SANDIA REPORT
SAND2023-05863
Printed June 2023

Sandia National Laboratories

# GeoTess User's Manual

Sanford Ballard, James R. Hipp, Brian Kraus, Andrea Conley, Patrick Hammond

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.



Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods/

## ABSTRACT

GeoTess is a model parameterization for multi-dimensional Earth models and an extendable software system that implements the construction, population, storage, and interrogation of data stored in the model. A constructed GeoTess model is comprised of 2D triangular tessellations of a unit sphere with 1D radial arrays of nodes associated with each vertex of the 2D tessellations. Variable spatial resolution in both geographic and radial dimensions is supported. Users have considerable flexibility in how to define the data stored on the grid. The Java version of the GeoTess software can be downloaded along with the tools PCalc and LocOO3D, which use GeoTess models to do traveltime prediction and event location, respectively, at:

https://github.com/sandialabs/Salsa3DSoftware

or it can be downloaded individually at:

https://github.com/sandialabs/GeoTessJava

A C++ version is also available at:

https://github.com/sandialabs/GeoTessCPP

The software is packaged with this user's manual and a set of example scripts in all cases.

This page left blank.

## CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# ACRONYMS AND DEFINITIONS

| Abbreviation | Definition |
|---|---|
| 1D | One dimensional |
| 2D | Two dimensional |
| 3D | Three dimensional |
| I/O | Input/Output |
| IERS | International Earth Rotation and Reference Systems Service Ellipsoid |
| IERS_RCONST | International Earth Rotation and Reference Systems Service Ellipsoid, Constant Radius |
| GRS80 | Geodetic Reference System 1980 Ellipsoid |
| GRS80_RCONST | Geodetic Reference System 1980 Ellipsoid, Constant Radius |
| WGS84 | World Geodetic System 1984 Ellipsoid |
| WGS84_RCONST | World Geodetic System 1984 Ellipsoid, Constant Radius |

# 1. INTRODUCTION

GeoTess is a model parameterization for multi-dimensional Earth models and an extendable software system that implements the construction, population, storage, and interrogation of data stored in the model. GeoTess is not limited to any particular type of data. To GeoTess, the data are just 1D arrays of values associated with each point in the grid.

GeoTess is distributed through GitHub in one of three ways. First, the user can download GeoTess in Java as part of the Salsa3DSoftware package at https://github.com/sandialabs/Salsa3DSoftware. This package includes LocOO3D (used to perform event locations using GeoTess velocity models) and PCalc (used for raytracing and travel-time computation using GeoTess velocity models) in addition to GeoTess. Second, the user can download GeoTess in Java individually at https://github.com/sandialabs/GeoTessJava. Finally, a C++ version of GeoTess can be downloaded at https://github.com/sandialabs/GeoTessCPP; this version includes a specialized extension GeoTessAmplitude that allows GeoTess to create and manipulate amplitude, i.e., attenuation, models.

In all cases, GeoTess is packaged with the latest version of this user's manual and two sets of examples, one for the GeoTessBuilder application (Section 5) and one for general usages of GeoTess. To compile the Java version of GeoTess from the source code (either as part of the Salsa3DSoftware package or individually), the user will need to have the Maven software package (https://maven.apache.org/index.html) and Java ver. ≥ 10 installed. To compile the C++ version of GeoTess from the source code, a gcc compiler needs to be installed.

Applications can access GeoTess as a library. In this mode of interaction, applications can perform the following tasks:

- Read grids and models from, and write them to, files in ASCII and binary formats.
- Query a model grid for information about the nodes, cells, or tessellations.
- Associate data structures with the nodes of the geometry.
- Query the model for the data associated with a specified node.
- Modify the data associated with a node.
- Find arbitrary positions within the grid hierarchy (point searching).
- Retrieve the interpolation coefficients at arbitrary locations in space. GeoTess currently implements linear and natural neighbor interpolation algorithms.
- Interpolate data values at arbitrary positions using the interpolation coefficients described above.
- Given a sequence of points that defines a ray path through a model, retrieve the weights (data kernels) associated with the grid nodes in the model that were influenced by the ray path.

These functions are described more fully in Section 4. Complete interface documentation for every publicly accessible function in the library is provided for each language in html format (C++/C interface, Java).

The GeoTess library is available in Java and C++ with a C interface to the C++ library (see Appendix C). In the case of the C++ version of the software, source code, precompiled binaries for Linux, MacOS and Windows operating systems, and Makefiles are provided. The provided Makefiles can also compile on SunOS, but note that this operating system is no longer supported.

In addition to the standard GeoTess package, two applications are provided:

- *GeoTessExplorer* – provides functions to extract, compare, and modify GeoTess model data
- *GeoTessBuilder* – provides capability to construct variable resolution 2D triangular tessellations.

These applications are described more fully in Section 5 (GeoTessBuilder) and Section 6 (GeoTessExplorer).

In the following sections, installation instructions are provided, followed by an introduction to relevant GeoTess concepts, and a tutorial of some relevant GeoTess functionalities. Note that the tutorial is intended to get the user familiar with GeoTess; for brevity it does not cover the full range of capabilities.

# 2. INSTALLATION

There are three ways to install GeoTess depending on the language desired and whether related software packages LocOO3D (used for event location) and PCalc (used for traveltime prediction) are also desired. These three installations are described below.

## 2.1. Install GeoTess (Java) as Part of Salsa3DSoftware Package

To install the Java version of GeoTess as part of the Salsa3Dsoftware package (including LocOO3D and PCalc) clone the Salsa3DSoftware package located at https://github.com/sandialabs/Salsa3DSoftware to the desired location. This will create a Salsa3dSoftware directory. Once the directory is cloned, run the following steps:

1. cd into the Salsa3dSoftware directory and build the executable jar file using Maven by running:

   **mvn clean package**

   This command creates the jar file **salsa3d-software-1.2023.4-jar-with-dependencies.jar** in the target folder (~/Salsa3DSoftware/target). Note that this jar file contains the classes for all three Salsa3DSoftware packages (LocOO3D, GeoTess, PCalc).
2. Run the **configure.sh** script located in ~/Salsa3DSoftware. This script creates executable shell scripts that can be used to access and run the desired tool contained within the jar file. In this case, the necessary executables are the **geotess** and **geotessbuilder** shell scripts.
3. The configure.sh script also outputs a recommendation on how to update a **.bash_profile** or similar shell configuration file to include the generated executables on the user's path. For example:

   Add this line to your .bash_profile
   export PATH=/Users/username/Salsa3DSoftware:$PATH

   It is highly recommended the user follow this recommendation for ease of use.

## 2.2. Install GeoTess (Java) Individually

To install the Java version of GeoTess individually, clone the GeoTess package located at https://github.com/sandialabs/GeoTessJava to the desired location.

1. (Optional) cd into the GeoTessJava directory and build the executable jar file using Maven by running:

   **mvn clean package**

   This command creates the jar file **geo-tess-java-2.6.18-SNAPSHOT-jar-with-dependencies.jar** in the target folder (~/GeoTessJava/target). Note that a precompiled jar file is already available in the target folder. Thus, compiling a new jar file from source code is optional.

2. Run the **configure.sh** script located in ~/GeoTessJava. This script creates executable shell scripts that can be used to access and run the desired tool contained within the jar file. For GeoTess, these executables are the **geotess** and **geotessbuilder** shell scripts.
3. The configure.sh script also outputs a recommendation on how to update a **.bash_profile** or similar shell configuration file to include the generated executables on the user's path. For example:

   Add this line to your .bash_profile
   export PATH=/Users/username/GeoTessJava:$PATH'

   It is highly recommended the user follow this recommendation for ease of use.

## 2.3. Install GeoTess (C++)

To install the C++ version of GeoTess individually, clone the GeoTess package located at https://github.com/sandialabs/GeoTessCPP to the desired location. This install will also come with a C shell interface (under directories GeoTessCShell), an application to apply GeoTess to amplitudes called GeoTessAmplitude (see Section 4.5), and LibCorr3D (Ballard et al., 2009; see Section 4.5). Note that the C shell interface cannot be used for LibCorr3D.

The included Makefiles can run on Linux, MacOS, and Windows.

1. cd into the GeoTessCPP directory and compile by running:

   **make all**

   This command will cause the main Makefile in ~/GeoTessCPP to call the Makefiles under each subdirectory to build the entire project. Note that the Makefile will automatically determine the OS environment to build on, so the user does not need to specify it.
2. The Makefile also outputs a recommendation on how to update a **.bash_profile** or similar shell configuration file to include the generated executables on the user's path. For example, on a Mac, the message would be:

   recommended environment variables for Darwin:

   export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:/Users/username/GeoTessCPP/lib

   It is highly recommended the user follow this recommendation for ease of use.

3. If needed, the Makefile can be used to remove object files by running:

   **make clean_objs**

The Makefile can also be used to remove all object and .exe/.dll/.so/.dylib files. This target should return the module directory to the same state it was in before a call to "make all" was executed. To perform this removal, run:

**make clean**

4. Example output from running the Makefile on a Mac is as follows under each subdirectory:
    a. ~/GeoTessCPP/lib
        i. libgeotessamplitudecpp.dylib
        ii. libgeotessamplitudecshell.dylib
        iii. libgeotesscpp.dylip
        iv. libgeotesscshell.dylib
        v. liblibcorr3d.dylib
    b. ~/GeoTessCPP/GeoTessCPPExamples/bin
        i. The following are all binary executables compiled from source code in ~/GeoTessCPP/GeoTessCPPExamples/src.
            1. extendedmodel
            2. interrogatelibcorr3d
            3. interrogatemodel
            4. populatemodel2d
            5. populatemodel3d
            6. tomography2d
    c. ~GeoTessCPP/GeoTessCExamples/bin
        i. The following are all binary executables compiled from source code in ~/GeoTessCPP/GeoTessCExamples/src.
            1. integrate3d
            2. populatemodel2d
            3. populatemodel3d
            4. testcrust20
            5. tomography2d

5. To create executables such as those in the Examples directories, the user must create a properties file, compile it into an executable, then link it to any libraries or executables that should be included.

    For example, on a Mac the commands below create the extendedmodel executable (bullet 4b in this list) based on the ExtendedModel.cc and GeoTessModelExtended.cc source codes in ~/GeoTessCPP/GeoTessCPPExamples/src.

    **g++ -m64 -O3 -Iinclude -I../GeoTessCPP/include -I../LibCorr3D/include -c src/GeoTessModelExtended.cc**

    **g++ -m64 -O3 -Iinclude -I../GeoTessCPP/include -I../LibCorr3D/include -c src/ExtendedModel.cc**

    **g++ -m64 -o bin/extendedmodel ExtendedModel.o GeoTessModelExtended.o -L../lib -lgeotesscpp -llibcorr3d -lstdc++ -lm**

Linking the executables for ExtendedModel.cc and GeoTessModelExtended.cc allows for the implementation of an extension to the GeoTessModel class (Section 4.5).

# 3. GEOTESS MODEL COMPONENTS AND CONCEPTS

While many Earth models use regular latitude longitude grids to describe the geographic geometry and topology, GeoTess uses a triangular tessellation (see Appendix A for a geometrical description). These two approaches are compared in Figure 3-1. While software algorithms that use regular latitude-longitude grids are much more straightforward to develop, the grids suffer from severe unintended variability in cell areas, with cell areas approaching zero near the poles. Software for triangular tessellations are more complicated to develop but they result in grids with much more uniform cell size and approximately 25% fewer vertices.

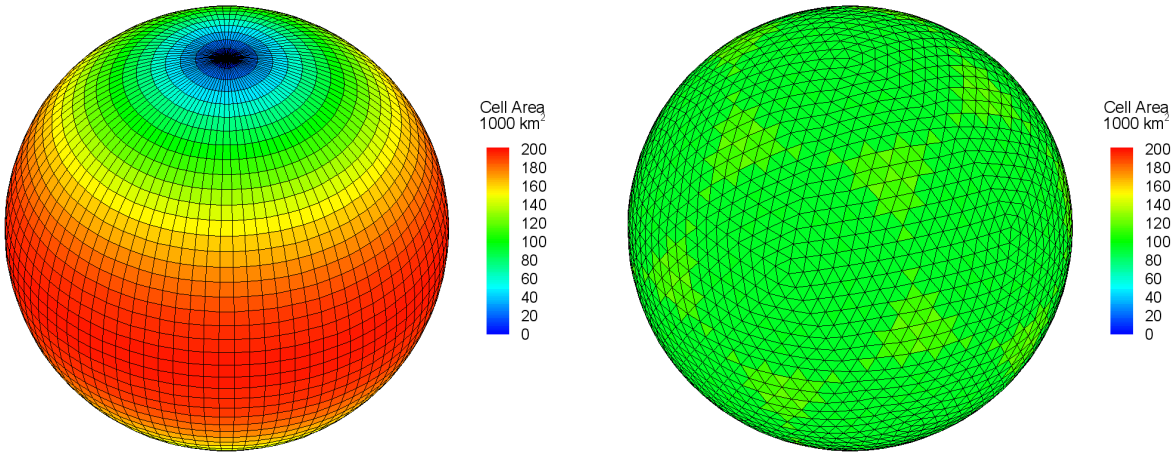


**Figure 3-1. Comparison of a regular latitude longitude grid and a uniform triangular tessellation. In both grids the edge lengths are approximately 4°.**

A GeoTess model is comprised of the following elements:

- A set of layers (Figure 3-2). Each layer spans the entire 2D geographic extent of the model. The boundaries at the top and bottom of a layer may have topography. Within each layer, model data values are continuous, both geographically and radially. Model data values may be discontinuous across layer boundaries. Layers may have zero thickness at some or all geographic locations. An important limitation of the parameterization used by GeoTess is that layer boundaries may not fold back on themselves, i.e., any radial line emanating from the center of the Earth must intersect each layer boundary exactly one time.

- A set of multi-level tessellations (Figure 3-3). Each layer will be associated with one multi-level tessellation but many layers may be associated with each multi-level tessellation, i.e., there is a many-to-one relationship between layers and multi-level tessellations. By associating layers that are deep in the Earth with low resolution multi-level tessellations and layers at shallower levels in the Earth with higher resolution multi-level tessellations, the resolution of the model can be varied radially as necessary to achieve more appropriate sampling.

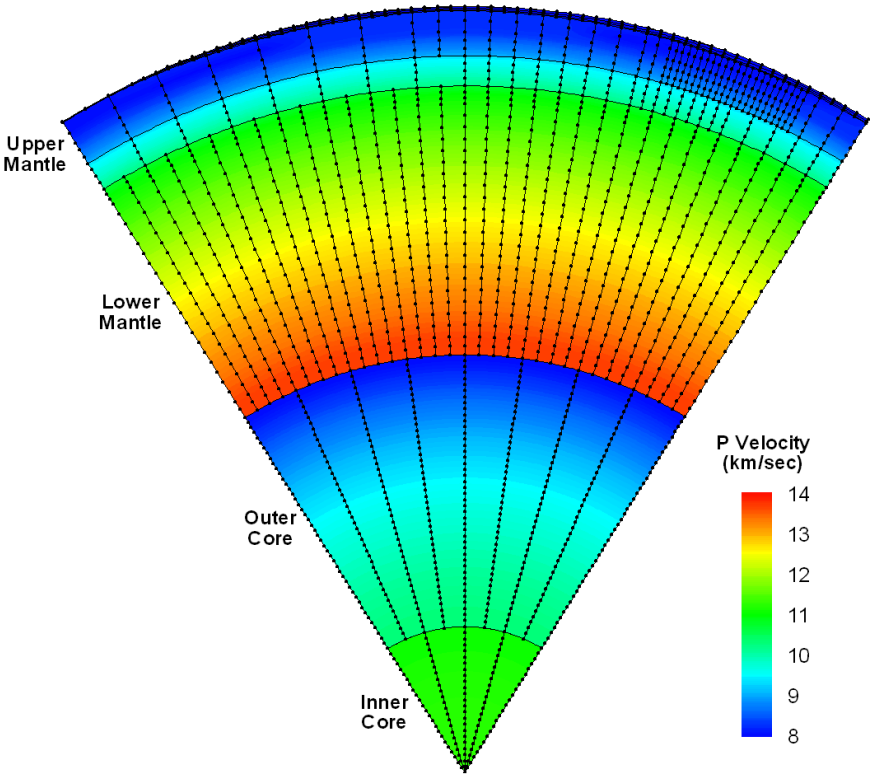


**Figure 3-2. A slice through a portion of a global 3D P velocity model. The model consists of several layers, such as the Inner Core, Outer Core, etc. Each layer is associated with a separate multi-level tessellation, providing variable resolution in the radial direction. Profiles are defined as a set of nodes, all within a single layer of the model, positioned along a line with constant geographic position. Note the variable resolution in the geographic dimensions in the upper mantle.**

- The topology of each multi-level tessellation will consist of a set of levels (see Figure 3-3), with each level consisting of a set of triangles that spans the surface of a unit sphere, without gaps or overlaps. The triangles on a given tessellation level are obtained by subdivision of the triangles on the previous tessellation level, with the first tessellation level being an icosahedron. Each multi-level tessellation may have variable resolution in the geographic dimensions (i.e. the triangles can be subdivided into smaller triangles arbitrarily). Note the variable resolution of the final tessellation level in the bottom right panel.

- The geometry of each multi-level tessellation will consist of a set of vertices that defines the positions of the corners of the triangles. If a model is comprised of more than one multi-level tessellation, they will share common vertices, to the extent possible.

- Data arrays. Each data array is a 1D array of data values that may be of type double, float, long, int, short or byte. All the data arrays in the model must be of the same type and must have the same number of elements.

