

GEOTESS USERS MANUAL

Version 2.2

Sandy Ballard, Jim Hipp, Brian Kraus

Sandia National Laboratories

June 11, 2014

Table of Contents

Table of Contents	2
Introduction	4
Model Components	5
Library Interactions	9
Model population	9
Step 1 – Specify MetaData.....	9
Step 2 – Construct a Model.....	10
Step 3 – Add Data.....	10
Model I/O	11
Model interrogation	12
Grid Information.....	12
Accessing Data Stored in the Model.....	12
Interpolating Attribute Values at Arbitrary Locations.....	13
Extending GeoTess	14
GeoTessBuilder	14
GeoTessBuilder Properties File	14
Model Refinement Mode.....	15
Construction-From-Scratch Mode.....	15
GeoTessBuilder Examples	17
Example 1.....	17
Example 2.....	18
Example 3.....	19
Example 4.....	20
Polygons	21
Great circles	24
Ellipsoids	25
Manipulation of Geographic Locations on an Ellipsoidal Earth	26
Distance between two points.....	29
Azimuth from one point to another.....	30
Points on a great circle.....	30
Finding a new point some distance and azimuth from another point.....	30
References.....	31
GeoTessModelExplorer	31
Installation Instructions	33
Setup.....	33
Build Environments.....	34
Makefile Usage.....	35

Makefile Results	35
Makefile Production.....	36
Changing between 32 and 64 bit modes	37
C and FORTRAN Shells.....	38
C Shell	38
Headers	38
Naming Conventions	39
C Shell Source	39
F Shell	41
Headers	41
Naming Conventions	42
FORTRAN Shell Source	42
File formats	43
Binary Format	43
Binary Model File.....	43
Binary Profile Objects	45
Binary Data Objects.....	46
Binary Grid Files.....	46
Ascii Format	48
Ascii Model Files.....	48
Ascii Profile Objects	50
Ascii Data Objects.....	51
Ascii Grid Files.....	51

Introduction

GeoTess is a model parameterization for multi-dimensional Earth models and an extensible 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 node in the grid.

Users can interact with GeoTess in several ways: 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 and may implement additional methods in the future.
- 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 the section **Library Interactions** below. Complete interface documentation for every publically accessible function in the library is provided for each language, in html format. To locate the documentation, visit the GeoTess website or search the GeoTess directory tree for the directory for the desired computer language. Within that directory will be a file called *source_code_documentation.html* that will lead to the desired information.

The GeoTess library is available in Java and C++ with C and FORTRAN interfaces to the C++ library. Source code and make files are provided, and precompiled binaries are included for SunOS, Linux, Mac OSX and Windows operating systems.

In addition to accessing GeoTess through its libraries, two applications are provided: **GeoTessExplorer** implements extraction of data from a GeoTessModel as boreholes, maps, and 3D blocks, as well as a few other utility-type functions. **GeoTessBuilder** implements the construction of variable resolution 2D triangular tessellations. These applications are described more fully in separate sections below.

Model Components

While many Earth models use regular latitude longitude grids to describe the geographic geometry and topology, GeoTess uses a triangular tessellation. These two approaches are compared in Figure 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, on the other hand, is somewhat more complicated to develop but results in grids with much more uniform cell size and approximately 25% fewer vertices.

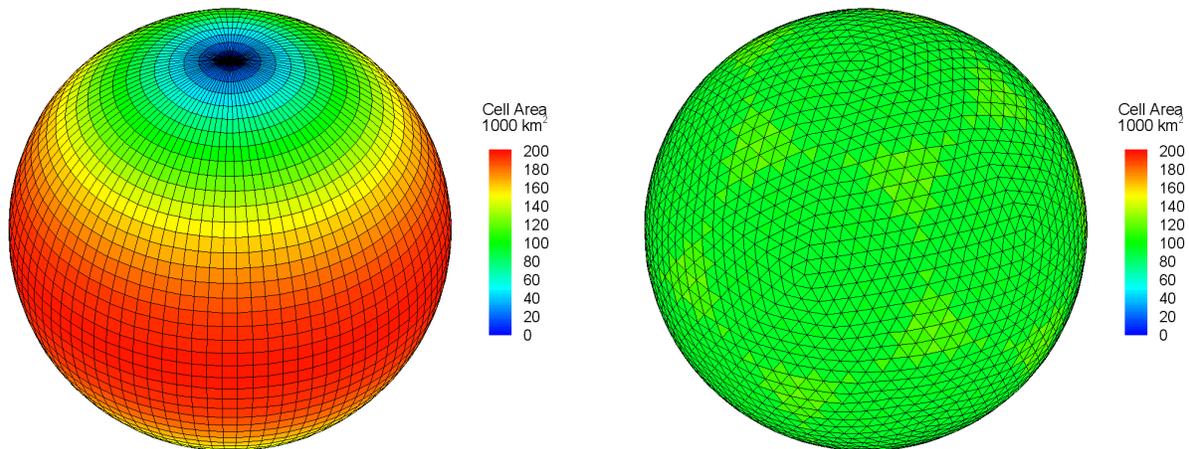


Figure 1 – Comparison of a regular latitude longitude grid and a uniform triangular tessellation. In both grids the edge lengths are approximately 4° .

A GeoTess model is comprised of the following elements:

- A set of **layers** (Figure 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). 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.

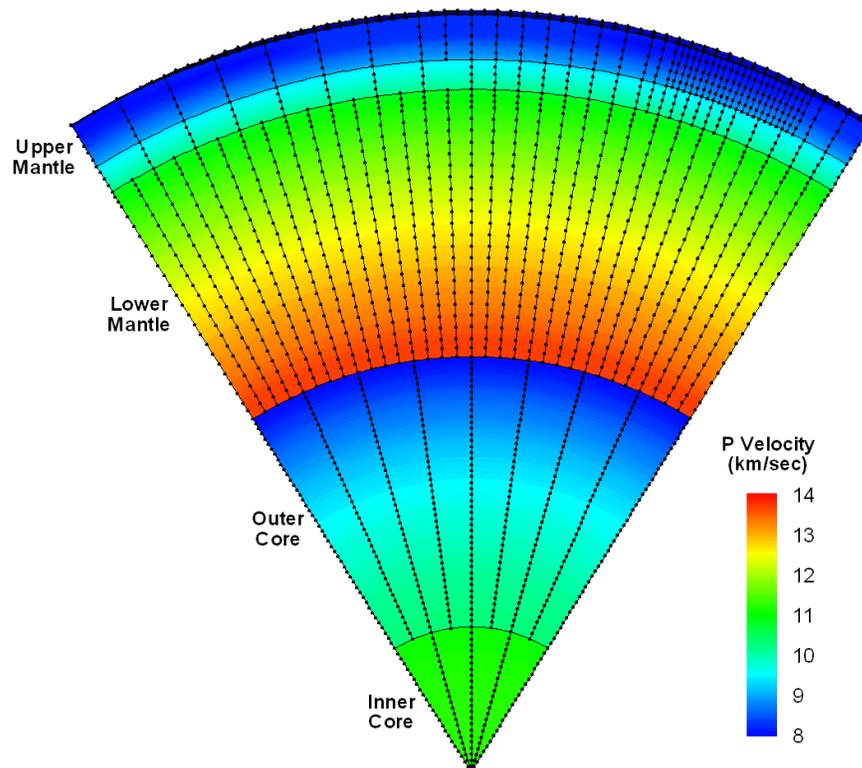


Figure 2 – A slice through a portion of a global 3D P velocity model. The model consists of a number of layers, such as the Inner Core, Outer Core, etc. Each layer is associated with a separate multi-level tessellation, providing variable resolution in the radial direction. Profiles are defined as a 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), 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.

