The TauP Toolkit:
Flexible Seismic Travel-Time and Raypath Utilities

Version 1.1
Documentation

H. Philip Crotwell, Thomas J. Owens, Jeroen Ritsema
Department of Geological Sciences
University of South Carolina
http://www.seis.sc.edu
crotwell@seis.sc.edu

Copyright (c) 1998-2000 Board of Trustees of the University of South Carolina,
All Rights Reserved

Contents

Disclaimer and License

```
The TauP Toolkit: Flexible Seismic Travel-Time and Raypath Utilities.
Copyright (C) 1998-2000 University of South Carolina

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.

The current version of The TauP Toolkit can be found at
http://www.seis.sc.edu

Bug reports and comments should be directed to
H. Philip Crotwell, crotwell@seis.sc.edu or
Tom Owens, owens@seis.sc.edu
```

The TauP Toolkit is available free under the terms of the GNU General Public License, found in the COPYING file within the distribution. It gives you specific rights to use, modify and redistribute this software. Please be aware of section 2 of the license which specifically prevents you from redistributing this software incorporated, in whole or on part, into to a work, unless that work is also covered by the GNU General Public License. Please see the Free Software Foundation's web site, www.fsf.org, for more information.

1 Overview

The algorithms employed within the TauP package are based on the method of Buland and Chapman (1983). The IASPEI *ttimes* package is a widely-used implementation of the methodology.

The main benefit of this new package is a marked increase in flexibility. It will handle many types of velocity models, instead of being limited to just a few. A new phase parser allows times to be computed for virtually any seismic phase. The use of Java enables this code to be run on a variety of machine and operating system types, without recompiling. This package also offers the extraction of derivative information, such as ray paths through the earth, pierce and turning points, as well as travel time curves.

A paper has been submitted to Seismological Research Letters, (Crotwell et al., 1998), that is intended to be used as a companion to this manual. While this manual mainly focuses on the praticalities of using the codes, the paper is able to go into more detail on the methodology.

2 Distribution

2.1 What and Where

The current distribution of the TauP package is 1.1, dated February 9, 2001.

The distribution directory obtained from either the gzipped tar file or the jar file contains:

README	getting started information
taup.jar	the jar file with all the classes and standard models
taup.html	a simple web page that loads a rudimentary applet to use the TauP package. Be warned that this is not really meant to be used over the Internet as the download time for 1.5Mb may be too great and most browsers do not yet support the 1.1 version of Java.
exampleProperties	example properties file
HISTORY	change log
License	the free for non-commercial license
bin	a directory with wrapper scripts appropriate for UNIX installations
doc	a directory with Postscript and pdf versions of this manual.
html	a directory with the javadoc output from the source code, mainly useful for writing new java programs that use the TauP package.
jacl	a directory with Jacl examples for accessing the TauP package directly within scripts.
native	a directory with a C library and example program that use the Java Native Interface, providing a basic interface between C programs and the TauP package.
Maple	a directory with Maple scripts showing the time and distance equations used.
src	a directory with all of the java source code.
modelFiles	a directory with the text files for each of the velocity models.

The taup.jar file contains everything needed for a working version of the package. This greatly simplifies the installation process and reduces potential errors. See appendix A for detailed installation instructions.

2.2 Pros and Cons of the Current Release

The increased flexibility of this package provides significant advantages. Among these are:

1. The ability to use many different models. We include a variety of previously created models as well as the option of creating your own models. A conscious effort was made to make as few assumptions as possible about the nature of the model. Therefore, even models that have very different structures than common global models can be used.
2. Phase parsing. Phases are not hard coded into the program, instead the phase names are parsed. This creates an opportunity for the study of less common phases that are not present in previous travel time calculators.
3. Programming interface for Java. Because of the use of the Java programming language, all of the tools exist simultaneously as both applications and libraries. Thus, any Java code that has a need for travel times can load and manipulate the objects within this package. In addition, Jacl, the Java implementation of the popular Tcl scripting language, provides a simple means of directly accessing the public methods within the package.

Of course, there are always drawbacks. The main difficulty at present is speed. The tools in this package are not as fast as natively compiled *C* or FORTRAN. Execution speed, however, is not always the best measure of usefulness.

A extremely fast code that can't use your velocity model, or that won't run on your machine is worse than a slower, but more flexible tool. In addition, processor speed is increasing at a fast rate, and codes that were considered too slow yesterday, are usable today, and will have insignificant execution times tomorrow. One last point is that there is a significant effort within the commercial world to improve the speed of Java. The educational and research world will benefit significantly from these efforts without incurring any cost.

2.3 Future Plans

There are several ideas for improvements that we may pursue, such as:

1. A GUI. A graphical user interface would greatly improve the usefulness of this package, especially for non command line uses such as on the Macintosh or within web browsers. The beginings of such a GUI are there in the TauP tool, but at present it cannot access all of the functionality of the tools.
2. Non-UNIX platforms. In spite of Java's platform neutral nature, our installation instructions and reliance on command line style input and output limits the usability of the package on non-UNIX operating systems. We would like to extend support as much as possible to other operating systems. This should not be a large effort, and thus should be completed soon.
3. Use of the τ function. In spite of the name, TauP does not yet use Tau splines. At present I do not believe that this would provide a large improvement over the current linear interpolation, but it is likely worth doing.
4. Web based applet. One of Java's main uses currently is for the development of web based applets. An applet is a small application that is downloaded and executed within a web browser. This is an attractive opportunity and we have a simple example of one included in this distribution. There are difficulties as the network time to download the model files may be unacceptable, as well as the lack of support for Java 1.1 in current browsers. A client server architecture as well as the continued improvement of commercial web browsers may be able to address these issues.
5. 1.1D models. There is nothing in the method that requires the source and receiver velocity models to be the same. With this idea, a separate crustal model appropriate to each region could be used for the source and receiver.
6. WKBJ synthetics. The calculation of τ is a necessary step for WKBJ synthetics, and so this is a natural direction. It likely involves significant effort, however.

3 Tools

Tools included with the TauP package:

`taup_time`	calculates travel times.
`taup_pierce`	calculates pierce points at model discontinuities and specified depths.
`taup_path`	calculates ray paths, depth versus epicentral distance.
`taup`	a GUI that incorporates the time, pierce and path tools. This requires swing, and hence may not work on some java1.1 systems.
`taup_curve`	calculates travel time curves, time versus epicentral distance.
`taup_table`	outputs travel times for a range of depths and distances in an ASCII file
`taup_setsac`	puts theoretical arrival times into sac header variables.
`taup_create`	creates a .taup model from a velocity model.
`taup_peek`	peeks at a saved model, useful only for debugging.

Each tool is a Java application and has an associated wrapper to make execution easier: sh scripts for UNIX, JBindary double-clickable app;ications for MacOS, and bat files for windows. The applications are machine independent but the wrappers are OS specific. For example, to invoke TauP_Time under UNIX, you could type

```
java -Dtaup.model.path=${TAUPPATH} edu.sc.seis.TauP.TauP_Time -mod prem
```

or simply use the script that does the same thing,

```
taup_time -mod prem
```

Each tool has a `-help` flag that will print a usage summary, as well as a `-version` flag that will print the version.

3.1 Default Parameters

Each of the tools use Java Properties to allow the user to specify values for various parameters. The properties all have default values, which are overridden by values from a Properties file. The tools use `.taup` in the current directory, which overwrites values read in from `.taup` in the user's home directory. In addition, many of the properties can be overridden by command line arguments.

The form of the properties file is very simple. Each property is set using the form

```
taup.property.name=value
```

one property per line. Comment lines are allowed, and begin with a #. Additionally, the names of all of the properties follow a convention of prepending "`taup.`" to the name of the property. This helps to avoid name collisions when new properties are added.

