Some notes for 3-D ray tracing program TRABOX

Malcolm Sambridge,

Research School of Earth Sciences,

Australian National University

Latest revision February 1999

(Original version 1987)

TRABOX is a driver program for tracing rays through a 3-D mesh of velocities in Cartesian co-ordinates. All two-point ray tracing is performed by calling the routine trac3d. One two-point ray is traced for each call to trac3d. The program was not originally designed for general distribution and is rather ugly in parts (especially the driver program trabox). The aim was to be versatile in the number and type of ray-tracing problems that could be dealt with from a simple input. The program has been added to and modified many times and some features of the code are obsolete (even some input parameters are ignored!). The following is a reasonably complete description all its features. The distribution also contains some example sets of input files to demonstrate the various modes of the program. The routine trac3d which performs all the actual ray tracing could, if necessary, be incorporated into another more problem specific program (if you are willing to try !).

Reference

TRABOX is based on the algorithm described in Sambridge & Kennett (1990).

Boundary value ray-tracing in a heterogeneous medium: a simple and versatile algorithm , Sambridge, M.S. and Kennett, B.L.N *Geophys. J. Int.*, **101**, 157-168, 1990. doi:10.1111/j.1365-246X.1990.tb00765.x

Conditions of use

published work contains a short acknowledgement and a reference to the above paper. I can be contacted at malcolm@rses.anu.edu.au

Compilation

To compile the source codes type,

```
> make traclib tracer trabox
```

or,

```
> make all
```

The program expects to find the files `Tra_mod3D.in' and `Tra_rays.in' in the directory from which the program is run. If the input model is being read in in binary format, as in Example 4, it also expects to find the file `Tra_mod3D.bin'.

A second program called `readray.f' is provided as an example of how to read in a binary ray file and interpret it.

I/O structure

Input files:

`Tra_mod3D.in'	contains details about the input 3-D mesh of velocity points. If the ascii read option is used (ifile <= 0) then the velocity model is also read in here.
`Tra_mod3D.bin'	contains the input 3-D mesh of velocity points in binary if the unformatted read option is used (ifile > 0).
`Tra_rays.in'	contains details of the rays to be traced.

Output files:

`Tra_iterations.out'	contains a summary info of each iteration in two-point ray solution.
`Tra_raypath.out'	contains position and velocity model along each raypath (unformatted if option is set).
`Tra_details.out'	contains full details of each raypath at every point of numerical integration. (Optional, used for debugging)
`Tra_iterations.out'	a summary of each ray traced
`Tra_summary.out'	contains 1 line summary of two-point solution rays.

Velocity model representation

The velocity model represented by a 3-D mesh of knots and uses a Cardinal spline interpolation (from Martin Smith).

Two velocity models can be input.

The ***model 1*** consists of two independent 3-D meshes of velocities separated by an irregular interface, which can contain a velocity discontinuity. The shape of the interface is specified by a series of connected triangular plates. The vertices of the triangles are the depths of the interface (see Fig. 2 of Sambridge 1990, *Geophys. J. Int.*, **102**, 653-677). These vertices may be read in if desired, or fixed to a constant depth (horizontal) interface. (This model has not been used for some time and remains as a historical feature of the program.)

The ***model 2*** is a single 3-D mesh of velocities with no interface.

