Veni, Vidi, Voronoi: Attacking Viruses using spherical Voronoi diagrams in Python

Tyler Reddy

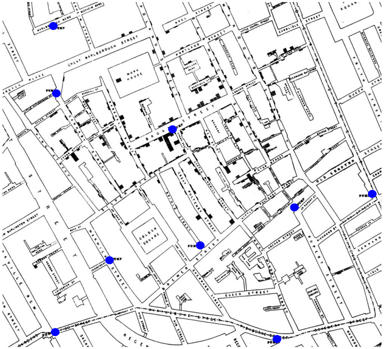
SBCB Unit, Department of Biochemistry, University of Oxford

PyData London

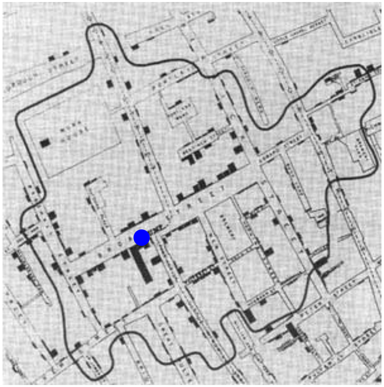
June 20 2015; 10:00 - 10:40 am

Room LG6

(Planar) Voronoi Diagrams



Original Soho Cholera Outbreak Map, Dr. John Snow, 1854

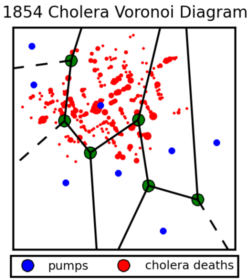


Walking time-based Voronoi Diagram

Fundamental concept: visual assessment of **generator** (pump) proximity

Planar Voronoi Diagrams in Python

```python
#example of using Python to produce a planar Voronoi diagram
import scipy.spatial
import matplotlib.pyplot as plt
import cPickle as pickle
%matplotlib inline

#load data modified from Robin Wilson's (University of Southampton) blog
array_pump_coords = pickle.load(open('array_pump_coords.p','rb'))
array_cholera_data = pickle.load(open('array_cholera_data.p','rb'))

#plot
figure_voronoi = plt.figure()
ax = figure_voronoi.add_subplot('111')
vor = scipy.spatial.Voronoi(array_pump_coords)
voronoi_figure = scipy.spatial.voronoi_plot_2d(vor, ax = ax)
ax.scatter(array_cholera_data[...,0], array_cholera_data[...,1], s = array_cholera_data[...,2] * 2,
            c = 'red', edgecolor='none') #scale scatter point area by number of fatalities
```



Some Voronoi algorithms in the plane

Authors	Year	Citations	Paradigm	Performance
Shamos and Hoey	1975	1101	divide-and-conquer	$O(n \log n)$
Green and Sibson	1978	861	incremental	$O(n^2)$
Guibas and Stolfi	1985	1506	quad-edge data structure	$O(n \log n)$
Fortune	1987	1402	sweepline	$O(n \log n)$

scipy uses Qhull library to compute the Voronoi Diagram via the Delaunay Triangulation

Computational Geometry Concepts 1: The Convex Hull

Convex Hull Algorithms

Paradigm	2D Complexity	3D Complexity
Incremental	$O(n^2)$	$O(n^2)$*
Gift-wrapping	$O(nh)$	$O(nf)$
Graham scan	$O(n \log n)$	N/A
divide-and-conquer (recursion)	$O(n \log n)$	$O(n \log n)$*

*4D and up: $\Omega(n^{d/2})$

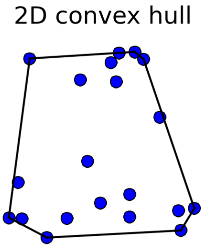
*often implemented because of conceptual simplicity

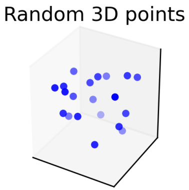
2D Hulls (in Python)