The currently used properties are:

taup.model.name the name of the initial model to be loaded, iasp91 by default.

taup.model.path search path for models. There is no default, but the value in the .taup file will be concatinated with any value of taup.model.path from the system properties. For example, the environment variable TAUPPATH is put into the system property taup.model.path by the wrapper shell scripts.

taup.source.depth initial depth of the source, 0.0 km by default.

taup.phase.list initial phase list, combined with taup.phase.file. The defaults are p, s, P, S, Pn, Sn, PcP, ScS, Pdiff, Sdiff, PKP, SKS, PKiKP, SKiKS, PKIKP, SKIKS.

taup.phase.file initial phase list, combined with taup.phase.list. There is no default value, but the default value for taup.phase.list will not be used if there is a taup.phase.file property.

taup.depth.precision precision for depth output, the default is 1 decimal digit. Note that this is precision, not accuracy. Just because you get more digits doesn't imply that they have any meaning.

taup.distance.precision precision for distance output, the default is 2 decimal digits. Note that this is precision, not accuracy. Just because you get more digits doesn't imply that they have any meaning.

taup.latlon.precision precision for latitude and longitude output, the default is 2 decimal digits. Note that this is precision, not accuracy. Just because you get more digits doesn't imply that they have any meaning.

taup.time.precision precision for time, the default is 2 decimal digits. Note that this is precision, not accuracy. Just because you get more digits doesn't imply that they have any meaning.

taup.rayparam.precision precision for ray parameter, the default is 3 decimal digits. Note that this is precision, not accuracy. Just because you get more digits doesn't imply that they have any meaning.

taup.table.locsat.maxdiff maximum distance in degrees for which Pdiff or Sdiff are put into a locsat table. Beyond this distance Pdiff and Sdiff will not be added to the table, even though they may show up in the output of TauP_Time. Instead, the next later arriving phase, if any, will be used instead. The default is 105 degrees.

taup.create.minDeltaP Minimum difference in slowness between successive slowness samples. This is used to decide when to stop adding new samples due to the distance check. Used by TauP_Create to create new models. The default is 0.1 sec/rad.

taup.create.maxDeltaP Maximum difference in slowness between successive slowness samples. This is used to split any layers that exceed this slowness gap. Used by TauP_Create to create new models. The default is 8.0 sec/rad.

taup.create.maxDepthInterval Maximum difference between successive depth samples. This is used immediately after reading in a velocity model, with layers being split as needed. Used by TauP_Create to create new models. The default is 115 km.

taup.create.maxRangeInterval Maximum difference between successive ranges, in degrees. If the difference in distance for two adjacent rays is greater than this, then a new slowness sample is inserted halfway between the two existing slowness samples. The default is 1.75 degrees.

taup.create.maxInterpError Maximum error for linear interpolation between successive sample in seconds. TauP_Create uses this to try to insure that the maximum error due to linear interpolation is less than this amount. Of course, this is only an approximation based upon an estimate of the curvature of the travel time curve for surface focus turning waves. In particular, the error for more complicated phases is greater. For instance, if the true error for P at 30 degrees is 0.03 seconds, then the error for PP at 60 degrees would be twice that, 0.06 seconds. Used by TauP_Create to create new models. The default is 0.03 seconds.

taup.create.allowInnerCoreS Should we allow J phases, S in the inner core? Used by TauP_Create to create new models. The default is true. Setting it to false slightly reduces storage and model load time.

Phase files, specified with the taup.phase.file property, are just text files with phase names, separated by either spaces, commas or newlines. In section 4 the details of the phase naming convention are introduced. By and large, it is compatible with traditional seismological naming conventions, with a few additions and exceptions. Also, for compatiblity with *ttimes*, you may specify `ttp`, `ttp+`, `tts`, `tts+`, `ttbasic` or `ttall` to get a phase list corresponding to the *ttimes* options.

3.2 TauP_Time

TauP_Time takes a `.taup` file generated by TauP_Create and generates travel times for specified phases through the given earth model.

The usage is:

```
piglet 1>taup_time -help
Usage: taup_time [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Time [arguments]

Arguments are:
-ph phase list     -- comma separated phase list
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.

-h depth           -- source depth in km

Distance is given by:

-deg degrees       -- distance in degrees,
-km kilometers     -- distance in kilometers,
                      assumes radius of earth is 6371km,

or by giving the station and event latitude and lonitude,
                      assumes a spherical earth,

-sta[tion] lat lon -- sets the station latitude and longitude
-evt       lat lon -- sets the event latitude and longitude


-rayp              -- only output the ray parameter
-time              -- only output travel time

-o outfile         -- output is redirected to "outfile"
-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!
```

The modelname is from `modelname.taup`, a previously created file from TauP_Create. If there is insufficient information given on the command line, then you start in interactive mode, otherwise it assumes you only want one set of times.

The phases are specified on the command line with the -ph option, in a phase file with the -pf option, or in a properties file. The model, phases, depth and distance can be changed within the interactive section of TauP_Time.

For example: `taup_time -mod prem -h 200 -ph S,P -deg 57.4`

gives you arrival times for S and P for a 200 kilometer deep source at a distance of 57.4 degrees.

```
piglet 2>taup_time -mod prem -h 200 -ph S,P -deg 57.4

Model: prem
```

```
Distance   Depth   Phase   Travel    Ray Param   Purist    Purist
  (deg)     (km)   Name    Time (s)  p (s/deg)  Distance   Name
------------------------------------------------------------------
    57.4   200.0   P        566.77     6.968       57.4  = P
    57.4   200.0   S       1028.60    13.018       57.4  = S
```

3.3 TauP_Pierce

TauP_Pierce uses a `.taup` file generated by TauP_Create to determine the angular distances from the epicenter at which the specified rays pierce discontinuities or specified depths in the model.

The usage is:

```
piglet 3>taup_pierce -help
Usage: taup_pierce [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Pierce [arguments]

Arguments are:
-ph phase list     -- comma separated phase list
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.

-h depth           -- source depth in km

Distance is given by:

-deg degrees       -- distance in degrees,
-km kilometers     -- distance in kilometers,
                      assumes radius of earth is 6371km,

or by giving the station and event latitude and lonitude,
                      assumes a spherical earth,

-sta[tion] lat lon -- sets the station latitude and longitude
-evt       lat lon -- sets the event latitude and longitude


-az azimuth        -- sets the azimuth (event to station)
                      used to output lat and lon of pierce points
                      if the event lat lon and distance are also
                      given. Calculated if station and event
                      lat and lon are given.
-baz backazimuth   -- sets the back azimuth (station to event)
                      used to output lat and lon of pierce points
                      if the station lat lon and distance are also
                      given. Calculated if station and event
                      lat and lon are given.

-rev               -- only prints underside and bottom turn points, e.g. ^ and v
-turn              -- only prints bottom turning points, e.g. v
-under             -- only prints underside reflection points, e.g. ^
```

```
-pierce depth      -- adds depth for calculating pierce points
-nodiscon          -- only prints pierce points for the depths added with -pierce

-o outfile         -- output is redirected to "outfile"
-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!
```

The `-rev`, `-turn` and `-under` flags are useful for limiting the output to just those points you care about. The `-pierce depth` option allows you to specify a "pierce" depth that does not correspond to an actual discontinuity. For instance, where does a ray pierce 300 kilometers above the CMB?

For example:

```
taup_pierce -mod prem -h 200 -ph S,P -deg 57.4
```

would give you pierce points for S, and P for a 200 kilometer deep source at a distance of 57.4 degrees.

While

```
taup_pierce -turn -mod prem -h 200 -ph S,P -deg 57.4
```

would give you just the points that each ray turns from downgoing to upgoing.

Using `-rev` would give you all points that the ray changes direction and `-under` gives just the underside reflections.

Using the `-pierce` option

```
taup_pierce -mod prem -h 200 -ph S -sta 12 34.2 -evt -28 122 -pierce 2591 -nodiscon
```

would give you just the points at which S crossed a depth of 2591 kilometers from an event at (28° S, 122° E) to a station at (12° N, 34.2° E). Because we specified the latitudes and longitudes, we also get the latitudes and longitudes of the pierce points, useful for making a map view of where the rays encounter the chosen depth. Here is the output, distance, depth, time, latitude and longitude, respectively.

```
> S at  1424.10 seconds at    93.70 degrees for a    200.0 km deep source in the prem mode
   31.58  2591.0   552.8    -17.86     89.39
   61.44  2591.0   821.8     -3.90     62.43
```

3.4 TauP_Path

TauP_Path takes a `.taup` file generated by TauP_Create and generates the path that the phases travel. The output is in GMT (Wessel and Smith, 1995) "psxy" format, and is placed into the file "taup_path.gmt". If you specify the "-gmt" flag then this is a complete script with the appropriate "psxy" command prepended, so if you have GMT installed, you can just:

```
taup_path -mod iasp91 -h 550 -deg 74 -ph S,ScS,sS,sScS -gmt
sh taup_path.gmt
ghostview taup_path.ps
```

and you have a plot of the ray paths. To avoid possible plotting errors for phases like `Sdiff`, the ray paths are interpolated to less than 1 degree increments.

If the -gmt arguement is not given, then the time for each point in the path is also printed. In addition, if the -station and -event options are used, then the latitude and longitude are also printed.