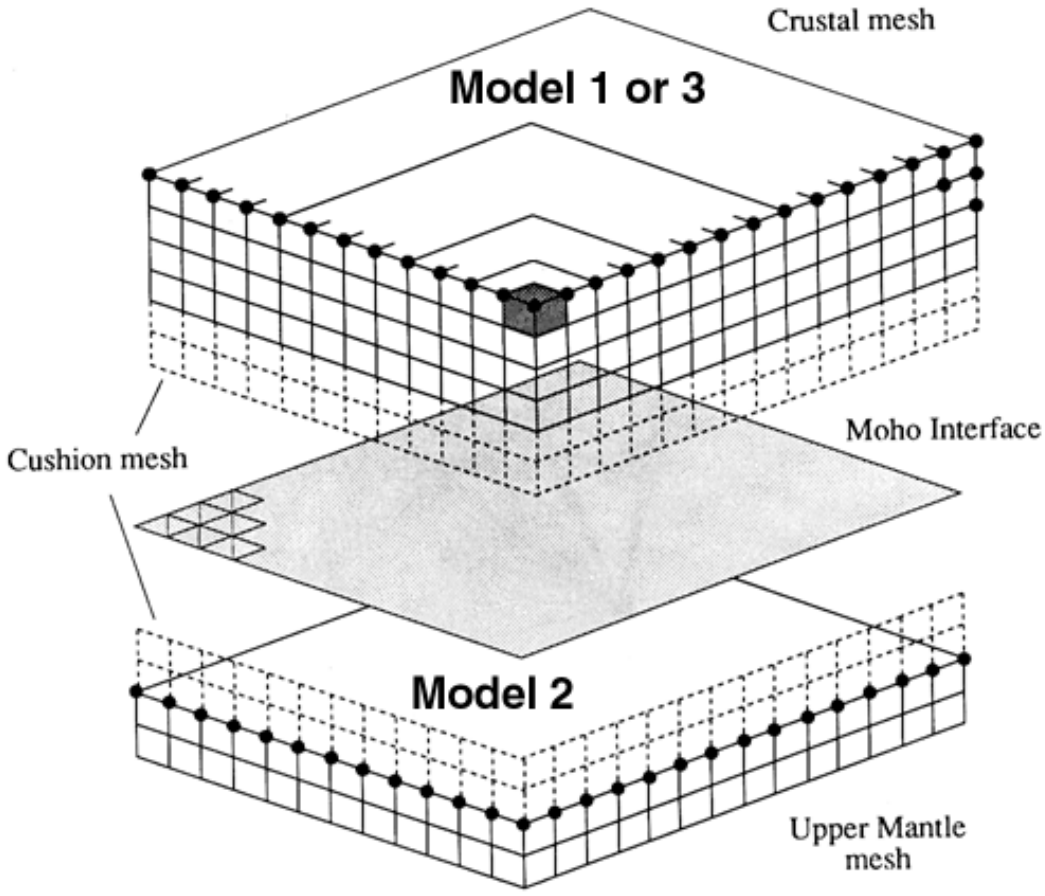

Figure 1. Illustration of nodal mesh used in TRABOX. The interface is only used between mesh 1 and 2. As an alternative mesh 3 can be used without an interface.

Input format of velocity models

The velocity model may be input from an ascii file Tra_mod3D.in (inefficient), or from an unformatted file Tra_mod3D.bin (quicker) in the order,

```
(((v(i,j,k),i=1,nx),j=1,ny),k=1,nz)
```

All three 3-D meshes may be input (two for model 1, one for model 2) or only those which are actually being used. These options are controlled by the parameter IFILE read in on line 5 of the input file Tra_mod3D.in.

IFILE Possible values (-2,-1,0,1,2)

Format code for reading in 3-D mesh. Several formats for reading in 3-D mesh are allowed.

ifile <= 0	means read in mesh from ascii file Tra_mod3D.in
ifile > 0	means read in mesh from binary file Tra_mod3D.bin
ifile = -2 or 3	means only Model 1 present in file.
ifile = -1 or 2	means only Model 2 present in file.
ifile = 0 or 1	means both Model 1 & 2 present.

The number of nodes in the X and Y directions (nx,ny) are read in from line ? and the three 3-D meshes may have different numbers of nodes in the Z-direction (nz1,nz2,nz3).

Array sizes

The internal sizes of the arrays should be large enough for the 3-D velocity models. If they are not then the program writes an error message advising that the current defaults in the parameter statement

```
parameter (nxmax=602,nymax=602,nzmax=5,nzPgmax=1,nzPnmax=1,nzSmax=5)
```

must be adjusted be editing the include file "meshsizes.param" and the program must then be recompiled by typing

```
make tracer traboX
```

(Note: parameter nzmax should be the maximum of nzPgmax,nzPnmax & nzSmax, which are the maximum values of nz1,nz2 & nz3 respectively.) These parameters determine the main memory demands of the program and so it is a good idea to keep them equal to or just bigger than that required by your velocity model.

