# Compgeom: A Package of SAS IML Functions for Computational Geometry

The Compgeom package was released in SAS Viya 2024.10. It is a library of SAS IML modules that demonstrate how to solve classic problems in two-dimensional computational geometry by using the CONVEXHULL, DELAUNAY, and VORONOI functions in SAS IML. It also includes a set of visualization modules that you can use to visualize 2-D convex hulls, Delaunay triangulations, and Voronoi diagrams.

Most computations operate on a set of *sites*, which is a set of 2-D points. The points are assumed to be in *general position*. Planar points are in general (linear) position if every line in the plane contains at most two of the points.

The functions and subroutines that begin with the prefix CG\_ are the public methods for the package. These are documented, and their syntax should remain unchanged in future releases. The functions that begin with the prefix Compgeom\_ are helper functions. You are welcome to use these functions, but their names and syntax are subject to change.

The visualization routines are as follows:

* CG\_DrawConvexHull: Visualize the convex hull for a set of sites
* CG\_DrawDelaunay: Visualize the Delaunay triangulation for a set of sites
* CG\_DrawEmptyCircle: Visualize the largest empty circle whose center is inside the convex hull of a set of sites
* CG\_DrawTri: Visualize an arbitrary 2-D triangulation
* CG\_DrawVoronoi: Visualize the bounded portion of a Voronoi diagram for a set of sites

In addition, the computational routines are as follows:

* CG\_EmptyCircle: Obtain the center and radius of the largest empty circle whose center is inside the convex hull of a set of sites
* CG\_FindNearestNbr: For each site, find another site that is closest
* CG\_FindPtsInTri: Given a triangulation and a set of query points, find which triangle contains each query point.

# Visualization Routines

### Drawing the Convex Hull

**call CG\_DrawConvexHull(sites <, opt >)**

Creates a graph that visualizes the convex hull of the **sites** matrix, which is an n x 2 matrix. Optionally, display and label the sites that are on or inside the convex hull.

The input arguments are as follows:

* **sites** : An n x 2 matrix. Each row is the coordinate of a point in the plane.
* **opt** : An optional vector that specifies visualization options.
  + **opt[1]** : Specifies whether to display the vertices of the convex hull.
    - 0: Do not display vertices
    - 1: Display vertices (default)
    - 2: Display vertices and label them by using the rows of sites
  + **opt[2]** : Specifies whether to display the sites interior to the convex hull.
    - 0: Do not display interior sites
    - 1: Display interior sites (default)
    - 2: Display sites and label them by using the rows of sites
  + **opt[3]** : Specifies the transparency level for the fill attribute for the convex hull polygon.
    - alpha: Display the fill by using transparency level alpha (. The default value is 0.5, which results in a partially transparent polygon.
    - 0: Display an opaque convex polygon
    - 1: Do not fill the interior of the convex polygon
  + **opt[4]** : Specifies the PROC SGPLOT statements should be displayed in the log. This enables the programmer to modify the marker attributes and line attributes in the graph, or to overlay reference lines or additional data layers.
    - 0: Do not display the PROC SGPLOT statements (default)
    - 1: Display the PROC SGPLOT statements

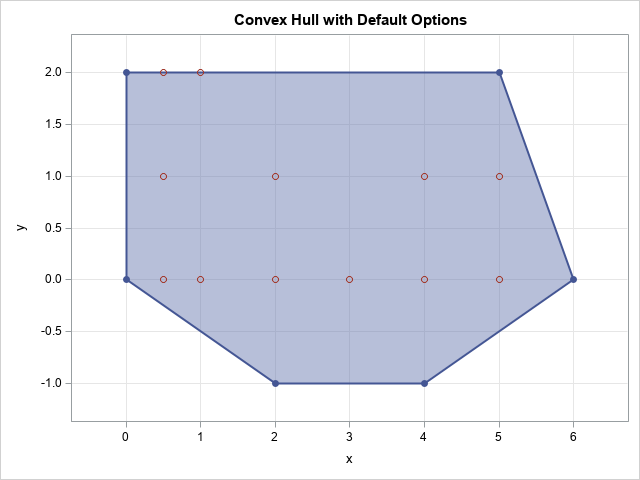
Examples:  
sites = {0 2, 0.5 2, 1 2, 0.5 1, 0 0, 0.5 0, 1 0,

2 -1, 2 0, 2 1, 3 0, 4 1, 4 0, 4 -1,

5 2, 5 1, 5 0, 6 0 };

title "Convex Hull with Default Options";

run CG\_DrawConvexHull(sites);



/\* test on larger set of bivariate normal data \*/

call randseed(123);

