Riskaware

capability through technology

OpenEAGGR

Software Design Document

Issue 1

16th June 2017

Riskaware Ltd
Colston Tower
Colston Street
Bristol
BS1 4XE
+44 117 929 1058
www.riskaware.co.uk

Author: Ian Bush
ian.bush@riskaware.co.uk
+44 117 933 0539

For the Record

Table of Contents

1 Introduction 5

2 High-Level Design 5

3 OpenEAGGR Library Core 7

3.1 Overview 8

3.2 Polyhedron 8

3.3 Projection 8

3.3.1 Coordinate Conversion 8

3.4 Grid Partitioning 8

3.4.1 Hierarchical Grid 8

3.4.2 Offset Grid 9

3.5 Grid Indexing 9

3.6 Cell Identifier 9

4 Shapes 10

4.1 Overview 10

4.2 Shape Types 10

4.2.1 Points and Cells 10

4.2.2 Linestrings 10

4.2.3 Polygons 10

4.2.4 Shapes 10

5 Spatial Analysis 11

6 Import / Export 13

7 Utilities 13

8 Third Party Libraries 13

8.1 GDAL 13

8.2 Proj4 13

8.3 GoogleTest 13

8.4 Boost 13

9 API 14

9.1 C 14

9.2 Java 14

9.3 Python 2/3 14

10 Extensions to Third Party Software 15

10.1 PostgreSQL/PostGIS 15

10.2 Elasticsearch 16

11 Development Environment 17
11.1 Eclipse 17
11.2 Projects 17
11.3 Other Directories 17
12 Bibliography 18

List of Figures and Tables

Figure 1 – High level OpenEAGGR components 5
Figure 2 – Class diagram for the OpenEAGGR library core 7
Figure 3 – Class diagram for the OpenEAGGR spatial analysis module 11

1 Introduction

This document details the design for the Open Equal Area Global GRid (OpenEAGGR) software library. It outlines the high-level structure of the library and a detailed class-level description of the library. The library has APIs in C, Java and Python which are described at a high level.

2 High-Level Design

The high-level component diagram for the library is shown in Figure 1. The library accepts point, linestring or polygon inputs as WGS84 latitude/longitude coordinates (either directly or as WKT/GeoJSON strings). The outputs from the library are DGGS cell identifiers that represent the input locations. Similarly, the library accepts lists of DGGS cell identifiers and returns the latitude/longitude points that are represented by the DGGS cells.

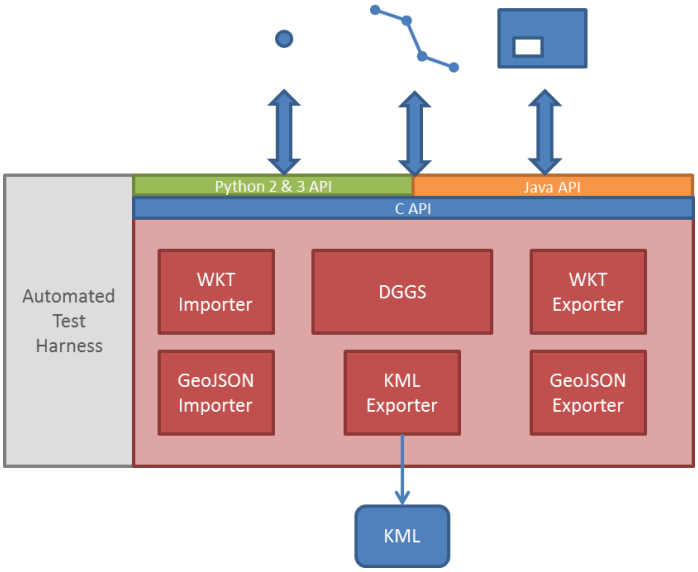

Figure 1 – High level OpenEAGGR components

The library is written in C++ which is compiled to a shared library using GNU's GCC (and MinGW 32- or 64-bit for Windows). The interface to the library is via a C API which is compiled into the library. In addition to the C API, Java and Python APIs allow the library to be used from these languages. These APIs provide the same functionality by wrapping the native library using the C API.

The OpenEAGGR library is comprised of a core component that implements the DGGS functionality surrounded by a number of components that allow import and export of geometry data in various forms. The library is able to process point, linestring and polygon data in the form of WKT [2] or GeoJSON [3] strings. These strings can be input to the library for conversion to DGGS cells or exported by the library when converting DGGS cells to point data. Additionally it is possible to export the DGGS cell data to files in KML format which allows the data to be visualised in an external application (e.g. Google Earth).