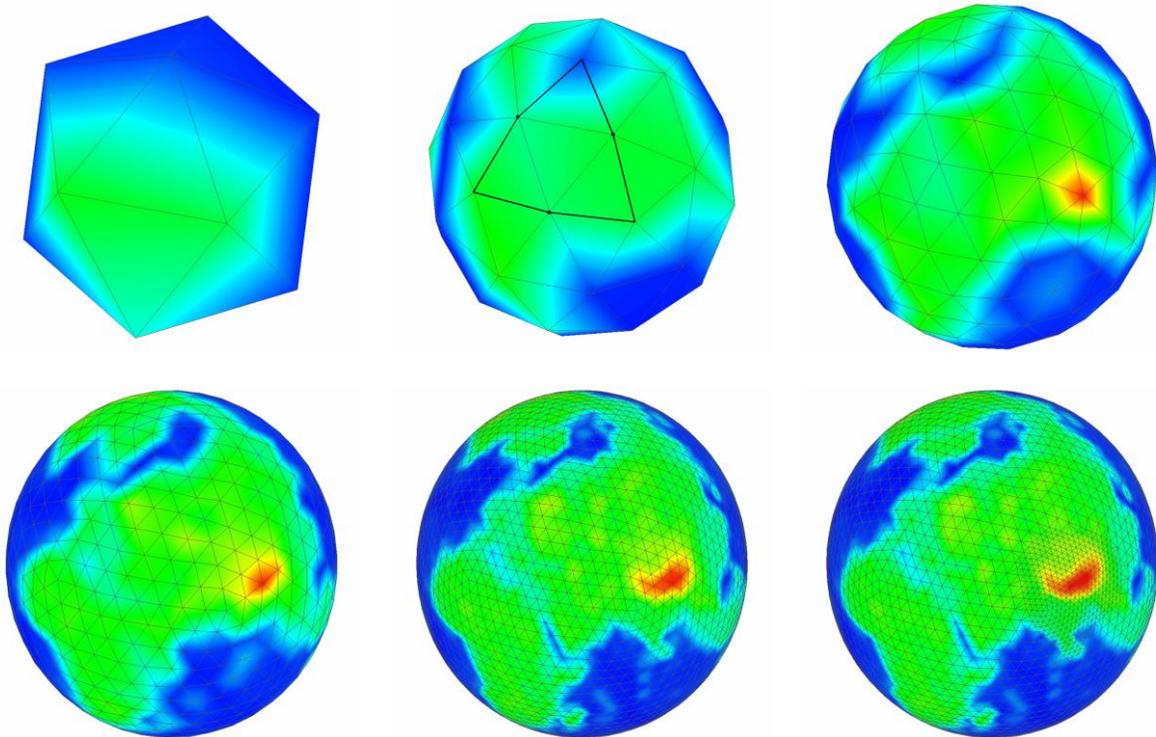


Figure 3 – Construction of a multi-level tessellation by iterative subdivision of triangles. Each image represents one level and together the levels comprise a single multi-level tessellation.

- **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 $nVertices \times nLayers$ 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.

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 publically accessible function in the library is provided for each language, in html format. To locate the documentation, visit the GeoTess website or search the GeoTess directory tree for the directory for the desired computer language. Within that directory will be a file called *source_code_documentation.html* that will lead to the desired information. In addition to the documentation, there are example programs in each computer language that illustrate how to implement basic functions.

Model population

Applications are going to want to generate a new GeoTessModel populated with their data. Example code that performs this operation is provided for each supported language. There are three major steps involved in this task:

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 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”.

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.

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.getConnectionedVertices(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 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 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.

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 a separate section.

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 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 data set 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 data sets 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 on to the vertices of a grid. There may be separate data sets 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 data sets 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 data set 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 an exception is thrown.

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. The sample codes supplied with each supported language provide simple examples of the most common model interrogation functions.

Model interrogations functions fall into 3 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.

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 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 *i*'th vertex in the grid. GeoTessUtils provides the capability to convert back and forth between a unit vector and geographic latitude and longitude.

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 is 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.

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 or cubic spline 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(attributeIndex)*. 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 online documentation for more information about these methods.

Extending GeoTess

The Data structures attached to the nodes of a GeoTessModel may not always be able to capture all of the information that an application may need to store. When this is the case, Java and C++ applications can extend GeoTessModel to store the additional information. Examples of Java and C++ classes that do this are provided. 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.

GeoTessBuilder

A number of GeoTessGrid files are delivered as part of the GeoTess delivery package. Each of these grids is comprised of a single multi-level tessellation with uniform geographic resolution. Grids with triangle edge lengths ranging from 64° down to $\frac{1}{2}^\circ$ are provided. If these grids do not meet the needs of an application, custom grids can be constructed that may be comprised of multiple multi-level tessellations in order to achieve variable resolution in the radial direction, and/or variable resolution in the geographic dimensions. GeoTessBuilder is a java application that can construct these grids.

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 following section defines the properties that can be defined in the properties file. Sample properties files are supplied with the software delivery.

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) 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, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, 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 of the methods, in any combination, is allowed. Also, multiple multi-level tessellations may be defined in the properties file and each may be refined independently from the others.

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.

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.

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), tetrahedron, 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 '*spherical_cap*', then the remaining tokens are interpreted as:
 - latitude of the center of the spherical cap
 - longitude of the center of the spherical cap
 - radius of the spherical cap in degrees
 - tessellation index
 - triangle edge length specifying the size of the triangles desired within the spherical cap.

- 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. 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".

