PASSIVE LOCALIZATION OF ACOUSTIC SOURCES IN MEDIA WITH NON-CONSTANT SOUND VELOCITY

A Thesis

by

THOMAS SCOTT BRANDES

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 1998

Major Subject: Interdisciplinary Engineering

PASSIVE LOCALIZATION OF ACOUSTIC SOURCES IN MEDIA WITH NON-CONSTANT SOUND VELOCITY

A Thesis

by

THOMAS SCOTT BRANDES

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Robert H. Benson
(Chair of Committee)

William E. Evans
(Member)

Karan L. Watson
(Member)

Joseph A. Morgan
(Member)

Karan L. Watson
(Head of Department)

May 1998

Major Subject: Interdisciplinary Engineering

ABSTRACT

Passive Localization of Acoustic Sources in Media
with Non-Constant Sound Velocity. (May 1998)

Thomas Scott Brandes, B.S., Georgia Institute of Technology

Chair of Advisory Committee: Dr. Robert Benson

There is a growing concern about the effects of low frequency sounds (LFS) on marine mammals. One way to assess these effects on marine mammals involves the study of disturbance reactions. Detailed research of disturbance reactions of submerged marine mammals requires 3-dimensional localization and tracking of the animals. An acoustic source is localized passively with the use of travel time differences (TTD) of a signal's reception received by multiple hydrophones at known positions. An initial approximation of source position is found using straight-line paths of sound propagation between source and receiver. An algorithm is then used to iteratively pinpoint source position in a medium with a non-constant sound speed. This algorithm calculates direct eigenrays connecting the approximate source position and each of the four buoys. These eigenrays are used to generate a set of TTD values that are subtracted from TTD values recorded in the field, giving TTD differences (TTDF). T_i = travel time to buoy i. $TTD_{1i} = T_1 - T_i$. $TTDF_i = TTD'_{1i} - TTD_{1i}$. The depth coordinate of the source position is adjusted until $TTDF_3 \approx 0$. Then one of the horizontal components of the source position is adjusted until $TTDF_1 \approx \pm TTDF_2$. Then the other horizontal component of the source position is adjusted until $TTDF_1 \approx \mp TTDF_2$. This process is repeated until $TTDF_3 \approx 0$

after adjusting both horizontal components of the source position. Five hydrophone array configurations are tested, each with 30 pseudo-randomly generated source positions. Average errors of the 150 source position calculations, (x, y, depth) in meters, are (±1.58, ±1.70, ±10.44) for the straight-line, and (±0.72, ±0.83, ±1.10) for the algorithm. On average, the algorithm improves the source depth calculation by an order of magnitude.

This dissertation is dedicated
to my Parents.

ACKNOWLEDGMENTS

I would like express my deepest gratitude to my family, who has given enduring support in my scholastic pursuits. I would like express my sincere thanks to Dr. Robert Benson for his encouragement and guidance in this project. I would also like to thank the other members of my committee, Dr. Bill Evans, Dr. Joseph Morgan, and Dr. Karan Watson for making my degree in this field possible. A special thanks to Troy Sparks for directing much of the field work that my project is a part of. I am grateful to Dr. Rainer Fink and John T. Willis for substantial work on buoy design and construction. I would like to thank Stewart Robinson for helping with the setup of field operations. I would like to thank Brad Dawes, Kristie Jenkin, and Susan Eckelman for helping with the field work on the R/V *Acadiana*. I would like to thank Francisco "Paco" Ollervides, Sarah Stienessen, and Suzanne Yin for helping with the field work on the BP oil rig. I would like to thank the Office of Naval Research for funding this project, and British Petroleum for providing an oil platform in the Gulf of Mexico for this project. The research vessel *Acadiana* from Cocodrie, LA was provided by the Louisiana Universities Marine Consortium.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

x, y	horizontal distances (m)
z	depth (m)
t	travel time (sec)
T_i	travel time to receiver *i* (sec)
τ_{1i}	travel time difference (TTD) between receiver 1 and receiver *i* (sec)
$\mathbf{r}_i$	position vector of receiver *i*
$\mathbf{s}_0$	position vector of the source
$\mathbf{s}$	approximate position vector of the source
c	sound speed (m/s)
c_0	sound speed at beginning of layer (m/s)
g	change in sound speed with respect to depth (m/s^2)
θ	grazing angle (rad)
n	number of receivers

INTRODUCTION & BACKGROUND

Recently, there has been an increased interest in the effects of man-made noise on marine mammals. Concerns over the effects of noise on marine mammals initially became focused with the U.S. Marine Mammal Protection Act (MMPA) of 1972, which established a moratorium on harassment, hunting, killing, and the capturing of marine mammals. Interactions involving human generated noise have drawn attention because the MMPA treats noise related disturbances as a form of harassment and thus a violation of the act. Currently, there is particular interest in the acoustic effects of military and industrial operations and research activities in the oceans of the world (Richardson et al., 1995).

Most human activities in the offshore environment produce low-frequency sound (LFS) between 5 Hz and 500 Hz. Most LFS in the ocean is generated by ship traffic (Urick, 1983) and little is known about its effect on marine mammals (Richardson et al., 1995). However, highly visible research activities are attracting the attention of environmental groups (Herman 1994, Holing 1994, NRC/OSB 1994), particularly when research involves acoustic tomography. Acoustic tomography measures physical properties of ocean sections such as sound speed, density, salinity, and temperature. This field is of particular concern for MMPA regulators since these researchers in acoustic tomography usually use LFS projected into the deep sound channel. Depending on the initial source level of these signals, they can be heard by marine mammals

---

The Journal of the Acoustical Society of America is selected as the model for style and format.

entering the deep sound channel thousands of kilometers away. An example of current research of this nature is the Acoustic Thermometry of Ocean Climate (ATOC) project to monitor temperature trends in the Pacific Ocean. In this project, coded signals centered around 75 Hz are periodically projected into the deep sound channel at a source level around 195 dB rel 1μPa from Hawaii and Central California. These signals are received at various sites in the North Pacific as well as New Zealand (Richardson et al.., 1995).

Documented disturbance reactions

In evaluating whether or not a marine mammal is disturbed by a stimulus, a set of recognizable disturbance reactions are sought. These reactions usually involve cessation of feeding, resting, vocalizing, or social interaction, and the onset of alertness or avoidance (Richardson et al. 1995). For whales, avoidance includes hasty diving or swimming away. Sperm Whales (*Physeter macrocephalus*) are of particular concern because of their endangered status. Although no studies on sperm whale hearing have been published (Richardson et al., 1995), some insight into their hearing range comes from examining the frequency range in which they vocalize. Sperm whale clicks have a frequency range from < 100 Hz to 30 kHz, with the highest energy in the 2 kHz - 4 kHz and 10 kHz -16 kHz bandwidths. Their clicks are repeated at rates from 1 to 90 per second (Watkins and Schevill 1977, Watkins et al. 1985, Watkins 1980).

There is some information about sperm whale reaction to human generated sound. Watkins and Schevill (1975) found that nearby sperm whales temporarily interrupted their sound production without exception when exposed to a short sequence of pulses at 6-13 kHz. Additionally, they found that the duration of the whale's silence was correlated with received levels of the pulsed sounds. Similarly Bowles et al. (1994) noticed that sperm whales stopped calling when exposed to seismic pulses, even though these seismic pulses where only 10-15 dB above the ambient noise level, and generated over 300 km away. Watkins et al. (1985) found that sperm whales not only ceased clicking, but also scattered away from loud sonar pulses from military submarines. These signals were between 3.25 kHz -8.4 kHz and were in a sequence of 4 - 20 pulses at a rate of 1 - 5 per minute. Similarly Mate et al. (1994) suggests that sperm whales moved over 50 km away from an active seismic exploration vessel in the Gulf of Mexico. However, Mate based his observations on a single event. More recently, investigations suggest that sperm whales frequently do not cease vocalizations in the presence of very loud seismic pulses (Norris, 1997). Additionally, Backus and Schevill (1966) found that sperm whales did not cease calling while exposed to continual pulsing from an echo sounder at 12 kHz, and one whale even adapted its clicks to match the echo sounder pulses.

In a study using LFS transmissions similar to the ATOC signal, all sperm whales encountered ceased calling during the sound transmissions, and were not heard again for 36 hours after the transmission ended (Bowles et al. 1994). These signals were centered around 57 Hz with an output level of 220 dB rel 1 μPa, and emitted in the deep sound channel for one of every three hours for a period of days. Much more research into the effects of sound, particularly of continuous low frequency sound, on sperm whales is needed in order to have a more accurate understanding of whether or not sperm whales are significantly disturbed by human generated noise.

Current work at Texas A&M

There is a project currently underway at Texas A&M University's Center for Bioacoustics to study the behavioral responses of sperm whales to ATOC-like LFS transmitted for intervals lasting several minutes. The protocol for this study requires that behavioral responses be measured by tracking the whale's movements underwater. One way to do this is by analyzing travel time differences of a whale's vocalization received at four separate hydrophones (Fig. 1). Each of the four hydrophones is attached to its own free-drifting sonobuoy. Each sonobuoy collects audio from its hydrophone, GPS data on its current position, and time. From this, a time of arrival for a particular sperm whale click received at a particular location is recorded. Similar studies have successfully located source position of a whale with similar data (Watkins and Schevill 1972, Cummings and Holliday 1987, Freitag and Tyack 1993).

The characteristics of sperm whale vocalizations ("clicks") allow measurement of arrival times. For example, the frequencies at which their clicks have their highest power are 2 kHz - 4 kHz and 10 kHz - 16 kHz, which are both in a frequency range easily detected. Since sperm whale clicks have a source level between 160-180 dB rel 1μPa (Levenson 1974, Watkins 1980), the animals can be detected from approximately water are detectable by hydrophone up to 15 km away from the whale (Watkins and Moore 1982). Furthermore, sperm whale clicks appear to be emitted omnidirectionally

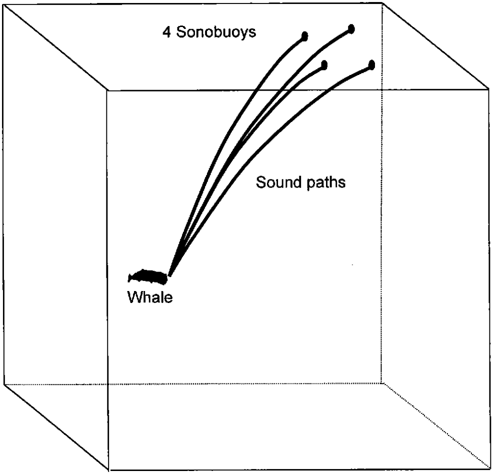


FIG. 1. Diagram of the field set up of the sonobuoys and a whale. All four sonobuoys are at the surface. Only direct eigenrays between the whale and each buoy are used.

11 km (Norris et al.. 1996). Likewise, Watkins (1980) detected sperm whale clicks at a distance of 10 km from the whale, and estimates that sperm whale clicks made in deep (Watkins and Schevill 1974, Watkins 1980). Lastly, sperm whale clicks are heard most frequently during diving and foraging behavior (Whitehead and Weilgart 1991).

This project

The goal of this project is to develop computer software to passively locate sperm whales using vocalizations recorded at known locations. Rudimentary algorithms to do this type of analysis have been written and implemented before (Watkins and Schevill 1972, Cummings and Holliday 1987, Spiesberger and Fristrup 1990, Freitag and Tyack 1993). Previous methods used a ray-tracing model and assumed a straight-line propagation path for sound. The propagation path for sound (in the ray-tracing model) is linear only in a medium of constant sound speed. In the ocean, sound speed is not constant and is dependent on temperature, pressure, and salinity. Sound wave fronts in the "real" ocean propagate in arced paths (Urick 1983, Clay and Medwin 1977). Even though sound speed is a function of several parameters, the sound speed versus depth profile for a particular region of the ocean varies only seasonally, and sound speed varies little along a horizontal plane (Urick 1983), Fig. 2. Therefore, a representative sound speed versus depth profile can be used throughout the study area for a particular calculation. The algorithm used in this project starts with a straight-line ray-tracing model for the propagation of sound, and adds in the complexity of a depth dependent sound speed to more accurately predict the locations of whales.

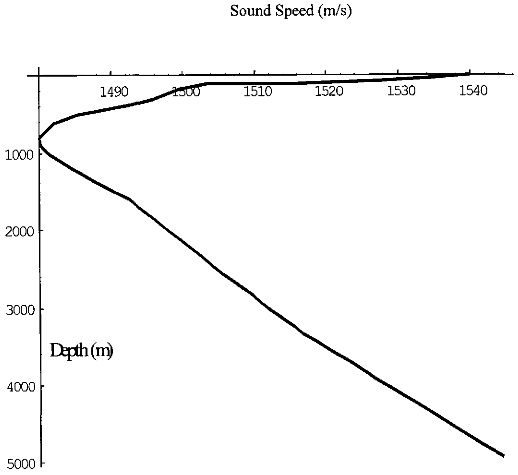


FIG. 2. Typical sound speed vs. depth profile for the Pacific and Atlantic, low latitudes (40°) (Urick 1983). This is the profile used for all calculations in the project. The list of data points used for this profile is given in Appendix D.

MATHEMATICAL SETUP

In this section, the three main mathematical approaches used in this project are explained. They consist of a straight-line model, a ray-tracing model for arced paths, and hyperbolic considerations. Each of these three areas is presented in sections involving introductory thoughts, mathematical setup, and then programming specifics.

The inverse problem

In this project, only the sonobuoy positions and the relative arrival times of a signal are known, and they constitute the sum of the data for a particular event. The properties of the propagation model for this signal are unknown, meaning that the signal's source location, the sound speed, and the travel time of the signal to each of the buoys are unknown. This setup, where the goal is to describe the properties of a propagation model corresponding to specific data collected from a propagating wave, is termed the inverse problem. Since the primary focus of the software for this project is to find the signal's source location, a solution to the inverse problem is sought. The simplest propagation model is one in which the propagation path between source and receiver is a straight-line, and this is the starting point.

Mathematical setup of the straight-line model

The path of same phase loci on a wave front, termed a "ray", is a straight-line in a medium with a constant sound speed. Here, lines are described by vectors and these vectors are arranged in linear equations describing the system. A sufficiently large group of linear equations is then used to solve for previously unknown values.

