Thinner: An Algorithm for Reducing Points in Vector Graphic Data

Dave Anderson – UMass Computing Center – March 1984

When you're digitizing a map by hand, it's impossible to determine just how many points you need to adequately describe a geographic shape. You tape your map to the digitizer tablet, mentally break the map down into polygons, and start moving the cursor around the outlines, clicking on a point every sixteenth of an inch or so. Around rivers, islands and peninsulas, you probably click a lot more than that, attempting to capture every nuance of the shoreline. And after a few hours, bleary-eyed and on the verge of a carpal tunnel meltdown, you click indiscriminately on everything.

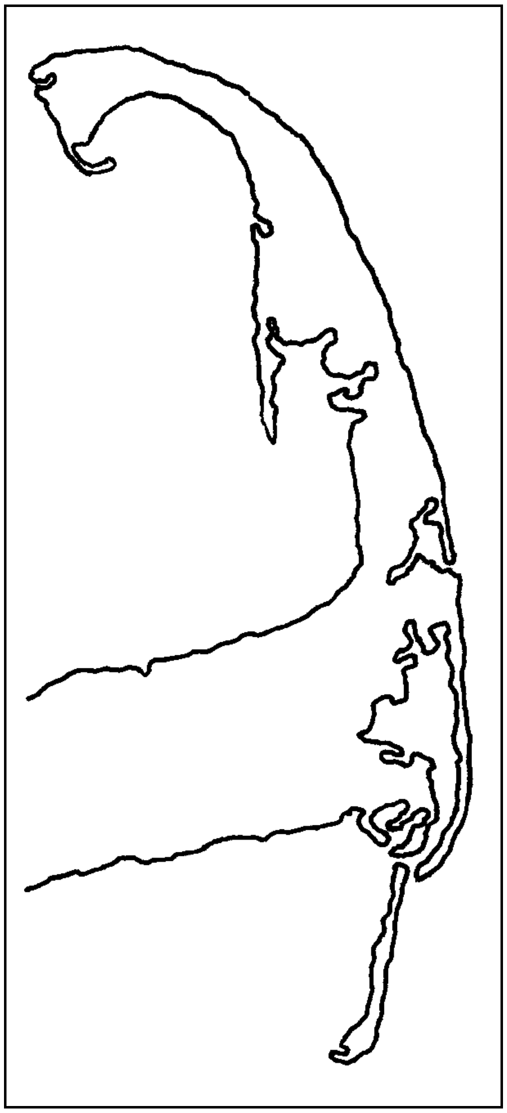
Figure 1. The original vector graphic file of Cape Cod, with more than 1000 data points.

The result is an enormous file of vector graphic data, probably a lot more data than is actually useful. Your efforts to capture every little twist and turn of rivers and coastlines ends up working against you. What's needed is a software "comb" to intelligently simplify the file, stripping out those points that don't contribute much visual information.

What's the criteria for a point that "contributes visual information?" One of the main factors is the resolution of the plotting device. On an analog device like a pen plotter, any point on a line that moves the pen less than half the width of the line is probably not necessary. On a digital device like a computer monitor, any change in line direction less than a half pixel or so in size can't be seen, due to the normal aliasing on the screen. Such points just take up storage space and slow down the rendering process. And if your end goal is to compute the area of the land mass, unnecessary points can generate huge numbers of tiny polygons, slowing the process down to a crawl.

Figure 1 shows the outline of Cape Cod, digitized by hand. There are over one thousand points in this data set, and when it's plotted out at full resolution, it's a faithful copy of the original image. Figure 2 shows a thinned version of the same data, slimmed down to a lean 160 points. The second data set contains points in the first, but only those points determined "significant" by the algorithm.

"Thinner" is a C procedure for thinning a linked list of X-Y points. The code can be easily ported to any language that supports linked lists and floating-point arithmetic. The code can also be used to set multiple levels of resolution for one data set. This way, one data file can be used to render either a full-page image or a postage stamp sized icon. It doesn't matter what kind of values X and Y are – latitude and longitude, or millimeters on the digitizer tablet will both work, as long as the noise threshold is set appropriately.

The algorithm used is a recursive divide-and-conquer approach. Each continuous line is represented by a linked list of points (and the outline of Cape Cod shown here is one long line). Each point consists of an X and Y real value and a boolean flag, which gets set if the point is to be retained in the stripped-down data.

For the line segment in question (which is initially the entire set of points), Thinner flags both end point for retention. Then it calculates a straight reference line between the two ends. It walks through the line segment point by point to find the point causing the greatest excursion off of this reference line, to see if it's less than the noise threshold. If so, then the work on this line segment is finished – rendering a straight line from beginning point to end point is sufficient to represent the line. But if the excursion is greater than the noise threshold, Thinner recursively passes the left part (beginning to high point) and the right part (high point to end) back through the algorithm.