2-D read in option

Internally the program always uses a 3-D mesh of velocities and the minimum number of nodes in any direction is 4 (to allow cubic interpolation to work properly). For ease of use a 2-D mesh may be read in by specifying a node size of 1. If the value of IFILE and combination of nx,ny,nz1,nz2,nz3 is such that a 2-D mesh is being read in, then it is converted to a 3-D mesh internally by duplicating the layers corresponding to the parameter with node size=1, e.g with ifile=1,nx=101,ny=101,nz1=*,nz2=*,nz3=1 then a single XY layer is read in and converted to a 3-D mesh of 4 nodes in the Z direction.

Units

The units used for length velocity etc are defined by the sizes given for the node spacing dx,dy,dz. Other quantities, e.g. origin position and velocities are assumed to be in a consistent unit (usually km & s).

Conventions

All angles in input file are in degrees. Azimuth is the angle clockwise from the x-axis and declination is measured from the upward vertical, i.e. horizontal is declination 90, and straight down is declination 180. A right-handed axes system

is used. For most geographical problems this means that x-, y- and z-axis can be in the East, South and downwards respectively or maybe North, East, downwards. In actual fact it does matter what the directions of the axes are, as long as they make up a right handed system. This must be remembered when designing the input velocity model. Declination is set to 90 degrees auotmatically if 2-D surface ray tracing option is used.

Linear gradient models surrounding the 3-D mesh

To avoid problems associated with rays passing outside of the model, a set of linear gradient models are used, i.e. outside of the model the velocity field takes the form `v = vo + vx*x + vy*y + vz*z`. For each of the two velocity models two linear gradient models may be specified (lines 31-41 of Tra_mod3D.in). The first is used when a ray passes out above the 3-D mesh or to the sides of the 3-D mesh and the second is used when the ray passes out below of the 3-D mesh. To allow accurate ray tracing the numerical equation solver requires that the velocity model (and both its first and second derivatives) are continuous. Within the 3-D mesh this condition is satisfied by the interpolation procedure but in general it will not be met across the boundaries of the box. Therefore a ray will incur a numerical error as it traces across the box edges. This is usually o.k. if it occurs during iterations of two-point ray tracing so long as the final (solution) ray ends up entirely within the box.

Ideally the entire 3-D mesh should be surrounded by a `cushion' in which the interior 3-D model is smoothly `knitted' into the surrounding linear gradient model. (I always intended to make the program do this automatically but could never be bothered.) In practice it is hardly necessary, as one can always make the mesh bigger in nx and ny than needed. The Z-direction is more serious as rays can easily go out of the bottom of the box during iterations for a two-point ray. The `cushion' facility does exist in this direction. The program automatically puts 3 extra layers of mesh points below the model where the 3-D model is knitted smoothly into the linear gradient model below. The knitting only occurs for the z component of the linear gradient model, i.e. the 3-D mesh is knitted into the model v = vo +vz*z (again I didn't get around to knitting into the full linear gradient component model).

In short then, the surrounding linear gradient models are no big deal. Their main function is to refract rays back (usually upwards) towards the model. It is advisable to choose the models (usually with vx=vy=0) so that large velocity jumps at the edges of the box are avoided. If the 3-D mesh is a perturbation about a 1-D model then its a good idea to make the two linear gradient models equal to the background 1-D models at the top and bottom of the 3-D region.

If any rays pass outside of the model during iterations for a two-point solution then these are flagged by a `*' in the output summary file `Tra_summary.out'.

Irregular or flat interface/discontinuity

If you wish to use just the 3-D mesh without an interface (model 2) then you can ignore this section.

