GENERAL READ ME TEXT

This a quick guide to using the triangle library that is designed to support tessellation of Lagrangian point data. A somewhat conversation description of the approach and historical development of the algorithms is available from the Genwest.com website Home => Publications => 11-001 Triangular Tessellation Documentation. For a more in depth look at the theory the reader should have a look at the references for the paper.

What is presented here is an advanced beta release of the library. By “advanced beta” I mean to convey the following: 1) It is functional (I use it all the time); and, 2) I still occasionally run into a bug in the code (example – reference to null pt, etc). Using these routines you may also run into a bug, or even unexpected behavior. If you send me an email I will fix them and/or try and explain what is going wrong. The actual library routines are written in Java and contain a bit over 7000 lines of code. The source code and standard JavaDoc associated with it are contained in this release.

The intension of this release is that it can be used by anyone on any platform or operating system with little or no Java or any other coding necessary. It is not quite done in that regard and is under active development. It is intended to include a complete programming independent XML driver by about the end of calendar year 2015. At present all sample programs are invoked using a single command line call to an executable Jar file which is created by running a simple Unix/Linux script on Mac and Linux machines or a PowerShell script on Windows machines. From a user's point of view the interface is identical, except that the string argument in the PowerShell script call needs to be surrounded by quotes. This guide will present a walk through tutorial on how to use the triangle library and interact with it.

I will start with a warning. Java controls its “name space” which determines how to handle conflicting or redundant names by “packages”, which are really sequences of directories in the file system. The library and sample programs described are referenced within the code and include the following two paths:

hbl/jag/tri/lib (containing all the triangle library files)

hbl/jag/tri/sam (containing all the sample program files)

For the included script to work these directories must be present in the tessellation folder which should be the working directory where command line calls are made. The tessellation director on the ftp site is set up exactly as it should be for the tutorial. For convenience it will also include a .zip file of all the files that you should be able to down load as a unit and then unzip (preserving the directory structure) and get the right set-up.

The second point has to do with which version of the JRT you happen to have on the machine that you are using. I have run these routines on JTR's from 1.5... to 1.8 with out a problem, but if you compile the byte code using javac on a later version (say 1.8.), and then run it on a earlier version (say 1.6.) then I have received a “run time error” indicating an “unsupported call”. To keep this from being a problem the script will re-compile the byte code each time it is run so that things will be in sync. This takes a few extra seconds and amounts to a “clean build” but it seems to work.

GETTING STARTED: STEP 1

The first exercise is to build a Java Jar file of all the Classes in the triangle library. To see what these are you can open the index.html file (found in the Doc directory) using any Browser for the standard JavaDoc coverage of the public and protected interface. For more details you can open any of the *.java files in hbl=>jag=>tri=>lib to see the actual code. (All this is meant to be open source)

A Java Jar file is nothing more then a .zip compression of all the compiled *.class (plus possibly other source code) files in the .jar along with a MANIFEST file that contains a dictionary of META-DATA about the JAR that the JRT java tool can use.

To create the trilib.jar navigate in the command line (or PowerShell) window to the “tessellation” directory and type the command:

sh makeTriJar.sh (on Linux like machines)
or
./makeTriJar.pl1 (on Windows like machines)

