Javier Jimenez
Jose L. Navalon

Some Experiments in Image Vectorization

The application of vectorization algorithms to digital images derived from natural scenes is discussed. It is argued that the fractal nature of these scenes precludes some of the savings in storage expected from vector over raster representation, although considerable savings still result. Experimental results are given. Algorithms for contour following, line thinning, and polygonal approximation well adapted to complex images are presented. Finally, the Map Manipulation System, an experimental program package designed to explore the interaction between vector and raster information, is described briefly.

Introduction

It is often convenient to transform digital images between raster format, in which the images are represented by two-dimensional arrays of intensity values, and some vector code, such as a set of polygonal approximations to the contours of homogeneous regions. Both representations have their advantages, and in many applications they coexist [1]. In this paper we refer to the operation of going from raster to vector format as vectorization. This operation is well understood in principle, but its application to complicated images presents practical problems, some of which are discussed here; also, there is still a lot of room for improvement in the efficiency of the algorithms used in the process, and some new algorithms, or variants of old ones, are described. Finally we give a short overview of the Map Manipulation System, a program package which is being developed at the IBM Scientific Center in Madrid as an experimental vehicle for the study of the vector-raster interface and which has been used in a variety of studies during the last few years [2].

A particularly interesting subject is the amount of space needed to store an image. It is widely felt that the code for a vector image is much more compact than that for a raster one, and this has been mentioned often as the key advantage of the former over the latter, especially when data base applications are involved [3]. The basic argument is that, if the linear dimension of a region is D, measured in pixels, its area, and therefore the number of bits needed to store it as a binary array, is proportional to D^2, while its contour contains only a number of points proportional to its perimeter, which should be of order D. Since each point needs on the average $2 \times \log_2(D)$ bits to encode its coordinates, the reduction in storage achieved by the contour representation should be at least of order $D/\log_2(D)$, which can be substantial for large regions. In practice, contours are often approximated by polygons (linear or otherwise), and only the vertices of the polygons have to be retained; the final amount of storage depends crucially on the complexity of the image and on the degree of approximation desired and, depending on these factors, it might be desirable to use different types of coding. We address this question in the next two sections before discussing the details of how contours are actually obtained.

Approximation and storage compression

Consider a line in a digital image. If certain smoothness conditions are satisfied, it is possible to show that its length L, as measured by the total number of points, is proportional to its outer dimension D, defined, for instance, as the square root of the area of the circumscribed rectangle. As noted above, such a line can be coded in $2L \times \log_2(D)$ bits by just listing the coordinates of all its points. Chain codes [4] are somewhat more compact since only the direction to the next pixel has to be stored at each point and, as there are usually only eight different directions, only $3L$ bits are required. A much shorter representation can be achieved by approximat-

ing the line by some piecewise smooth function and storing only the discontinuity nodes and the parameters of the interpolating functions. Under this scheme the number of nodes clearly depends on the degree of approximation desired; this dependence can be complicated but, if the approximation error is far from any of the scales characteristic of the original line, it follows from simple dimensional arguments that the number of nodes, $N(d)$, associated with an approximation threshold d should behave as

$$N(d) \sim d^{-H} \tag{1}$$

for some exponent H. The argument applies only to problems which are random enough that reasonably different scales can be considered independent, such as turbulent motion [5] or, presumably, physical landscapes. Under these circumstances the ratio between the number of nodes needed for two different approximations can only depend on the ratio of the accepted errors, and the only functions satisfying that condition for all possible pairs of values are powers. Note that the argument breaks down as soon as the error approaches a preferred scale of the problem. In a digital image two such scales are the outer dimension of the object and the pixel size. It is shown below that, far from those limits, Eq. (1) is approximately satisfied in all the cases studied. The constant of proportionality can be estimated by noting that, for large approximation errors (of order D) the whole line is well represented by a single segment and, therefore,