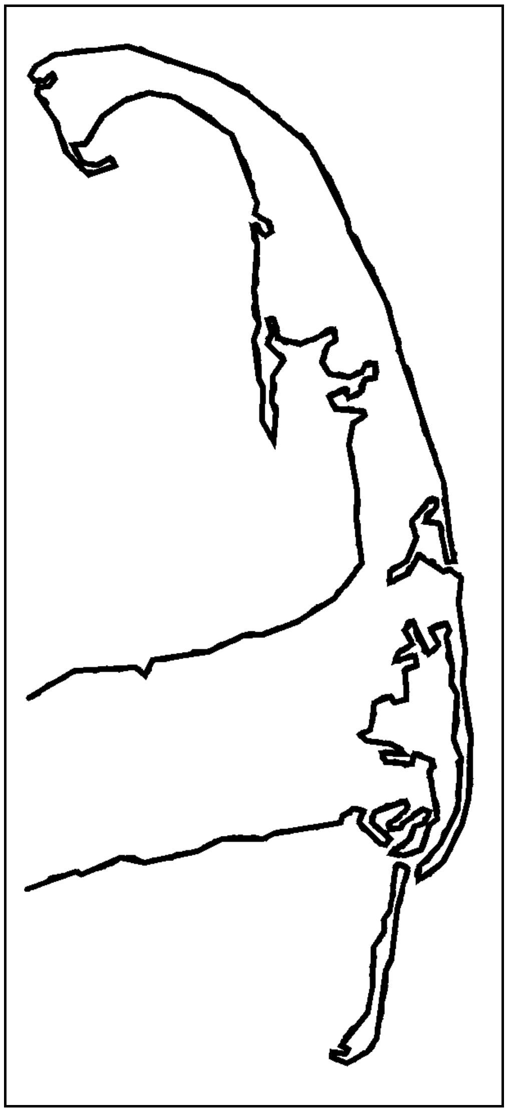

Figure 2. After thinning, the outline has only 160 points, yet looks quite similar.

Figure 3 illustrates the Thinner algorithm at work. On this sample line segment, a reference line is computed between the start and ending points of the segment. The desired resolution is shown as the threshold distance – points closer to the reference line than this distance aren't going to be flagged for retention. The high point is found, and is determined to be farther from the reference than the threshold distance. Therefore, the line segments from the beginning to the high point and from the high point to the end are recursively thinned, as shown in Figure 3B.

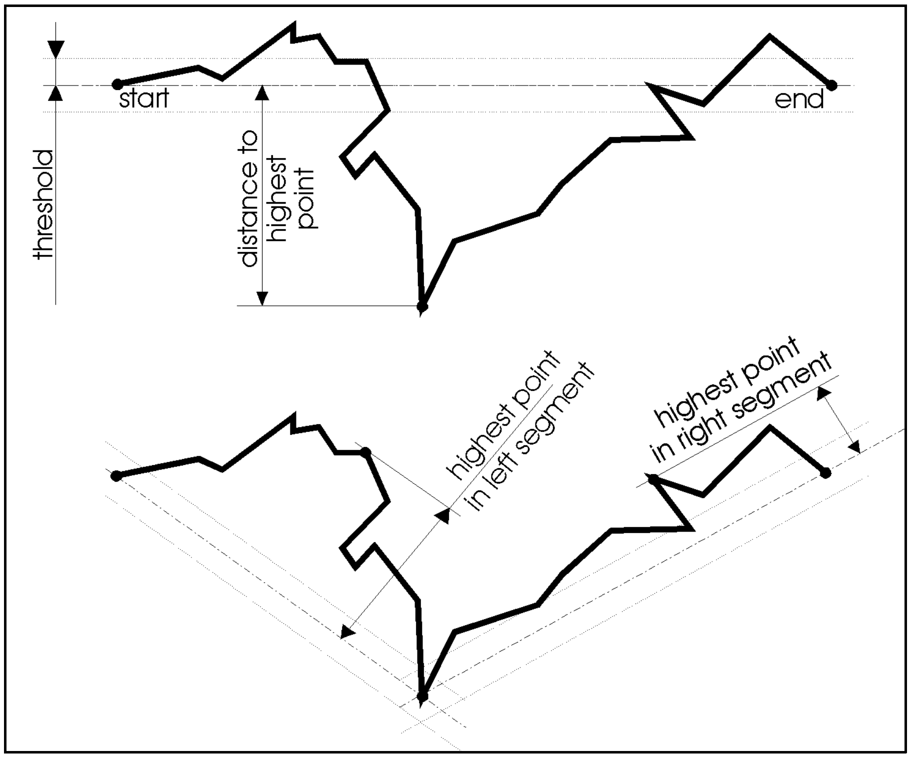
start
end
threshold
distance to highest point
highest point in left segment
highest point in right segment

Figure 3. Find the highest point, flag it for retention, and recursively process points before and after.

When the highest point is within the threshold distance, only the two end points are flagged for retention, and the procedure returns.

One significant exception is illustrated in Figure 4. Points are considered according to their height from the reference line, measured perpendicularly. But what about points that extend off of either end of the reference line? It's possible for a point to be farther from an end than the threshold, but still be technically below the threshold. This would result in removing the point from the data set, when it actually contributes significant information.

