UCSB, Physics 129AL, Computational Physics: Section Worksheet, Week 2

Zihang Wang (UCSB), zihangwang@ucsb.edu

January 13, 2025

Section Participation

Section attendance is required, but you do not need to complete all the work during the section. At each section, the TA will answer any questions that you might have, and you are encouraged to work with others and look for online resources during the section and outside of sections. **Each task should be synced to Github separately on your own Github account in order to have section grades.**

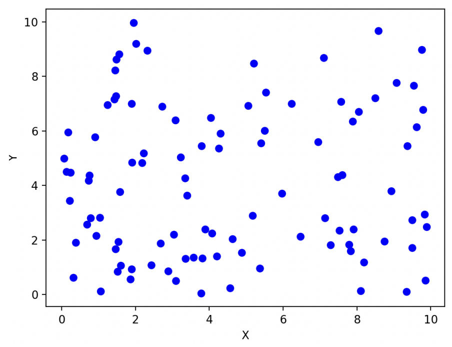
Unless otherwise stated, the work will be due one week from the time of assignment. The TA will give you 1 point for each task completed. You can see your grades on Canvas.

Task 1: Convex hull in 2D

In the class, we discussed how to find the **convex hull** of a given point cloud in 2 dimensions. We want to compare different convex hull algorithms and their preferences. **All of the following work should be tracked via git and pushed on github with the task name**.

1. Build Algorithms

a). In the course project site, download a point cloud data (.dat), and visualize it (it should be done in the Docker container, and remember to git it!).



b). Write the following four functions in Python that find the convex hull of a given point cloud:

- 2D Graham Scan
- Jarvis march
- Quickhull
- Monotone chain

The function takes an input of a 2D numpy array, and output the coordinates of the convex hull. It will be something like this:

“ **def** function(points,args):
. . .
return hull

c). use four functions that you created earlier, output the convex hull, then plot them in different color to visualize them (also plot the point cloud data in the same figure). You should see that they enclosed the boundary.

2 Time Complexity of a point cloud

Time Complexity is a measure of the computational efficiency of an algorithm, specifically the amount of time it takes to run as a function of the size of its input. It provides a way to analyze how an algorithm scales. In the following, we will compare the time complexity of the above functions.

a). Create a function that generate a n-point 2D uniform point cloud. The x,y coordinates are **bounded between 0 and 1**.

b). for n=[10,50,100,200,400,800,1000], generate the point cloud and input in all four functions that you have previously. For each function, calculate the time required to complete the task (look online!) In the end, you should have a plot showing 4 sets of data (x-axis: n, y-axis: runtime). Make a conclusion on their difference and similarities. Put your conclusion in a text file in the same directory. You can set the random seed so that you are comparing the same point cloud.

c). Do the same analysis in **a) and b)**, but now the x,y coordinates are **bounded between -5 and 5**. In other words, two sets of point clouds have different variance. Does the runtime depend on the variance of the point cloud? visualize the difference and put your conclusion in a text file in the same directory.

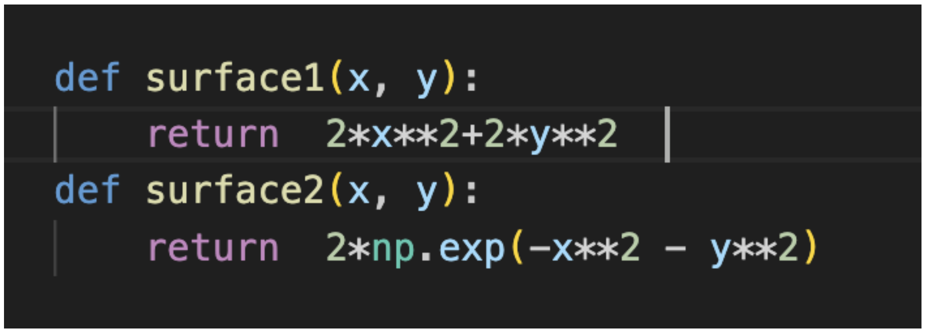
c). Do the same analysis in **a) and b)**, but now you need to sample the point cloud from a Gaussian distribution with variance 1. Does the runtime depend on the variance of the point cloud? visualize the difference and put your conclusion in a text file in the same directory.

d). let's take n=50. For each algorithm, generate 100 sets of point clouds and find the distribution of the run time and visualize them in 4 histograms. what are the worse and best run time for each one, and what distribution it approximately follows? Put your conclusion in a text file in the same directory. Make sure to set the random seed so that the conclusion is reproducible.

Task 2: Delaunay triangulation in 3D

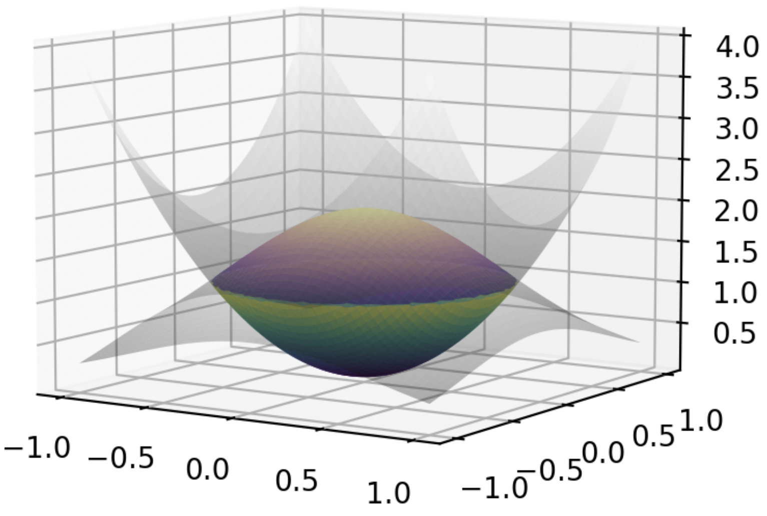
Package or function you may want to look into:
""
from collections import Counter
from scipy.spatial import Delaunay
np.isclose
"

a) Let's design a function that generates surface point cloud of the **closed surface formed by the following two functions**,



```
def surface1(x, y):
    return  2*x**2+2*y**2
def surface2(x, y):
    return  2*np.exp(-x**2 - y**2)
```