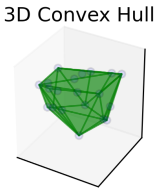
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.spatial import ConvexHull,
convex_hull_plot_2d
# 20 random points in 2-D:
points = np.random.rand(20, 2)
hull = ConvexHull(points)
convex_hull_plot_2d(hull)
```

3D Hulls (in Python)

```
# 20 random points in 3-D
points_3D = np.random.rand(20, 3)
hull_3D = ConvexHull(points_3D)
hull_facets = hull_3D.points[hull_3D.simplices]
#plotting code excluded
```







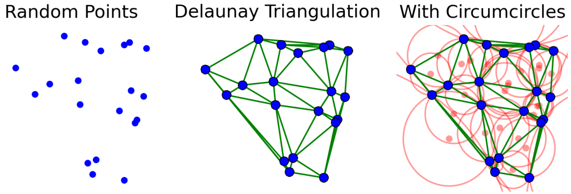
For additional reading: Devadoss and O'Rourke (2011) Discrete and Computational Geometry

Computational Geometry Concepts 2: Triangulations

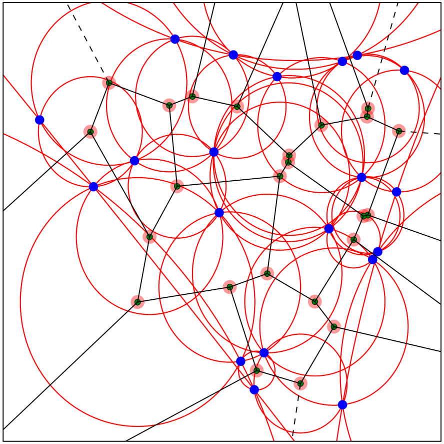
2D Definition: Subdivision of plane that maximizes number of non-intersecting edges connecting the point set.

- Using Euler's Formula: $t = 2k + h - 2$, where k and h are for interior and hull points, respectively, and t is the number of triangles for the planar point set.
- Number of distinct triangulations of a planar point set: 30^n maximum (Sharir *et al.*, 2009)
- **Delaunay Triangulations** (i.e., terrain reconstruction on Earth's surface with limited altitude samples)
- **Delaunay Triangulation** definition: a triangulation that has only legal edges and is the fattest of all possible triangulations of the planar point set. [Can build with edge flipping to remove illegal edges]
- Alternatively, a **Delaunay Triangulation** of a point set is one that excludes all points from the circumcircles of the constitutent triangles.
- Minimum weight triangulation reduces the amount of wire needed in a network.

