Riskaware

capability through technology

OpenEAGGR

Programmer's Guide

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

This document acts as a guide to programmers integrating the Open Equal Area Global GRid (OpenEAGGR) library into client applications. It should be used as an accompaniment to the Doxygen or Javadoc documentation that is generated from the source code. Each of the APIs for the OpenEAGGR library are described and code examples are provided.

2 OpenEAGGR Library Overview

The OpenEAGGR library is an implementation of a Discrete Global Grid System (DGGS) that provides a series of finer resolution grids that cover the entire globe. Each of the cells in the grid at each resolution level represents an equal area of the globe, allowing statistical comparisons to be made between cells in the grid. The Literature Review and Software Design Document provide further details[1].

The OpenEAGGR library is written in C++ and compiled using GCC (GNU Compiler Collection) in Eclipse Mars CDT, with a port of GCC (part of the MinGW tool set) used for the Windows 32 and 64 bit environments. Built on top of the C++ code is a C API which is exposed on the interface of the OpenEAGGR library. In addition to the C API there are Java and Python 2/3 APIs which call the OpenEAGGR library through the C API. Each of the APIs follow a similar naming convention for the functions and parameters.

The OpenEAGGR Library has been integrated as an extension to PostgreSQL and an Elasticsearch plug-in. The PostgreSQL extension is written in C and builds on top of the OpenEAGGR C API. It provides the ability to convert between geometries in WKT format and DGGS cells for both storage and within queries. The Elasticsearch plug-in is written in Java and builds on top of the OpenEAGGR Java API. It provides the ability to index JSON documents with point or shape objects as DGGS cells and to query based on either a DGGS cell or latitude, longitude and accuracy.

[1] Available in Documents folder of the OpenEAGGR repository https://github.com/riskaware-ltd/eaggr

3 C API

The C API for OpenEAGGR is exposed on the interface of the OpenEAGGR library. It consists of:

- Functions to create and use the functionality of the DGGS model
- Data structures that represent the points, linestrings and polygons in the latitude/longitude and DGGS domains
- Enumerations and constants used in the data structures and function return codes

The C API is defined by a single header file to be included by the client application (eaggr_api.h). The following sections describe the components of the C API.

Usage examples for the C API are shown in Appendix A.

3.1 Enumerations

3.1.1 *DGGS_ReturnCode*

The *DGGS_ReturnCode* enumeration defines the possible values that each of the API functions will return. Each function will return *DGGS_ReturnCode.DGGS_SUCCESS* if the function executes successfully. If the function returns any other value then a detailed error message can be obtained using the *EAGGR_GetLastErrorMessage()* function (Section 3.3.18).

3.1.2 *DGGS_Model*

The *DGGS_Model* enumeration defines the DGGS implementations that are supported by the library, i.e. ISEA4T or ISEA3H. This enumeration is used when creating the DGGS model.

3.1.3 *Shape Type Enumerations*

The *DGGS_LatLongShapeType* and *DGGS_ShapeType* enumerations define the types of shape (point, linestring or polygon) that are supported in the latitude/longitude and DGGS domains respectively. These are used in the data structures to define the type of shape being represented.

3.1.4 *DGGS_ShapeStringFormat*

The *DGGS_ShapeStringFormat* enumeration defines the types of geometry text formats that can be imported or exported by the OpenEAGGR library.

3.1.5 *DGGS_ShapeLocation*

The *DGGS_ShapeLocation* enumeration is output by the OpenEAGGR library and can be used by a client application to determine where on the polyhedron face the DGGS cell is located. It has been implemented for point conversion only – linestring and polygon structures add further complexity where the cells might each be on a single face but the faces may be different. ISEA4T cells should always have the value *DGGS_ShapeLocation.DGGS_ONE_FACE* since the cells are congruent with the face. The ISEA3H implementation will return different values depending on the location of the cell.

This has been provided so that during the evaluation period it is possible to ascertain the frequency at which the DGGS cells returned by the library overlap two faces (i.e. on the join between two faces) or many faces (i.e. at the vertex of the polyhedron).

3.1.6 *DGGS_AnalysisType*

The *DGGS_AnalysisType* enumeration defines the spatial analysis operations that are supported by the OpenEAGGR library. When the client application wishes to compare two *DGGS_Shape* objects, the *DGGS_AnalysisType* is used to define the operation that is to be performed.

3.2 Data Structures

Two sets of data structures are defined in the API which correspond to the data in the latitude/longitude and DGGS domains. Latitude/longitude values should be in the WGS84 coordinate system.

3.2.1 *Latitude/Longitude Domain*

The OpenEAGGR library is able to convert point, linestring or polygon data from latitude/longitude to DGGS cells. The latitude/longitude data is defined by creating a *DGGS_LatLongShape* structure. This structure defines the type of shape being represented and the shape data itself. The shape data is a union of *DGGS_LatLongPoint*, *DGGS_LatLongLinestring* and *DGGS_LatLongPolygon* structures, only one of which can be set at any time. The *m_type* field of the *DGGS_LatLongShape* structure must be set to the corresponding value of the *DGGS_LatLongShapeType* enumeration for the type of shape that is represented.

3.2.1.1 Latitude/Longitude Points

A point in the latitude/longitude domain is represented by the *DGGS_LatLongPoint* structure. The point consists of a latitude, a longitude and an accuracy which defines an area in metres squared that represents the uncertainty in the location of the point. This value will determine the resolution of the DGGS cell when the point is converted.

3.2.1.2 Latitude/Longitude Linestrings

A linestring in the latitude/longitude domain is represented by the *DGGS_LatLongLinestring* structure which consists of an array of *DGGS_LatLongPoint* structures and an indication of the number of points making up the linestring.

3.2.1.3 Latitude/Longitude Polygons

A polygon in the latitude/longitude domain is represented by the *DGGS_LatLongPolygon* structure which consists of a *DGGS_LatLongLinestring* structure that represents the outer ring of the polygon and an array of *DGGS_LatLongLinestring* structures that represent zero or more inner rings or holes in the polygon.

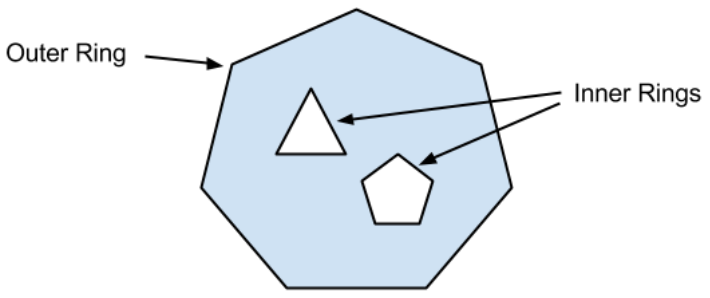

Figure 1 - Outer and inner rings of a polygon

3.2.2 *DGGS Domain*

When data is converted to the DGGS domain from the latitude/longitude domain, a set of structures similar to those defined for the latitude/longitude domain are used. The main data structure is *DGGS_Shape* which defines the type of shape and the shape data. The shape data is a union of *DGGS_Cell*, *DGGS_Linestring* and *DGGS_Polygon* and the *m_type* field of the *DGGS_Shape* structure indicates which type of shape is represented. The DGGS library only supports conversion of cells from the DGGS domain back to the latitude/longitude domain.

3.2.2.1 DGGS Cells

A point in the latitude/longitude domain are represented by a DGGS cell in the DGGS domain. A cell is encoded into a char array defined by the *DGGS_Cell* type.

3.2.2.2 DGGS Linestrings

The *DGGS_Linestring* structure is similar to the latitude/longitude *DGGS_LatLongLinestring* structure, but instead of of *DGGS_LatLongPoint* structures, it stores *DGGS_Cell* strings. Like the *DGGS_LatLongLinestring* structure it also contains the size of the array. When converting from the latitude/longitude domain to the DGGS domain, each latitude/longitude point in the *DGGS_LatLongLinestring* is represented by a *DGGS_Cell* in the *DGGS_Linestring*.

3.2.2.3 DGGS Polygons

Similar to the latitude/longitude *DGGS_LatLongPolygon* structure, the *DGGS_Polygon* structure consists of a DGGS_*LatLongLinestring* representing the outer ring of the polygon and an array of zero or more *DGGS_LatLongLinestring* structures representing the inner rings or holes.

3.3 API Functions

Each of the functions on the OpenEAGGR C API returns a value from the *DGGS_ReturnCode* enumeration. If the return value is not *DGGS_SUCCESS* then a more detailed error message can be obtained from the *EAGGRGetLastErrorMessage()* function (Section 3.3.18). This section describes each of the functions on the API.