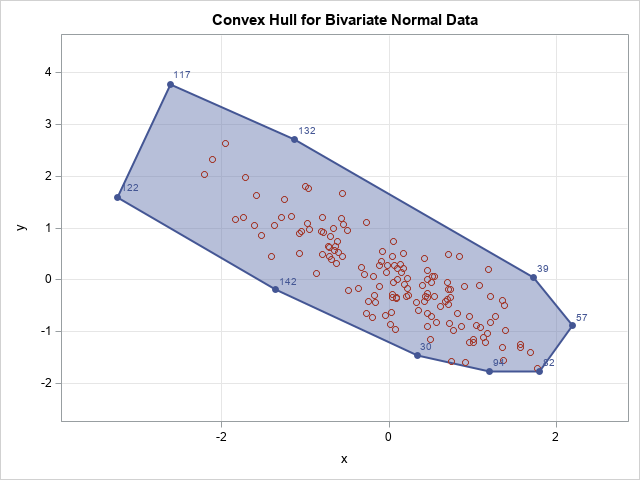
mu = {0 0};

Sigma = {1 -0.8, -0.8 1};

x = randnormal(150, mu, Sigma);

title "Convex Hull for Bivariate Normal Data";

run CG\_DrawConvexHull(x, {2, 1, 0.5});



### Drawing a Delaunay Triangulation

**call CG\_DrawDelaunay (sites <, opt >)**

This subroutine calls the DELAUNAY function to obtain the Delaunay triangulation of the **sites** matrix, which is an n x 2 matrix of unique sites. Each row is the (x,y) coordinates of a planar point. The subroutine then visualizes the Delaunay triangulation. Optionally, the routine displays and label the sites and the triangles.

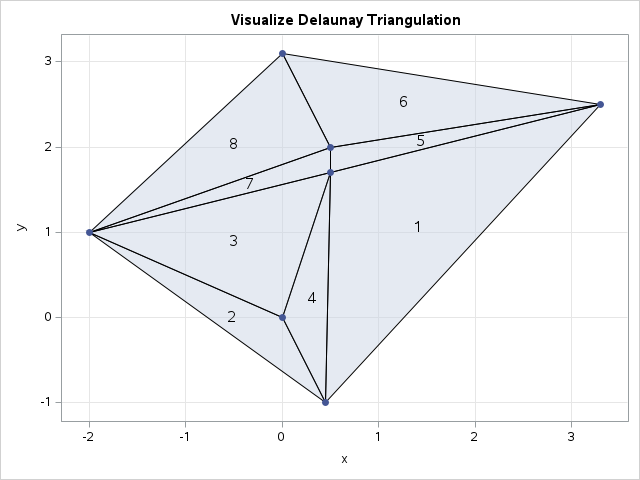
The input arguments are as follows:

* **sites** : An *n* x 2 matrix. Each row is the coordinate of a point in the plane.
* **opt** : An optional vector that specifies visualization options. The default option is {1, 1, 0.5, 0}.
  + **opt[1]** : Specifies whether to display the vertices of the triangulation.
    - 0: Do not display vertices
    - 1: Display vertices (default)
    - 2: Display vertices and label them by using the rows of **sites**
  + **opt[2]** : Specifies whether to display labels for the triangles.
    - 0: Do not display triangle labels
    - 1: Display triangle labels (default)
  + **opt[3]** : Specifies the transparency level for the fill attribute for the triangles.
    - alpha: Display the fill by using transparency level alpha (. The default value is 0.5, which results in partially transparent triangles.
    - 0: Display opaque triangles
    - 1: Do not fill the interior of the triangles
  + **opt[4]** : Specifies the PROC SGPLOT statements should be displayed in the log. This enables the programmer to modify the marker attributes and line attributes in the graph, or to overlay reference lines or additional data layers.
    - 0: Do not display the PROC SGPLOT statements (default)
    - 1: Display the PROC SGPLOT statements

Example:  
sites = {0 3.1, 0.5 2, 3.3 2.5, 0.5 1.7, 0 0, 0.45 -1, -2 1};

title "Visualize Delaunay Triangulation";

run CG\_DrawDelaunay(sites);



### Drawing a largest empty circle

**call CG\_DrawEmptyCircle (sites)**

Creates a graph that shows the sites and the location of the largest circle whose center is in the convex hull of the sites. The interior of the circle does not contain any sites. The argument is

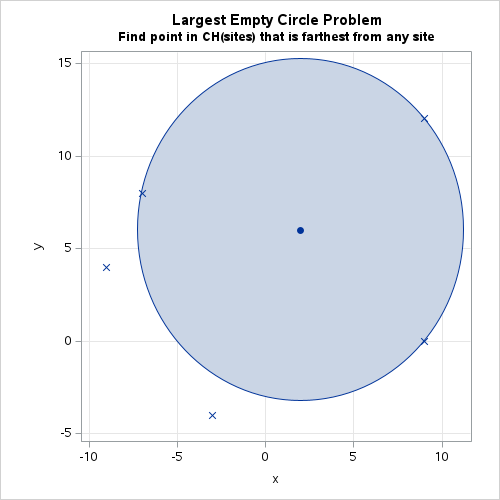
* **sites** : An *n* x 2 matrix. Each row is the coordinate of a point in the plane.