$$N(d) \approx (D/d)^H. \tag{2}$$

In digital lines the accuracy desired is often one pixel, which is in a sense the "intrinsic" accuracy of the original image. The storage needed to code all the nodes is then proportional to

$$D^H \log_2 (D), \tag{3}$$

which is to be compared with D^2 for the raster representation. Whether Eq. (3) represents a savings of storage or not depends on the value of H. A case which is especially well studied is the approximation of smooth lines by polynomial splines [6, 7]. The results depend on the approximation norm used; in this paper we always measure error by the maximum Euclidean deviation between the original line and the approximation. This norm appears to be more appropriate for natural lines, which are often discontinuous and convoluted, than "smoother" norms like the mean square deviation [8]. Under those conditions, and for splines of order n, it can be shown that [7]

$$H = 1/(n + 1), \tag{4}$$

so that linear polygons use a number of vertices which is only proportional to the square root of their size.

Unfortunately, this estimate is not applicable to lines derived from natural scenes. Equation (4) requires that the original line together with its $(n + 1)$th first derivatives be continuous and bounded, and natural lines are often so convoluted and full of small detail that they cannot even be considered differentiable. When such "fractal" lines are approximated by polygons with sides of fixed length, the number of sides needed follows a law of the type (1) with an exponent which can be identified with the fractal dimension of the line [9]. Smooth lines have a fractal dimension of 1, while more complicated objects have higher dimensions. When such lines are approximated by polygons using the maximum deviation norm, we can still expect a dependency of the type (1), but the specific value of the exponent given in (4) cannot be expected to hold anymore.

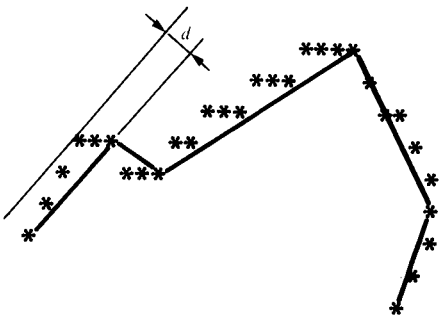


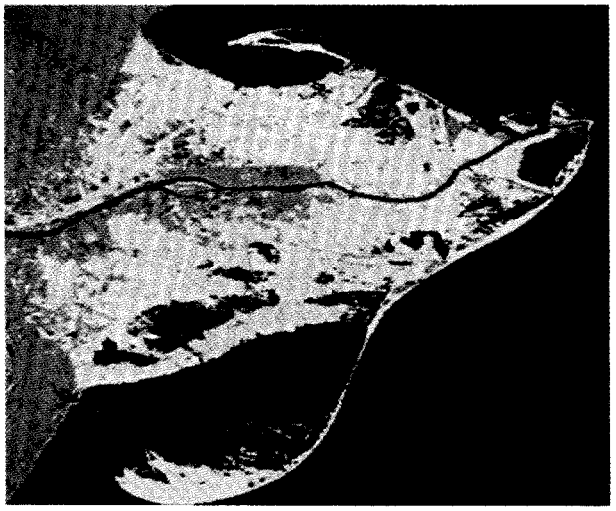
Figure 1 Definition of error in polygonal approximation algorithm.

Some experimental exponents found in the approximation of natural lines are given in the next section. They are found to be of the same order as the measured fractal dimension, although not quite equal to it.

Experimental results

The different possible implementations of the polygonal approximations discussed previously can be classified into two broad groups, according to whether the positions of the nodes are optimized globally using some iterative process [8, 10], or computed sequentially along the curve by optimizing each node as a function of the known position of the last one [11]. Algorithms of the first kind can generally be shown to converge to an optimum, at least if some suitable initial guess is available, while those of the second are probably always suboptimal due to their essentially local nature. However, sequential algorithms are usually much faster than iterative ones and have been found experimentally to behave almost as well [11].

