CS 274 Final Project: Delaunay Triangulation Divide & Conquer

Daniel Li

Professor Shewchuk

University of California, Berkeley

April 24th, 2017

Introduction:

This following report will be structured with the required materials (per website), instructions, graphs, required libraries, and credits. Everything was run on a 2.7 Ghz 2015 dual core Macbook pro with 8 gigabytes of RAM.

Deliverables:

The necessary files to execute the code, the generated point set, and 3 .ele files for the 10,000, 100,000, and 1,000,000 million point sets. Graphs will be included in this write up of a few of these examples and other *.node files from the website.

Prerequisites & Credits:

Everything is implemented with Python 2.7. Using Python 3 will give you problems (such as print statement parenthesis and etc.)
I used the following libraries in my implementation of the divide and conquer algorithm for vertical and alternating cuts: numpy, matplotlib, geompreds, time, and Triangle.
Numpy and matplotlib are fairly standard libraries but to install them on a Mac you can typically do the following command:

user$ sudo pip install numpy
user$ sudo pip install matplotlib

For geompreds I used the following package:
https://pypi.python.org/pypi/geompreds/1.0.1
You can follow the instructions on the page to install this and download it from the official python website. This is a wrapper class that someone implemented for Professor Shewchuk's robust predicates. This also assumes you have Cython installed. The two functions that are predominantly used are Orient2D and incircle.
For Triangle (Python) I found someone's wrapper class around the C Triangle utility functions that were particularly helpful:
http://dzhelil.info/triangle/installing.html
The instructions follow from how to install from their Github repo. The functions I used here were for reading in .ele and .node files for plotting and debugging. (Though I did modify the plot functions in order to change the matplotlib vertex size parameters for increased visibility).

Installing this library is not entirely necessary as I will include graphs. Just make sure in the next section not to mark the visualization flag.

In order to visualize the D&C process, I used the http://www.geom.uiuc.edu/~samuelp/del_project.html for the visualize diagrams to better understand merge (up until constrained DT's).

Other things worth noting, since this is done in Python without garbage collection (and all these objects are basically in lists and stuff like that for pointers), memory issues are kind of a hassle. More on that in run time.

Running the code:

To run the code you should have 8 Python files located in the same file directory. DT1.py is the vertical cuts algorithm and DT2.py is the alternating cuts algorithm. Edge.py, vertex.py, face.py, visualize.py are all supporting files. Generate_points.py is a standalone file to create a point set in which vertical cuts has a non trivial faster run time than the alternating cuts algorithm. Lastly, main.py links them all together and makes things nice. In main.py we have the following format:

user$ python main.py <filename> <algorithm> <plot>

Filename should be a string for the input *.node file in the standard Triangle format, algorithm should be a character "v" for vertical or "a" for alternating cuts, and lastly plot should be "t" to generate a plot and "f" to not generate a plot. When a plot is generated, a scatter of the points is first shown. Close it and then the DT will be shown.

A sample instance of running the code may look like this

user$ python main.py 10000.node a t

This will run the code on alternating cuts with the 10000.node input file and generate a plot. This will also write out a .ele file as 10000DT2.ele. The plot reads in the .ele file.

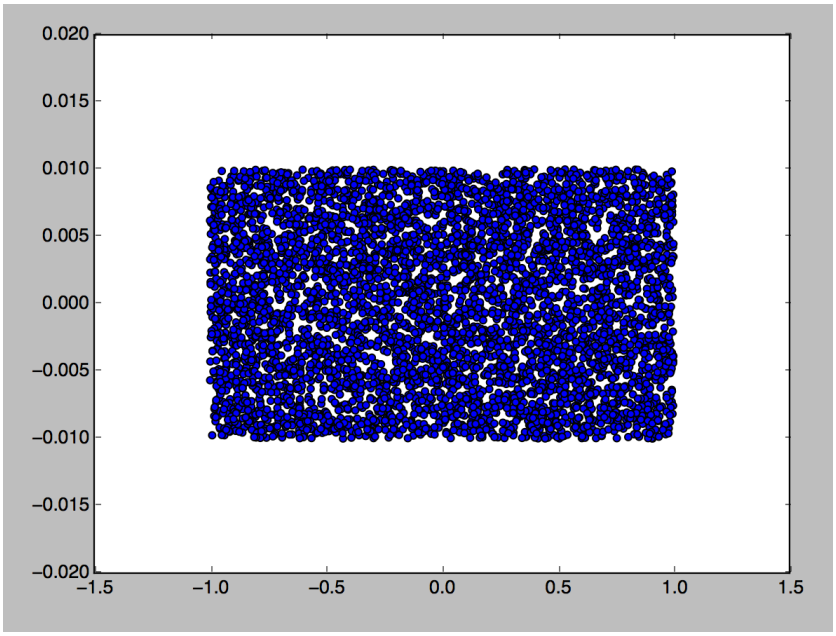
In order to generate a point set run the following:

