HW #8

JAY LAURA

1. Line Segment Algorithm

My implementation of the line segment algorithm consists of three phases and is designed to work on `NumPy` arrays. Each line segment is represented as start and end coordinates. To keep the implementation simple, only single segment lines are supported. The implementation could be extended to support multi-segment lines by adding an additional level of looping. The phases to check for line intersection are:

(1) First, the algorithm checks for overlap in the bounding boxes. This is an inexpensive check that is able to rule out many possible intersections. The use of the keyword `continue` breaks exits the for loop for the current iteration. This is in contrast to `break` that completely exits the for loop.
(2) Next we use a matrix representation (vector) of each line to check for intersection. This is accomplished by computing four determinants, one for each point pair. The sign (positive or negative) of each determinant indicates which side of the line a point falls. The determinants are assessed in pairs, and if the signs are opposite for each pair, i.e. det(A) and det(B) have opposite signs, then we know that the points fall on opposite sides of the vector. If the determinants for both pairs pass the opposite test, we know that a vector connecting those points must intersect. This check is an extension of a transitional 3-D vector intersection test. The test also support parallel and collinnear lines, as the determinant of one or more point pairs would be 0. This implementation does not alert the user to this, but does perform this check. This check is implemented in `NumPy` and is therefore quite efficient.
(3) Finally, if the vectors do intersect, we compute the point of intersect by using an orthogonal representation of the vectors and the dot product. This is described fully in the code.

2. General Program Overview

The code used is an extension of the lecture 8 GUI code. I have implemented buttons to open a pair of shapefiles using the `TkInter.FileDialog` and restricting the available files to shapefiles. Using the code from homework 7, the shapefile is converted to pixel (monitor) space and visualized. Once visualized two intersetion checking implementations are available to the end-user. First, they can interactively watch as each intersection is identified and plotted using `turtle`. As this involves extensive GUI interaction, the process is quite slow. For more interest is a fast implementation.

The fast implementation utilizes the Python `multiprocessing` module. For this implementation a queue is created to allow for the passing and aggregation of a list of intersection vertices. This implementation works as follows:

- Determine the number of available processing cores
- Compute the breaks in the input line shapefile to load balance the computation over all cores
  - This is not a spatial domain decomposition and total line length is not accounted for. Therefore, the total processing time will not be truly balanced over all cores as a core with greater total line length, i.e. the sum of line lengths in a particular process, will likely compute for a greater total time[1].
- Initialize a processing queue to manage communication between the child processes and the mother processes.
- Generate a list of processes and and a list of process objects
- Start the multiprocessing using `start()` and `join()`
- Within each child processes append a list of intersection and place those into the parent's processing queue.
- Combine the lists of intersections, in serial, and plot them.

Total processing time, using the fast implementation, is then reported to the end-user. We see total processing times of approximately 4 seconds for $n = 100$. That is $100 \cdot 100$ intersection checks.

3. Data Sets

The data used in these tests has been artificially generated to simplify code implementation and testing. Using `OGR`, a GDAL library, a user is able to generate n polylines with random end points. The `create_polyline.py` script contains the sample data generation code. Users supply a shapefile name, (e.g. 1000_lines) and a total number of lines, (e.g. 1000).

4. Improvements

This code could be further improved in two ways. First, support for multi-part polylines should be implemented. Second, the code swaps between Python `list` objects and `NumPy ndarray` objects. This is a clear performance hit and representation solely as an `ndarray` should be implemented. The output intersection shapefile is currently rotating 90 degrees. This functionality has not been debugged.

[1]This is readily apparent when processing two shapefiles with 1000 lines.

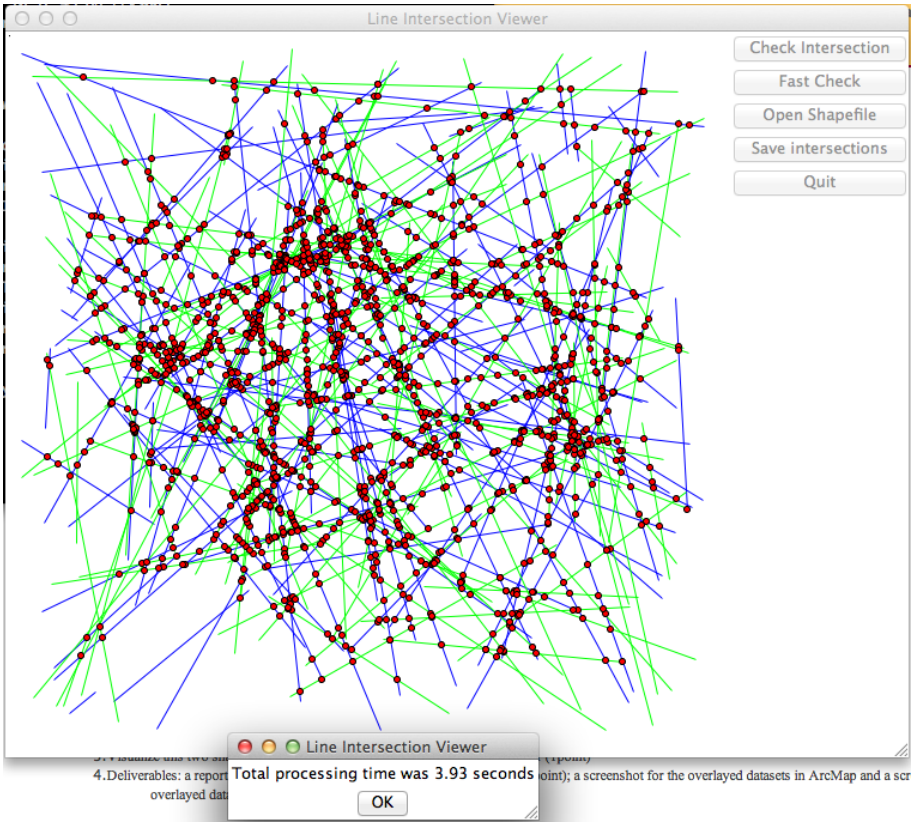

FIGURE 1. Visualization in TkInter

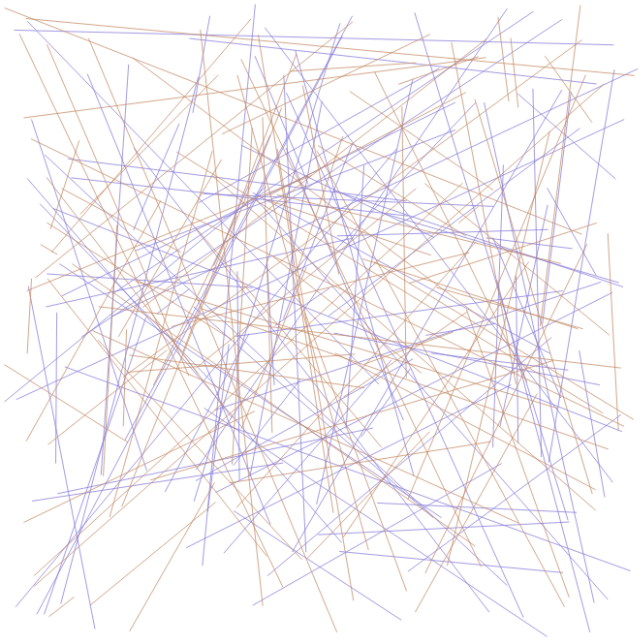

FIGURE 2. Visualization in QGIS