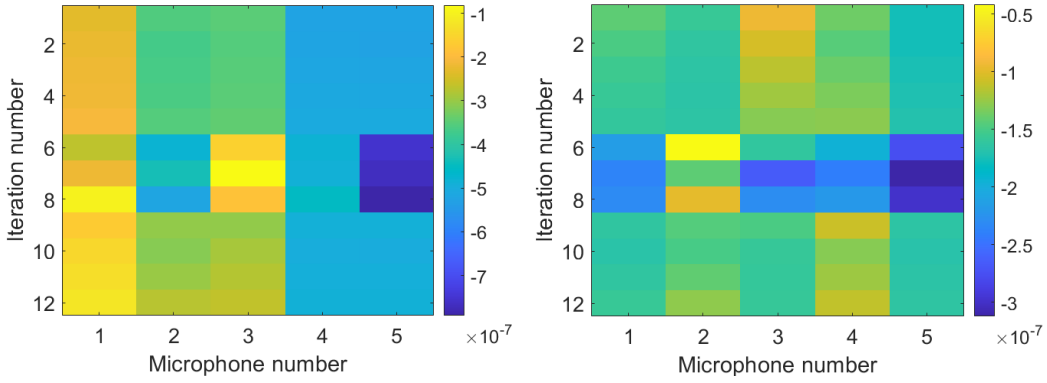
The results for the first test indicate that a standard deviation of the time error of 0.4 μs can be achieved using a 1D grid dimension of 10 or an angular resolution of 0.2° for the simulated geometry. At a frequency of 1 kHz, this corresponds to a phase error of 2.9 milliradians or 0.17°. This error value should be acceptable for most applications. If more accuracy is required, the time increment step in the FDTD simulation can also be decreased.

The interpolation algorithms were also tested using grid angles for the ray bundles that were calculated by the simulation. Rays were launched from a source at an initial height of 500 m with an initial launch angle of $\varphi = 26°$ and $\theta = 30°$ and received by a tetrahedral array with 1-m length arms and with an additional microphone at (0,0,0). The simulated source is moving with a velocity of (1, -1, 0.5) m/s. The rays were specified to intersect a 5- × 5-m area on the ground. The source position was updated every 1 s. The locations of the microphones are shown in Table 2.

Table 2 Simulated position of the microphones

Microphone position (m)
(1,0,0)
(0.5, -0.86,0)
(-0.5, 0.86,0)
(0,0,2)
(0,0,0)

The simulations are run using 10 × 10 and 20 × 20 bundles of rays in a homogenous atmosphere with slightly incrementing source locations. The time propagation errors are shown in Fig. 17. The half-angular coverage of the ray bundle is $\Delta\varphi = 0.0144$ and $\Delta\theta = 0.062$ radian. This corresponds to an angular resolution for 10 × 10 bundle of $\varphi_r = 0.0029$ and $\theta_r = 0.012$ radian or $\varphi_r = 0.165°$ and $\theta_r = 0.71°$. As expected, using more rays results in smaller time errors. The standard deviation of the all the time errors for each ray bundle are 1.6×10^{-7} for 10 × 10 bundle and 4.9×10^{-8} for 20 × 20 bundle. At a frequency of 1 kHz, this corresponds to a phase error of 1.0 and 0.30 milliradians or 0.059° and 0.017°, respectively. This error values should be acceptable for most applications.



a. ray bundle 10 × 10 b. ray bundle 20 × 20

Fig. 17 Time propagation errors associated with interpolation for two bundles of rays propagating in a homogenous atmosphere at slightly different locations

The final output of the simulation is a waveform at each microphone. The simulation generated data for a tetrahedral array using the parameters defined previously with the 20 × 20 bundle of rays. The ground is specified to be grass on a flat surface with a surface roughness of 1 cm, it is a sunny day with a temperature of 23 °C at the ground, the wind speed at a height of 1 m is 2 m/s with a direction of 90°. The source is at a range of 100 m with a frequency of 400 Hz, SNR = 30 dB, azimuth and elevation angle of the source are 35° and 20°, respectively. The results of the simulation are shown in Fig. 18. Each curve represents the waveform received by the four microphones in a tetrahedral array and the legend corresponds to microphone number and corresponding location described in Table 2.