The method used for our experiment is a sequential one. Briefly, a point on the curve to be approximated is chosen as origin, and a search is made for the farthest point along the curve for which the approximation error does not exceed a given threshold (see Fig. 1). That point is stored as a node



Water

Rice

Swamp

Unclassified

Figure 2 Classification map extracted from a LANDSAT image [14]. Several classes, used in the text, are shown by themselves. Image size is 380 × 450.

and used as a new origin. We treat the error associated with a segment as a nondecreasing function of the position of its end point and consider only points on the original curve as candidates for nodes. The first assumption holds strictly only

for sufficiently short segments and for smooth curves but is true on the average in most situations. It represents nevertheless the most important tradeoff of our algorithm in terms of optimization performance; the second one, on the other hand, does not introduce any substantial degradation and simplifies the computation considerably. Because of these two assumptions, a binary search algorithm can be used in the optimization, requiring only a logarithmic, as opposed to linear, number of trials to find the farthest allowable point [12].

Binary searches can be initiated in several ways, and the results can depend on the initial guess. To check whether this was so in our case, we tested three different starting methods for the search of each segment. In the first one, the search was always started from the end of the line, and the algorithm was allowed to proceed inwards; in the second, the next contiguous point was chosen, and the search worked by doubling the distance until the approximation threshold was first exceeded; in the third, the number of pixels used for the last segment was used as an initial guess for the length of the next one, and was doubled or halved depending on whether the resulting error was acceptable or not. In all these cases, the search ended with a classical complete binary search [12].

The results obtained by the three methods were generally statistically indistinguishable, providing some confidence that the general search strategy was not unduly biased in any direction. This is also one of the main reasons to assume that the monotonicity assumption discussed above does not introduce a serious degradation of the approximation. Since the extrapolation method converges somewhat faster than the other two, it was the one used in all the results presented in this work.

It can easily be shown that sequential algorithms with binary search need, to approximate a curve of length L using N nodes, a number of operations that is proportional to L, while those using linear searches need L^2N and global methods use L times the number of trials needed for the iteration to arrive at the solution in an N-dimensional space. In the usual case in which the simplex method is used as the iteration method there is no theoretical estimate for this number, but a practical bound is usually given as linear in N [13], making those methods almost as good as linear searches.

We have tested our algorithm on lines arising from two different natural images. The first one is a thematic map derived from a Landsat scene [14]. The map shows different land use classes which span a wide range of "smoothness" or "fractality" when inspected visually, and the outlines of those classes are used as the test lines (see Fig. 2). No effort was made to smooth the regions or to filter the "salt and pepper" noise before extracting the contours. The second image is a direct raster scan of the contour lines in a pre-existing printed map published by the National Geographic Institute of Spain. A reproduction of the original scribe sheet was used, instead of the final map, to avoid interference from other symbols or types of objects (see Fig. 3).

Figure 3 Contour lines in a portion of a digitized map. Image size is 400 × 400.

In Fig. 4 we show results on the evolution of the number of nodes as a function of the approximation error. That number is normalized with the total number of pixels in the original lines, and a large circle (300 pixels in radius) is included for comparison. It is clear that errors below and above one pixel behave differently, obviously because, under that limit, the approximating polygon has to follow all the stray irregularities introduced in the curve by the digitization, while above it those irregularities are bypassed and only the "natural" shape of the line remains. For large error thresholds the circle follows well the square root law given by Eq. (4) for straight line polygons ($n = 1$) and actually approaches closely the theoretical optimum solution (a regular polygon) given by the dashed line in the figure. All other lines are, however, much steeper. While they can still be represented (for large thresholds) by functional forms of the type of Eq. (1), the exponents, H, are greater than 1/2; Table 1 gives a list of those exponents.