```python
import numpy as np
from scipy.spatial import Delaunay

#20 random 2D points:
points = np.random.rand(20,2)
tri = Delaunay(points)
```



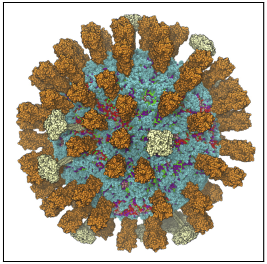
Delaunay to Voronoi

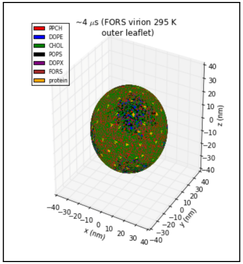


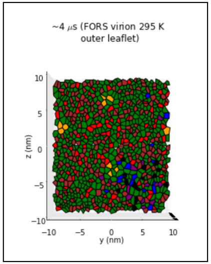
- Delaunay was a Ph.D. student of Voronoi at Kiev University
- There are deep, beautiful connections between Delaunay triangulations, Voronoi diagrams, and the convex hull

Applications of Spherical Voronoi Diagrams

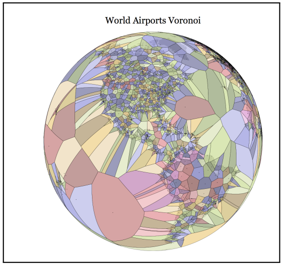
Computational Virology







Geographic Analysis



- Jason Davies – London
- Javascript
- Closed Source?
- Interactive visualization rather than data input / output

- Search and Rescue
- Airspace territory assignment / interception
- Global Epidemiology
- Telecommunicatinos

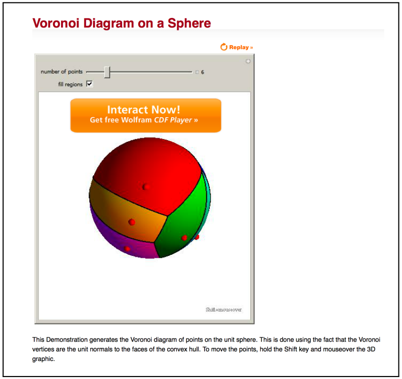
Python Spherical Voronoi: Heterogeneous Resources

Primary Literature – Spherical Voronoi Algorithms

Authors	Year	Citations	Paradigm	Performance
Augenbaum	1985	89	progressive insertion	$O(n^2)$
Renka	1997	146	incremental	$O(n \log n)$
Gold and Mostafavi	2000	33	point-line model	?
Na *et al.*	2002	50	stereographic projection	$O(n \log n)$
Chen *et al.*	2003	34	spherical mesh	?
Caroli *et al.*	2009	3	incremental	?
Dinis and Mamede	2010	3	sweep line	$O(n \log n)$

Attempted to implement for a few months – no success.

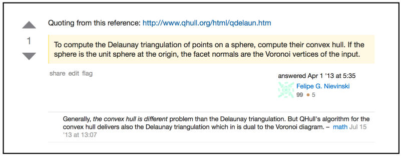
Closed Source Example



Voronoi Diagram on a Sphere

This Demonstration generates the Voronoi diagram of points on the unit sphere. This is done using the fact that the Voronoi vertices are the unit normals to the faces of the convex hull. To move the points, hold the Shift key and mouseover the 3D graphic.

stackoverflow



1

Quoting from this reference: http://www.qhull.org/html/qdelaun.htm

To compute the Delaunay triangulation of points on a sphere, compute their convex hull. If the sphere is the unit sphere at the origin, the facet normals are the Voronoi vertices of the input.

share edit flag

answered Apr 1 '13 at 5:35
Felipe G. Nievinski
99 ● 5

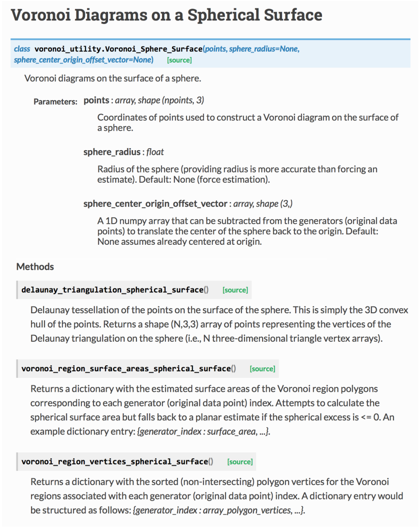
Generally, *the convex hull is different* problem than the Delaunay triangulation. But QHull's algorithm for the convex hull delivers also the Delaunay triangulation which in is dual to the Voronoi diagram. – math Jul 15 '13 at 13:07

The internet 'suggests' computing the convex hull, which is equivalent to the Delaunay Triangulation on the sphere, and then taking the normals to the faces of the convex hull to obtain the Voronoi vertices.

Python Spherical Voronoi: Implementation

GitHub: `https://github.com/tylerjereddy/py_sphere_Voronoi`

Docs: `http://py-sphere-voronoi.readthedocs.org/en/latest/voronoi_utility.html`



Voronoi Diagrams on a Spherical Surface

class **`voronoi_utility.Voronoi_Sphere_Surface`**(*points, sphere_radius=None, sphere_center_origin_offset_vector=None*) [source]

Voronoi diagrams on the surface of a sphere.

Parameters: **points** : *array, shape (npoints, 3)*

Coordinates of points used to construct a Voronoi diagram on the surface of a sphere.

sphere_radius : *float*

Radius of the sphere (providing radius is more accurate than forcing an estimate). Default: None (force estimation).

sphere_center_origin_offset_vector : *array, shape (3,)*

A 1D numpy array that can be subtracted from the generators (original data points) to translate the center of the sphere back to the origin. Default: None assumes already centered at origin.

Methods

`delaunay_triangulation_spherical_surface`() [source]

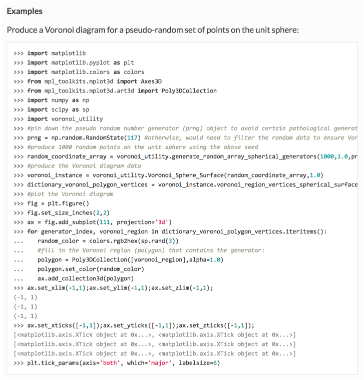
Delaunay tessellation of the points on the surface of the sphere. This is simply the 3D convex hull of the points. Returns a shape (N,3,3) array of points representing the vertices of the Delaunay triangulation on the sphere (i.e., N three-dimensional triangle vertex arrays).

`voronoi_region_surface_areas_spherical_surface`() [source]

Returns a dictionary with the estimated surface areas of the Voronoi region polygons corresponding to each generator (original data point) index. Attempts to calculate the spherical surface area but falls back to a planar estimate if the spherical excess is <= 0. An example dictionary entry: *{generator_index : surface_area, ...}*.

`voronoi_region_vertices_spherical_surface`() [source]

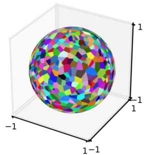
Returns a dictionary with the sorted (non-intersecting) polygon vertices for the Voronoi regions associated with each generator (original data point) index. A dictionary entry would be structured as follows: *{generator_index : array_polygon_vertices, ...}*.



Examples

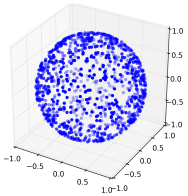
Produce a Voronoi diagram for a pseudo-random set of points on the unit sphere:

```
>>> import matplotlib
>>> import matplotlib.pyplot as plt
>>> import matplotlib.colors as colors
>>> from mpl_toolkits.mplot3d import Axes3D
>>> from mpl_toolkits.mplot3d.art3d import Poly3DCollection
>>> import numpy as np
>>> import scipy as sp
>>> import voronoi_utility
>>> #pin down the pseudo random number generator (prng) object to avoid certain pathological generat
>>> prng = np.random.RandomState(117) #otherwise, would need to filter the random data to ensure Vor
>>> #produce 1000 random points on the unit sphere using the above seed
>>> random_coordinate_array = voronoi_utility.generate_random_array_spherical_generators(1000,1.0,pr
>>> #produce the Voronoi diagram data
>>> voronoi_instance = voronoi_utility.Voronoi_Sphere_Surface(random_coordinate_array,1.0)
>>> dictionary_voronoi_polygon_vertices = voronoi_instance.voronoi_region_vertices_spherical_surface
>>> #plot the Voronoi diagram
>>> fig = plt.figure()
>>> fig.set_size_inches(2,2)
>>> ax = fig.add_subplot(111, projection='3d')
>>> for generator_index, voronoi_region in dictionary_voronoi_polygon_vertices.iteritems():
...    random_color = colors.rgb2hex(sp.rand(3))
...    #fill in the Voronoi region (polygon) that contains the generator:
...    polygon = Poly3DCollection([voronoi_region],alpha=1.0)
...    polygon.set_color(random_color)
...    ax.add_collection3d(polygon)
>>> ax.set_xlim(-1,1);ax.set_ylim(-1,1);ax.set_zlim(-1,1);
(-1, 1)
(-1, 1)
(-1, 1)
>>> ax.set_xticks([-1,1]);ax.set_yticks([-1,1]);ax.set_zticks([-1,1]);
[<matplotlib.axis.XTick object at 0x...>, <matplotlib.axis.XTick object at 0x...>]
[<matplotlib.axis.XTick object at 0x...>, <matplotlib.axis.XTick object at 0x...>]
[<matplotlib.axis.XTick object at 0x...>, <matplotlib.axis.XTick object at 0x...>]
>>> plt.tick_params(axis='both', which='major', labelsize=6)
```



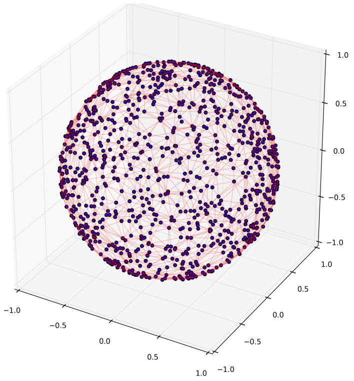
Spherical Voronoi Algorithm Stage 1: Random points on unit sphere