The usage is:

```
piglet 5>taup_path -help
Usage: taup_path [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Path [arguments]

Arguments are:
-ph phase list     -- comma separated phase list
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.

-h depth           -- source depth in km

Distance is given by:

-deg degrees       -- distance in degrees,
-km kilometers     -- distance in kilometers,
                      assumes radius of earth is 6371km,

or by giving the station and event latitude and lonitude,
                      assumes a spherical earth,

-sta[tion] lat lon -- sets the station latitude and longitude
-evt       lat lon -- sets the event latitude and longitude


-gmt             -- outputs path as a complete GMT script.

-o outfile         -- output is redirected to "outfile"
-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!
```

A small section of an output file generated by the command `taup_path -h 100 -sta 10 10 -event 10 30 -ph P` looks like this.

```
> P at    19.69 degrees for a    100.0 km deep source in the iasp91 model.
    0.00    6271.0      0.00     10.00     30.00
    0.01    6270.5      0.12     10.00     29.99
    0.16    6260.7      2.47     10.00     29.84
    0.23    6255.9      3.65     10.01     29.77
    0.30    6251.0      4.83     10.01     29.69
    0.31    6250.7      4.91     10.01     29.69
    0.32    6250.3      4.99     10.01     29.68
    0.37    6246.9      5.83     10.01     29.63
    0.42    6243.4      6.67     10.01     29.57
```

3.5 TauP

TauP is unlike the rest of the tools in that it doesn't have any functionality beyond the other tools. It is just a GUI that uses TauP_Time, TauP_Pierce and TauP_Path. This is a nice feature of the java language in that each of these

applications exists simultaneously as a library. The GUI does not currently have full access to all the things that these three tools can do, and certainly has a few rough edges, but can be useful for browsing. Lastly, it currently does more work than it has to in that it always calculates times, pierce points and paths, even if only one is actually needed. So, it may be a bit pokey.

3.6 TauP_Curve

TauP_Curve creates a GMT style xy formated file of time versus distance. This can be used to create the familar travel time curves, but for only the specified phases and depth. The curves are linearly interpolated between known sample points, and can thus be used to get a feel for the coarseness of sampling. For example, curves for s, S, ScS and Sdiff for a 500 kilometer deep event in PREM could be generated by:

```
taup_curve -mod prem -h 500 -ph s,S,ScS,Sdiff -gmt
```

The `-gmt` option prepends a GMT `psxy` command to the output file, creating a runnable script instead of just a data file. The output is put in taup_curve.gmt by default, so to view the results:

```
sh taup_curve.gmt
ghostview taup_curve.ps
```

The uasage is:

```
piglet 6>taup_curve -help
Usage: taup_curve [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Curve [arguments]

Arguments are:
-ph phase list     -- comma separated phase list
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.

-h depth           -- source depth in km


-gmt               -- outputs curves as a complete GMT script.
-reddeg velocity   -- outputs curves with a reducing velocity (deg/sec).
-redkm velocity    -- outputs curves with a reducing velocity (km/sec).

-o outfile         -- output is redirected to "outfile" instead of taup_curve.gmt
-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!
```

3.7 TauP_SetSac

TauP_SetSac uses the depth and distance information in SAC (Tull, 1989) file headers to put theoretical arrival times into the `t0`–`t9` header variables. The header variable for a phase can be specified with by a dash followed by a number, for instance `S-9` puts the S arrival time in `t9`. If no header is specified then the time will be inserted in the first header variable not allocated to another phase, starting with 0. If there are no header variables not

already allocated to a phase, then the additional phases will not be added to the header. Note that this does not refer to times that are already in the SAC file before TauP_SetSac is run. They will be overwritten.

Note that triplicated phases are a problem as there is only one spot to put a time. For example, in iasp91 S has three arrivals at 20 degrees but only one can be put into the choosen header. TauP_SetSac assumes that the first arrival is the most important, and uses it. An improved method would allow a phase to have several header variables associated with it, so that all arrivals could be marked. Currently however, only the first arrival for a phase name is used.

Warning: TauP_SetSac assumes the EVDP header has depth in meters unless the -evdpkm flag is used, in which case kilometers are assumed. This may be a problem for users that improperly use kilometers for the depth units. Due to much abuse of the SAC depth header units, a warning message is printed if the depth appears to be in kilometers, i.e. it is < 1000, and -evdpkm is not used. This can be safely ignored if the event really is less than 1000 meters deep. See the SAC manual (Tull, 1989) for confirmation.

The SAC files must have EVDP and the O marker set. Also, if GCARC or DIST is not set then TauP_SetSac can calculate a distance only if STLA, STLO, EVLA and EVLO are set.

The user should be very careful about previously set header variables. TauP_SetSac will overwrite any previously set `t` headers. A future feature may do more careful checking, but the current version makes no effort to verify that the header is undefined before writing.

The usage is:

```
piglet 7>taup_setsac -help
Usage: taup_setsac [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_SetSac [arguments]

Arguments are:
-ph phase list     -- comma separated phase list,
                      use phase-# to specify the sac header,
                      for example, ScS-8 puts ScS in t8
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.

-evdpkm            -- sac depth header is in km, default is meters


-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!

sacfilename [sacfilename ...]

Ex: taup_setsac -mod S_prem -ph S-8,ScS-9 wmq.r wmq.t wmq.z
puts the first S arrival in T8 and ScS in T9
```

3.8 TauP_Table

TauP_Table creates an ASCII table of arrival times for a range of depths and distances. Its main use is for generating travel time tables for earthquake location programs such as LOCSAT. The `-generic` flag generates a flat table with all arrivals at each depth and distance, one arrival per line. The `-locsat` flag generates a LOCSAT

style travel time table with only the first arrival of all the phases listed at each distance and depth. Thus, the program must be run several times in order to generate files for several phases. Also, both options write to standard out unless a file is given with the -o flag.

The usage is:

```
piglet 1>taup_table -help
Usage: taup_table [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Table [arguments]

Arguments are:
-ph phase list     -- comma separated phase list
-pf phasefile      -- file containing phases

-mod[el] modelname -- use velocity model "modelname" for calculations
                      Default is iasp91.


-header filename   -- reads depth and distance spacing data
                      from a LOCSAT style file.
-generic           -- outputs a "generic" ascii table
-locsat            -- outputs a "locsat" style ascii table


-o outfile         -- output is redirected to "outfile"
-debug             -- enable debugging output
-verbose           -- enable verbose output
-version           -- print the version
-help              -- print this out, but you already know that!
```

3.9 TauP_Create

TauP_Create takes a ASCII velocity model file, samples the model and saves the tau model to a binary file. The output file holds all information about the model and need only be computed once. It is used by all of the other tools. There are several parameters controlling the density of sampling. Their values can be set with properties. See section 3.1, above.

The usage is:

```
piglet 8>taup_create -help
TauP_Create starting...
Usage: taup_create [arguments]
  or, for purists, java edu.sc.seis.TauP.TauP_Create [arguments]

Arguments are:

   To specify the velocity model:
-nd modelfile       -- "named discontinuities" velocity file
-tvel modelfile     -- ".tvel" velocity file, ala ttimes

-debug              -- enable debugging output
-verbose            -- enable verbose output
-version            -- print the version
-help               -- print this out, but you already know that!
```

`modelfile` is the ASCII text file holding the velocity model. The `-nd` format is preferred because the depths, and thus identities, of the major internal boundaries can be unambiguously determined, making phase name parsing easier. See section 7 for an example. For compatiblity, we support the `-tvel` format currently used by the latest ttimes package, (Kennett et al., 1995).

The output will be a file named after the name of the velocity file, followed by `.taup`. For example

```
taup_create -nd prem.nd
```

produces `prem.taup`.

3.10 TauP_Peek

TauP_Peek is mainly for debugging, it just lets you peek at the insides of a model generated with TauP_Create. The user interface (if you can call it that) is pretty weak, but it came in handy as I was building the codes.

The usage is:

```
taup_peek -mod[el] modelfile
```

4 Phase naming in TauP

A major feature of the TauP Toolkit is the implementation of a phase name parser that allows the user to define essentially arbitrary phases through the earth. Thus, the TauP Toolkit is extremely flexible in this respect since it is not limited to a pre-defined set of phases. Phase names are not hard-coded into the software, rather the names are interpreted and the appropriate propagation path and resulting times are constructed at run time. Designing a phase-naming convention that is general enough to support arbitrary phases and easy to understand is an essential and somewhat challenging step. The rules that we have developed are described here. Most of phases resulting from these conventions should be familiar to seismologists, e.g. pP, PP, PcS, PKiKP, etc. However, the uniqueness required for parsing results in some new names for other familiar phases.

In traditional "whole-earth" seismology, there are 3 major interfaces: the free surface, the core-mantle boundary, and the inner-outer core boundary. Phases interacting with the core-mantle boundary and the inner core boundary are easy to describe because the symbol for the wave type changes at the boundary (i.e. the symbol P changes to K within the outer core even though the wave type is the same). Phase multiples for these interfaces and the free surface are also easy to describe because the symbols describe a unique path. The challenge begins with the description of interactions with interfaces within the crust and upper mantle. We have introduced two new symbols to existing nomenclature to provide unique descriptions of potential paths. Phase names are constructed from a sequence of symbols and numbers (with no spaces) that either describe the wave type, the interaction a wave makes with an interface, or the depth to an interface involved in an interaction.