GeoTessBuilder Examples

Example 1

The first example of building a grid will construct a single GeoTessGrid object that is comprised of three multi-level tessellations. Each tessellation has uniform resolution in the geographic dimensions. See section on Model Population to learn about how to use a GeoTessGrid like this 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
```

Figure 4 illustrates the 3 multi-level tessellations that result from running this example.

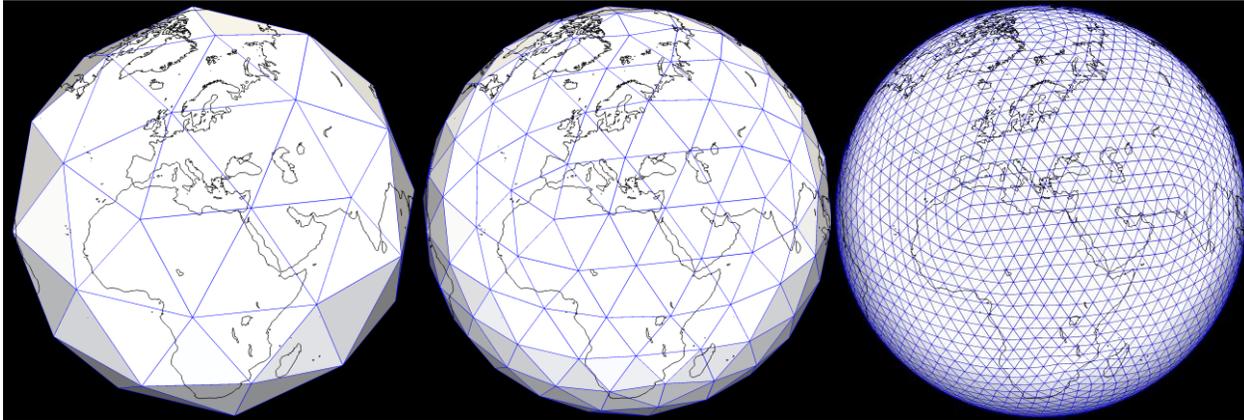


Figure 4 – Three multi-level tessellations generated by example 1. The tessellations have triangle sizes of approximately 32° , 16° , and 4° . Each image shows only the top level of the corresponding multi-level tessellation. These multi-level tessellations are all components of a single GeoTessGrid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

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 of triangles mainly of 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 $\frac{1}{8}^\circ$.

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
```

Figure 5 illustrates the variable resolution tessellation that is generated by this example.

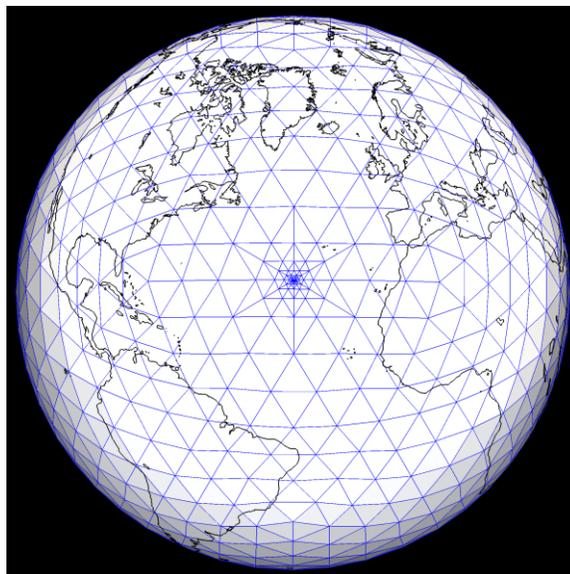


Figure 5 – Top level of the multi-level tessellation generated by example 2. Most of the triangles shown have edge lengths of approximately 8° but triangles near the refinement point are as small as $\frac{1}{8}^\circ$.

Example 3

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. 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 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 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

```

Figure 6 illustrates the results of this example.

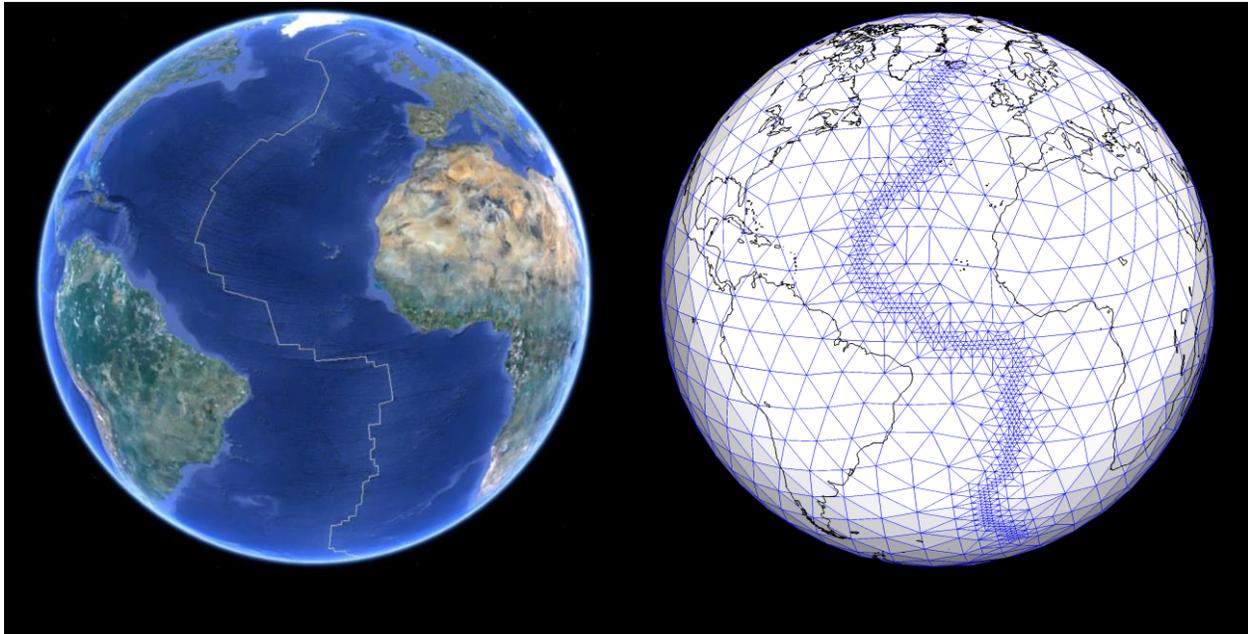


Figure 6 – (left) The trace of the mid_Atlantic Ridge as viewed with Google Earth. (right) Top level of the multi-level tessellation generated by example 3. Most of the triangles shown have edge lengths of approximately 8° but triangles that span the mid-Atlantic Ridge have triangles with edge lengths about 1° .

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 $\frac{1}{8}^\circ$.

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 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

```

Figure 7 illustrates the results of this example.