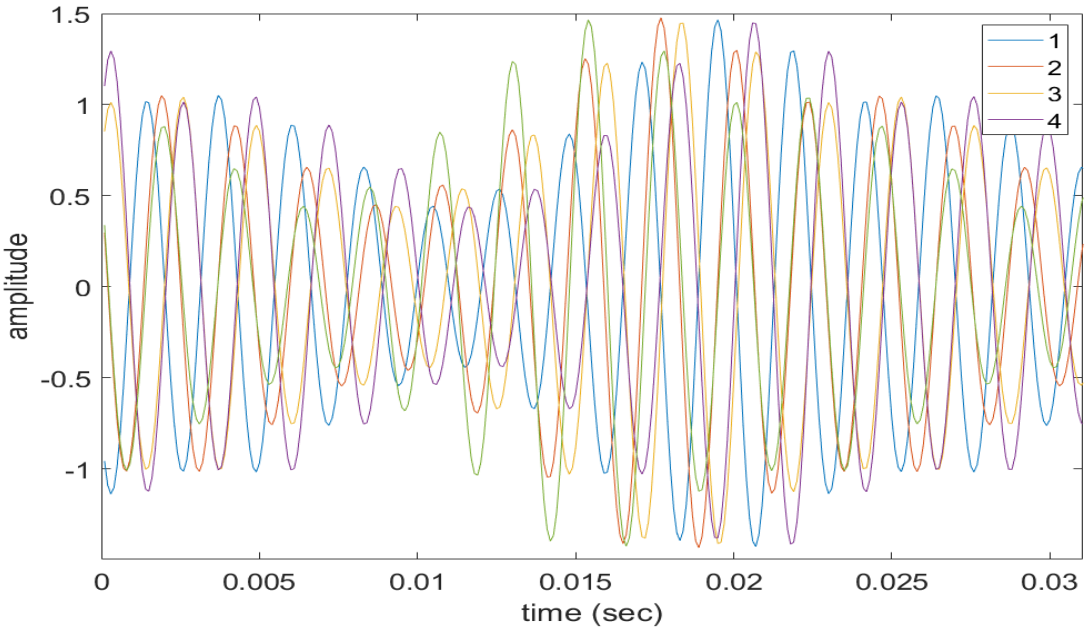


Fig. 18 Simulated waveform on a tetrahedral array of microphones

9. Conclusion

Standard models that incorporate all the significant phenomenology and a custom 3D interpolation algorithm were implemented in MATLAB to simulate the propagation of harmonic signals from elevated sources in the lower audio band to a small microphone array near the ground. Effects from ground reflection, wind and temperature gradients, and turbulence are included in the simulation. The 3D interpolation algorithm was tested by propagating bundles of rays propagating through a homogenous atmosphere and comparing the results to analytical calculations. The analysis indicates that the time errors due to interpolation are small enough for most applications. An algorithm was also developed to calculate the launch angles for a bundle of rays to intersect an area on the ground.

This code is a building block for simulating more complex signals and a tool for testing signal processing algorithms such as beamforming. The interface for the code is still in its developmental stage and not suitable for general distribution. Many of the input parameters to the code require subject-matter expertise to properly select them and the code would benefit from additional algorithms to set them automatically. However, the present code can be used to perform analysis of different signal processing algorithms and microphone array configurations.

10. References

Attenborough K, Li KM, Horoshenkov K. Predicting outdoor sound. New York (NY): Taylor & Francis Group; 2007.

Attenborough K, Bashir I, Taherzadeh S. Outdoor ground impedance models. Journal of the Acoustical Society of America. 2011;129.

Delany ME, Bazley EN. Acoustical properties of fibrous absorbent materials. Applied Acoustics. 1970;3:105–116.

Gallager RG. Principles of digital communication. Cambridge (UK): Cambridge University Press; 2008.

Kozick RJ, Sadler BM. Source localization with distributed sensor arrays and partial spatial coherence. IEEE Transactions on Signal Processing. 2004;52(3):601–616.

Kozick RJ, Sadler BM, Wilson DK. Signal processing for aeroacoustic sensor networks. In: Distributed sensor networks. Boca Raton (FL): CRC Press; 2003.

Monin A.S., Obukhov AM Basic laws of turbulent mixing in the surface layer of the atmosphere. Geophys Inst Acad Sci USSR. 1954;24:163–187.

Monin AS, Yaglom AM. Statistical fluid mechanics: mechanics of turbulence, vol 1. Cambridge (MA): MIT Press; 1979.

Officer C. Introduction to the theory of sound transmission: with applications to the ocean. New York (NY): McGraw-Hill Book Co; 1958.

Ostashev VE, Scanlon MS, Wilson DK, Vecherin SN. Source localization from an elevated acoustic sensor array in a refractive atmosphere. J Acoust Soc Am. 2008;124(6):3413–3420.

Ostashev VE, Cheinet S, Collier SL, Reiff C, Ligon DA, Wilson DK, Noble JM, Alberts II, WCK. Source localization corrections for airborne acoustic platforms based on a climatological assessment of temperature and wind velocity profiles. SPIE Defense, Security, and Sensing; 2012; Baltimore, MD.

Pierce AD. Acoustics: an introduction to its physical principles and applications. Melville (NY): Acoustic Society of America. 1991.

Rebonato R, Jäckel,P. The most general methodology to create a valid correlation matrix for risk management and option pricing purposes. Journal of Risk. 2000;2:17–27.

Sadler BM, Kozick RJ, Collier SL. Algorithms and performance of small baseline acoustic sensor arrays. Proceedings of SPIE. 2004.

Salomons E. Computational atmospheric acoustics. Amsterdam (The Netherlands): Kluwer Acedemic Publishers; 2001.

Schmidt V. Ray trace. Mathworks Central File Exchange; 2014 Feb 12 [accessed 2020 Mar 3]. https://www.mathworks.com/matlabcentral/fileexchange/26253-raytrace.

Wikipedia. Complex Gaussian distribution. Wikimedia Foundation; 2020a Sep 7 [accessed 2020 Aug 1]. https://en.wikipedia.org/wiki/Complex_normal_distribution.

Wikipedia. Spherical coordinate system. Wikimedia Foundation; 2020b Sep 3 [accessed 2020]. https://en.wikipedia.org/wiki/Spherical_coordinate_system.

Wikipedia. Trilinear interpolation. Wikimedia Foundation; 2020c May 10 [accessed 2020]. https://en.wikipedia.org/wiki/Trilinear_interpolation.