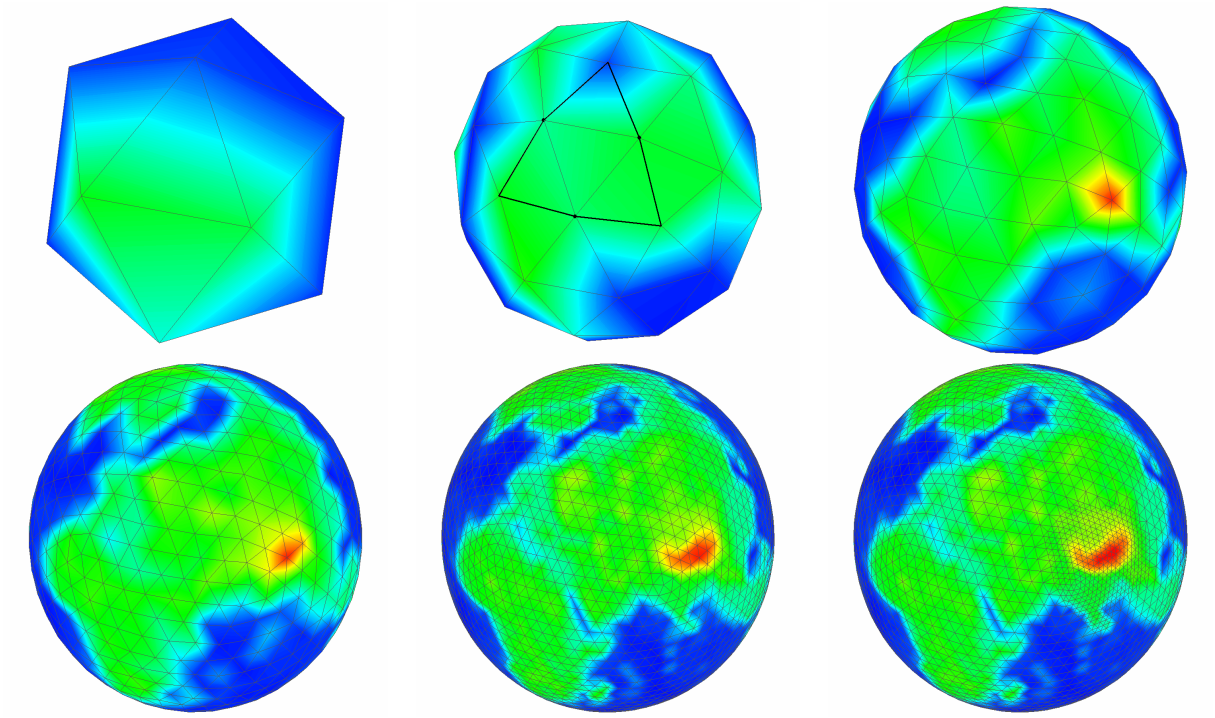

**Figure 3-3. Construction of a multi-level tessellation by iterative subdivision of triangles. Each image represents one level and together the levels comprise a single multi-level tessellation.**

- Profiles. Each profile is composed of a set of monotonically increasing radii and a set of data arrays. Each profile is associated with a single vertex and a single layer in the model. The first and last radii in a profile define the bottom and top of the associated layer at the geographic position of the vertex. Several types of profiles are supported:
    - N-Point profiles consist of two or more radii and an equal number of data arrays, with one data array associated with each radius.
    - Constant value profiles consist of two radii and a single data array that defines the data values for the entire radial span of the profile.
    - Thin profiles consist of a single radius and a single data array. They have zero thickness, i.e., the radius of the bottom and top of the profile are equal.
    - Empty profiles consist of two radii but no data arrays.
    - Surface profiles consist of only a single data array. They have no radius values. Together with Empty Surface profiles, these are used to support 2D models. Surface profiles are incompatible with all other profile types. If a model contains any surface profiles, it cannot contain any profiles of any other type.
    - Empty Surface profiles consist of no radii and no data.

The data values within a profile are continuous.

- A 2D array of Profiles with *nVertices* × *nLayers* elements. The first index refers to one of the vertices of the model geometry and the second index refers to one of the layers of the model. For a given vertex index, the 1D array of profiles contains a profile for each layer of the model, stored in order of increasing radius. The last radius of each profile in a 1D profile array must be equal to the first radius of the next profile in the same 1D profile array. While the data values within a single profile are continuous, data values may be discontinuous across profile (i.e. layer) boundaries.

- Radial interpolators that interpolate data values within an individual profile. These include linear interpolators, cubic spline interpolators, and potentially others.

- 2D interpolators that interpolate values in the 2 geographic dimensions. These include linear interpolators that interpolate values within a single triangle of the 2D tessellations, and higher order interpolators that provide continuous spatial derivatives of the data values.

- (2+1)D interpolators that combine 1D and 2D interpolators to interpolate data in 3D. They first use a 1D interpolator to interpolate values at a specified radius in a neighborhood of profile arrays, and then apply a 2D interpolator to those values to find an interpolated value at the desired 3D location.

Referencing Data objects is complicated by the fact that a particular Data object has a node index within a Profile which in turn is associated with a vertex and a layer. The following definitions are relevant:

- *Vertex* – refers to a point in a 2D triangular tessellation where multiple triangles intersect. Each vertex is represented internally by a unit vector whose origin resides at the center of the earth, x-component points to lat , lon = 0°N, 0°E, y-component points to lat, lon = 0°N, 90°E, and z component points to the north pole. A vertex has no information about radial position in the model. Functions are provided in GeoTessUtils to convert back and forth between unit vectors and geographic latitude and longitude.

- *Node* - refers to a Data object associated with a Profile. Nodes within a Profile are stored in order of increasing radius. Every Profile has a node with index 0 which is the node with the smallest radius.

- *Point* -To facilitate indexing the Data objects in a model, the term *point* is introduced. A *point* is conceptually a triplet of indexes including the vertex index, the layer index and node index. Applications can refer to Data objects in a model either by their pointIndex or by the combination of vertexIndex, layerIndex and nodeIndex. Each model maintains a PointMap object to manage this capability.

## 3.1. Polygons

GeoTess makes use of polygons for several purposes. GeoTessBuilder (Section 5), an application used to create GeoTess grids, uses polygons to define grid resolution and the class GeoTessModel can use polygons to select a subset of all Points in a GeoTessModel for inclusion in a PointMap (see

Section 4 for an example). To directly construct and make use of polygons in Java and C++ applications, consult the html documentation for those languages (C++, Java). This section describes polygon file formats.

GeoTess uses two kinds of polygons, 2D polygons and 3D polygons. A 2D polygon consists of an ordered list of geographic positions that define a closed loop on the surface of a unit sphere. A 3D polygon is like a 2D polygon with regard to the geographic dimensions but adds a 'top' and a 'bottom' in the radial direction. The top and bottom are 2D surfaces of constant depth, constant radius, or constant fractional position within a layer.

A very convenient way to generate a 2D polygon is to use Google Earth (http://www.google.com/earth). It provides a tool to define and edit 2D polygons by clicking on an image of the Earth. The polygon can then be saved in either ASCII (kml) or binary (kmz) formats. Kml/kmz files have three limitations with respect to GeoTess applications: 1) They are only accessible via the Java version of GeoTess; the C++ and C versions cannot read these formats, 2) 3D polygons cannot be stored in kml/kmz files because they have no ability to store the 2D surfaces that define the top and bottom of the 3D polygons, 3) with polygons stored in kml/kmz files, it is not possible to record which 'side' of the polygon is 'inside' and which is 'outside'. GeoTessExplorer (Section 6), an application used to query and manipulate GeoTess models, has a utility function called *translatePolygon* to translate polygons back and forth between ASCII and kml/kmz formats.

When generating polygons, it is important to note that polygon edges are great circle paths and are interpreted as taking the shortest path between adjacent points. When manually entering points into an ASCII file, ensure that adjacent points are less than 180 degrees apart. For example, if two points at 0N, 100W and 0N, 100E are specified, the great circle path connecting those two points will pass through 0N, 180E, not 0N, 0E as might be expected.

ASCII files are parsed as follows:

1. Records that start with '#' are comment lines and are ignored.

2. If there is a record that starts with *'lat'* then all boundary point records will be assumed to be in order *lat-lon*. If there is a record that starts with *'lon'* then all boundary point records will be assumed to be in order *lon-lat*. If no record starts with *'lat'* or *'lon'*, boundary point records are assumed to be in order *lat-lon*.

3. If there is a record that starts with *'reference'* then the record is assumed to contain information about the *referencePoint* which is used to determine which 'side' of the polygon is 'inside' and which is 'outside'. The second and third tokens in the record are interpreted as the latitude and longitude of the *referencePoint*, in degrees (the order depends on the *lat-lon* record described above). If the fourth and final token starts with *'in'* then the reference point is *'inside'* the polygon, otherwise it is '*outside'* the polygon.

4. For kmz/kml files and for ASCII files which do not specify a *referencePoint* as described above, the reference point will be the antipode of the normalized vector sum of the polygon boundary points and will be deemed to be *'outside'* the polygon.

5. All other records are assumed to specify a boundary point in *lat-lon* or *lon-lat* order, in degrees. If a record is encountered that cannot be parsed as two floating-point values, the record is simply ignored without issuing any error or warning messages.

6. It is not necessary to ensure that the polygon is 'closed'. If the first and last points of the polygon definition are not identical, the polygon will be closed automatically.

7. If the first record of an ASCII file is the string 'POLYGON3D' then the file defines a 3D polygon, otherwise it defines a 2D polygon. If the file defines a 3D polygon, then it must also contain two records which define the top and bottom surfaces of the polygon. Each of these records must consist of 4 tokens as follows:

   [ *top* | *bottom* ], [ *radius* | *depth* | *layer* ], Z, *layerIndex*

   a. The first token specifies whether the top or bottom surface is being defined. The file must contain one record that starts with '*top*' and one record that starts with '*bottom*'.

   b. The second token specifies how the surface is to be specified. The surface can be defined as a constant radius or a constant depth or as a layer. When specified as a layer, the surface is everywhere constrained to reside in the layer specified by *layerIndex*, which must correspond to a valid layer.

   c. The third token, Z, specifies the radius/depth value in km. If the second token in bullet b is set to *layer*, Z specifies a fractional position within the layer. If $Z \leq 0$, then the surface will track the bottom of the specified layer. If $Z \geq 1$, the surface will track the top of the layer. For intermediate values, the surface will track the corresponding fractional position within the layer.

   d. The final token, *layerIndex*, specifies whether the surface is constrained to a particular layer. The second token in bullet b must be set to *layer* for *layerIndex* to have an effect. If *layerIndex* is negative, then the surface is not constrained to any particular layer and it will be equal to the specified radius/depth, no matter what layer that radius/depth corresponds to. If *layerIndex* specifies a valid layer index, then the surface will track the specified radius/depth value so long as the radius/depth resides in the specified layer. If the specified radius/depth is above the top of the specified layer, then the surface will track the top of the layer. If the radius is below the bottom of the layer, then the surface will track the bottom of the layer.

The following is an example of the contents of a 2D polygon file:

```
Polygon2D
Reference 5 15 inside
lon-lat
0 0
20 0
20 30
0 30
0 0
```

This 2D polygon will consist of a simple box from 0N, 0E at the bottom left corner, extending to 20N, 30E at the upper left corner. The reference point is specified to be located at 5N, 15E and is 'inside' the polygon. Because the first 3 lines all specify default behavior, an equivalent specification for this polygon would have been simply:

```
0 0
20 0
20 30
0 30
```

Here is an example of the contents of a 3D polygon file:

```
#This file defines a 3D polygon
Polygon3D
TOP Layer 1.000     6
BOTTOM Depth 4000.000 -1
Reference 5 15 outside
lon-lat
0 0
20 0
20 30
0 30
```

The top of the polygon is defined by a surface that conforms to the top of layer 6. The bottom surface is defined by a constant depth at 4000 km below the surface of the ellipsoid and is unconstrained to conform to any particular layer (it may reside in different layers at different geographic locations). The boundary points are specified in longitude, latitude order so the box extends from 0N 0E in the lower left to 30N, 20E in the upper right. The reference point is located at 15N, 5E and is 'outside' the polygon.

## 3.2. Ellipsoids

GeoTess is only modestly dependent on the ellipsoids often used to define the shape of the Earth. In general, ellipsoids are important for two purposes: 1) they are used to convert between geographic and geocentric latitude and 2) they are used to convert between radius, measured from the center of the Earth to depth measured from the surface of a specified ellipsoid. GeoTess is inherently not dependent on either of these factors because 1) the geographic locations of all vertices are stored and manipulated as Earth-centered unit vectors, which are independent of ellipsoid (see Appendix A), and 2) the radial positions of all grid nodes are specified, stored, and manipulated as radii measured in km from the center of the Earth.

That said, model developers may have used a particular Earth ellipsoid when they converted depth information to radii for purposes of model population and they may wish to store within the model the name of the ellipsoid that was used to populate that model. So, starting with GeoTess version 2.2.0 the binary and ASCII model file formats were modified to include storage of the name of the Earth ellipsoid associated with the model. There are functionalities provided in the GeoTess

software to manipulate geographic information and to retrieve the radius of the ellipsoid at a specified latitude. See function *GeoTessModel.getEarthShape*() in the Java and/or C++ documentation for more information. Appendix A describes how GeoTess manages geographic information and Earth ellipsoids.

The Earth ellipsoids supported by GeoTess include:

**Table 3-1. Ellipsoids Supported by GeoTess.**

| Ellipsoid | Inverse flattening parameter | Equatorial Radius (km) |
|---|---|---|
| SPHERE | ∞ | 6371.000 |
| GRS80 | 298.257222101 | 6378.137 |
| GRS80_RCONST | 298.257222101 | 6371.000 |
| WGS84 | 298.257223563 | 6378.137 |
| WGS84_RCONST | 298.257223563 | 6371.000 |
| IERS | 298.25642 | 6378.1366 |
| IERS_RCONST | 298.25642 | 6371.000 |

Ellipsoid SPHERE treats the Earth as a sphere and hence does not convert between geocentric and geographic latitudes. All other ellipsoids use the specified inverse flattening parameter to convert between geocentric and geographic latitudes. SPHERE and all ellipsoids that end in 'RCONST' assume, for purposes of converting between depth and radius, that the radius of the earth is a constant equal to 6371 km. Ellipsoids GRS80, WGS84, and IERS assume that the radius decreases from equator to poles according to the specified parameters.

# 4. LIBRARY INTERACTIONS

In this section, general information is provided about how to accomplish some of the most important functions implemented by the GeoTess library. Not all functions are described here. Complete interface documentation for every publicly accessible function in the library is provided for the C++ version and the Java version.

In addition to the documentation, there are example programs provided that illustrate how to implement basic functions in both the C++ and Java versions of the code. In the C++ version, the precompiled examples can be found in **~/GeoTessCPP/GeoTessCPPExamples**. Additional examples using the C interface can be found in **~/GeoTessCPP/GeoTessCExamples**. In Java, the examples can be found in **~/GeoTessJava/src/main/java/gov/sandia/geotess/examples** or in **~/Salsa3DSoftware/src/main/java/gov/sandia/geotess/examples**, depending on whether GeoTessJava was installed individually or as part of the Salsa3DSoftware package.

## 4.1. Model population

This section describes how to generate and populate a new GeoTessModel with data in three steps. These steps are applicable to both GeoTess Java and GeoTess C++. Specific examples detailing model population (PopulateModel2D, PopulateModel3D) are provided in both languages in the examples folders described in Section 4. In addition, the example PopulateModel3D is included in Appendix A in both Java and C++.

### *4.1.1. Step 1 – Specify MetaData*

Implement a *GeoTessMetaData* object and populate it with the following required general information about the model:

1) *Description* – a description of the model. GeoTess does not process this information in any way; it simply stores it in the model and returns it on request. Users can put whatever they want in here.

2) *Layer names* – a list of the names of the layers that comprise the model, listed in order of increasing radius. An example might be "core, mantle, crust".

3) *data type* – the type of the data stored in the model. Options are double, float, long, int, short, or byte. All the data stored in a model must be of the same type.

4) *attribute names* – a list of the names of the data attributes stored in the model. An example might be "pvelocity; svelocity; density". In the example, there would be a 3-element array of data values associated with each grid node in the model.

5) *attribute units* – a list of the units of each attribute. If the attribute names were "pvelocity; svelocity; density", then the attribute units might be "km/sec; km/sec; g/cc". If one of the attributes is a unitless quantity, the corresponding attribute unit would be blank, e.g., "km/sec; ; g/cc". The number of units must equal the number of attribute names.

6) *model-population software* - the name and version number of the application used to generate the model. GeoTess does not process this information in any way; it simply stores it in the model and returns it on request.

7) *model generation date* - GeoTess does not process this information in any way; it simply stores it in the model and returns it on request.

8) *LayerTessIds* – a list of tessellation indexes, with one element for each layer of the model, establishing a map from layer index to a tessellation index. Consider the model in Figure 3-2 as an example. Ignoring the crust, the model has 5 layers (inner core, outer core, lower mantle, transition zone and upper mantle). The deeper layers have many fewer profiles than the shallower layers, imparting variable resolution in the radial dimension to the overall model. To accomplish this, the GeoTessGrid manages 5 distinct multi-level tessellations, one for each layer. The *LayerTessIds* in this case would be the 5-element array "*0, 1, 2, 3, 4*" specifying that the first layer is associated with the first multi-level tessellation, etc. For a different model that consisted of 3 layers where all the layers could reference a single multi-level tessellation, *LayerTessIds* should be specified as "*0, 0, 0*".

### 4.1.2. Step 2 – Construct a Model

Construct a GeoTessModel object, specifying the GeoTessMetaData object instantiated in Step 1 and the name of a file containing a GeoTessGrid object. Files containing standard GeoTessGrid objects are available on the GeoTess website or custom GeoTessGrids can be constructed using the GeoTessBuilder application described later in this document. For this discussion, it is assumed that the desired GeoTessGrid exists in an accessible file.

After instantiating the GeoTessModel, the model will have instantiated a 2D array of Profile objects, which are all null, meaning that the model contains no data.

### 4.1.3. Step 3 – Add Data

Loop over every layer of the model. For the current layer, request the set of connected vertices for that layer. This is accomplished by calling the method *model.getConnectedVertices(layer)*.
Loop over every vertex of the grid, including those in the set of connected vertices as well as those that are not. The vertices are numbered from 0 to *nVertices*-1. Functions are provided in GeoTessGrid to retrieve the geographic location of the vertex, either as a unit vector or as a latitude, longitude pair.

For the current layer and vertex, construct a Profile object. If the current vertex id is not a member of the set of vertices connected together in the current layer, then construct a Profile of type ProfileEmpty. Otherwise, construct a Profile of one of the types defined below.

A Profile is basically a 1D array of nodes deployed along a radial line that spans a single layer at a single vertex. Referring to Figure 3-2, each small black dot is a node. Each array of black dots that spans a single layer is a Profile. A node is comprised of a radius and a Data object, which is a 1D array of data values, one data value for each attribute specified in the GeoTessMetaData definition. To instantiate a Profile object, you supply an array of monotonically increasing radius values, and an array of Data objects. Each Data object is itself an array of attribute values (e.g., pvelocity, svelocity, etc.). See the sample code for an example of how to do this.

There are 5 types of Profile objects:

1) N-Point profiles consist of two or more radii and an equal number of data arrays, with one data array associated with each radius.

2) Constant value profiles consist of two radii and a single data array that defines the data values for the entire radial span of the profile.

3) Thin profiles consist of a single radius and a single data array. They have zero thickness, i.e., the radius of the bottom and top of the profile are equal.

4) Empty profiles consist of two radii but no data arrays.

5) Surface profiles consist of only a single data array. They have no radius values. Together with Empty Surface profiles, these are used to support 2D models. Surface profiles are incompatible with all other profile types. If a model contains any surface profiles, it cannot contain any profiles of any other type.

6) Empty Surface profiles consist of no radii and no data. They are essentially place holders for null. An Empty Surface profile object will return NaN in response to any request for information about radius or data information.

After construction, a Profile, *p*, is added to the model by calling *model.setProfile(i, j, p)*, where *i* is the index of a vertex, *j* is the index of a layer, and *p* is the just-constructed Profile.
After Profiles have been specified for all layers of all vertices of the model, the model is complete and ready for use.

## 4.2. Model I/O

GeoTessModels and GeoTessGrids can be written to and read from files, either in ASCII or binary format. Complete format definitions are supplied in Appendix D.