Following the reasoning that led to Eq. (3), the number of nodes needed for an approximation threshold of one pixel should approximately satisfy the equation

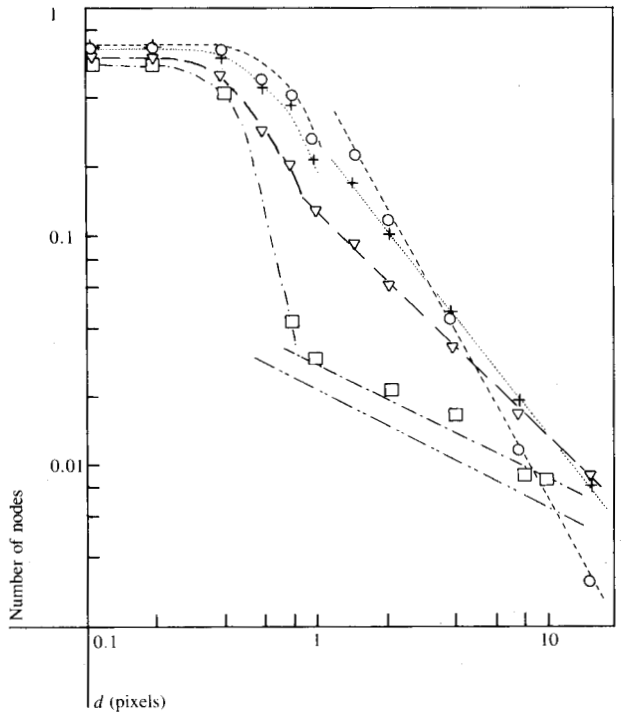


Figure 4 Number of nodes in approximating polygons as a function of error threshold. Test contours are ∇, water; +, rice; ○, unclassified; □, circle; – · · –, optimum circle.

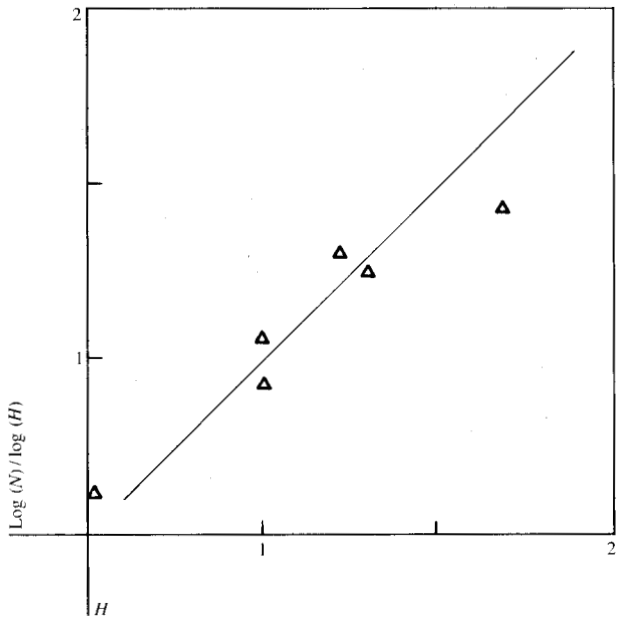


Figure 5 Number of nodes in different contours as a function of the degree of fractality (approximation error = 1 pixel in all cases). Continuous line is Eq. (5).

Table 1 Characteristics of the different classes used in the approximation experiments. Symbols are defined in the text.

Class	*N (1)*	*D*	*H*	F_d	F_m	*L*	*Area*
Water	590	400	1.01	1.12	1.17	4539	83200
Rice	2568	400	1.22	1.88	1.63	12280	36800
Swamp	1712	400	1.29	2.05	1.77	7484	8900
Unclassified	5709	400	1.68	1.75	1.76	21269	50500
Contour line	283	400	1.00	1.20	1.40	1692	—
Circle	55	636	0.50	1.00	1.00	1886	—

$$H \approx \log(N)/\log(D), \tag{5}$$

where D is a typical outer dimension. Both sides of this equation are compared in Fig. 5. The characteristic dimension used for both images is the square root of image area (400 pixels), while for the circle we use the diameter. It is clear that the relation is approximately satisfied and can be used to give a rough *a priori* estimate of storage, although it is also clear that there is substantial scatter due to the particular shape of the objects which cannot be predicted in this way.