```
#step 1: generate pseudo-random set of points on the unit sphere
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
import numpy as np
import scipy as sp
import voronoi_utility

#pin down the pseudo random number generator (prng) object to avoid certain pathological generator sets
#otherwise, would need to filter the random data to ensure Voronoi diagram is possible
prng = np.random.RandomState(117)
#produce 1000 random points on the unit sphere using the above seed
random_coordinate_array = voronoi_utility.generate_random_array_spherical_generators(1000,1.0,prng)

#examine the random coordinates on the unit sphere by visual inspection:
fig_random_coords_unit_sphere = plt.figure()
ax = fig_random_coords_unit_sphere.add_subplot('111',projection='3d')
ax.scatter(random_coordinate_array[...,0],random_coordinate_array[...,1],random_coordinate_array[...,2],
        edgecolors='none')
ax.set_xlim(-1,1); ax.set_ylim(-1,1); ax.set_zlim(-1,1)
fig_random_coords_unit_sphere.set_size_inches(5,5)
fig_random_coords_unit_sphere.savefig('spherical_algorithm_stage_1.png', dpi = 300)
```



Spherical Voronoi Algorithm Stage 2: Delaunay triangulation

```
#step 2: produce Delaunay triangulation coordinates
hull_instance = sp.spatial.ConvexHull(random_coordinate_array)
list_points_vertices_Delaunay_triangulation = []
for simplex in hull_instance.simplices: #for each simplex (triangle) of the convex hull
    convex_hull_triangular_facet_vertex_coordinates = hull_instance.points[simplex]
    list_points_vertices_Delaunay_triangulation.append(convex_hull_triangular_facet_vertex_coordinates)
array_points_vertices_Delaunay_triangulation = np.array(list_points_vertices_Delaunay_triangulation)
#the above should be a shape (N,3,3) numpy array of the Delaunay triangle vertex coordinates
#on the surface of the sphere.

#plotting code omitted for clarity
```



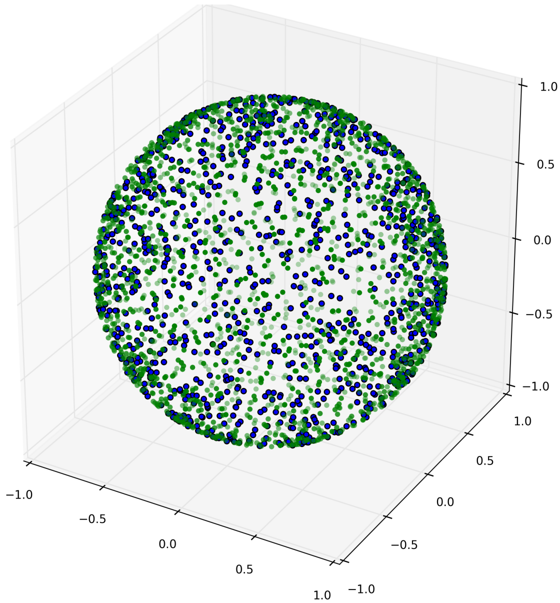
generators

Delaunay

Spherical Voronoi Algorithm Stage 3: Voronoi Vertices