The variable interface is an `historic' feature of the program and has not been used or tested for a number of years. (I have had no cause to use it other than in the original project for which the code was written.) No guarantees are given

that it still works, but I hope it does. If it doesn't it should not be too difficult to make it work again. Remember the interface is only present in Model 1. If Model 1 is not read in (i.e. ifile = -1 or 2) or if tracing is only in model 2, then it is ignored.

The interface can be input in two ways. If the parameter `meshformat' (first parameter line 5 of `Tra_mod3D.in') is equal to 1, then a horizontal interface is assumed at the nth layer from the top of the upper mesh in Model 1, where n is read in on line 27. It is advisable (but not necessary) that there are at least two more layers of the upper mesh below the interface level, i.e. that n <= nz1-2. This produces an `overlap' of the upper mesh below the interface, which is used by the cubic velocity interpolation procedure above the interface. If this condition is not met then a slight divergence from cubic interpolation may occur, which is not serious. For the same reason the position of the lower mesh is fixed by the position of the interface such that the third layer of the lower mesh is at the interface depth. Again there are two overlapping layers above it, which are only used by the interpolation procedure. (The interpolation uses the 64 nearest velocity nodes, in 3-D, to calculate the velocity at any point in the model.)

As an example, if the upper mesh is nz1=15,dz=10,zo=0 the lower mesh is nz2=10, and the interface meshknot=13, then the interface is at a depth of 130km and immediately below this is the third layer of the lower model, (the tenth layer is at a depth of 190km).

Any type of discontinuity is allowed between the two meshes on either side of the interface. Note the `hidden' velocity layers should contain velocities appropriate to the velocity distribution in the respective mesh (i.e. some smooth continuation as they will be used to interpolate for velocities within one layer (+-dz) of the interface.

If meshformat=2 then a set of depth nodes is read in using the following code

```
Input       line 26      read(lu,*)
input       line 27     read(lu,*)mohox,mohoy
input       line 28     read(lu,*)
input       line 29     read(lu,*)
input       line 30     read(lu,*)
                        do 311 i = 1,mohoy+1
                              read(lu,*)(moho(i,j),j=1,mohox+1)
                        311 continue
```

which comes from subroutine vmod. In this case it is even more important that depths read in are centred around a value which has an `overlapping' segment in the upper model as before. The lower model is assumed to be positioned such that the third layer is at the depth of point moho(1,1). (If this is inconvenient change the value of of parameter `depthm' in subroutine vmod to an appropriate value.)

Tracing Rays

The driver program trabox is designed for a range of initial value and two-point ray tracing problems. The subroutine that does all the work (trac3d) always performs two point ray tracing until the desired target has been hit or some other termination criteria have been reached. The algorithm is of a shooting type which is not as efficient as most bending methods but far more versatile.

Two-point ray tracing

The code was originally developed for two-point ray tracing, i.e. find the ray between a given pair of receivers in a 3-D model. For complex velocity models there is no guarantee that a solution exists to this problem (the receiver may be in the shadow of the source) and there is also no guarantee that any solution found will be any particular arrival (i.e. the first), however, some precautions can be taken to encourage a convergence to the first arrival (starting with a suitable starting ray). Source and receiver positions are input in file Tra_rays.in together with all other parameters which control the number and type of rays to be traced.

Converge criteria for two-point ray tracing

For two point ray tracing the program stops tracing a ray when it hits the HORIZONTAL PLANE THROUGH THE RECEIVER. It then compares the endpoint of the ray with the target to decide whether convergence has been achieved. A tolerance is therefore required in Z (to hit the target depth) and in (X,Y) to hit the target position on the target depth. Values of xtol, ytol and ztol are read in on line 8. Also an internal interface tolerance is required (line 8) to decide on when the ray has hit the interface (applies only to tracing in model 1). The complete ray converge criteria consists of hitting the horizontal plane though the receiver and hitting A BOX OF SIZE XTOL AND YTOL CENTRED ON THE TARGET.

The program will stop tracing a ray on the first time it hits the horizontal plane through the receiver, so if the receiver is below the source it will meet the plane on the downward path of the ray rather than the upward path. The source must therefore always be below the receiver.