user$ python generate_points.py

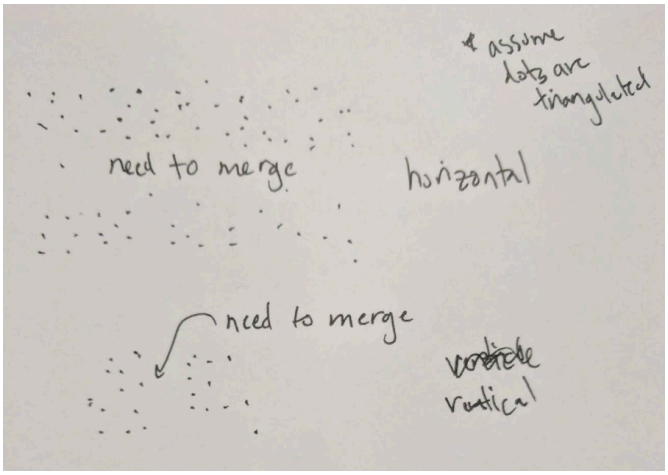
This will create a g.node file that you can run as the input using the previous instructions. The standard parameters for creating this point set are 5,000 points in a long rectangle manner (long in the x axis). The bounding box is given by the following parameters:
x_bounds=[-1,1], y_bounds=[-.01, .01], num_points=5000

Generated Point set:

The generated point set looks like the following and was generated uniform at random around the specified parameters for the bounding box. One can change the parameters to make the rectangle skinnier. The skinnier the rectangle (along the x), the better vertical cuts will perform w.r.t. alternating cuts.



The reason why vertical cuts works better on this is because during the horizontal merge process, we have to merge fewer points on the x-axis than on the y-axis. An example is shown below (assume that the distinct separated point clouds are already triangulated and we are at the merge stage).



The run time for vertical cuts was 2.28 seconds and the run time for alternating cuts was 5.66 seconds.

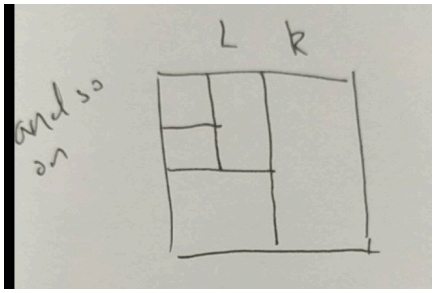
Run times:

The **first** entry is the vertical cuts algorithm; the **second** entry is the alternating cuts algorithm. This excludes the IO time and recovery time which can take awhile. The way I implemented recovery was to keep track of faces and walk along the faces to recover the triangles.

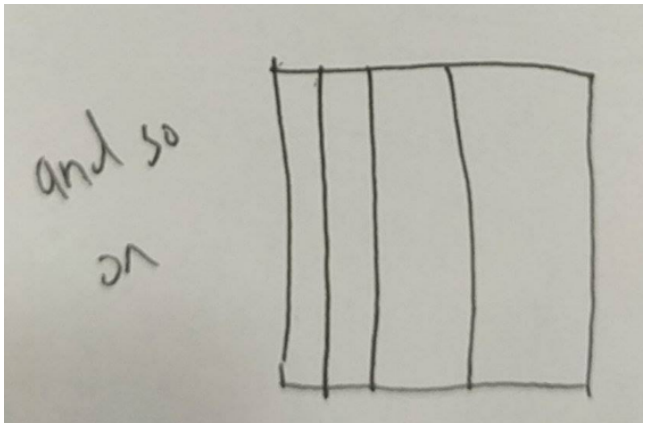
	10,000	100,000	1,000,000
Trial 1	14.66, 5.877	209.455, 65.13	Failed, 799.67
Trial 2	14.48, 5.729	207.97, 69.199	3511.8, 801.22
Trial 3	15.123, 5.618	212.66, 74.95	Did not run, 792.53

For trial's 1, 3 for the 1 million point set on the vertical cuts algorithm, the failed one was because of memory issues – I was passing in a new sliced array each time and I think that blew it up. For the next iteration, I passed in indices (updated from the D&C) and sliced them in the algorithm. However, I still ran into memory problems and I believe that this affected the run time. I checked my Mac's activity monitor and CPU usage and was hitting ~7 gigs of RAM and it was suppressing lots of memory. I think this led to page faults which gives an inaccurate run time.