The initial time the signal is transmitted is not known. The separation in time between reception of the signal at various receivers is measured. Instead of dealing with the four unknown transmission times between the source and each receiver, it is best to describe each of these times as a sum of the unknown travel time to the first receiver, T_1, with the known time delay between the reception of the signal at the first receiver and the reception at the ith receiver, τ_{1i}. This difference, $T_i - T_1 = \tau_{1i}$, is termed the travel time difference (TTD). In this way, all unknown propagation times are described in terms of only one variable, the travel time to the first receiver. Therefore, the basic relation $\| \mathbf{r}_i - \mathbf{s}_o \| = c\, T_i$ can be written for the straight-line path from the source to every ith receiver ($\| \mathbf{a} \| = a$). Here, the sound speed is c, the source position is $\mathbf{s}_o = (s_{ox}, s_{oy}, s_{oz})$, and the receiver position is $\mathbf{r}_i = (r_{ix}, r_{iy}, r_{iz})$; $i = 1, 2, .., n$ where n is the number of hydrophones. Now, T_i is replaced with $T_1 + \tau_{1i}$ and with a vector identity, it follows that,

$$\| \mathbf{a} - \mathbf{b} \|^2 = \| \mathbf{a} \|^2 + \| \mathbf{b} \|^2 - 2(a_x b_x + a_y b_y + a_z b_z)$$

$$\| \mathbf{r}_i - \mathbf{s}_o \|^2 = c^2 (T_1 + \tau_{1i})^2$$

$$2 r_{ix} s_{ox} + 2 r_{iy} s_{oy} + 2 r_{iz} s_{oz} + 2 T_1 \tau_{1i} c^2 = -c^2 \tau_{1i}^2 + \| \mathbf{r}_i \|^2$$

$i = 1, 2, .., n-1$, since these equations are with respect to receiver R_1. This becomes a set of n-1 linear equations of the form $\mathbf{Ax} = \mathbf{b}$. $\mathbf{x} = [s_{ox} \ \ s_{oy} \ \ s_{oz} \ T_1]^T$ and is the vector of unknowns for which a solution can be found. **A** is a matrix with its *i*th row of the form

$$[2r_{ix} \ \ 2r_{iy} \ 2r_{iz} \ 2\tau_{1i}c^2]$$

and **b** is a column matrix with its *i*th row of the form

$$[-c^2\tau_{1i}^{\ 2} + \| \mathbf{r}_i \|^2]$$

In this system of n-1 equations, there are four unknown values. Since there are only four buoys used in this project, there is a set of only three equations. The problem of too many unknowns to solve for is remedied by setting all four hydrophone at the same depth, 9.75 m in this project.

A lack of precision in measured arrival time of clicks along with errors in the measured positions of the buoys, make an exact solution problematic. Use of these raw data will lead to a non-solvable set of equations. To correct for this problem, a least squares approximation to calculate the closest solution, $\bar{x}$ is used. Then the equation set $\mathbf{A}^T\mathbf{A}\bar{x} = \mathbf{A}^T\mathbf{b}$ solves for the least squares solution, $\bar{x}$, where $\bar{x}$ is still of the same form as **x**, and its elements represent the closest solution, in the event of a nonexistent exact solution (Strang, 1993).

Straight-line programming specifics

Since the straight-line calculations require a constant sound speed, and an average sound speed along a propagation path depends on the source depth, multiple calculations of a straight-line solution are made, each one using a more accurate sound speed based on the previous calculation. A weighted average of sound speed, from the sound speed versus depth profile for depths ranging between the hydrophone depth and the previously calculated source depth, is used for each source depth calculation. Initially, an average sound speed is calculated for a source at a depth of 100m. Subsequent calculations of average sound speed and source position are made until consecutive calculations of source position vary little.

Problems created by non-constant sound speed

The previous formalism works only for a constant sound speed; not only does a non-constant sound speed create an extra variable to solve for, but it also creates an arced propagation path between source and receiver. The equation of an arced path is straight forward, providing the radius of curvature remains constant. This requires that the change in sound speed with respect to depth, $\frac{dc}{dz} \equiv c'$, remains constant. This works for small changes in depth, but not overall with $c = c(z)$ as in Fig. 2. Continuing on in this way, solving sets of equations for each increment of depth preserving a constant c' is quite involved, so another approach is taken.

The forward problem

In the inverse problem, data collected from wave propagation are used to determine the particular model for the propagation of this wave. This set up can be reversed, where the model for wave propagation is known, and data about this wave propagation are calculated by using the model. This set up is termed the forward problem. To apply the forward problem methodology to this project, a source location has to be known. With both a known starting point (the source position) and an end point (a particular buoy position), a depth dependent sound speed is used to calculate the actual path of the direct eigenray connecting these points. In this way the direct eigenray between the source and each buoy is found. As mentioned previously, a ray is formed by the path of same phase loci on a wave front (ray paths are shown in Fig. 2 and 3). An eigenray is a ray that connects two points, usually a source and receiver. Eigenrays that do not have reflected paths are termed direct eigenrays.

Mathematical setup of the ray-tracing model

The method of calculating the path of these rays is termed ray-tracing. The typical ray-tracing model works on the principle that a ray will have a uniform arc in a medium with constant c' (Urick, 1983, Clay and Medwin, 1977). A depth dependent sound speed is incorporated by discretizing the sound speed profile into a piecewise continuous profile of constant c' segments. The ray paths are then calculated piecewise. Each segment of the arc has a particular radius of curvature, based on the particular c' for that depth. In this formalism, it is also necessary to keep up with the initial and final grazing angles of the arc in each segment. The equations of time and position for the arced paths in each region of constant c ' follow:

$$t = t_i + \frac{1}{2g}\left\{\ln\left[\frac{1+\sin\theta_f}{1-\sin\theta_f}\right] - \ln\left[\frac{1+\sin\theta_i}{1-\sin\theta_i}\right]\right\}$$

$$x = x_i + \frac{c_o}{g\cos\theta_i}\left(\sin\theta_f - \sin\theta_i\right)$$

$$z = z_i + \frac{c_o}{g\cos\theta_i}\left(\cos\theta_f - \cos\theta_i\right)$$

$g = \frac{dc}{dz}$; c_o = sound speed at the beginning of the layer; θ = grazing angle; t = time; x = distance along the water surface; z = depth.

Once a direct eigenray is found, its time of propagation is calculated. Subtracting the propagation time to each buoy from the propagation time to the first buoy (the first buoy to receive the signal) yields a set of TTD values. These calculated TTD values are compared with actual TTD values collected in the field to give information on the accuracy of the source position used in this calculation. The initial source position used in this calculation is the one calculated by the straight-line model mentioned earlier. Subsequent adjustments in source position are made by examining the differences in the calculated and actual TTD values. The method for source position adjustments is understood once further mathematical significance of the TTD values and how they correspond to hyperboloids is made.

Ray-tracing programming specifics

Examples of calculated rays corresponding to particular sound speed versus depth profiles are shown in Fig. 3 and 4. In Fig. 3, rays at a variety of initial grazing angles are calculated, originating near the deep sound channel axis. In Fig. 4, ray paths are shown from two different starting depths to show two different features. Near the surface, rays with a shallow grazing angle reflect along the water surface in a path that is termed a surface duct. Rays with large enough grazing angles, as well as rays originating deeper, are reflected along the bottom and (or) the surface. With this type of sound speed versus depth profile, no deep sound channel is present. The ray paths presented in Fig. 3 and 4 correspond well with ray paths in similar set ups (Urick, 1983, Clay and Medwin, 1977). In the ray-tracing program, the sound speed versus depth profile is read in as a file of data points. The list of points is used to calculate discrete intervals of constant c'. In this way, the resolution of arc segments is dependent upon the number of points included in the sound speed versus depth profile.

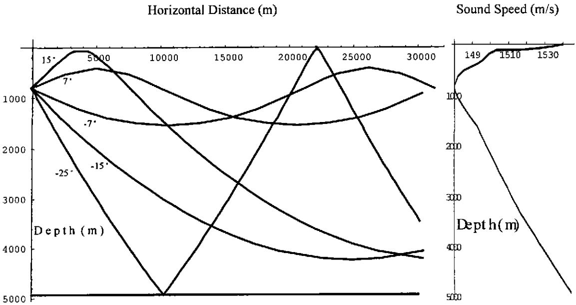


FIG. 3. Ray path calculations with the ray-tracing program for the corresponding profile on the right. Rays with initial grazing angles of -7°, 7°, -15°, and 15° at 800m are internally refracted, revealing a deep sound (SOFAR) channel. The ray with an initial grazing angle of -25° at 800m has bottom and surface reflections.

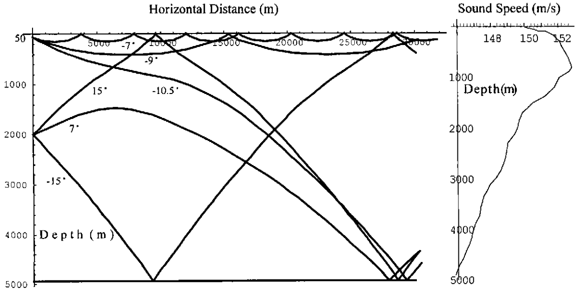


FIG. 4. Ray path calculations with the ray-tracing program with another profile, on the right. Initial grazing angles of -7°, and -9° at 50m reveal surface ducts. Initial grazing angle of -10.5° at 50m, and 15°, 7°, and -15° at 2000m show surface and bottom reflected paths.

Mathematical significance of hyperbolas

The solution set of possible source positions corresponding with a single TTD between two sonobuoys is described by a hyperboloid, providing the sound speed is constant. The mathematical reasoning for this is best shown in the 2-dimensional case.

A ***hyperbola*** is the locus of points P in a plane such that the difference $|\overline{PF_1} - \overline{PF_2}|$ between the distances from P to two distinct points F_1 and F_2 is a constant. (Shenk, 1988) (Fig. 5).

This definition is applied to the setup with F_1 and F_2 each representing a sonobuoy position and with the constant distance $|\overline{PF_1} - \overline{PF_2}|$ representing TTD * c. In this way, hyperbolas corresponding to each TTD are set up, and their intersection represents the source position. Since this method uses a constant sound speed, it will not provide an exact solution (Fig. 6). In this project, 3-dimensions are used, so the surfaces are hyperboloids. Three hyperboloids are required for a single point solution. Solving a set of three generalized hyperboloids is quite involved, since they are each required to be on independent axes. Calculating a solution in this way is further complicated by the depth dependent sound speed, which needs to be transformed into piecewise continuous constant fragments.

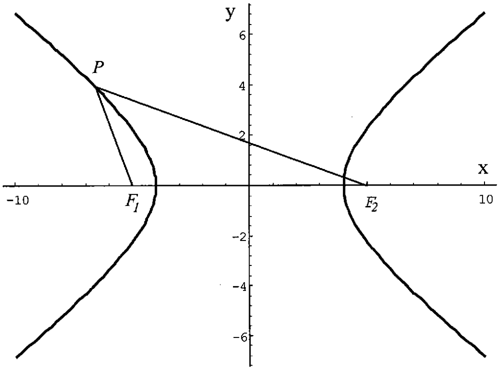


FIG. 5. A hyperbola. The hyperbola $9x^2 - 16y^2 = 144$ with foci F_1 and F_2. P is an arbitrary point on the hyperbola.

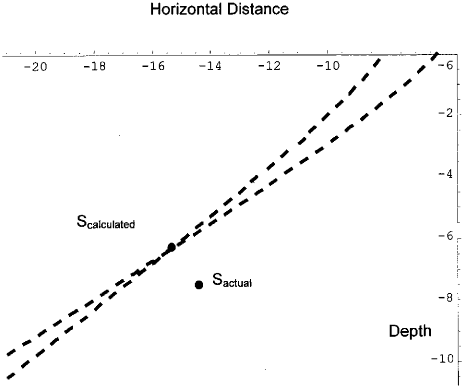


FIG. 6. The intersection of hyperbolas generated with constant sound speed. The hyperbolas generated with constant sound speed intersect near, but not on, the actual source position.

The TTDF algorithm

There is another way to approach a solution to this problem. Instead of calculating the source position exactly, use an approximate calculation of source position, **S**, and adjust it to a good approximation to the actual position. This process is now described. **S** is found with the straight-line model. Once **S** is found, the ray-tracing model is used to find eigenrays between **S** and each of the buoys. This provides a set of TTD values corresponding to **S** and the array. Since the source position is not exact, these calculated TTD values will differ from the TTD values corresponding to $\mathbf{S_0}$. This difference, $TTD_{generated}$ - TTD_{actual}, is termed the TTD difference (TTDF).

A hyperbola is closer to its axis with an increase in TTD*c. The TTDF represents a shift in TTD*c. Correspondingly, a hyperbola with a positive TTDF is closer to its axis than a hyperbola with TTDF ≈ 0 (Fig. 7). With the understanding of how the TTDF value shifts a hyperbola, adjustments to **S** can now be considered. When working in 2 dimensions, only 2 hyperbolas are needed. They provide a $TTDF_1$ and $TTDF_2$, each representing shifts in a hyperbola. The goal is to minimize the $TTDF_i$'s in order to work with hyperbolas with very little shift from the actual hyperbolas. From Fig. 6, the calculated source position needs to be adjusted until it corresponds with the actual source position. It is logical to adjust one coordinate of **S**, x or z, at a time.

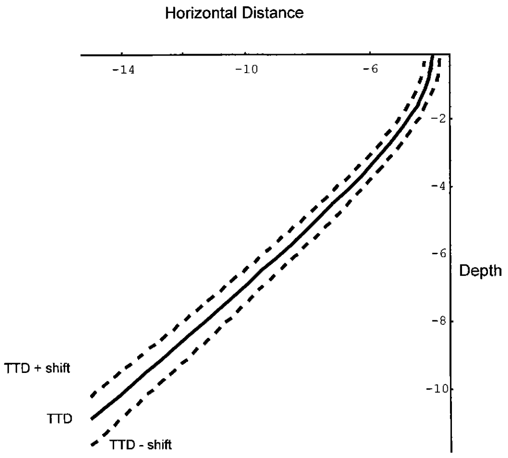


FIG. 7. Shift in the hyperbola with a change in sound speed. The central hyperbola gets closer to its axis with an increase in TTD*c, and further away with a decrease in TTD*c.

Adjusting a coordinate of **S** until $TTDF_1 \approx TTDF_2$ is an option which provides a stopping point. This situation is shown in Fig. 8. The actual hyperbolas are solid lines, where as the approximations are dashed. Notice that the intersection of hyperbolas formed by the TTDF's is always in front of or behind both of the actual hyperbolas. Furthermore, the solution set of all possible intersections with $TTDF_1 \approx TTDF_2$ forms a line intersecting the actual solution and approximately normal to the hyperbolas at that point.

Once one of the coordinates is adjusted to where $TTDF_1 \approx TTDF_2$, the next step is to adjust the other coordinate to where $TTDF_1 \approx -TTDF_2$. A diagram of this interaction is shown in Fig. 9. In this case, notice that the intersection of hyperbolas formed by the TTDF's will always be in between the two actual hyperbolas and that the solution set of all possible intersections with $TTDF_1 \approx -TTDF_2$ forms a curve in between the actual hyperbolas and intersecting the solution.