Appendix A. FDTD Ray Tracing Code

```
function [x_array, s_array,
dtime_array]=ray_tracer_v3h(theta,phi,x0,z,c_z,w_x,w_y
,rayt)

% finite difference time domain (fdtd) raytracing using
pierce equation 8-1.10a and 8-1.1.10b, pg 375, 1991
% written by Geoffrey Goldman 2/4/2020

% output parameters
% x_array = position of ray path as a function of time
% s_array = wave-slowness vector of ray as a function of
time, parellel to n
% dtime_array = time step value for each increment =
dt(z)

% input parameters

% theta and phi of ray propagating on the phase owards
the mic array without wind effects in radians,
n=wavefront normal
% x0 = initial position of source and ray path, 3d-
vector in meters
% z = array of heights for wind and temperature data in
meters
% w_x = array of wind speeds in x direction for each z
value
% w_y = array of wind speeds in y direction for each z
value
% rayt = structure with ray tracing parameters
% rayt.DT_reduction1  reduce time step size at height 1
% rayt.DT_reduction2  reduct time step size at height 2
% rayt.DT_reduction1_height reduce time step size at
this height
% rayt.DT_reduction2_height  % reduce time step size at
this height
% rayt.K_factor  increase the number of array elements
in predicted value of K , try to account for dynamic
time step
% rayt.K_stop_factor stopping condition if ray does not
hit the ground
% k < k_max_est*K_stop_factor
% rayt.validate 0 or 1, increment z in opposite
direction, normally validate=0, 1=configure to compare
to a benchmark code
% rayt.min_elevation_angle minimum elevation angle input
to raytracing algorithm
```

```
% notes : old optics eikonal = W(x) = c0*T(x)
% now eikonal = T(x)

% algorithm parameters, leave them in the code for now

debug_flag=0;
FIRST_STEP_FACTOR=1e-1;

n=-[cos(phi)*sin(theta)              sin(phi)*sin(theta)
cos(theta)]; % normal vector of wave front in moving
media

%x=x0;   % initial position of ray
zt=x0(3); % z location > 0
dt=rayt.dt0;

cz=interp1(z,c_z,zt); % propagation speed
wx=interp1(z,w_x,zt); % wind x
wy=interp1(z,w_y,zt);
w=[wx wy 0]; % initial wind vector
s=n./(cz  +  w*n');  %  initial  slowness  vector  8-1.3,
approximate n
omega=1-w*s';
xdot=cz^2*s/omega + w;

dz=dt*xdot(3);

if (rayt.validate==1) % goes up, then down
  K=900;
  K_stop=900;
else
  K=round(-(zt*rayt.K_factor)/dz);   % estimate size of
array
  K_stop=round(-(zt*rayt.K_stop_factor     )/dz);     %
calculate maximum number of loops
  if (K==0 | isnan(K))
    error('K==0 | isnan(K)')
  end
  if (K < 20 | K > 1e4)
    error(['K   is   unusually   large   or   small,
K=',num2str(K)])
  end
end

% allocate memory for arrays

x_array=zeros(K,3)/0; % position of wavefronts
```

```
s_array=zeros(K,3); % slowness array
psuedo_dist_array=zeros(K,1);    %    psuedo    distance
increment array
dtime_array=zeros(K,1); % time increment array

if (debug_flag==1)
  xdot_array=zeros(K,3);
  xdot_array(1,:)=xdot1;
  sdot_array=zeros(K,3);
end

k=1;
x_array(k,:)=x0; % starting point of ray
s_array(k,:)=s;
psuedo_dist_array(k)=0;
dtime_array(k)=0;
x=x0;
sdot=zeros(1,3);

while ((x(3) > 0 & rayt.validate==0 & k < K_stop) | (x(3)
< 501 & rayt.validate==1 & k < K_stop)) % loop until ray
hits the ground
  k=k+1;

  cz_last=cz;
  w_last=w;
  zt=x(3) + dz;
  %
  %  if (zt < 0)
  %    dz=-x(3) -1e-4;
  %    dt=dz/xdot(3);
  %  end

  cz=interp1(z,c_z,zt);  % propagation  speed,  note,  dz
negative
  wx=interp1(z,w_x,zt); % wind x
  wy=interp1(z,w_y,zt); % wind y

  w=[wx wy 0]; % initial wind vector

  dc_dz=(cz-cz_last)/dz;
  dwx_dz=(wx-w_last(1))/dz;
  dwy_dz=(wy-w_last(2))/dz;

  omega=1-(w_last*s');
  xdot=cz_last^2*s/omega + w_last;
```

```
  sdot(3)=  -((omega/cz_last)*dc_dz  +  s(1)*dwx_dz  +
s(2)*dwy_dz);

  x=x+dt*xdot; % update equations
  s=s+dt*sdot;

  x_array(k,:)=x; % starting point of ray
  s_array(k,:)=s;
  psuedo_dist_array(k)=cz*dt;
  dtime_array(k)=dt;

  if (x(3) < rayt.DT_reduction2_height)
    dt=rayt.dt0/rayt.DT_reduction2;
  elseif (x(3) < rayt.DT_reduction1_height) % change dt
to be more sensitive near the ground
    dt=rayt.dt0/rayt.DT_reduction1;
  else
    dt=rayt.dt0;
  end

  dz=dt*xdot(3);


end

if (isnan(x(1))| isnan(x(2))| isnan(x(3)))
  warning('warning x(3) is NaN')
end
end
```

Appendix B. Ground Impedance Code

```
function [Znorm] = gi_1_or_2_para(f,para)

% calculate ground impedance using a 1 or 2 parameter
model for locally reacting surface

% input parameters
% f= frequency
% para = structure
% parameter model=1: Delany and Bazley - good for porous
materials
% parameter model=2: Attenborough


%output parameters

% Znorm = normalized impedance = rho1c1 / rhoc
% k= propagation constant

% sigma = flow resistivity
% asphalt= 1e7 Pa s m-2

NF=length(f);

if (para.imp_model==1) % handbook of acoustics, pg 123
  Znorm=ones(1,NF)+0.0571*(f/para.sigma_e).^(-0.754)  +
1i*0.087*(f/para.sigma_e).^(-0.732);  %  corrected  for
typo
elseif (para.imp_model==2) % two parameter model

  Znorm     =     0.0436*(1-1i)*(para.sigma_e/f)^(0.5)-
19.74*1i*(para.alpha_e/f)^(0.5);

end
end
```