An automated test harness is used to ensure that the library functions as expected. The majority of the tests use the C++ code directly but some tests call the library through the APIs.

The library currently has two DGGS implementations available both built on an icosahedral globe using the Snyder Equal Area projection (ISEA). ISEA4T extends the globe and projection to use a hierarchical equilateral triangle grid with aperture 4. ISEA3H uses a hexagonal grid of aperture 3 with offset coordinate cell indexing. For details on the terminology of the DGGS implementations see the Literature Review and Prototype Evaluation [4].

The OpenEAGGR library has been integrated into PostgreSQL and Elasticsearch as an extension/plug-in. The PostgreSQL extension allows DGGS functionality to be used in SQL statements and calls the library through

the C API. The Elasticsearch plug-in allows documents with geo point and shape functionality to be indexed and queried and calls the library through the Java API.

3 OpenEAGGR Library Core

The class diagram for the core of the OpenEAGGR library is shown in Figure 2.

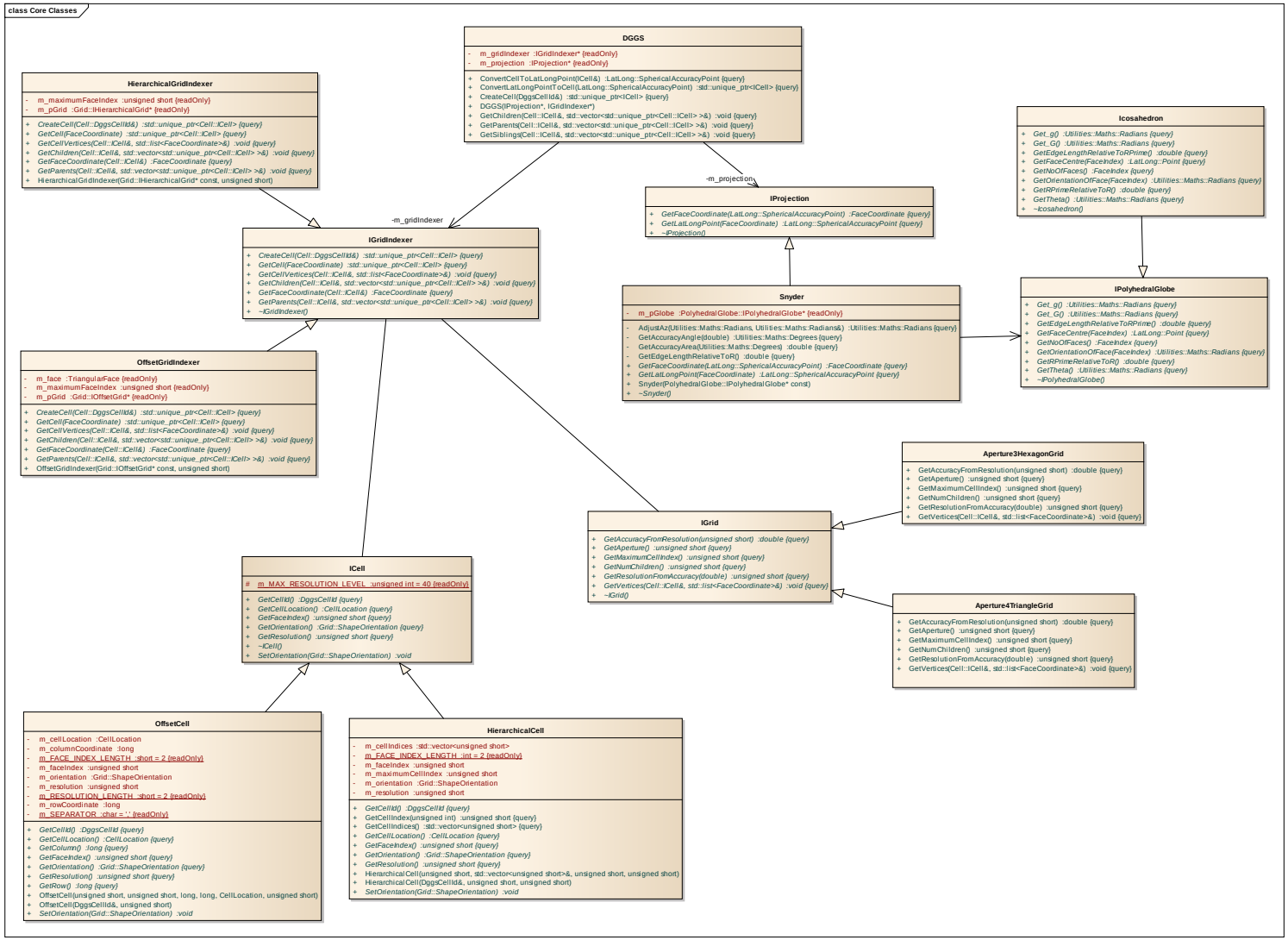

Figure 2 – Class diagram for the OpenEAGGR library core

3.1 Overview

The core DGGS component is also known as the Model, and is responsible for converting between WGS84 latitude and longitude coordinates and DGGS cell identifiers.

The core component can be further broken down into the following sub-components:

- Polyhedron
- Projection
- Grid Partitioning
- Grid Indexing
- Cell Identifier

Each of the subsections has a pure-virtual base class each with one or more implementation classes. This allows DGGS implementations to be built up using different combinations of the sub-components. The base classes can be found in the EAGGR::Model namespace and the implementations can be found in a sub-folder and namespace that uses the name of the base class (e.g. the polyhedron implementations are in folder *IPolyhedralGlobe* in namespace *EAGGR::Model::PolyhedralGlobe*).

The sub-components are described in the following sections.

3.2 Polyhedron

The *IPolyhedralGlobe* virtual base class defines the interface for these components. In the library there is currently a single implementation of this interface (*Icosahedron*) which defines the geometry of the icosahedral globe.

3.3 Projection

The *IProjection* virtual base class defines the interface for these components. In the library there is currently a single implementation of this interface (*Snyder*) which implements the Snyder Equal Area projection. The implementation is based on the paper by John Snyder that describes the algorithm for the projection [5]. The projection has two public methods – one to project a latitude/longitude point to a DGGS cell and one to perform the reverse projection.

3.3.1 *Coordinate Conversion*