```python
#step 3: produce an array of Voronoi vertices
facet_coordinate_array_Delaunay_triangulation = array_points_vertices_Delaunay_triangulation
facet_normals_array = np.cross(facet_coordinate_array_Delaunay_triangulation[...,1,...] -
        facet_coordinate_array_Delaunay_triangulation[...,0,...],
        facet_coordinate_array_Delaunay_triangulation[...,2,...]
        - facet_coordinate_array_Delaunay_triangulation[...,0,...])
facet_normal_magnitudes = np.linalg.norm(facet_normals_array,axis=1)
facet_normal_unit_vector_array = facet_normals_array / np.column_stack((facet_normal_magnitudes,
    facet_normal_magnitudes,facet_normal_magnitudes))
#try to ensure that facet normal faces the correct direction (i.e., out of sphere)
triangle_centroid_array = np.average(facet_coordinate_array_Delaunay_triangulation,axis=1)
#normalize the triangle_centroid to unit sphere distance for the purposes of the following directionality
#check
array_triangle_centroid_spherical_coords =
voronoi_utility.convert_cartesian_array_to_spherical_array(triangle_centroid_array)
array_triangle_centroid_spherical_coords[...,0] = 1.0
triangle_centroid_array =
voronoi_utility.convert_spherical_array_to_cartesian_array(array_triangle_centroid_spherical_coords)
#the Euclidean distance between the triangle centroid and the facet normal should be smaller than the
#sphere centroid to facet normal distance, otherwise, need to invert the vector
triangle_to_normal_distance_array = np.linalg.norm(triangle_centroid_array
        - facet_normal_unit_vector_array,axis=1)
sphere_radius = 1. ; sphere_centroid = np.average(random_coordinate_array, axis = 0)
sphere_centroid_to_normal_distance_array = np.linalg.norm(sphere_centroid
        - facet_normal_unit_vector_array,axis=1)
delta_value_array = sphere_centroid_to_normal_distance_array - triangle_to_normal_distance_array
#need to rotate the vector so that it faces out of the circle
facet_normal_unit_vector_array[delta_value_array < -0.1] *= -1.0
facet_normal_unit_vector_array *= sphere_radius #adjust for radius of sphere
array_Voronoi_vertices = facet_normal_unit_vector_array
```

Spherical Voronoi Algorithm Stage 3: Voronoi Vertices



generators

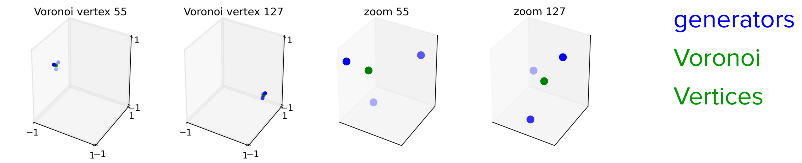
Voronoi Vertices

Challenges:

- Associate vertices with generators
- Order polygon vertices in close proximity
- Problems with missing vertices and self-intersection
- Am I missing something here?

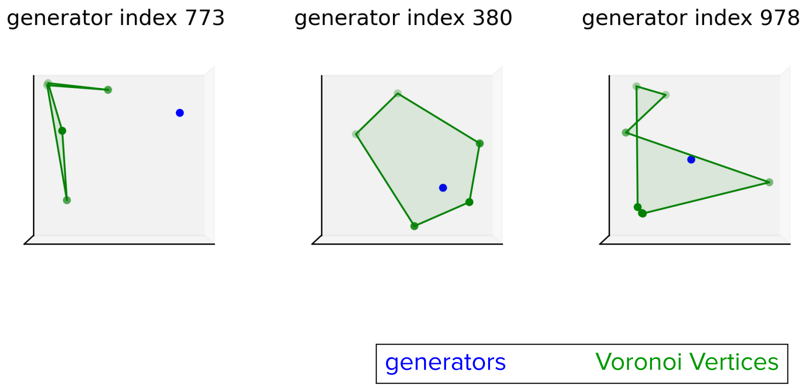
Spherical Voronoi Algorithm Stage 4a: Voronoi vertex generator associations