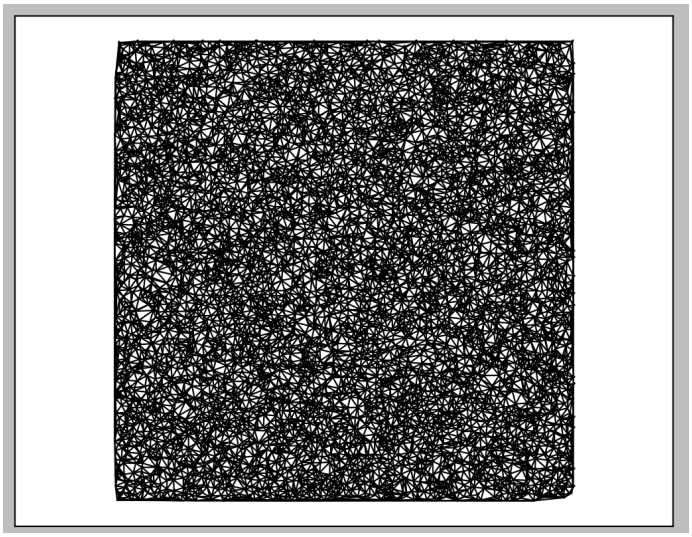
Notice that these point sets are generated at uniform random (I think) around a bounding square. Typically, the alternating cuts algorithm performs better assuming the points are relatively distributed in a position that doesn't extend in particular towards an axis. This is because it tries to evenly distribute the points after every other cut. For example, if I cut a square in half, then I get a rectangle. But if I get the rectangle in half I get another square. Building bottom up from when I merge, two triangulated point sets will have merging sides that look something like this:



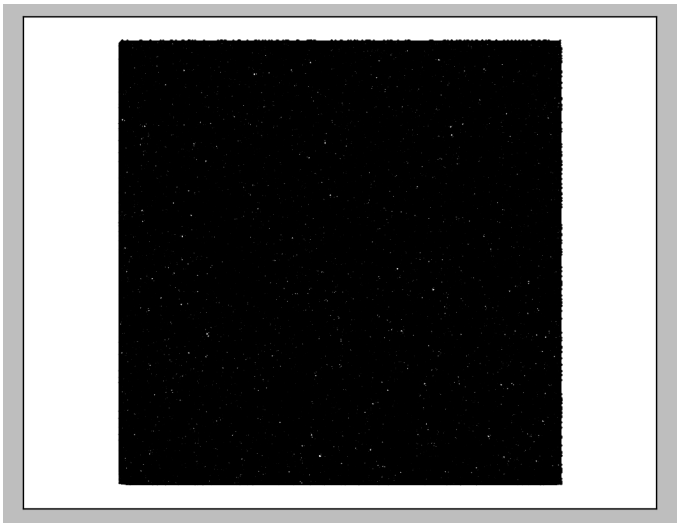
Thus each time I merge two parts where the longitudinal and latitudinal points that are facing each other are roughly around the same size, the merge costs along any direction pays roughly the same. However, assume I only cut vertically each time, the sets become skinny and each time there's a merge, I have to pay the roughly the same cost as the original length of the vertical side of the square to merge.



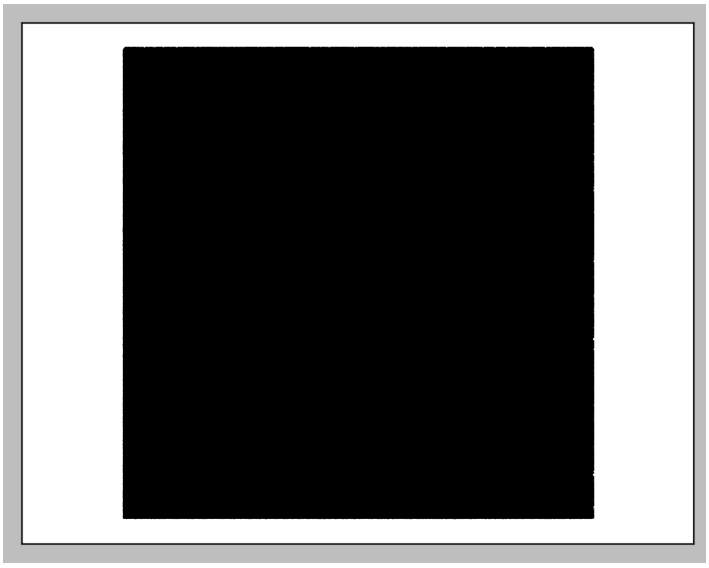
10,000 points triangulation:



100,000 points triangulation:



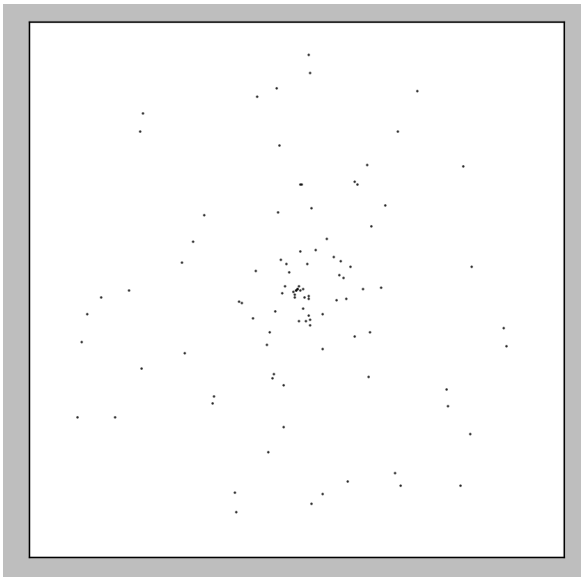
1,000,000 points (basically a black box)

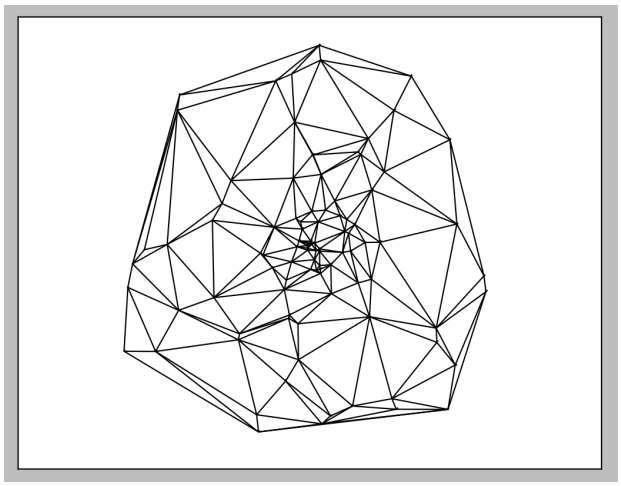


Miscellaneous:

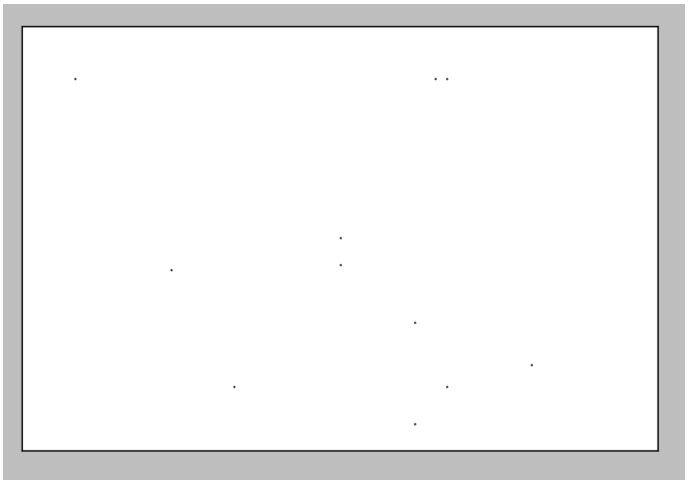
Graphs (which I used to check if my DT's were correct). Again courtesy of Triangle library (with some modifications).

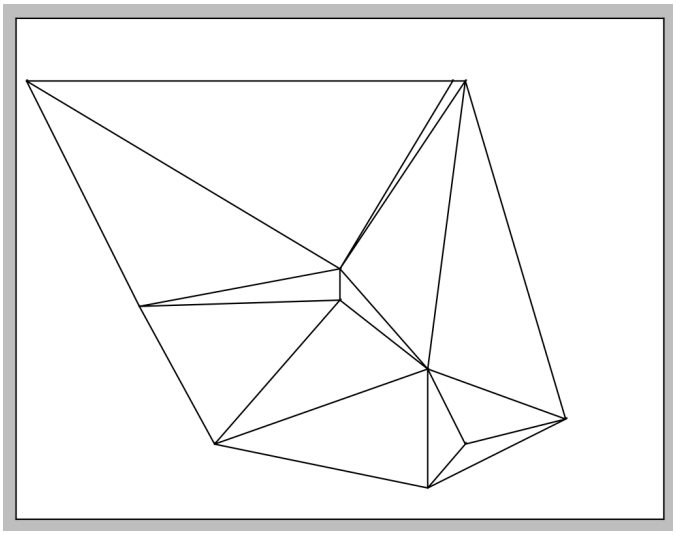
Dots.node:





8.node (custom file)





flag.node