The OpenEAGGR library API takes latitude/longitude coordinates in WGS84 however the Snyder Equal Area projection is based on a Spherical Earth model. The supplied coordinates therefore require conversion from the WGS84 ellipsoid to a sphere before being input to the projection and conversion back to the WGS84 ellipsoid after using the reverse projection from the polyhedron back to the Earth.

The OpenEAGGR library uses the third-party Proj4 library to perform the conversion. The Proj4 API is wrapped by the *EAGGR::CoordinateConversion::CoordinateConverter* class which exposes methods to perform the required conversion.

3.4 Grid Partitioning

The *IGrid* virtual base class defines the interface for these components. There are two additional virtual classes which extend the base interface to define the interfaces for grid classes that are based on hierarchical and offset coordinates (*IHierarchicalGrid* and *IOffsetGrid* respectively).

3.4.1 *Hierarchical Grid*

The *IHierarchicalGrid* interface defines methods that are specific to grids that can be indexed using a hierarchical structure. It provides the ability to calculate the cell partition a given point on the polyhedron face is located in and for a given cell can provide the coordinate of the cell centre relative to the centre of

the polyhedron face. In the library there is a single implementation of the *IHierarchicalGrid* interface (*Aperture4TriangleGrid*).

3.4.2 *Offset Grid*

The *IOffsetGrid* interface defines methods that are specific to grids that can be indexed using an offset coordinate system. It provides the ability to calculate the offset coordinate of the cell corresponding to a given point on the polyhedron face and for a given cell coordinate can provide the coordinate of the cell centre relative to the centre of the face. Additionally there are methods to get the parent and child cells for a given DGGS cell. Unlike the hierarchical cell indexing system, parents and children of cells in the offset coordinate system cannot be determined without some knowledge of the grid structure. In the library there is a single implementation of the *IOffsetGrid* interface (*Aperture4TriangleGrid*).

3.5 Grid Indexing

The *IGridIndexer* virtual base class defines the interface for these components. In the library there are currently two implementations of this interface – *HierarchicalGridIndexer* and *OffsetGridIndexer*. Each implementation takes an implementation of IGrid on construction (an *IHierarchicalGrid* and *IOffsetGrid* respectively) to define the geometry of the grid being indexed.

The grid indexer classes contain a method *CreateCell()* which takes a DGGS cell ID and returns a Cell object that is the correct type to be used with that indexer (i.e. a *HierarchicalCell* for the *HierarchicalGridIndexer* and an *OffsetCell* for the *OffsetGridIndexer*).