Example:

sites = {-7 8, -9 4, -3 -4, 9 0, 9 12};  
title "Largest Empty Circle Problem";

title2 "Find point in CH(sites) that is farthest from any site";

run CG\_DrawEmptyCircle(sites);



### Drawing a Triangulation

**call CG\_DrawTri (sites, tri, <, opt >)**

Creates a graph that visualizes the triangulation formed from the n x 2 matrix of unique sites ((x,y) coordinates, n > 2) and the triangles determined by the rows of a k x 3 matrix of indices. Optionally, display and label the sites and label the triangles.

* **sites** : An *n* x 2 matrix. Each row is the coordinate of a point in the plane.
* **tri** : a *k* x 3 matrix of integers. Each row of **tri** specifies a triangle. The *i*th triangle has vertices whose coordinates are equal to **sites[ tri[i,], ]**.
* **opt** : An optional vector that specifies visualization options. The default option is {1, 1, 0.5, 0}. For the valid values, see the CG\_DrawDelaunay subroutine.

Example:  
sites = {0 3.1, 0 2, 3.3 2.5, 1 1, 0 0, 0.45 -1, -2 1};

tri = {4 6 3,

6 5 7,

5 4 2,

4 5 6,

2 4 1,

1 4 3,

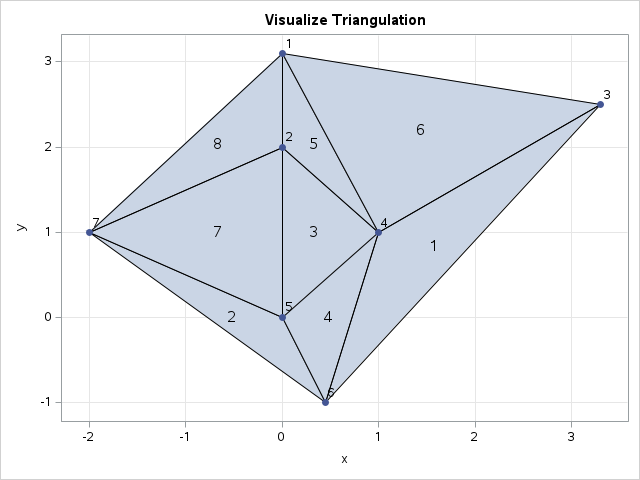
5 2 7,

2 1 7};

title "Visualize Triangulation";

opt = {2, 1, 0}; /\* label verts, triangles, opaque triangles \*/

run CG\_DrawTri(sites, tri, opt);



### Drawing the Voronoi Diagram

**call CG\_DrawVoronoi(sites <, BBox >)**

Creates a graph that visualizes the Voronoi diagram of the **sites** matrix, which is an n x 2 matrix. The full Voronoi diagram is unbounded, so it is more correct to state that the routine visualizes the portion of the Voronoi diagram that is inside a bounding box. By default, the bounding box is 10% larger than the extent of the sites and the Voronoi vertices.

The input arguments are as follows:

* **sites** : An n x 2 matrix. Each row is the coordinate of a point in the plane.
* **BBox** : An optional four-element vector that specifies the lower-left and upper-right corner of a rectangle that contains all sites and all Voronoi vertices.

Examples:

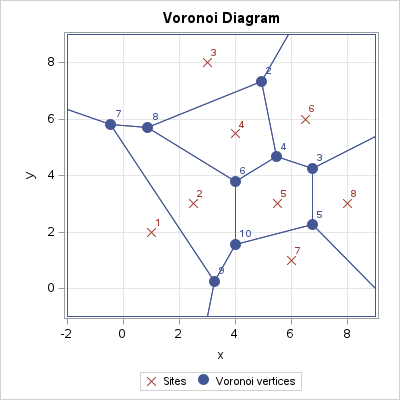
sites = {1 2, 2.5 3, 3 8, 4 5.5, 5.5 3,

6.5 6, 6 1, 8 3};

ods graphics / width=400px height=400px;

title "Voronoi Diagram";

run CG\_DrawVoronoi(sites);

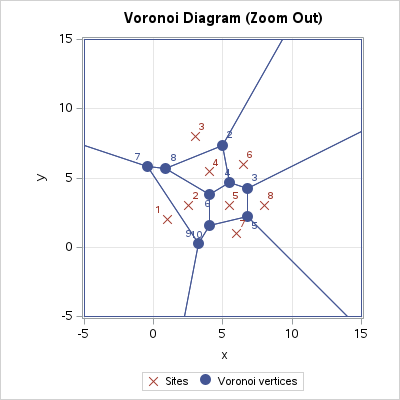


/\* zoom out \*/

BBox = {-5 -5 15 15};

title "Voronoi Diagram (Zoom Out)";

run CG\_DrawVoronoi(sites, BBox);



# Computational Routines

### Compute the center and radius of the largest empty circle

**CG\_EmptyCircle( sites )**

Find the center and radius of the *largest empty circle* for the sites. The center of the circle is inside the convex hull of the sites. An “empty circle” means that no sites are inside the circle’s interior. The function accepts a single input argument.