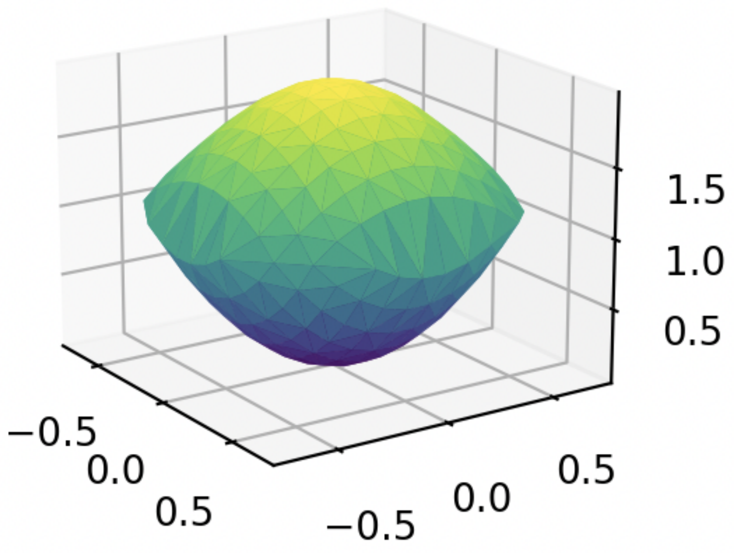
You will see something like this.



b) Using Delaunay triangulation to generate the surface mesh, and visualize it **This is not easy!** Here are things you may want to consider:

- To access the surface in Delaunay triangulation, you can use **"tri = Delaunay(points[:,:2])"**.
- First do the triangulation on the top, then do the triangulation on the bottom. You will see smooth meshes individually. Visualize it to see if it makes sense.
- Then, combine two meshes, be really careful on the **vertex id** output by Delaunay! They are identical! You should look into matching boundary points, and **reassign vertex id according to the boundary matching condition! Hint: if at boundary, how many times an edge will be shared by different triangles?**
- Make sure you have different vertex id for two surfaces, and you can then concatenate Delaunay triangulation for both surfaces in to one. This will give you the full triangulation.

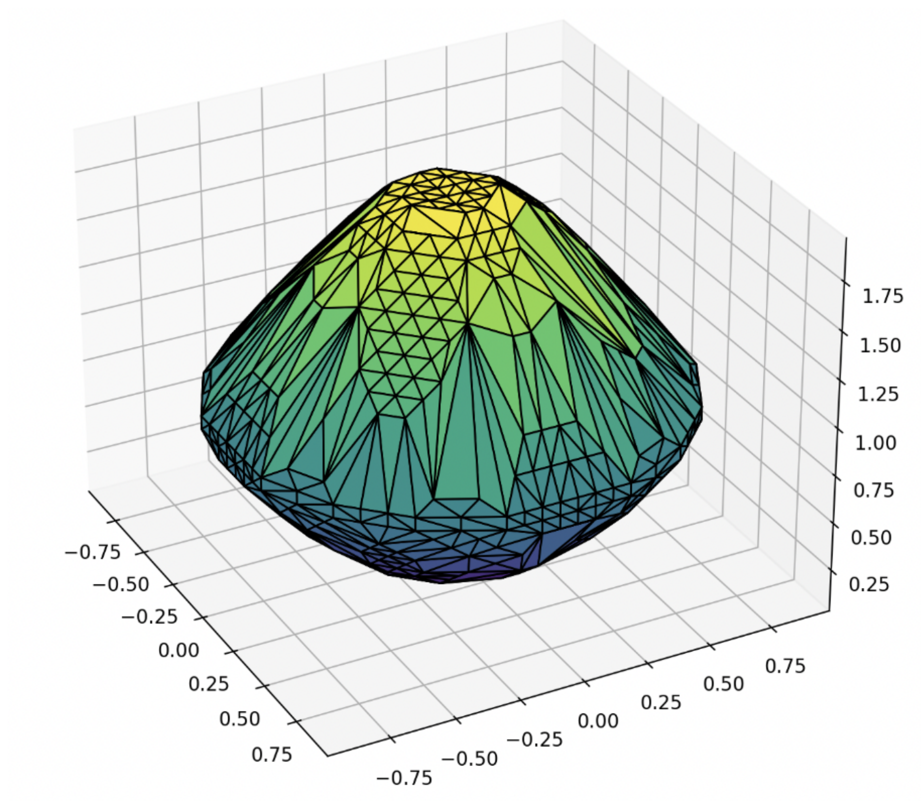
You should see something below when you visualize the surface triangulation using the function plot_trisurf in matplotlib. If you did not see something like this, it means that you are doing something not right.



c) Using Delaunay triangulation to generate the **volume mesh**, and visualize it (you might reduce the tetrahedron density for better visualization). **Hint: look into the following: from scipy.interpolate import griddata**

d) Use the volume mesh to generate the surface mesh and visualize it. **Hint: if at boundary, how many times a triangle will be shared by different tetrahedron?** What is the difference between d) and b)?

You should see something below when you visualize the surface triangulation using the function plot_trisurf in matplotlib. If you did not see something like this, it means that you are doing something not right.



−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
0.25
0.50
0.75
1.00
1.25
1.50
1.75