3.3.1 *EAGGR_GetVersion*

This function can be used to determine the version of the OpenEAGGR library being used. A char array of size *EAGGR_VERSION_STRING_LENGTH* should be passed to the method to be populated with the version. This is the only function in the API for which the *EAGGR_GetLastErrorMessage()* function cannot be used in the event of an error.

3.3.2 *EAGGR_OpenDggsHandle*

This function is used to create an instance of the DGGS model. The type of DGGS implementation created is specified using the *DGGS_Model* enumeration passed to this method. A handle to the newly created DGGS model is generated and passed back to the client application through the *a_pHandle* parameter. This handle is required to be passed to each of the other methods on the API.

3.3.3 *EAGGR_CloseDggsHandle*

Once the client application has finished using the DGGS model, this function should be called passing in the handle returned by the *EAGGR_OpenDggsHandle* method. This will clean up the resources assigned to the DGGS model within the library. The supplied handle parameter will be set to NULL after calling this method.

3.3.4 *EAGGR_ConvertPointsToDGGSCells*

This function converts an array of points in the latitude/longitude domain to the DGGS domain. The function takes an array of *DGGS_LatLongPoint* structures and the size of the array. The *a_pDggsCell* parameter will be populated with the cells obtained by converting the input points, so should be an array of *DGGS_Cell* structures of the same length as the input array.

3.3.5 *EAGGR_ConvertShapesToDggsShapes*

This function converts shape structures in the latitude/longitude domain to the DGGS domain. The function takes an array of *DGGS_LatLongShape* structures and the size of the array. The function will allocate memory for an array of *DGGS_Shape* structures in the *a_pDggsShapes* parameter and populate it with the converted shape data. The number of *DGGS_Shape* structures in the output array will be the same as the number of *DGGS_LatLongShape* structures supplied with a one-to-one mapping between the two. Once the returned DGGS data is no longer required, the memory should be freed using the *EAGGR_DeallocateDggsShapes()* function (Section 3.3.7).

3.3.6 *EAGGR_ConvertShapeStringToDggsShapes*

This function converts shapes in the latitude/longitude domain defined in a standard string format to the DGGS domain. The formats supported are detailed in the *DGGS_ShapeStringFormat* enumeration. The function takes the string defining the shapes, the format of the string and an accuracy that should be assigned to each of the points defined in the shape string. The function will allocate memory for an array of *DGGS_Shape* structures in the *a_pDggsShapes* parameter and populate it with the converted shape data. The number of shapes in the array is provided by the *a_pNoOfShapes* parameter. Once the returned DGGS data is no longer required, the memory should be freed using the *EAGGR_DeallocateDggsShapes()* function (Section 3.3.7).

3.3.7 *EAGGR_DeallocateDggsShapes*

This function cleans up the memory allocated to hold *DGGS_Shape* structures created when converting shapes from the latitude/longitude domain to the DGGS domain. The function takes an array of *DGGS_Shape* objects and the size of the array and frees all of the memory used by those shapes.

3.3.8 *EAGGR_DeallocateString*

This function cleans up the memory allocated to hold string values returned by the OpenEAGGR library. The function takes a pointer to a *char** object, frees the memory and sets the char* object to NULL.

3.3.9 *EAGGR_ConvertDggsCellsToPoints*

This function converts cells in the DGGS domain to points in the latitude/longitude domain. The point will represent the centre of the DGGS cell and the accuracy on the point represents the area of the DGGS cell in metres squared. The cells are supplied as an array of *DGGS_Cell* strings along with the size of the array. An array of latitude/longitude points should be created with the same number of elements as the number of cells being converted and passed to the function in the *a_points* parameter. This array will be populated by the function with the converted points.

3.3.10 *EAGGR_ConvertDggsCellsToShapeString*

This function converts cells in the DGGS domain to a string format representing the points in the latitude/longitude domain. The cells are supplied as an array of *DGGS_Cell* strings along with the size of the array and the required format of the output string. The resulting string is allocated in memory by the function and returned in the *a_pString* parameter. This memory should be freed afterwards using the *EAGGR_DeallocateString* function (3.3.8).

3.3.11 *EAGGR_GetDggsCellParents*

This function returns the parent cell IDs for a supplied cell ID. Parent cells are the cells in the resolution above that share area with the supplied cell. The cell is supplied in the *a_cell* parameter. An array of *DGGS_Cell* strings should be created with size *EAGGR_MAX_PARENT_CELLS* and passed in the *a_parentCells* parameter. This array will be populated with the parent cell IDs for the supplied cell. Note that the number of parents is not necessarily the same for all DGGS implementations. The number of parent cells for the supplied cell will be returned in the *a_pNoOfParents* parameter.

For the ISEA4T implementation there will only ever be one parent because the cells are congruent. For ISEA3H implementation there could be one or three parents depending on the cells location relative to the cells in the resolution below.

3.3.12 *EAGGR_GetDggsCellChildren*

This function returns the child cell IDs for a supplied cell ID. Child cells are the cells in the resolution above that share area with the supplied cell. The cell is supplied in the *a_cell* parameter. An array of *DGGS_Cell* strings should be created with size *EAGGR_MAX_CHILD_CELLS* and passed in the *a_childCells* parameter.

This array will be populated with the child cell IDs for the supplied cell. Note that the number of children is not necessarily the same for all DGGS implementations. The number of child cells for the supplied cell will be returned in the *a_pNoOfChildren* parameter.

3.3.13 *EAGGR_GetDggsCellSiblings*

This function returns the sibling cell IDs for a supplied cell ID. Sibling cells are defined as cells that share a parent. For ISEA4T this is a simple relationship since all cells have a single parent and three sibling cells. For ISEA3H the relationship is more complex since cells can have either one or three parent cells and consequently either six or fifteen siblings respectively. The cell is supplied in the *a_cell* parameter. An array of *DGGS_Cell* strings should be created with size *EAGGR_MAX_SIBLING_CELLS* and passed in the *a_siblingsCells* parameter. This array will be populated with the sibling cell IDs for the supplied cell. The number of sibling cells in the array will be output in the *a_pNoOfSiblings* parameter. This value is important since the number of siblings is not the same for all DGGS implementations.

3.3.14 *EAGGR_GetBoundingDggsCell*

This function takes an array of DGGS cell IDs and returns the smallest DGGS cell that contains all of the supplied cells. The bounding cell ID will be returned in the *a_pBoundingCell* parameter. If the supplied cells do not all lie on the same face of the polyhedron then there is no cell that contains all of the supplied cells and the function will return a *DGGS_INVALID_PARAM* return code.

In the ISEA3H implementation this method may not return the smallest cell that bounds the supplied cells. It will return a cell that bounds the supplied cells however, depending on the parent structure of the cells, it may be a larger cell. This is due to the fact that some cells have multiple parents and so traversing up the parent tree covers a larger area than if the grids were congruent.

3.3.15 *EAGGR_CompareShapes*

This function takes two *DGGS_Shape* objects and performs a spatial analysis operation on them. The spatial analysis operation is defined using the *DGGS_AnalysisType* enumeration which is passed in the *a_spatialAnalysisType* parameter. The shapes for comparison are passed in via the *a_baseShape* and *a_comparisonShape* parameters. For some spatial analysis operations, e.g. *EQUALS*, it does not matter which shape is passed into which parameter. For other operations, e.g. *CONTAINS*, the order of the shapes does matter. The OpenEAGGR library uses the base shape as the main geometry and performs the operation on the comparison geometry. In the case of *CONTAINS*, for example, the function will output true if the *a_baseShape CONTAINS a_comparisonShape*. The result of the operation is output as a flag in the *a_shapeComparisonResult* parameter.

Currently supported spatial analysis operations are *CONTAINS*, *COVERED_BY*, *COVERS*, *CROSSES*, *DISJOINT*, *EQUALS*, *INTERSECTS*, *OVERLAPS*, *TOUCHES* and *WITHIN*.

3.3.16 *EAGGR_CreateDggsKmlFile*

This function allows an array of cells to be exported to a KML file for visualisation in an external application. The file name and the array of cells are supplied to the function and a KML file will be created at the requested location. The KML file will contain a marker for the centre of each cell in the supplied array and a polygon that defines the boundary of the cell.

3.3.17 *EAGGR_ConvertDggsCellOutlineToShapeString*

This function exports the geometry of a DGGS cell object to a WKT or GeoJSON string. It is exported as a polygon containing a linestring to represent the boundary of the cell. The function takes a *DGGS_Cell* object to be exported and a *DGGS_ShapeStringFormat* enumeration to set the format of the output string. The library will allocate memory for the output string and return in through the *a_pString* parameter. This memory should be freed afterwards using the *EAGGR_DeallocateString* function (3.3.8).