1. Symbols that describe wave-type are:
   - `P` compressional wave, upgoing or downgoing, in the crust or mantle
   - `p` strictly upgoing P wave in the crust or mantle
   - `S` shear wave, upgoing or downgoing, in the crust or mantle
   - `s` strictly upgoing S wave in the crust or mantle
   - `K` compressional wave in the outer core
   - `I` compressional wave in the inner core
   - `J` shear wave in the inner core
2. Symbols that describe interactions with interfaces are:
   - `m` interaction with the moho
   - `g` appended to P or S to represent a ray turning in the crust
   - `n` appended to P or S to represent a head wave along the moho
   - `c` topside reflection off the core mantle boundary
   - `i` topside reflection off the inner core outer core boundary
   - `^` underside reflection, used primarily for crustal and mantle interfaces
   - `v` topside reflection, used primarily for crustal and mantle interfaces
   - `diff` appended to P or S to represent a diffracted wave along the core mantle boundary
   - `kmps` appended to a velocity to represent a horizontal phase velocity (see 10 below)
3. The characters `p` and `s` **always** represent up-going legs. An example is the source to surface leg of the phase `pP` from a source at depth. `P` and `S` can be turning waves, but always indicate downgoing waves leaving the source when they are the first symbol in a phase name. Thus, to get near-source, direct P-wave arrival times, you need to specify two phases `p` and `P` or use the "*ttimes* compatibility phases" described below. However, `P` may represent a upgoing leg in certain cases. For instance, `PcP` is allowed since the direction of the phase is unambiguously determined by the symbol `c`, but would be named `Pcp` by a purist using our nomenclature.
4. Numbers, except velocities for `kmps` phases (see 10 below), represent depths at which interactions take place. For example, `P410s` represents a P-to-S conversion at a discontinuity at 410km depth. Since the S-leg is given by a lower-case symbol and no reflection indicator is included, this represents a P-wave converting to an S-wave when it hits the interface from below. The numbers given need not be the actual depth, the closest depth corresponding to a discontinuity in the model will be used. For example, if the time for `P410s` is requested in a model where the discontinuity was really located at 406.7 kilometers depth,

the time returned would actually be for `P406.7s`. The code "taup_time" would note that this had been done. Obviously, care should be taken to ensure that there are no other discontinuities closer than the one of interest, but this approach allows generic interface names like "410" and "660" to be used without knowing the exact depth in a given model.

5. If a number appears between two phase legs, e.g. `S410P`, it represents a transmitted phase conversion, not a reflection. Thus, `S410P` would be a transmitted conversion from `S` to `P` at 410km depth. Whether the conversion occurs on the down-going side or up-going side is determined by the upper or lower case of the following leg. For instance, the phase `S410P` propagates down as an `S`, converts at the 410 to a `P`, continues down, turns as a P-wave, and propagates back across the 410 and to the surface. `S410p` on the other hand, propagates down as a `S` through the 410, turns as an `S`, hits the 410 from the bottom, converts to a `p` and then goes up to the surface. In these cases, the case of the phase symbol (P vs. p) is critical because the direction of propagation (upgoing or downgoing) is not unambiguously defined elsewhere in the phase name. The importance is clear when you consider a source depth below 410 compared to above 410. For a source depth greater than 410 km, `S410P` technically cannot exist while `S410p` maintains the same path (a receiver side conversion) as it does for a source depth above the 410.

   The first letter can be lower case to indicate a conversion from an up-going ray, e.g. `p410S` is a depth phase from a source at greater than 410 kilometers depth that phase converts at the 410 discontinuity. It is strictly upgoing over its entire path, and hence could also be labeled `p410s`. `p410S` is often used to mean a reflection in the literature, but there are too many possible interactions for the phase parser to allow this. If the underside reflection is desired, use the `p^410S` notation from rule 7.

6. Due to the two previous rules, `P410P` and `S410S` are over specified, but still legal. They are almost equivalent to `P` and `S`, respectively, but restrict the path to phases transmitted through (turning below) the 410. This notation is useful to limit arrivals to just those that turn deeper than a discontinuity (thus avoiding travel time curve triplications), even though they have no real interaction with it.

7. The characters `^` and `v` are new symbols introduced here to represent bottom-side and top-side reflections, respectively. They are followed by a number to represent the approximate depth of the reflection or a letter for standard discontinuities, `m`, `c` or `i`. Reflections from discontinuities besides the core-mantle boundary, `c`; or inner-core outer-core boundary, `i`, must use the `^` and `v` notation. For instance, in the TauP convention, `p^410S` is used to describe a near-source underside reflection.

   Underside reflections, except at the surface (`PP`, `sS`, etc.), core-mantle boundary (`PKKP`, `SKKKS`, etc.), or outer-core-inner-core boundary (`PKIIKP`, `SKJJKS`, `SKIIKS`, etc.), must be specified with the `^` notation. For example, `P^410P` and `P^mP` would both be underside reflections from the 410km discontinuity and the Moho, respectively.

   The phase `PmP`, the traditional name for a top-side reflection from the Moho discontinuity, must change names under our new convention. The new name is `PvmP` or `Pvmp` while `PmP` just describes a P-wave that turns beneath the Moho. The reason the Moho must be handled differently from the core-mantle boundary is that traditional nomenclature did not introduce a phase symbol change at the Moho. Thus, while `PcP` makes sense since a P-wave in the core would be labeled `K`, `PmP` could have several meanings. The `m` symbol just allows the user to describe phases interaction with the Moho without knowing its exact depth. In all other respects, the `^`-`v` nomenclature is maintained.

8. Currently, `^` and `v` for non-standard discontinuities are allowed only in the crust and mantle. Thus there are no reflections off non-standard discontinuities within the core, (reflections such as `PKKP`, `PKiKP` and `PKIIKP` are still fine). There is no reason in principle to restrict reflections off discontinuities in the core, but until there is interest expressed, these phases will not be added. Also, a naming convention would have to be created since "`p` is to `P`" is not the same as "`i` is to `I`".

9. Currently there is no support for `PKPab`, `PKPbc`, or `PKPdf` phase names. They lead to increased algorithmic complexity that at this point seems unwarranted. Currently, in regions where triplications develop, the triplicated phase will have multiple arrivals at a given distance. So, `PKPab` and `PKPbc` are both labeled just `PKP` while `PKPdf` is called `PKIKP`.

10. The symbol `kmps` is used to get the travel time for a specific horizontal phase velocity. For example, `2kmps` represents a horizontal phase velocity of 2 kilometers per second. While the calculations for these are trivial, it is convenient to have them available to estimate surface wave travel times or to define windows of interest for given paths.

11. As a convenience, a *ttimes* phase name compatibility mode is available. So `ttp` gives you the phase list corresponding to `P` in *ttimes*. Similarly there are `tts`, `ttp+`, `tts+`, `ttbasic` and `ttall`.

5 Creating and Saving Velocity Models

5.1 Velocity Model Files

There are currently two variations of velocity model files that can be read. Both are piecewise linear between given depth points. Support for cubic spline velocity models would be useful and is planned for a future release.

The first format is that used by the most recent ttimes codes (Kennett et al., 1995), `.tvel`. This format has two comment lines, followed by lines composed of depth, Vp, Vs and density, all separated by whitespace. TauP ignores the first two lines of this format and reads the remaining lines.

The second format is based on the format used by Xgbm, (Davis and Henson, 1993a; Davis and Henson, 1993b). It is referred to here as the `.nd` format for "named discontinuities." Its biggest advantage is that it can specify the location of the major boundaries and this makes it the preferred format. The file consists of two types of lines, those that specify velocity at a depth, and those that specify the name of a discontinuity.

The first type of line has between 3 and 6 numbers on a line separated by whitespace. They are, in order, depth in kilometers to the sample point, P velocity in kilometers per second, S velocity in kilometers per second, density in grams per cubic centimeter, Q_p attenuation for compressional waves and Q_s attenuation for shear waves. Only depth, V_p and V_s are required. The remaining parameters, while not needed for travel time calculations, are included to allow the model to be used for other purposes in the future. The model is assumed to be linear between given depths and repeated depths are used to represent discontinuities.

The second type of line within the `.nd` format specifies one of the three major internal boundaries, *mantle* for the crust-mantle boundary, *outer-core* for the outer core-mantle boundary, or *inner-core* for the inner core-outer core boundary. These labels are placed on a line by themselves between the two lines representing the sample points above and below the depth of the discontinuity. These help to determine where a particular phase propagates. For instance, in a model that has many crustal and upper mantle layers, from which discontinuity does the phase `PvmP` reflect? Explicit labeling eliminates potential ambiguity.

One further enhancement to these model file formats is the support for comments embedded within the model files. As in shell scripting, everything after a # on a line is ignored. In addition, *C* style `/* ... */` and *C++* style `// ...` comments are recognized.