3.6 Cell Identifier

The *ICell* interface defines a cell on the grid. Through the interface methods it is possible to get the cell's location and orientation relative to the polyhedron. There are two implementations of the interface – *HierarchicalCell* and *OffsetCell*. The implementations differ by how their store their location. The *HierarchicalCell* class uses an index generated by the *HierarchicalGridIndexer* class, and the *OffsetCell* is intend for use with the *OffsetGridIndexer* class.

4 Shapes

4.1 Overview

This section covers the classes that are used to store data about the shapes the OpenEAGGR library converts. The OpenEAGGR library converts shapes between two different domains – the conventional WGS84 latitude and longitude based system and the new DGGS cells. There is also a third domain, which is only used internally when converting being the two main domains. This is a spherical latitude and longitude based system, which the WGS84 points must be converted to before being input into Snyder's projection.

4.2 Shape Types

At the DGGS library level the conversions are only concerned with points and cells, but the C API layer adds for two more shape types – linestring and polygon. The C API breaks these shapes down into their individual points or cells, converts them and reconstructs them back into shapes in the new coordinate domain.

The latitude and longitude shapes are defined in their own namespace – *EAGGR::LatLong*. Whereas the DGGS shapes are included in the *Model* namespaces with the core DGGS library classes.

4.2.1 *Points and Cells*

Points or cells (depending on which domain they are in) are used to reference a single area on the globe. Points use latitude and longitude and also have an associated accuracy in metres squared. Cells have an index and a resolution. The resolution determines the area of the cell which represents the accuracy.

4.2.2 *Linestrings*

Linestrings are made up of multiple points or cells, which define where the line segments start and finish. Each segment leads on from the last so a linestring is always a continuous line.

4.2.3 *Polygons*

Polygons are made up of linestrings. As a minimum polygon must have one linestring which defines the outer limits of the shape, but they can also have one or more inner rings to define holes inside the shape.

4.2.4 *Shapes*

At the top level of both coordinate domains is a container class that can store any of the three different types of shapes. The class allows shapes of different types to be converted in a single API function call.

In the DGGS coordinate domain, the shape class also stores a cells location relative to the vertices of the polyhedron (not applicable to linestring and polygon shapes). This allows client applications to determine whether the cell spans on to two or faces of the polyhedron.

5 Spatial Analysis

The spatial analysis functionality in the DGGS library is contained in the *EAGGR::SpatialAnalysis* namespsace. The class diagram for the spatial analysis module is shown in Figure 3.