3.3.18 *EAGGR_GetLastErrorMessage*

This method provides the last error message that has been generated by the DGGS model identified by the supplied handle. The error message is allocated in memory by the library and returned in the *a_pErrorMessage* parameter and the length of the message is provided in the *a_pMessageLength* parameter. This memory should be freed afterwards using the *EAGGR_DeallocateString* function (3.3.8).

If no error has occurred for this DGGS model then *a_pErrorMessage* will be set to NULL.

4 Java API

The Java API for OpenEAGGR is provided by a jar file that wraps the native library. This can be included in client applications to provide access to the DGGS functionality. The jar file exposes:

- An *Eaggr* class that provides the methods to access the OpenEAGGR functionality
- Data structures that represent the points, linestrings and polygons in the latitude/longitude and DGGS domains
- Enumerations used on the API
- Exceptions that are thrown by the methods in the *Eaggr* class

When creating a Java project that uses the OpenEAGGR Java API, the following steps are required:

- The eaggr.jar and jna-4.1.0.jar files should be included on the build classpath
- The dlls folder containing the native libraries supplied with the Java API should be on the classpath when the application is run

Code examples of the use of the Java API are shown in Appendix B.

4.1 Enumerations

4.1.1 *ReturnCode*

The *ReturnCode* enumeration defines the possible values that can be returned by the native OpenEAGGR library. If an error occurs in the native code then the *ReturnCode* value will be included in the *EaggrLibraryException* thrown by the *Eaggr* class.

4.1.2 *DggsModel*

The *DggsModel* enumeration defines the DGGS implementations that are supported by the library. This enumeration is used when constructing the *Eaggr* class.

4.1.3 *Shape Type Enumerations*

The *LatLongShapeType* and *DggsShapeType* enumerations define the types of shape (point, linestring or polygon) that are supported in the latitude/longitude and DGGS domains respectively. These are used in the data structures to define the type of shape being represented.

4.1.4 *ShapeStringFormat*

The *ShapeStringFormat* enumeration defines the types of geometry text formats that can be imported or exported by the OpenEAGGR library.

4.1.5 *DggsShapeLocation*

The *DggsShapeLocation* enumeration is output by the OpenEAGGR library and can be used by a client application to determine where on the polyhedron face the DGGS cell is located. It has been implemented for point conversion only – linestring and polygon structures add further complexity where the cells might each be on a single face but the faces may be different. The ISEA4T implementation should always return *DggsShapeLocation.DGGS_ONE_FACE* since the cells are congruent with the face. The ISEA3H implementation will return different values depending on the location of the DGGS cell.

This has been provided so that during the evaluation period it is possible to ascertain the frequency at which the DGGS cells returned by the library overlap two faces (i.e. on the join between two faces) or many faces (i.e. at the vertex of the polyhedron).

4.1.6 *AnalysisType*

The *AnalysisType* enumeration defines the spatial analysis operations that are supported by the OpenEAGGR library. When a client application wishes to compare two *DggsShape* objects, the *AnalysisType* is used to define the operation that is to be performed.

4.2 Data Structures

Two sets of data structures are defined in the API which correspond to the data in the latitude/longitude and DGGS domains. Latitude/longitude values should be in the WGS84 coordinate system.

4.2.1 *Latitude/Longitude Domain*

The OpenEAGGR library is able to convert point, linestring or polygon data from latitude/longitude to DGGS cells. The latitude/longitude data is defined by creating an instance of the *LatLongShape* class. This class is constructed using an instance of *LatLongPoint*, *LatLongLinestring* or *LatLongPolygon*. There are getter methods that return the *LatLongPoint*, *LatLongLinestring* or *LatLongPolygon* plus a method *getShapeType()* to determine which type of shape is being represented. Only one of the *getPoint()*, *getLinestring()* and *getPolygon()* methods will return a value (depending on the return type of the *getShapeType()* method); the other getters will return null.

4.2.1.1 Latitude/Longitude Points

A point in the latitude/longitude domain is represented by the *LatLongPoint* class. The point consists of a latitude, a longitude and an accuracy which defines an area in metres squared that represents the uncertainty in the location of the point. This value will determine the resolution of the DGGS cell when the point is converted.

4.2.1.2 Latitude/Longitude Linestrings

A linestring in the latitude/longitude domain is represented by the *LatLongLinestring* class which consists of an array of *LatLongPoint* objects. The linestring is constructed with no points and the *addPoint()* method is used to add the points of the linestring.

4.2.1.3 Latitude/Longitude Polygons

A polygon in the latitude/longitude domain is represented by the *LatLongPolygon* class which consists of a *LatLongLinestring* structure that represents the outer ring of the polygon and an array of *LatLongLinestring* structures that represent zero or more inner rings or holes in the polygon. The polygon is constructed with the outer ring object and inner rings are added using the *addInnerRing()* method.

4.2.2 *DGGS Domain*

When data is converted to the DGGS domain from the latitude/longitude domain, a set of structures similar to those defined for the latitude/longitude domain are used. The main data structure is *DggsShape* which defines the type of shape and the shape data. This class cannot be constructed by client code; it is returned by API methods. Getters are available for the *DggsCell*, *DggsLinestring* or *DggsPolygon*. Similarly to the *LatLongShape* class, only one of these getters will return a value (determined by the return value of the *getShapeType()* method); the others will return null. Additionally there is a *getShapeLocation()* method which provides information about the location of the shape on the face of the polyhedron.

4.2.2.1 DGGS Cells

A point in the latitude/longitude domain is represented by a *DggsCell* object in the DGGS domain. A cell is constructed with a string cell identifier.

4.2.2.2 DGGS Linestrings

The *DggsLinestring* class is similar to the latitude/longitude *LatLongLinestring* class, but instead of *LatLongPoint* objects, it stores *DggsCell* objects. When converting from the latitude/longitude domain to the DGGS domain, each latitude/longitude point in the *LatLongLinestring* is represented by a *DggsCell* in the *DggsLinestring*.

This class cannot be constructed by client code; it is returned by API methods.

4.2.2.3 DGGS Polygons

Similarly to the latitude/longitude *LatLongPolygon* structure, the *DggsPolygon* structure consists of a Dggs*LatLongLinestring* representing the outer ring of the polygon and an array of zero or more Dggs*LatLongLinestring* structures representing the inner rings or holes.

This class cannot be constructed by client code; it is returned by API methods.

4.3 API Functions

The *Eaggr* class contains the methods to access the OpenEAGGR library functionality. It is constructed with one of the *DggsModel* enumeration values to determine the type of DGGS implementation that should be used. This section describes each of the methods on the *Eaggr* class.

4.3.1 *getVersion*

This method can be used to determine the version of the OpenEAGGR library being used. It returns the OpenEAGGR version as a string.

4.3.2 *convertPointToDggsCell*

This method converts a single point to a DGGS cell object. It is provided to allow a single *LatLongPoint* object to be converted to a *DggsCell* object without the need to build up the more complex shape structures required for the other methods.

4.3.3 *convertShapesToDggsShapes*

This method converts shape structures in the latitude/longitude domain to the DGGS domain. The method takes an array of *LatLongShape* objects and returns an array of *DggsShape* objects (one for each *LatLongShape* object in the supplied array).

4.3.4 *convertShapeStringToDggsShapes*

This method converts shapes in the latitude/longitude domain defined in a standard string format to the DGGS domain. The formats supported are detailed in the *ShapeStringFormat* enumeration. The method takes the string defining the shapes, the format of the string and an accuracy that should be assigned to each of the points defined in the shape string. The method returns an array of *DggsShape* objects, one for each shape defined in the input string.

4.3.5 *convertDggsCellToPoint*

This method converts a single *DggsCell* object to a *LatLongPoint* object. It is provided as the reverse conversion equivalent of the *convertPointToDggsCell* method (4.3.2) and is a simplified version of the *convertDggsCellsToPoints* method (avoiding the need to create an array to wrap a single point).

4.3.6 *convertDggsCellsToPoints*

This method converts cells in the DGGS domain to points in the latitude/longitude domain. The point will represent the centre of the DGGS cell and the accuracy of the point represents the area of the DGGS cell in metres squared. The method takes an array of *DggsCell* objects and returns an array of *LatLongPoint* objects (one for each of the *DggsCell* objects in the supplied array).

4.3.7 *convertDggsCellsToShapeString*

This method converts cells in the DGGS domain to a string format representing the points in the latitude/longitude domain. The cells are supplied as an array of *DggsCell* objects and the required format of the output string should be specified. The method returns a string representing the points generated by converting the *DggsCells*.