The simplest way to save a model to a file is to call the *model.writeModel(string filename)* method. If the file name has the extension 'ASCII', then the file is written in ASCII format. Otherwise it is saved in binary format. The model data and the grid are written to the same file. Similarly, to load a model from a file, construct a new model by calling one of the model constructors that does not take the 'relativeGridPath' argument. This action assumes that the data and grid are contained in the same file.

The model grid and the model data can either be stored together in the same file or they can be stored in separate files. In many applications, a model will consist of a single dataset and a single grid, in which case it will make the most sense to store the data and grid together in the same file. In other applications there might be numerous datasets that are all stored on the same grid. For example, the data may consist of pre-computed travel time predictions for a single station-phase to every point on the Earth discretized onto the vertices of a grid. There may be separate datasets for each station in a large network of stations, all of which use the same grid of source positions. If the application needs to be able to load only a subset of the datasets at any one time, it would be most efficient for the grid to be stored separately from the data so the grid could be loaded once and serve the needs of any dataset that might be loaded.

GeoTess supports the ability to store the grid and data in separate files. To accomplish this, every grid has stored within it a unique string that identifies that grid. Typically, this is an MD5 hash of the contents of the grid (node positions, connectivity, etc.). When a model is stored without its grid, it stores two pieces of information: the name of the file containing the grid and the grid's unique gridID.

When an application wants to write a model to file, it supplies two parameters: the name of the file to receive the model and the name of the file to receive the grid. If the supplied grid file name is the single character '*', then the grid is stored in the same file as the model, right after the model data. If a separate file name is specified for the grid, then the model metadata and model data are written to the model file along with the name of the grid file and the gridID. The grid file name stored in the model file does not include any directory information; just the name of the file.

When an application wishes to read a model from file, it supplies two pieces of information: the name of the model file and the relative path from the directory where the model is stored to the directory that is to be searched for the associated grid file. If the model file contains the grid, then the supplied grid directory name is ignored. If the model file does not contain the grid, then the full path to the grid file is constructed from the name of the directory where the model is stored, the relative path to the grid directory supplied by the application, and the grid file name stored in the model file. After the grid is loaded from the separate file, the gridID in the grid file and the gridID in the model file are compared and if they are not the same and exception is thrown.

## 4.3. Model Interrogation

Once a GeoTessModel has been loaded into memory, applications will need to access information in the model. This section gives a general overview of the types of model interrogation that can be accomplished. For a complete definition of all available functionality please consult the online documentation (Java, C++). Examples of the most common model interrogation functions are provided in both languages in the examples folders described in Section 4, specifically in the code TestCrust20.

Model interrogations functions fall into three general categories: information about the grid, the values of data attributes stored at grid nodes, and interpolation of data attribute values at off-node locations. These three attributes are described in the following sections.

### *4.3.1. Grid Information*

GeoTessGrid manages the geometry and topology of a model but has no information about any data attached to the grid. It has some 45 functions that start with *'get…'* to retrieve information about the vertices, triangles, tessellation levels and multi-level tessellations that comprise the grid. The html documentation (Java, C++) is the best source of information about these functions.

The most fundamental query made on a GeoTessGrid object is the *getVertex(i)* method, which returns the unit vector which defines the location of the *ith* vertex in the grid. GeoTess provides the capability to convert back and forth between a unit vector and geographic latitude and longitude.

### 4.3.2. Accessing Data Stored in the Model

General information about a model can be retrieved from the GeoTessMetaData object accessible from the model. Information that is available includes:

- The model description
- The names, units, and indexes of the attributes
- The type of the data (double, float, long, int, short or byte)
- The names and indexes of the layers
- The names of the files from which the model and grid were loaded and the amount of time required to load the model and grid.
- The name and version number of the software that generated the model and the date that the model was generated.
- The map between layer and tessellation indexes.

GeoTessMetaData will allow most of this information to be modified also.

Actual data values stored on grid nodes can be retrieved/modified in one of two ways. The first is to make the request through the model's PointMap (*model.getPointMap().getValue(ptIndex, aIndex);* and *model.getPointMap().setValue(ptIndex, aIndex, newValue);* where *ptIndex* is the index of one of the points in the model and *aIndex* is the index of the attribute). The second is to retrieve data through the model's array of Profiles (*model.getProfile[i][j].getValue(k, aIndex);* where *i* is the index of a vertex, *j* is the index of a layer, *k* is the index of a node, and *aIndex* is the index of the attribute). To modify values using Profiles, it is necessary to create a new Data object with the new value(s) and replace the existing Data object in the Profile by calling *profile.setData(index, data)*. The radii of the nodes can similarly be accessed/modified through the model's PointMap or through its array of Profiles.

### 4.3.3. Interpolating Attribute Values at Arbitrary Locations

GeoTessPosition objects manage the interpolation of attribute values at off-grid locations. Applications obtain a GeoTessPosition object by calling either *model.getGeoTessPosition()* or *GeoTessPosition.getGeoTessPosition()*. These accessors take optional parameters that specify the type of interpolation that the GeoTessPosition object should perform. Options are linear or natural neighbor interpolation in the geographic dimensions and linear interpolation in the radial dimension.

Once a GeoTessPosition object has been instantiated, users call one of the 4 *set()* methods to specify the point in model space where the interpolation is to be performed. All 4 *set()* methods take a 3D position in space, either as a latitude, longitude, depth or as a unit vector and radius. Two of the *set()* methods take a layer id in addition to the spatial position. If the layer id is not supplied then the *set()* method will determine which layer the supplied position is in and store that layer id. If the layer id is supplied in the *set()* method, then the GeoTessPosition object will use that layer id, regardless of which layer the supplied position is in. It is important to note that the supplied position does not need to be located in the layer that corresponds to the layered stored by the GeoTessPosition object.

After one of the *set()* methods has been called, a request can be made to interpolate an attribute value by calling *getValue(atttributeIndex)*. If the current position is located within the boundaries of the current layer stored by the GeoTessPosition object, then the interpolated attribute value is returned. If the current position is not within the current layer stored by the GeoTessPosition object, the behavior is controlled by the parameter *radiusOutOfRangeAllowed*. If the parameter is true (the default) then the interpolated value at the top or bottom of the layer is computed and returned. If *radiusOutOfRangeAllowed* is false, *getValue()* will return NaN. A getter and a setter are provided to retrieve or modify the value of *radiusOutOfRangeAllowed*.

It is also possible to change only the radius/depth of the current interpolation position, without changing the geographic position. See methods *setDepth()* and *setRadius()*.
A GeoTessPosition object can return many other values of interest relative to the position most recently set, including the radii/depths of the top and bottom of the current or any other layer, the radial and geographic interpolation coefficients for the current position, the index of the triangle in which the current position is located, and more. See the html documentation (Java, C++) for more information about these methods.

## 4.4. Using Great Circles

Finally, as great circles are often used in GeoTess, they will be described in detail here. The GreatCircle class manages the information about a great circle path that extends from one point to another point, both of which are located on the surface of a unit sphere. It supports great circles where the distance from the *firstPoint* to the *lastPoint* are 0 to 2*PI radians apart, inclusive. Either one or both points may coincide with one of the poles of the Earth.

There is a method to retrieve a point that is located on the great circle at some specified distance from the first point of the great circle.

The method *getIntersection(other, inRange)* will return a point that is located at the intersection of two great circles. In general, two great circles intersect at two points, and this method returns the one that is encountered first as one moves away from the first point of the first GreatCircle. If the Boolean argument *inRange* is true, then the method will only return a point if the point falls within the range of both great circles. In other words, the point of intersection must reside in between the first and last point of both great circles. If *inRange* is false, then that constraint is not applied.
GreatCircle can transform the coordinates of an input point so that it resides in the plane of the great circle. This is useful for extracting slices from a 3D model for plotting purposes.
The z-coordinate of the transformed point will point out of the plane of the great circle toward the observer. The y-coordinate of the transformed point will be equal to the normalized vector sum of the first and last point of the great circle and the x-coordinate will be y cross z.

The key to successfully defining a great circle path is successfully determining the unit vector that is normal to the plane of the great circle (*firstPoint* cross *lastPoint*, normalized to unit length). For great circles where the distance from *firstPoint* to *lastPoint* is more than zero and less than *PI* radians, this is straightforward. But for great circles longer than PI radians, great circles of exactly zero, PI or 2*PI radians length, or great circles where the first point resides on one of the poles, complications arise. To determine the normal to the great circle, three constructors are provided (besides the default constructor that does nothing).

The first constructor is the most general. It takes four arguments: *firstPoint* (unit vector), *intermediatePoint* (unit vector), *lastPoint* (unit vector) and *shortestPath* (boolean). The *normal* is computed as *firstPoint* cross *lastPoint* normalized to unit length. If the distance from *firstPoint* to *lastPoint* is greater than zero and less than *PI* radians, then the resulting normal will have finite length and will have been successfully computed. If, however, the distance from *firstPoint* to *lastPoint* is exactly 0 or PI radians, then *normal* will have zero length. In this case, a second attempt to compute the *normal* is executed by computing *firstPoint* cross *intermediatePoint*. If this is successful, the calculation proceeds. If not successful, then the *normal* is computed as the first of: *firstPoint* cross *Z*, *firstPoint* cross *Y* or *firstPoint* cross *X*, whichever produces a finite length normal first. *Z* is the north pole, *Y* is (0N, 90E) and *X* is (0N, 0E). One of these calculations is guaranteed to produce a valid *normal*. Once the *normal* has been computed, then the *shortestPath* argument is considered. If *shortestPath* is true, then no further action is taken, resulting in a great circle with length less than or equal to *PI* radians. If *shortestPath* is false then the normal is negated, effectively forcing the great circle to go the long way around the globe to get from *firstPoint* to *lastPoint*. When *shortestPath* is false the length of the great circle will be >= *PI* and <= 2**PI*. For example, when *shortestPath* is true, a great circle path from (10N, 0E) to (30N, 0E) will proceed in a northerly direction for a distance of 20 degrees to get from *firstPoint* to *lastPoint*. But if *shortestPath* is false, the great circle will proceed in a southerly direction for 340 degrees to get from *firstPoint* to *lastPoint*.

The second constructor is a simplification of the first, taking only 3 arguments: *firstPoint* (unit vector), *lastPoint* (unit vector) and *shortestPath* (boolean). It calls the first constructor with *intermediatePoint* set to NULL. This is useful in cases where the calling application is certain that great circles of length exactly 0 or *PI* radians will not happen or is willing to accept an arbitrary path if it does happen.

There is a third constructor that takes 3 arguments: *firstPoint* (unit vector), distance (radians) and azimuth (radians). The *lastPoint* of the great circle is computed by moving the first point the specified distance in the specified direction. This constructor can produce great circles where the distance from *firstPoint* to *lastPoint* is >= 0 and <= 2*PI, inclusive. It can fail, however, if *firstPoint* coincides with either of the poles because the notion of azimuth from a pole in undetermined.

## 4.5. Extending GeoTess

The Data structures attached to the nodes of a GeoTessModel may not always be able to capture all information that an application may need to store. When this is the case, Java and C++ applications can extend a GeoTessModel to store additional information. An example of an extended GeoTessModel is the GeoTessModelSiteData extension, which allows for the storage of station site term data that cannot be stored on the grid nodes of a standard GeoTessModel.

Extensions included with the current version of GeoTess are LibCorr3DModel, GeoTessModelSiteData, GeoTessModelSLBM (i.e., an RSTT extension; see website), and GeoTessAmplitude. To learn what type of extension (if any) a GeoTessModel has, use the getClassName or toString functions of the GeoTessExplorer application (see Section 6).

Examples of Java and C++ classes that perform GeoTessModel extensions are provided in the folders described in Section 4, specifically the ExtendedModel and GeoTessModelExtended

examples. The basic idea is that the derived class implements the data structures and methods needed to fulfill its requirements, implements all the constructors of a GeoTessModel, and overrides several key protected GeoTessModel IO methods. These IO methods first call the super class IO method and then read/write their own data structures in either ASCII or binary format. See the examples for more information.

# 5. GEOTESSBUILDER

GeoTessBuilder is a GeoTess application used to generate GeoTess grids that can be populated using the library interactions described in Section 4.1.

Note that GeoTessBuilder is only included with GeoTessJava, but the grids it generates can be populated using either language. Further, the user can interact with GeoTessBuilder as a Java library if desired. In this section only the command line application version of GeoTessBuilder will be described. To use GeoTessBuilder as a library, refer to the library interactions in Section 4.1

GeoTessBuilder is a command line driven application that takes as its only argument the name of a properties file that contains information needed to generate the GeoTessGrid. The creation of the application executable is described in Sections 2.1 and 2.2. The following sections describe the information in a generic properties file, the different modes of GeoTessBuilder, and several GeoTessBuilder examples. The examples described are provided in **~/GeoTessJava/Examples** or in **~/Salsa3DSoftware/Examples/GeoTessJava** depending on whether GeoTess was downloaded individually or as part of the Salsa3DSoftware package.

To run any of the following properties files using the command line application, enter

**geotessbuilder *properties_file_name*.properties**

on the command line.

## 5.1. GeoTessBuilder Properties File

The following considerations apply to property files:

- All property names are case-sensitive, but property values are not.
- If a default value is defined for a property, then it is not necessary to specify that property in the properties file.
- If a property value ends with the string ' \' (i.e., a space or tab followed by a backslash character) it is interpreted as a line continuation string. This allows long property values to be split over several lines.
- Tessellation indexes are zero-based, i.e., the first tessellation has index 0 and the last tessellation has index *nTessellations*-1.
- The term *triangle edge length* refers to the approximate length of a triangle's edge measured in degrees. Values should be a power of two, less than or equal to 64, i.e., 64, 32, 16, 8, 4, 2, 1, ½, ¼, ⅛, etc. These values are approximate and assume that the *initialSolid* is an icosahedron.
- Files specified in property values can be either ASCII files or Google Earth kmz/kml files. ASCII files contain points defined as either latitude-longitude or longitude-latitude pairs, in

degrees. Latitude-longitude order is the default, but if the file contains the line 'lon-lat', then points are assumed to be in lon-lat order. Latitude and longitude values can be separated by either a comma or white space. Kml/kmz files contain points, paths or polygons as defined by Google Earth.

GeoTessBuilder operates in one of two distinct modes: *model refinement* and *construction from scratch*. In *model refinement mode* an existing GeoTessModel is refined in the neighborhood of a subset of the points in the model. The user supplies both the name of the file containing the GeoTessModel, and the name of a file containing the indices of all the points in the model about which refinement is to take place. The output is a new GeoTessModel. In *construction from scratch mode*, a new GeoTessGrid is constructed using specifications defined in below.

In *construction-from-scratch mode*, points, paths and polygons are not mutually exclusive. Refinement of the same tessellation using any or all methods in any combination is allowed. Also, multiple multi-level tessellations may be defined in the properties file and each tessellation may be refined independently from the others.

### 5.1.1. Model Refinement Mode

*gridConstructionMode* – if this property is equal to '*model refinement*', then an existing model will be refined in the neighborhood of a list of specified points. The following properties are relevant.

*modelToRefine* – The full path to the file containing the GeoTessModel that is to be refined.

*fileOfPointsToRefine* – The name of the file containing the indices of the points in the model that are to be refined.

*polygonToRefine* – The name of the file containing a polygon. All the points inside the polygon will be refined. See the section of this document about Polygons for more information.

*outputModelFile* – The name of the file to receive the new GeoTessModel that will be generated by GeoTessBuilder.

Note that in *model refinement mode*, none of the properties defined in the next section are accessed by the code.

### 5.1.2. Construction-From-Scratch Mode

*gridConstructionMode* – if this property is equal to '*scratch'*, then GeoTessBuilder will construct a new GeoTessGrid from scratch using properties defined in this section.

*initialSolid* – this property specifies the initial solid that defines the first level of each of the multi-level tessellations that will be included in the new grid. The options are: icosahedron (default), tetrahexahedron, octahedron, and tetrahedron. Any other value will cause an exception.

*nTessellations* – The number of multi-level tessellations to be included in the grid. The default is 1.

*baseEdgeLengths* – the minimum *triangle edge length* of the triangles in the top level of each tessellation. The number of values must be equal to *nTessellations*. If no points, paths or polygons are specified as

described shortly, then uniform tessellations with this geographic resolution will be constructed. If points, paths and/or polygons are specified, this property specifies the triangle size far from any of the points, paths or polygons.

*points* – specification of a list of geographic locations about which refinement is to take place. The supplied value is parsed as follows: First, the property value is split into substrings based on the semicolon character (';'). Each of these substrings defines a single point about which refinement is to occur. Each substring is split on the comma character (',') into a number of tokens.

- If the resulting array of strings contains 3 tokens, they are interpreted to be (1) a *file name*, (2) a *tessellation index*, and (3) a *triangle edge length*. Points are read from the specified file and the *multi-level tessellation* with the specified index will be refined around all the points to the specified *triangle edge length*.

- If the resulting array of tokens contains 5 elements, they are interpreted as follows:

    1. Either '*lat-lon*' or '*lon-lat*' defining the order of latitude and longitude in entries 4 and 5 below.

    2. The index of the *multi-level tessellation* to refine.

    3. The *triangle edge length* specifying how small the refined triangles around the point should be.

    4. Latitude or longitude of the point in degrees.

    5. Latitude or longitude of the point degrees.

*paths* – specification of lists of points that define paths. All triangles that contain any segment of the specified paths will be refined to the specified level. The property value is parsed as follows: First, the property value is split into substrings based on the semicolon character (';'). Each substring includes the specification of a single *path*. Each substring is split on the comma character (','). The resulting array of tokens must contain 3 tokens, which are interpreted to be (1) a file name, (2) a *tessellation index*, and (3) a *triangle edge length*. Points are read from the specified *file* and the multi-level tessellation with the specified index will be refined around all the paths to the specified *triangle edge length*.

*polygons* – specification of one or more polygons. All triangles that have at least one corner inside one of the polygons will be refined. The property value is parsed as follows: First, the property value is split into substrings based on the semicolon character (';'). Each substring is the specification of a single *polygon*. Each substring is split on the comma character (','). The resulting tokens are interpreted as follows:

- If the first token is equal to '*small_circles'*, then the remaining tokens are interpreted as:

    - latitude of the center of the small circles
    - longitude of the center of the small circles

    - 'in' or 'out' specifying whether the center of the small circles is considered inside or outside the polygon
    - an arbitrary number of small circle radii, in degrees.
    - tessellation index
    - triangle edge length specifying the size of the triangles desired inside the polygon.

  The small circle radii, which must be > 0 and < 180 degrees, will define a series of bands surrounding the center point, with the bands alternating between being inside and outside the polygon. For example, to define a spherical cap surrounding the central point, specify a single radius at the desired distance from the central point and specify that the central point is 'in'. A polygon appropriate for defining the region of validity for seismic phase PKPbc, valid from 145 to 155 degrees, can be specified with two small circle radii and specifying that the center of the polygon is 'out'.

- Otherwise, the tokens are interpreted as follows:
    - The name of a *file* containing the definition of a polygon. Files can be either an ASCII file or a Google Earth kmz/kml file. ASCII files contain a list of points defining a closed polygon. Each point is specified as either a latitude-longitude or longitude-latitude pair, in degrees. Latitude-longitude order is the default, but if the file contains the line 'lon-lat', then points are assumed to be in lon-lat order. Latitude and longitude values can be separated by either a comma or white space. Kml/kmz files contain a single polygon as defined by Google Earth.
    - *tessellation* index
    - *triangle edge length* specifying the size of the triangles desired within the polygon.