The script should run annotating it progress in the command window with something like the following:

```
script will create library
the library package is hbl/jag/tri/lib/
rebuilding library class files
building new trilib.jar
Class-Path: .
Manifest-Version: hbl.jag.tri.3.1.4
Version-Date: 20150801
added manifest
adding: hbl/jag/tri/lib/Arc.class(in = 955) (out= 537)(deflated 43%)
adding: hbl/jag/tri/lib/BNA2VERDAT$1.class(in = 213) (out= 165)(deflated 22%)
adding: hbl/jag/tri/lib/BNA2VERDAT$Segment.class(in = 758) (out= 420)(deflated 44%)
adding: hbl/jag/tri/lib/BNA2VERDAT.class(in = 3885) (out= 2206)(deflated 43%)
adding: hbl/jag/tri/lib/BnaReadCheck.class(in = 2625) (out= 1566)(deflated 40%)
adding: hbl/jag/tri/lib/CoordTransform.class(in = 2866) (out= 1769)(deflated 38%)
adding: hbl/jag/tri/lib/DagData.class(in = 21600) (out= 11017)(deflated 48%)
adding: hbl/jag/tri/lib/DisplayWindow.class(in = 1074) (out= 668)(deflated 37%)
adding: hbl/jag/tri/lib/Edge.class(in = 1310) (out= 717)(deflated 45%)
adding: hbl/jag/tri/lib/EdgePoint.class(in = 5832) (out= 3089)(deflated 47%)
adding: hbl/jag/tri/lib/EditMetaData.class(in = 5049) (out= 2675)(deflated 47%)
adding: hbl/jag/tri/lib/Endian.class(in = 1979) (out= 1033)(deflated 47%)
adding: hbl/jag/tri/lib/GAxmlHandler.class(in = 10350) (out= 5089)(deflated 50%)
adding: hbl/jag/tri/lib/GISFORCST2SPLOTLST.class(in = 3378) (out= 1832)(deflated 45%)
adding: hbl/jag/tri/lib/ImportPolygons.class(in = 3071) (out= 1700)(deflated 44%)
adding: hbl/jag/tri/lib/Incircle.class(in = 3966) (out= 1848)(deflated 53%)
adding: hbl/jag/tri/lib/MOSS2SPLOTLST.class(in = 2472) (out= 1372)(deflated 44%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$1.class(in = 901) (out= 500)(deflated 44%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$2.class(in = 802) (out= 454)(deflated 43%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$3.class(in = 1220) (out= 662)(deflated 45%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$4.class(in = 1272) (out= 739)(deflated 41%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$5.class(in = 968) (out= 540)(deflated 44%)
```

adding: hbl/jag/tri/lib/MapEdit$MapFrame$6.class(in = 913) (out= 511)(deflated 44%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame$7.class(in = 1013) (out= 567)(deflated 44%)
adding: hbl/jag/tri/lib/MapEdit$MapFrame.class(in = 3028) (out= 1622)(deflated 46%)
adding: hbl/jag/tri/lib/MapEdit$MapsPanel$1.class(in = 1205) (out= 675)(deflated 43%)
adding: hbl/jag/tri/lib/MapEdit$MapsPanel$2.class(in = 3128) (out= 1676)(deflated 46%)
adding: hbl/jag/tri/lib/MapEdit$MapsPanel.class(in = 3928) (out= 2231)(deflated 43%)
adding: hbl/jag/tri/lib/MapEdit.class(in = 6874) (out= 3333)(deflated 51%)
adding: hbl/jag/tri/lib/MicroPoint.class(in = 443) (out= 321)(deflated 27%)
adding: hbl/jag/tri/lib/Node.class(in = 1271) (out= 697)(deflated 45%)
adding: hbl/jag/tri/lib/PlotContours$1.class(in = 943) (out= 542)(deflated 42%)
adding: hbl/jag/tri/lib/PlotContours$2.class(in = 1279) (out= 676)(deflated 47%)
adding: hbl/jag/tri/lib/PlotContours.class(in = 4184) (out= 2240)(deflated 46%)
adding: hbl/jag/tri/lib/PlotDepth$1.class(in = 865) (out= 524)(deflated 39%)
adding: hbl/jag/tri/lib/PlotDepth$2.class(in = 1201) (out= 655)(deflated 45%)
adding: hbl/jag/tri/lib/PlotDepth.class(in = 2851) (out= 1604)(deflated 43%)
adding: hbl/jag/tri/lib/PlotMaps$1.class(in = 882) (out= 534)(deflated 39%)
adding: hbl/jag/tri/lib/PlotMaps$2.class(in = 1218) (out= 672)(deflated 44%)
adding: hbl/jag/tri/lib/PlotMaps.class(in = 4678) (out= 2493)(deflated 46%)
adding: hbl/jag/tri/lib/PlotTri$1.class(in = 855) (out= 519)(deflated 39%)
adding: hbl/jag/tri/lib/PlotTri$2.class(in = 1191) (out= 658)(deflated 44%)
adding: hbl/jag/tri/lib/PlotTri.class(in = 3501) (out= 1989)(deflated 43%)
adding: hbl/jag/tri/lib/Polygon.class(in = 19977) (out= 10345)(deflated 48%)
adding: hbl/jag/tri/lib/Splot.class(in = 777) (out= 476)(deflated 38%)
adding: hbl/jag/tri/lib/ThinBNAMap$DisplayWindowWithSave.class(in = 2439) (out= 1149)(deflated 52%)
adding: hbl/jag/tri/lib/ThinBNAMap.class(in = 4855) (out= 2730)(deflated 43%)
adding: hbl/jag/tri/lib/TriNeighbor.class(in = 410) (out= 297)(deflated 27%)
adding: hbl/jag/tri/lib/VERSION.class(in = 920) (out= 530)(deflated 42%)
adding: hbl/jag/tri/lib/VariableSplots.class(in = 175) (out= 145)(deflated 17%)
adding: hbl/jag/tri/lib/Verdat.class(in = 1495) (out= 867)(deflated 42%)