4.3.8 *getCellParents*

This method returns the parent cell(s) for a supplied cell. Parent cells are the cells in the resolution above that share area with the supplied cell. The parent cell(s) are returned as an array of *DggsCells*.

4.3.9 *getCellChildren*

This method returns the child cells for a supplied cell. Child cells are the cells in the resolution below that share area with the supplied cell. The child cells are returned as an array of *DggsCells*.

4.3.10 *getCellSiblings*

This method returns the sibling cells for a supplied cell. Sibling cells are the cells in the same resolution as the supplied cell that share a parent. The sibling cells are returned as an array of *DggsCells*.

4.3.11 *getBoundingCell*

This method takes an array of *DggsCells* and returns the smallest cell that contains all of the supplied cells.

4.3.12 *createKmlFile*

This method allows an array of *DggsCells* to be exported to a KML file for visualisation in an external application. The file name and the array of cells are supplied to the function and a KML file will be created at the requested location. The KML file will contain a marker for the centre of each cell in the supplied array and a polygon that defines the boundary of the cell.

4.3.13 *convertDggsCellOutlineToShapeString*

This method exports the geometry of the DGGS cell as a WKT or GeoJSON string. It is exported as a polygon (with no inner rings) representing the boundary of the cell. The cell is supplied as a *DggsCell* object and the required format of the output string should be specified. The method returns a string containing the cell boundary definition in the requested format.

4.3.14 *compareShapes*

This method allows spatial analysis of two *DggsShape* objects. The spatial analysis to be performed is defined using the *AnalysisType* enumeration. The shapes for comparison are passed in as a base shape (*baseShape*) and a comparison shape (*comparisonShape*). For some spatial analysis operations, e.g. *EQUALS*, it does not matter which shape is passed into which parameter. For other operations, e.g. *CONTAINS*, the order of the shapes does matter. The OpenEAGGR library uses the base shape as the main geometry and performs the operation on the comparison geometry. In the case of *CONTAINS*, for example, the function will return true if the *baseShape CONTAINS comparisonShape*.

4.4 Exceptions

There are two types of exceptions that are thrown by the methods on the *Eaggr* class. For exceptions that are generated by the Java code an instance of *EaggrException* will be thrown which provides and error message. Errors that occur in the native library are wrapped in a *EaggrLibraryException* which provides the error code returned by the native function along with an error message.

5 Python API

The Python API for OpenEAGGR is provided as a set of source files that wrap the native library to provide access to the OpenEAGGR functionality. Alternatively, a Python wheel file is provided to allow the OpenEAGGR package to be installed via pip (the Python package manager). The Python API provides:

- An *Eaggr* class that provides the methods to access the OpenEAGGR functionality
- Data structures that represent the points, linestrings and polygons in the latitude/longitude and DGGS domains
- Pre-defined constants used for the API's inputs and outputs
- Exception class that is thrown when an error occurs in the underlying DLL
- Compatibility with both Python 2 and Python 3

Code examples of the use of the Python API are shown in Appendix C.

5.1 Enumerations

These classes are not true enumerated types, but classes which contain only constants. Enumerated types were not introduced until Python 3.4 and so are not compatible with Python 2. However their functionality has been replicated to be consistent with the other APIs which do use enumerations.

5.1.1 *DggsReturnCode*

The *DggsReturnCode* enumeration defines the possible values that can be returned by the native OpenEAGGR library. If an error occurs in the native code then the *DggsReturnCode* value will be included in the exception thrown by the *Eaggr* class.

5.1.2 *Model*

The *Model* enumeration defines the DGGS implementations that are supported by the library. This enumeration is used when constructing the *Eaggr* class.

5.1.3 *DggsShapeType*

The *DggsShapeType* enumeration defines the types of shape (cell, linestring or polygon) that are supported in the DGGS domain. These are used in the data structures to define the type of shape being represented.

5.1.4 *ShapeStringFormat*

The *ShapeStringFormat* enumeration defines the types of geometry text formats that can be imported or exported by the DGGS library.

5.1.5 *DggsShapeLocation*

The *DggsShapeLocation* enumeration is output by the OpenEAGGR library and can be used by a client application to determine where on the polyhedron face the DGGS cell is located. It has been implemented for point conversion only – linestring and polygon structures add further complexity where the cells might each be on a single face but the faces may be different. The ISEA4T implementation should always return *DggsShapeLocation.DGGS_ONE_FACE* since the cells are congruent with the face. The ISEA3H implementation will return different values depending on the location of the DGGS cell.

This has been provided so that during the evaluation period it is possible to ascertain the frequency at which the DGGS cells returned by the library overlap two faces (i.e. on the join between two faces) or many faces (i.e. at the vertex of the polyhedron).

5.1.6 *AnalysisType*

The *AnalysisType* enumeration defines the spatial analysis operations that are supported by the OpenEAGGR library. When a client application wishes to compare two *DggsShape* objects, the *AnalysisType* is used to define the operation that is to be performed.

5.2 Data Structures

Two sets of data structures are defined in the API which correspond to the data in the latitude/longitude and DGGS domains. Latitude/longitude values should be in the WGS84 coordinate system.

5.2.1 *Latitude/Longitude Domain*

The OpenEAGGR library is able to convert point, linestring or polygon data from latitude/longitude to DGGS cells. The latitude/longitude data is defined by creating an instance of one of the *LatLongPoint*, *LatLongLinestring* or *LatLongPolygon* classes.

5.2.1.1 Latitude/Longitude Points

A point in the latitude/longitude domain is represented by the *LatLongPoint* class. The point consists of a latitude, a longitude and an accuracy which defines an area in metres squared that represents the uncertainty in the location of the point. This value will determine the resolution of the DGGS cell when the point is converted.

5.2.1.2 Latitude/Longitude Linestrings

A linestring in the latitude/longitude domain is represented by the *LatLongLinestring* class which consists of an array of *LatLongPoint* objects. The linestring is constructed with no points and the *add_point()* method is used to add the points of the linestring.

5.2.1.3 Latitude/Longitude Polygons

A polygon in the latitude/longitude domain is represented by the *LatLongPolygon* class which consists of a *LatLongLinestring* structure that represents the outer ring of the polygon and an array of *LatLongLinestring* structures that represent zero or more inner rings or holes in the polygon. The polygon is constructed with the outer ring object and inner rings are added using the *add_inner_ring()* method.

5.2.2 *DGGS Domain*

When data is converted to the DGGS domain from the latitude/longitude domain, a set of structures similar to those defined for the latitude/longitude domain are used. The main data structure is *DggsShape* which defines the type of shape and the shape data. The class is constructed with one of a *DggsCell*, *DggsLinestring* or *DggsPolygon* object. Getters are available for the *DggsCell*, *DggsLinestring* or *DggsPolygon* that the shape represents. The type of the shape can be determined from the *getShapeType()* method and the *getShapeLocation()* method provides information about the location of the shape on the face of the polyhedron.

5.2.2.1 DGGS Cells

A point in the latitude/longitude domain is represented by a DggsCell object in the DGGS domain. A cell is constructed with a string cell identifier.

5.2.2.2 DGGS Linestrings

The *DggsLinestring* class is similar to the latitude/longitude *LatLongLinestring* class, but instead of *LatLongPoint* objects, it stores *DggsCell* objects. When converting from the latitude/longitude domain to the DGGS domain, each latitude/longitude point in the *LatLongLinestring* is represented by a *DggsCell* in the *DggsLinestring*.

5.2.2.3 DGGS Polygons

Similarly to the latitude/longitude *LatLongPolygon* structure, the *DggsPolygon* structure consists of a Dggs*LatLongLinestring* representing the outer ring of the polygon and an array of zero or more Dggs*LatLongLinestring* structures representing the inner rings or holes.

5.2.2.4 Native Structures

The representations of native data structures are in the native_structures.py file.

These structures are internal to the DGGS Python API and should not be used directly.

5.3 API Functions

The *Eaggr* class contains the methods to access the OpenEAGGR library functionality. It is constructed with one of the *DggsModel* enumeration values to determine the type of DGGS implementation that should be used. This section describes each of the methods on the *Eaggr* class.

5.3.1 *get_version*

This method can be used to determine the version of the OpenEAGGR library being used. It returns the OpenEAGGR version as a string.

5.3.2 *convert_point_to_dggs_cell*

This method converts a single point to a DGGS cell. It is provided to allow a single *LatLongPoint* object to be converted to a *DggsCell* object without the need to build up the more complex shape structures required for the other methods.