A very simple *named discontinuities* model file might look like this:

```
/* below is a simple named discontinuities model. */
0.0  5.0  3.0  2.7
20   5.0  3.0  2.7
20   6.5  3.7  2.9
33   6.5  3.7  2.9
mantle          # the word "mantle" designates that this is the moho
33   7.8  4.4  3.3
410  8.9  4.7  3.5
410  9.1  4.9  3.7
670  10.2 5.5  4.0
670  10.7 5.9  4.4
2891 13.7 7.2  5.6
outer-core      # "outer-core" designates that this is the core mantle boundary
2891 8.0  0.0  9.9
5149.5 10.3 0.0 12.2
inner-core      # "inner-core" makes this the inner-outer core boundary
5149.5 11 3.5 12.7
6371 11.3  3.7  13
```

5.2 Using Saved Tau Models

There are three ways of finding a previously generated model file. First, as a standard model as part of the distribution. Second, a list of directories and jar files to be searched can be specified with the taup.model.path property. Lastly, the path to the actual model file may be specified. TauP searches each of these places in order until it finds a model that matches the name.

1. Standard Model.

   TauP first checks to see if the model name is associated with a standard model. Several standard models are included within the distributed jar file. They include iasp91 (Kennett and Engdahl, 1991), prem (Dziewonski and Anderson, 1984), ak135 (Kennett, Engdahl, and Buland, 1995), jb (Jeffreys and Bullen, 1940), 1066a (Gilbert and Dziewonski, 1975), 1066b (Gilbert and Dziewonski, 1975), pwdk (Weber and Davis, 1990), sp6 (Morelli and Dziewonski, 1993) and herrin (Herrin, 1968). Lastly, we have included qdt, which is a coarsely sampled version of iasp91 (Kennett and Engdahl, 1991). It is samller, and thus loads quicker, but has significantly reduced accuracy. We will consider adding other models to the distribution if they are of wide interest. They are included within the distribution jar file but taup can locate them with just the model name.

2. Within the taup.model.path property.

   Users can create custom models, and place the stored models in a convenient location. If the taup.model.path property includes those directories or jar files, then they can be located. The search is done in the order of taup.model.path until a model matching the model name is found. While taup.model.path is a Java property, the shell scripts provided translate the environment variable TAUPPATH into this property. The user generally need not be aware of this fact except when the tools are invoked without using the provided shell scripts. A more desirable method is to set the taup.model.path in a properties file. See section 3.1 for more details.

   The taup.model.path property is constructed in the manner of standard Java CLASSPATH which is itself based loosely on the manner of the UNIX PATH. The only real differences between CLASSPATH and PATH are that a jar file may be placed directly in the path and the path separator character is machine dependent, UNIX is ':' but other systems may vary.

   The taup.model.path allows you to have directories containing saved model files as well as jar files of models. For instance, in a UNIX system using the c shell, you could set your TAUPPATH to be, (all one line):

   ```
   setenv TAUPPATH /home/xxx/MyModels.jar:/home/xxx/ModelDir:
   /usr/local/lib/localModels.jar
   ```

   or you could place a line in the `.taup` file in your home directory that accomplished the same thing, again all one line:

   ```
   taup.model.path=/home/xxx/MyModels.jar:/home/xxx/ModelDir:
   /usr/local/lib/localModels.jar
   ```

   If you place models in a jar, TauP assumes that they are placed in a directory called `Models` before they are jarred. For example, you might use taup_create to create several taup models in the Models directory and then create a jar file.

   ```
   jar -cf MyModels.jar Models
   ```

   Including a "." for the current working directory with the taup.model.path is not necessary since we always check there, see 3 below, but it may be used to change the search order.

3. The last place TauP looks is for a model file specified on the command line. So, if you generate newModel.taup and want to get some times, you can just say: `taup_time -mod newModel.taup` or even just `taup_time -mod newModel` as TauP can add the taup suffix if necessary. A relative or absolute pathname may precede the model, e.g. `taup_time -mod ../OtherDir/newModel.taup`.

6 Programming Interface

In addition to the command line interface, there are three ways to access the toolkit from within other programs. The most straightforward is through Java. Using Jacl provides a very nice way to write scripts that use the tools without repeatedly starting up the Java virtual machine and reloading models. Lastly, there is a C language interface, but it is a bit less friendly. Descriptions of all three, with example programs are below.

6.1 Java

The TauP package should be easily used by future Java programs. An example is given illustrating the basics of using the package to generate travel times.

First, instantiate a TauP_Time object. This provides methods for generating and using travel times and should be sufficient for most purposes. However, in order to actually generate anything useful, the TauP_Time object needs a TauModel. It can be loaded within the constructor for TauP_Time as a TauModel or with the model name. It can changed later using either the `TauP_Time.setTauModel(TauModel)` method of TauP_Time, or by passing the modelname to `TauP_Time.loadTauModel(String)`. The later is likely easier, and has the advantage of searching for the model in the distribution jar file, the locations in the taup.model.path property, and the current directory.

```
TauP_Time timeTool = new TauP_Time("mymodel");
```

In addition to the TauModel, a collection of phases is also needed. Again, there are several ways of accomplishing this. `parsePhaseList(String)` is likely the easiest method. A String is passed with the phase names separated by commas and the phases are extracted and appended. Phases can also be input more directly with `setPhaseNames(String[])` , which sets the phases to be those in the array, and `appendPhaseName(String)` which appends a phase to the list. Note that these methods do not do any checking to assure the names are valid phases, this is done at a later stage. Of additional interest are `clearPhaseNames()` which deletes all current phase names, and `getPhaseNames()` which returns an array of Strings with the phase names.

```
timeTool.parsePhaseList("P,Pdiff,S,Sdiff,PKP,SKS");
```

The next step is to correct the TauModel for the source depth. The TauModel is created with a surface source, but can be corrected for a source at depth, given in kilometers, with the `depthCorrect(double)` method. In addition, if a correction was actually needed, it calls `recalcPhases()` which verifies that the times and distances for the phases in the phase list are compatible with the current model and depth. `recalcPhases()` is also called by `calculate()` in case changes were made to the list of phase names.

```
timeTool.depthCorrect(100.0);
```

It remains only to calculate arrivals for a particular distance using the `calculate(double)` method, which takes an angular distance in degrees. The arrivals are stored as Arrival objects, which contain `time`, `dist`, `rayParam`, `sourceDepth`, and `name` fields. The Arrivals can be accessed through either the `getArrival(int)` method which returns the ith arrival, or the `getArrivals()` method which returns an array of Arrivals. Of additional interest is the `getNumArrivals()` method that returns the number of arrivals.

```
timeTool.calculate(40);
Arrival[] arrivals = timeTool.getArrivals();
for (int i=0; i<arrivals.length; i++) {
   System.out.println(arrivals[i].getName+" arrives at "+
      (arrivals[i].getDist*180.0/Math.PI)+" degrees after "+
      arrivals[i].getTime+" seconds.");
}
```

It is important to realize that all internal angular distances are stored as radians, hence the conversion, and times in seconds. This also means that the ray parameters are stored as seconds per radian.

6.2 Jacl

One of the problems with Java based tools is that there is overhead associated with starting a Java program due to the fact that the virtual machine must first be started. While with normal interactive computing this is not such a large problem, it can become very wastful when repeated calling a tool from within a script. Significant time savings can be had if the tool and its associated virtual machine can be kept alive for the duration of the script. Jacl, a Java implementation of the popular Tool Command Language or Tcl, makes writing scripts that use the TauP Toolkit easy, and allows one instance of both the virtual machine as well as the tool to remain active for the whole script. You may download jacl from http://www.scriptics.com/java.

Jacl allows a script to create Java objects, call any public method of those objects and manipulate their attributes. Thus, creating a script to do many similar calcuations or a custom application that makes these tools usable in the way you want is as easy as writing a tcl script. We present a brief walkthrough of a Jacl script that calculates pierce points for numerous station event pairs.