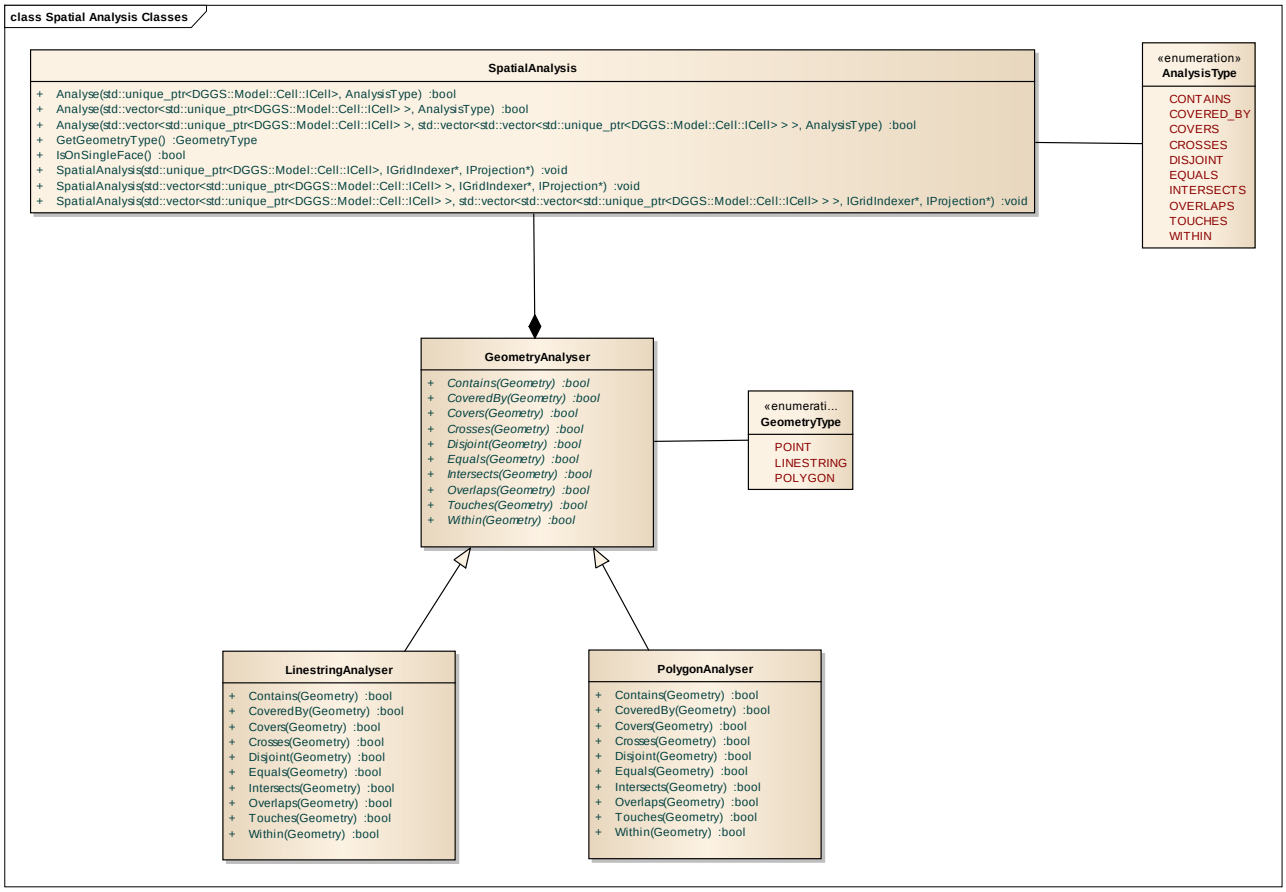

Figure 3 – Class diagram for the OpenEAGGR spatial analysis module

The *SpatialAnalysis* class provides the functions that the OpenEAGGR APIs call in order to perform spatial analysis operations. It can be constructed from either a cell, linestring or polygon in the DGGS domain. Once constructed, the *Analyse* method provides the ability to perform spatial analysis of another cell, linestring or polygon against the shape that was used to construct the *SpatialAnalysis* object. The *Analyse* method also takes one of the supported spatial analysis operations as given by the *AnalysisType* enumeration. The *SpatialAnalysis* class converts the DGGS cell objects that make up the DGGS shape into a set of coordinates on the face of the polyhedron. If all cells are on the same face of the polyhedron then the shapes can be analysed directly. Since each face of the polyhedron has an independent coordinate system, if the shapes span more than a single face it is not possible to use the face coordinates to compare the shapes. In this case the face coordinates are projected back to latitude/longitude coordinates and the spatial comparison is done in the latitude/longitude domain. The *IsOnSingleFace* method of the *SpatialAnalysis* class allows client code to determine whether the shape that was used to construct the object is contained on a single face.

The *SpatialAnalysis* class contains two *GeometryAnalyser* objects (one is created for the case when all shapes are on one face and one for the case when the analysis is performed in the latitude/longitude domain). The *GeometryAnalyser* class performs analysis on shapes in Cartesian coordinates (either face coordinates or latitude/longitude). There are two implementations of the *GeometryAnalyser* class – *LinestringAnalyser* and *PolygonAnalyser*. There is no need for a *PointAnalyser* since cells are treated as cell outline polygons. Similarly to the *SpatialAnalysis* class these are constructed with a shape object and then the methods that perform the spatial analysis operations take the shape that should be compared.

The *GeometryAnalyser* implementations all use the Boost C++ library [6] to perform the spatial analysis operations. Some geometry comparisons are invalid and have not been implemented in Boost (for example if a point contains a polygon). In these cases the *GeometryAnalyser* will return false.

6 Import / Export

There are a number of ways data can be input and output from the OpenEAGGR Library. The principal way is to use the shape structures discussed in Section 4. Shape data in the latitude and longitude coordinate domain can also be defined using WKT or GeoJSON shape strings e.g. “POINT (30 10)”. The OpenEAGGR library can accept shape string inputs and generate output strings in the same formats. Additionally the library can output the shape of the DGGS cell (i.e. a polygon) as a WKT or GeoJSON string. It is also possible to export DGGS cells to a KML file which can be used to visualise cell data produced by the library in applications like Google Earth.