In addition to the output in the command window there will be two new directories in the “tessellation” directory:

1) The first will be “trilib.jar” which will be the Java library of all the triangle library classes. If you are working in Java and using an API (for example Eclipse) you simply place this file in the class path of any project you are working on and the sample code you are working on should fill the code and link it in.

2) The second will be an unzipped version of the library in a directory called “LibraryJARcontent” which is just an human readable listing of what is in the “trilib.jar” file.

LINKING IN YOUR OWN CODE: STEP 2

The next exercise will be to link in your own code which can call the triangle library. To introduce this I have added a “HelloTess” sample program to the hbl=>jag=>tri=>sam directory. The annotated listing is as follows:

```
//////////////////////////////////////////////////
// put original program is hbl.jag.tri.sam package
//////////////////////////////////////////////////
package hbl.jag.tri.sam;

// import parts of the triangle.lib that are needed
import hbl.jag.tri.lib.DagData;
import hbl.jag.tri.lib.CoordTransform;

/**
 * Simple example of a user built driving program that uses
 * the triangle.lib.  It initializes a DagData object and
 * checks to see that it is non-null then prints out a
 * summary string.  All followed by the standard "Greeting"
 * @author jag
 *
 */
public class HelloTess {

	public static void main(String[] args){
		// define some points
		long[] x = new long[100];
		long[] y = new long[100];
		for(int i=0;i<100;i++){
			x[i] = (long)(1e6*Math.random());
			y[i] = (long)(1e6*Math.random());
		}
		// make call to library
		DagData dd = new DagData(x,y,CoordTransform.CARTESIAN);
		// echo output to show it works
		if(dd!=null){
			System.out.println("dd is not null");
			System.out.println(dd);
		}
		System.out.println("Hello Tess");
	}

}
```

This is clearly a toy routine, but it shows how to write a short calling program, access classes that are in the triangle library and create some output that indicates the linking worked.

To build a standalone, one-line executable Jar file incorporating this calling program we run the script again, but in this case we also include a single string command-line argument which must be the name of sample “string”.java file that is placed in the hbl=>jag=>tri=>sam directory.

From the “tessellation” working directory we enter:

```
sh makeTriJar.sh HelloTess
        or
./makeTriJar.pl1 "HelloTess"
```

Once again the script will run giving diagnostic output to the screen and two more directories will be added to the “tessellation” directory:

1) a runnable standalone jar directory “HelloTess.jar”. To execute this program type the command

java -jar HelloTess.jar

2) The second will be an unzipped version of the library in a directory called “SampleJARcontent” which is just an human readable listing of what is in the “HelloTess.jar” file.

This is the general format for linking any sample program to the triangle library so that you can then distribute a standalone jar file that can be run with a one line call of the form:

java -jar “SampleName.jar” -- In the real call don't include the "" marks.

EXTENDED EXAMPLES: STEP 3

The directory hbl=>jag=>tri=>sam contains a number of additional files that exercise various options in the triangle library. These can all be invoked using the same two command line options as outlined in the "HelloTess"

The user calling program SampleDatTri.jav can be run by first calling the script

```
sh makeTriJar SampleDagTri
        or
./makeTriJar "SampleDagTri"
```

followed by:

```
java -jar SampleDagTri.jar
```

By looking at the code in this calling program you can see that this sample goes through the standard build steps that are required for any tessellation.