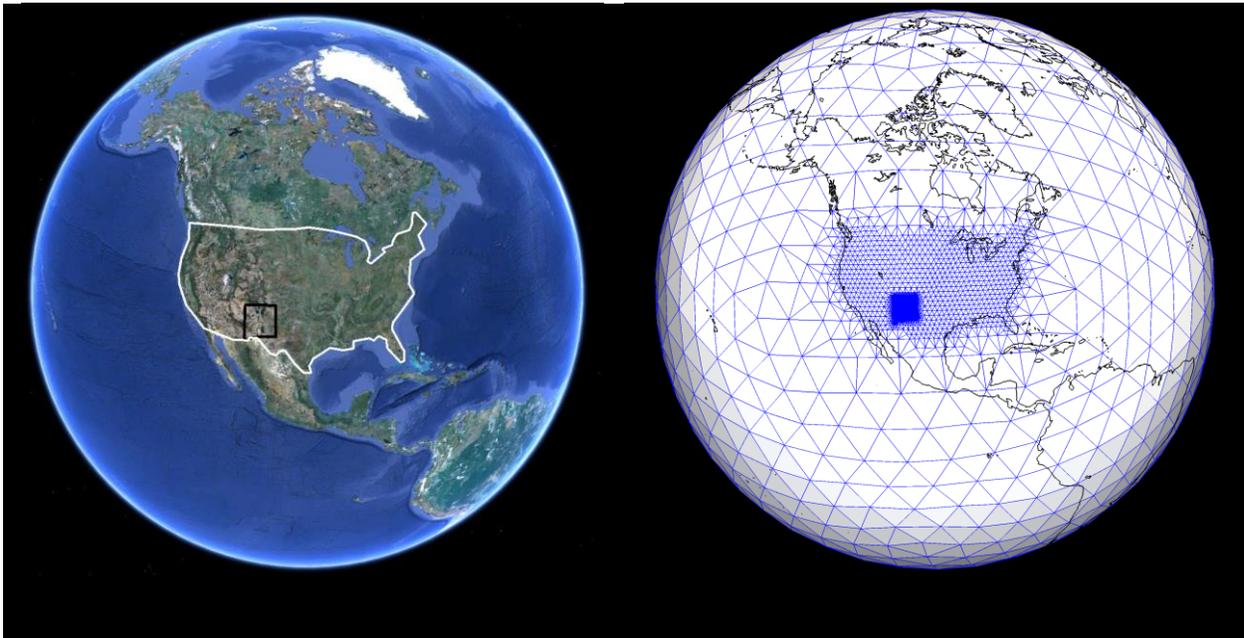


Figure 7 – (left) Polygons generated and viewed with Google Earth. (right) Top level of the multi-level tessellation generated by example 4. Triangles far from the conterminous US have edge lengths of approximately 8° but triangles with at least one corner inside the polygon surrounding the US have triangles with edge lengths about 1° . Triangles with a corner in the polygon defining the state of New Mexico have edge lengths of approximately $\frac{1}{8}^\circ$.

Polygons

GeoTess makes use of polygons for several purposes. GeoTessBuilder uses polygons to define grid resolution, and GeoTessModel can use polygons to select a subset of all Points in a GeoTessModel for inclusion in a PointMap. To directly construct and make use of polygons in Java and C++

applications, consult the html documentation for those languages. 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 similar to 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++, C and FORTRAN 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 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:

Records that start with '#' are considered to be comment lines and are ignored.

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*.

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 considered to be '*inside*' the polygon, otherwise it is considered to be '*outside*' the polygon.

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

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.

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.

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*

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'. The second token specifies how the surface is to be specified. The following possibilities are defined:

- *radius/depth* – The surface is defined by a constant radius/depth. The third token, *Z*, specifies the radius/depth value in km. The final token, *layerIndex*, specifies whether or not the surface is constrained to a particular layer. If *layerIndex* is negative, then the surface is not constrained to any particular layer, 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.
- *layer* – In this case, the surface is everywhere constrained to reside in the layer specified by *layerIndex*, which must correspond to a valid layer. *Z* specifies a fractional position within the layer. If *Z* is ≤ 0 , then the surface will track the bottom of the specified layer. If *Z* is ≥ 1 , the surface will track the top of the layer. For intermediate values, the surface will track the corresponding fractional position within 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.

Great circles

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 2π radians apart, inclusive. Either or both of the 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 has to reside in between the first and last point of both great circles. If *inRange* is false, then that constraint is not applied.

GreatCircle has the ability to 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 $y \times z$.

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 π radians, this is straightforward. But for great circles longer than π radians, great circles of exactly

zero, π or 2π 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 π 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 π 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 π 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 $\geq \pi$ and $\leq 2\pi$. 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 π 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 $\leq 2\pi$, inclusive. It can fail, however, if *firstPoint* coincides with either of the poles because the notion of azimuth from a pole is undetermined.

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, 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 in the model the name of the ellipsoid that was used to populate the 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 an Earth ellipsoid that is associated with the model. This entailed incrementing the model file format number from 1 to 2 (see the appendix). There are facilities provided in the GeoTess software to manipulate geographic information and for retrieving the radius of the ellipsoid at a specified latitude. See functions *GeoTessModel.getEarthShape()* for more information. The section *Manipulation of Geographic Locations on an Ellipsoidal Earth* describes how GeoTess manages geographic information and Earth ellipsoids.

The Earth ellipsoids supported by GeoTess include:

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 of the other ellipsoids use the specified inverse flattening parameter to convert between geocentric and geographic latitudes. The SPHERE and all of the 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.

Manipulation of Geographic Locations on an Ellipsoidal Earth

In order 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, \mathbf{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 $0^\circ, 0^\circ$; v_1 points toward latitude, longitude $0^\circ, 90^\circ$ and v_2 points toward the north pole (Figure 8).

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, ϕ' , longitude, θ' , 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 (Figure 9). 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 ϕ' , and geographic longitude θ' , of a point, the geocentric latitude, ϕ , and geocentric longitude, θ , are (Snyder, 1987)

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

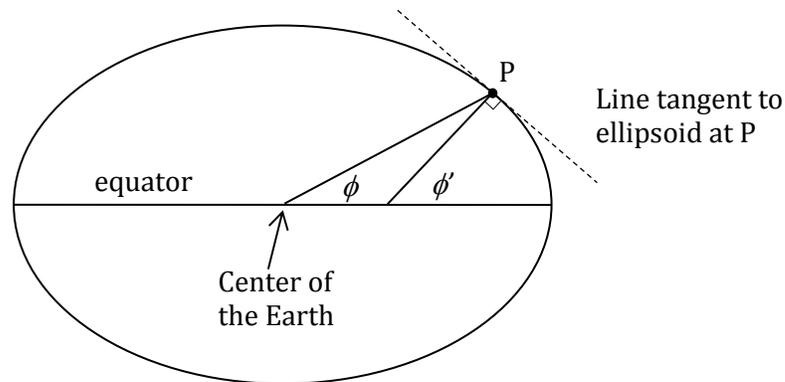


Figure 9 – An exaggerated ellipsoid illustrating the difference between geographic latitude, ϕ' , and geocentric latitude, ϕ .