Appendix C. Angular Grid Estimation Code

```
  % estimate range of angles for interpolation

  ray_tracer_out.phi_end_actual=atan2(-n_meas(2),-
n_meas(1));
  ray_tracer_out.theta_end_actual=acos(-
n_meas(3)/norm(n_meas));

  A                  =                  Range_start*([-
sin(ray_tracer_out.theta_end_meas)*sin(ray_tracer_out.
phi_end_meas)
cos(ray_tracer_out.theta_end_meas)*cos(ray_tracer_out.
phi_end_meas);

sin(ray_tracer_out.theta_end_meas)*cos(ray_tracer_out.
phi_end_meas)
cos(ray_tracer_out.theta_end_meas)*sin(ray_tracer_out.
phi_end_meas)]); % observation matrix for LSQF

  det_A=det(A);

  if (abs(det_A)<1e-2) % make sure there is a solution,
I don't think this is still needed
    A=A+eye(2);
    error('abs(det_A)<1e-2')
  end

  %
phi_mod=mod(ray_tracer_out.phi_end_meas+pi/4,pi/2);

  phi_angle_from_45=min(abs(angle(exp(-
1i*ray_tracer_out.phi_end_meas)*exp(1i*(pi/4          +
(0:3)*pi/2)))));

  % don't know the sign, try ++ and +-

  dphi_dtheta0p=A\refr.xy_corner_intp_p;  % find range
of angles for interpolation using lsq to generate lookup
table
  dphi_dtheta0m=A\refr.xy_corner_intp_m;  % find range
of angles for interpolation using lsq to generate lookup
table

  if (phi_angle_from_45 < pi/8) % near 45 degrees


dphi_dtheta0=(abs(dphi_dtheta0p)+abs(dphi_dtheta0m))/2
;
```

```
    xy_corner_intp0=refr.xy_corner_intp_p;

  else % near cardinal angles

    if (norm(dphi_dtheta0p,1) > norm(dphi_dtheta0m,1))
% pick the better fit
      dphi_dtheta0=dphi_dtheta0m;
      xy_corner_intp0=refr.xy_corner_intp_m;
    else
      dphi_dtheta0=dphi_dtheta0p;
      xy_corner_intp0=refr.xy_corner_intp_p;
    end
  end

  if (debug_code)
    check_result_xy_corner=A*dphi_dtheta0;
  end

  % find a good 2d grid for launching rays

  grid_counter=0;
  good_solution=0; % dphi_dtheta have adequate range
  dphi_dtheta=dphi_dtheta0; % inital conditions
  xy_corner_intp=xy_corner_intp0;
  cost_min=1e20; % make big

  while    (grid_counter    <    refr.max_grid_count    &
good_solution==0)


igrid_mod=mod(grid_counter+1,refr.grid_count_retry);


Max_ratio=refr.Max_ratio0+refr.ratio_inc*igrid_mod;   %
make easier

Min_ratio=refr.Min_ratio0/(1+refr.ratio_inc*igrid_mod)
;

    grid_counter=grid_counter+1;

    % test at points on a cross, not corners

    x0_pt=get_x0_ray_v2(theta                              +
dphi_dtheta(2),phi,x0,z,c_z,w_x,w_y,refr.rayt);  %  only
approximate z=0
```

```
    x0_mt=get_x0_ray_v2(theta                                -
dphi_dtheta(2),phi,x0,z,c_z,w_x,w_y,refr.rayt);
    x0_pp=get_x0_ray_v2(theta,phi+
dphi_dtheta(1),x0,z,c_z,w_x,w_y,refr.rayt);
    x0_mp=get_x0_ray_v2(theta,phi-
dphi_dtheta(1),x0,z,c_z,w_x,w_y,refr.rayt);

    if (isnan(x0_pt(1)))
      error('isnan(x0_pt(1))')
    else

      xy0_array=[x0_pt(1:2)  ;x0_mt(1:2);  x0_pp(1:2)  ;
x0_mp(1:2)];
      [xy_min,ixy_min]=min(xy0_array);
      [xy_max,ixy_max]=max(xy0_array);

      x_ave_dist=xy_max(1)-xy_min(1);
      y_ave_dist=xy_max(2)-xy_min(2);

      ratio_x=2*abs(xy_corner_intp0(1))/x_ave_dist;   %
make = 1 , if ratio > 1 , increase xy_corner_intp
      ratio_y=2*abs(xy_corner_intp0(2))/y_ave_dist;
      cost_ratio=abs(ratio_x-1.05) + abs(ratio_y-1.05);
% little bigger is better than a little small

      if (cost_ratio < cost_min)
        cost_min=cost_ratio;
        xy_corner_intp_best=xy_corner_intp;
        dphi_dtheta_best=dphi_dtheta;
      end

      if (ratio_x > Min_ratio & ratio_x < Max_ratio &
ratio_y > Min_ratio & ratio_y < Max_ratio)
        good_solution=1; % big number
      elseif (igrid_mod > 0)
        dphi_dtheta_debug=A\xy_corner_intp;
        xy_corner_intp=xy_corner_intp.*[ratio_x       ;
ratio_y];
        if (phi_angle_from_45 < pi/8) % near 45 degrees
          dphi_dtheta0p=A\ xy_corner_intp; % find range
of angles for interpolation using lsq to generate lookup
table
          dphi_dtheta0m=A\(xy_corner_intp.*[1  ;-1]);  %
find range of angles for interpolation using lsq to
generate lookup table

dphi_dtheta=(abs(dphi_dtheta0p)+abs(dphi_dtheta0m))/2;
```

```
        else
          dphi_dtheta=A\xy_corner_intp; % find range of
angles for interpolation using lsq to generate lookup
table
        end
      else

dphi_dtheta=dphi_dtheta_best.*(abs(ones(2,1)+randn(2,1
)*refr.random_guess_perc)+ones(2,1)*1e-3);
        xy_corner_intp=xy_corner_intp0;
      end
    end
  end % while

  xy_corner_intp=xy_corner_intp_best;
```