5.3.3 *convert_shapes_to_dggs_shapes*

This method converts shapes in the latitude/longitude domain to the DGGS domain. The method takes a list of shape objects in the latitude/longitude domain (i.e. *LatLongPoint*, *LatLongLinestring* or *LatLongPolygon*) and returns a list of *DggsShape* objects (one for each object in the supplied array).

5.3.4 *convert_shape_string_to_dggs_shapes*

This method converts shapes in the latitude/longitude domain defined in a standard string format to the DGGS domain. The formats supported are detailed in the *ShapeStringFormat* enumeration. The method takes the string defining the shapes, the format of the string and an accuracy that should be assigned to each of the points defined in the shape string. The method returns a list of *DggsShape* objects, one for each shape defined in the input string.

5.3.5 *convert_dggs_cell_to_point*

This method converts a single *DggsCell* object to a *LatLongPoint* object. It is provided as the reverse conversion equivalent of the *convert_point_to_dggs_cell* method (5.3.2) and is a simplified version of the *convert_dggs_cells_to_points* method (avoiding the need to create an array to wrap a single point).

5.3.6 *convert_dggs_cells_to_points*

This method converts cells in the DGGS domain to points in the latitude/longitude domain. The point will represent the centre of the DGGS cell and the accuracy of the point represents the area of the DGGS cell in metres squared. The method takes a list of *DggsCell* objects and returns a list of *LatLongPoint* objects (one for each *DggsCell* object in the supplied array).

5.3.7 *convert_dggs_cells_to_shape_string*

This method converts cells in the DGGS domain to a string format representing the points in the latitude/longitude domain. The cells are supplied as an array of *DggsCell* objects and the required format of the output string should be specified. The method returns a string representing the points generated by converting the *DggsCells*.

5.3.8 *get_dggs_cell_parents*

This method returns the parent cell(s) for a supplied cell. Parent cells are the cells in the resolution below that share area with the supplied cell. The parent cell(s) are returned as a list of *DggsCells*.

5.3.9 *get_dggs_cell_children*

This method returns the child cells for a supplied cell. Child cells are the cells in the resolution above that share area with the supplied cell. The child cells are returned a list of *DggsCells*.

5.3.10 *get_cell_siblings*

This method returns the sibling cells for a supplied cell. Sibling cells are the cells in the same resolution as the supplied cell that share a parent. The sibling cells are returned as a list of *DggsCells*.

5.3.11 *get_bounding_dggs_cell*

This method takes a list of *DggsCells* and returns the smallest cell that contains all of the supplied cells.

5.3.12 *create_dggs_kml_file*

This method allows a list of *DggsCells* to be exported to a KML file for visualisation in an external application, such as Google Earth. The file name and the array of cells are supplied to the function and a KML file will be created at the requested location. The KML file will contain a marker for the centre of each cell in the supplied array and a polygon that defines the boundary of the cell.

5.3.13 *convert_dggs_cell_outline_to_shape_string*

This method exports the geometry of the DGGS cell as a WKT or GeoJSON string. It is exported as a polygon (with no inner rings) representing the boundary of the cell. The cell is supplied as a *DggsCell* object and the required format of the output string should be specified. The method returns a string containing the cell boundary definition in the requested format.

5.3.14 *compare_shapes*

This method allows spatial analysis of two *DggsShape* objects. The spatial analysis to be performed is defined using the *AnalysisType* enumeration. The shapes for comparison are passed in as a base shape (*baseShape*) and a comparison shape (*comparisonShape*). For some spatial analysis operations, e.g. *EQUALS*, it does not matter which shape is passed into which parameter. For other operations, e.g. *CONTAINS*, the order of the shapes does matter. The DGGS library uses the base shape as the main geometry and performs the operation on the comparison geometry. In the case of *CONTAINS*, for example, the function will return true if the *baseShape CONTAINS comparisonShape*.

5.3.15 *Internal Methods*

There are two other methods in the *Eaggr* class (*_deallocate_dggs_shapes* and *_get_last_error_message*). These methods are used internally by other methods on the class and should not be used by client applications.

6 PostgreSQL Extension

The OpenEAGGR PostgreSQL extension provides the ability to use the OpenEAGGR functionality within queries on a PostgreSQL database. Geometry data can be converted to DGGS cells to be stored in a database table or used as part of a query. Examples of the use of the PostgreSQL extension can be found in Appendix D.

The PostgreSQL Extension can only be used with the version of PostgreSQL that it is built against. The extension currently builds against PostgreSQL version 9.5.

6.1 Installation

During the build process for the PostgreSQL extension, the lib_eaggr_postgres.dll library and all of its dependencies are placed in the EAGGRPostGIS/deploy folder, in a folder that corresponds to the build configuration (e.g. Release64). In order to install the PostgreSQL extension, several libraries and supporting files need to be copied.

- The files eaggr--x.x.sql and eaggr.control need to be copied from …/EAGGRPostGIS/deploy/ to the PostgreSQL installation, within the folder …/PostgreSQL/9.5/share/extension/
- The file lib_eaggr_postgres.dll needs to be copied from …/EAGGRPostGIS/Release64 to the PostgreSQL installation, within the folder …/PostgreSQL/9.5/lib
- The dependencies of the OpenEAGGR PostgreSQL extension need to be added to the PATH environment variable. One option to do this is to copy them from …/EAGGRPostGIS/deploy/ to the PostgreSQL installation, within the folder …/PostgreSQL/9.5/bin which is usually placed on the PATH by the PostgreSQL installer.
    - Note that if PostGIS is installed there can be conflicts between the OpenEAGGR dependencies and the libraries installed with PostGIS (libraries with the same name that have been compiled with different options). Care should be taken to resolve these issues. Ideally all software should be compiled from source using the same compiler with the same options, however if this is not possible then it may be sufficient to replace the PostGIS libraries with the OpenEAGGR equivalents, since the OpenEAGGR libraries were compiled with most options enabled and testing of the OpenEAGGR library has not revealed any problems.

After these files have been copied, the following needs to be entered as an SQL statement for each database in order to load the DGGS extension and make the functions available in PostgreSQL.

```
        CREATE EXTENSION eaggr
```

6.2 Functions

The eaggr—x.x.sql file contains the definitions of the functions created in PostgreSQL by the DGGS extension. This section describes each of the functions that are part of the extension.

6.2.1 *EAGGR_Version*

This method can be used to determine the version of the OpenEAGGR library being used. It returns the OpenEAGGR version as a string.

6.2.2 *EAGGR_ToCells*

This method converts shapes in a WKT string to the DGGS domain. The method takes a list of shape objects in the form of a WKT string, a double indicating the required accuracy and a string indicating the DGGS model (i.e. ISEA4T or ISEA3H), and returns a string representing the shapes in the DGGS domain. The shapes appear in the string in the same order they appeared in the WKT string argument. Shapes are separated by a forward slash character, linestrings within the shapes are separated by a colon character and cells within the linestrings are separated by a semi-colon character. Polygons in this representation use the first linestring to represent the outer linestring and all subsequent linestrings to represent the inner rings of the polygon. Each new cell, linestring and polygon in the shape string, is preceded by a shape identification character. This allows linestrings and polygons with no inner rings to be differentiated. The characters ‘1’, ‘2’ and ‘3’, are used to identify a cell, a linestring and a polygon respectively. This character is separated from the cells that make up the shape, by a tilde character.

6.2.3 *EAGGR_ToCellArray*

This method converts shape strings in the format produced by *EAGGR_ToCells* to an array of DGGS cells. This method takes a list of shape objects in the DGGS domain, represented by a string in the format produced by *EAGGR_ToCells*. The method returns a string array of DGGS cells, where the cells are those found in the shape string argument and appear in the same order.

6.2.4 *EAGGR_GetBoundingCell*

This method identifies the DGGS cell that bounds an array of DGGS cells. This method takes in a string array of DGGS cells and a string indicating the DGGS model (i.e. ISEA4T or ISEA3H), and returns a string representing the DGGS cell that bounds the array of DGGS cells used as an argument.

6.2.5 *EAGGR_CellGeometry*

This method converts the geometry of a DGGS cell to a shape string in WKT format. This method takes a string representing a DGGS cell and a string indicating the DGGS model (i.e. ISEA4T or ISEA3H), and returns a string representing the polygon boundary of the DGGS cell in WKT format.

6.2.6 *EAGGR_CellToPoint*

This method converts a DGGS cell to a shape string in WKT format. This method takes a string representation of a DGGS cell and a string indicating the DGGS model (i.e. ISEA4T or ISEA3H), and returns a string representing the point at the centre of the cell in WKT format.