It first generates a couple of arrays of type "long" which will hold the latitude and longitude (in micro-degrees) components of a user specified number of random coordinate points. The simple constructor then takes these arrays and a flag indicating that a Cartesian coordinate system is assumed, and then creates the initial data structures. The next step is to enter a loop that adds the remaining vertices's one at a time. Within the loop each new point is located in an existing triangle, distance of the new point to its closest pre-existing triangle point is available for the routine to act on; and, tessellation is extended to include the new vertex. On exiting the tessellation loop the routine calls a library routine to calculate triangle areas and Thiessen data. Then an output option is available in the form of a binary Java object (but it is commented out in the example). Finally a window is presented to show the triangularization. The window will zoom its contents using the up and down arrow keys and re-center the triangle view

based on a mouse click. The code is generally commented and should be easy to follow and serve as a general template for the "unconstrained" tessellation problem.

The next sample program SampleContourZ.java follows the same format as the previous one, but adds a bit more complexity. The latitude and longitude points are again randomly chosen from a square region, but they are then used to sample an analytical function to obtain a corresponding z value which represents an array of position dependent variables. (note these are “doubles”, not “longs” like the micro-degree position data). A different constructor is used that has an additional argument for the dependent z array. After the tessellation and triangle build of the Thiessen polygons is complete some data structures are defined to hold contour objects and an additional loop calls the triangle library's contour routine. And finally windows are displayed showing the data and analysis.
The sample is created and run by the commands:

```
sh makeTriJar.sh SampleContourZ
        or
./makeTriJar.pl1 "SampleContourZ"
```

followed by:

```
java -jar SampleContour.jar
```

The third non-trivial sample program is SampleLtoEContour.java. This again represents an incremental change to the previous example. In this case the random latitude and longitude data is not uniformly distributed over a square domain, but instead it is the output from a simple Gaussian Plume model for an instantaneous release into a steady wind field. (Like what might be expected from an atmospheric dispersion model similar Aloha (NOAA) or HySplit (NIST).) The Lagrangian particles in this case represent equal mass elements of the released material. After the tessellation is complete the triangle library method getvA() returns the area associated with the Voronoi diagram or Thiessen polygon which is used to represent the Eulerian density of the models Lagrangian distribution. These values are then run through a library “Laplacian smoother”, contoured and displayed. The sample is created and run by the commands:

```
sh makeTriJar.sh SampleLtoEContour
        or
./makeTriJar.pl1 "SampleLtoEContour"
```

followed by:

```
java -jar SampleLtoEContour.jar
```

The fourth sample program addesses the constrained tessellation problem. In this case the first operation of the program is read in data file which is a .bna map of the coastline (and for this example is hard coded to "test.bna" which is included in the "tessellation" directory. After this an optional echo check in included (but commented out). If you want to see what it gives as input just remove the /* ... */ bounds. The next step is to create a fake Gaussian dispersion model output for the upper section of the domain, making sure that all of the random points chosen are interior to the map boundary. These

interior Lagangian particles are add to the list of boundary points and DagData object is initialized, using a different Constructor that takes a bSeg argument which flags the problem as “constrained” and thus identifies the “sacred sides” used to modify the general tessellation routine. The sample then enters the standard loop of adding one vertex point at a time which completes the tessellation and also provides the opportunity to flag special characteristics that may be relevant during any continuing analysis. The next step in the algorithm sets some flags and calculated the Eulerian density associate with each point and assigns it to the z array. After this smoothing opportunities are presented and finally the contouring options are carried out is some detail. The final step is to plot the triangles and Eulerian contours. The sample is created and run by the commands:

```
sh makeTriJar.sh SampleConstrained
        or
./makeTriJar.pl1 "SampleConstrained"
```

followed by:

```
java -jar SampleLConstrained.jar
```

Give these routines a try, comments and feedback are welcome. Development will continue by the Genwest team.