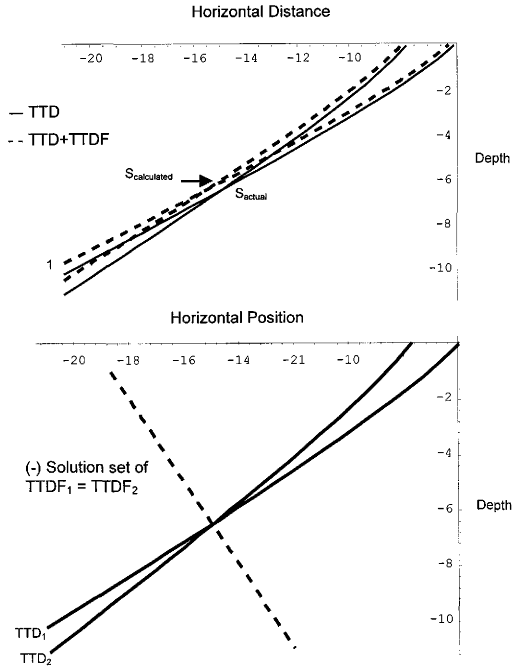


FIG. 8. Interaction of hyperbolas with $TTDF_1 \approx TTDF_2$. (Top) From Fig. 6, the Sx coordinate is adjusted. The actual solution hyperbolas have solid lines. The shifts in the hyperbolas due to the TTDF's are represented with dashed lines. The intersection of the dashed lines will always be either in front of or behind both hyperbolas, depending on the TTDF sign (+ in this example). (Bottom) The dotted line nearly perpendicular to the hyperbolas corresponds to the solution set of $TTDF_1 \approx TTDF_2$.

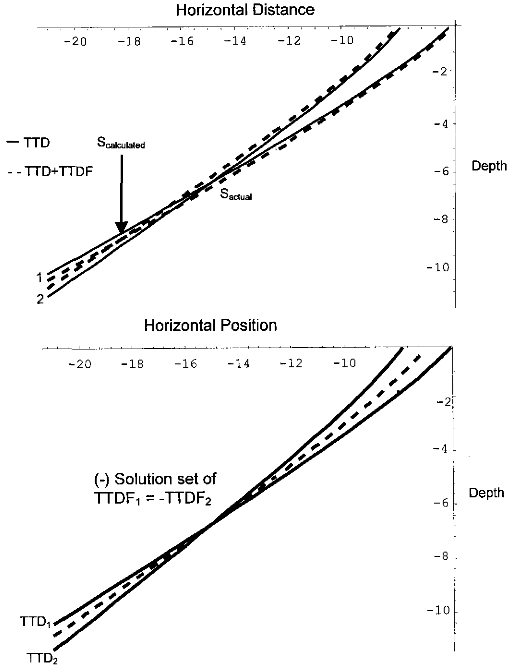


FIG. 9. Interaction of hyperbolas with $TTDF_1 \approx -TTDF_2$. (Top) From the stopping position in Fig. 8, the Sz coordinate is adjusted. The actual solution hyperbolas have solid lines. The shifts in the hyperbolas due to the TTDF's are represented with dashed lines. The intersection of the dashed lines will always be in between the 2 actual hyperbolas. (Bottom) The dotted line in between the hyperbolas corresponds to the solution set of $TTDF_1 \approx -TTDF_2$.

By adjusting the coordinates in successive order in this fashion, the actual solution is approached iteratively without the need of any equations involving hyperbolas (Fig. 10). However, this method can lead to divergence. If so, a convergent solution is found by reversing the coordinate adjusted when finding $TTDF_1 \approx TTDF_2$ or $TTDF_1 \approx -TTDF_2$. This method for finding a solution for source position is termed the "TTDF algorithm".

For the 3-dimensional case, first adjust the depth coordinate so that $TTDF_3 \approx 0$. Then, adjust x and y to get $TTDF_1 \approx TTDF_2$ and $TTDF_1 \approx -TTDF_2$ as explained earlier. All three coordinate adjustments are repeated during each iteration. After a number of iterations, all three TTDF's approach 0, indicating the source position is nearly on all three hyperboloids, and corresponds well with the actual solution. This method circumvents using involved mathematics to find a solution.

Example of one iteration

To better explain the TTDF algorithm, a numerical example of one iteration is given. For a particular buoy arrangement, the following TTD values are measured for a single event, where $TTD_{1i} = T_1 - T_i$, the delay in arrival times of a signal received by 2 buoys: $TTD_{12} = 0.104\ 175$ s, $TTD_{13} = 0.196\ 795$ s, $TTD_{14} = 0.348\ 067$ s

With these TTD values and the buoy positions, the straight-line calculation for source position is , (x, y, depth), S = (105.31, 34.30, 78.90) with respect to buoy 1, the first

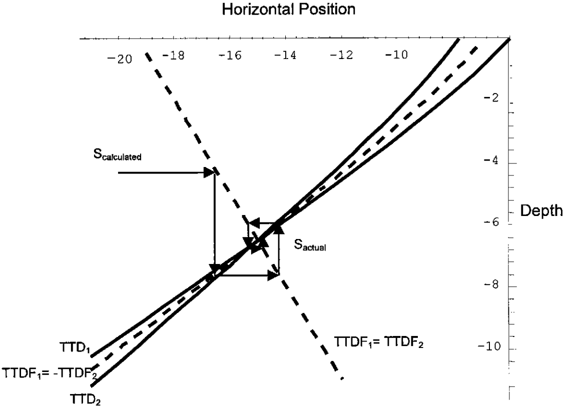


FIG. 10. Iterations using matched TTDF values. $S_{calculated}$ = the starting point. The dotted line nearly perpendicular to the hyperbolas corresponds to the solution set of $TTDF_1 \approx TTDF_2$. The dotted line in between the hyperbolas corresponds to the solution set of $TTDF_1 \approx -TTDF_2$. The solid lines are the actual hyperbolas corresponding to the actual TTD values. The source position is adjusted from $S_{calculated}$ in the x direction until $TTDF_1 \approx TTDF_2$. Then the z coordinate is adjusted until $TTDF_1 \approx -TTDF_2$. This is repeated until $TTDF_1 \approx TTDF_2 \approx 0$, indicating that the source position corresponds with the intersection of the hyperbola, and in its correct location.

buoy to receive the signal. From this initial approximation of the source position, an eigenray between this source position and each of the 4 buoys is calculated. The travel time along each eigenray is calculated, and these 4 travel times are used to calculate a set of TTD values corresponding to this approximate source position. They are:

$$TTD'_{12} = 0.105\ 998 \text{ s},\ \ TTD'_{13} = 0.199\ 111 \text{ s},\ \ TTD'_{14} = 0.350\ 716 \text{ s}$$

Now TTDF values, difference in calculated and actual TTD values, can be calculated for this approximate source location. They are, $TTDF_i = TTD'_{1i} - TTD_{1i}$, :

$$TTDF_1 = 0.001\ 823 \text{ s},\ \ TTDF_2 = 0.002\ 316 \text{ s},\ \ TTDF_3 = 0.002\ 649 \text{ s}$$

This is the starting point for the TTDF algorithm. The following format is now used to keep track of values throughout a series of calculations:

Sx	Sy	Sz
$TTDF_1$	$TTDF_2$	$TTDF_3$

We start with:

105.31	34.30	78.90
0.001 823	0.002 316	0.002 649

The first step is to adjust the Sz coordinate until $TTDF_3 \approx 0$. This corresponds with the intersection of the hyperboloid formed with buoys 1 and 4.

We now have:

105.31	34.30	**89.90**
-0.000 100	-0.000 064	**0.000 000**

Now, adjust Sx until $TTDF_1 \approx -TTDF_2$. This puts the source calculation in between the hyperboloid formed by buoys 1 and 2, and the hyperboloid formed by buoys 1 and 3 (Fig. 9). Adjusting Sx $TTDF_1 \approx TTDF_2$ leads to divergence.

We now have:

105.06	34.30	89.90
0.000 086	**-0.000 086**	-0.000 018

Now, adjust Sy until $TTDF_1 \approx TTDF_2$. This puts the source calculation either in front of or behind both the hyperboloid formed by buoys 1 and 2, and the hyperboloid formed by buoys 1 and 3 (Fig. 8).

We now have:

105.06	**34.51**	89.90
-0.000 080	**-0.000 080**	-0.000 111

These three steps complete one iteration. Notice that the TTDF values after one full iteration are all smaller than before the iteration, indicating that the calculated source position after one iteration is more accurate than before the iteration. This process is now continued by adjusting Sz until $TTDF_3 \approx 0$. These iterations will continue until $TTDF_3 < 0.000\,000\,49$ (sec) after the Sy adjustment.

METHOD OF TESTING

In testing the accuracy of the TTDF algorithm's calculation of source position, 5 array configurations of hydrophones were tested, each with 30 pseudo-randomly generated source positions (Fig. 11 - 15). These source positions were used to calculate accurate TTD values that were used by the TTDF algorithm to solve for source position. The calculated TTD values are analogous to the actual field data used by the TTDF algorithm. In this way, the source position calculations from the TTDF algorithm can be compared with an accurate value of source position.

The calculation of TTD values associated with a particular source position and set of hydrophone locations is done with the ray-tracing method described earlier. A non-constant sound speed is used, based on a sound speed vs. depth profile. Eigenrays between the source and each receiver are calculated. In this way, propagation time for each eigenray is calculated and the corresponding TTD values are found. Even though the TTDF algorithm for locating source position uses this same ray-tracing model, it does not lead to circular reasoning. Only the data available in the field is used by the TTDF algorithm, and the ray-tracing model is used only for directional shifts in the source position calculation; it does not calculate a value for source position. The only confounding of the accuracy of calculations with this method is due to inaccuracies in the ray-tracing model. Such inaccuracies would be introduced by a medium with variation in the sound speed vs. depth profile in the horizontal plane.

Five hydrophone array configurations were tested. In these configurations, each buoy was several hundred meters apart. A basic shape for an array is a square; an array in a line does not provide a unique solution, even with the straight-line method. Subsequent array configurations were chosen based on drift possibilities. The two ways a square configuration is likely to drift apart form either a rhomboidal or trapezoidal shape. Two cases of each are included, with different degrees of dispersal.

Each configuration was tested with 30 pseudo-randomly generated source positions. The 30 points were generated with a pseudo random number generator in the C programming language. All the sound propagation models where written in the C programming language. The points were generated to have a depth of 20m - 4000m and have direct eigenrays to each buoy. The horizontal coordinates of each point where generated in 3 categories. Of the 30 points, 5 points were generated to be within 500 m of the array, 10 points within 1500 m, and 15 points within 5000 m of the array. This break down was chosen in order to have a higher density of points close to the array, which corresponds with expected field conditions.

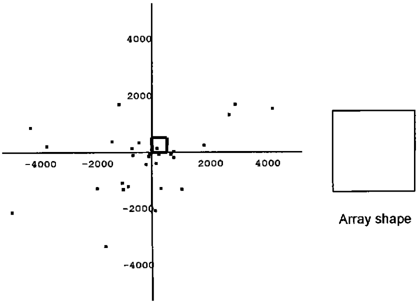


FIG. 11. Square array configuration with 30 test points. Sonobuoys at the corners. Distances in meters.

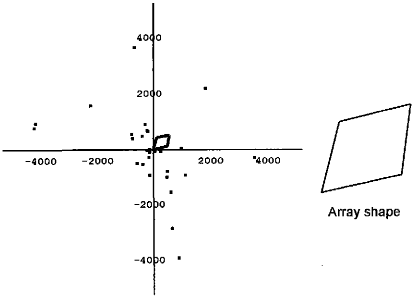


FIG. 12. Wide rhomboidal array configuration with 30 test points. Sonobuoys at the corners. Distances in meters.

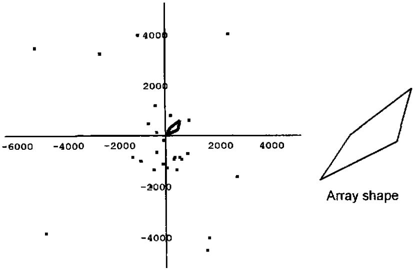


FIG. 13. Narrow rhomboidal array configuration with 30 test points. Sonobuoys at the corners. Distances in meters.

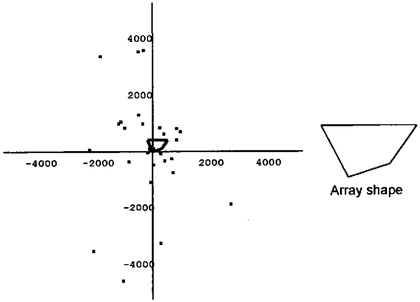


FIG. 14. Wide trapezoidal array configuration with 30 test points. Sonobuoys at the corners. Distances in meters.

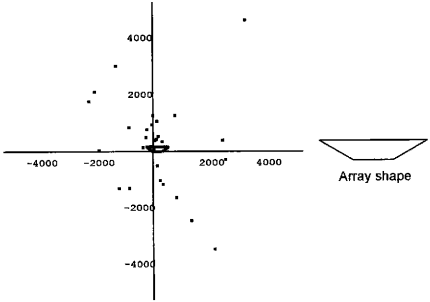


FIG. 15. Narrow trapezoidal array configuration with 30 test points. Sonobuoys at the corners. Distances in meters

RESULTS

Of the 150 generated data point, the average errors for the source position calculation with the TTDF algorithm are, (x, y, depth) in meters, (± 0.72, ± 0.83, ± 1.10), and for the straight-line approximation (± 1.58, ± 1.70, ± 10.44). Overall, the TTDF algorithm reduced the straight-line approximation's error in horizontal positioning (x, y) by a factor of 0.5, and reduced the error in depth by an order of magnitude (Table I). For all 150 points tested, the TTDF algorithm lead to a more accurate calculation of source position than the straight-line approximation. Surprisingly, of the five array configurations, the square configuration did not yield the lowest error for either the straight-line approximation or the TTDF algorithm. Instead, the two rhomboidal configurations provided the best results for the TTDF algorithm, with average errors of (± 0.18, ± 0.17, ± 0.25) m for the wider configuration, and (± 0.11, ± 0.17, ± 0.17) m for the narrower one. For the straight-line approximation, the wide rhomboidal configuration provided the smallest average error, with errors of (± 0.24, ± 0.32, ± 9.44) m. The worst configuration of the five for the TTDF algorithm was the narrow trapezoidal shape, with average errors of (± 1.37, ± 1.42, ± 2.29) m. For the straight-line model, the greatest average error was generated with the narrow rhomboidal shape, with an average error of (± 3.51, ± 3.32, ± 10.59) m.

Table I. Average errors in the source position calculations of both the straight-line model and the TTDF algorithm for all 5 array configurations.