*outputGridFile* – the name of the file to receive the new GeoTessGrid.

*vtkFile* – the name of the file to receive the GeoTessGrid in vtk format (http://www.vtk.org/VTK/img/file-formats.pdf). These files can be opened with *ParaView*, which is free software for visualization of 3D objects (http://www.paraview.org). A separate file will be generated for each multi-level tessellation. Include the substring '%d' in the file name. It will be replaced with the tessellation number in the filename. If only one tessellation is being generated then the '%d' substring is not required. The filename must end with the extension ".vtk".

*rotateGrid* – This property will cause the grid to be rotated such that grid vertex 0 will be located at the specified position. Specify **geographic** latitude and longitude in degrees. This is equivalent to using property *eulerRotationAngles* with values *lon*+90, *geocentric_colat*, -90.

*eulerRotationAngles* – It is possible to rotate the triangular tessellation produced by GeoTessBuilder, relative to the traditional latitude, longitude grid, by providing 3 Euler rotations angles, in degrees. Given two coordinate systems xyz and XYZ with common origin, starting with the axis z and Z overlapping, the position of the second can be specified in terms of the first using three rotations with angles A, B, C as follows:

1. Rotate the xyz-system about the z-axis by A.

2. Rotate the xyz-system again about the now rotated x-axis by B.

3. Rotate the xyz-system a third time about the new z-axis by C.

Clockwise rotations, when looking in direction of vector, are positive (see http://mathworld.wolfram.com/EulerAngles.html).

For example, to rotate the grid such that grid vertex 0, which normally coincides with the north pole, resides instead at position *geocentric_lat0*, *lon0* (in degrees), then supply the 3 Euler rotation angles *lon0*+90, 90 - *geocentric_lat0*, -90.

## 5.2. GeoTessBuilder Examples

The examples described in the following sections can be found in **~/GeoTessJava/Examples** or in **~/Salsa3DSoftware/Examples/GeoTessJava** depending on whether GeoTess was downloaded individually or as part of the Salsa3DSoftware package.

### *5.2.1. Example 1*

The first example of building a grid will construct a single GeoTessGrid object that is comprised of three multi-level tessellations. The example properties file can be found in the Examples subfolder **gridbuilder_3_tessellation_grid**. Each tessellation has uniform resolution in the geographic dimensions. See Section 4.1 to learn how to use a GeoTessGrid such as this one to build a GeoTessModel.

The property file for this example contains:

```
# file: gridbuilder.properties
# this properties file will result in a single GeoTessGrid
# object consisting of 3 multi-level tessellations. The
# triangles on the top level of each tessellation will each be
# approximately uniform in the geographic dimensions.
# Tessellation 0 will have triangles with edge lengths of about 32 degrees
# Tessellation 1 will have triangles with edge lengths of about 16 degrees
# Tessellation 2 will have triangles with edge lengths of about 4 degrees.
# All three tessellations will be stored together in the same output file.
# Separate vtk files will be generated for each tessellation for visualization.

# specify GeoTessBuilder grid construction mode.
gridConstructionMode = scratch

# number of multi-level tessellations to build
nTessellations = 3

# the triangle size that is to be achieved on the
# top tessellation level of each multi-level tessellation
baseEdgeLengths = 32 16 4

# file to receive the GeoTessGrid definition
outputGridFile = three_uniform_tessellations.geotess

# file to receive the vtk files used for visualization with ParaView.
# Since there are three tessellations, the '%d' substring is required.
# Three vtk files will be produced, one for each tessellation.
# The '%d' substrings will be replaced with the tessellation index.
vtkFile = three_uniform_tessellations_%d.vtk
```

To run this example, run **geotessbuilder example.properties** on the command line. This action will output a GeoTess model (three_uniform_tesellations.geotess) and four ParaView vtk files (continent_boundaries.vtk, three_uniform_tessellations_0.vtk, three_uniform_tessellations_1.vtk,

three_uniform_tessellations_2.vtk). The vtk files, plotted in Figure 5-1, illustrate the 3 multi-level tessellations that result from running this example.

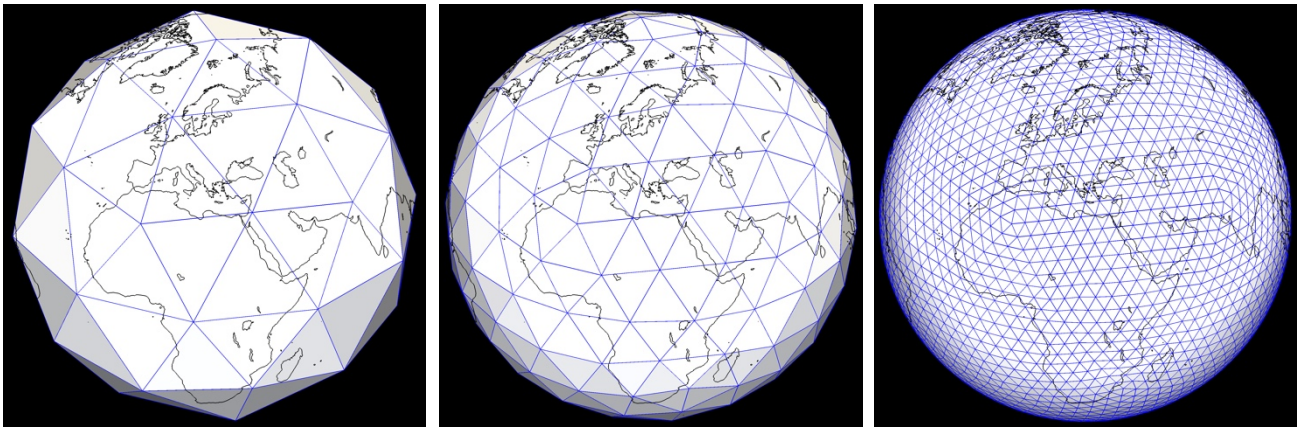

**Figure 5-1. Three multi-level tessellations generated by example 1. The tessellations have triangle sizes of approximately 32°, 16°, and 4°. Each image shows only the top level of the corresponding multi-level tessellation. These multi-level tessellations are all components of a single GeoTessGrid object, stored in a single GeoTessGrid output file. The continental outlines are for illustrative purposes only and are not part of the GeoTessGrid object.**

### *5.2.2. Example 2*

The next example constructs a single GeoTessGrid object that is comprised of a single multi-level tessellation. The top level of this tessellation will be composed mainly of triangles with approximately 8° edge lengths. But in the neighborhood of a single point in the North Atlantic, the triangles are refined down to triangles with edge lengths of approximately ⅛°. The properties file for this example can be found in the Examples subfolder **gridbuilder_point_refinement**.

The property file for this example contains:

```
# file: gridbuilder_point_example.properties
# this properties file will result in a single GeoTessGrid
# object consisting of 1 multi-level tessellation with the
# triangles on the top tessellation level having edge lengths
# of about 8 degrees. In the neighborhood of a point located
# at about 32N, 36W, the triangles are refined down to a
# triangle size of about 1/8th of a degree.

# specify GeoTessBuilder grid construction mode.
gridConstructionMode = scratch

# number of multi-level tessellations to build
nTessellations = 1

# the triangle size that is to be achieved on the
# top tessellation level, far from refinement point
baseEdgeLengths = 8

# specify a single point. The tokens in the property value are:
# 1) lat-lon, 2) tessellation index, 3) triangle edge length in degrees,
# 4) latitude and 5) longitude. More points could have been
# specified by including similar strings, separated by semi-colons.
points = lat-lon, 0, 0.125, 31.88984, -36.000000

# file to receive the GeoTessGrid definition
outputGridFile = gridbuilder_point_example.geotess
```

```
# file to receive the vtk file used for visualization with ParaView
vtkFile = gridbuilder_point_example.vtk
```

To run this example, run **geotessbuilder example.properties** on the command line. This action will output a GeoTess model (gridbuilder_point_example.geotess) and two ParaView vtk files (continent_boundaries.vtk, gridbuilder_point_example.vtk). The vtk files, plotted in Figure 5-2, illustrate the variable resolution tessellation that is generated by this example.

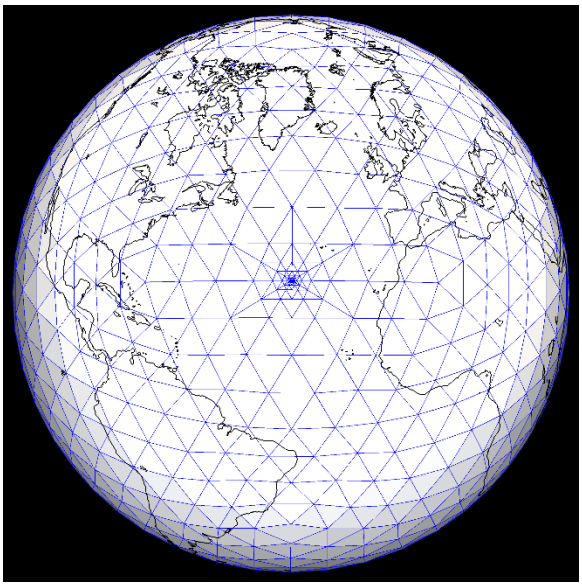

**Figure 5-2. Top level of the multi-level tessellation generated by example 2. Most of the triangles shown have edge lengths of approximately 8° but triangles near the refinement point are as small as ⅛°.**

### *5.2.3. Example 3*

In this example a single GeoTessGrid object comprised of a single multi-level tessellation is constructed. The top level of this tessellation will be composed of triangles mainly of approximately 8° edge lengths. But in the neighborhood of a path describing the mid-Atlantic Ridge, the triangles are refined down triangles of edge length of approximately 0.5°. The path that defines the mid-Atlantic Ridge is stored in a Google Earth .kmz file. The properties file for this example, along with the .kmz file defining the Atlantic Ridge, can be found in the Examples subfolder **gridbuilder_path_refinement**.

The property file for this example contains:

```
# file: gridbuilder_path_example.properties
# this properties file will result in a single GeoTessGrid
# object consisting of 1 multi-level tessellation with the
# triangles on the top tessellation level having edge lengths
# of about 8 degrees. In the neighborhood of a path describing
# the trace of the mid-Atlantic Ridge, the triangles are refined
# down to a triangle size of about 1 degree.

# specify GeoTessBuilder grid construction mode.
gridConstructionMode = scratch

# number of multi-level tessellations to build
nTessellations = 1
```

```
# the triangle size that is to be achieved on the
# top tessellation level from the path defined below.
baseEdgeLengths = 8

# specify a single path. The tokens in the property value are:
# 1) the name of the file containing the path, 2) tessellation
# index, and 3) triangle size for triangles near the path.
paths = mid_atlantic_ridge.kmz, 0, 1.0

# file to receive the GeoTessGrid definition
outputGridFile = gridbuilder_path_example.geotess


# file to receive the vtk file used for visualization with ParaView
vtkFile = gridbuilder_path_example.vtk
```

To run this example, run **geotessbuilder example.properties** on the command line. This action will output a GeoTess model (gridbuilder_path_example.geotess) and two ParaView vtk files (continent_boundaries.vtk, gridbuilder_path_example.vtk). The vtk files, plotted in Figure 5-3Figure 5-2, illustrate the variable resolution tessellation around the Atlantic Ridge that is generated by this example.

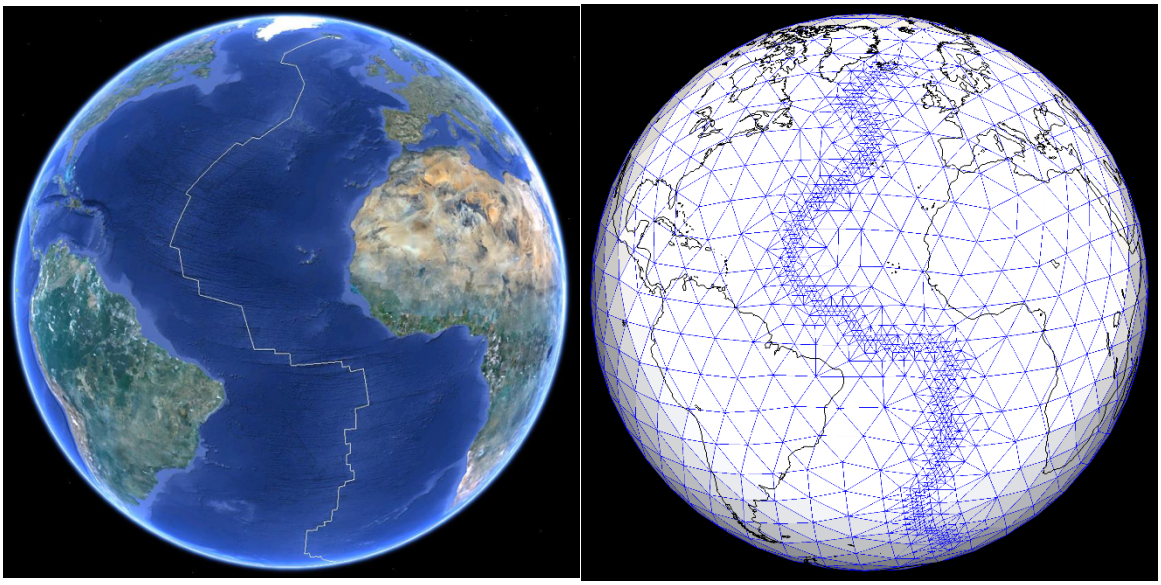

**Figure 5-3. (left) The trace of the mid-Atlantic Ridge as viewed with Google Earth. (right) Top level of the multi-level tessellation generated by example 3. Most of the triangles shown have edge lengths of approximately 8° but triangles that span the mid-Atlantic Ridge have triangles with edge lengths about 1°.**

### *5.2.4. Example 4*

In this example a single GeoTessGrid object is constructed that is comprised of a single multi-level tessellation. The top level of this tessellation will be composed of triangles mainly of approximately 8° edge lengths. All triangles with a corner inside a polygon surrounding the lower 48 states of the US are refined to about 1° and triangles with a corner inside a polygon outlining the state of New Mexico are refined to about ⅛°. The properties file for this example, along with the .kml file defining the United States polygon and the .kmz file defining the New Mexico polygon, can be found in the Examples subfolder **gridbuilder_polygon_refinement**.

The property file for this example contains:

```
# file: gridbuilder_polygon_example.properties
# this properties file will result in a single GeoTessGrid
# object consisting of 1 multi-level tessellation with the
# triangles on the top tessellation level having edge lengths
# of about 8 degrees. Triangles with at least one corner
# inside a polygon surrounding the lower 48 states in the US
# are refined to about 1 degree. Triangles with at least
# one corner inside a polygon outlining the state of New
# Mexico are further refined to about 1/8 the of a degree.

# specify GeoTessBuilder grid construction mode.
gridConstructionMode = scratch

# number of multi-level tessellations to build
nTessellations = 1

# the triangle size that is to be achieved on the
# top tessellation level from the path defined below.
baseEdgeLengths = 8

polygons = \
 united_states.kml, 0, 1.0 ; \
 new_mexico.kmz, 0, 0.125

# file to receive the GeoTessGrid definition
outputGridFile = gridbuilder_polygon_example.geotess

# file to receive the vtk file used for visualization with ParaView
vtkFile = gridbuilder_polygon_example.vtk
```

To run this example, run **geotessbuilder example.properties** on the command line. This action will output a GeoTess model (gridbuilder_path_example.geotess) and two ParaView vtk files (continent_boundaries.vtk, gridbuilder_path_example.vtk). The vtk files, plotted in Figure 5-4Figure 5-2, illustrate the variable resolution tessellation around the United States and New Mexico that is generated by this example.

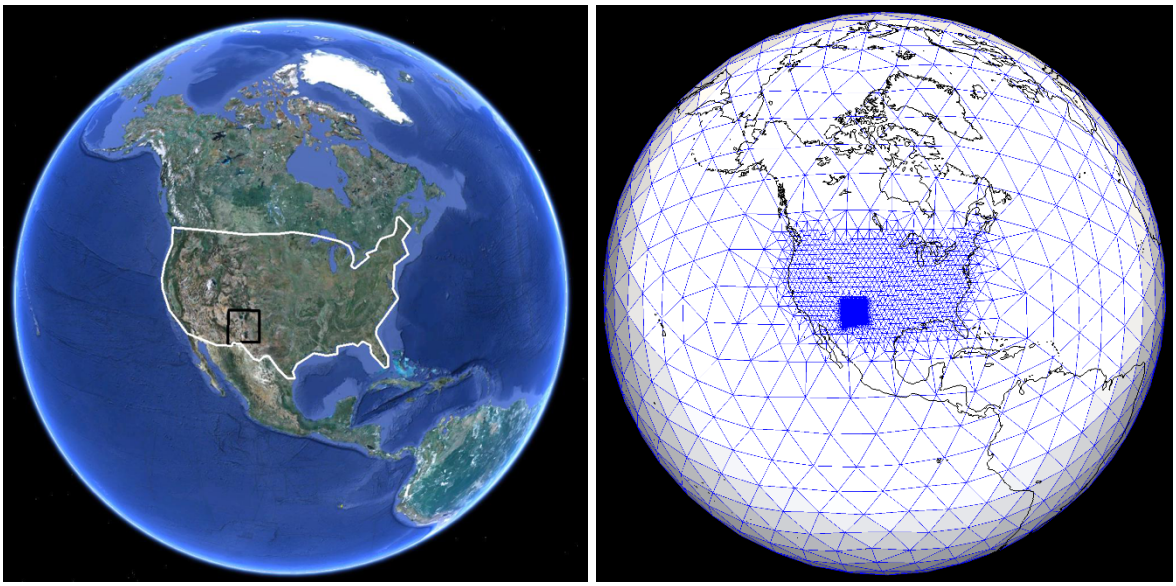

**Figure 5-4. (left) Polygons generated and viewed with Google Earth. (right) Top level of the multi-level tessellation generated by example 4. Triangles far from the conterminous US have edge lengths of approximately 8° but triangles with at least one corner inside the polygon surrounding**

**the US have triangles with edge lengths about 1°. Triangles with a corner in the polygon defining the state of New Mexico have edge lengths of approximately ⅛°.**

# 6. GEOTESSEXPLORER

GeoTessExplorer is a GeoTess application application that implements a set of command line driven functions to extract or modify information in a GeoTessModel in several ways, including extracting maps, generating vtk plot files, extracting site terms, etc. Note that GeoTessExplorer is only included with GeoTessJava, but its functions can be applied to GeoTess models generated by either language.

To view a list of available functions included in GeoTessExplorer along with the needed inputs, run the executable

**geotess**

on the command line. The creation of this executable is described in Sections 2.1 and 2.2.