Ray tracing in a 2-D plane

There are two ways of ray tracing through a 2-D model.

For velocity models that are 2-D, i.e. have no variation in the Y or X direction the program will trace rays in exactly the same way as in a 3-D model, with the same convergence criteria above. Although the ray will not move out of the source/receiver plane. As far as the program is concerned it is still dealing with a 3-D model. (The initial azimuth can even be set out of plane and the program should converge to a correct ray in the source/receiver plane. Although in practice the only point in doing this would be to check the accuracy of the ray tracer.)

For convenience a special option has been introduced for tracing a ray on a 2-D (X-Y) surface. The only really fundamental difference between this and the above mode is that the convergence criteria is different. To use this `surface' ray tracing option a 2-D model in X-Y is assumed, i.e. no variation of velocity with depth (see

reading in of a 2-D velocity model above). The mode is activated by a parameter read in from line 6. If its value is 1 then the 2-D mode is assumed and the convergence criteria of hitting the depth plane through the receiver is replaced with hitting A GIVEN RADIUS FROM THE SOURCE ON THE XY PLANE. This radius is read in from line 38. If the value is 0 then the source-receiver separation is calculated and used. The second criteria (for two-point ray tracing) of hitting a box around the target is unchanged. The 2-D option can be used with either the two-point ray tracing mode or pure shooting at regular interval mode (see below).

This option is useful when the normal convergence criteria for 3-D models is unsuitable for a particular 2-D model. E.g. for mapping paths of surface waves, or for tracing from borehole to borehole when the sources may be above the receivers. (In this latter case the X-Z components of a 2-D borehole velocity model could be fed into the X-Y parameters of the program.)

Shooting at regular angles in declination and azimuth

As a convenience an option has been included to shoot rays at specified (regular) intervals in azimuth and declination. (The alternative is to enter every source receiver pair explicitly using the multiple read format described below.) Although in this mode rays are only shot out at known azimuths and declinations, internally two-point ray tracing is being performed and X and Y tolerance is set to 10**6 km so that they always converge in 1 iteration. (See line 12 of input file Tra_rays.in.)

The parameters read in on line 12 are ndec,deldec,nazi,delazi and the sequence of rays shot out from the source takes the form

```
if(ndec > 0, nazi > 0)
      do 1 i = 1,ndec
            dec(i) = dec_initial + (i-1)*del_dec
            do 1 i = 1,ndec
                  az(j) = azi_initial + (j-1)*del_azi
```

If ndec=nazi=0, then only the ray with (dec_initial,azi_initial) is traced, and these values are read in on line 46 (see below).

***Parameters controlling numerical integratio*n**

Three parameters control the numerical integration that is used in time stepping the rays.

line 14:steplength,epsilon,ifixed (internally they may have different names)

The program has the option of either a fixed steplength numerical integration (4th order Runge Kutta, which is slow but reliable and sometimes necessary), or an adaptive steplength (5th order Runge Kutta, which is usually much more efficient but requires using the parameter epsilon to control required accuracy). The fixed steplength is achieved with ifixed=1. In this case the steplength is given by `steplength' (in seconds) and epsilon is ignored. For ifixed = 0 the adaptive steplength procedure is used. The parameter steplength is the starting steplength, i.e. that used for the very first step from the source, and the very first step after hitting an internal surface of discontinuity. The adaptive steplength

routine may decide to increase or decrease this value depending on the required tolerance and the nature of the velocity model.

The parameter epsilon controls the accuracy imposed on the adaptive steplength numerical integration. It can be difficult to use because it actually controls the fractional accuracy in the increments to the raypath variables at each step. This is necessary because in ray tracing we are usually worried about the cumulative error of parameters like X, and Y over the whole ray and not the absolute error at each step. The tolerance criteria is the same as eq. 15.2.6 of Numerical Recipes (Press et.al. 1987, 1st ed., p557). Refer to section 15.2 of Numerical Recipes for more details. The accuracy control is currently applied only to the (X,Y) position of the ray path and the length (parameters 1,2 and 16 respectively). The stepsize is automatically adjusted to meet the error criteria of the worst offender. [It is possible to apply the error criteria to other parameters (e.g. dec, azimuth or geometrical spreading variables) by changing the array `wscal' in subroutine trac3d at lines around 232 and 272.]