Array configuration	Number of data points	Straight line (x, y, depth) m	TTDF algorithm (x, y, depth)m
Square (Fig. 11)	30	(± 1.20, ± 0.70, ± 11.20)	(± 1.20, ± 0.70, ± 1.86)
Wide rhomboidal (Fig. 12)	30	(± 0.24, ± 0.32, ± 9.44)	(± 0.18, ± 0.17, ± 0.25)
Narrow rhomboidal (Fig. 13)	30	(± 3.51, ± 3.32, ± 10.59)	(± 0.11, ± 0.17, ± 0.17)
Wide trapezoidal (Fig. 14)	30	(± 1.58, ± 2.75, ± 10.28)	(± 0.80, ± 1.67, ± 0.91)
Narrow trapezoidal (Fig. 15)	30	(± 1.37, ± 1.42, ± 10.70)	(± 1.37, ± 1.42, ± 2.29)
All five configurations	150	(± 1.58, ± 1.70, ± 10.44)	(± 0.72, ± 0.83, ± 1.10)

DISCUSSION

The calculations of the 150 source positions tested indicate that the TTDF algorithm yields a more accurate calculation than the straight-line approximation. Without exception in the points tested, the TTDF algorithm produced a more accurate calculation for source depth, usually with an improvement by an order of magnitude over the straight-line approximation. Surprisingly, 58 of the data sets used lead to an unavoidable divergence after the initial source depth correction. However, this does not present a problem for this project. When this divergence occurs, the calculations are stopped, and the depth corrected source position is used. Since no horizontal corrections are used, the x & y straight-line components are kept. For the data sets this case applies to, the average source position errors are (±1.44, ± 1.48, ± 1.32)m, where as the straight-line error in the depth coordinate is ±10.10 m. Of the 58 data sets stopped after an initial depth correction, 22 are from the square array configuration. This disproportional amount suggests geometric reasons behind this divergence.

The ideas behind geometric dilution of precision (GDOP) of array configurations, a calculation used by the Global Positioning System (GPS), might provide a way to determine volumes within which the TTDF algorithm would expect to have convergent solutions. The GDOP concept provides an error coefficient in position calculations, based entirely on the geometric orientation of satellites (Parkinson 1996, Spilker 1996). This GDOP value is used to select the best configuration of satellites available at the time, for a particular position calculation. The GDOP value is calculated with the unit vectors from the ground position to each of the satellites. GDOP values decrease (less error) as the position is surrounded more completely by satellites, as the volume wedge formed by the position and all the satellites gets larger. With the sonobuoy array, GDOP values could be found, possibly indicating source positions likely to lead to divergence using the TTDF algorithm, making the algorithm more efficient.

To a certain degree, the resolution of the source position calculation is user defined. In the calculations included in this paper, the stopping condition of $TTDF_3 < 0.000\,000\,49$ sec is used. This value is set arbitrarily; it roughly corresponds to the limiting resolution of the equipment used in the signal analysis. This value could be increased to speed up the calculation of source position. In doing so, the average errors in source position will increase. However, they should not be greatly effected, and would be no worse on average than the average errors generated in the previously described case, where only the initial depth correction is used.

A typical source position calculation with full iterations takes from 2 to 20 minutes with a Pentium© 166MHz personal computer. The straight-line calculation is done almost immediately, and the first calculation of depth correction takes 5 to 30 seconds. This research is set up to post-process the data, calculating whale positions once back in the lab. In a situation where source accuracy on the order of ± a few meters is sufficient, the source position calculation after only the first depth correction could be used for systems intended for nearly real time calculations of source position, while in the field.

CONCLUSION

The TTDF algorithm provides a more accurate calculation of source position than the straight-line approximation, reducing the error in the calculation of horizontal positioning by a factor of 0.5, and reducing the error in depth by an order of magnitude. The rhomboidal array configurations provide the lowest average source position calculation errors. In some cases, the TTDF algorithm leads to an unavoidable divergence, likely due to geometric orientation of the source and the array. In these cases, the algorithm is designed to stop after the initial depth correction, preventing divergence. With this depth correction, the source position calculation improves the error in the depth coordinate by nearly an order of magnitude. Since this algorithm provides a more accurate calculation of source position than the straight-line approximation, the goals of this project are met.

REFERENCES

Backus, R. H. and Schevill, W. E. (**1966**). "Physeter clicks," in *Whales, Dolphins, and Porpoises*, edited by K. S. Norris. (Univ. of Calif., Berkeley, CA), pp.510-527.

Bowles, A. E., Sumultea, M., Würsig, B., DeMaster, D. P., and Palka, D. (**1994**). "Relative abundance and behavior of marine mammals exposed to transmissions from the Heard Island Feasibility Test," J. Acoust. Soc. Am. **96**(4), 2469-2484.

Clay, C. S. and Medwin, H. (**1977**). *Acoustical Oceanography, Principles and Applications.* (John Wiley and Sons, New York).

Cummings, W. C. and Holliday, D. V. (**1987**). "Sounds and source levels from bowhead whales off Pt. Barrow, Alaska," J. Acoust. Soc. Am. **82**(3), 814-821.

Freitag, L. E. and Tyack, P. L. (**1993**). "Passive acoustic localization of the Atlantic bottlenose dolphin using whistles and echolocation clicks," J. Acoust. Soc. Am. **93**(4, Pt.1), 2197-2205.

Herman, L. M. (**1994**). "Hawaiian humpback whales and ATOC: A conflict of interests," J. Environ. Devel. **3**(2), 63-76.

Holing, D. (**1994**). "The sound and the fury: Debate gets louder over ocean noise pollution and marine mammals," Aicus J. **16**(3), 18-23.

Levenson, C. (**1974**). "Source level and bistatic target strength of the sperm whale (*Physeter catodon*) measured from an oceanographic aircraft," J. Acoust. Soc. Am. **55**(5), 1100-1103.

Mate, B. R., Stafford, K. M., and Ljungblad, D. K. (**1994**). "A change in sperm whale (*Physeter macrocephalus*) distribution correlated to seismic surveys in the Gulf of Mexico," J. Acoust. Soc. Am. **96**(5, Pt. 2), 3268-3269.

Norris, J.C. (**1997**). Personal Communication, Texas A&M Univ., Galveston, TX.

Norris, J.C., Evans, W. E., Benson, R., Sparks, T. D. (**1996**). "Acoustic surveys," pp. 133-187, "Distribution and abundance of cetaceans in the north-central and western Gulf of Mexico, final report. Volume II: Technical report," edited by Davis, R. W. and Fargion, G. S. U.S. Department of the Interior, Minerals Mgmt. Services, Gulf of Mexico OSC Region, New Orleans, LA. 357p.

NRC/OSB. **(1994)**. *Low-Freguency Sound and Marine Mammals/Current Knowledge and Research Needs*. U.S. Natl. Res. Counc., Ocean Stud. Board, Committee on Low-Frequency Sound and Marine Mammals. Green, D. M., DeFerrari, H. A., McFadden, D., Pearse, J. S., Popper, A. N., Richardson, W. J., Ridgway, S. H., and Tyack, P. L., (Natl. Acad. Press, Washington, DC.) 75p.

Parkinson, B. W. **(1996)**. "GPS error analysis," in *Global Positioning System: Theory and Applications, vol 1. Progress in Astronautics and Aeronautics,* edited by Parkinson, B. W., and Spilker, J. J. Jr., (Am. Inst. of Aeronautics and Astronautics, Inc., Washington, DC) **163**, pp. 469-483.

Richardson, W. J., Greene, C. R. Jr., Malme, C. I., and Thomson, D. H. **(1995)**. *Marine Mammals and Noise* (Academic Press, Inc, San Diego, CA).

Shenk, A. **(1988)**. *Calculus and Analytic Geometry*. (Scott, Foresman and Co., Glenview, IL).

Spiesberger, J. L. and Fristrup, K. M. **(1990)**. "Passive localization of calling animals and sensing of their acoustic environment using acoustic tomography," Am. Nat. **135**(1), 107-153.

Spilker, J. J. Jr. **(1996)**. "Satellite constellation and geometric dilution of precision", *Global Positioning System: Theory and Applications, vol 1. Progress in Astronautics and Aeronautics,* edited by Parkinson, B. W., and Spilker, J. J. Jr, (Am. Inst. of Aeronautics and Astronautics, Inc., Washington, DC) **163**, pp. 177-208.

Strang, G. **(1993)**. *Introduction to Linear Algebra*. (Wellesley, Cambridge, MA).

Urick, R. J. **(1983)**. *Principles of Underwater Sound, 3rd ed.* (McGraw-Hill, New York).

Watkins, W. A. **(1980)**. "Acoustics and the behavior of sperm whales," in *Animal Sonar Systems*, edited by R. G. Busnel,. and J. F. Fish (Plenum, New York). pp. 283-290.

Watkins, W. A and Moore, K. E. **(1982)**. "An underwater acoustic survey for sperm whales (*Physeter catodon*) and other cetaceans in the southeastern Caribbean," Cetology **46**, 1-7.

Watkins, W. A. and Schevill, W. E. **(1972)**. "Sound source location by arrival-times on a non-rigid three-dimentional hydrophone array," Deep-Sea Res. **19**(10), 691-706.

Watkins, W. A. and Schevill, W. E. **(1974)**. "Listening to Hawaiian spinner porpoises, *Stenella* cf. *longirostris*, with a three-dimensional hydrophone array," J. Mammal. **55**(2), 319-328.

Watkins, W. A. and Schevill, W. E. **(1975)**. "Sperm whales (*Physeter catodon*) react to pingers," Deep-Sea Res. **22**(3), 123-129.

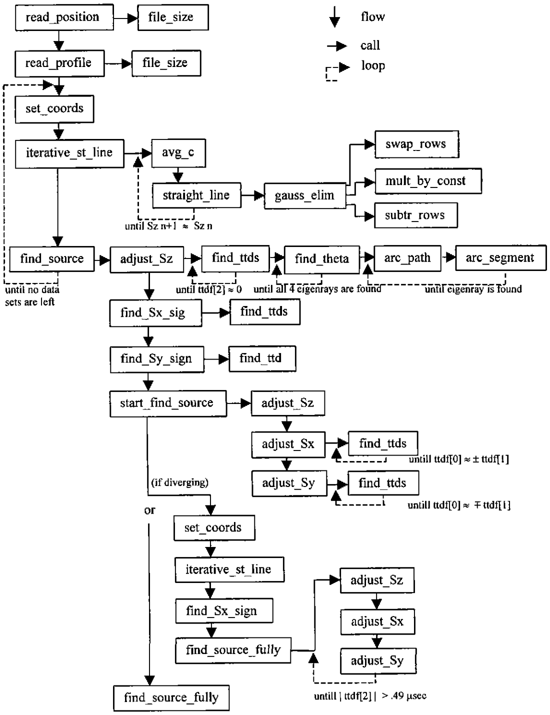
Watkins, W. A. and Schevill, W. E. **(1977)**. "Sperm whale codas," J. Acoust. Soc. Am. **62**(6), 1485-1490.

Watkins, W. A., Moore, K., and Tyack, P. **(1985)**. "Sperm whale acoustic behaviors in the southeast Caribbean," Cetology **49**, 1-15.

Whitehead, H. and Weilgart, L. **(1991)**. "Patterns of visually observable behaviour and vocalizations in groups of female sperm whales," Behaviour **118**(3/4), 275-296.

APPENDIX A

FLOW CHART OF THE PROGRAM



APPENDIX B

SOURCE POSITION PROGRAMMING CODE

SCR.H

```
/*==================================================================
      This C++ code is set up to run as a "project" with several
programs.  These programs are "src_m.cpp", "src_hyp.cpp", and
src_arc.cpp", and should be set up in this order.  They all call the
header file "src.h".
      This program returns the calculated position of a signal's
source, based on signal reception times at four known locations.  This
data is read in as "position[4][5][100]" from the file pointed to by
"ifp" in "read position", and includes multiple events.  Propagation
for sound underwater is determined using a sound speed vs. depth
profile read in as "profile[3][100]" from the file pointed to by "ifp"
in "read_profile".  The output of this program is sent to both the
screen and the file pointed to by "ofp" in "main".  All these file
names are input by the user.
==================================================================*/

#include <stdio.h>
#include <math.h>
#define PI  3.14159265
#define DEPTH 9.75      //hydrophone depth(m), 32 feet


double abs(double);      //returns the absolute value of a number.
void adjust_sz(int);     //adjusts the source position's z coordinate
void adjust_sx(int);     //adjusts the source position's x coordinate
void adjust_sy(int);     //adjusts the source position's y coordinate
void arc_path(double, double, double);  //calculates actual arced path
                                         // & outputs it to file *ofp1
double arc_seg(double, double, double, double, double, double, double);
                         //calculates the exact end of the last arced
                         // segment sets all final values of a ray
double avg_c(double);    //returns a weighted average sound speed for a
                         // depth interval
int file_size(void);     //outputs the number of data pts. in a file.
void find_source(int);   //Controls the flow for finding the source
                         // position using arced paths.
void find_source_fully(int); //finished finding the source position
                             // with arced paths
double find_sx_sign (int);   //finds the direction to shift sx on the x
                             // axis
double find_sy_sign (int);   //finds the direction to shift sy on the y
                             // axis
void find_theta (double, double, double, double); //iteratively finds
                                                  // the initial angle
                                                  // of a specific
                                                  // eigenray
```

```
void find_ttds(int);     //Finds the difference between calculated and
                         // measured TTDs.
void gauss_elim(void);   //puts eqn matrix in a lower block 0's form


void iterative_str_line(int); //finds the straight line solution for
                              // source position iteratively, with a
                              // weighted sound speed
void mult_by_const(int, double);//multiplies a const & matrix row
void read_position(char *);     //reads in position & time data to
                                // position[4][5][100]
void read_profile(char *);      //reads in the velocity profile to
                                // profile[3][100]
void set_coords(int);    //sets the coordinates from deg & min to meters
double sq(double);       //returns the square of the input number
void start_find_source(int); //completes 2 iterations of finding the
                             // source with arced paths.
double straight_line(double, int);//calculates straight line solution,
                                  // returns T1
void subtr_rows(int, int); //subtracts the 2 rows in a matrix
void swap_rows(int, int);  //swaps rows within a matrix
```