7 Utilities

General low-level classes which contain methods for basic mathematical functions or string manipulations are stored in the *EAGGR::Utilities* namespace. These classes are normally used throughout the code and therefore do not belong to one of the other more specific namespaces.

8 Third Party Libraries

The OpenEAGGR C/C++ code uses a number of third party libraries to provide additional functionality. Details of each library are given in the following sections.

8.1 GDAL

License: X11/MIT

GDAL (Geospatial Data Abstraction Library) is a translator library for raster and vector geospatial data formats released by the Open Source Geospatial Foundation. It is used in the OpenEAGGR library for reading and outputting WKT and GeoJSON shape strings [7].

8.2 Proj4

License: X11/MIT

Proj4 is a library and command line tool designed for converting between different coordinate systems. In the OpenEAGGR library the coordinate converter (Section 3.3.1) calls Proj4 to convert between the WGS84 coordinates supplied by the client application and the spherical coordinates required by the Snyder projection [8].

8.3 GoogleTest

License: New BSD License

GoogleTest is a framework for writing C++ tests on a variety of platforms (including Windows and Linux). It has wide range of features and integrates well into the Eclipse IDE. It can also generate XML test reports, which makes it well suited to be running as part of a continuous integration system. GoogleTest is used for all the tests written for the OpenEAGGR C/C++ code [9].

8.4 Boost

License: Boost software license

Boost [6] is a C++ libarary with a wide range of features but OpenEAGGR uses this library to perform spatial analysis operations.

9 API

The APIs for the different languages are designed to be as similar as possible so it is easy to use the other APIs once you are familiar with one of them. All of the APIs have the same functionality with matching function or method names.

The functions/methods of the API can be categorised into three groups:

- **Conversion from Latitude/Longitude to DGGS cells**
  - *EAGGR_ConvertPointsToDggsCells()*
  - *EAGGR_ConvertShapesToDggsShapes()*
  - *EAGGR_ConvertShapeStringToDggsShapes()*
- **Conversion from DGGS cells to Latitude/Longitude**
  - *EAGGR_ConvertDggsCellsToPoints()*
  - *EAGGR_ConvertDggsCellsToShapeString()*
- **Querying and displaying DGGS cells**
  - *EAGGR_GetDggsCellParents()*
  - *EAGGR_GetDggsCellChildren()*
  - *EAGGR_GetBoundingDggsCell()*
  - *EAGGR_CreateDggsKmlFile()*
  - *EAGGR_GetDggsCellShapeString()*
  - *EAGGR_CompareShapes()*

There are also functions/methods for setting up the model, reading any error messages and deallocating memory. For more information about the API see the Programmer's Guide [10].

APIs for the core library are available in C, Java and Python 2/3, and like the core library, are thread-safe.

9.1 C

The C API defines the functions that are exported in the library's DLL. Although the C API source code contains C++, any code which will be used by the client such as the header file and top level function prototypes must be compatible with C. For example, every top level function includes a try-catch block to ensure no exceptions get passed back to the client application.

9.2 Java

The Java API forms a layer above the C API, by calling the library DLL and wrapping the functionality in a set of classes. The code interfaces to the DLL using JNA (Java Native Access), a third party library. The classes in the Java API are packaged in a JAR file so they can be easily imported into client applications.

9.3 Python 2/3

The Python API wraps the library DLL in a similar way to the Java API. A single API is provided for Python which supports both versions 2 and 3 of the interpreter. The Python classes can be built into a Python wheel package that can be installed for use by client applications using the pip package manager. The code uses the *ctypes* module to call the functions of the DLL. The *ctypes* module has been a part of the standard Python interpreter since version 2.5.

The Python API itself is written entirely in native Python, without any dependencies on third party modules. However additional modules are required if you wish to run the tests or package the module into a wheel file:

- *xmlrunner* - Used to create the XML report file containing the test results, so the tests can be run as part of a continuous integration system.
- *wheel* – Required for building Python modules into wheel files.

Both modules can be easily installed via the *pip* package manager.

10 Extensions to Third Party Software

The OpenEAGGR library has been integrated into other third-party software to extend the functionality to provide DGGS operations. This section describes the design for these extensions.

10.1 PostgreSQL/PostGIS

PostgreSQL [11] is an open-source relational database and PostGIS [12] is an extension to PostgreSQL that adds spatial functionality to PostgreSQL. Both are written in C++ and PostgreSQL has an extension architecture to allow third party software to be integrated into PostgreSQL databases. PostgreSQL extensions have the ability to call native code through a C API, which the OpenEAGGR library already exposes.