```
#step 4a: For all Voronoi vertices, determine the full set of generators for which they form part
#of the Voronoi polygon

#I already have the Voronoi vertices and the generators, so work based off a distance matrix
#between them
distance_matrix_Voronoi_vertices_to_generators =
                sp.spatial.distance.cdist(array_Voronoi_vertices,random_coordinate_array)

#now, each row of the above distance array corresponds to a single Voronoi vertex, with each
#column of that row representing the distance to the respective generator point
#if we iterate through each of the rows and determine the indices of the minimum distances, we
#obtain the indices of the generators for which that voronoi vertex is a polygon vertex
generator_Voronoi_region_dictionary = {} #store the indices of the generators for which a given
#Voronoi vertex is also a polygon vertex
for Voronoi_point_index, Voronoi_point_distance_array in enumerate(distance_matrix_
Voronoi_vertices_to_generators):
    Voronoi_point_distance_array = np.around(Voronoi_point_distance_array,decimals=3)
    indices_of_generators_for_which_this_Voronoi_point_is_a_polygon_vertex =
np.where(Voronoi_point_distance_array == Voronoi_point_distance_array.min())[0]
    assert indices_of_generators_for_which_this_Voronoi_point_is_a_polygon_vertex.size >= 3,
"By definition, a Voronoi vertex must be equidistant to at least 3 generators."
    generator_Voronoi_region_dictionary[Voronoi_point_index] =
            indices_of_generators_for_which_this_Voronoi_point_is_a_polygon_vertex
#so dictionary looks like 0: array(12,17,27), ...
```



Spherical Voronoi Algorithm Stage 4b: Collect unsorted polygon vertices

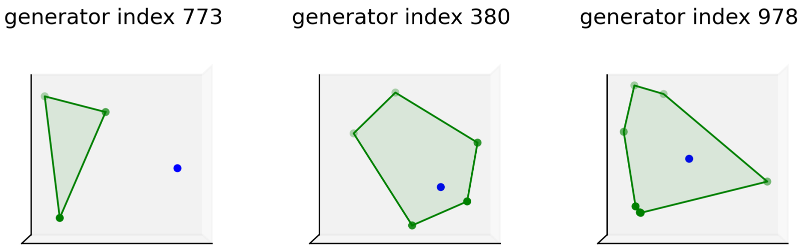
```python
#step 4b: collect the Voronoi point indices forming the polygon for each generator index
from collections import defaultdict
dictionary_Voronoi_point_indices_for_each_generator = defaultdict(list)
for Voronoi_point_index, indices_of_generators_for_which_this_Voronoi_point_is_a_polygon_vertex in
generator_Voronoi_region_dictionary.iteritems():
    for generator_index in indices_of_generators_for_which_this_Voronoi_point_is_a_polygon_vertex:
        dictionary_Voronoi_point_indices_for_each_generator[generator_index].append(Voronoi_point_index)
#dictionary should have format: {generator_index: [list_of_Voronoi_indices_forming_polygon_vertices]}
```

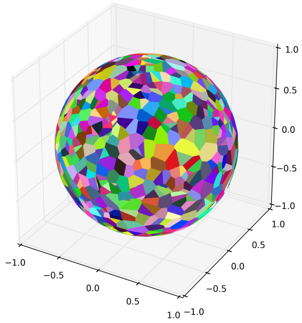


Spherical Voronoi Algorithm Stage 4c: Sort Voronoi Region Vertices