The first three lines of the script should start up jacl. The second line is a bit of trickery, it hides the third line from jacl while allowing sh to see it. Jacl takes the backslash to be a line continuation marker, and therefore accepts the third line as part of the comment started on the second line. This just makes it easier to start up jacl without having to know the exact path in advance.

```
#!/bin/sh
# \
exec jacl $0 $*
```

Next, we will set up latitudes and longitudes for our stations and events. This was modified from a script that read from a CSS database, but in order to keep the script self contained, we have hardwired it here.

```
set slat(0) 35
set slon(0) -5
set elat(0) 125
set elon(0) 5
set edepth(0) 100
set elat(1) -10
set elon(1) 110
set edepth(1) 100
set elat(2) 40
set elon(2) 140
set edepth(2) 200
set elat(3) 65
set elon(3) -5
set edepth(3) 10
```

Now we start up the pierce tool with the prem model and add the phases we are interested in. We will only do P and S in PREM for simplicity.

```
set taup [java::new [list edu.sc.seis.TauP.TauP_Pierce String] "prem"]
$taup clearPhaseNames
$taup {parsePhaseList java.lang.String} "P,S"
```

Here we get, and then loop over, all the discontinuities in the model in order to find the one closest to 400 kilometers depth.

```
set disconArray [$taup getDisconDepths]
set maxDiff 99999999
```

```
set bestDepth 0
for {set i 0} {$i < [$disconArray length]} {incr i} {
   set depth [$disconArray get $i]
if { [expr abs($depth - 400)] < $maxDiff} {
      set maxDiff [expr abs($depth - 400)]
      set bestDepth $depth
   }
}
```

Loop over all events and stations and output the pierce point at the 400 kilometer discontinuity. We use the getLastPiercePoint(depth) method as we want the receiver side pierce point. If we wanted the source side point we could have used the getFirstPiercePoint(depth) method.

```
for {set eventIndex 0} {$eventIndex < [array size elat]} {incr eventIndex} {
   $taup depthCorrect $edepth($eventIndex)
   for {set staIndex 0} {$staIndex < [array size slat]} {incr staIndex} {
      set gcarc [java::call edu.sc.seis.TauP.SphericalCoords distance \
         $elat($eventIndex) $elon($eventIndex) \
         $slat($staIndex) $slon($staIndex)]
      set az     [java::call edu.sc.seis.TauP.SphericalCoords azimuth \
         $elat($eventIndex) $elon($eventIndex) \
         $slat($staIndex) $slon($staIndex)]
      $taup calculate $gcarc
      set numArrivals [$taup getNumArrivals]

      if {$numArrivals == 0} {
         puts "No arrivals for event $eventIndex"
      }
      for {set k 0} {$k< $numArrivals} {incr k} {
         set OneArrival [$taup getArrival $k]
         set name [ $OneArrival getName]

         if [ catch \
         {set OnePierce [$OneArrival getLastPiercePoint $bestDepth] }] {
            puts "$name doesn't pierce $bestDepth for event $eventIndex"
            continue
         }

         set dist [ $OnePierce getDist]
         set dist [expr $dist * (180./3.14159)]
         set plat [java::call edu.sc.seis.TauP.SphericalCoords latFor \
            $elat($eventIndex) $elon($eventIndex) $dist $az ]
         set plon [java::call edu.sc.seis.TauP.SphericalCoords lonFor \
            $elat($eventIndex) $elon($eventIndex) $dist $az ]
         puts [format "(%-7.3f, %-7.3f) $name from event number $eventIndex" \
            $plat $plon ]
      }
   }
}
```

And here is the output:

```
piglet 56>./pierce.jacl
No arrivals for event 0
```

```
(-7.218 , 36.679 ) P from event number 1
(-7.214 , 36.676 ) S from event number 1
(-2.185 , 35.266 ) P from event number 2
(-2.205 , 35.264 ) S from event number 2
(-3.262 , 34.492 ) P from event number 3
(-3.142 , 34.457 ) S from event number 3
```

This script, along with another simple travel time script, is included in the distribution in the jacl subdirectory.

6.3 C

A C language interface to the TauP package is provided. A shared library libtaup.so, provides access to the core functionality for generating travel times. An example program using these interface routines is also provided, gettimes.c.

The native interface is distributed as C source code that you must compile on your local machine. A makefile is provided to generate a shared library and an example code to call the library. The makefile was created for use under Solaris, but doesn't do anything particularly special, and should be easily modifiable for other operating systems.

Of course, the system must be able to find this library, as well as the Java libraries. Under Solaris, this can be accomplished with the `LD_LIBRARY_PATH` environment variable. Other systems may vary. The `CLASSPATH` environment variable must also contain the taup.jar file as well as the default java jar files. Note that under Java1.2 the command line tools may work fine while the C interface has problems. This is due to the java executable finding the standard files without the `CLASSPATH`. The C interface bypasses the executable, and so does not benefit from this. Properly setting the `CLASSPATH` is thus even more important for calling Java from C.

The current C interface only provides method calls for the most basic operations for getting travel times. If less common methods need to be called then a quick look at the source code in the native directory should be sufficient to create new hooks into those methods.

The state of the travel time calculator is preserved from call to call within a TauPStruct structure. This contains references to the java virtual machine, each of the method calls and the current model. This structure is always the first argument to all of the method calls. While I believe this is the least complicated style of interaction, it is not particularly memory or processor efficient for uses involving more than one travel time calculator active simultaneously. Primarily this is due to having more than one java virtual machine running at the same time. Still, it is a good example of how C can interact with Java.

The currently implemented method calls are:

TauPInit initializes the java virtual machine and properly fills in the TauPStruct passed as the first argument. The second argument is the name of the model to be used. The method signature is
int TauPInit(TauPStruct *taupptr, char *modelName) ;

TauPSetDepth sets the source depth within the model. A initialized TauPStruct is passed as the first argument, with the source depth passed as the second. With the exception of creating a new model, this is the most CPU intensive operation. The method signature is
int TauPSetDepth(TauPStruct taup, double depth) ;

TauPClearPhases clears any previously added phases. This should be followed by a call to TauPAppendPhases, below, to add new phases. An initialized TauPStruct is passed as the first argument. The method signature is
int TauPClearPhases(TauPStruct taup) ;

TauPAppendPhases appends new phases for calculation. An initialized TauPStruct is passed as the first argument and the phase names are passed as a comma or space separated string in the second argument. All of the

phase names that can be used in the interactive code can be used here. Also, duplicates are checked for and eliminated before being added. The method signature is
int TauPAppendPhases(TauPStruct taup, char *phaseString) ;

TauPCalculate calculates all arrivals for all of the current phases for the distance specified in the second argument. An initialized TauPStruct is passed as the first argument. The method signature is
int TauPCalculate(TauPStruct taup, double degrees) ;

TauPGetNumArrivals returns the number of arrivals found with the last call to TauPCalculate, above. A negative number indicates an error. An initialized TauPStruct is passed as the first argument. The method signature is
int TauPGetNumArrivals(TauPStruct taup) ;

TauPGetArrival returns the ith arrival found with the last call to TauPCalculate, above. The arrival is returned as a jobject, which is mainly useful if it will be used as an argument for another java method call. NULL is returned if an error occurs. An initialized TauPStruct is passed as the first argument. The method signature is
jobject TauPGetArrival(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalName returns the name of the ith arrival found with the last call to TauPCalculate, above, as a character pointer. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. NULL is returned if there is an error. The method signature is
char * TauPGetArrivalName(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalPuristName returns the purist's version of the name of the ith arrival found with the last call to TauPCalculate, above, as a character pointer. The puris's name replaces depths with the true depth of interfaces in the phase name, for example Pv410P might really be Pv400P. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. NULL is returned if there is an error. The method signature is
char * TauPGetArrivalPuristName(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalTime returns the travel time of the ith arrival found with the last call to TauPCalculate, above. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. A negative number is returned if there is an error. The method signature is
double TauPGetArrivalTime(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalDist returns the travel distance of the ith arrival found with the last call to TauPCalculate, above. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. A negative number is returned if there is an error. The method signature is
double TauPGetArrivalDist(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalRayParam returns the ray parameter of the ith arrival found with the last call to TauPCalculate, above. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. A negative number is returned if there is an error. The method signature is
double TauPGetArrivalRayParam(TauPStruct taup, int arrivalNum) ;

TauPGetArrivalSourceDepth returns the source depth of the ith arrival found with the last call to TauPCalculate, above. An initialized TauPStruct is passed as the first argument and the arrival number is passed as the second. A negative number is returned if there is an error. The method signature is
double TauPGetArrivalSourceDepth(TauPStruct taup, int arrivalNum) ;

TauPDestroy destroys the java virtual machine and frees the used memory. An initialized TauPStruct is passed as the first argument. A nonzero error is returned if there is an error. The method signature is
int TauPDestroy(TauPStruct taup) ;

7 Examples

Here is a walk through of a use of the tools on a UNIX system.

7.1 Velocity Model Files

First, we want to create a model. There are several models contained within the TauP distribution, but for completeness we will create a new one from scratch.

A very simple model file might look like this:

```
0.0     5.0  3.0   2.7
20      5.0  3.0   2.7
20      6.5  3.7   2.9
33      6.5  3.7   2.9
mantle
33      7.8  4.4   3.3
410     8.9  4.7   3.5
410     9.1  4.9   3.7
670    10.2  5.5   4.0
670    10.7  5.9   4.4
1000   11.5  6.4   4.6
2000   12.9  6.9   5.1
2891   13.7  7.2   5.6
outer-core
2891    8.0  0.0   9.9
3500    9.0  0.0  10.8
5149.5 10.3  0.0  12.2
inner-core
5149.5 11    3.5  12.7
5500   11.1  3.6  12.9
6371   11.3  3.7  13
```

Note that we have chosen the "named discontinuities" format so that we could specify the location of the major boundaries. The file consists of two types of lines, those that specify velocity at a depth, and those that specify the name of a discontinuity. See section 5.1 for more details.