The solution is to break the area up into three *zones*: an end zone beyond both the first and the last points, and a middle zone. Any point lying within an end zone has its height computed radially from that end. Any point between the two ends has its height computed perpendicularly from the reference line.

Preliminary thinning can be applied to a data set while it's been digitized, to clean up the data set to a preset maximum level of detail. At the start of each new line segment, the operator generally enters some kind of alphanumeric id for the new segment, which is not CPU intensive. This will allow thinning to be done in the background, without inconveniencing the operator.

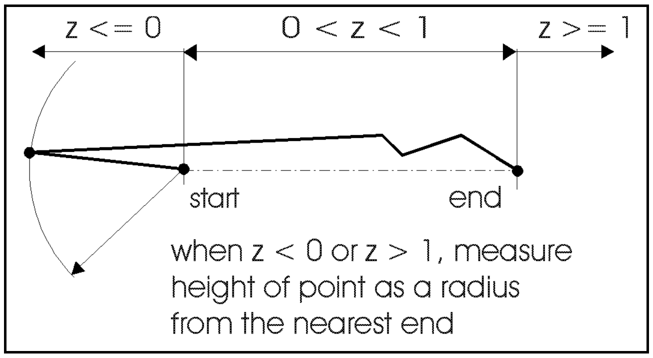
z <= 0
0 < z < 1
z >= 1
start
end
when z < 0 or z > 1, measure height of point as a radius from the nearest end

Figure 4. Zone computation prevents clipping points beyond the ends.

The ideal solution might be to include a set of bits for each point. Each bit would indicates a level of detail the point contributes to. Instead of feeding one particular threshold to the algorithm, pass in an array of thresholds, from finer to coarser.

In this case, the concept of flagging a point for retention is expanded to setting a bit that indicates its inclusion at that level of detail or above. When a line segment has no more points at one level of detail, continue at the next coarser level. Continue in this fashion until all levels have been exhausted. Points which do not qualify for the finest level of detail may be permanently deleted from the data set.

```
#include "math.h"
typedef int bool;             /* define true or false data type */

struct POINT
{
      double x, y;            /* location coordinates */
      bool flag;              /* TRUE if retained at this resolution */
      POINT *next;            /* next point in linked list */
};

/* double high_point(first,last,mid) - returns the elevation of the
   highest point in this segment, measured perpendicularly from the
   baseline computed between the two end points.

      POINT *first, *last     - start and end points on the list
      POINT **mid             - points to address of highest point

      returns: elevation of highest point from base line
*/

double high_point(POINT *first, POINT *last, POINT **mid)
{
      double dx, dy;                /* range of x & y */
      double this_dx, this_dy;      /* distance from start of chain */
      double z;                     /* zone indicator: fractional distance
                                          to the current point (0..1) */
      double sq_hyp;                /* square of the hypotenuse */
      double max_height;
      double this_height;           /* elevation from base line */

      POINT *ptr;                   /* list iterator */

      if (first->next == NULL       /* only one point */
       || first->next == last)      /* only two points */
      {
            *mid = first;           /* nothing to calculate */
            return 0.0;
      }

      max_height = 0.0;             /* will hold highest elevation */
      ptr = first->next;
      dx = last->x - first->x;      /* compute delta x & y */
      dy = last->y - first->y;
      sq_hyp = dx*dx + dy*dy;       /* hypotenuse squared - saves time */

      do {
            this_dx = first->x - ptr->x;
            this_dy = first->y - ptr->y;

            if (sq_hyp == 0.0)            /* first == last: a polygon */
                   z = 0.0;
            else z = -(this_dx*dx + this_dy*dy) / sq_hyp;

            if (z < 0.0)                  /* height = radius from first */
                  z = 0.0;
            else if (z > 1.0)             /* height = radius from last */
                  z = 1.0;

            this_height = sqrt(sqr(first->x + (z*dx) - ptr->x)
                             + sqr(first->y + (z*dy) - ptr->y));

            if (this_height > max_height)       /* new max found */
            {
                  max_height = this_height;
                  *mid = ptr;
            }

            ptr = ptr->next;        /* advance to the next point */
      } until (ptr == last);

      return max_height;
}

/* void thinner(first,last,threshold) - finds and flags all points which
   have a height of threshold or greater.

   Strategy: find highest point. If it's greater than the threshold,
             flag it and recursively thin first point to high point,
             and high point to last point.

      POINT *first, *last     - start and end points on the list
      double threshold        - sets the threshold for thinning
*/

void thinner(POINT *first, POINT *last, double threshold)
{
      POINT *mid;             /* pointer to high point */

      first->flag = TRUE;     /* always flag ends for retention */
      last->flag = TRUE;

      if (high_point(first, last, &mid) >= threshold)
      {
            mid->flag = TRUE;                   /* flag the high point */
            thinner(first, mid, threshold);     /* recursively do the */
            thinner(mid, last, threshold);      /*   points around mid */
      }
}
```