```python
#step 4c: sort voronoi region (polygon) vertices
dictionary_sorted_Voronoi_point_coordinates_for_each_generator = {}

def sort_voronoi_coords(voronoi_vertex_array, projection_start_index, projection_end_index, test_index):
    #trying to project to 2D for edge ordering, and then restore to 3D after:
    polygon_hull_object = sp.spatial.ConvexHull(voronoi_vertex_array[...,projection_start_index:
        projection_end_index])
    point_indices_ordered_vertex_array = polygon_hull_object.vertices
    current_array_Voronoi_vertices = voronoi_vertex_array[point_indices_ordered_vertex_array]
    voronoi_utility.test_polygon_for_self_intersection(current_array_Voronoi_vertices[...,test_index:])
    return current_array_Voronoi_vertices

for generator_index, list_unsorted_Voronoi_region_vertices in dictionary_Voronoi_point_indices_for_each_
generator.iteritems():
    current_array_Voronoi_vertices = array_Voronoi_vertices[list_unsorted_Voronoi_region_vertices]
    if current_array_Voronoi_vertices.shape[0] > 3:
        try:
            current_array_Voronoi_vertices = sort_voronoi_coords(current_array_Voronoi_vertices, 0,
                    2, 1,)
        except voronoi_utility.IntersectionError: #try a different sorting / projection
            current_array_Voronoi_vertices = sort_voronoi_coords(current_array_Voronoi_vertices, 1,
                    current_array_Voronoi_vertices.shape[0] + 2, 1)
    assert current_array_Voronoi_vertices.shape[0] >= 3, "Polygon has >= 3 vertices."
    dictionary_sorted_Voronoi_point_coordinates_for_each_generator[generator_index] = current_array_
Voronoi_vertices
```

Spherical Voronoi Algorithm Stage 4c: Sort Voronoi Region Vertices

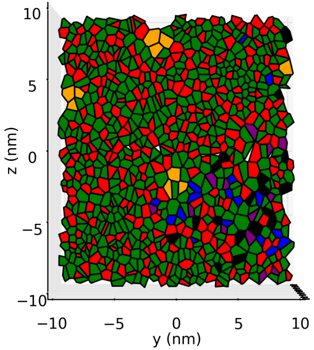


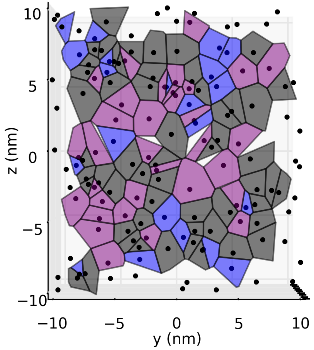


- generally $> 95\%$ surface area reconstitution
- generators outside of assigned Voronoi regions
- residual cases of polygon self-intersection
- floating point instability
- surface area – planar only

Example Results and Problems

Variable sensitivity to blank space problem





Spherical Polygon Area Calculation Unstable

$$S = [\theta - (n-2)\pi]R^2$$

Conclusions

- ultimately have the spherical Voronoi vertices – challenge seems to be rebuilding the Voronoi regions
- perhaps someday spherical Voronoi will be available in scipy.spatial

Acknowledgements

SBCB

- Prof. Mark S.P. Sansom
- Dr. Heidi Koldsø
- Dr. David Shorthouse
- Dr. Matthieu Chavent
- Dr. Philip W. Fowler
- Elizabeth Jefferys
- Dr. Phillip J. Stansfeld
- Dr. Antreas Kalli

Former SBCB:

- Dr. Danny Parton
- Dr. Joseph Goose
- Dr. Sarah Rouse
- Dr. Greg Ross
- Dr. Jussi Aittoniemi

IBPC (Paris):

- Dr. Marc Baaden
- Baaden Lab Members

Mathematical Institute (Oxford):

- Dr. Rebecca Gower
- Dr. Mason Porter

Computer Science (Oxford):

- Dr. Joe Pitt-Francis

The Leverhulme Trust

MD ANALYSIS

CINES Centre Informatique National de l'Enseignement Supérieur

University of Oxford

CIHR IRSC

PRACE

python™

ScalaLife

DEPARTMENT OF BIOCHEMISTRY
UNIVERSITY OF OXFORD

Somerville College
University of Oxford

HPC-Europa2
Pan-European Research Infrastructure on High Performance Computing

wellcome trust