The accuracy criteria cannot be applied to travel time because that is the variable, which has been used as the independent parameter. So in effect there is (by definition) no error in the travel time but the ray position is calculated (with error) for some given travel time. It is possible to change the independent parameter to say arc length (or some other parameter), by changing the equations in subroutine Diff. I have never found it necessary to do this, but if you try you should note that some of the program's logic has been designed using the travel time as the independent parameter and this may have to be adjusted.

Note: the efficiency of the adaptive steplength integration (and hence the whole program) will be sensitive to the value of epsilon. Small epsilon (~0.001) means high accuracy, which means short steps, which means more steps, which takes more time!

Several other parameters are needed to avoid the possibility of infinite loops in the ray tracing procedure. They are the maximum number of steps used (line 18) and the maximum number of iterations for two-point ray tracing (line 20). The first is hardly ever needed but must be set to stop a ray continuing forever. (One circumstance where this might occur is when the ray leaves the 3-D model and passes into a homogeneous surrounding model and the declination at the point where the ray leaves the model is > 90 degrees. So it will never come back to the target surface above it.) The second may often be needed if the two-point ray tracer fails to hit the target within the specified tolerance. This can frequently occur. Some of the reasons are: 1) There is no two-point ray between source and receiver (target is in a shadow of the source), 2) the starting angle is too far away, or lies in a range prone to divergence, 3) The accuracy of the numerical integration is too low. Usually 1 or 2 is responsible.

Printing out details for debugging, and complete raypaths

Lines 24, and 27 contain parameters, which determine the diagnostic level of output. Line 24 is a print switch (usually set to 1). In this mode the program prints only summary information to file Tra_details.out after each two-point ray has converged. If its value is 0 then full details of every nth step of the each ray is printed out to the same file. This mode is usually used for debugging or finding

out what actually happened to a ray that failed for some reason. The value of n is read in on line 27. (n=1 can give and enormous output file).

line 31 contains a parameter which is now obsolete (sorry I should have removed it but haven't got around to it). It was originally used to calculate Frechet derivatives of travel time with respect to each velocity node for each raypath. It was used by a 3-D seismic inversion program. The section of code that was activated by it is intact but corresponds to a particular size of velocity mesh. I do not recommend trying to re-activate it. The parameter is ignored by the program.

line 35 contains a switch for writing out the complete raypaths to an unformatted file `Tra_raypath.out'. Possible values are n (n>=1),0,-1,-2.

If n >= 1 then the raypath of iteration n is output.
If n = 0 then the no raypath of is output.
If n = -1 then only the final (converged) raypath is output.
If n = -2 then all iterations of all raypaths are output.

When output is activated the parameters x(i), y(i), z(i), azi(i) dec(i), velocity model and derivatives, and travel time are output. The frequency of output is determined by the value of n read in on line 27 (which also controls the print out frequency above), e.g. if n = 10 every tenth point will be written out. [Note if adaptive step size is being used (see above) and n is large (say 100) you may only get the source and receiver point output (because the ray may have less than 100 points in total). If fixed steplength is being used then equal time steps will be printed out at a frequency of n x steplength.] A simple program `readray.f' is provided to read the output ray file Tra_raypath.out and write information to the screen about all rays in the file. This program can be used as a template for reading raypath information into other programs.

Multiple sources and receivers

The (x,y,z) of the source is read in on line 31, while the receiver (target) is on line 44. On the source line the `ray code' is also read in. If both model 1 and model 2 have been readin (see above) then the ray code determines in which of these the ray will be traced. If ray code = 1 then model 1 is used (with interface), if ray code = 2 or 11 (model two is used).