This action will output the list of currently available functions shown below.

```
GeoTessExplorer 1.2023.4

Specify one of the following functions:
version                 -- output the GeoTess version number
toString                -- print summary information about a model
updateModelDescription  -- update the description information for a model
statistics              -- print summary statistics about the data in a model
getClassName            -- discover the class name of a specified model
equal                   -- given two GeoTessModels test that all radii and attribute values of all nodes are ==.  Metadata can differ.
extractGrid             -- load a model or grid and write its grid to stdout, vtk, kml, ascii or binary file
resample                -- resample a model onto a new grid
extractActiveNodes      -- load a model and extract the positions of all active nodes
replaceAttributeValues  -- replace the attribute values associated with all active nodes
reformat                -- load a model and write it out in another format
getValues               -- interpolate values at a single point
getValuesFile           -- interpolate values at points specified in an ascii file
interpolatePoint        -- interpolate values at a single point (verbose)
borehole                -- interpolate values along a radial profile
profile                 -- extract model values at vertex closest to specified latitude, longitude position
findClosestPoint        -- find the closest point to a supplied geographic location and return information about it
slice                   -- interpolate values on a vertical plane defined by a great circle connecting two points
sliceDistAz             -- interpolate values on a vertical plane defined by a great circle defined by a point, a distance and a direction
mapValuesDepth          -- interpolate values on a lat, lon grid at constant depths
mapValuesLayer          -- interpolate values on a lat, lon grid at fractional radius in a layer
mapLayerBoundary        -- depth of layer boundaries on a lat, lon grid
mapLayerThickness       -- layer thickness on a lat, lon grid
values3DBlock           -- interpolate values on a regular lat, lon, radius grid
function                -- new model with attributes calculated from two input models
vtkLayers               -- generate vtk plot file of values at the tops of layers
vtkDepths               -- generate vtk plot file of values at specified depths
vtkDepths2              -- generate vtk plot file of values at specified depths
vtkLayerThickness       -- generate vtk plot file of layer thicknesses
vtkLayerBoundary        -- generate vtk plot file of depth or elevation of layer boundary
vtkSlice                -- generate vtk plot file of vertical slice
vtkSolid                -- generate vtk plot file of entire globe
vtk3DBlock              -- generate vtk plot file of values on a lat-lon-depth grid
vtkPoints               -- generate vtk plot of point data
vtkRobinson             -- generate vtk plot of a Robinson projection of model data
vtkRobinsonLayers       -- generate vtk plot of a Robinson projection of model data at tops of multiple layers
vtkRobinsonPoints       -- generate vtk plot of a Robinson projection of point data
vtkRobinsonTriangleSize -- generate vtk plot of triangle size on Robinson projection
vtkLayerAverage         -- generate vtk plot of the average values within the crust of a designated model
reciprocalModel         -- generates a reciprocal GeoTessModel, where all values are inverted
renameLayer             -- renames an individual layer in a GeoTessModel
renameLayers            -- renames all layers in a GeoTessModel
getLatitudes            -- array of equally spaced latitude values
getLongitudes           -- array of equally spaced longitude values
getDistanceDegrees      -- array of equally spaced distances along a great circle
translatePolygon        -- translate polygon between kml/kmz and ascii format

GeoTessModelSiteData:
extractSiteTerms        -- extract site terms from a GeoTessModelSiteData and print to screen
replaceSiteTerms        -- replace site terms in a GeoTessModelSiteData with values loaded from a file

RSTT:
extractPathDependentUncertaintyRSTT -- extract all the path dependent uncertainty information from a GeoTessModelSLBM
replacePathDependentUncertaintyRSTT -- replace all the path dependent uncertainty information in a GeoTessModelSLBM

Note that when a function requests a 'list of attributes'
specify a string like '0' or '0,2' or '0-2' or 'n' or '1-n' or 'all'
where 'n' is interpreted to be the index of the last attribute
```

**Figure 6-1. List of GeoTessExplorer Functions.**

To view a list of required arguments for any of the above functions, run **geotess functionName**. For example, running **geotess toString** will output the following instructions:

*Must supply either 2 or 3 arguments:*
*1 -- toString*
*2 -- name of file containing a GeoTessModel or GeoTessGrid*
*3 -- relative path to grid directory  (only needed when (2) is a model and grid is stored in separate file)*

Note that if any required arguments for a function are missing during a run, its list of required arguments will be output, indicating the user needs to correct the arguments and run again.

The intention of GeoTessExplorer is for users to either pipe the output to a file or to insert a call to this program into a script with the output piped to some other program. For example, to save the output of the toString function to a text file, run **geotess toString SALSA3D.geotess > toString.txt**.

Many functions require a *'list of attributes'* as one of the command line arguments. This list can be a string similar to *'0,2,4-n'*, which would return attributes 0, 2 and 4 through the number of available attributes. '*n*' would return only the last attribute. *'all'* and *'0-n'* would both return all attributes. The list may not include any spaces.

For most functions, the first two arguments after the function name are the name of the input model file and the relative path to the grid directory. Some models have the grid stored in the same file with the model while other models reference a grid stored in a separate file. If the grid is stored in the same file with the model, then the relative path to the grid directory is irrelevant but something must be supplied to maintain the order of the argument list (typically this is the single character '.').

If the grid is stored in a separate file, then the name of the file that contains the grid, without any directory information, is stored in the model file. When the model is loaded, it must be told the relative path from the directory where the model is located to the directory where the grid file is located. If the grid is in a separate file located in the same directory as the model file, provide the single character *'.'*. Note that models and grids also contain an MD5 hash of the grid file contents, so the danger of a model referencing the wrong grid is vanishingly small.

All the functions whose names start with *'vtk'* extract information from a GeoTessModel and store it in a file in vtk format (http://www.vtk.org/VTK/img/file-formats.pdf). These files can be visualized with free software called ParaView. Visit http://www.paraview.org for more information and downloads for various platforms.

Finally, most of the functions listed above can be applied to any type of GeoTessModel, with the exception of the functions listed under GeoTessModelSiteData. These are specifically for application to GeoTessModels with the site data extension, i.e., GeoTessModelSiteData. If the user applies these functions to another type of GeoTess model, a warning is printed to screen indicating the function cannot be applied to that model type. To learn what type of extension (if any) a GeoTessModel has, use the GeoTessExplorer getClassName function.

# APPENDIX A. EXAMPLE GEOTESS PROPERTY FILES

### *A.1.1. PopulateModel3D.cc*

```
#include "CPPUtils.h"
#include "GeoTessGrid.h"
#include "GeoTessModel.h"
#include "GeoTessPosition.h"
#include "AK135Model.h"

using namespace geotess;

/**
 * An example of populating a 3D model with data.  The application
 * loads a existing GeoTessGrid object from a file, populates it
 * with information from the ak135 model to build a 3D version of
 * 1D ak135 model. The resulting model has 7 layers supported by
 * 3 multi-level tessellations.  The first tessellation (index 0)
 * supports the inner and outer core. The second tessellation
 * supports the three mantle layers. The third and final tessellation
 * supports the lower and upper crust.
 * <p>
 * The program takes one command line argument which specifies the
 * full path to the file GeoTessModels/crust20.geotess
 * that was delivered with the GeoTess package.
 */
int main(int argc, char** argv)
{
        try
        {
                if(argc < 2)
                {
                        cout << "Must supply a single command line argument specifying path to the
GeoTessModels directory" << endl;
                        return -1;
                }

                string path = argv[1];

                // specify the path to the file containing the grid to be used for
                // this test.  This information was passed in as a command line
                // argument.  Grids were included in the software delivery and
                // are available from the GeoTess website.  Grids can also be
                // constructed using the GeoTessBuilder software.  The grid
                // required for this example is in a file called
                // small_model_grid.ASCII which is in the GeoTessModels
                // directory delivered with GeoTess.
                string inputGridFileName = CPPUtils::insertPathSeparator(path,
"small_model_grid.ASCII");

                cout << "Example that illustrates how to populate a 3D model." << endl << endl;

                // Create a MetaData object in which we can specify information
                // needed for model construction.
                GeoTessMetaData* metaData = new GeoTessMetaData();

                // specify the ellipsoid that is to be used when interacting with this model.
                // This call is really unnecessary here because WGS84 is the default,
                // but other options are possible.
                metaData->setEarthShape("WGS84");

                // Specify a description of the model. This information is not
                // processed in any way by GeoTess. It is carried around for
                // information purposes.
                metaData->setDescription("Simple example of populating a 3D GeoTess model\ncomprised of
3 multi-level tessellations\n");

                // Specify a list of layer names delimited by semi-colons
                metaData->setLayerNames("INNER_CORE; OUTER_CORE; LOWER_MANTLE; TRANSITION_ZONE;
UPPER_MANTLE; LOWER_CRUST; UPPER_CRUST");
```

```
                // Specify the relationship between grid tessellations and model layers.
                // the list has nLayers elements where each element specifies the
                // index of the multilevel tessellation that supports the
                // corresponding layer.  In this example, the model has
                // 7 layers and 3 multi-level tessellations.
                // Layers 0 (inner core) and 1 (outer core) are
                //     supported by tessellation 0 which has 64 degree triangles (huge!)
                // Layers 2,3 and 4 (3 mantle layer) are supported by tessellation 1
                //     which has 32 degree triangles.
                // Layers 5 and 6 (crust) are supported by tessellation 2.
                //     which has 16 degree triangles
                int layerTessIds[] = {0, 0, 1, 1, 1, 2, 2};

                // set the layerTessIds in the model.  setLayerTessIds() must be called after
                // setLayerNames(), not before.
                metaData->setLayerTessIds(layerTessIds);

                // specify the names of the attributes and the units of the
                // attributes in two Strings delimited by semi-colons.
                metaData->setAttributes("Vp; Vs; rho", "km/sec; km/sec; g/cc");

                // specify the GeoTessDataType for the data. All attributes, in all
                // profiles, will have the same data type.  Note that this
                // applies only to the data; radii are always stored as floats.
                metaData->setDataType(GeoTessDataType::FLOAT);

                // specify the name of the software that is going to generate
                // the model.  This gets stored in the model for future reference.
                metaData->setModelSoftwareVersion("GeoTessCPPExamples.PopulateModel3D 1.0.0");

                // specify the date when the model was generated.  This gets
                // stored in the model for future reference.
                metaData->setModelGenerationDate(CpuTimer::now());

                // call a GeoTessModel constructor to build the model. This will
                // load the grid, and initialize all the data structures to null.
                // To be useful, we will have to populate the data structures.
                // GeoTessModel assumes ownership of the pointer to metaData  and
                // will delete it in its destructor.
                GeoTessModel* model = new GeoTessModel(inputGridFileName, metaData);

                // retrieve a reference to the EarthShape that should be used when interacting with
                // this model.  This object will be used to convert between geographic and geocentric
                // coordinates, and between depth and radius.
                EarthShape& ellipsoid = model->getEarthShape();

                // We need a source of model data that we can use to populate
                // our new model.  AK135Model is a representation of the 1D
                // radially symmetric velocity model which we have hardcoded
                // into this example.
                AK135Model ak135;

                vector<vector<float> > attributeValues;
                vector<float> radii;

                // Now we will populate the model with Profiles.  At this point, the
                // model has a 2D array of GeoTessProfile objects with dimensions
                // nVertices x nLayers with all the elements of the array initialized
                // to null.  We will now populate the Profiles array.
                //
                // loop over the 7 layers of the model (inner_core, outer_core, etc)
                for (int layer=0; layer<model->getNLayers(); ++layer)
                {
                        // now loop over every vertex in the grid, connected and not-connected.
                        for (int vtx = 0; vtx < model->getNVertices(); ++vtx)
                        {
                                // retrieve a reference to the unit vector corresponding to the i'th
vertex
                                const double* vertex = model->getGrid().getVertex(vtx);

                                // find the latitude and longitude of vertex, in degrees
                                double lat = ellipsoid.getLatDegrees(vertex);
                                double lon = ellipsoid.getLonDegrees(vertex);
```

```
                                // for the current vertex and layer, we need a 1D array of radii
                                // and a 2D array of attribute values (nNodes by nAttributes) that
                                // together define the distribution of model attributes (vp, vs and
                                // density) along a radial profile through this layer.

                                // Note that the number of radii and number of attribute nodes
                                // are not always the same.  Here are the possibilities:
                                // <br>In the crustal layers, there will be two radii and
                                // a single attribute node, indicating that vp, vs and density
                                // are constants in the profiles through crustal layers.
                                // <br>In the core and mantle, the number of radii and
                                // the number of attibute nodes will be equal, indicating
                                // that vp, vs and density vary continuously within each layer
                                // in the radial direction.

                                ak135.getLayerProfile(lat, lon, layer, radii, attributeValues);

                                // pass the vertexID, layer number, radii and values to
                                // the GeoTessModel.  The radii and attribute values will
                                // be copied from these structures into GeoTess objects.
                                model->setProfile(vtx, layer, radii, attributeValues);

                        }
                }

                // At this point, we have a fully functional GeoTessModel object
                // that we can work with.

                // print a bunch of information about the model to the screen.
                cout << model->toString() << endl << endl;

                cout << "Radial profile at the south pole:" << endl;
                cout << "Radius (km) Vp (km/sec) Vs (km/sec) Density (g/cc)" << endl;
                cout << fixed << setprecision(3);
                for (int layer=0; layer < model->getNLayers(); ++layer)
                {
                        GeoTessProfile* p = model->getProfile(11, layer);
                        cout << "Layer " << layer << "  " << model->getMetaData().getLayerName(layer)
<<
                                        " of type " << p->getType().toString() << endl;
                        for (int j=0; j<p->getNRadii(); ++j)
                        {
                                cout << setw(10) << p->getRadius(j);
                                if (j < p->getNData())
                                        for (int k=0; k<model->getMetaData().getNAttributes(); ++k)
                                                cout << setw(10) << p->getData(j)->getFloat(k);
                                cout << endl;
                        }
                }

                cout << endl << endl;

                // write the model to a file.  Note that before the model is
                // written to file, a test is performed to ensure that all the layer
                // radii are consistent.  It must be true that no layer has negative
                // thickness and that the radius of the top of each layer is equal
                // to the radius of the bottom of the next layer, within a
                // small tolerance. If any of these conditions are not satisfied,
                // the model is not written and an exception is thrown.

//              string outputFile = CPPUtils::insertPathSeparator(path, "small_model.ASCII");
//              model->writeModel(outputFile, "*");
//              cout << "model written to file: " << outputFile << endl << endl;

                // Now let's test the model by interpolating some data from it.

                // Instantiate a GeoTessPosition object, giving it a reference to the model.
                // Specify which type of interpolation is to be used: linear or natural neighbor.
                GeoTessPosition* position = model-
>getPosition(GeoTessInterpolatorType::NATURAL_NEIGHBOR);

                // set the latitude and longitude of the GeoTessPosition object.  This is
                // the position on the Earth where we want to interpolate some data.
                double lat = 30.;
```

```
                double lon = 90.;

                // convert the latitude and longitude into an earth-centered unit vector.
                double v[3];
                ellipsoid.getVectorDegrees(lat, lon, v);

                // get the index of the layer that supports the upper crust.
                int layerID = model->getMetaData().getLayerIndex("UPPER_CRUST");

                // specify layer index and geographic position where we want to interpolate data.
                // By calling 'setTop', the radius of the interpolation position will be set to
                // the top of layer with index layerID.
                position->setTop(layerID, v);

                cout << fixed << setprecision(3);

                cout << "Interpolation lat, lon, depth = "
                                << ellipsoid.getLatDegrees(position->getVector()) << " deg, "
                                << ellipsoid.getLonDegrees(position->getVector()) << " deg, "
                                << position->getDepth() << " km" << endl << endl;


                // retrieve the interpolated model values at the most recent location specified
                // in the GeoTessPostion object.
                double vp = position->getValue(0);
                double vs = position->getValue(1);
                double rho = position->getValue(2);

                // Output the interpolated values
                cout << "Interpolated vp  = " << setw(6) << vp << " " << model-
>getMetaData().getAttributeUnit(0) << endl;
                cout << "Interpolated vs  = " << setw(6) << vs << " " << model-
>getMetaData().getAttributeUnit(1) << endl;
                cout << "Interpolated rho = " << setw(6) << rho << " " << model-
>getMetaData().getAttributeUnit(2) << endl;
                cout << endl;

                // print out the index of the triangle in which point resides.
                cout << "Interpolated point resides in triangle index = " << position->getTriangle() <<
endl;

                // print out a table with the node indexes, node lat, node lon and
                // interpolation coefficients for the nodes of the triangle that
                // contains the point.
                cout << "  Node        Lat        Lon      Coeff" << endl;

                // get the indexes of the vertices that contribute to the interpolation.
                const vector<int>& x = position->getVertices();

                // get the interpolation coefficients used in interpolation.
                const vector<double>& coef = position->getHorizontalCoefficients();

                const GeoTessGrid& gridnew = model->getGrid();
                for (int j = 0; j < (int) x.size(); ++j)
                {
                        cout << setw(6) << x[j] <<
                                        setprecision(4) << setw(11) <<
                                        ellipsoid.getLatDegrees(gridnew.getVertex(x[j])) <<
                                        setprecision(4) << setw(11) <<
                                        ellipsoid.getLonDegrees(gridnew.getVertex(x[j])) <<
                                        setprecision(6) << setw(11) <<
                                        coef[j] << endl;
                }

                delete position;
                delete model;

                cout << endl << "End of populateModel3D" << endl << endl;

        }
        catch (const GeoTessException& ex)
        {
                cout << endl << ex.emessage << endl;
                return 1;
```

```
        }
        catch (...)
        {
                cout << endl << "Unidentified error detected " << endl
                        <<  __FILE__ << "  " << __LINE__ << endl;
                return 2;
        }
        return 0;
}
```