7.2 Creating the Model

If we put this into a file called "simpleMod.nd" then we can run `taup_create` to create a model sampling. We use the `-verbose` option to get some additional output. In particular, it outputs the radius to the surface for this model. Having an incorrect radius, which could happen for instance if the last line of the model file was lost, will generate incorrect times for phases that otherwise appear fine. This can be a difficult error to track down after the fact because there is nothing wrong with the model, it is just not what was intended. As they say, "garbage in, garbage out."

```
piglet 10>taup_create -nd simpleMod.nd -verbose
TauP_Create starting...
filename =./simpleMod.nd
Done reading velocity model.
Radius of model simpleMod is 6371.0
```

```
Parameters are:
taup.create.minDeltaP = 0.1 sec / radian
taup.create.maxDeltaP = 8.0 sec / radian
taup.create.maxDepthInterval = 115.0 kilometers
taup.create.maxRangeInterval = 1.75 degrees
taup.create.maxInterpError = 0.03 seconds
taup.create.allowInnerCoreS = true
Slow model time=39714 801 P layers,907 S layers
T model time=7480
Done Saving ./simpleMod.taup
Done!
Done!
piglet 11>ls
simpleMod.nd     simpleMod.taup
```

The file `simpleMod.taup` contains all of the information about the model. This process needs to be done only once for each velocity model. The times appearing in the output are in milliseconds, and do not reflect the startup time for Java.

7.3 Travel Times

Now that we have the model sampled, computing travel times is easy. We will use taup_time to get the travel times for some familiar phases, P, S, PcP, ScS, SKS, sS, and SS in our simple model for a 143.2 kilometer deep source and at a distance of 75 degrees. We use the "-mod" command line flag to specify the model, and then do the rest after it starts.

First taup_time reads a standard Java Properties file, ".taup", that it finds in my home directory. See section 3.1 for more details. If there are phases you are interested in frequently, or model you use often, or source depth, then you can put them in this file as your defaults. Then we enter a depth for the source, 143.2 kilometers, using the `h` option. By default, the model is for a surface source.

Some phase names have been read in from the file, but we want to specify our own phase list, so we use the 'c' option to clear the phases and are prompted to enter the new phases. Enter them separated by commas or spaces. After that we just need to enter the distance, 75 degrees. The arrivals are printed as distance, depth, phase name, time and then ray parameter. The last two entries represent a "purists" view of the distance and phase name. For instance, PKKP travels the long way around the earth, and so the true distance traveled is not the event to station distance. The purist's view of the name is to show the difference between the true depths of discontinuities and the depth specified in the phase name. For instance, Pv400P in our simple model is really a reflection off of the discontinuity at 410 kilometers depth. The purist's name reflects this and is preceded by an asterisk to make the difference easier to notice. The distance is repeated to make it easier to parse the output from within scripts.

```
piglet 4>taup_time -mod simpleMod
Enter:
h for new depth
r to recalculate
p to append phases,
c to clear phases
l to list phases
s for new station lat lon
e for new event lat lon
a for new azimuth
b for new back azimuth
m for new model or
q to quit.
```

```
Enter Distance or Option [hrpclseabmq]: h
Enter Depth: 143.2
Enter Distance or Option [hrpclseabmq]: c
Enter phases (ie P,p,PcP,S): P,S,PcP,ScS,SKS,sS,SS,PKKP
Enter Distance or Option [hrpclseabmq]: 75

Model: simpleMod
Distance   Depth   Phase   Travel    Ray Param   Purist    Purist
  (deg)     (km)   Name    Time (s)  p (s/deg)  Distance   Name
------------------------------------------------------------------
    75.0   143.2   P        686.33     5.721       75.0  = P
    75.0   143.2   PcP      700.51     4.312       75.0  = PcP
    75.0   143.2   S       1263.17    11.040       75.0  = S
    75.0   143.2   SKS     1293.35     7.283       75.0  = SKS
    75.0   143.2   ScS     1298.74     8.135       75.0  = ScS
    75.0   143.2   sS      1326.73    11.152       75.0  = sS
    75.0   143.2   SS      1571.74    14.640       75.0  = SS

Enter Distance or Option [hrpclseabmq]: q
```

We could also have done this same example by just using the command line options.

```
piglet 5>taup_time -mod simpleMod -h 143.2 -deg 75 -ph P,S,PcP,ScS,SKS,sS,SS,PKKP

Model: simpleMod
Distance   Depth   Phase   Travel    Ray Param   Purist    Purist
  (deg)     (km)   Name    Time (s)  p (s/deg)  Distance   Name
------------------------------------------------------------------
    75.0   143.2   P        686.33     5.721       75.0  = P
    75.0   143.2   PcP      700.51     4.312       75.0  = PcP
    75.0   143.2   S       1263.17    11.040       75.0  = S
    75.0   143.2   SKS     1293.35     7.283       75.0  = SKS
    75.0   143.2   ScS     1298.74     8.135       75.0  = ScS
    75.0   143.2   sS      1326.73    11.152       75.0  = sS
    75.0   143.2   SS      1571.74    14.640       75.0  = SS
```

7.4 Pierce Points

Now, where are the turning points for these rays? We can run taup_pierce with the "-turn" flag and find out. Lets specify the parameters on the command line. The output is distance in degrees followed by depth in kilometers. Note that SS has two turning points.

```
piglet 7>taup_pierce -mod simpleMod -h 143.2 -deg 75 \
? -ph P,S,PcP,ScS,SKS,sS,SS,PKKP -turn
> P at 686.33 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   37.23 2110.32
> S at 1263.17 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   37.20 2005.24
> PcP at 700.51 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
```

```
   37.30 2891.00
> ScS at 1298.74 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   37.29 2891.00
> SKS at 1293.35 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   37.31 2975.11
> sS at 1326.73 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   37.81 1971.17
> SS at 1571.74 seconds at 75.0 degrees for a 143.2 km deep source in the sim-
pleMod model.
   18.09 1001.76
   56.03 1001.76
```

7.5 Path

Perhaps now we should make a plot of the paths. Lets use only command line options and send the output to the file "simpleModPaths.gmt" instead of the default "taup_path.gmt".

```
piglet 8>taup_path -mod simpleMod -h 143.2 -deg 75 \
? -ph P,S,PcP,ScS,SKS,sS,SS,PKKP \
? -o simpleModPaths.gmt -gmt
piglet 9>ls
simpleMod.taup       simpleMod.nd         simpleModPaths.gmt
piglet 10>sh simpleModPaths.gmt
piglet 11>ls
simpleMod.taup       simpleModPaths.ps
simpleMod.nd         simpleModPaths.gmt
```

Now we have a Postscript file, simpleModPaths.ps, that we can look at or print. Notice that we used the `-gmt` flag so that the output is a complete GMT script. If you don't use `-gmt`, then the output is just the XY points, which might be later used by another script. Of course, this only works if you have GMT installed.

7.6 Travel Time Curves

If we want to see the travel time curves for these phases, we can do that using taup_curve. It works very similarly to taup_path except that we don't need to specify a distance.

```
piglet 12>taup_curve -mod simpleMod -h 143.2 -ph P,S,PcP,ScS,SKS,sS,SS,PKKP \
? -o simpleModCurves.gmt -gmt
piglet 13>ls
simpleModCurves.gmt   simpleMod.nd          simpleModPaths.gmt
simpleMod.taup        simpleModPaths.ps
piglet 14>sh simpleModCurves.gmt
piglet 15>ls
simpleMod.nd          simpleModCurves.gmt   simpleModPaths.gmt
simpleMod.taup        simpleModCurves.ps    simpleModPaths.ps
```

Again we have a Postscript file to view. Both of these commands generate scripts that are ok for a quick look, but you will almost certainly want to modify them for any important use.

A Installing

The installation for TauP under UNIX is quite simple. And with Java's platform independence, the package should be usable on a Mac or Windows machine.

A.1 Unix

1. Install a Java 1.1 or better virtual machine. If your system already has Java 1.1 or better installed then you can skip to the next step. You can test this with "java -version". If it isn't there or the version is less than 1.1 you need to get and install Java.

   If you have a Sun Solaris workstation, you need only down load the version from JavaSoft. Point your browser to http://www.javasoft.com/products/ and download either the Java Development Kit or the Java Runtime Environment.

   The Java Runtime Environment (jre) is the smaller of the two, only allowing you to run java applications and applets. The Java Development Kit (jdk) is larger but allows you to compile and run java programs. Unless space is at a premium, I suggest getting the jdk. A quick test to see if you have the jdk is to see if javac, the java compiler is installed. The command "which javac" or just "javac" should tell you.

   There are ports of Java for many other operating systems, and as long as they are a Java 1.1 or better compatible implementation, the tools should execute correctly. JavaSoft maintains a list of these ports at http://www.javasoft.com/cgi-bin/java-ports.cgi.