Appendix D. Bilinear Interpolation Code

```
function
[intp_value,error_value_vec,neighbor_status]=interp3_i
reg_v14(x3_array,z_array,Tdata3D,desired_rc_location,p
ara,i_highest_match,force_tri_intp)

% modified bilinear interpolation

% output values
% intp_value interpolated time value
% neighbor_status  0=nearest neighbor, 1 = 8 modified
trilinear interpolatation

% input values

% x3_array = 4d array of phi index, theta index, z index,
x,y position, size=(Nphi,Ntheta,Ntime,2)
% z_array = 1d array of interpolated z values
% Tdata3D = propagation time data, size=(Nphi,Ntheta,
Ntime)
% desired_rc_location to get propagation time value,
% para = structure of parameters for interpolation
%  i_highest_match  =  use  point  associated  with
i_highest_match, highest best match to nn, 1 is best, 2
= second best, ...
% force_tri_intp 0 = regular mode, use 4 points, then 3,
1=test model, force interpolation using three points

% initialize

neighbor_status=0;  %  0=nearest  neighbor,  1=modified
trilinear interpolatation

[Nphi,Ntheta,Ntime,two3]=size(x3_array);

% search in z or time first

[error_abs_z,iminz]=min(abs(z_array-
desired_rc_location(3)));

if (error_abs_z > para.error_nn_dist) % interpolate
  [error_z]=z_array(iminz)-desired_rc_location(3);

  if (iminz==1)
    if (error_z > 0)
      imin2=[ iminz (iminz+1)];
    else
      imin2=0; % bad value
```

```
    end
  elseif (iminz==length(z_array))
    if (error_z < 0)
      imin2=[ (iminz-1) iminz];
    else
      imin2=0; % bad value
    end
  elseif (error_z < 0)
    imin2=[(iminz-1) iminz];
  elseif (error_z > 0)
    imin2=[iminz (iminz+1)];
  end
else
  imin2=0; % no interpolation in z, use best xy array
  %   neighbor_status=1;
end

if (imin2==0) % check this code

x2_array=reshape(x3_array(:,:,iminz,:),Nphi*Ntheta,2);
% make 2d array for searching
  t2_array=reshape(Tdata3D(:,:,iminz),Nphi*Ntheta,1); %
make 2d array for searching
else
  z_error2=z_array(imin2)-desired_rc_location(3);
  zd=z_error2(2)/diff(z_error2);   %   wait   for   first
element


x21_array=reshape(x3_array(:,:,imin2(1),:),Nphi*Ntheta
,2)*zd; % make 2d array for searching

x22_array=reshape(x3_array(:,:,imin2(2),:),Nphi*Ntheta
,2)*(1-zd); % make 2d array for searching
  x2_array=x21_array+x22_array;


t21_array=reshape(Tdata3D(:,:,imin2(1)),Nphi*Ntheta,1)
*zd; % make 2d array for searching

t22_array=reshape(Tdata3D(:,:,imin2(2)),Nphi*Ntheta,1)
*(1-zd); % make 2d array for searching
  t2_array=t21_array+t22_array;
  if (0) % check code
    z_test=z_array(imin2(1))*zd + z_array(imin2(2))*(1-
zd);
    if (abs(z_test-desired_rc_location(3))>1e-6)
```

```
      error(z_test)
    end
  end
  if (0)
    t2_plot_array=reshape(t2_array,Nphi,Ntheta);
    figure
    imagesc(t2_plot_array)
    colorbar
  end
end

[min_2_error_sq_array,imin_error_array]=sort(sum((x2_a
rray' -desired_rc_location(1:2)').^2)) ;

ntheta=ceil(imin_error_array(i_highest_match)/(Nphi));
nphi=  ceil((imin_error_array(i_highest_match)-(ntheta-
1)*Nphi));

est_location=x2_array(imin_error_array(i_highest_match
),:);
error_value_vec=est_location-
desired_rc_location(1,1:2); % estimate - desired

if (sum(abs(error_value_vec))<para.error_nn_dist)  % if
estmated location is almost exact

intp_value=t2_array(imin_error_array(i_highest_match))
; % use nearest neighbor estimate
  neighbor_status=1; % good value, even though not fully
interpolated
else
  if (nphi==Nphi)
    index2=nphi-1+Nphi*(ntheta-1); % decrement iphi
    est_location2=x2_array(index2,:);
    dphi=est_location-est_location2;  %  higher  index
minus lower index
  else
    index2=nphi+1+Nphi*(ntheta-1); % increment iphi
    est_location2=x2_array(index2,:);
    dphi=est_location2-est_location;
  end
  if (ntheta==Ntheta)
    index2=nphi+Nphi*(ntheta-2); % decrement itheta
    est_location2=x2_array(index2,:);
    dtheta=est_location-est_location2;  %  higher  index
minus lower index
  else
```

```
    index2=nphi+Nphi*(ntheta); % increment itheta
    est_location2=x2_array(index2,:);
    dtheta=est_location2-est_location;  % higher  minus
lower index
  end
  if (abs(dphi(1)) > para.angle_factor*abs(dtheta(1)))
    dir_x=1; % direction of gradient in the x direction,
largest in the phi direction, 1=phi 2=theta 0=either
  elseif        (para.angle_factor*abs(dphi(1))        <
abs(dtheta(1)))
    dir_x=2; % theta
  else
    dir_x=0; % either direction
  end
  if (abs(dphi(2)) > para.angle_factor*abs(dtheta(2)))
    dir_y=1; % direction of gradient in the y direction,
largest in the phi direction, 1=phi 2=theta 0=either
  elseif        (para.angle_factor*abs(dphi(2))        <
abs(dtheta(2)))
    dir_y=2; % theta
  else
    dir_y=0; % either direction
  end
  if (dir_x==dir_y & dir_x>0) % if max x and y gradient
in same angular direction
    if  (dir_x==1)  % if direction always largests for
phi, use theta component with largest gradient
      if  (abs(dtheta(1))  >  abs(dtheta(2)))   %  select
index associated with the largest gradient
        dir_x=2;
        dir_y=1;
      else
        dir_x=1;
        dir_y=2;
      end
    else % dir_x=2 , theta best direction