### *A.1.2. PopulateModel3D.java*

```
/**
 * Copyright 2009 Sandia Corporation. Under the terms of Contract
 * DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government
 * retains certain rights in this software.
 *
 * BSD Open Source License.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 *    * Redistributions of source code must retain the above copyright notice,
 *      this list of conditions and the following disclaimer.
 *    * Redistributions in binary form must reproduce the above copyright
 *      notice, this list of conditions and the following disclaimer in the
 *      documentation and/or other materials provided with the distribution.
 *    * Neither the name of Sandia National Laboratories nor the names of its
 *      contributors may be used to endorse or promote products derived from
 *      this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
package gov.sandia.geotess.examples;

import java.util.Date;

import gov.sandia.geotess.GeoTessGrid;
import gov.sandia.geotess.GeoTessMetaData;
import gov.sandia.geotess.GeoTessModel;
import gov.sandia.geotess.GeoTessModelUtils;
import gov.sandia.geotessbuilder.GeoTessBuilderMain;
import gov.sandia.gmp.util.globals.DataType;
import gov.sandia.gmp.util.numerical.vector.EarthShape;
import gov.sandia.gmp.util.numerical.vector.VectorGeo;
import gov.sandia.gmp.util.propertiesplus.PropertiesPlus;

/**
 * An example of how to generate a 3D GeoTessModel and populate it with data.
 * <p>The GeoTessGrid is generated by this example.
 * <p>The data used to populate the model come from the ak135 model,
 * which is hardcoded into the source code for the example.
 *
 * @author sballar
 *
 */
public class PopulateModel3D
{
        /**
         * An example of how to generate a 3D GeoTessModel and populate it with data.
```

```
         * The data are stored on a GeoTessGrid comprised of 3 multi-level
         * tessellations, one for the core, one for the mantle and one for the crust.
         * <p>The data used to populate the model come from the ak135 model,
         * which is hardcoded into the source code for the example.
         */
        public static void main(String[] args)
        {
                try
                {
                        System.out.println("Example that illustrates how to populate a 3D model.");
                        System.out.println();

                        new PopulateModel3D().run();

                        System.out.println("Done.");
                }
                catch (Exception ex)
                {
                        ex.printStackTrace();
                }
        }

        protected void run() throws Exception
        {
                // GeoTessModel design motivations:
                // 1 - efficient distribution of vertices in geographic dimensions.
                // 2 - variable resolution in geographic and radial dimensions achieved by specifying
                //     polygons or refining vertex density based on model values.
                // 3 - ability to specify layers with interfaces in between where Snell's
                //     Law can be applied.
                // 4 - layer interfaces can have topography (e.g., variable MOHO depth)
                // 6 - built-in support for ellipsoidal earth (no ellipticity corrections)


                // There are 4 steps involved in building a model:
                //    1 - specify GeoTessMetaData
                //    2 - specify a GeoTessGrid
                //    3 - construct a model from the grid and metadata
                //    4 - populate the model with data

                GeoTessMetaData metaData = getMetaData();

                GeoTessGrid grid = getGrid();

                GeoTessModel model = new GeoTessModel(grid, metaData);

                populateModel(model);

                // At this point, we have a fully functional GeoTessModel object
                // that we can work with.

                // print a bunch of information about the model to the screen.
                System.out.println(model.toString());

                // generate vtk files of the triangle size for each multilevel tessellation of teh grid
                GeoTessModelUtils.vtkRobinsonTriangleSize(grid,
                                "/Users/sballar/Desktop/populatedModel/grid_tesselation_%d.vtk", -105);

                // print a profile for a single vertex, spanning all layers.
                System.out.println(GeoTessModelUtils.profileToString(model, 1));

                // if you want to save the model:
                model.writeModel("/Users/sballar/Desktop/populatedModel/ak135_model.geotess");

                // for a vtk file of the model, uncomment this line:
                GeoTessModelUtils.vtk(model, "/Users/sballar/Desktop/populatedModel/model_%s.vtk", 0,
99, false, null);
        }

        protected GeoTessMetaData getMetaData() throws Exception
        {
                // Create a MetaData object in which we can specify information
                // needed for model construction.
                GeoTessMetaData metaData = new GeoTessMetaData();
```

```
                // Specify a description of the model. This information is not
                // processed in any way by GeoTess. It is carried around for
                // information purposes.
                metaData.setDescription("Simple example of populating a 3D GeoTess model%n" +
                                        "comprised of 3 multi-level tessellations%n"
                                        + "author: Sandy Ballard%n"
                                        + "contact: sballar@sandia.gov");

                // Specify a list of layer names delimited by semi-colons
                metaData.setLayerNames("INNER_CORE; OUTER_CORE; LOWER_MANTLE; " +
                                "TRANSITION_ZONE; UPPER_MANTLE; LOWER_CRUST; UPPER_CRUST");

                // Specify the relationship between grid tessellations and model layers.
                // the list has nLayers elements where each element specifies the
                // index of the multilevel tessellation that supports the
                // corresponding layer.  In this example, the model has
                // 7 layers and 3 multi-level tessellations.  The first
                // tessellation (index 0) supports the core layers (layers
                // 0 and 1).  The second tessellation (index 1) supports the
                // three mantle layers (layer indices 2, 3 and 4).  The third
                // and final tessellation (index 2) supports the two crustal
                // layers (layer indices 5 and 6).
                metaData.setLayerTessIds(new int[] {0, 0, 1, 1, 1, 2, 2});

                // specify the names of the attributes and the units of the
                // attributes in two Strings delimited by semi-colons.
                metaData.setAttributes("Vp; Vs; rho", "km/sec; km/sec; g/cc");

                // specify the DataType for the data. All attributes, in all
                // profiles, will have the same data type.  Note that this
                // applies only to the data; radii are always stored as floats.
                metaData.setDataType(DataType.FLOAT);

                // specify the name of the software that is generating
                // the model.  This gets stored in the model for future reference.
                metaData.setModelSoftwareVersion(this.getClass().getCanonicalName());

                // specify the date when the model was generated.  This gets
                // stored in the model for future reference.
                metaData.setModelGenerationDate(new Date().toString());

                // specify the ellipsoid used to convert lat-lon to unit vectors
                // and used to convert between depth and radius as a function of
                // latitude.  (Optional. Defaults to WGS84).
                metaData.setEarthShape(EarthShape.WGS84);

                return metaData;
        }

        protected GeoTessGrid getGrid() throws Exception
        {
                PropertiesPlus properties = new PropertiesPlus();

                // build a new grid from scratch as opposed to refining an existing grid.
                properties.setProperty("gridConstructionMode = scratch");

                // do it silently
                properties.setProperty("verbosity = 0");

                // build a single grid with 3 multilevel tessellations
                properties.setProperty("nTessellations = 3");

                // specify the base edge length of the triangles on each mulilevel tessellation, in
degrees
                properties.setProperty("baseEdgeLengths = 32 16 8");

                // specify resolution that varies geographically by specifying a polygon
                // defined by a spherical cap centered on albuquerque, nm, with a horizontal
                // radius of 30 degrees.  Only the crustal layers are refined (tessellation index 2).
                // Triangle size inside the polygon is 2 degrees.
                // For more complicated polygons, see the GeoTess User's Manual.
                properties.setProperty("polygons = spherical_cap, 35.12, -106.62, 10., 2, 2");
```

```
        // build the grid
        GeoTessGrid grid = (GeoTessGrid) GeoTessBuilderMain.run(properties);

        return grid;
}

protected void populateModel(GeoTessModel model) throws Exception
{
        // Now we will populate the model with Profiles.  At this point, the
        // model has a 2D array of Profile objects with dimensions
        // nVertices x nLayers with all the elements of the array initialized
        // to null.  We will now populate the Profiles array.
        //
        // now loop over every vertex in the grid, connected and not-connected.
        for (int vertex = 0; vertex < model.getNVertices(); ++vertex)
        {
                // retrieve the unit vector corresponding to the i'th vertex
                double[] unit_vector = model.getVertex(vertex);

                // find the latitude and longitude of vertex, in degrees
                double lat = VectorGeo.getLatDegrees(unit_vector);
                double lon = VectorGeo.getLonDegrees(unit_vector);

                // loop over the 7 layers of the model (inner_core, outer_core, etc)
                for (int layer=0; layer<model.getNLayers(); ++layer)
                {
                        // get the profile radii for the current lat, lon, and layer.
                        // This is a 1D array of radius values where the first radius is
                        // the radius at the bottom of the layer, the last radius is the
                        // radius of the top of the layer and there can be as many in between
                        // as desired.  This function will be different for every real-life
                        // application.
                        float[] radii = getAK135Radii(lat, lon, layer);

                        // get the profile data values for current lat, lon, layer.
                        // This is a 2D array of values, nNodes by nAttributes.
                        // In this example, nAttributes is 3 corresponding to
                        // attributes vp, vs and rho.  The number of nodes can
                        // vary as a function of vertex and layer.
                        float[][] data = getAK135Values(lat, lon, layer);

                        // send the radii and data to the model and it will construct
                        // the correct type of Profile: NPOINT, CONSTANT, THIN or EMPTY
                        model.setProfile(vertex, layer, radii, data);
                }
        }

        // update the PointMap.
        model.setActiveRegion();

        // test the model to make sure that radius at top of layer i == radius at bottom of
        // layer i+1.  Small discrepancies are repaired, large ones cause an exception.
        model.testModelIntegrity();

        // model is now fully populated.

}

/**
 * Returns a 1D profile of monotonically increasing radius values
 * that define the radial positions of nodes along a radial
 * profile through a single layer in the model.
 * <p>
 * For this example, we will return the radius positions of the
 * nodes in the AK135 model, stretched a little bit so that
 * the top of the model will coincide with the radius of the
 * WGS84 ellipsoid instead of the ak135 value of 6371 km.
 * @param lat the latitude of the profile in degrees
 * @param lon the longitude of the profile in degrees
 * @param layer the index of the layer.
 * @return 1D array of radius values, in km.
 * @throws Exception
 */
protected static float[] getAK135Radii(double lat, double lon, int layer) throws Exception
```

```
{
        // convert lat, lon in degrees to unit vector.
        double[] vertex = EarthShape.WGS84.getVectorDegrees(lat, lon);

        // find the radius of the WGS84 ellipsoid at the latitude of vertex.
        float earthRadius = (float) EarthShape.WGS84.getEarthRadius(vertex);

        // find a stretching factor that will stretch the radius values so that they
        // span the range from zero at the center of the earth to the radius of the
        // WGS84 ellipsoid at the surface of the Earth.  This is not geophysically
        // realistic, but sufficient for this simplistic example.
        float stretch = earthRadius / 6371F;

        float r0, rn;

        // There are 7 layers corresponding to the inner core,
        // outer core, lower mantle, transition zone, upper mantle,
        // lower crust and upper crust.
        if (layer < 2)
        {
                // inner and outer core
                r0 = ak135[layer][0][0] * stretch;
                rn = ak135[layer][ak135[layer].length-1][0] * stretch;
        }
        else if (layer == 2)
        {
                // lower mantle
                r0 = ak135[layer][0][0] * stretch;
                rn = earthRadius - 660F;
        }
        else if (layer == 3)
        {
                // transition zone
                r0 = earthRadius - 660F;
                rn = earthRadius - 410F;
        }
        else if (layer == 4)
        {
                // upper mantle
                r0 = earthRadius - 410F;
                rn = earthRadius - 35F;
        }
        else if (layer == 5)
        {
                // lower crust
                r0 = earthRadius - 35F;
                rn = earthRadius - 20F;
        }
        else if (layer == 6)
        {
                // upper crust
                r0 = earthRadius - 20F;
                rn = earthRadius;
        }
        else
                throw new Exception(String.format("%d is not a recognized layer index."
                                + "The ak135 model has only 7 layers.", layer));

        // copy the ak135 radii for current layer into a new array of floats.
        float[] radii = new float[ak135[layer].length];
        for (int i=0; i<radii.length; ++i)
                radii[i] = ak135[layer][i][0];

        // stretch the radii so that they span radius range r0 to rn
        return stretch(radii, r0, rn);
}

private static float[] stretch(float[] radii, float r0, float rn)
{
        float scale = (rn-r0)/(radii[radii.length-1]-radii[0]);

        float[] stretchedRadii = new float[radii.length];

        for (int i=0; i<radii.length; ++i)
```

```
                stretchedRadii[i] = r0 + (radii[i]-radii[0])*scale;

        return stretchedRadii;
}

/**
 * Retrieve a 2D array of floats with nNodes x nAttributes elements.
 * The number of attributes is 3: [vp, vs, rho].  nNodes
 * varies in the different layers.  For core and mantle layers,
 * nNodes will equal the number of radii in the corresponding
 * layers of the AK135 model. For the crustal layers, nNodes
 * will be one, reflecting the fact that the attribute values are
 * constant in the crustal layers of the ak135 model.
 * <p>
 * In this example, the data returned are independent of latitude and
 * longitude since ak135 is a '1D' model, but this will not generally
 * be true for real 3D models.
 * @param lat
 * @param lon
 * @param layer
 * @return
 */
protected static float[][] getAK135Values(double lat, double lon, int layer)
{
        // get a reference to the ak135 model values for the layer of interest.
        float[][] ak135Layer = ak135[layer];

        // nNodes is the number of radial positions defined in this layer in the
        // ak135 model.
        int nNodes = ak135Layer.length;

        // data is the nNodes x nAttributes array that will be returned.
        float[][] data = null;

        if (nNodes == 2 && ak135Layer[0][1] == ak135Layer[1][1])
        {
                // this layer consists of only two nodes and the vp values are equal,
                // which is true for the two crustal layers. Make a float[1][3] array
                // containing the values from only the first node.
                data = new float[1][3];
                data[0][0] = ak135Layer[0][1];
                data[0][1] = ak135Layer[0][2];
                data[0][2] = ak135Layer[0][3];
        }
        else
        {
                // this layer has more than 2 nodes or maybe it has two
                // nodes but the vp values are not equal.  Make an nNodes
                // by 3 array of data values.
                data = new float[nNodes][3];
                for (int i=0; i<nNodes; ++i)
                {
                        data[i][0] = ak135Layer[i][1];
                        data[i][1] = ak135Layer[i][2];
                        data[i][2] = ak135Layer[i][3];
                }
        }
        return data;
}

/**
 * A 3D array of floats with nLayers x nNodes x nAttributes+1
 * elements.  The attributes are radius in km,
 * vp in km/sec, vs in km/sec and density in g/cc.
 * There are 7 layers corresponding to the inner core,
 * outer core, lower mantle, transition zone, upper mantle,
 * lower crust and upper crust.  The number of nodes in
 * each layer is variable by layer.
 */
static float[][][] ak135 = new float[][][] {{
        // inner core:
        // radius      vp          vs       density
        {   0.000F,  11.2622F,   3.6678F,  13.0122F},
        {  50.710F,  11.2618F,   3.6675F,  13.0117F},
```

```
        { 101.430F,  11.2606F,   3.6667F,  13.0100F},
        { 152.140F,  11.2586F,   3.6653F,  13.0074F},
        { 202.850F,  11.2557F,   3.6633F,  13.0036F},
        { 253.560F,  11.2521F,   3.6608F,  12.9988F},
        { 304.280F,  11.2477F,   3.6577F,  12.9929F},
        { 354.990F,  11.2424F,   3.6540F,  12.9859F},
        { 405.700F,  11.2364F,   3.6498F,  12.9779F},
        { 456.410F,  11.2295F,   3.6450F,  12.9688F},
        { 507.130F,  11.2219F,   3.6396F,  12.9586F},
        { 557.840F,  11.2134F,   3.6337F,  12.9474F},
        { 659.260F,  11.1941F,   3.6202F,  12.9217F},
        { 709.980F,  11.1832F,   3.6126F,  12.9072F},
        { 760.690F,  11.1715F,   3.6044F,  12.8917F},
        { 811.400F,  11.1590F,   3.5957F,  12.8751F},
        { 862.110F,  11.1457F,   3.5864F,  12.8574F},
        { 912.830F,  11.1316F,   3.5765F,  12.8387F},
        { 963.540F,  11.1166F,   3.5661F,  12.8188F},
        {1014.250F,  11.0983F,   3.5551F,  12.7980F},
        {1064.960F,  11.0850F,   3.5435F,  12.7760F},
        {1115.680F,  11.0718F,   3.5314F,  12.7530F},
        {1166.390F,  11.0585F,   3.5187F,  12.7289F},
        {1217.500F,  11.0427F,   3.5043F,  12.7037F}
},
        {
        // outer core:
        // radius      vp          vs       density
        {1217.500F,  10.2890F,   0.0000F,  12.1391F},
        {1267.430F,  10.2854F,   0.0000F,  12.1133F},
        {1317.760F,  10.2745F,   0.0000F,  12.0867F},
        {1368.090F,  10.2565F,   0.0000F,  12.0593F},
        {1418.420F,  10.2329F,   0.0000F,  12.0311F},
        {1468.760F,  10.2049F,   0.0000F,  12.0001F},
        {1519.090F,  10.1739F,   0.0000F,  11.9722F},
        {1569.420F,  10.1415F,   0.0000F,  11.9414F},
        {1670.080F,  10.0768F,   0.0000F,  11.8772F},
        {1720.410F,  10.0439F,   0.0000F,  11.8437F},
        {1770.740F,  10.0103F,   0.0000F,  11.8092F},
        {1821.070F,   9.9761F,   0.0000F,  11.7737F},
        {1871.400F,   9.9410F,   0.0000F,  11.7373F},
        {1921.740F,   9.9051F,   0.0000F,  11.6998F},
        {1972.070F,   9.8682F,   0.0000F,  11.6612F},
        {2022.400F,   9.8304F,   0.0000F,  11.6216F},
        {2072.730F,   9.7914F,   0.0000F,  11.5809F},
        {2123.060F,   9.7513F,   0.0000F,  11.5391F},
        {2173.390F,   9.7100F,   0.0000F,  11.4962F},
        {2223.720F,   9.6673F,   0.0000F,  11.4521F},
        {2274.050F,   9.6232F,   0.0000F,  11.4069F},
        {2324.380F,   9.5777F,   0.0000F,  11.3604F},
        {2374.720F,   9.5306F,   0.0000F,  11.3127F},
        {2425.050F,   9.4814F,   0.0000F,  11.2639F},
        {2475.380F,   9.4297F,   0.0000F,  11.2137F},
        {2525.710F,   9.3760F,   0.0000F,  11.1623F},
        {2576.040F,   9.3205F,   0.0000F,  11.1095F},
        {2626.370F,   9.2634F,   0.0000F,  11.0555F},
        {2676.700F,   9.2042F,   0.0000F,  11.0001F},
        {2727.030F,   9.1426F,   0.0000F,  10.9434F},
        {2777.360F,   9.0792F,   0.0000F,  10.8852F},
        {2827.700F,   9.0138F,   0.0000F,  10.8257F},
        {2878.030F,   8.9461F,   0.0000F,  10.7647F},
        {2928.360F,   8.8761F,   0.0000F,  10.7023F},
        {2978.690F,   8.8036F,   0.0000F,  10.6385F},
        {3029.020F,   8.7283F,   0.0000F,  10.5731F},
        {3079.350F,   8.6496F,   0.0000F,  10.5062F},
        {3129.680F,   8.5692F,   0.0000F,  10.4378F},
        {3180.010F,   8.4861F,   0.0000F,  10.3679F},
        {3230.340F,   8.4001F,   0.0000F,  10.2964F},
        {3280.680F,   8.3122F,   0.0000F,  10.2233F},
        {3331.010F,   8.2213F,   0.0000F,  10.1485F},
        {3381.340F,   8.1283F,   0.0000F,  10.0722F},
        {3431.670F,   8.0382F,   0.0000F,   9.9942F},
        {3479.500F,   8.0000F,   0.0000F,   9.9145F}
        },
        {
                // lower mantle:
```

```
                        // radius      vp          vs       density
                        {3479.500F,  13.6602F,   7.2811F,   5.5515F},
                        {3531.670F,  13.6566F,   7.2704F,   5.5284F},
                        {3581.330F,  13.6530F,   7.2597F,   5.5051F},
                        {3631.000F,  13.6494F,   7.2490F,   5.4817F},
                        {3681.000F,  13.5900F,   7.2258F,   5.4582F},
                        {3731.000F,  13.5312F,   7.2031F,   5.4345F},
                        {3779.500F,  13.4741F,   7.1807F,   5.4108F},
                        {3829.000F,  13.4156F,   7.1586F,   5.3869F},
                        {3878.500F,  13.3585F,   7.1369F,   5.3628F},
                        {3928.000F,  13.3018F,   7.1144F,   5.3386F},
                        {3977.500F,  13.2465F,   7.0931F,   5.3142F},
                        {4027.000F,  13.1894F,   7.0720F,   5.2898F},
                        {4076.500F,  13.1336F,   7.0500F,   5.2651F},
                        {4126.000F,  13.0783F,   7.0281F,   5.2403F},
                        {4175.500F,  13.0222F,   7.0063F,   5.2154F},
                        {4225.000F,  12.9668F,   6.9855F,   5.1904F},
                        {4274.500F,  12.9096F,   6.9627F,   5.1652F},
                        {4324.000F,  12.8526F,   6.9418F,   5.1398F},
                        {4373.500F,  12.7956F,   6.9194F,   5.1143F},
                        {4423.000F,  12.7382F,   6.8972F,   5.0887F},
                        {4472.500F,  12.6804F,   6.8742F,   5.0629F},
                        {4522.000F,  12.6221F,   6.8515F,   5.0370F},
                        {4571.500F,  12.5631F,   6.8286F,   5.0109F},
                        {4621.000F,  12.5031F,   6.8052F,   4.9847F},
                        {4670.500F,  12.4426F,   6.7815F,   4.9584F},
                        {4720.000F,  12.3819F,   6.7573F,   4.9319F},
                        {4769.500F,  12.3185F,   6.7326F,   4.9052F},
                        {4819.000F,  12.2550F,   6.7073F,   4.8785F},
                        {4868.500F,  12.1912F,   6.6815F,   4.8515F},
                        {4918.000F,  12.1245F,   6.6555F,   4.8245F},
                        {4967.500F,  12.0577F,   6.6285F,   4.7973F},
                        {5017.000F,  11.9895F,   6.6008F,   4.7699F},
                        {5066.500F,  11.9200F,   6.5727F,   4.7424F},
                        {5116.000F,  11.8491F,   6.5439F,   4.7148F},
                        {5165.500F,  11.7766F,   6.5138F,   4.6870F},
                        {5215.000F,  11.7026F,   6.4828F,   4.6591F},
                        {5264.500F,  11.6269F,   6.4510F,   4.6310F},
                        {5314.000F,  11.5495F,   6.4187F,   4.6028F},
                        {5363.500F,  11.4705F,   6.3854F,   4.5744F},
                        {5413.000F,  11.3896F,   6.3512F,   4.5459F},
                        {5462.500F,  11.3068F,   6.3160F,   4.5173F},
                        {5512.000F,  11.2221F,   6.2798F,   4.4885F},
                        {5561.500F,  11.1353F,   6.2426F,   4.4596F},
                        {5611.000F,  10.0558F,   6.2095F,   4.4305F},
                        {5661.000F,  10.9229F,   6.0897F,   4.4010F},
                        {5711.000F,  10.7900F,   5.9600F,   4.3714F}
                },
                {
                        // transition zone:
                        // radius      vp          vs       density
                        {5711.000F,  10.2000F,   5.6100F,   4.0646F},
                        {5761.000F,  10.0320F,   5.5040F,   4.0028F},
                        {5811.000F,   9.8640F,   5.3980F,   3.9410F},
                        {5861.000F,   9.6960F,   5.2920F,   3.8793F},
                        {5911.000F,   9.5280F,   5.1860F,   3.8175F},
                        {5961.000F,   9.3600F,   5.0800F,   3.7557F}
                },
                {
                        // upper mantle:
                        // radius      vp          vs       density
                        {5961.000F,   9.0300F,   4.8700F,   3.5470F},
                        {6011.000F,   8.8475F,   4.7830F,   3.5167F},
                        {6061.000F,   8.6650F,   4.6960F,   3.4864F},
                        {6111.000F,   8.4825F,   4.6090F,   3.4561F},
                        {6161.000F,   8.3000F,   4.5230F,   3.4258F},
                        //                              {6161.000F,   8.3000F,   4.5180F,   3.4258F},
// ignore small S discontinuity at 210 km depth
                        {6206.000F,   8.1750F,   4.5090F,   3.3985F},
                        {6251.000F,   8.0500F,   4.5000F,   3.3713F},
                        {6293.500F,   8.0450F,   4.4900F,   3.3455F},
                        {6336.000F,   8.0400F,   4.4800F,   3.3198F}
                },
                {
```

```
                        // lower crust:
                        // radius      vp          vs       density
                        {6336.000F,   6.5000F,   3.8500F,   2.9200F},
                        {6351.000F,   6.5000F,   3.8500F,   2.9200F}
                },
                {
                        // upper crust:
                        // radius      vp          vs       density
                        {6351.000F,   5.8000F,   3.4600F,   2.7200F},
                        {6371.000F,   5.8000F,   3.4600F,   2.7200F}}};
}
```