Line 46 contains the initial azimuth and declination of the ray used for two point ray tracing. If the shooting options are used (from line 12) then these are the initial azimuth and declination for the spray of rays. If the special values (0.,0.) are read in then the azimuth and ray is automatically set for the direct ray in a homogeneous medium. (It may be worthwhile changing this so that the declination is appropriate for a ray in a linear gradient medium, possibly one rThe first three examples us the same velocity model and three different ray input files `Tra_rays.in'. The fourth example uses a 3-D velocity model in binary format, and a different ray input file.ead in for the surrounding mesh. The value of `dec(ray)' would have to be changed in main routine TRABOX at about line 404 in an obvious fashion.)

The program allows input of multiple sources and receivers. The parameter read in on line 49 (irepeat_condition) determines what the program expects to see in the lines 56 onwards. Possible values are (0,1,2,-1)

= 0	means read in only new ray end points (lines 38 - 55) and trace the ray.
= 1 & 2	means read complete set of parameters (lines 1 - 55) but use the final take- off angles from the previous ray as the starting guess for the next ray. One use of this is in re-tracing a ray more accurately, i.e. by decreasing tolerances. (If irepeat_condition = 1 and the ray failed to converge then program is aborted, if = 2 it continues on to next ray.)
= -1	means read complete set of parameters (lines 1 - 55). This is used when the next ray is a new one and one may wish to reset some parameters to different values.

Examples

The following examples with input files are included in the distribution. Input files for each and a README file is contained in the subdirectory of the same name. The first three examples use the same velocity model and three different ray input files `Tra_rays.in'. The fourth example uses a 3-D velocity model in binary format, and a different ray input file.

Example1: Twopoint ray tracing of a single ray

Example2: Twopoint ray tracing of multiple rays

Example3: Twopoint ray tracing of multiple rays using repeat option

Example4: Shooting through a 2-D model (using 2-D options)

Examples 1,2 and 3 require arrays defined in meshsizes.param1, while Examples 4 and 5 require array sizes defined in meshsizes.param2

To build executable for examples 1-3

```
cp meshsizes.param1 meshsizes.param
make tracer trabox
```

and for examples 4-5

```
cp meshsizes.param2 meshsizes.param
make tracer trabox
```

Example1

This example uses the following features of the program trabox:

1. 3-D ascii input from file Tra_mod3D.in: trabox reads in a 3-D (XYZ) model in ascii. nx=16,ny=128,nz=32 (see Tra_mod3D.in line 11). Only model 2 is read in. (By changing the value of `ifile' model 1 could also be read in, see Trabox_manual.)

2. (Note: this is an inefficient way of reading in the 3-D velocity model. See example 4 called 'Example4_2D' for an example of using the binary read option.)
3. Two-point ray tracing is performed (see Tra_rays.in line 6) in model 2 for a single source/receiver pair. The parameter (epsilon, line 15 of `Tra_rays.in') that controls the accuracy of the adaptive numerical integration is 1/50 th of the value used in Example3_2D. (The allowed tolerance in x,y and arc length is therefore 1/50th of the value in Example4_2D.)
4. Only a summary of the iterations is written to file `Tra_details.out'. (No raypath file is written out see line 36 of 'Tra_rays.in')

Example2

This example is similar to the previous Example1_twopoint.

1. The same model is read in as in the previous example `Example1_twopoint'. (You must move the file Tra_mod3d.in to this directory to run it)
2. Two-point ray tracing is performed for two source receiver pairs. The repeat condition (line 50 of Tra_rays.in') is set to zero which means that at line 57 either there can be an end of file (as in previous example, or another source receiver pair can be read in as is the case in this example.
3. The raypath switch is set to -1 for both source receiver pairs. This means that the final (converged) raypath is written out to file `Tra_raypath.out'. (Use the utility program readray.f to check the contents of this file.

Example3

This example is similar to the previous Example2_twopoint_multiple.