The DGGS extension exposes some of the DGGS functionality to:

- Convert WKT to DGGS cells
  - *EAGGR_ToCells()*
- Determine the bounding DGGS cell for a set of DGGS cells
  - *EAGGR_GetBoundingCell()*
- Get the cell outline as WKT
  - *EAGGR_CellGeometry()*
- Convert a DGGS cell to a point in WKT
  - *EAGGR_CellToPoint()*
- Compare two DGGS shapes
  - *EAGGR_ShapeComparison()*

The OpenEAGGR extension does not have any dependency on PostGIS, although it can be used in conjunction with PostGIS if the database tables contain PostGIS geometry objects. Where the OpenEAGGR library requires geometry objects they are provided as WKT. If database tables contain geometry objects then these can be converted to WKT by PostGIS before being passed to the OpenEAGGR extension functions. Within the code and documentation, PostgreSQL and PostGIS are used interchangeably to describe the extension.

The architecture of the PostgreSQL extension requires two configuration files. The dggs.control file contains metadata about the extension. The OpenEAGGR extension uses default values, only setting the version number explicitly in this file. The dggs-x.x.sql file contains the definition of all of the database functions that make up the extension. This file is run in the database when the extension is added to the database.

The extension itself is a single C source file that implements each of the functions defined in the .sql file. It compiles to a C DLL which must be placed in the lib folder of the PostgreSQL installation. A post-build step of the project copies the DLL along with all of the dependencies into a *deploy* folder inside the EAGGRPostGIS project directory. The dependency libraries must be on the path in order for PostgreSQL to be able to load the extension library. The easiest option is to place them in the bin folder of PostgreSQL (since this is usually on the path when PostgreSQL is installed). Note that if PostGIS is installed there are DLLs that already exist in the PostgreSQL bin directory that can conflict with the OpenEAGGR dependencies and prevent the OpenEAGGR extension from loading correctly. Care should be taken to resolve this in order that both PostGIS and OpenEAGGR can function correctly. Ideally PostgreSQL and PostGIS would be recompiled against the same versions of MinGW, GDAL and Proj4 as OpenEAGGR so that all components have the same dependencies. If that is not possible then it may be possible to replace the PostGIS libraries with the OpenEAGGR equivalents (since the OpenEAGGR libraries were compiled with most options enabled and so should be more complete versions of the libraries).

PostgreSQL extensions must be built against the version of PostgreSQL that they are to be deployed against. The current OpenEAGGR extension is built against Postgres 9.5.

10.2Elasticsearch

The plug-in for Elasticsearch provides DGGS functionality in addition to the existing geohashing feature that already exists in Elasticsearch. New index mappings are provided that allow geo_point and geo_shape fields to be converted to DGGS cells in the index. The DGGS functionality reflects the geohash functionality that is already available in Elasticsearch.

The OpenEAGGR plug-in compiles to a jar file (eaggr_es-x.x.jar) which can be built using the Ant build file supplied with the source code.

The main entry point of the plug-in is the *EaggrPlugin* class, which extends the Elasticsearch *Plugin* abstract class. The *EaggrPlugin* implements the *onModule(IndicesModule)* method which allows the plug-in to register the index mappers and query parsers with the Elasticsearch core. When the plug-in is initialised the native libraries are extracted from the eaggr .jar file comprising the OpenEAGGR Java API and saved to a temporary location on the local machine in order that they can be loaded. This location is given by the *java.io.tmpdir* system property of the JVM.

The new indexing fields are implemented by the *DggsPointFieldMapper* and *DggsShapeFieldMapper* classes for point and shape data respectively. These extend the existing *GeoPointFieldMapper* and *GeoShapeFieldMapper* to add DGGS cell indexing in addition to the existing functionality that is available for *geo_point* and *geo_shape* fields. The new datatypes that are available in the index mapping are *dggs_geo_point* and *dggs_geo_shape*. The index mapping settings take parameters that define the DGGS model to be used (ISEA4T is used as the default) and the accuracy (in metres squared). For *dggs_geo_point* fields, the index contains the DGGS cell that corresponds to the point at the given accuracy and all of that cell's parent cells up to the full face of the polyhedron. For *dggs_geo_shape* fields, the index contains the DGGS cell that bounds the shape and all of the bounding cell's parent cells up to the full face of the polyhedron. The DGGS cells are stored in the index under a new parameter *dggs-cellid*.