# APPENDIX B. MANIPULATION OF GEOGRAPHIC LOCATIONS ON AN ELLIPSOIDAL EARTH

To manipulate points on or near the surface of the Earth, it is convenient to work in a Cartesian coordinate system where points are defined by a unit vector, **v**, with its origin at the center of the Earth, and a radial distance from the center of the Earth, *r*, measured in km. We choose our coordinate system such that $v_0$ points from the center of the Earth towards the point on the surface with latitude and longitude 0°, 0°; $v_1$ points toward latitude, longitude 0°, 90° and $v_2$ points toward the north pole (Figure 6-2).

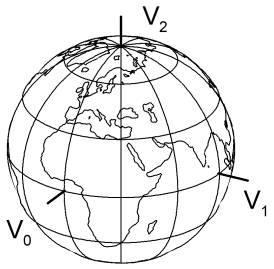


**Figure 6-2. Earth centered coordinate system. $v_0$ points from the center of the Earth towards the point on the surface with latitude and longitude 0°, 0°; $v_1$ points toward latitude, longitude 0°, 90° and $v_2$ points toward the north pole.**

The parameters that define the GRS80 ellipsoid are:

$$a = 6378.137 \text{ km}$$
$$b = 6356.7523141 \text{ km}$$
$$f = \frac{a-b}{a} = 1/298.257222101$$
$$e^2 = \frac{a^2-b^2}{a^2} = 2f - f^2 = 0.006694380022900787$$

where *a* and *b* are the equatorial and polar radii of the Earth, respectively, *f* is the flattening parameter, and *e* is the eccentricity. Any two of these parameters are sufficient to completely define the ellipsoid. In the equations presented in this paper *a* and $e^2$ are used (Snyder, 1987).

Geographic data, such as station locations, seismic event locations, etc., are generally given in geographic latitude, $\phi'$, longitude, $\theta'$, and depth, $z$. Geographic latitude is the acute angle between the equatorial plane and a line drawn perpendicular to the tangent of the reference ellipsoid at the point of interest (**Error! Reference source not found.**). Geodesic latitude is another term for geographic latitude. Geocentric latitude is the acute angle between the equatorial plane and a line from the center of the Earth to the point in question. Geographic, geodesic, and geocentric longitudes are all equivalent.

To convert the position of a point in space from geographic to Cartesian coordinates, we must first convert from geographic to geocentric coordinates. Given the geographic latitude $\phi'$, and geographic longitude $\theta'$, of a point, the geocentric latitude, $\phi$, and geocentric longitude, $\theta$, are (Snyder, 1987)

$$\begin{aligned} \phi &= \arctan((1-e^2)\tan\phi') \\ \theta &= \theta' \end{aligned} \tag{1}$$

Then we convert from geocentric to Cartesian coordinates (Zwillinger, 2003)

$$\begin{aligned} v_0 &= \cos\phi\cos\theta \\ v_1 &= \cos\phi\sin\theta \\ v_2 &= \sin\phi \end{aligned} \tag{2}$$

We must also convert depth, $z$, to radius

$$r = R(\phi) - z \tag{3}$$

where $R(\phi)$, the radius of the Earth at geocentric latitude $\phi$, is given by

$$R(\phi) = a\left(1 + \frac{e^2}{1-e^2}\sin^2\phi\right)^{-1/2} \tag{4}$$

To convert a unit vector, v, and radius $r$ back to geographic latitude, longitude and depth

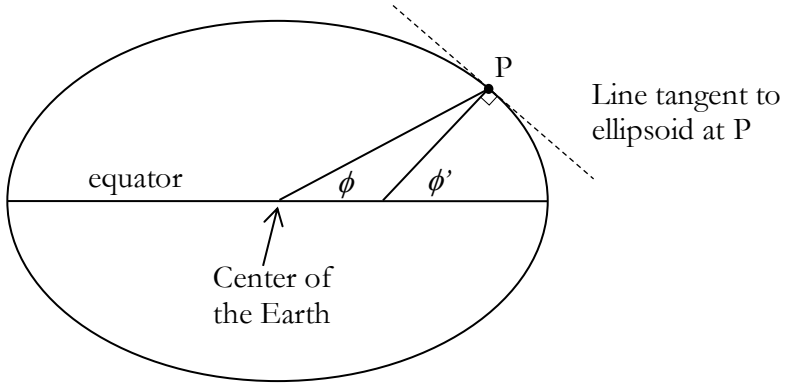


**Figure 6-3. An exaggerated ellipsoid illustrating the difference between geographic latitude, $\varphi'$, and geocentric latitude, $\varphi$.**

$$\begin{aligned} \phi' &= \arctan\left(\frac{\tan(\arcsin v_2)}{1-e^2}\right) \\ \theta' &= \arctan\left(v_1 / v_0\right) \\ z &= R(\phi) - r \end{aligned} \tag{5}$$

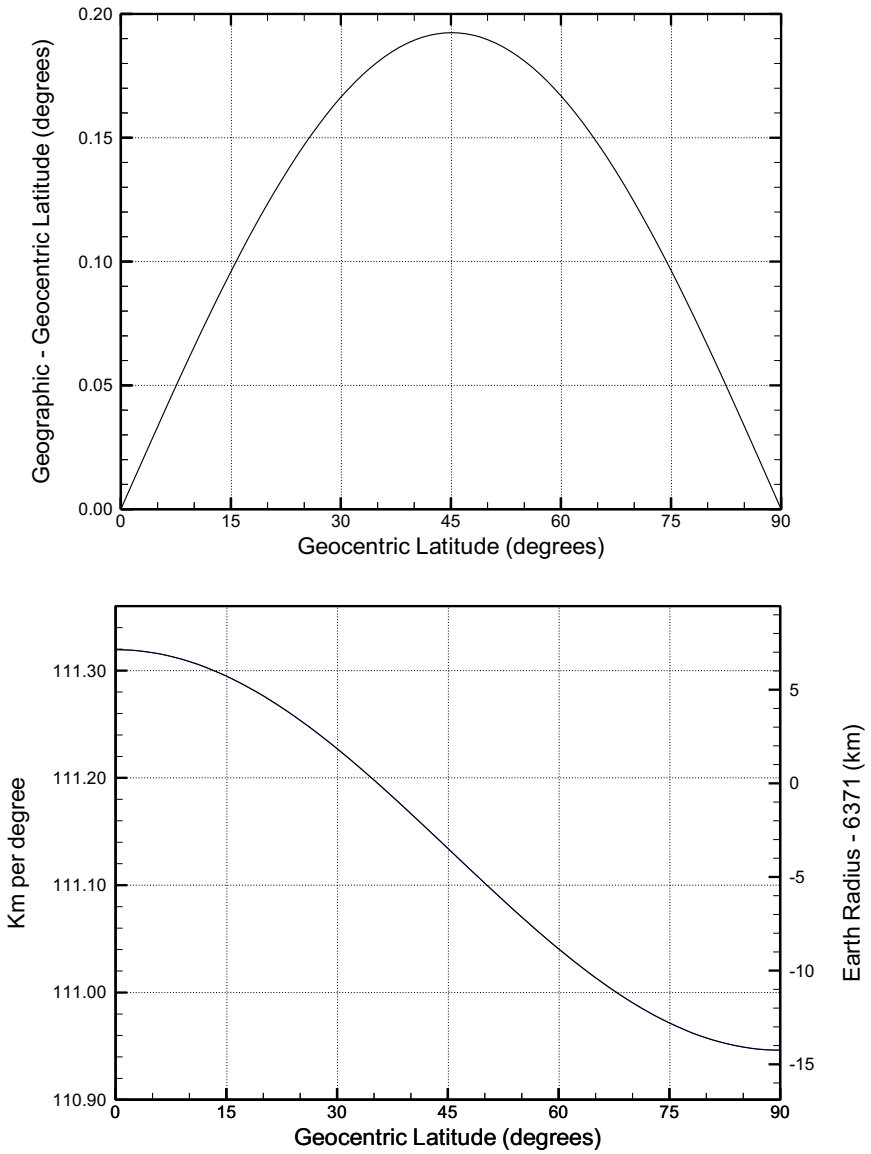


**Figure 6-4. (top) Comparison of geographic and geocentric latitudes for the GRS80 ellipsoid. (bottom) km per degree and Earth radius as a function of geocentric latitude.**

Once geographic information has been converted to unit vectors, a variety of useful calculations can be performed.

## 6.1. Distance between two points

Given two points defined by unit vectors, u and v, the angular separation of the two points is

$$\Delta = \arccos(\mathbf{u} \cdot \mathbf{v}) \tag{6}$$

To find the separation of u and v at the surface of the Earth in km, it is necessary to either perform the following integration numerically

$$d = \int_{\mathbf{u}}^{\mathbf{v}} R(\phi) d\delta \tag{7}$$

or consider the algorithm of Vincenty (1975).

### *B.1.1. Azimuth from one point to another*

The azimuth, $\alpha$, from u to v, measured clockwise from north, is

$$\begin{aligned} &\alpha = \arccos\left(|\mathbf{u}\times\mathbf{v}|\cdot|\mathbf{u}\times\mathbf{n}|\right) \\ &if\left(|\mathbf{u}\times\mathbf{v}|\cdot\mathbf{n} < 0\right) \qquad \alpha = 2\pi - \alpha \end{aligned} \tag{8}$$

where n is the vector pointing to the north pole, n = [0, 0, 1].

### *B.1.2. Points on a great circle*

Given two points defined by unit vectors u and v, to find another point, w, that lies on the great circle defined by u and v, at some angular distance $\delta$, measured from u in the direction of v (see **Error! Reference source not found.**)

$$\begin{aligned} &\mathbf{t} = |(\mathbf{u}\times\mathbf{v})\times\mathbf{u}| \\ &\mathbf{w} = \mathbf{u}\cos\delta + \mathbf{t}\sin\delta \end{aligned} \tag{9}$$

Note that if many points, $w_i$, are to be found along the same great circle defined by u and v, the normalized vector triple product, t, only needs to be computed once.

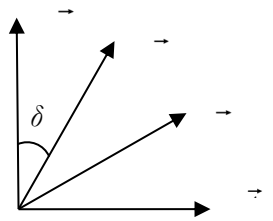


**Figure 6-5. Calculation of w given u and v. All vectors are of unit length and lie entirely in the plane of the figure.**

### *B.1.3. Finding a new point some distance and azimuth from another point*

To find a point, w, that is some specified distance $\delta$ from p in direction $\varphi$, (see **Error! Reference source not found.**) we first find an intermediate point u, distance $\delta$ north of p by applying Equation 9 with v = n = [0, 0, 1]. Then w is found by rotating u around p by angle $\alpha = -\varphi$.

$$\mathbf{w} = \mathbf{u}\cos\alpha + \mathbf{p}(\mathbf{p}\cdot\mathbf{u})(1-\cos\alpha) + (\mathbf{p}\times\mathbf{u})\sin\alpha \tag{10}$$

α is equal to $-\varphi$ because rotations defined by equation 10 are positive clockwise when viewed in the direction of the pole of rotation p.

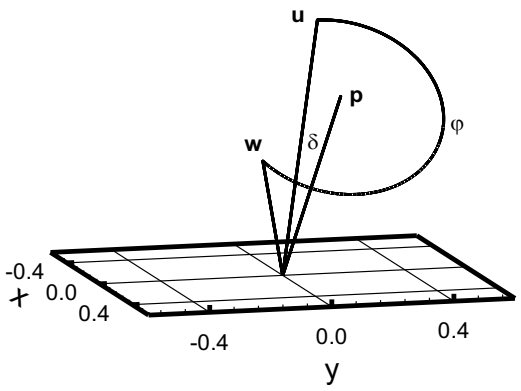


**Figure 6-6. Start at point p, located at latitude, longitude 45°, 0°. Find a new point u δ = 20° north of p. Then rotate u φ = 235° around p to position w. Note that ∠pu = ∠pw = δ = 20°.**

# APPENDIX C. C SHELL

This section will detail how and why the C shell was constructed in the way it was. The main idea was to use the C++ implementation for the heavy lifting. Re-implementing the library in C would be unreasonable and as a result the shell (also known as an interface) is very light.

The C Shell is implemented in C++ in such a way that pure C code can access the shared library without any notion that it is C++.

NOTE: Any C Shell files that start with an underscore are not to be used by users of the C Shell. They are private and for implementation uses only.

## C.1. Headers

There are a few points that are common within each header file in the C Shell. First off, the header files must be 100% C code, no C++. The only caveat is the usage of macros:

```
#ifdef __cplusplus
extern "C"
{
#endif

<body of header file>

#ifdef __cplusplus
}
#endif
```

This is so that the C++ code implementing the C Shell can understand the header files and when imported by pure C code it can understand them as well.

The second point to notice is the usage of the "GEO_TESS_EXPORT_C" definition used on every public function. This is defined in "GeoTessCShellGlobals.h". This is used on Windows operating systems to define "__declspec(dllimport/dllexport)" which is used when creating dlls. This will define the functions that need to be imported from a dll, or exported into a dll upon compile time. On other platforms this is defined as either "extern" or nothing. Extern isn't needed, but is used for future extension.

### *C.1.1. Naming Conventions*

- For the C wrappers of GeoTess objects they are named the same thing as the base object with a "C" appended to the name – GeoTessModel -> GeoTessModelC. The same is done for the files implementing the objects.

- The functions of each object keep the same base name as the C++ variants, but place a shortened version of the object name on the front separated by an underscore. GeoTessModel.writeModel() -> geomodel_writeModel(). This is done to avoid symbolic conflicts with the C++ library. In the case where a function is overridden, a number starting from 1 is appended to the name to avoid symbolic conflicts internally.

### C.1.2. C Shell Source

The basic form of every function in the C Shell is to extract the C++ object from the given C wrapper, call the appropriate method with the given arguments, and return the result watching for any exceptions. Results may need to be converted from C++ objects to C friendly ones first. The main issues with simply providing a wrapper to the C++ library are: exception handling, data structures, and GeoTess objects.

### C.1.3. Exception Handling

Clearly C has no notion of exceptions. To get around this limitation, every call to a GeoTess function in the implementation is wrapped in a try-catch block. Note that this implementation does increase the time for each call to a GeoTess function. Once each exception is caught, it is then placed into a C data structure made for keeping a short log of exceptions. This is the ErrorCache object. It is basically a stack that keeps the 19 latest error messages. This object will be given the error message of the thrown exception and can be asked if it has any messages stored. The error messages can then be popped off and used by the user of the C Shell. The interface for this object is public to the C users.