      if  (abs(dphi(1))  >  abs(dphi(2)))   %  select  index
associated with the largest direction
        dir_x=1;
        dir_y2=2;
      else
        dir_x=2;
        dir_y=1;
      end
    end
  else % both are not pointing in the same direction
    if (dir_x==0 & dir_y==0) % either good
```

```
      if (abs(dtheta(1)) > abs(dtheta(2))) % que on
theta
        dir_x=2;
        dir_y=1;
      else
        dir_x=1;
        dir_y=2;
      end
    elseif (dir_x==1)
      dir_y=2;
    elseif (dir_y==1)
      dir_x=2;
    elseif (dir_y==2)
      dir_x=1;
    elseif (dir_x==2)
      dir_y=1;
    else
      error('check code, statement should not be
executed')
    end
  end

  phi_neighbor_status=0; % start with phi, then theta
  if (dir_x==1) % phi coordinate
    icord=1; % x
  else
    icord=2; % y
  end
  if (nphi == Nphi)
    error_p1=NaN; % plus 1
    index2=nphi-1+Nphi*(ntheta-1); % decrement iphi
    error_m1=x2_array(index2,icord)-
desired_rc_location(icord); % minus 1
  elseif (nphi==1)
    index2=nphi+1+Nphi*(ntheta-1); % increment iphi
    error_m1=NaN; % can not calculate - index error
    error_p1=x2_array(index2,icord)-
desired_rc_location(icord);
  else
    index2=nphi+1+Nphi*(ntheta-1); % increment iphi
    error_p1=x2_array(index2,icord)-
desired_rc_location(icord);
    index2=nphi-1+Nphi*(ntheta-1); % decrement iphi
    error_m1=x2_array(index2,icord)-
desired_rc_location(icord);
  end
```

```
  if (error_value_vec(icord) <= 0 & error_p1 >= 0) %
estimate desired estimate+1
    inc_phi=1;
    phi_neighbor_status=1;
  elseif (error_value_vec(icord) <= 0 & error_m1 >= 0)
% estimate  desired estimate-1
    inc_phi=-1;
    phi_neighbor_status=1;
  elseif (error_value_vec(icord) > 0 & error_p1 < 0)   %
estimate+1 desired estimate
    inc_phi=1;
    phi_neighbor_status=1;
  elseif (error_value_vec(icord) > 0 & error_m1 < 0)  %
estimate-1 desired estimate
    inc_phi=-1;
    phi_neighbor_status=2;
  else
    increment_status=1; % while loop
    inc=1; % plus or minus

    if (nphi==1)
      inc_dir=1;
    elseif (nphi==Nphi)
      inc_dir=-1;
    elseif (abs(error_p1) > abs(error_m1))
      inc_dir=-1;
    else
      inc_dir=1;
    end

    while (increment_status)
      inc=inc+1;
      if (inc*inc_dir+nphi > 0 & inc*inc_dir+nphi <=
Nphi)
        index2=nphi+inc*inc_dir+Nphi*(ntheta-1);      %
decrement iphi
        error_inc=x2_array(index2,icord)-
desired_rc_location(icord);
      end
      if (error_inc*error_m1 < 0) % sign changed
        inc_phi=inc*inc_dir;
        phi_neighbor_status=1;
        increment_status=0;
      end
      if (inc > para.max_increment_intp)
        increment_status=0;
        phi_neighbor_status=0; % no values found
```

```
      end
    end
  end

  if (dir_x==1) % increment theta coordinate
    icord=2; % x
  else
    icord=1;
  end

  theta_neighbor_status=0; % search in theta

  if (ntheta == Ntheta)
    error_p1=NaN;
    index2=nphi+Nphi*(ntheta-2); % decrement itheta
    error_m1=x2_array(index2,icord)-
desired_rc_location(icord);
  elseif (ntheta==1)
    error_m1=NaN;
    index2=nphi+Nphi*(ntheta); % decrement itheta
    error_p1=x2_array(index2,icord)-
desired_rc_location(icord);
  else
    index2=nphi+Nphi*(ntheta); % increment itheta
    error_p1=x2_array(index2,icord)-
desired_rc_location(icord);
    index2=nphi+Nphi*(ntheta-2); % decrement itheta
    error_m1=x2_array(index2,icord)-
desired_rc_location(icord);
  end

  if (error_value_vec(icord) <= 0 & error_p1 >= 0) %
estimate desired estimate+1
    inc_theta=1;
    theta_neighbor_status=1;
  elseif (error_value_vec(icord) <= 0 & error_m1 >= 0)
% estimate  desired estimate-1
    inc_theta=-1;
    theta_neighbor_status=1;
  elseif (error_value_vec(icord) > 0 & error_p1 < 0)   %
estimate+1 desired estimate
    inc_theta=1;
    theta_neighbor_status=1;
  elseif (error_value_vec(icord) > 0 & error_m1 < 0)  %
estimate-1 desired estimate
    inc_theta=-1;
    theta_neighbor_status=1;
```

```
  else
    increment_status=1;
    inc=1; % plus or minus
    if (ntheta==1)
      inc_dir=1;
    elseif (ntheta==Ntheta)
      inc_dir=-1;
    elseif (abs(error_p1) > abs(error_m1))
      inc_dir=-1;
    else
      inc_dir=1;
    end
    while (increment_status)
      inc=inc+1;

      if (inc*inc_dir+ntheta > 0 & inc*inc_dir+ntheta <=
Ntheta)
        index2=nphi+Nphi*(ntheta-1+inc*inc_dir);      %
itheta
        error_inc=x2_array(index2,icord)-
desired_rc_location(icord);
      end
      if (error_inc*error_m1 < 0) % sign changed
        inc_theta=inc*inc_dir;
        theta_neighbor_status=1;
        increment_status=0;
      end
      if (inc > para.max_increment_intp)
        increment_status=0;
        theta_neighbor_status=0;
      end
    end
  end

  neighbor_status=theta_neighbor_status                &
phi_neighbor_status;

  if (neighbor_status ) % do full interpolation

    % set up matrices to do interpolation in x first

    xy_array=zeros(2,2,2);    % xy coordiantes for 4
corner points, wikipedia termology
    Ct=zeros(2,2); % time data at corner points

    % align data indexed over angle to indexed over
cartesian coord
```