   Just follow the instructions that come with your Java distribution.

2. Download TauP.X.X.X.tar.gz or TauP.X.X.X.jar. Make sure to get the most recent version, replacing the X's in the file name. They can be found at

   ```
   http://www.seis.sc.edu
   ```

3. Unpack the distribution.
   ```
   gunzip TauP.X.X.X.tar.gz
   tar -xvf TauP.X.X.X.tar
   ```
   or
   ```
   jar -xvf TauP.X.X.X.jar
   ```
   This will create a directory called TauP. Inside will be the following:

README	getting started information
License	license information, free for non-commercial
taup.jar	the jar file with all the classes
taup.html	a simple web page that loads a rudimentary applet to use the TauP package. Be warned that this is not really meant to be used over the Internet as the download time for 1.5Mb may be too great and most browsers do not yet support the 1.1 version of Java.
exampleProperties	example properties file
HISTORY	change log
bin	a directory with wrapper scripts appropriate for UNIX installations
doc	a directory with Postscript and pdf versions of this manual.
html	a directory with the javadoc output from the source code, mainly useful for writing new java programs that use the TauP package.
jacl	a directory with example Jacl scripts for accessing the TauP package.
native	a directory with a C library and example program that use the Java Native Interface, providing a basic interface between C programs and the TauP package.
Maple	a directory with Maple scripts showing the time and distance equations used.

4. Put the taup.jar file someplace. It really doesn't matter where, although a central place might make administration easier, /usr/local/classes or /usr/local/lib are good choices. If you don't have superuser privileges then your home directory is fine.

   For java 2 (a.k.a java 1.2) you can install taup.jar as a *standard extension* by placing the jar file into the jre/lib/ext subdirectory of your java installation. If you do this then you do not need to add it to your CLASSPATH and can skip the next step.

5. Add the location of taup.jar to your CLASSPATH environment variable. This should be done in your .cshrc or .login. For instance, if you put taup.jar in /usr/local/classes, then you could set the CLASSPATH to be:

   ```
   setenv CLASSPATH /usr/local/classes/taup.jar:/usr/local/jdk1.1.6/lib/classes.zip
   ```

   This should be all one line, of course. The java virtual machine also uses the CLASSPATH environment variable to find its class files, so make sure this is set up correctly.

6. Put the wrapper scripts in a directory referenced by your PATH environment variable, /usr/local/bin for instance. These wrapper scripts are not essential, but they cut down on typing. They are in the bin directory of the distribution and are simple UNIX sh scripts. Of course, they will only work on UNIX.

7. Lastly, you may need to either source your .login and .cshrc files or execute the `rehash` command to make the shell reevaluate the contents of your PATH.

That's it. If you have problems or encounter bugs, please mail them to me. Please try to be as specific as possible. I am also interested in ideas for additional features that might make this a more useful program. Of course, I can make no promises, but I would be glad to hear about them.

I can be reached via email at *crotwell@seis.sc.edu*.

A.2 MacOS

Please read the Unix instructions, since much of it is applicable to all platforms.

For MacOS users, you can add taup.jar to your classpath by putting it in the MRJclasses folder which is located in the MRJ libraries folder within the Extensions folder within the System Folder. You may also wish to change the file type and creator to "ZIP " and "java" in order to fix the icon. Note that ZIP has a space at the end. This can be done with the DataViz FileViewer application that is, or at least was, bundled with the system. Also note that you must haveMacOS 8 or better in order to run Apple's java 1.1 compatible java, MRJ 2.1. Lastly, JBindary is the MacOS version of the java command. With it you can create double-clickable applications that accomplish the same thing as the sh scripts under unix. If you do not have MRJ installed, or would like to get a more recent version, it can be downloaded from Apple's web site, `http://www.apple.com/java`.

A.3 Windows

Please read the Unix instructions, since much of it is applicable to all platforms.

For Java 1.1, you need to add taup.jar to your CLASSPATH. This should likely be done in you autoexec.bat file for window 95 or 98 and in the Control Panel under windows NT, start the Control Panel, select System, then click the Environment tab. The other difference with UNIX is that the separator is a semicolon and the forward slashes should be replaced with backslashes.

For Java 1.2, aka Java 2, you can skip the CLASSPATH step by putting the taup.jar file into the `jre\lib\ext` directory of your java installation.

In either case you will likely want to add the bat files that start up the tools to your PATH. You can either put them in and existing directory referenced in you PATH, or add the distribution bat directory to the PATH environment

variable. I have put the bat files in the bat directory to keep them separate from the UNIX sh scripts, and so you may wish to delete bin and rename bat to bin.

B Troubleshooting

There are a few idiosyncrocies about the codes and Java in general that you may run into.

1. Out of memory errors. By default Java sets its maximum memory to be 16 megabytes. For most uses this is sufficient, but some very complicated models using a large number of phases may exceed this limit. A simple fix is to change the maximum memory to be a larger amount. The -mx command line argument to the java command does this. So, to set the maximum amount of memory to 32 megabytes you could say `java -mx32m edu.sc.seis.TauP.TauP_Path`. For convenience you may wish to make this change more permanent by adding it to the scripts, i.e. taup_time, etc.

2. Garbled jar files. Care should be taken with the jar files when transferring them from one operating system to another. Certain file transfer utilities make an attempt to *fix* text files by changing `RETURN LINEFEED` sequences to just `LINEFEED` or just `RETURN` or vice versa. This is useful for real text files, but dangerous for jar files. I have noticed this when transferring files between UNIX and Macintosh, and it likely can happen between any two operating systems with differing end of line identifiers. Using binary mode for ftp transactions is likely wise.

3. Trouble with applet and appletviewer. There have been problems with appletviewer and the applet related to the CLASSPATH. For the applet to work, taup.jar needs to be in the CLASSPATH. However, other files in the CLASSPATH may confuse either the applet when it looks for models, or cause the appletviewer to not run at all with any applets. If you experience any problems with TauPApplet or appletviewer in general, try running with a simplified CLASSPATH. For example, on UNIX machines in the csh:

   ```
   setenv CLASSPATH /path/to/the/jar/taup.jar
   appletviewer taup.html
   ```

4. Trouble with bat files. I don't use windows and so I do not know if the bat files are really useful or not. If you find a better method, I would be happy to include it.

References

Buland, R. and C. H. Chapman (1983). The Computation of Seismic Travel Times, *Bull. Seism. Soc. Am.* **73**(5), 1271–1302.

Crotwell, H. P., T. J. Owens, and J. Ritsema (1998). The TauP ToolKit: Flexible Seismic Travel-Time and Raypath Utilities, *Seismological Research Letters*. In Preperation.

Davis, J. P. and I. H. Henson (1993a). Development of an X-Windows tool to compute Gaussian bean synthetic seismograms. Technical Report TGAL-93-03, Phillip Laboratory, Hancom AFB, MA.

Davis, J. P. and I. H. Henson (1993b). *User's Guide to Xgbm: An X-Windows System to compute Gaussian bean synthetic seismograms* (1.1 ed.). Alexandria, VA: Teledyne Geotech Alexandria Laboratories.

Dziewonski, A. M. and D. L. Anderson (1984). Structure, elastic and rheological properties and density of the earth's interior, gravity and pressure. In K. Fuch and H. Soffel (Eds.), *Landoldt-Bönstein, Group V*, Volume 2a, pp. 84–96. Berlin: Springer.

Gilbert, F. and A. M. Dziewonski (1975). An application of normal mode theory to the retrieval of structural parameters and source mechanisms from seismic spectra, *Philisophical Transactions of the Royal Society, London A* **278**, 187–269.

Herrin, E. (1968). 1968 seismological tables for P phases, *Bull. Seism. Soc. Am.* **58**(4), 1193–1241.

Jeffreys, H. and K. E. Bullen (1940). *Seismological Tables*. London: British Association for the Advancement of Science, Burlington House.

Kennett, B. L. N. and E. R. Engdahl (1991). Traveltimes for global earthquake location and phase identification, *Geophysical Journal International* **105**, 429–465.

Kennett, B. L. N., E. R. Engdahl, and R. Buland (1995). Constraints on seismic velocities in the Earth from traveltimes, *Geophysical Journal International* **122**, 108–124.

Morelli, A. and A. M. Dziewonski (1993). Body wave traveltimes and a spherically symmetric P- and S-wave velocity model, *Geophysics Journal International* **112**(2), 178–194.

Tull, J. E. (1989). *SAC - Seismic Analysis Code: User's Manual* (Revision 2 ed.). Livermore, CA: Lawrence Livermore National Laboratory.

Weber, M. and J. P. Davis (1990). Evidence of a laterally variable lower mantle structure from P- and S-waves, *Geophysics Journal International* **102**(1), 231–255.

Wessel, P. and W. H. F. Smith (1995). New Version of the Generic Mapping Tools released, *Eos* **76**, 329.