6.2.7 *EAGGR_ShapeComparison*

This method compares two shape strings in the format produced by *EAGGR_ToCells*. The first input to the method is a string representation of a *DGGS_AnalysisType* (see 4.1.6), indicating the analysis to perform on the shapes. This should be provided in capitals. The shape strings are then provided, with the base string first and then the comparison string, both in the format produced by *EAGGR_ToCells*. Each string should only contain one shape. This method uses the *EAGGR_CompareShapes* method on the C API, so the order of the shapes can be important, as discussed in 3.3.15. The final input parameter is a string indicating the DGGS model (i.e. ISEA4T or ISEA3H). The method returns either true or false, representing the outcome of the shape comparison.

7 Elasticsearch Plug-in

The OpenEAGGR Elasticsearch plug-in allows indices to be created for spatial data in the DGGS domain. It extends the existing geo_point and geo_shape datatypes to allow indexing of DGGS cells. The index mappings specify the DGGS model and accuracy to use for the index. For point fields the DGGS cell at the specified accuracy and all of the cell parents are indexed. For shape fields, the DGGS cell that bounds the shape and all of the bounding cell's parents are added to the index. Once indexed the data can be queried by supplying either a DGGS cell or a latitude/longitude location with an associated accuracy.

The Elasticsearch plug-in can only be used with the version of Elasticsearch that the plug-in is built against. Currently the plug-in is built against Elasticsearch version 2.4.0. The windows distribution of this version is supplied in …/EAGGRElasticsearch/testData/elasticsearch.

7.1 Installation

The Elasticsearch installation contains a *plugins* folder where plug-ins should be added in order that Elasticsearch loads them when the process starts. Elasticsearch has a utility that can install plugins from a zip file that has been generated by the OpenEAGGR build process.

- Build the plug-in using the ant build.xml file in …/EAGGRElasticsearch/
  - This should generate a 'jar' folder which contains the built plug-in and its dependencies contained in a zip file
- Open a command line console in the Elasticsearch distribution's 'bin' directory
- Run 'plugin install file:///path/to/built/plugin.zip' where the path to the plug in is the path to the zip file in the generated jar folder
  - The plugin requires permissions to extract the dependent native libraries from the jar and to load the libraries at runtime. The plugin installation process will prompt to confirm the grant of these permissions

7.2 Launching Elasticsearch

Elasticsearch is run using a batch file in the …/bin/ folder of the Elasticsearch distribution. Once the plugin is installed it will be automatically loaded by Elasticsearch when it is launched.

7.3 Elasticsearch Plug-in Functionality

Examples of the use of the OpenEAGGR plug-in for Elasticsearch can be found in Appendix E.

The Elasticsearch plug-in provides new datatypes *dggs_geo_point* and *dggs_geo_shape* which extend the existing *geo_point* and *geo_shape* types. These datatypes take two additional parameters:

- accuracy: the size in metres squared of the DGGS cell the location data should be converted to.
- model: the DGGS model to be used (i.e. ISEA4T or ISEA3H). This is an optional parameter and ISEA4T is used by default if it is not specified.

Once documents are added to the index there are two methods of querying the data:

- Specifying a 'match' query on the *dggs-cellid* property of the *dggs_geo_point* or *dggs_geo_shape* field, supplying the Id of the DGGS cell to query against.
- Using a dggs_cell query on the *dggs-cellid* property of the *dggs_geo_point* or *dggs_geo_shape* field, supplying the latitude, longitude and accuracy of a point representing the cell to query with.

8 Appendix A – C API Code Examples

This appendix shows some examples of how to use the OpenEAGGR C API. The OpenEAGGR Test Harness provides a comprehensive example of using all of the functions in the C API (SystemTests/DLL/DLLTest.cpp).

8.1 Convert Points to Cells

```
static const unsigned short NO_OF_POINTS = 2U;
DGGS_LatLongPoint latLongPoints[NO_OF_POINTS] =
{
  { 1.234, 2.345, LatLong::SphericalAccuracyPoint::AngleAccuracyToSquareMetres(1.0e-5)},
  { 3.456, 4.567, LatLong::SphericalAccuracyPoint::AngleAccuracyToSquareMetres(1.0e-5)}
};

DGGS_Handle handle = NULL;
DGGS_ReturnCode returnCode;

returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);

DGGS_Cell cells[NO_OF_POINTS];
returnCode = EAGGR_ConvertPointsToDggsCells(handle, latLongPoints, NO_OF_POINTS, cells);
```

8.2 Convert Cells to Points

```
DGGS_Handle handle = NULL;
DGGS_ReturnCode returnCode;

DGGS_Cell cells[] =
{
  "07231131111113100331032", "07012000001303022011321"
};
const unsigned short noOfCells = sizeof(cells) / sizeof(cells[0]);

returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);
ASSERT_EQ(DGGS_SUCCESS, returnCode);

DGGS_LatLongPoint points[noOfCells];
returnCode = EAGGR_ConvertDggsCellsToPoints(handle, cells, noOfCells, points);
```

8.3 Convert Latitude/Longitude Shapes to DGGS Shapes

```
DGGS_LatLongPoint point1 =
{ 1.234, 2.345, LatLong::SphericalAccuracyPoint::AngleAccuracyToSquareMetres(1.0e-5)};

DGGS_LatLongPoint point2 =
{ 3.456, 4.567, LatLong::SphericalAccuracyPoint::AngleAccuracyToSquareMetres(1.0e-5)};

static const unsigned short NO_OF_SHAPES = 4U;
DGGS_LatLongShape latLongShapes[NO_OF_SHAPES];

// Add two points to the shapes
latLongShapes[0].m_type = DGGS_LAT_LONG_POINT;
latLongShapes[0].m_data.m_point = point1;
latLongShapes[1].m_type = DGGS_LAT_LONG_POINT;
latLongShapes[1].m_data.m_point = point2;

// Add one linestring to the shapes
latLongShapes[2].m_type = DGGS_LAT_LONG_LINESTRING;

static const unsigned short NO_OF_LINESTRING_POINTS = 2U;
latLongShapes[2].m_data.m_linestring.m_noOfPoints = NO_OF_LINESTRING_POINTS;
latLongShapes[2].m_data.m_linestring.m_points
  = static_cast<DGGS_LatLongPoint *>(malloc(NO_OF_LINESTRING_POINTS * sizeof(DGGS_LatLongPoint)));
latLongShapes[2].m_data.m_linestring.m_points[0] = point1;
latLongShapes[2].m_data.m_linestring.m_points[1] = point2;

// Add one polygon to the shapes
latLongShapes[3].m_type = DGGS_LAT_LONG_POLYGON;

static const unsigned short NO_OF_OUTER_RING_POINTS = 2U;
```

```
latLongShapes[3].m_data.m_polygon.m_outerRing.m_noOfPoints = NO_OF_OUTER_RING_POINTS;
latLongShapes[3].m_data.m_polygon.m_outerRing.m_points
= static_cast<DGGS_LatLongPoint *>(malloc(NO_OF_OUTER_RING_POINTS * sizeof(DGGS_LatLongPoint)));
latLongShapes[3].m_data.m_polygon.m_outerRing.m_points[0] = point1;
latLongShapes[3].m_data.m_polygon.m_outerRing.m_points[1] = point2;

static const unsigned short NO_OF_POLYGON_INNER_RINGS = 2U;
latLongShapes[3].m_data.m_polygon.m_noOfInnerRings = NO_OF_POLYGON_INNER_RINGS;
latLongShapes[3].m_data.m_polygon.m_innerRings
  = static_cast<DGGS_LatLongLinestring *>
    (malloc(NO_OF_POLYGON_INNER_RINGS * sizeof(DGGS_LatLongLinestring)));

static const unsigned short NO_OF_1ST_INNER_RING_POINTS = 2U;
latLongShapes[3].m_data.m_polygon.m_innerRings[0].m_noOfPoints = NO_OF_1ST_INNER_RING_POINTS;
latLongShapes[3].m_data.m_polygon.m_innerRings[0].m_points
  = static_cast<DGGS_LatLongPoint *>(malloc(NO_OF_1ST_INNER_RING_POINTS * sizeof(DGGS_LatLongPoint)));
latLongShapes[3].m_data.m_polygon.m_innerRings[0].m_points[0] = point1;
latLongShapes[3].m_data.m_polygon.m_innerRings[0].m_points[1] = point2;

static const unsigned short NO_OF_2ND_INNER_RING_POINTS = 2U;
latLongShapes[3].m_data.m_polygon.m_innerRings[1].m_noOfPoints = NO_OF_2ND_INNER_RING_POINTS;
latLongShapes[3].m_data.m_polygon.m_innerRings[1].m_points
  = static_cast<DGGS_LatLongPoint *>(malloc(NO_OF_2ND_INNER_RING_POINTS * sizeof(DGGS_LatLongPoint)));
latLongShapes[3].m_data.m_polygon.m_innerRings[1].m_points[0] = point1;
latLongShapes[3].m_data.m_polygon.m_innerRings[1].m_points[1] = point2;

DGGS_Handle handle = NULL;
DGGS_ReturnCode returnCode;

returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);

DGGS_Shape * shapes;
returnCode = EAGGR_ConvertShapesToDggsShapes(handle, latLongShapes, NO_OF_SHAPES, &shapes);

// Use the DGGS shapes stored in the shapes variable


returnCode = EAGGR_DeallocateDggsShapes(handle, &shapes, NO_OF_SHAPES);

returnCode = EAGGR_CloseDggsHandle(&handle);
```