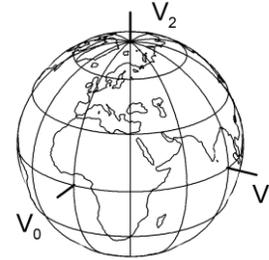


Figure 8 – Earth centered coordinate system. v_0 points from the center of the Earth towards the point on the surface with latitude and longitude $0^\circ, 0^\circ$; v_1 points toward latitude, longitude $0^\circ, 90^\circ$ and v_2 points toward the north pole.

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 ϕ 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, \mathbf{v} , and radius r back to geographic latitude, longitude and depth

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

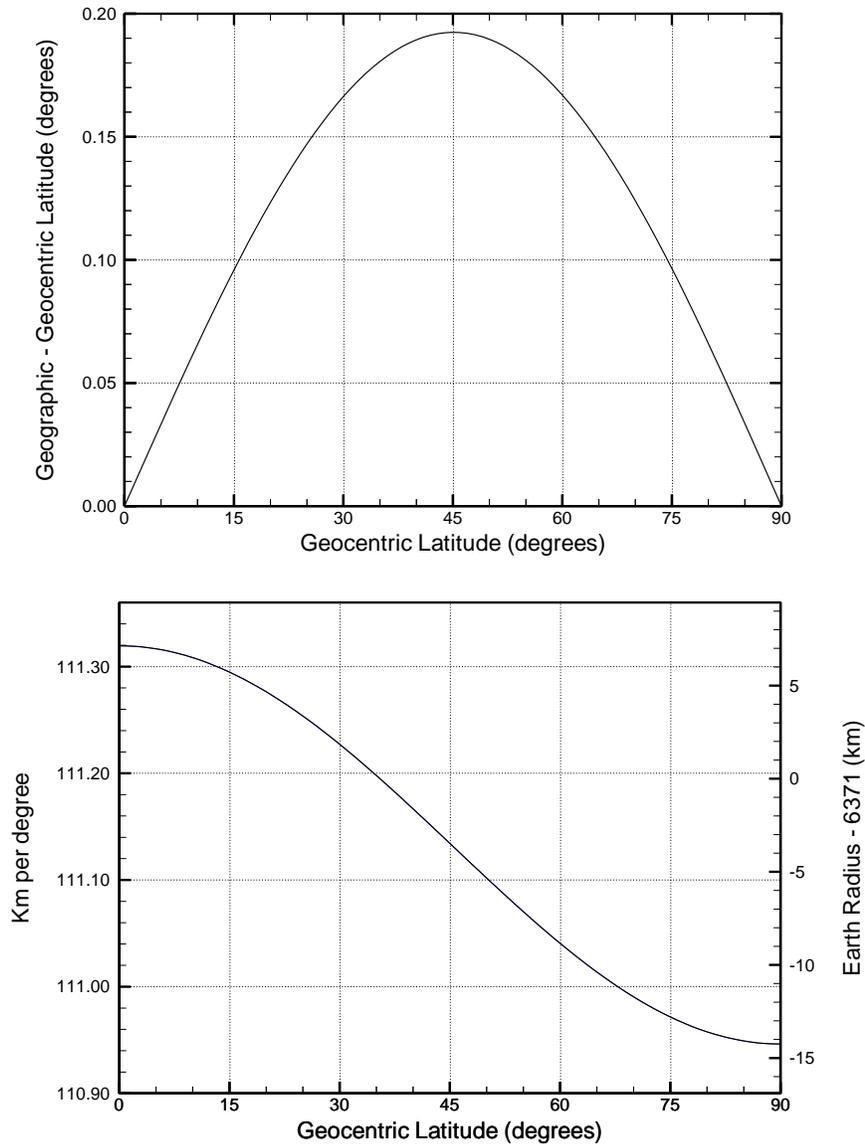


Figure 10 – a) Comparison of geographic and geocentric latitudes for the GRS80 ellipsoid. b) Km per degree and Earth radius as a function of geocentric latitude.

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

Distance between two points

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

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

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

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

or consider the algorithm of Vincenty (1975).

Azimuth from one point to another

The azimuth, α , from \mathbf{u} to \mathbf{v} , measured clockwise from north, is

$$\alpha = \arccos\left(\frac{|\mathbf{u} \times \mathbf{v}| \cdot |\mathbf{u} \times \mathbf{n}|}{|\mathbf{u} \times \mathbf{v}| \cdot |\mathbf{n}|}\right) \quad (8)$$

if $(|\mathbf{u} \times \mathbf{v}| \cdot \mathbf{n} < 0)$ $\alpha = 2\pi - \alpha$

where \mathbf{n} is the vector pointing to the north pole, $\mathbf{n} = [0, 0, 1]$.

Points on a great circle

Given two points defined by unit vectors \mathbf{u} and \mathbf{v} , to find another point, \mathbf{w} , that lies on the great circle defined by \mathbf{u} and \mathbf{v} , at some angular distance δ , measured from \mathbf{u} in the direction of \mathbf{v} (see Figure 11)

$$\mathbf{t} = \frac{(\mathbf{u} \times \mathbf{v}) \times \mathbf{u}}{|\mathbf{u} \times \mathbf{v}| \cdot |\mathbf{u}|} \quad (9)$$

$$\mathbf{w} = \mathbf{u} \cos \delta + \mathbf{t} \sin \delta$$

Note that if many points, \mathbf{w}_i , are to be found along the same great circle defined by \mathbf{u} and \mathbf{v} , the normalized vector triple product, \mathbf{t} , only needs to be computed once.

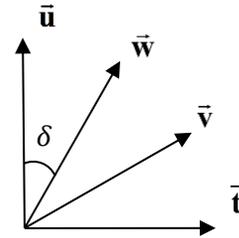


Figure 11 – Calculation of \mathbf{w} given \mathbf{u} and \mathbf{v} . All vectors are of unit length and lie entirely in the plane of the figure.

Finding a new point some distance and azimuth from another point

To find a point, \mathbf{w} , that is some specified distance δ from \mathbf{p} in direction φ , (see Figure 12) we first find an intermediate point \mathbf{u} , distance δ north of \mathbf{p} by applying Equation 9 with $\mathbf{v} = \mathbf{n} = [0, 0, 1]$. Then \mathbf{w} is found by rotating \mathbf{u} around \mathbf{p} by angle $\alpha = -\varphi$.

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

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

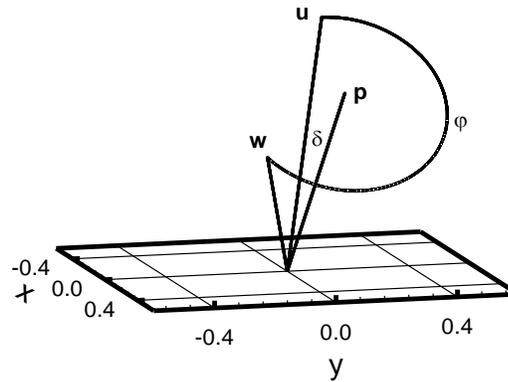


Figure 12 – Start at point **p**, located at latitude, longitude $45^\circ, 0^\circ$. Find a new point **u** $\delta = 20^\circ$ north of **p**. Then rotate **u** $\varphi = 235^\circ$ around **p** to position **w**. Note that $\angle pu = \angle pw = \delta = 20^\circ$.