* **sites** : An *n* x *2* matrix. Each row is the Euclidean coordinate of a planar point.

The function returns a 2 x 2 matrix. The first row contains the center of the circle. The first element of the second row contains the radius of the circle. Mathematically, the center of the circle is a vertex of the Voronoi diagram for the sites.

Example:

sites = {-7 8, -9 4, -3 -4, 9 0, 9 12};

result = CG\_EmptyCircle(sites);

center = result[1,];

r = result[2,1];

print center[c={'x' 'y'} L=""] r;



### Locate the closest site to each site

**CG\_FindNearestNbr( sites )**

For each site, , find a site, , such that the distance from to is the minimum distance from to any other different site.

* **sites** : An *n* x *d* matrix. Each row is the Euclidean coordinate of a point in *d* dimensions.

The function returns an *n* x 2 matrix. The i\_th row contains information about the closest site to . The first column is the row number (*j*) of the closest site to . The second column is the distance between and .

Example:

The following example uses the same sites and triangulation as the **CG\_DrawTri** subroutine. See that example for a figure that shows the triangulation.

sites = {0 3.1, 0 2, 3.3 2.5, 1 1, 0.5 0.5, 0.45 -1, -2 1};

run CG\_DrawDelaunay(sites, {2,0,1}); /\* label only the sites \*/

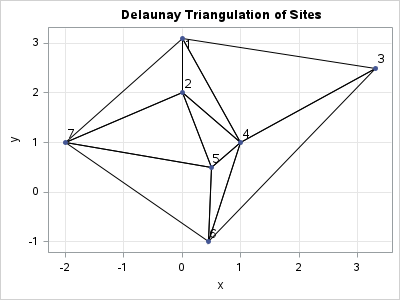
NN = CG\_FindNearestNbr( sites );

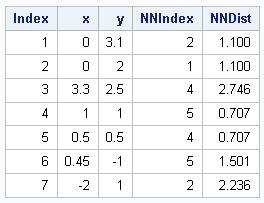
Index = T(1:nrow(sites));

NNIndex = NN[,1];

NNDist = NN[,2];

print Index sites[c={'x' 'y'} L=""] NNIndex NNDist[F=5.3];





The output shows the following:

* Sites 1 and 2 are nearest neighbors of each other. They are 1.1 units apart.
* Sites 4 and 5 are nearest neighbors of each other. They are units apart.
* The nearest neighbor to site 3 is site 4.
* The nearest neighbor to site 6 is site 5.
* The nearest neighbor to site 7 is site 5.

### Locate the Triangle That Contains Each Point

**CG\_FindPtsInTri (sites, tri, q)**

For each query point, q, find the triangle in a triangulation that contains q. If q is not in any triangle, return a missing value.

* **sites** : An *n* x 2 matrix. Each row is the coordinate of a point in the plane.
* **tri** : a *k* x 3 matrix of integers. Each row of **tri** specifies a triangle. The *i*th triangle has vertices whose coordinates are equal to **sites[ tri[i,], ]**.
* **q** : An *m* x 2 matrix of 2-D points.

For each row of **q**, return an integer in the range 1 – *n* that identifies the triangle that contains **q**. If **q** is on the edge of a triangle or at a vertex, the return value is one of the triangles that contains **q**.

Example:

The following example uses the same sites and triangulation as the **CG\_DrawTri** subroutine. See that example for a figure that shows the triangulation.

sites = {0 3.1, 0 2, 3.3 2.5, 1 1, 0 0, 0.45 -1, -2 1};

tri = {4 6 3, /\* triangle 1 \*/

6 5 7,

5 4 2,

4 5 6,

2 4 1,

1 4 3,

5 2 7,

2 1 7}; /\* triangle 8 \*/

q = {-1 2, /\* in the 8th triangle \*/

-1 1, /\* in the 7th triangle \*/

-1 0, /\* not in any triangle \*/

0 1, /\* on an edge shared by 3 and 7 \*/

1 2, /\* in the 6th triangle \*/

1 1}; /\* a vertex shared by 1, 3, 4, 5, and 6 \*/

ID = CG\_FindPtsInTri(sites, tri, q);

print (q || ID)[c={'x' 'y' "Triangle"} L='Results'];