SCR_M.CPP

```
/*=====================================================================
      See more complete program decription in "src.h".  In this
program, all the global variables are defined, overall control of flow
is set up, all files are read in or out, all output to the screen is
controled, and a straight-line solution is calculated.  Sub routines
included are: abs, avg_c, file_size, gauss_elim, iterative_str_line,
mult_by_const, read_position, read_profile, set_coords, sq,
straight_line, subtr_rows, & swap_rows.
=====================================================================*/

#include "src.h"

double profile [3][100], //sound velocity profile
                    // [data type][data point #]
                    // data type [0] = c, sound speed (m/s)
                    // data type [1] = z, depth (+m)
                    // data type [2] = g, slope of the segment of z vs. c
       r[4][3],     //x,y, & z coordinates (m) for each buoy
                    // [bouy #], [coordinate]
       s[3],        //x,y, & z coordinates (m) for source approximation
       t1,          //time for signal to get to receiver 1
       arc[4],      //end-point data for eigen ray; [0] = final
                    // distance(x),
                    // [1] = final depth(z), [2] = final time(t),
                    // [3] = final angle(th)
       ttdf[3],     //difference in the time travel difference
                    // calculated and in data file between
                    // buoys 1 & 2 [0], 1 & 3 [1], and 1 & 4 [2]
       time_1,      //TTDF3 after iteration 1
       time_2,      //TTDF3 after iteration 2
       st1[3],      //first source position calculation
       ttdf1[3],    //TTDFs for the first source position calculation
       time_tot1,   //time sum used to estimate accuracy
       sx_sign,     //direction to adjust sx on x axis
       sy_sign,     //direction to adjust sy on y axis
       sol_sign,    //solution set sign, indicates solution set
                    // TTDF1 = TTDF2  or TTDF1 = - TTDF2
       eqn[3][4],   //equation matrix
       position [4][5][100]; //position and time data [receiver #],
                    // [data type], [data set #]
                    // data type [0] = degrees latitude
                    // data type [1] = minutes lat. (w/ decimal seconds)
                    // data type [2] = degrees longitude
                    // data type [3] = minutes long. (w/ decimal seconds)
                    // data type [4] = time travel difference wrt buoy 1

int pos_max,        //the number of data sets in a position file
    prof_max;       //the number of data sets in a sound profile file

FILE *ifp,  //*ifp = input file pointer (sound speed - depth profile)
            //                         & (position and time data table)
```

```
     *ofp,  //*ofp = output file pointer (file contains points along
            // arc path)
     *ofp2; //output file of source positions


int main(void)
{
   int i;               //data set counter
   char file_name[50];  //file name entered by user

   printf("%s%s%s%s%s",
   "\nWhen entering file names, include file type (\".txt\") if any .",
    "\nIf the file is not in the working directory,",
    "\ninclude the path and use \"/\" in place of \"\\\".\n",
    "\"C:/temp/file.txt\" instead of \"C:\\temp\\file.txt\" \n\n",
    "Enter POSITION & TIME data file name:  ");
   scanf(" %s", &file_name);
   read_position(file_name); //reads in the position & time data
   printf("\nEnter the SOUND SPEED vs. DEPTH profile file name:  ");
   scanf(" %s", &file_name);
   read_profile(file_name);  //reads in the profile
   printf("\nEnter file name for data output:  ");
   scanf(" %s", &file_name);
   printf("\n");
   printf("%s%s\n",
    "Coordinates (x, y, depth) meters, where latitude direction is x,",
    "\nand longitude is y.\n");
   ofp2 = fopen(file_name, "a");
   fprintf(ofp2, "%s%s\n",
    "Coordinates (x, y, depth) meters, where latitude direction is x,",
    "\nand longitude is y.\n");
   fclose(ofp2);
   for (i = 0; i <= pos_max; ++i){  //for all data sets
      ofp2 = fopen(file_name, "a");
      printf("Data Point %d ...", i + 1);
      fprintf(ofp2, "Data Point %d ...", i + 1);
      set_coords(i);
      iterative_str_line (i);
      printf("\nStraight line s = (%.2f, %.2f, %.2f)", s[0], s[1],
              s[2]+ DEPTH);
      fprintf(ofp2, "\nStraight line s = (%.2f, %.2f, %.2f)", s[0],
               s[1], s[2]+ DEPTH);
      find_source(i);
      printf("\nCalculated    s = (%.2f, %.2f, %.2f)\n", s[0], s[1],
              s[2] + DEPTH);
      printf(" ttdf1 %f, ttdf2 %f, ttdf3 %f\n", ttdf[0], ttdf[1],
              ttdf[2]);
      printf("With respect to \n %.1f deg  %f min Latitude\n",
              position[0][0][i], position[0][1][i]);
      printf(" %.1f deg  %f min Longitude\n\n",
              position[0][2][i], position[0][3][i]);
      fprintf(ofp2, "\nCalculated    s = (%.2f, %.2f, %.2f)\n", s[0],
            s[1], s[2] + DEPTH);
      fprintf(ofp2, " ttdf1 %f, ttdf2 %f, ttdf3 %f\n", ttdf[0],
```

```
            ttdf[1], ttdf[2]);
      fprintf(ofp2, "With respect to \n %.1f deg  %f min Lattitude\n",
            position[0][0][i], position[0][1][i]);
      fprintf(ofp2, " %.1f deg  %f min Longitude\n\n",
            position[0][2][i], position[0][3][i]);
      fclose(ofp2);
   }
   return 0;
}

void read_position(char *file_name) //reads position & time data into
                                    // position[4][5][100]
{
   int i, j; //loop counters

   ifp = fopen(file_name, "r");
   pos_max = file_size()/20 - 1;      //number of data sets
   ifp = fopen(file_name, "r");

   for (i = 0; i <= pos_max; ++i){    //read in each data set
      for (j = 0; j <= 3; ++j){       //read in for each buoy
         fscanf(ifp, "%lf", &position [j][0][i]); //degrees latitude
         fscanf(ifp, "%lf", &position [j][1][i]); //minutes latitude
         fscanf(ifp, "%lf", &position [j][2][i]); //degrees longitude
         fscanf(ifp, "%lf", &position [j][3][i]); //minutes longitude
         fscanf(ifp, "%lf", &position [j][4][i]); //time travel
      }                                            // difference
   }
   fclose(ifp);
}

int file_size(void)  //returns the number of (float) data points in a
                     // file
{
   int counter = 0, c; //integer value of character
   char ch;            //character

   while ((c = getc(ifp)) != EOF) {
      ch = c;
      if (ch == 46)      // ch == 46 is "."
         counter = counter + 1;
   }
   fclose(ifp);
   return counter;
}

void read_profile(char *file_name) //reads the sound speed - depth
                                   // profile from ifp into
                                   // profile[3][100]
{
   int i;

   ifp = fopen(file_name, "r");
   prof_max = file_size()/2 - 1;
```

```
   ifp = fopen(file_name, "r");


   for (i = 0; i <= prof_max; ++i){ //read in values from profile file
      fscanf(ifp, "%lf", &profile [0][i]);       //read in sound speed
      fscanf(ifp, "%lf", &profile [1][i]);       //read in depth
   }

   for (i = 0; i <= prof_max-1; ++i)     //calculate slope values
      profile[2][i] = (profile[0][i]-profile[0][i+1])
                      / (profile[1][i]-profile[1][i+1]);
   fclose(ifp);
}

void set_coords(int i) //set the buoy coordinates in Cartesian
                       // coordinates from Lat-Lon GPS coordinates
                       // i = data set number
{
   r[0][0] = 0.0;
   r[0][1] = 0.0;
   r[0][2] = 0.0;
   r[1][0] = (position[1][0][i] - position[0][0][i]) * 60 * 1852
           + (position[1][1][i] - position[0][1][i]) * 1852;
   r[1][1] = (position[1][2][i] - position[0][2][i]) * 60 * 1852
           + (position[1][3][i] - position[0][3][i]) * 1852;
   r[1][2] = 0.0;
   r[2][0] = (position[2][0][i] - position[0][0][i]) * 60 * 1852
           + (position[2][1][i] - position[0][1][i]) * 1852;
   r[2][1] = (position[2][2][i] - position[0][2][i]) * 60 * 1852
           + (position[2][3][i] - position[0][3][i]) * 1852;
   r[2][2] = 0.0;
   r[3][0] = (position[3][0][i] - position[0][0][i]) * 60 * 1852
           + (position[3][1][i] - position[0][1][i]) * 1852;
   r[3][1] = (position[3][2][i] - position[0][2][i]) * 60 * 1852
           + (position[3][3][i] - position[0][3][i]) * 1852;
   r[3][2] = 0.0;
}

void iterative_str_line (int i)  //finds str_line sol'n with accurate c
{                                // i = data set number
   double z_temp=0.0, //temporary depth
          c;          //sound speed

   c = avg_c(100.0);   //initial value of sound depth is 100 m
   s[2] = straight_line(c, i);
   while (abs(z_temp - s[2]) > .01){ //continue until there is little
      z_temp = s[2];                 // change
      c = avg_c(s[2]);
      s[2] = straight_line(c, i);
   }
}

double avg_c(double z) //returns average c; z = maximum depth
{
```

```
   double z_temp,            //temporary depth
          weighted_c = 0.0, //reset the running total of weighted sound
                            // speed
          total_z = 0.0,    //reset the running total depth
          c_stop,           //sound speed at stopping depth
          c_avg;            //average sound speed for an interval
   int i = 0, j;  //loop counters

   while (profile[1][i] < z) //count out the number of data points to
      i += 1;                // use
   for (j=1; j <= i-1; ++j){ //for each interval between data points
      z_temp = profile[1][j] - profile[1][j-1];
      c_avg = (profile[0][j-1] + profile[0][j])/2;
      weighted_c += z_temp * c_avg;
      total_z += z_temp;
   }
   c_stop = profile[0][i-1] + profile[2][i-1]
            * (z - profile[1][i-1]);
   z_temp = z - profile[1][i-1];
   c_avg = (profile[0][i-1] + c_stop)/2;
   weighted_c += z_temp * c_avg;
   total_z += z_temp;
   return weighted_c / total_z;
}

double straight_line(double c, int m) //calculates straight line
                                      // solution, returns s[2]
                                      // c = sound speed
                                      // m = data set number
{
   int i, j, k;             //counters
   double a[3][3],          //matrix A
          at[3][3],         //matrix A transpose
          b[3],             //vector b
          ata[3][3],        //matrix At * A
          atb[3],           //vector At * b
          sum,              //summation during matrix multiplication
          r_length;         //length of the position vector of a buoy

   // ===============
   //  make matrix A
   // ===============
   a[0][0] = r[1][0];
   a[0][1] = r[1][1];
   a[0][2] = position[1][4][m] * sq(c);
   a[1][0] = r[2][0];
   a[1][1] = r[2][1];
   a[1][2] = position[2][4][m] * sq(c);
   a[2][0] = r[3][0];
   a[2][1] = r[3][1];
   a[2][2] = position[3][4][m] * sq(c);

   // ===============
   //  make vector b
```

```
    // ===============
    for (i = 0; i <= 2; ++i){
       r_length = sqrt( sq(r[i + 1][0]) + sq(r[i + 1][1])
                  + sq(r[i + 1][2]) );
       b[i] = .5 * sq(r_length) -.5 * sq((c * position[i + 1][4][m]));
    }

    // ==================
    //  make A transpose
    // ==================
    for (i=0; i<=2; ++i){
       for (j=0; j<=2; ++j)
          at[j][i] = a[i][j];
    }

    // =============
    //  make At * A
    // =============
    for (i=0; i<=2; ++i){
       for (j=0; j<=2; ++j){
          sum = 0.0;
          for(k=0; k<=2; ++k)
             sum = sum + at[i][k] * a[k][j];
          ata[i][j] = sum;
       }
    }

    // =============
    //  make At * b
    // =============
    for (i=0; i<=2; ++i){
       sum = 0.0;
       for(k=0; k<=2; ++k)
          sum = sum + at[i][k] * b[k];
       atb[i] = sum;
    }

    // ======================
    //  make equation matrix
    // ======================
    for (i=0; i<=2; ++i){
       for (j=0; j<=2; ++j)
          eqn[i][j] = ata[i][j];
       eqn[i][3] = atb[i];
    }

    gauss_elim();   //puts eqn matrix in a lower block 0's form
    t1 = eqn[2][3] / eqn[2][2];
    s[1] = (eqn[1][3] - t1 * eqn[1][2]) / eqn[1][1];
    s[0] = (eqn[0][3] - s[1] * eqn[0][1] - t1 * eqn[0][2]) / eqn[0][0];
    s[2] = sqrt(sq((c * t1)) - sq(s[0]) - sq(s[1]));
    return s[2];
}
```

```
double sq(double x)  //returns the square of the input (x)
{
   x = x * x;
   return x;
}

double abs (double x)  //returns the absolute value of the input (x)
{
   if(x < 0)
      x = -x;
   return x;
}

void gauss_elim(void)  //puts eqn matrix in a lower block 0's form
{
   int row,     //row number
       col=0,  //column number
       rcnt,   //row counter
       j;      //counter
   double cnst; //constant used in making a pivot = 1

   for (row = 0; row <= 2; ++row){
      rcnt = row;
      while (rcnt < 2) {    //search for  a nonzero pivot element
         if (eqn[rcnt][col] != 0.0)
            break;           //nonzero pivot is found
         else
            rcnt = rcnt + 1;    //go down one row
      }
      if (row != rcnt)  //if nonzero pivot is on another row (rcnt)
         swap_rows(row, rcnt);
      for (j=row; j<=2; ++j){ //make pivot = 0
         cnst = 1/eqn[j][col];
         mult_by_const(j, cnst);
         eqn[j][col] = 1.0;
      }
      for (j=row + 1; j<=2; ++j) //make all elements below pivot = 0
         subtr_rows(j, row);
      col = col + 1;
   }
}

void swap_rows(int rcnt, int row) //swap the rows in a matrix;
                                   // want pivot on the row rcnt
{
   int j; //column #
   double temp; //temporary value holder

   for(j=0; j <= 3; ++j){
      temp = eqn[rcnt][j];
      eqn[rcnt][j] = eqn[row][j];
      eqn[row][j] = temp;
   }
}
```

```
void mult_by_const(int i, double cnst) //multiply a row of a matrix by
                                       // a constant.  i is the row
                                       // cnst is the constant
{
   int j; //column #

   for(j = 0; j<=3; ++j)
      eqn[i][j] = eqn[i][j] * cnst;
}

void subtr_rows(int i, int row) //subtract rows of a matrix.
                                // i=# of all subsequent rows;
                                // row = row number
{
      int j;  //column number

      for (j = 0; j<=3; ++j)
            eqn[i][j] = eqn[i][j] - eqn[row][j];
}
```