```
    index2=nphi+Nphi*(ntheta-1);
    xy_array(1,1,:)=x2_array(index2,:);
    Ct(1,1)=t2_array(index2);

    index2=nphi+inc_phi+Nphi*(ntheta-1+inc_theta);    %
increment phi and theta
    xy_array(2,2,:)=x2_array(index2,:);
    Ct(2,2)=t2_array(index2);

    index12=nphi+inc_phi+Nphi*(ntheta-1);  %  increment
phi
    index21=nphi+Nphi*(ntheta-1+inc_theta); % increment
theta

    if (dir_x==1) % setup matrices to average over first
array index
      xy_array(2,1,:)=x2_array(index12,:);
      Ct(2,1)=t2_array(index12);
      xy_array(1,2,:)=x2_array(index21,:);
      Ct(1,2)=t2_array(index21);
    else
      xy_array(1,2,:)=x2_array(index12,:);
      Ct(1,2)=t2_array(index12);
      xy_array(2,1,:)=x2_array(index21,:);
      Ct(2,1)=t2_array(index21);
    end

    if (0)
      figure
      hold all
      plot(xy_array(1,1,1),xy_array(1,1,2),'x')
      plot(xy_array(1,2,1),xy_array(1,2,2),'x')
      plot(xy_array(2,1,1),xy_array(2,1,2),'x')
      plot(xy_array(2,2,1),xy_array(2,2,2),'x')

plot(desired_rc_location(1),desired_rc_location(2),'o'
)
    end


    x5_array=[xy_array(1,1,1)           xy_array(1,2,1)
xy_array(2,1,1)                         xy_array(2,2,1)
desired_rc_location(1)];
    y5_array=[xy_array(1,1,2)           xy_array(1,2,2)
xy_array(2,1,2)                         xy_array(2,2,2)
desired_rc_location(2)];
```

```
    [tempx,itempx]=sort(x5_array);
    [tempy,itempy]=sort(y5_array);

    if (itempx(3)==5 & itempy(3)==5 & force_tri_intp==0)
% desired x and y values in middle, order does not matter
      % interpolate over x first
      y2_array=zeros(1,2); % y values averaged over x
      C=zeros(1,2);  % time values averaged over x
      xd=(desired_rc_location(1)-
xy_array(1,1,1))/(xy_array(2,1,1)-xy_array(1,1,1));   %
second minus first
      y2_array(1)     =     xy_array(1,1,2)*(1-xd)     +
xy_array(2,1,2)*xd;
      C(1)=Ct(1,1)*(1-xd) + Ct(2,1)*xd;
      if (0) % check
        x_temp      =      xy_array(1,1,1)*(1-xd)      +
xy_array(2,1,1)*xd;
        zero_error=x_temp-desired_rc_location(1);
      end
      xd=(desired_rc_location(1)-
xy_array(1,2,1))/(xy_array(2,2,1)-xy_array(1,2,1));   %
second minus first
      y2_array(2)     =     xy_array(1,2,2)*(1-xd)     +
xy_array(2,2,2)*xd;
      C(2)=Ct(1,2)*(1-xd) + Ct(2,2)*xd;

      yd=(desired_rc_location(2)-
y2_array(1))/(y2_array(2)-y2_array(1)); % second minus
first
      intp_value=C(1)*(1-yd) + C(2)*yd;

    else % triangulation , find critical y point

      % find closest point to desired point, then
selected a point to the
      % left of the vector and to the right of the vector

      t4_array=[Ct(1,1)         Ct(1,2)         Ct(2,1)
Ct(2,2)];
      vec_best=desired_rc_location(1:2)-[x5_array(1)
y5_array(1)]; % vector that is closest to the desired
point
      vec3=[x5_array(1,2:4)'         y5_array(1,2:4)']-
ones(3,1)*[x5_array(1)  y5_array(1)];  %  three  other
vectors
```

```
      angle_best=atan2(vec_best(2),vec_best(1));      %
angle of closest point
      angle3=atan2(vec3(:,2),vec3(:,1));  %  angle  of
other 3 points

      s_angle_diff=sign(angle(exp(-
1i*angle_best)*exp(1i*angle3))); % find angles between
closested point and other points
      st_angle_diff=sum(s_angle_diff);  %  if  all  the
points  are  to  the  left  or  the  right,  then  no
triangulation

      if  (abs(st_angle_diff)<3)  %  good  values,  can
triangulate
        sq_error=sum([x5_array(1,2:4)'
y5_array(1,2:4)']' -desired_rc_location(1:2)').^2
        % need both a positive and negative angles
relative to closest point
        i1p=find(s_angle_diff==1);   %   indices   for
positive angles
        i1n=find(s_angle_diff== (-1));  % indices with
megative angles
        if (st_angle_diff==1) % two positive angles

[best_pos_value_error,ibest_pos_value]=min(sq_error(i1
p));

best_pos_value=[x5_array(i1p(ibest_pos_value)+1)
y5_array(i1p(ibest_pos_value)+1)];
          best_neg_value=[x5_array(i1n+1)
y5_array(i1n+1)];
          xy3_array=[x5_array(1)           y5_array(1);
best_pos_value ; best_neg_value];
          time3_array=[t4_array(1)
t4_array(i1p(ibest_pos_value)+1) t4_array(i1n+1)];
          [intp_value,neighbor_status]                =
tri_intp(xy3_array,time3_array,desired_rc_location(1:2
));
        else

[best_neg_value_error,ibest_neg_value]=min(sq_error(i1
n));

best_neg_value=[x5_array(i1n(ibest_neg_value)+1)
y5_array(i1n(ibest_neg_value)+1)];
          best_pos_value=[x5_array(i1p+1)
y5_array(i1p+1)];
```

```
          xy3_array=[x5_array(1)           y5_array(1);
best_neg_value ; best_pos_value];
          time3_array=[t4_array(1)
t4_array(i1n(ibest_neg_value)+1) t4_array(i1p+1)];
          [intp_value,neighbor_status]                =
tri_intp(xy3_array,time3_array,desired_rc_location(1:2
));
        end
      else % nothing worked use nn

intp_value=t2_array(imin_error_array(i_highest_match))
; % use nearest neighbor estimate
        neighbor_status=0;
      end
    end
  else % nearest neighbor

intp_value=t2_array(imin_error_array(i_highest_match))
; % use nearest neighbor estimate
    neighbor_status=0;
  end
end % do interpolation
end
```