There are two blocks in the catch statement of every try-catch. One for the GeoTessException, which has should have a detailed explanation of the problem, and "…" meaning everything else. In the last case, a string meaning of the exception can't be concluded so one is generated with the file and line number where it was caught.

All GeoTess wrappers contain a reference to an ErrorCache, in fact to the same one. The constructor for the ErrorCache is a singleton and it keeps a reference count to know when to delete the memory.

### C.1.4. Data Structures

Data Structures such as Vectors or Maps in C++ don't have a representation in the C standard library. They must be converted into basic arrays for C users. Most cases go from String to char* and from Vectors to arrays. More complicated forms are Maps to dual arrays of keys and values. The implementations of these conversions are in "_Util.cc".

Another type of conversion needed is the various enums in the C++ library to the int based enums in C. The method used here was to create the C enums and have the elements listed in the same order as the C++ enums. From there, conversion between the two is easy. The C++ enums are put into an array, so the int value of the C enums are used as the index of that array. Starting with the C++ enums, their index in the array is found and that index is type cast to the type of C enum desired.

### C.1.5. GeoTess Objects

The various objects that GeoTess provides also don't have a C representation. A struct for each object type is created with two pointers: void* and ErrorCache*. The void* points to the actual C++ object, but hides the type allowing C code to use it. When this struct is given to the C Shell for use as an object, the shell pulls the actual C++ object out of the void* and uses that. This keeps the interface thin and easy to use.

The implementations of the constructors for the various GeoTess objects always create the wrapper since this includes the ErrorCache with it. If an exception is thrown in the library call to the constructor, then the wrapper is returned with a null void* and the error cache holding the exception.

The downside of using the wrappers is that sometimes a GeoTess C++ object can be associated with another C++ object in the library so it should not be deleted, yet your use of the wrapper has come to an end. The C Shell destructors of the wrappers allow users to select whether they want just the wrapper freed or the wrapper and the underlying C++ object. The C header files should document when it is safe and when it is not safe to fully delete a wrapper.

# APPENDIX D. FILE FORMATS

GeoTess models are stored in files in two different formats: ASCII and binary. It is possible to convert between these two formats using functionality provided in the GeoTess software.

GeoTess model information can be logically divided into 3 sections: metadata, profiles, and grid. This information can be stored in either one or two files. If the model is stored in a single file then the three components are written to the file in the order specified. It is also possible to write the metadata and profile information to one file and the grid to another. In the latter case, the file with the metadata and the profiles also includes a reference to the file that contains the grid information.

## D.1. Binary Format

In this section, the format of GeoTess binary model and grid files is described.

### *D.1.1. Binary Model File*

**Table 6-1. Description of Binary Model Format Parameters.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| File identification string | The 12 characters: GEOTESSMODEL | 12 character string NOT preceded by integer string length |
| Model file format version number | Model file format version number in range 1 to 65535. The two least significant bytes store the version number and the two most significant bytes are zero. This value is also used to determine if the file is stored in big-endian or little-endian format. | 4-byte integer in range of 1 to 65535 |
| Software version | The name of the software that was used to generate the content of the model, and its version number | Integer length of string followed by string. |
| Date | The date that the content of the model was generated | Integer length of string followed by string. |
| EarthShape | File format version 2 or greater, only. The name of the ellipsoid used by GeoTess. Valid options are SPHERE, GRS80, GRS80_RCONST, WGS84, WGS84_RCONST, IERS, IERS_RCONST. This parameter was not present in model file format 1 (WGS84 was assumed). | Integer length of string followed by string. |
| Model description | Model description. | Integer length of string followed by string. |

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Attribute names | A list of the names of all the attributes stored in the model, separated by semi-colons. For example: 'vp; vs; density'. *nAttributes* is the number of attributes specified. | Integer length of string followed by string. |
| Attribute units | The units of the defined attributes, separated by semi-colons. For example: 'km/sec; km/sec; g/cc'. The number of entries must be equal to *nAttributes*. | Integer length of string followed by string. |
| Layer names | The names of all the layers that define the model, separated by semi-colons and listed in order of increasing radius. For example: 'core; mantle; crust'. *nLayers* is the number of layer names specified. | Integer length of string followed by string. |
| Data Object type | The type of the Data objects stored in this model. Must be one of DOUBLE, FLOAT, LONG, INT, SHORT or BYTE. | Integer length of string followed by string. |
| nVertices | Number of vertices defined in the grid. | Integer |
| Layer index – tessellation index map. | An integer for each layer in the model specifying the index of the multi-layer tessellation that supports that layer. | *nLayers* integers. |
| Profile objects | A Profile object for each layer at each vertex in the model. See section Profiles for Profile definitions. | *nVertices* * *nLayers* Profile objects. Layer index varies fastest. Profiles associated with the same vertex are listed in order that increases with radius. |
| Grid file specifier | String specifying the file in which the grid information is stored. If the grid file specifier is the single character '*', then the grid information is stored in the same file as the model data, immediately following the gridID. Otherwise, the grid file specifier indicates the name of the file that contains the grid information. | Integer length of string followed by string. |
| gridID | Every grid has a unique gridID that is stored in both the grid file | Integer length of string followed by string. |

| Parameter Name | Definition | Binary Value |
|---|---|---|
| | and in all the model files that use that grid. When the model and grid are loaded, a check is performed to ensure that the two gridIDs match exactly. While any string can be used as a gridID, an MD5 hash of the vertices, triangle indices, level indices and tessellation indices is an excellent choice. | |

### *D.1.2. Binary Profile Objects*

**Table 6-2. ProfileEmpty – Profile Object Consisting of a Bottom and Top Radius but no Data.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Profile type index | ProfileEmpty objects have index 0 | Byte 0 |
| radiusBottom | Radius at the bottom of the profile, in km | Float |
| radiusTop | Radius at the top of the profile, in km | Float |

**Table 6-3. ProfileThin – Profile Object that Represents a Zero-Thickness Profile.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Profile type index | ProfileThin objects have index 1 | Byte 1 |
| Radius | Radius of the profile, in km. | Float |
| Data | Data object associated with this profile | Data object |

**Table 6-4. ProfileConstant – A Finite Thickness Profile Characterized by a Single Data Object.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Profile type index | ProfileConstant objects have index 2 | Byte 2 |
| radiusBottom | Radius at the bottom of the profile, in km | Float |
| radiusTop | Radius at the top of the profile, in km | Float |
| Data | Data object associated with this profile | Data object |

**Table 6-5. ProfileNPoints – A Profile Object Comprised of Two or More Radii and an Equal Number of Data Objects.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Profile type index | ProfileNPoints objects have index 3 | Byte 3 |
| nNodes | Number of nodes on profile | Integer |
| Radius values and Data objects | Radius values and Data objects | Float, followed by a Data object. This combination is repeated *nNodes* times. |

**Table 6-6. ProfileSurface – A Profile Object that Represents Data, but no Radius**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Profile type index | ProfileSurface objects have index 4 | Byte 4 |
| Data | Data object associated with this profile | Data object |

### *D.1.3. Binary Data Objects*

Data Objects consist of a 1D array of numeric values, where all of the values are of type double, float, long, int, short or byte.

All Data Objects in the model must be of the same type and must have the same number of elements. The number of elements in every Data Object must be equal to *nAttributes*, which is the number 'attribute names' specified in the file. Whenever a Data Object is specified in the file format specification sections of this document, the *nAttributes* data primitives that comprise the Data Objects are specified in the file in sequential order.

### *D.1.4. Binary Grid Files*

**Table 6-7. Description of Binary Grid File Parameters.**

| Parameter Name | Definition | Binary Value |
|---|---|---|
| File identification string | The 11 characters: GEOTESSGRID | 11 character string NOT preceded by integer string length |
| Grid file format version number | Grid file format version number in range 1 to 65535. The two least significant bytes store the version number and the two most significant bytes are zero. This value is also used to determine if the file is stored in big-endian or little-endian format. | 4 byte integer in range 1 to 65535. |

| Parameter Name | Definition | Binary Value |
|---|---|---|
| Software version | The name of the software that was used to generate the content of the grid, and its version number | Integer length of string followed by string. |
| Date | The date that the content of the grid was generated | Integer length of string followed by string. |
| gridID | Every grid has a unique gridID that is stored in both the grid file and in all the model files that use that grid. When the model and grid are loaded, a check is performed to ensure that the two gridIDs match exactly. While any string can be used as a gridID, an MD5 hash of the vertices, triangle indices, level indices and tessellation indices is an excellent choice. | Integer length of string followed by string. |
| nTessellations | The number of multi-level tessellations that define the grid | 4 byte integer |
| nLevels | The total number of tessellation levels that define the grid. This is the sum of the number of tessellation levels in all the multi-level tessellations in the grid. | 4 byte integer |
| nTriangles | The total number of triangles that defines the grid. This is the sum of the number of triangles in all tessellation levels of all multi-level tessellations. | 4 byte integer |
| nVertices | The number of vertices that define the grid. Each vertex is a 3D unit vector. | 4 byte integer |
| Tessellation level indices | For each tessellation two integers are specified: the index of the first level and the index of the last level plus one, that defines the tessellation. | *nTessellations**2 4-byte integers. |
| Level triangle indices | For each tessellation level two integers are specified: the index of the first triangle and the index of the last triangle plus one, that define the level. | *nLevels**2 4-byte integers. |
| Vertex positions | For each vertex 3 doubles are specified that define the x, y and z components of the unit vector | *nVertices**3 8-byte doubles |

| Parameter Name | Definition | Binary Value |
|---|---|---|
| | corresponding to the position of the vertex | |
| Triangle indices | For each triangle 3 integers are specified that define the indices of the 3 vertices that define the triangle | *nTriangles*3 4-byte integers.* |

## D.2. ASCII Format

In this section, the format of GeoTess ASCII model and grid files is described.

### *D.2.1. ASCII Model Files*

**Table 6-8. Description of ASCII Model File Parameters.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| File identification string | The 12 characters: GEOTESSMODEL | 12 character string followed by line terminator. |
| Model file format version number | Model file format version number in range 1 to 65535. | Integer in range 1 to 65536, followed by a line terminator. |
| Software version | The name of the software that was used to generate the content of the model, and its version number | String followed by line terminator. |
| Date | The date that the content of the model was generated | String followed by line terminator. |
| EarthShape | File format version 2 or greater, only. The name of the ellipsoid used by GeoTess. Valid options are SPHERE, GRS80, GRS80_RCONST, WGS84, WGS84_RCONST, IERS, IERS_RCONST. This parameter was not present in model file format 1 (WGS84 was assumed). | String followed by line terminator. |
| Model description. | Model description. | As many strings as desired, separated by line terminators. |
| End model description | The string "</model_description>" on a line by itself. | String followed by line terminator. |
| Attribute names | A list of the names of all the attributes stored in the model, separated by semi-colons. For example: 'vp; vs; density'. | String "attributes: " followed by a semi-colon delimited list of attribute names. List is followed by a line terminator. |

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| | *nAttributes* is the number of attributes specified. | |
| Attribute units | The units of the defined attributes, separated by semi-colons. For example: 'km/sec; km/sec; g/cc'. The number of entries must be equal to *nAttributes*. | String "units: ", followed by a semi-colon delimited list of units. List is followed by a line terminator. |
| Layer names | The names of all the layers that define the model, separated by semi-colons and listed in order of increasing radius. For example: 'core; mantle; crust'.<br><br>*nLayers* is the number of layer names specified. | String "layers: ", followed by a semi-colon delimited list of layer names. List is followed by a line terminator. |
| Data Object type | The type of the Data objects stored in this model. Must be one of DOUBLE, FLOAT, INT, SHORT or BYTE. | String followed by a line terminator. |
| nVertices | Number of vertices defined in the grid. | Integer, followed by a line terminator. |
| Layer index – tessellation index map. | An integer for each layer in the model specifying the index of the multi-layer tessellation that supports that layer. | *nLayers* integers, with the last one followed by a line terminator. |
| Profile objects | A Profile object for each layer at each vertex in the model. See section Profiles for Profile definitions. | *nVertices* * *nLayers* Profile objects. Layer index varies fastest. Profiles associated with the same vertex are listed in order that increases with radius. |
| Grid file specifier | String specifying the file in which the grid information is stored. If the grid file specifier is the single character '*', then the grid information is stored in the same file as the model data, immediately following the gridID. Otherwise, the grid file specifier indicates the name of the file that contains the grid information. | String followed by a line terminator. |
| gridID | Every grid has a unique gridID that is stored in both the grid file and in all the model files that use that grid. When the model and grid are loaded, a check is | String followed by a line terminator. |

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| | performed to ensure that the two gridIDs match exactly. While any string can be used as a gridID, an MD5 hash of the vertices, triangle indices, level indices and tessellation indices is an excellent choice. | |

### *D.2.2. ASCII Profile Objects*

**Table 6-9. ProfileEmpty – Profile Object Consisting of a Bottom and Top Radius but no Data.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| Profile type index | ProfileEmpty objects have index 0 | Byte 0 |
| radiusBottom | Radius at the bottom of the profile, in km | Float |
| radiusTop | Radius at the top of the profile, in km | Float followed by a line terminator. |

**Table 6-10. ProfileThin – Profile Object that Represents a Zero-Thickness Profile.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| Profile type index | ProfileThin objects have index 1 | Byte 1 |
| radius | Radius of the profile, in km. | Float |
| data | Data object associated with this profile | Data object followed by a line terminator. |

**Table 6-11. ProfileConstant – A Finite Thickness Profile Characterized by a Single Data Object.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| Profile type index | ProfileConstant objects have index 2 | Byte 2 |
| radiusBottom | Radius at the bottom of the profile, in km | Float |
| radiusTop | Radius at the top of the profile, in km | Float |
| data | Data object associated with this profile | Data object followed by a line terminator. |

**Table 6-12. ProfileNPoints – A Profile Object Comprised of Two or More Radii and an Equal Number of Data Objects.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| Profile type index | ProfileNPoints objects have index 3 | Byte 3 |
| nNodes | Number of nodes on profile | Integer |
| Radii and Data objects | Radii and Data objects | Floating point value, followed by a Data object followed by a line terminator. This combination is repeated *nNodes* times. |

**Table 6-13. ProfileSurface – A Profile Object that Represents Data, but no Radius.**

| Parameter Name | Definition | ASCII Value |
|---|---|---|
| Profile type index | ProfileSurface objects have index 4 | Byte 4 |
| Data | Data object associated with this profile | Data object followed by a line terminator. |

### *D.2.3. ASCII Data Objects*

Data Objects consist of a 1D array of numeric values, where all values are of type double, float, int, short or byte.

All Data Objects in the model must be of the same type and must have the same number of elements. The number of elements of every Data Objects must be equal to *nAttributes*, which is the number of 'attribute names' specified in the file. Whenever a Data Object is specified in the file format specification sections of this document, the *nAttributes* data primitives that comprise the Data Objects are specified in the file in sequential order.

### *D.2.4. ASCII Grid Files*

**Table 6-14. Description of ASCII Grid File Parameters.**

| Parameter Name | Value | ASCII Value |
|---|---|---|
| File identification string | The 11 characters: GEOTESSGRID | 11 character string followed by line terminator. |
| Grid file format version number | Grid file format version number in range 1 to 65535. | Integer in range 1 to 65536, followed by a line terminator. |
| Software version | The name of the software that was used to generate the content of the grid, and its version number | String followed by line terminator. |
| Date | The date that the content of the grid was generated | String followed by line terminator. |

| Parameter Name | Value | ASCII Value |
|---|---|---|
| Comment | Comment that serves to make the file more readable. | ASCII text starting with '#' and ending with a line terminator. |
| gridID | Every grid has a unique gridID that is stored in both the grid file and in all the model files that use that grid. When the model and grid are loaded, a check is performed to ensure that the two gridIDs match exactly. While any string can be used as a gridID, an MD5 hash of the vertices, triangle indices, level indices and tessellation indices is an excellent choice. | String followed by a line terminator. |
| Comment | Comment that serves to make the file more readable. | ASCII text starting with '#' and ending with a line terminator. |
| nTessellations | The number of multi-level tessellations that define the grid | Integer |
| nLevels | The total number of tessellation levels that define the grid. This is the sum of the number of tessellation levels in all the multi-level tessellations in the grid. | Integer |
| nTriangles | The total number of triangles that define the grid. This is the sum of the number of triangles in all tessellation levels of all multi-level tessellations. | Integer |
| nVertices | The number of vertices that define the grid. Each vertex is a 3D unit vector. | Integer followed by line terminator. |
| Comment | Comment that serves to make the file more readable. | ASCII text starting with '#' and ending with a line terminator. |
| Tessellation level indices | For each tessellation two integers are specified: the index of the first level and the index of the last level plus one, that defines the tessellation. | *nTessellations**2 integers with each pair of integers followed by a line terminator. |
| Comment | Comment that serves to make the file more readable. | ASCII text starting with '#' and ending with a line terminator. |
| Level triangle indices | For each tessellation level two integers are specified: the index of the first triangle and the index of the last triangle plus one, that define the level. | *nLevels**2 integers with each pair of integers followed by a line terminator. |

| Parameter Name | Value | ASCII Value |
|---|---|---|
| Comment | Comment that serves to make the file more readable. | ASCII text starting with '#' and ending with a line terminator. |
| Vertex positions | For each vertex 3 doubles are specified that define the x, y and z components of the unit vector corresponding to the position of the vertex | *nVertices**3 doubles with each triple of doubles followed by a line terminator. |
| Comment | Comment that serves to make the file more readable. | *ASCII text starting with '#' and ending with a line terminator.* |
| Triangle indices | For each triangle 3 integers are specified that define the indices of the 3 vertices that define the triangle | *nTriangles*3 integers with each triple of integers followed by a line terminator.* |

## REFERENCES

[1] Snyder, J. P., Map Projections – A Working Manual, USGS Prof. Paper 1395, 1987.
[2] Vincenty, T., Survey Review, 23, No 176, p 88-93, 1975
[3] Zwillinger, D., CRC Standard Mathematical Tables and Formulae, 31rst Edition, 2003.

# DISTRIBUTION

Email—Internal

| Name | Org. | Sandia Email Address |
|---|---|---|
| Andrea Conley | 06752 | acconle@sandia.gov |
| John Merchant | 06752 | bjmerch@sandia.gov |
| Daniel Gonzales | 06752 | dgonza2@sandia.gov |
| Stephanie Teich-McGoldrick | 06756 | steichm@sandia.gov |
| Brian Young | 08861 | byoung@sandia.gov |
| Kathy Davenport | 06756 | kdavenp@sandia.gov |
| Robert Porritt | 06756 | rwporri@sandia.gov |
| Julia Sakamoto | 06752 | jsakomo@sandia.gov |
| Technical Library | 01977 | sanddocs@sandia.gov |

Email—External (encrypt for OUO)

| Name | Company Email Address | Company Name |
|---|---|---|
| Jorge Roman-Nieves | jorge.roman-nieves.1@us.af.mil | AFTAC |
| Gregory Wagner | gregory.wagner@us.af.mil | AFTAC |
| Mike Begnaud | mbegnaud@lanl.gov | Los Alamos National Laboratory |
| Sanford Ballard | sballar@sandia.gov | Retired |

This page left blank

Sandia National Laboratories

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.