SCR_HYP.CPP

```
/*=====================================================================
      See more complete program description in "src.h".  This program
handles the iterative calculation of the source position using TTD's
corresponding to hyperboloids.  The subroutines included in this
program are:  adjust_sx, adjust_sy, adjust_sz, find_source,
find_source_fully, find_sx_sign, find_sy_sign, find_theta, find_ttds, &
start_find_source.
=====================================================================*/

#include "src.h"

void find_source(int m) //This subroutine controls the flow for
                        // calculating the source position using
                        // arced paths.
                        // m = data set number
{
   extern double time_1,   //ttdf[2] after first iteration
                 time_2,   //ttdf[2] after second iteration
                 sx_sign,  //direction to shift sx to find solution
                 sy_sign,  //direction to shift sy to find solution
                 sol_sign; //order of adjusting sx & sy with
                           // corresponding solution sets

   sol_sign = 1.0; //initialize
   adjust_sz(m);
   sx_sign = find_sx_sign(m);
   sy_sign = find_sy_sign(m);
   start_find_source(m);   //go through the first two iterations
   if( abs(time_2) > abs(time_1) ){ //apparent divergence
      time_2 = 1.0; //reset
      set_coords(m);
      iterative_str_line(m);
      sol_sign = -1.0; //go the other direction
      sx_sign = find_sx_sign(m);
      sy_sign = find_sy_sign(m);
      find_source_fully(m);
   }
   else{      //calculation is converging
      adjust_sz(m);
      find_source_fully(m);
   }
}

void start_find_source(int m) //goes through the first two iterations
                              // without any changes.
{
   extern double time_1,    //ttdf[2] after first iteration
                 time_2,    //ttdf[2] after second iteration
                 ttdf[3],   //differences in calculated & recorded TTDs
                 s[3],      //current source position
                 ttdf[3],   //differences in calculated & recorded TTDs
```

```
                  st1[3],    //first source position calculation
                  ttdf1[3],  //TTDFs for the first source position
                             // calculation
                  time_tot1; //time sum used to estimate accuracy
   int i; //loop counter

   adjust_sz(m);        //first iteration
   adjust_sx(m);
   adjust_sy(m);
   time_1 = ttdf[2];
   for(i=0; i<= 2; ++i){  //save these values as temporary 1
      st1[i] = s[i];
      ttdf1[i] = ttdf[i];
   }
   time_tot1 = abs(ttdf[0]) + abs(ttdf[2]);

   adjust_sz(m);        //second iteration
   adjust_sx(m);
   adjust_sy(m);
   time_2 = ttdf[2];
}

void find_source_fully (int m)//continue to solve for the source
                              // location, and also check for
                              // divergence again.  If there is
                              // divergence in both directions, start
                              // over and stop after one iteration.
                              // m = data set number
{
   extern double ttdf[3],  //differences in calculated & recorded TTDs
                 s[3],     //current source position
                 time_1,   //ttdf[2] after first iteration
                 time_2,   //ttdf[2] after second iteration
                 st1[3],   //first source position calculation
                 ttdf1[3], //TTDFs for the first source position
                           // calculation
                 time_tot1,//time sum used to estimate accuracy
                 sx_sign,  //direction to shift sx to find solution
                 sy_sign,  //direction to shift sy to find solution
                 sol_sign; //order of adjusting sx & sy with
                           // corresponding solution sets
   double time_3,     //ttdf[0] after first iteration
          time_4,     //ttdf[0] after fourth iteration
          st2[3],     //first source position calculation in this
                      // subroutine
          ttdf2[3],   //TTDFs for the first source position calculation
                      // here
          time_tot2;  //time sum used to estimate accuracy
   int i,          //loop counter
       cnt = 0;    //iteration number counter

   adjust_sz(m);
   adjust_sx(m);
   adjust_sy(m);
```

```
time_1 = ttdf[2];
for(i=0; i<= 2; ++i){  //save these values as temporary 2
   st2[i] = s[i];
   ttdf2[i] = ttdf[i];
}
time_tot2 = abs(ttdf[0]) + abs(ttdf[2]);

while(abs(ttdf[2]) > .00000049){ //stopping condition,
                                 // .49 microsecond accuracy
   adjust_sz(m);
   adjust_sx(m);
   adjust_sy(m);
   cnt += 1;
   if (cnt == 1){   //second iteration in this subroutine
      time_2 = ttdf[2];
      time_3 = ttdf[0];
      if (abs(time_2) > abs(time_1)){ //provides a second check for
                                      // divergence
         if (time_tot1 > time_tot2) { //use the closest calculation
            s[0] = st2[0];
            s[1] = st2[1];
            s[2] = st2[2];
            ttdf[0] = ttdf2[0];
            ttdf[1] = ttdf2[1];
            ttdf[2] = ttdf2[2];
            break;   //stop this iterative calculation
         }
         else {  //use the other calculation
            s[0] = st1[0];
            s[1] = st1[1];
            s[2] = st1[2];
            ttdf[0] = ttdf1[0];
            ttdf[1] = ttdf1[1];
            ttdf[2] = ttdf1[2];
            break;   //stop this iterative calculation
         }
      }
   }
   if (cnt == 2){   //third iteration in this subroutine
      time_4 = ttdf[0];
      if (abs(time_4) > abs(time_3)){ //provides a third check for
                                      // divergence
         if (time_tot1 > time_tot2) { //use the closest calculation
            s[0] = st2[0];
            s[1] = st2[1];
            s[2] = st2[2];
            ttdf[0] = ttdf2[0];
            ttdf[1] = ttdf2[1];
            ttdf[2] = ttdf2[2];
            break;  //stop this iterative calculation
         }
         else { //use the other calculation
            s[0] = st1[0];
            s[1] = st1[1];
```

```
                s[2] = st1[2];
                ttdf[0] = ttdf1[0];
                ttdf[1] = ttdf1[1];
                ttdf[2] = ttdf1[2];
                break;  //stop this iterative calculation
             }
          }
       }
    }
}

void adjust_sz (int m) //adjust the Sz coordinate until it matches the
                       // hyperboloid corresponding to TTD3
{
   extern double ttdf[3], //differences in calculated & recorded TTDs
                 s[3];    //current source position
   double time;    //ttdf[2] = t[3]-t[0] - position[3][4][m];
                   // calculated - data

   find_ttds(m);
   time = ttdf[2];
   if (time > 0.0){ //positive time difference
      while (time > 0.0){ //while too shallow, make deeper
         s[2] += 1.0;   //adjust sz
         find_ttds(m);  //find new eigenrays
         time = ttdf[2];
      }
      s[2] -= 1.0;  //back up one step
      find_ttds(m);
      time = ttdf[2];

      while (time > 0.0){ //while too shallow, make deeper
         s[2] += .1;
         find_ttds(m);
         time = ttdf[2];
      }
      s[2] -= .1;   //back up one step
      find_ttds(m);
      time = ttdf[2];

      while (time > 0.0){ //while too shallow, make deeper
         s[2] += .01;
         find_ttds(m);
         time = ttdf[2];
      }
      s[2] -= .01;  //back up one step
      find_ttds(m);
      time = ttdf[2];

      while (time > 0.0){ //while too shallow, make deeper
         s[2] += .001;
         find_ttds(m);
         time = ttdf[2];
      }
```

```
    s[2] -= .001; //back up one step
    find_ttds(m);
    time = ttdf[2];

    while (time > 0.0){ //while too shallow, make deeper
       s[2] += .0001;
       find_ttds(m);
       time = ttdf[2];
     }
    s[2] -= .0001;  //back up one step
    find_ttds(m);
    time = ttdf[2];
 }
 else{ //negative time difference
    while (time < 0.0){ //while too deep, make shallower
       s[2] -= 1.0;
       find_ttds(m);
       time = ttdf[2];
    }
    s[2] += 1.0;  //back up one step
    find_ttds(m);
    time = ttdf[2];

    while (time < 0.0){ //while too deep, make shallower
       s[2] -= .1;
       find_ttds(m);
       time = ttdf[2];
    }
    s[2] += .1;   //back up one step
    find_ttds(m);
    time = ttdf[2];

    while (time < 0.0){ //while too deep, make shallower
       s[2] -= .01;
       find_ttds(m);
       time = ttdf[2];
    }
    s[2] += .01;  //back up one step
    find_ttds(m);
    time = ttdf[2];

    while (time < 0.0){ //while too deep, make shallower
       s[2] -= .001;
       find_ttds(m);
       time = ttdf[2];
    }
    s[2] += .001;  //back up one step
    find_ttds(m);
    time = ttdf[2];

    while (time < 0.0){ //while too deep, make shallower
       s[2] -= .0001;
       find_ttds(m);
       time = ttdf[2];
```

```
      }
      s[2] += .0001; //back up one step
      find_ttds(m);
      time = ttdf[2];
   }
}

double find_sx_sign (int m) //find direction towards line of solutions
                            // m = solution set number
{
   extern double ttdf[3],   //differences in calculated & recorded TTDs
                 s[3],      //current source position
                 sol_sign;  //order of adjusting sx & sy with
                            // corresponding solution sets
   double time1,       //ttdf time after first iteration
          time2,       //ttdf time after second iteration
                       // time needs to approach 0.0
          sign = 1.0; //direction to shift sx

   if( abs(s[0]) > 1000.0){ // |sx| > 10000
      s[0] -= .1;     //adjust sx
      find_ttds(m);
      time1 = ttdf[0] + sol_sign * ttdf[1]; //indicates distance from
                                            //this solution set
      s[0] -= .1;     //adjust sx
      find_ttds(m);
      time2 = ttdf[0] + sol_sign * ttdf[1];

      s[0] += .2;      //back up 2 steps
      find_ttds(m);
   }
   else{ // |s[0]| <= 1000.0
      s[0] -= .01;     //adjust sx
      find_ttds(m);
      time1 = ttdf[0] + sol_sign * ttdf[1]; //indicates distance from
                                            // this solution set
      s[0] -= .01;     //adjust sx
      find_ttds(m);
      time2 = ttdf[0] + sol_sign * ttdf[1];

      s[0] += .02;    //back up 2 steps
      find_ttds(m);
   }
   if (time1 < time2) //if getting further away from solution set,
      sign = -1.0;    // then switch directions
   return sign;
}

double find_sy_sign (int m) //find the direction towards line of
                            // solutions m = solution set number
{
   extern double ttdf[3],  //differences in calculated & recorded TTDs
                 s[3],     //current source position
                 sol_sign; //order of adjusting sx & sy with
```

```
                          // corresponding solution sets
   double time1,      //ttdf time after first iteration
          time2,      //ttdf time after second iteration
                      // time needs to approach 0.0
          sign = 1.0; //direction to shift sx

   if( abs(s[0]) > 1000.0){ // |sy| > 10000
      s[1] -= .1;     //adjust sy
      find_ttds(m);
      time1 = ttdf[0] - sol_sign * ttdf[1]; //indicates distance from
                                            // this solution set
      s[1] -= .1;     //adjust sy
      find_ttds(m);
      time2 = ttdf[0] - sol_sign * ttdf[1];

      s[1] += .2;     //back up 2 steps
      find_ttds(m);
   }
   else{ //|sy| <= 1000.0
      s[1] -= .01;    //adjust sy
      find_ttds(m);
      time1 = ttdf[0] - sol_sign * ttdf[1]; //indicates distance from
                                            // this solution set
      s[1] -= .01;    //adjust sy
      find_ttds(m);
      time2 = ttdf[0] - sol_sign * ttdf[1];

      s[1] += .02;    //back up 2 steps
      find_ttds(m);
   }
   if (time1 < time2) //if getting further away from solution set, then
      sign = -1.0;    // switch directions
   return sign;
}

void adjust_sx (int m) //adjust sx until intersection with solution set
                       // TTDF1 = TTDF2 or TTDF1 = -TTDF2 is found.
                       // m = data set number
{
   extern double ttdf[3],  //differences in calculated & recorded TTDs
                 s[3],     //current source position
                 sx_sign,  //direction to shift sx to find solution
                 sol_sign; //order of adjusting sx & sy with
                           // corresponding solution sets
   double time;  //indicates distance from solution set
   int cnt=0,    //iteration number
       stop = 0; //stop = 0 -> continue, 1 -> stop

   time = ttdf[0] + sol_sign * ttdf[1];
   if (time > 0.0){ //on one side of the solution set
      while (time > 0.0){ //while on this side
         s[0] -= sx_sign* 1.0; //adjust sx
         find_ttds(m);  //find new eigenrays
         time = ttdf[0] + sol_sign * ttdf[1];
```

```
      cnt +=1;
      if (cnt >= 20){ //going the wrong direction on x axis
         stop = 1; //need to change direction and bypass all
                   // subsequent steps in this subroutine.
         s[0] += sx_sign*(cnt+1)*1.0;
         sx_sign *= -1.0;  //change direction
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];
         break;
      }
   }
   cnt = 0;  //reset
   s[0] += sx_sign* 1.0; //back up one step
   find_ttds(m);
   time = ttdf[0] + sol_sign * ttdf[1];

   while ((time > 0.0)&&(stop == 0)){ //while on this side
      s[0] -= sx_sign* .1; //adjust sx
      find_ttds(m);
      time = ttdf[0] + sol_sign * ttdf[1];
      cnt +=1;
      if (cnt >= 20){ //redundant check -> going the wrong direction
                      // on x axis
         stop = 1;    //need to change direction
         s[0] += sx_sign*(cnt+1)*.1;
         sx_sign *= -1.0;  //change direcition
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];
         break;
      }
   }
   cnt = 0;   //reset
   s[0] += sx_sign* .1;  //back up one setp
   find_ttds(m);
   time = ttdf[0] + sol_sign * ttdf[1];

   if (stop == 0){ //if progressing properly
      while (time > 0.0){ //while on this side
         s[0] -= sx_sign* .01; //adjust sx
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];
      }
      s[0] += sx_sign* .01;   //back up one step
      find_ttds(m);
      time = ttdf[0] + sol_sign * ttdf[1];

      while (time > 0.0){  //while on this side
         s[0] -= sx_sign* .001;  //adjust sx
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];
      }
      s[0] += sx_sign* .001;  //back up one step
      find_ttds(m);
      time = ttdf[0] + sol_sign * ttdf[1];
```

```
        while (time > 0.0){ //while on this side
           s[0] -= sx_sign* .0001; //adjust sx
           find_ttds(m);
           time = ttdf[0] + sol_sign * ttdf[1];
        }
        s[0] += sx_sign* .0001;  //back up one step
        find_ttds(m);
        time = ttdf[0] + sol_sign * ttdf[1];
     }
  }
  else{ // time is negative
     while (time < 0.0){  //while on this side
        s[0] += sx_sign* 1.0;  //adjust sx
        find_ttds(m);  //find new eigenrays
        time = ttdf[0] + sol_sign * ttdf[1];
        cnt +=1;
        if (cnt >= 20){ //going the wrong direction on x axis
           stop = 1;    //need to change direction and bypass all
                        // subsequnt steps in this subroutine
           s[0] -= sx_sign*(cnt+1)*1.0;
           sx_sign *= -1.0;  //change direction
           find_ttds(m);
           time = ttdf[0] + sol_sign * ttdf[1];
           break;
        }
     }
     cnt = 0;  //reset
     s[0] -= sx_sign* 1.0; //back up one step
     find_ttds(m);
     time = ttdf[0] + sol_sign * ttdf[1];

     while ((time < 0.0)&&(stop == 0)){ //while on this side
        s[0] += sx_sign* .1;  //adjust sx
        find_ttds(m);
        time = ttdf[0] + sol_sign * ttdf[1];
        cnt +=1;
        if (cnt >= 20){ //redundant check -> going the wrong direction
                        // on x axis
           stop = 1;    //need to change direction
           s[0] -= sx_sign*(cnt+1)*.1;
           sx_sign *= -1.0; //change direction
           find_ttds(m);
           time = ttdf[0] + sol_sign * ttdf[1];
           break;
        }
     }
     cnt = 0; //reset
     s[0] -= sx_sign* .1; //back up one step
     find_ttds(m);
     time = ttdf[0] + sol_sign * ttdf[1];

     if (stop == 0){ //if progressing properly
        while (time < 0.0){ //while on this side
```

```
            s[0] += sx_sign* .01;  //adjust sx
            find_ttds(m);
            time = ttdf[0] + sol_sign * ttdf[1];
         }
         s[0] -= sx_sign* .01; //back up one step
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];

         while (time < 0.0){ //while on this side
            s[0] += sx_sign* .001; //adjust sx
            find_ttds(m);
            time = ttdf[0] + sol_sign * ttdf[1];
         }
         s[0] -= sx_sign* .001;  //back up one step
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];

         while (time < 0.0){ //while on this side
            s[0] += sx_sign* .0001; //adjust sx
            find_ttds(m);
            time = ttdf[0] + sol_sign * ttdf[1];
         }
         s[0] -= sx_sign* .0001;  //back up one step
         find_ttds(m);
         time = ttdf[0] + sol_sign * ttdf[1];
      }
   }
   if (stop == 1)
      adjust_sx(m); //call again, going the other direction on x axis
}

void adjust_sy (int m) //adjust sy until intersection with solution set
                       // TTDF1 = -TTDF2 or TTDF1 = TTDF2 is found.
                       // m = data set number
{
   extern double ttdf[3],  //differences in calculated & recorded TTDs
                 s[3],     //current source position
                 sy_sign,  //direction to shift sy to find solution
                 sol_sign; //order of adjusting sx & sy with
                           // corresponding solution sets
   double time; //indicates distance from solution set
   int cnt= 0,    //iteration number counter
        stop = 0; //stop = 0 -> continue, 1 -> stop

   time = ttdf[0] - sol_sign * ttdf[1];
   if (time > 0.0){
      while (time > 0.0){ //while on this side
         s[1] -= sy_sign*1.0; //adjust sy
         find_ttds(m); //find new eigenrays
         time = ttdf[0] - sol_sign * ttdf[1];
         cnt += 1;
         if (cnt >= 20){ //going the wrong direction on y axis
            stop = 1;    //need to change direction and bypass all
                         // subsequent steps in this subroutine
```

```
            s[1] += sy_sign*(cnt+1)*1.0;
            sy_sign *= -1.0; //change direction
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
            break;
        }
    }
    cnt = 0; //reset
    s[1] += sy_sign*1.0; //back up one step
    find_ttds(m);
    time = ttdf[0] - sol_sign * ttdf[1];

    while ((time > 0.0)&&(stop == 0)){ //while on this side
        s[1] -= sy_sign*.1;    //adjust sy
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];
        cnt += 1;
        if (cnt >= 20){ //redundant check -> going the wrong direction
                        // on y axis
            stop = 1;    //need to change direction
            s[1] += sy_sign*(cnt+1)*.1;
            sy_sign *= -1.0;  //change direction
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
            break;
        }
    }
    cnt = 0;  //reset
    s[1] += sy_sign*.1;  //back up one step
    find_ttds(m);
    time = ttdf[0] - sol_sign * ttdf[1];

    if (stop == 0){ //if progressing properly
        while (time > 0.0){  //while on this side
            s[1] -= sy_sign*.01;  //adjust sy
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
        }
        s[1] += sy_sign*.01;  //back up one step
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];

        while (time > 0.0){  //while on this side
            s[1] -= sy_sign*.001; //adjust sy
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
        }
        s[1] += sy_sign*.001;   //back up one step
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];

        while (time > 0.0){ //while on this side
            s[1] -= sy_sign*.0001;  //adjust sy
            find_ttds(m);
```

```
            time = ttdf[0] - sol_sign * ttdf[1];
         }
         s[1] += sy_sign*.0001;    //back up one step
         find_ttds(m);
         time = ttdf[0] - sol_sign * ttdf[1];
      }
   }
   else{  // negative time
      while (time < 0.0){ //while on this side
         s[1] += sy_sign*1.0; //adjust sy
         find_ttds(m); //find new eigenrays
         time = ttdf[0] - sol_sign * ttdf[1];
         cnt += 1;
         if (cnt >= 20){ //going the wrong direction on y axis
            stop = 1;     //need to change direction and bypass all
                          // subsequnt steps in this subroutine
            s[1] -= sy_sign*(cnt+1)*1.0;
            sy_sign *= -1.0; //change direction
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
            break;
         }
      }
      cnt = 0;  //reset
      s[1] -= sy_sign*1.0;  //back up one step
      find_ttds(m);
      time = ttdf[0] - sol_sign * ttdf[1];

      while ((time < 0.0)&&(stop == 0)){ //while on this side
         s[1] += sy_sign*.1;  //adjust sy
         find_ttds(m);
         time = ttdf[0] - sol_sign * ttdf[1];
         cnt += 1;
         if (cnt >= 20){ //redundant check -> going the wrong direction
                         // on y axis
            stop = 1;   //need to change direction
            s[1] -= sy_sign*(cnt+1)*.1;
            sy_sign *= -1.0; //change direction
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
            break;
         }
      }
      cnt = 0;   //reset
      s[1] -= sy_sign*.1;  //back up one step
      find_ttds(m);
      time = ttdf[0] - sol_sign * ttdf[1];

      if (stop == 0){ //if progressing properly
         while (time < 0.0){ //while on this side
            s[1] += sy_sign*.01;   //adjust sy
            find_ttds(m);
            time = ttdf[0] - sol_sign * ttdf[1];
         }
```

```
        s[1] -= sy_sign*.01;  //back up one step
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];

        while (time < 0.0){ //while on this side
           s[1] += sy_sign*.001;   //adjust sy
           find_ttds(m);
           time = ttdf[0] - sol_sign * ttdf[1];
        }
        s[1] -= sy_sign*.001;  //back up one step
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];

        while (time < 0.0){ //while on this side
           s[1] += sy_sign*.0001;   //adjust sy
           find_ttds(m);
           time = ttdf[0] - sol_sign * ttdf[1];
        }
        s[1] -= sy_sign*.0001;  //back up one step
        find_ttds(m);
        time = ttdf[0] - sol_sign * ttdf[1];
     }
   }
   if (stop == 1)
      adjust_sy(m); //call again, going the other direction on y axis
}

void find_ttds (int m) //This subroutine calculates all three TTDs,
                       // based on the estimated source position.
                       // m = data set number.
                       // The eigenrays are calculated from the buoys
                       // to the source.
{
   extern double s[3],   //current source position
                 arc[4], //data on endpoints of the last ray calculated
                 ttdf[3], //differences in calculated & recorded TTDs
                 r[4][3], //buoy position in Cartesian coordinates
                 position[4][5][100]; //array of data from the
                                      // sonobuoys
   double x_max,  //maximum horizontal distance for a ray
          zf,     //depth of the source
          thi,    //initial grazing angle for the ray
          zi,     //initial depth of the ray (hydrophone depth)
          t[4];   //array to store the travel times of each ray

   zi = DEPTH; //buoy depth
   zf = s[2]; //not including hydrophone depth adjustment

   //================
   // Path to buoy 1
   //================
   x_max = sqrt( sq(r[0][0] - s[0]) + sq(r[0][1] - s[1]) );
   thi = -atan(zf / x_max);   //initial guess
   find_theta(x_max, zi, thi, zf);  //iteratively finds eigenray
```

```
    t[0] = arc[2];

    //===============
    // Path to buoy 2
    //===============
    x_max = sqrt( sq(r[1][0] - s[0]) + sq(r[1][1] - s[1]) );
    thi = -atan(zf / x_max);     //initial guess
    find_theta(x_max, zi, thi, zf);  //iteratively finds eigenray
    t[1] = arc[2];

    //===============
    // Path to buoy 3
    //===============
    x_max = sqrt( sq(r[2][0] - s[0]) + sq(r[2][1] - s[1]) );
    thi = -atan(zf / x_max);     //initial guess
    find_theta(x_max, zi, thi, zf);  //iteratively finds eigenray
    t[2] = arc[2];

    //===============
    // Path to buoy 4
    //===============
    x_max = sqrt( sq(r[3][0] - s[0]) + sq(r[3][1] - s[1]) );
    thi = -atan(zf / x_max);     //initial guess
    find_theta(x_max, zi, thi, zf);  //iteratively finds eigenray
    t[3] = arc[2];

    ttdf[0] = t[1]-t[0] - position[1][4][m];  //calculated value - data
    ttdf[1] = t[2]-t[0] - position[2][4][m];
    ttdf[2] = t[3]-t[0] - position[3][4][m];
}

void find_theta (double x_max, double zi, double thi, double zf)
    //This subroutine iteratively finds the eigenray beginning at
    // (x=0, zi) and ending at (x_max, zf).  An initial begining grazing
    // angle (thi) is used,and then adjusted iteratively until the exact
    // eigenray is found.
{
    extern double arc[4];  //end-point data on the last ray calculated
    double z_temp;    //temporary z

    arc_path(x_max, zi, thi);  //calculate ray with these parameters
    z_temp = arc[1];           //= the final depth of the ray just
                               // calculated
    if(z_temp - zf > .00001){  //if last ray ended too deep
       while(z_temp > zf){ //while calculated ray ends too deep,
                           // decrease the initial grazing angle
                           // (0 at horizontal)
          thi += .1*(PI/180);
          arc_path(x_max, zi, thi);
          z_temp = arc[1];
       }
       thi -= .1*(PI/180);       //go back one step
       arc_path(x_max, zi, thi);
       z_temp = arc[1];
```

```
while(z_temp > zf){      //while calculated ray ends too deep,
   thi += .01*(PI/180); // decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .01*(PI/180);      //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];

while(z_temp > zf){       //while calculated ray ends too deep,
   thi += .001*(PI/180); // decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .001*(PI/180);     //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];

while(z_temp > zf){        //while calculated ray ends too deep,
   thi += .0001*(PI/180); // decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .0001*(PI/180);     //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];

while(z_temp > zf){          //while calculated ray ends too deep,
   thi += .00001*(PI/180);  // decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .00001*(PI/180);     //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];

while(z_temp > zf){          //while calculated ray ends too deep,
   thi += .000001*(PI/180); // decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .000001*(PI/180);     //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];

while(z_temp > zf){          //while calculated ray ends too deep,
   thi += .0000001*(PI/180);// decrease the initial grazing angle
   arc_path(x_max, zi, thi);
   z_temp = arc[1];
}
thi -= .0000001*(PI/180);   //go back one step
arc_path(x_max, zi, thi);
z_temp = arc[1];
```

```
      while(z_temp > zf){           //while calculated ray ends too deep,
         thi += .00000001*(PI/180);// decrease initial grazing angle
         arc_path(x_max, zi, thi);
         z_temp = arc[1];
      }
      thi -= .00000001*(PI/180);  //go back one step
      arc_path(x_max, zi, thi);
   }
   else if (z_temp - zf < -.00001){ //if last ray ended too shallow
      while(z_temp < zf){     //while calculated ray ends too shallow,
         thi -= .1*(PI/180); // increase the initial grazing angle
         arc_path(x_max, zi, thi);  // (0 at horizontal)
         z_temp = arc[1];
      }
      thi += .1*(PI/180);         //go back one step
      arc_path(x_max, zi, thi);
      z_temp = arc[1];

      while(z_temp < zf){      //while calculated ray ends too shallow,
         thi -= .01*(PI/180); // increase the initial grazing angle
         arc_path(x_max, zi, thi);
         z_temp = arc[1];
      }
      thi += .01*(PI/180);        //go back one step
      arc_path(x_max, zi, thi);
      z_temp = arc[1];

      while(z_temp < zf){       //while calculated ray ends too shallow,
         thi -= .001*(PI/180); // increase the initial grazing angle
         arc_path(x_max, zi, thi);
         z_temp = arc[1];
      }
      thi += .001*(PI/180);       //go back one step
      arc_path(x_max, zi, thi);
      z_temp = arc[1];

      while(z_temp < zf){       //while calculated ray ends too shallow,
         thi -= .0001*(PI/180);// increase the initial grazing angle
         arc_path(x_max, zi, thi);
         z_temp = arc[1];
      }
      thi += .0001*(PI/180);      //go back one step
      arc_path(x_max, zi, thi);
      z_temp = arc[1];

      while(z_temp < zf){      //while calculated ray ends too shallow,
         thi -= .00001*(PI/180); // increase the initial grazing angle
         arc_path(x_max, zi, thi);
         z_temp = arc[1];
      }
      thi += .00001*(PI/180);     //go back one step
      arc_path(x_max, zi, thi);
      z_temp = arc[1];
```

```
        while(z_temp < zf){       //while calculated ray ends too shallow,
                                  // increase the initial grazing angle
           thi -= .000001*(PI/180);
           arc_path(x_max, zi, thi);
           z_temp = arc[1];
        }
        thi += .000001*(PI/180);     //go back one step
        arc_path(x_max, zi, thi);
        z_temp = arc[1];

        while(z_temp < zf){       //while calculated ray ends too shallow,
                                  // increase the initial grazing angle
           thi -= .0000001*(PI/180);
           arc_path(x_max, zi, thi);
           z_temp = arc[1];
        }
        thi += .0000001*(PI/180);     //go back one step
        arc_path(x_max, zi, thi);
        z_temp = arc[1];


        while(z_temp < zf){       //while calculated ray ends too shallow,
                                  // increase the initial grazing angle
           thi -= .00000001*(PI/180);
           arc_path(x_max, zi, thi);
           z_temp = arc[1];
        }
        thi += .00000001*(PI/180);  //go back one step
        arc_path(x_max, zi, thi);
     }
}
```