1. The same model is read in as in the example `Example1_twopoint'. (You must move the file Tra_mod3d.in to this directory to run it)
2. Two-point ray tracing is performed for two source receiver pairs. For the second ray the repeat condition (line 66 of Tra_rays.in') is set to -1 which means that at line 73 a complete new set of parameters is read in (i.e. similar to lines 1-56) and a new ray is traced with these parameters using the final take-off angles from the converged second ray. The tolerances for the first two rays are rather loose (=0.1, see line 14 of Tra_rays.in) but the repeated ray is much higher (=0.005, line 86). This is an example of using loose tolerances to converge on a two-point ray and then repeating with increased accuracy. Usually, as in this case, the more accurate ray converges in 1 iteration and so an accurate two-point solution can be obtained without calculating rays at high accuracy for all iterations of the two-point ray tracer.
3. The raypath switch is set to -1 for the repeat run through the second ray. This means that the final (converged) raypath is written out to file `Tra_raypath.out'. (Use the utility program readray.f to check the contents of this file.)

Example4

This example uses the following features of the program trabox

1. 2-D input option: trabox reads in a binary 2-D (X-Y) model of size nx=601,ny=601,nz=1 (see Tra_mod3D.in line 11)
2. The 2-D surface ray tracing option is used. (see Tra_rays.in line 6)
3. Shooting rays only: 50 rays are traced from a source at varying azimuths separated by 1 degree target radius is 6900 km (see Tra_rays.in line 12).
4. All rays are written out to file `Tra_raypaths.out' (see lines 27 and 35 of Tra_rays.in). Use program readray.f to check on rays written to file Tra_raypath.out.

(Note option 2 could be used even if a 3-D model is read in.)

Array sizes

Note this example uses a large velocity model read in from a binary file. The arrays in the program are not big enough to read in the model and the parameter statement must be edited to allow the program to run

The correct parameter statement is commented out in the include file "meshsizes.param". The comment lines must be swapped over in this file, i.e.

```
c       parameter  (nxmax=602,nymax=602,nzmax=5,nzPgmax=1,nzPnmax=1,nzSmax=5)
        parameter (nxmax=20,nymax=130,nzmax=33,nzPgmax=1,nzPnmax=1,nzSmax=33)
```

must be replaced with

```
        parameter  (nxmax=602,nymax=602,nzmax=5,nzPgmax=1,nzPnmax=1,nzSmax=5)
c       parameter (nxmax=20,nymax=130,nzmax=33,nzPgmax=1,nzPnmax=1,nzSmax=33)
```

and then the program must be recompiled with `make tracer trabox'.

This is an example of the array size editing refered to in the manual. If machine memory permits set these values bigger than any model you are likely to need and you won't have to edit them again.

Perform the other examples first otherwise you will have to reverse these changes to run the other examples.

Tips on improving speed with trabox.

There are several options in the input file Tra_rays.in and depending on these the program efficiency can vary enormously. Important factors are the following:

Tip 1:

Have you used fixed or variable step-length ray tracing ? (see line 14 of Tra_rays.in)

fixed = can be very slow and wasteful if step size is small,

variable = can be much more efficient, accuracy is difficult to control which can affect final ray.

Accuracy of variable step-length solver is controlled mainly by second parameter on line 14 of Tra_rays.in.

Tip 2:

A simple trick that can speed up even the variable version is to use the option on line 49 of Tra_rays.in, set repeat_condition = 1 (or 2) which will read in a complete set of parameters (line 1-49). The trick is to trace an approximate two-point solution for the first ray, i.e. with loose tolerances on variable or fixed step-size, and use those take off angles for the second ray traced more accurately. The overall solution is then achieved by a two stage process. Also a good idea is to use loose tolerances on endpoint accuracy for first rays to prevent extra iterations.

***Tip 3*:**

Raypath need only be written out (if at all) for the converged ray i.e. write switch -1 on line 35.

***Tip 4*:**

Turn all debugging print switches off.

The question of failure to converge on a two-point solution is much more tricky and can be due to a number of reasons. Basically as the heterogeneity goes up (and 50% is getting large) you usually need a disproportionate amount of computation, intervention and tuning to get those final rays. More accuracy and different starting points are a help.

This manual last revised February 1999