Queries on the indexed documents can be performed either by a direct match on the dggs-cellid field to find documents that have locations contained in a specified DGGS cell or by using a new query type *dggs_cell* that allows the cell to be specified as a latitude, longitude and accuracy. This new query functionality is implemented by the *DggsQueryParser* class.

Elasticsearch plug-ins must be compiled against the version of Elasticsearch that they are to be deployed against. The current Elasticsearch plug-in is built against version 2.4.0.

11 Development Environment

11.1 Eclipse

Eclipse has been chosen as the main build environment for this project for the following reasons:

- Plug-ins for different languages allow the main C++ library and the C, Java and Python APIs to be developed in the same environment.
- Support for developing code on Linux and Windows.

Eclipse Mars was chosen as the version of Eclipse to use for this project as it supports a headless build of the code on a continuous integration system. The CDT, Java Development and PyDev plug-ins are also needed to allow development in the various programming languages, along with MinGW for building the C/C++ code on Windows.

11.2 Projects

In total there are 11 different Eclipse projects in the source code:

- **EAGGR** – Main OpenEAGGR library written in C++, with functionality exported as C functions.
- **EAGGRTestHarness** – Test harness for the OpenEAGGR library. Includes unit tests for the individual classes, system tests for top-level classes and library testing, and evaluation tests for comparing the performance of the different implementations.
- **EAGGRJava** – Java API and corresponding tests.
- **EAGGRJava64bit** – Equivalent of EAGGRJava to build against a 64 bit Java and use the 64 bit library.
- **EAGGRJavaJarTest** – Test to ensure that the Java API JAR file built by the EAGGRJava project can be imported into client applications.
- **EAGGRJavaJarTest64bit** – Equivalent of EAGGRJavaJarTest to test the jar built against the 64 bit Java.
- **EAGGRPython** – API for Python 2 and 3, including tests.
- **EAGGRPythonWheelTest** – Test to ensure that the Python wheel file has been successfully installed by the *pip* package manager.
- **EAGGRPostGIS** – OpenEAGGR extension for PostgreSQL.
- **EAGGRPostGISTest** – Tests for the PostgreSQL extension.
- **EAGGRElasticsearch** – OpenEAGGR plug-in for Elasticsearch.

11.3 Other Directories

There are four other directories at the top-level of the file structure, which are not project folders:

- **Doxygen** – Doxygen is a third party tool which is used in this project for generating source documentation from comments in the code. This folder contains any files which relate to this tool (although not actually the program itself, that must be installed separately).
- **Eclipse Exports** – Not all information in the Eclipse IDE is stored in the project files. This folder contains exported setting files (e.g. run configurations) from Eclipse, which can be used to set up the IDE.
- **Scripts** – Includes batch files which are run by Eclipse after a successful build and scripts for creating the Python wheel package.
- **Documents** – Contains a copy of this document along with the Programmer's Guide to guide users through integrating the OpenEAGGR library into other applications.

A *ReadMe.md* file is also included at the top-level, which contains instructions on how to set up the build environment and details about the current release of the code.

12 Bibliography

[1] “Geographic information - Well-known text representation of coordinate reference systems,” [Online]. Available: http://docs.opengeospatial.org/is/12-063r5/12-063r5.html.

[2] “GeoJSON,” [Online]. Available: http://geojson.org/.

[3] I. Bush and T. Culmer, “Literature Review & Prototype Evaluation,” 2015.

[4] J. P. Snyder, “An Equal-Area Map Projection for Polyhedral Globes,” *CARTOGRAPHICA Vol 29 No 1,* pp. 10-21, Spring 1992.

[5] “Boost C++ Libraries,” Boost, [Online]. Available: www.boost.org/.

[6] “GDAL - Geospatial Data Abstraction Library,” [Online]. Available: http://www.gdal.org/.

[7] “PROJ.4 - Cartographic Projections Library,” [Online]. Available: https://github.com/OSGeo/proj.4/wiki.

[8] “googletest,” [Online]. Available: http://code.google.com/p/googletest/.

[9] I. Bush and T. Culmer, “Programmer's Guide,” Riskaware, 2015.

[10] “PostgreSQL,” [Online]. Available: https://www.postgresql.org/.

[11] “PostGIS,” [Online]. Available: http://www.postgis.net/.

Riskaware
capability through technology

Riskaware Ltd
Colston Tower
Colston Street
Bristol
BS1 4XE
+44 117 929 1058
www.riskaware.co.uk