SCR_ARC.CPP

```
/*=====================================================================
      See more complete description of program in "src.h".  This
program calculates the eigenrays for particular end point parameters.
The data points along the ray path are output as a file named
"ray_path" into the working directory.  This program can be used as a
stand-alone ray tracing program.  The subroutines included are:
arc_path, & arc_seg.
=====================================================================*/

#include "src.h"

void arc_path (double x_max, double zi, double thi)
{
   extern double profile [3][100];  //array of sound speed-depth
                                    // profile data
   extern int prof_max; //the number data point in the profile
   extern FILE *ofp;    //output file pointer; file is for data
                        // points along the arced path
   double g,            //slope dz/dc
          ci,           //initial sound speed (for an arc segment)
          cf,           //final sound speed               "
          xi,           //initial x                       "
          xf,           //final x                         "
          zf,           //final z                         "
          thf,          //theta final                     "
          thl,          //last (previous) theta
          tht,          //constant theta used when g = 0
          ti = 0.0,     //initial time
          tf,           //final time
          xtemp,        //temporary x
          thtemp,       //temporary theta
          ztemp,        //temporary z
          t_temp,       //temporary time
          c_temp;       //temporary sound speed
   int i,         //integer loop counter
       cnt = -1;  //place counter for within the sound speed - depth
                  // profile
   enum boolean {false, true};
   typedef enum boolean boolean;
   boolean ref = false;      //refraction

   ofp = fopen("ray_path", "w");
   xi = 0.0;                 //starting point is always = 0

   fprintf(ofp, "%f %f\n", xi, -zi);//starting point
   for (i = 0; i <= prof_max; ++i){ //find the proper starting
      if(profile[1][i] < zi)        // point in the profile,
         cnt = cnt + 1;             // depending on initial depth
   }
   if (cnt == -1)   //zi == 0 = surface
      cnt = 0;
```

```
//==========================
//initial propagation step
//==========================

g = profile[2][cnt];
if (thi > 0){ //propagation towards the surface
   zf = profile[1][cnt];
   cf = profile[0][cnt];
   ci = cf + g*(zi - zf);
   cnt = cnt - 1;
   if (g<-0.0005){          //*** +th -g 2nd quadrant arc
      thl = acos (ci/cf);
      if(thi <= thl){       //*** refraction
         thf = 0;
         zf = zi + ci/(g*cos(thi))*(1 - cos(thi));
         xf = xi + ci*(-tan(thi))/g;
         cf = ci - g*(zi -zf);
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
             - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
            ref = false;
         }
         else {             //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
            ref = true;
         }
      }
      else    //*** no refraction
         thf = acos(cf/ci*cos(thi));
   }
   else       // +th +g 4th quadrant arc, no refraction
      thf = acos(cf/ci*cos(thi));
}
if (thi < 0){  //propagation towards the bottom
   zf = profile[1][cnt + 1];
   cf = profile[0][cnt + 1];
   ci = cf + g*(zi - zf);
   cnt = cnt + 1;
   if (g>0.0005){           // -th +g 3rd quadrant arc
      thl = -acos (ci/cf);
      if(thi >= thl){       //refraction
         thf = 0;
         zf = zi + ci/(g*cos(thi))*(1 - cos(thi));
         xf = xi + ci*(-tan(thi))/g;
         cf = ci - g*(zi -zf);
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
             - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
            ref = false;
```

```
            }
            else {          //continue on
               fprintf(ofp, "%f %f\n", xf, -zf);
               ref = true;
            }
         }
         else    //no refraction
            thf = - acos(cf/ci*cos(thi));
      }
      else       // -th -g 1st quadrant arc, no refraction
         thf = - acos(cf/ci*cos(thi));
   }
   if (ref == true){ //second part of refreacted arc
      xtemp = xf - xi;
      xi = xf;
      ztemp = zi;
      zi = zf;
      thtemp = thi;
      t_temp = tf - ti;
      ti = tf;
      if (thi>0)  //use stuff for thi<0 (now towards the bottom)
         cnt = cnt + 2;
      if (thi<0)  //use stuff for thi>0 (now towards the surface)
         cnt = cnt - 2;
      xf = xi + xtemp;
      zf = ztemp;
      thf = - thtemp;
      tf = ti + t_temp;
      if (xf > x_max) { //over shot
         thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
         xf = x_max;
      }
      else {            //continue on
         fprintf(ofp, "%f %f\n", xf, -zf);
      }
   }
   else{ //ref == false
      if (((g > 0.0005)||(g <- 0.0005)) && (thi!= 0)){
                                                  //no refraction
         xf = xi + ci/(g*cos(thi))*(sin(thf)-sin(thi));
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
             - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
         }
         else {          //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
         }
      }
      if ((thi == 0)){  //propgation entirely in the horizontal
         zf = zi;
         xf = x_max;
         tf = xf/ci;
```

```
            fprintf(ofp, "%f %f\n", xf, -zf);
        }
        if ((g <= 0.0005)&&(g >= -0.0005)){//no change in sound speed
            if(thi<0)                        // through this interval
                tht = -thi;
            else
                tht = thi;
            xf = xi + (zf - zi)/tan(tht);
            tf = ti + 2*(xf - xi) / (ci + cf);
            if (xf > x_max) { //over shot
                thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
                xf = x_max;
            }
            else {            //continue on
                fprintf(ofp, "%f %f\n", xf, -zf);
            }
        }
    }
    thi = thf;
    xi = xf;
    zi = zf;
    ti = tf;
    ref = false;

    //=====================================
    // all subsequent steps of propagation
    //=====================================

    while((xi >= 0)&&(xf < x_max)){

        //=====================================
        // one arc segment to top or to bottom
        //=====================================

        while((zi > 0) && (zi < profile[1][prof_max]) && (xi >= 0)
                && (xf < x_max)){
            g = profile[2][cnt];
            if (thi > 0){ //propagation towards the surface
                zf = profile[1][cnt];
                cf = profile[0][cnt];
                ci = profile[0][cnt+1];
                cnt = cnt - 1;
                if (g<-0.0005){          // +th -g 2nd quadrant arc
                    thl = acos (ci/cf);
                    if(thi <= thl){      //refraction
                        thf = 0;
                        zf = zi + ci/(g*cos(thi))*(1 - cos(thi));
                        xf = xi + ci*(-tan(thi))/g;
                        cf = ci - g*(zi -zf);
                        tf = ti + 1/(2*g)
                             *(log((1+sin(thf))/(1-sin(thf)))
                             - log((1+sin(thi))/(1-sin(thi))));
                        if (xf > x_max) { //over shot
                            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
```

```
            xf = x_max;
            ref = false;
         }
         else {          //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
            ref = true;
         }
      }
      else    //no refraction
         thf = acos(cf/ci*cos(thi));
   }
   else       // +th +g 4th quadrant arc, no refraction
      thf = acos(cf/ci*cos(thi));
}
if (thi < 0){  //propagation towards the bottom
   zf = profile[1][cnt + 1];
   cf = profile[0][cnt + 1];
   ci = profile[0][cnt];
   cnt = cnt + 1;
   if (g>0.0005){           // -th +g 3rd quadrant arc
      thl = -acos (ci/cf);
      if(thi >= thl){      //refraction
         thf = 0;
         zf = zi + ci/(g*cos(thi))*(1 - cos(thi));
         xf = xi + ci*(-tan(thi))/g;
         cf = ci - g*(zi -zf);
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
             - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
            ref = false;
         }
         else {          //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
            ref = true;
         }
      }
      else    //no refraction
         thf = - acos(cf/ci*cos(thi));
   }
   else       // -th -g 1st quadrant arc, no refraction
      thf = - acos(cf/ci*cos(thi));
}
if (ref == true){ //second part of refreacted arc
   xtemp = xf - xi;
   xi = xf;
   ztemp = zi;
   zi = zf;
   thtemp = thi;
   t_temp = tf - ti;
   ti = tf;
   if (thi>0)  //use stuff for thi<0 (now towards the bottom)
      cnt = cnt + 2;
```

```
      if (thi<0)  //use stuff for thi>0 (now towards the surface)
         cnt = cnt - 2;
      xf = xi + xtemp;
      zf = ztemp;
      thf = - thtemp;
      tf = ti + t_temp;
      if (xf > x_max) { //over shot
         thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
         xf = x_max;
      }
      else {            //continue on
         fprintf(ofp, "%f %f\n", xf, -zf);
      }
   }
   else{ //ref == false
      if (((g > 0.0005)||(g <- 0.0005)) && (thi!= 0)){
                                                  //no refraction
         xf = xi + ci/(g*cos(thi))*(sin(thf)-sin(thi));
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
             - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
         }
         else {            //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
         }
      }
      if ((thi == 0)){  //propagation entirely in the horizontal
         zf = zi;
         xf = x_max;
         tf = ti + (xf-xi)/ci;
         fprintf(ofp, "%f %f\n", xf, -zf);
      }
      if ((g <= 0.0005)&&(g >= -0.0005)){//no change in sound
                                       // speed through this interval
         if(thi<0){
            tht = -thi;
            c_temp = profile[0][cnt+1];
         }
         else {
            tht = thi;
            c_temp = profile[0][cnt];
         }
         xf = xi + (zf - zi)/tan(tht);
         tf = ti + (xf - xi) / c_temp;
         if (xf > x_max) { // over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
         }
         else {            //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
         }
      }
```

```
   }
   thi = thf;
   xi = xf;
   zi = zf;
   ti = tf;
   ref = false;
}  //end while loop due to surface or bottom reflection or
   // x = xmax

if (cnt >= prof_max)
   cnt = prof_max-1;
if (cnt <= 0)
   cnt = 0;
   thi = -thi;  //change direction at interface

//do one step so zi not = 0 or z max to continue propagation loop
if (xi < x_max){
   g = profile[2][cnt];
   if (thi > 0){ //propagation towards the surface
      zf = profile[1][cnt];
      cf = profile[0][cnt];
      ci = profile[0][cnt+1];
      thf = acos(cf/ci*cos(thi));
      cnt = cnt - 1;
   }
   if (thi < 0){  //propagation towards the bottom
      zf = profile[1][cnt+1];
      cf = profile[0][cnt+1];
      ci = profile[0][cnt];
      thf = - acos(cf/ci*cos(thi));
      cnt = cnt +1;
   }
   if (thi == 0){  //propgation entirely in the horizontal
      zf = zi;
      xf = x_max;
      tf = ti + (xf - xi)/ci;
      fprintf(ofp, "%f %f\n", xf, -zf);
      g = 0;
      thi =.0001;
   }
   if ((g <= 0.0005)&&(g >= -0.0005)){ //no change in sound speed
                                       // through this interval
      if(thi<0)  {
         tht = -thi;
         c_temp = profile[0][cnt + 1];
      }
      else  {
         tht = thi;
         c_temp = profile[0][cnt];
      }
      xf = xi + (zf - zi) / tan(tht);
      tf = ti + (xf - xi) / c_temp;
      if (xf > x_max) { //over shot
         thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
```

```
            xf = x_max;
         }
         else {           //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
         }
      }
      else {    //g != 0
         xf = xi + ci/(g*cos(thi))*(sin(thf)-sin(thi));
         tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
              - log((1+sin(thi))/(1-sin(thi))));
         if (xf > x_max) { //over shot
            thf = arc_seg(thi, g, ti, x_max, xi, zi, ci);
            xf = x_max;
         }
         else {           //continue on
            fprintf(ofp, "%f %f\n", xf, -zf);
         }
      }
      zi = zf;
      xi = xf;
      thi = thf;
    }
  } //end big while loop since x >= xmax
  fclose(ofp);
}

double arc_seg(double thi, double g, double ti, double x_max,
               double xi, double zi, double ci)
{             //returns theta final after calculating tf, xf, and zf.
              // This function calculates data for the last segment of
              // the arced path
  extern double arc[4]; //global array for storing end point data for
                        // eigen ray
  extern FILE *ofp;     //output file for data points along arced path

  double tf,  //final time
         xf,  //final x
         zf,  //final depth
         thf; //final theta

  thf = asin( sin(thi) + g * cos(thi) * (x_max - xi) / ci);
  tf = ti + 1/(2*g) * (log((1+sin(thf))/(1-sin(thf)))
       - log((1+sin(thi))/(1-sin(thi))));
  xf = xi + ci/(g*cos(thi))*(sin(thf)-sin(thi));
  zf = zi + ci/(g*cos(thi))*(cos(thf)-cos(thi));
  arc[0] = xf;
  arc[1] = zf;
  arc[2] = tf;
  arc[3] = thf;
  fprintf(ofp, "%f %f\n", xf, -zf);
  return thf;
}
```