8.4 Convert DGGS Cells to Latitude/Longitude Points

```
DGGS_Handle handle = NULL;
DGGS_ReturnCode returnCode;

DGGS_Cell cells[] =
{
  "07231131111113100331032", "07012000001303022011321"
};
const unsigned short noOfCells = sizeof(cells) / sizeof(cells[0]);

returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);

DGGS_LatLongPoint points[noOfCells];
returnCode = EAGGR_ConvertDggsCellsToPoints(handle, cells, noOfCells, points);

// Use the Lat/Long points stored in the points variable

returnCode = EAGGR_CloseDggsHandle(&handle);
```

8.5 Get Parent, Child and Sibling Cells

```
DGGS_Handle handle = NULL;
DGGS_ReturnCode returnCode;

DGGS_Cell cell = "07012212222221011101013";

returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);
```

```
    ASSERT_EQ(DGGS_SUCCESS, returnCode);

    DGGS_Cell parentCells[DGGS_MAX_PARENT_CELLS] =
    {};
    unsigned short noOfParentCells = 0U;

    DGGS_Cell childCells[DGGS_MAX_CHILD_CELLS] =
    {};
    unsigned short noOfChildCells = 0U;

    DGGS_Cell siblingCells[DGGS_MAX_SIBLING_CELLS] =
    {};
    unsigned short noOfSiblingCells = 0U;

    returnCode = EAGGR_GetDggsCellParents(handle, cell, parentCells, &noOfParentCells);

    // Use the parent cells stored in the parentCells variable

    returnCode = EAGGR_GetDggsCellChildren(handle, cell, childCells, &noOfChildCells);

    // Use the child cells stored in the childCells variable

    returnCode = EAGGR_GetDggsCellSiblings(handle, cell, siblingCells, &noOfSiblingCells);

    // Use the sibling cells stored in the siblingCells variable

    returnCode = EAGGR_CloseDggsHandle(&handle);
```

8.6 Spatial Analysis

```
    DGGS_Handle handle = NULL;
    DGGS_ReturnCode returnCode;
    bool shapeComparisonResult;

    DGGS_AnalysisType spatialAnalysisType = EQUALS;

    returnCode = EAGGR_OpenDggsHandle(DGGS_ISEA4T, &handle);

    DGGS_Cell cellOne = "0701";
    DGGS_Cell cellTwo = "0702";

    DGGS_Shape baseShape;
    baseShape.m_type = DGGS_CELL;
    strncpy(baseShape.m_data.m_cell, cellOne, DGGS_MAX_CELL_STRING_LENGTH);

    DGGS_Shape comparisonShape;
    comparisonShape.m_type = DGGS_CELL;
    strncpy(comparisonShape.m_data.m_cell, cellTwo, DGGS_MAX_CELL_STRING_LENGTH);

    returnCode
      = EAGGR_CompareShapes(handle, spatialAnalysisType, baseShape, comparisonShape, &shapeComparisonResult);
```

9 Appendix B – Java API Code Examples

This appendix shows some examples of how to use the OpenEAGGR Java API. The EAGGR Java Junit tests provide a comprehensive example of using all of the functions in the Java API (tests/uk/co/riskaware/eaggr/EaggrTest.java).

9.1 Convert Latitude/Longitude Point to DGGS Cell

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA4T);

        final LatLongPoint latLongPoint = new LatLongPoint(1.234, 2.345, 3.879);

        final DggsCell dggsCell = eaggr.convertPointToDggsCell(latLongPoint);
```

9.2 Convert DGGS Cell to Latitude/Longitude Point

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA4T);

        final DggsCell cell = new DggsCell("07231131111113100331032");

        final LatLongPoint point = eaggr.convertDggsCellToPoint(cell);
```

9.3 Convert Latitude/Longitude Shapes to DGGS Shapes

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA4T);

        final LatLongShape[] latLongShapes = new LatLongShape[4];

        latLongShapes[0] = new LatLongShape(new LatLongPoint(1.234, 2.345, 3.879));
        latLongShapes[1] = new LatLongShape(new LatLongPoint(3.456, 4.567, 3.879));

        final LatLongLinestring linestring = new LatLongLinestring();
        linestring.addPoint(new LatLongPoint(1.234, 2.345, 3.879));
        linestring.addPoint(new LatLongPoint(3.456, 4.567, 3.879));
        latLongShapes[2] = new LatLongShape(linestring);

        final LatLongLinestring outerRing = new LatLongLinestring();
        outerRing.addPoint(new LatLongPoint(1.234, 2.345, 3.879));
        outerRing.addPoint(new LatLongPoint(3.456, 4.567, 3.879));

        final LatLongLinestring innerRing1 = new LatLongLinestring();
        innerRing1.addPoint(new LatLongPoint(1.234, 2.345, 3.879));
        innerRing1.addPoint(new LatLongPoint(3.456, 4.567, 3.879));

        final LatLongLinestring innerRing2 = new LatLongLinestring();
        innerRing2.addPoint(new LatLongPoint(1.234, 2.345, 3.879));
        innerRing2.addPoint(new LatLongPoint(3.456, 4.567, 3.879));

        final LatLongPolygon polygon = new LatLongPolygon(outerRing);
        polygon.addInnerRing(innerRing1);
        polygon.addInnerRing(innerRing2);
        latLongShapes[3] = new LatLongShape(polygon);

        final DggsShape[] dggsShapes = eaggr.convertShapesToDggsShapes(latLongShapes);
```

9.4 Convert DGGS Cells to Latitude/Longitude Points

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA4T);

        final DggsCell[] cellIds = new DggsCell[] { new DggsCell("07231131111113100331032"),
                new DggsCell("07012000001303022011321") };

        final LatLongPoint[] points = eaggr.convertDggsCellsToPoints(cellIds);
```

9.5 Get Parent, Child and Sibling Cells

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA3H);

        final DggsCell[] parentCells = eaggr.getCellParents(new DggsCell("01033,6"));

        final DggsCell[] childCells= eaggr.getCellChildren(new DggsCell("01031,3"));


        final DggsCell[] siblingCells= eaggr.getCellSiblings(new DggsCell("01031,3"));
```

9.6 Spatial Analysis

```
        final Eaggr eaggr = new Eaggr (DggsModel.ISEA3H);

        final DggsCell dggsCell = new DggsCell("00021,1");
        final DggsShape dggsShapeCell= new DggsShape(dggsCell);


        final DggsCell anotherDggsCell = new DggsCell("0001");
        final DggsShape anotherDggsShapeCell= new DggsShape(anotherDggsCell);


        final boolean comparisonResult
          = eaggr.compareShapes(dggsShapeCell, anotherDggsShapeCell, AnalysisType.EQUALS);
```

10 Appendix C – Python API Code Examples

This appendix shows some examples of how to use the OpenEAGGR Python API. The EAGGR Python unit tests provide a comprehensive example of using all of the functions in the Python API (python_tests/eaggr/test_eaggr.py). Like the Java API, all the unit tests will run with both Python 2 and Python 3 interpreters.

10.1 Convert Latitiude/Longitude Point to DGGS Cell

```
# Create the lat/long point
lat_long_point = LatLongPoint(1.234, 2.345, 3.884);
# Initialise the DGGS model
eaggr = Eaggr(Model.ISEA4T)
# Convert the lat/long point
dggs_cell = eaggr.convert_point_to_dggs_cell(lat_long_point)
```

10.2 Convert DGGS Cell to Latitude/Longitude Point

```
# Create the DGGS cell
dggs_cell = DggsCell("07231131111113100331001")
# Initialise the DGGS model
eaggr = Eaggr(Model.ISEA4T)
# Convert the DGGS cell
lat_long_point =  eaggr.convert_dggs_cell_to_point(dggs_cell)
```

10.3 Convert Latitude/Longitude Shapes to DGGS Shapes