References

Snyder, J. P., Map Projections – A Working Manual, USGS Prof. Paper 1395, 1987.

Vincenty, T., Survey Review, 23, No 176, p 88-93, 1975

Zwillinger, D., CRC Standard Mathematical Tables and Formulae, 31st Edition, 2003.

GeoTessModelExplorer

GeoTessModelExplorer is a Java application that implements a set of command line driven utilities to extract maps, vertical slices, boreholes and vtk plot files from a GeoTessModel. In order to run GeoTessModelExplorer, locate the Java jar file geotess.jar in the Java section of the GeoTess delivery and type '*java -jar geotess.jar*'.

Available functions include:

- version -- output the GeoTess version number
- toString -- print summary information about a model
- statistics -- print summary statistics about the data in a 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
- 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
- `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
- `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

The intention is that users would either pipe the output to a file or insert the call to this program into a script with the output piped to some other program.

If no arguments are supplied, a list of the recognized functions is output. If the first argument is a recognized function but other required arguments are missing, a list of the required arguments is output.

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 in order to maintain the order of the argument list. 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 has to 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.

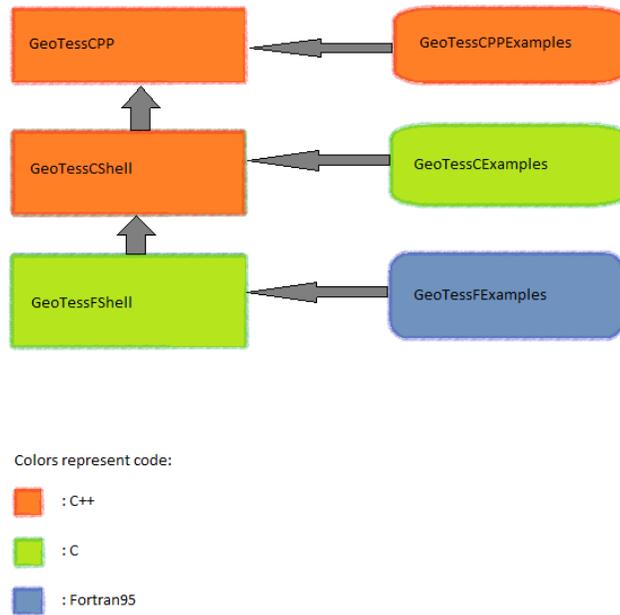
All the functions whose names start with 'vtk' extract information from a GeoTessModel and store it in a file in VTK format (<http://www.vtk.org/VTK/img/file-formats.pdf>). These files can be visualized with free software called ParaView. Visit <http://www.paraview.org> for more information and downloads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

Installation Instructions

Setup

All required GeoTess modules should be placed in the same directory, unless you wish to modify the makefiles. The makefiles assume their module is in the same top level directory as all the other modules.

There are no third party dependencies to GeoTess, however there are inter-dependencies. The GeoTessCPP and GeoTessJava modules are independent. The examples, GeoTessCShell, and GeoTessFShell are not independent. Each example requires its respective language module installed. GeoTessCShell requires GeoTessCPP and GeoTessFShell requires GeoTessCShell.



Build Environments

The various GeoTess modules come with makefiles for Linux, SunOS, MacOS, and Windows – except for Java (See section “Java Build”). This was done to keep the complexity to the command line, which is a common ground between these systems. These makefiles were made for common environments for each of these operating systems. Here are the assumptions that are made for each environment:

- **Windows:** Visual Studio is installed as well as the GNU Make program for windows. This can be found [here](#), and is used for running the makefiles on the windows command prompt. For the C++ and C code, it is compiled using the cl.exe compiler and link.exe linker from the Visual Studio tool chain. The FORTRAN example code is compiled with gfortran, which can be installed in windows via [MinGW](#).
- **SunOS:** GeoTess is built on SunOS using SunStudio 12 using CC and gfortran using gnu make.
- **Linux:** GeoTess is built on Red Hat linux with gcc and gfortran using gnu make.
- **MacOS:** GeoTess is built using gcc and gfortran using gnu make.

All of the required tools must be accessible via the command line through the environment.

It is assumed when building the modules that all other required modules are in the same top level directory. For example: To build GeoTessCShell, which is in the directory “GeoTessRoot”, since the C shell requires the C++ library, the makefile assumes that the C++ code is in “GeoTessRoot/GeoTessCPP”.

Makefile Usage

There are three makefiles that come with each non-java module. One master makefile, and two makefiles – one for Windows builds, and the other for the Linux/Unix/Mac builds. The complexity of making one makefile that could handle both windows path delimiters and other differences from Linux/Unix/Mac was too high, so it was split into two different makefiles. The master makefile chooses the right one to run depending on the operating system.

To call the makefile specific to your OS on the command line, just type “make” in the directory containing the makefiles. The master makefile will select the right sub-makefile for the OS and pass down any arguments given. To supply a target to the makefile, call make with the target as the first argument. There are three targets commonly used in these files:

- “all”: Builds all the module’s object files and then produces the .exe/.dll/.so/.dylib result. For executable results, they are written to the “bin” directory of the module. For libraries they are written to the “lib” directory of the module.
- “clean_objs”: Removes all object files but leaves the .exe/.dll/.so/.dylib results.
- “clean”: Removes all objects files and .exe/.dll/.so/.dylib results. This target should return the module directory to the same state it was in before a call to “make all” was executed.

To change the build between 32 and 64 bit the makefiles can be given the extra argument ARCH=X where X can be “32bit” or “64bit”. The default mode is 64 bit. If ARCH is set to something else or not provided, the build will be done in 64 bit mode. More detail about switching between 32 and 64 bit is given later.

Makefile Results

These are the results of running “make” in each module. Each path is relative to the module itself.