APPENDIX C

SAMPLE INPUT FILE FOR BUOY POSITIONS AND TTD VALUES

Format:

	Latitude		Longitude		
	Degrees	Minutes	Degrees	Minutes	$TTD_{with\ Buoy1}$(s)
Buoy 1	-	-	-	-	-
Buoy 2	-	-	-	-	-
Buoy 3	-	-	-	-	-
Buoy 4	-	-	-	-	-

Each block of data for all 4 buoys is a data set for one event. The Buoys are numbered in order of signal reception.

This file contains four data sets. A Maximum of 99 data sets can be read in as one file. File starts here:

```
28.000000 26.000000 88.000000 31.000000 0.000000
28.000000 26.107991 88.000000 31.161987 0.059143
28.000000 26.269978 88.000000 31.134989 0.133572
28.000000 26.323974 88.000000 31.323974 0.285751

28.000000 26.000000 88.000000 31.000000 0.000000
28.000000 26.107991 88.000000 31.161987 0.201277
28.000000 26.269978 88.000000 31.134989 0.254535
28.000000 26.323974 88.000000 31.323974 0.468107

28.000000 26.000000 88.000000 31.000000 0.000000
28.000000 26.107991 88.000000 31.161987 0.175250
28.000000 26.269978 88.000000 31.134989 0.294576
28.000000 26.323974 88.000000 31.323974 0.454781

28.000000 26.000000 88.000000 31.000000 0.000000
28.000000 26.107991 88.000000 31.161987 0.212361
28.000000 26.269978 88.000000 31.134989 0.336213
28.000000 26.323974 88.000000 31.323974 0.521846
```

APPENDIX D

SAMPLE INPUT FILE FOR SOUND SPEED VS. DEPTH PROFILE

Format:

	Sound Speed(m/s)	Depth(m)
Point 1	-	-
Point 2	-	-
.	-	-
.	-	-
.	-	-

This file contains 20 points. 99 data points is the maximum allowable. Data points must be input in incremental order from surface to bottom. The first data point must be the sound speed at the surface, and the last needs to be the sound speed at the bottom. Depth is positive.
File starts here (Data points in figure 2):

```
1540.0 0.0
1535.0 34.5
1527.9 69.0
1516.1 103.5
1503.3 113.9
1499.3 172.5
1495.7 300.15
1492.75 379.5
1485.3 517.5
1481.9 621.0
1480.0 803.85
1480.4 914.3
1481.5 1017.8
1484.6 1214.4
1488.0 1386.4
1492.6 1610.7
1505.6 2562.4
1516.75 3338.1
1536.6 4476.1
1544.4 4928.1
```

APPENDIX E

SAMPLE OUTPUT FILE

Coordinates are with respect to buoy 1.
File starts here:

Coordinates (x, y, depth) meters, where latitude direction is x,
and longitude is y.

Data Point 1 ...
Straight line s = (50.93, 9.86, 546.87)
Calculated s = (50.87, 9.83, 557.25)
ttdf1 0.000000, ttdf2 0.000000, ttdf3 0.000000
With respect to
28.0 deg 26.000000 min Latitude
88.0 deg 31.000000 min Longitude

Data Point 2 ...
Straight line s = (-488.64, -1388.89, 1008.57)
Calculated s = (-483.58, -1379.02, 1010.60)
ttdf1 0.000000, ttdf2 0.000000, ttdf3 0.000000
With respect to
28.0 deg 26.000000 min Latitude
88.0 deg 31.000000 min Longitude

Data Point 3 ...
Straight line s = (-1311.90, -876.63, 1441.44)
Calculated s = (-1309.58, -875.05, 1449.17)
ttdf1 0.000000, ttdf2 0.000000, ttdf3 0.000000
With respect to
28.0 deg 26.000000 min Latitude
88.0 deg 31.000000 min Longitude

Data Point 4 ...
Straight line s = (-4783.53, -3929.27, 2764.45)
Calculated s = (-4699.77, -3860.63, 2724.42)
ttdf1 0.000000, ttdf2 0.000000, ttdf3 0.000000
With respect to
28.0 deg 26.000000 min Latitude
88.0 deg 31.000000 min Longitude

VITA

Thomas Scott Brandes was born in Cincinnati, OH on August 6, 1971. After growing up in Atlanta, GA and getting a high school diploma from the Marist School in Dunwoody, GA, he graduated with a B.S. in Physics from the Georgia Institute of Technology in 1993. He began graduate school in the physics department at Texas A&M University in 1994, and switched to the interdisciplinary engineering department to work in the field of bioacoustics in 1995. His permanent address is: 3894 Vicar Ct., Atlanta, GA 30360.