Appendix E. Interpolation Code Using Three Points

```
function [f_intp_value,status] =
tri_intp(xy_array,f_array,desired_loc)

check_alg=1; % 0= fast, 1=debugging
status=1; % good interpolation

[tempx,ixtemp]=sort(xy_array(:,1));
[tempy,iytemp]=sort(xy_array(:,2));

f_intp_value=NaN;

if   (tempx(1)   >   desired_loc(1)   |   tempx(3)   <
desired_loc(1) ) % check to see if desired point in
midddle of xy_array
  status=0;
elseif   (tempy(1)   >   desired_loc(2)   |   tempy(3)   <
desired_loc(2) ) % check to see if desired point in
midddle of xy_array
  status=0;
else  % good values

  % xy_array = array with 3 xy pairs, closest location
in first element
  % f_array = function associated with xy_array
  % desired_loc - desired xy location for interpolation

  v0=xy_array(1,1:2)-desired_loc; % vector
  % find line
  m0=v0(2)/v0(1);
  b0=xy_array(1,2)-m0*xy_array(1,1);

  v12=xy_array(2,1:2)-xy_array(3,1:2);
  m12=v12(2)/v12(1);
  b12=xy_array(2,2)-m12*xy_array(2,1);

  xi=(b12-b0)/(m0-m12);
  yi=m0*xi+b0;

  D1=norm(xy_array(2,:)-[xi yi]);
  D2=norm(xy_array(3,:)-[xi yi]);

  a1=D1/(D1+D2);

  f_i=f_array(2)*(1-a1)+f_array(3)*a1;
  if (check_alg==1)
    vi=xy_array(2,:)*(1-a1)+xy_array(3,:)*a1;  %  =  [xi
yi]
```

```
    end

    D0=norm(v0);
    Dbig=norm(xy_array(1,:)-[xi yi]);
    a2=D0/(Dbig);

    f_intp_value=f_array(1)*(1-a2) + f_i*a2;
    if (check_alg==1)
      vd=xy_array(1,:)*(1-a2)+vi*a2;
    end

  end
  end
```

List of Symbols, Abbreviations, and Acronyms

1D	one-dimensional
2D	two-dimensional
3D	three-dimensional
DOA	direction of arrival
FDTD	finite-difference, time-domain
FFP	fast field program
M-O	Monin–Obukhov
SINR	signal-to-interference-plus-noise ratio
SNR	signal-to-noise ratio
TDOA	time difference of arrival

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) CCDC ARL FCDD RLD DCI TECH LIB

1 (PDF) CCDC ARL FCDD RLS SA G GOLDMAN