For Windows (including but not limited to):

- **GeoTessCPP:** lib\libgeotesscpp.dll, lib\libgeotesscpp.dll.manifest, lib\libgeotesscpp.lib, lib\libgeotesscpp.exp
- **GeoTessCPPEXamples:** bin\geotesscppexamples.exe
- **GeoTessCShell:** lib\libgeotesscshell.dll, lib\libgeotesscshell.dll.manifest, lib\libgeotesscshell.lib, lib\libgeotesscshell.exp
- **GeoTessCEXamples:** bin\crust.exe, bin\simple.exe
- **GeoTessFShell:** lib\libgeotessfshell.dll, lib\libgeotessfshell.dll.manifest, lib\libgeotessfshell.lib, lib\libgeotessfshell.exp
- **GeoTessFExamples:** bin\crust.exe, bin\simple.exe

For Linux/Unix/Mac:

- **GeoTessCPP:** lib/libgeotesscpp.so
- **GeoTessCPPEXamples:** bin/geotesscppexamples
- **GeoTessCShell:** lib/libgeotesscshell.so
- **GeoTessCExamples:** bin/crust, bin/simple
- **GeoTessFShell:** lib/libgeotessfshell.so
- **GeoTessFExamples:** bin/crust, bin/simple

Makefile Production

This section describes the general method that each makefile uses to produce the results. This includes what files from other modules are required. All paths are relative to the module itself.

- **GeoTessCPP:** Compiles source in “src” with the header files in “include” to produce the shared library result.
- **GeoTessCPPEXamples:** Compiles the source in “src” with the header files in “../GeoTessCPP/include” and links with the shared library in “../GeoTessCPP/lib” to produce the executable result.
- **GeoTessCShell:** Compiles the source in “src” with the header files in “include” and “../GeoTessCPP/include” and links the result with the shared library in “../GeoTessCPP/lib” to produce the result.
- **GeoTessCExamples:** Compiles the source in “src” with the header files in “../GeoTessCShell/include” and links with the shared library in “../GeoTessCShell/lib” to produce the result.
- **GeoTessFShell:** Compiles the source in “src” with the header files in “include” and “../GeoTessCShell/include” and links with the shared library in “../GeoTessCShell/lib” to produce the result.
- **GeoTessFExamples:** Compiles the source in “src” and the modules in “../GeoTessFShell/include” and links with the shared library in “../GeoTessFShell/lib” to produce the result.

There is one difference to this on SunOS however. Since FORTRAN puts a “_” on each symbol name the makefiles supply arguments to the compilers to remove them to be compatible with C. The problem on SunOS is that the intrinsic FORTRAN libraries still have the “_” on them, so compiling with those flags causes the build to break. To solve this, if the GeoTessFExamples makefile is run on SunOS it will copy the source code for the examples and add a “_” to the names of the intrinsic functions used.

Changing between 32 and 64 bit modes

In some cases switching between 32 and 64 bit modes is more complicated than giving the ARCH=X argument to the makefiles (discussed in usage). The complexity appears when trying to switch on SunOS and Windows. Here are some notes on what I had to do to get the build to work properly.

Windows: This is the most difficult environment to switch modes. First off, unlike gcc, simply changing a command line argument isn't enough to switch the cl compiler from x86 to x64. To compensate for this however, windows provides a batch file called "vcvarsall.bat" (Located in: "Microsoft Visual Studio <version>/VC") that can be used to set the environment in the current cmd prompt to use the correct compilers. Calling the batch file with no argument sets the 32 bit tools, and using the argument "amd64" sets the 64 bit tools.

Unfortunately, this does not completely resolve the issue. . While your compiler might be set to build the correct mode of binaries, the linked runtime dlls could still be incorrect. The best method of solving this is by using the "depends" program for windows found [at http://www.dependencywalker.com/](http://www.dependencywalker.com/). This program looks up all the dependencies a binary file has on your system using the same method that windows uses to resolve locations. Then it will tell you which dlls are the wrong bit mode, and if any are broken or missing. There is one important caveat to using this tool. There are 32 and 64 bit versions of this tool, and you have to make sure to use the same mode as what you are trying to compile. Windows will re-route any 32 bit process from entering "Windows/system32" to "Windows/sysWOW64" and vice versa. So if you use a 64 bit depends.exe program to trace problems with a 32 bit dll, it will think all of the system libraries are the wrong versions because it is being directed away from system32 where they are contained. Once you figure out what dlls are incorrect, search for the correct versions and modify your environment to point to those locations rather than the incorrect ones. Generally the user's PATH variable or the system's PATH variable are where Windows searches.

Another issue encountered involved switching between 32 and 64 bit when using the MinGW port of gfortran. It seems only one version of the required libraries "libgfortran-3.dll" and "libquadmath-0.dll" are given in an installation. I was able to find the missing version on the internet, and my preferred method of installation was to rename the existent libraries in "MinGW/bin" to something else, and replace them with the correct version. Do not remove the incorrect libraries. I did this because this is where gfortran looks for dlls and my efforts to make it look elsewhere were not effective.

That is all I can recall on the issue, and keep in mind after these changes using the argument ARCH=X is still required.

SunOS: Thankfully the problems found here are not nearly as complicated. Using the argument ARCH=X will still cause gcc to generate the correct mode of binaries, but the LD_LIBRARY_PATH environment variable needs to change to point to the correct version of libraries. To make this easy, I added a bit of code to my .bashrc which I will share here:

```
MACHINE=64
if [ $OS == SunOS ]
then
if [ $MACHINE == 32 ]
then
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SunStudio12/SUNWspro/lib:/usr/sfw/lib
else
```

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/SunStudio12/SUNWspro/lib/sparc/64:/usr/sfw/lib/64
fi
fi
```

All you have to do then is change the value of “MACHINE” manually and when you start your shell it will look in the correct areas for the libraries, assuming of course your installation is the same as mine.

C and FORTRAN Shells

This section will detail how and why the C and FORTRAN shells were constructed in the ways they were. The main idea was to use the C++ implementation and just refer to it for the heavy lifting. Re-implementing the library in C and FORTRAN would be unreasonable and as a result the shells (also known as interfaces) are very light. There were some complexities that were not foreseen that cause some cases to be less efficient than desired.

C Shell

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.

Headers

There are a few points to notice that are common with each header file in the C Shell. First off, the header files have to be 100% C code, no C++. The only caveat is the usage of the 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.

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 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.

Exception Handling:

Clearly C has no notion of exceptions. To get around this every call to a GeoTess function in the implementation is wrapped in a try-catch block. This does increase the time for each call to a GeoTess function, but aside from re-implementing GeoTess in C there didn't seem to be a better way. 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 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.

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 in to basic arrays for C users. Most cases is from String to char* which is simple, and 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.

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. This can get confusing at times, but the C header files should document when it is safe and when it is not to fully delete a wrapper.

F Shell

The FORTRAN Shell is more complicated in design and less complicated in implementation than the C Shell. It is designed to be used by FORTRAN95 code. The idea remains the same though, to provide a way for FORTRAN code to call the C++ functions. The additional complexity of creating a FORTRAN shell is offset because the C++ library has already been distilled a bit by the C Shell. So instead of calling the C++ library directly, the FORTRAN Shell calls the functions in the C Shell. The FORTRAN Shell is written in pure C, in header and source files.

Headers

Like the C Shell header files, the FORTRAN headers define "GEO_TESS_EXPORT_F" for control of what functions need to be imported and exported from a dll. This however is where the common points end. There are some very important points to note about the function prototypes in the FORTRAN Shell:

1. Every argument must be a primitive type pointer. FORTRAN passes arguments around via pointer, and only commonly shares primitives with C.
2. The exception to 1 is strings. You still pass a FORTRAN string to C through char* arguments, but FORTRAN also appends to the argument list the size of the string as well. Therefore every function that takes a string also allocates an int argument at the end of the argument list that is of type "int" – example:

```
void do_something(char* c1, char* c2, double* clutter, int s1, int s2);
```

Note that the sizes (s1 s2) are in order of the string arguments (c1 c2). The size arguments are automatically supplied by the FORTRAN language, so calling these functions with the sizes supplied is incorrect. Correct form of calling is:

```
call do_something("Hello", "World", 0.5)
```

If any additional arguments are supplied such as:

```
call do_something("Hello", "World", 0.5 5, 5)
```

This could result in a stack corruption as FORTRAN will give the explicit 5 arguments and append 2 more for the sizes of the strings. This is more arguments than the C code expects and will overwrite parts of the C stack.

3. Functions can only return a single primitive value or null. All array results must be handled by passing in an array and writing the result to it. This means there must be a pre-defined size to the array or it can be figured out before hand at runtime.
4. Since there are no enum types, the header file comments will describe the enum types that are valid to the function and give the value (int) that translates to that enum. This is the same with Booleans.
5. FOTRAN and C don't share the same names for primitive types. Here are the mappings that are used: (C -> FORTRAN)

- double : DOUBLE PRECISION
- float : REAL
- int : INTEGER
- short : INTEGER*2
- long(32) : INTEGER*4
- long(64) : INTEGER*8
- char : CHARACTER

Naming Conventions

The naming conventions also follow the method that the C Shell uses. Each function name is based off the C++ name, and the name of the GeoTess object is put in front of the name of the function connected by an underscore. In the case of FORTRAN however, “f” is the first character of every function to avoid symbolic conflicts. Also, since FORTRAN compilers ignore case (converts all characters to lowercase), they don’t understand the C symbols that have uppercase letters – because of this every FORTRAN Shell function flattens the name by replacing each uppercase letter with the lowercase variant, and putting in an underscore to improve readability. Otherwise the names still match what the C Shell uses.

FORTRAN Shell Source

Thankfully the C Shell makes implementing the FORTRAN Shell simpler because it already distills the C++ library to just primitives, arrays, and the GeoTess wrappers. There are other problems to be handled.

State-Machine

My understanding of FORTRAN is very limited, so I understand it to not be able to handle objects. Because of this the FORTRAN Shell is implemented as a state-machine. There are internal pointers to the various GeoTess C wrappers that can be set and updated through the FORTRAN Shell. For all of the examples this seemed sufficient. In cases where multiple instances of an object were required, they were simply expressed as the primitives required to access them from the model. Specifically this is the “fgeoprofile_” set of functions. Instead of having the idea of instances, they are represented as the vertex and layer at which they sit in the model.

Indices

Since FORTRAN is a language that the first index of an array is 1 and C starts at 0 the FORTRAN Shell converts indices between the two languages. All incoming indices (from FORTRAN) are reduced by one, and any outgoing (to FORTRAN) indices are increased by one. A side effect of this is if you print any information that is generated from within the C++ library and it includes indices, they will be off by 1 from what a FORTRAN programmer expects them to be.

Arrays as Results

Since arrays cannot be returned from FORTRAN functions, the C code implementing the FORTRAN shell can’t return any either. To get around this, the C code will ask for the appropriately sized array and copy its result to the given array. In most cases the size (or upper limit on the size) is

given either in documentation or can be discovered at runtime. For toString methods this isn't the case as it is hard to guess what the size is, and it isn't critical to get the size right. For this it's best to just get an array of 2-4kB and resize as necessary.

The functions asking for an array to write the result of their computations to also ask the size of the given array. While most of them will know ahead of time what the size should be, to be safe they ask for it as well. They will never write more data than what the given size says is possible.

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.

Binary Format

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

Binary Model File

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.
Attribute names	A list of the names of all the attributes stored in the model, separated by semi-colons. For example: 'vp; vs; density'. <i>nAttributes</i> 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 <i>nAttributes</i> .	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'. <i>nLayers</i> 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 – tellellation index map.	An integer for each layer in the model specifying the index of the multi-layer tessellation that supports that layer.	<i>nLayers</i> integers.
Profile objects	A Profile object for each layer at each vertex in the model. See section Profiles for Profile definitions.	<i>nVertices</i> * <i>nLayers</i> 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 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.

Binary Profile Objects

ProfileEmpty – Profile object consisting of a bottom and top radius but no data.

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

ProfileThin – Profile object that represents a zero-thickness profile.

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

ProfileConstant – A finite thickness profile characterized by a single data object.

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

ProfileNPoints – A profile object comprised of two or more radii and an equal number of data objects.

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 <i>nNodes</i> times.

ProfileSurface – Profile object that represents data, but no radius

Profile type index	ProfileSurface objects have index 4	Byte 4
Data	Data object associated with this profile	Data object

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.

Binary Grid Files

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.
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 corresponding to the position of the vertex	$nVertices*3$ 8-byte doubles
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.

Ascii Format

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

Ascii Model Files

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’. <i>nAttributes</i> is the number of attributes specified.	String “attributes: “ followed by a semi-colon delimited list of attribute names. List is followed by a line terminator.
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 <i>nAttributes</i> .	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’. <i>nLayers</i> 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.	<i>nLayers</i> 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.	<i>nVertices</i> * <i>nLayers</i> 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 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.

Ascii Profile Objects

ProfileEmpty – Profile object consisting of a bottom and top radius but no data.

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.

ProfileThin – Profile object that represents a zero-thickness profile.

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.

ProfileConstant – A finite thickness profile characterized by a single data object.

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.

ProfileNPoints – A profile object comprised of two or more radii and an equal number of data objects.

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 <i>nNodes</i> times.

ProfileSurface – Profile object that represents data, but no radius

Profile type index	ProfileSurface objects have index 4	Byte 4
Data	Data object associated with this profile	Data object followed by a line terminator.

Ascii Data Objects

Data Objects consist of a 1D array of numeric values, where all of the 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.

Ascii Grid Files

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.
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.	<i>nTessellations*2</i> 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.	<i>nLevels*2</i> 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.
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	<i>nVertices*3</i> doubles with each triple of doubles followed by a line terminator.
Comment	Comment that serves to make the file more readable.	<i>Ascii text starting with '#' and ending with a line terminator.</i>
Triangle indices	For each triangle 3 integers are specified that define the indices of the 3 vertices that define the triangle	<i>nTriangles*3</i> integers with each triple of integers followed by a line terminator.