```
# Create the lat/long shapes
point1 = LatLongPoint(1.234, 2.345, 3.884)
point2 = LatLongPoint(3.456, 4.567, 3.884)
linestring = LatLongLinestring()
linestring.add_point(point1)
linestring.add_point(point2)
polygon = LatLongPolygon(linestring)
polygon.add_inner_ring(linestring)
polygon.add_inner_ring(linestring)
# Add the lat/long shapes to the input list
lat_long_shapes = []
lat_long_shapes.append(point1)
lat_long_shapes.append(point2)
lat_long_shapes.append(linestring)
lat_long_shapes.append(polygon)
# Convert the lat/long shapes
eaggr = Eaggr(Model.ISEA4T)
dggs_shapes = eaggr.convert_shapes_to_dggs_shapes(lat_long_shapes)
```

10.4 Convert DGGS Cells to Latitude/Longitude Points

```
# Create the DGGS cells
dggs_cells = [DggsCell("07231131111113100331032"), DggsCell("07012000001303022011321")]
# Convert the DGGS cells
eaggr = Eaggr(Model.ISEA4T)
lat_long_points = eaggr.convert_dggs_cells_to_points(dggs_cells)
```

10.5 Get Parent, Child and Sibling Cells

```
# Create the DGGS cell
dggs_cell = DggsCell("07012212222221011101013")
# Get its parent cell
eaggr = Eaggr(Model.ISEA4T)
parents = eaggr .get_dggs_cell_parents(dggs_cell)
# Get its child cells
children = eaggr.get_dggs_cell_children(dggs_cell)
```

```
        # Get its sibling cells
        siblings = eaggr.get_dggs_cell_siblings(dggs_cell)
```

10.6Spatial Analysis

```
        eaggr = Eaggr(Model.ISEA4T)
        # Create a cell
        dggs_shape_cell = DggsShape(DggsCell("0700"), DggsShapeLocation.ONE_FACE)
        # Compare the cell with another cell
        another_dggs_shape_cell = DggsShape(DggsCell("0701"), DggsShapeLocation.ONE_FACE)
        analysisResult = eaggr.compare_dggs_shapes(dggs_shape_cell, another_dggs_shape_cell,
DggsAnalysisType.EQUALS)
```

11 Appendix D – PostgreSQL Code Examples

This appendix shows some examples of how to use the PostgreSQL extension for accessing available methods on the OpenEAGGR API. The EAGGR Post GIS Test Harness provides a comprehensive example of using all of the functions in the extension (UnitTests folder).

11.1 EAGGR_Version

```
        -- Retrieves the current DGGS version number
        SELECT EAGGR_Version()
```

11.2 EAGGR_ToCells

```
      -- Converts a point to a DGGS cell
      SELECT EAGGR_ToCells('POINT(2.345 1.234)', 3.879, 'ISEA4T')
      -- Converts a linestring to DGGS cells
      SELECT EAGGR_ToCells('LINESTRING(2.345 1.234, 4.567 3.456)', 3.879, 'ISEA4T')
      -- Converts a polygon to DGGS cells
      SELECT EAGGR_ToCells('POLYGON((2.345 1.234, 4.567 3.456), (6.789 5.678, 8.901 7.890))', 3.879,
'ISEA4T')
      -- Converts multiple shapes to DGGS cells
      SELECT EAGGR_ToCells('GEOMETRYCOLLECTION(POINT(2.345 1.234),LINESTRING(2.345 1.234, 4.567
3.456),POLYGON((2.345 1.234, 4.567 3.456), (6.789 5.678), (8.901 7.890)))', 3.879, 'ISEA4T')
```

11.3 EAGGR_ToCellArray

```
        -- Converts a string representing a DGGS cell, to an array of DGGS cells
        SELECT EAGGR_ToCellArray('1~07231131111113100331001')
        -- Converts a string representing a DGGS linestring, to an array of DGGS cells
        SELECT EAGGR_ToCellArray('2~07231131111113100331001;07012000001303022011321')
        -- Converts a string representing a DGGS polygon, to an array of DGGS cells
        SELECT
EAGGR_ToCellArray('3~07231131111113100331001;07012000001303022011321:07010120000123111312330;0701013333011023
1202021')
        -- Converts a string representing a multiple DGGS shapes, to an array of DGGS cells
        SELECT
EAGGR_ToCellArray('1~07231131111113100331001/2~07231131111113100331001;07012000001303022011321/3~072311311111
13100331001;07012000001303022011321:07010120000123111312330:07010133330110231202021')
```

11.4 EAGGR_GetBoundingCell

```
        -- Identifies a bounding DGGS cell for an array of DGGS cells
        SELECT EAGGR_GetBoundingCell('{07231,0723102,07230130}', 'ISEA4T')
        -- Note that ISEA3H cells need to be put in “” so that the ‘,’ is not used as an array separator
        SELECT EAGGR_GetBoundingCell('{"07040,0", "07040,1", "07040,-1"}', 'ISEA3H')
```

11.5 EAGGR_CellGeometry

```
        -- Identifies the geometry of a DGGS cell
        SELECT EAGGR_CellGeometry('07231131111113100331032', 'ISEA4T')
```

11.6 EAGGR_CellToPoint

```
        -- Convert a DGGS cell to a point in ISEA4T
        SELECT EAGGR_CellToPoint('07231131111113100331001', 'ISEA4T')
```

11.7EAGGR_ShapeComparison

```
        -- Compares two shape strings that represent the same cell, using the EQUALS analysis type
        SELECT EAGGR_ShapeComparison(‘EQUALS’,'1~0000','1~0000',’ISEA4T’)
        -- Compares two shape strings that represent different linestrings, using the CONTAINS analysis type
        SELECT EAGGR_ShapeComparison(‘CONTAINS’,'2~0001;0002;0003’,’2~0001;0002’,’ISEA4T’)
        -- Compares two shape strings that represent a polygon with an inner ring and a cell, using the
OVERLAPS analysis type
        SELECT
EAGGR_ShapeComparison(‘OVERLAPS’,'3~00020,2;00024,0;00024,4;00023,4;00020,3:00021,2;00023,1;00023,3',’1~00020
,2’,’ISEA3H’)
```

12 Appendix E – Elasticsearch Examples

This appendix shows some examples of how to use the PostgreSQL extension for accessing available methods on the OpenEAGGR API. The EAGGR Elasticsearch JUnit Tests provide a comprehensive example of using all of the functions accessible in the plug-in (tests/uk/co/riskaware/eaggr/elasticsearch folder). The examples here follow the same style as the examples in the Elasticsearch online documentation

12.1 Create Index

```
-- Creates an index called dggs with a point and shape index
PUT /dggs
{
  "mappings": {
    "city": {
      "properties": {
        "city": {"type": "string"},
        "dggs_point": {
          "type": "dggs_geo_point",
          "accuracy": 100,
          "model": "ISEA4T"
        },
        "dggs_point": {
          "type": "dggs_geo_shape",
          "accuracy": 100,
          "model": "ISEA4T"
        }
      }
    }
  }
}
```

12.2 Insert Document

```
-- Inserts a document into the index
POST /dggs/city/
{
  "city": "Anchorage",
  "dggs_point": {
    "lat": "61.2180556",
    "lon": "-149.9002778"
  },
  "dggs_shape": {
    "type": "linestring",
    "coordinates": [[-77.03653, 38.897676], [-77.009051, 38.889939]]
  }
}
```

12.3 Query Index Using Cell Id

```
-- Queries the index using a query clause
POST /dggs/_search
{
  "query": {
    "match": {
      "dggs_point.dggs-cellid": "00023302113"
    }
  }
}
-- Queries the index using a filter clause
POST /dggs/_search
{
  "query": {
    "match_all" : {}
  },
  "filter": {
    "term": {
      "dggs_point.dggs-cellid": "00023302113"
    }
  }
}
```

12.4Query Index Using Point and Accuracy

```
      -- Queries the index using a query clause
      POST /dggs/_search
      {
        "query": {
          "dggs_cell": {
            "dggs_point.dggs-cellid": {
              "lat": "61.2180556",
              "lon": "-149.9002778",
              "accuracy": "100"
            }
          }
        }
      }
      -- Queries the index using a filter clause
      POST /dggs/_search
      {
        "query": {
          "match_all" : {}
        },
        "filter": {
          "dggs_cell": {
            "dggs_point.dggs-cellid": {
              "lat": "61.2180556",
              "lon": "-149.9002778",
              "accuracy": "100"
            }
          }
        }
      }
```

Riskaware
capability through technology

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