Of some theoretical interest is the relation of the exponent H in Eq. (1) to the fractal dimension. This latter quantity can be estimated from the behavior of the number of vertices of the approximating polygons as a function of the *average* length s of their sides. Directly from the definition [9], this dependence should be of the form

$$N(s) \sim s^{-F},$$

where F is now the fractal dimension. This power law dependence is indeed found in our experiments (for approximation errors bigger than one pixel), and the corresponding dimension is given as F_d in Table 1. The same quantity can be estimated directly by approximating the lines by polygons with sides of *uniform* length s; this measure is given in the table as F_m. Even if there seems to be a rough correlation among the three quantities, it is clear that H and F are basically independent parameters.

Contour following

We turn now to the problem of how contours are extracted from images. We assume that all images have already been put into binary form and consist of an object (1) separated from a background (0); the way these binary forms are obtained is the subject of segmentation in classical image analysis and is treated in many places [15]. From the point of view of vectorization, images can be classified into two broad groups. Those in the first group contain mainly extended regions with well defined interiors and can be represented usefully by their outlines (Fig. 2). Those in the second consist

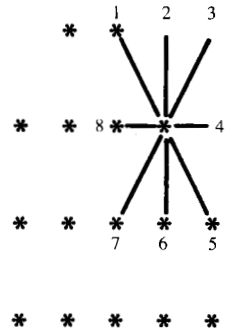


Figure 6 Pixel neighborhood and search order in the contour-following algorithm.

```
                array image,list,point(2),dir(2);
                input(point); output(list,klis,ext);
contour:        begin;
                first=:true; ext=:false; fin=:false;
                dir=:7;
                klis=:1; list(1)=:point;
search:         do k=:1 to 7;
                   dir=:1+mod(dir,8);
                   if image(point+dir) ≠ 0 then go to keep;
hole:              if first and dir=2 then ext=:true;
mark:              if dir=8 then image(point)=:2;
                end do;
                return;
keep:           point=:point+dir; first=:false;
                if fin and point=list(2) then return; else fin=:false;
                if point=list(1) then fin=:true;
normal:         dir=:dir+4;
update:         klis=:klis+1; list(klis)=:point;
                go to search;
                end contour;
```

Figure 7 A contour-following algorithm; directions of movement are defined in Fig. 6.

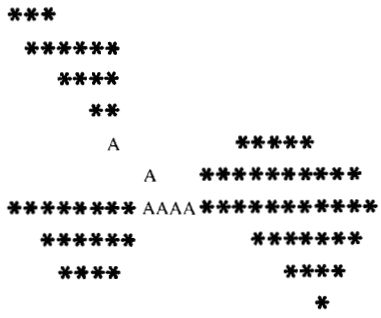


Figure 8 All pixels marked with A are traversed more than once in following the contour of the region.

Figure 9 All pixels in this region belong to at least two different contours.

mostly of elongated features (Fig. 3). The outlines of these features are not meaningful descriptors, and the regions are represented better by their skeletons or median lines. The extraction of these skeletons is usually referred to as thinning [16].

Contour following is a relatively easy operation. A commonly used method [17] explores in rotational order the eight neighboring pixels of a point already known to be on the contour and looks for a transition from outside to inside the object (see Fig. 6). In Fig. 7 we give an algorithm based on this idea. The image is scanned in row order, and the contour-following algorithm is assumed to be entered each time a transition from 0 to 1 is detected. From then on the program tracks the outline and returns only when a closed contour has been completed. To simplify the presentation we have ignored complications arising from the object reaching the edge of the image, and we have assumed that no buffering is needed and that the whole image can be stored in main memory. Both of these difficulties can be overcome easily [2].

Some features of the algorithm are interesting. The statement *normal* chooses the current exterior normal to the object as the starting point for the next search. This is done based on the knowledge of the direction of the last segment of the contour and ensures that the whole contour is followed without accidental shortcuts in cases like the one in Fig. 8. In *mark* some of the pixels already recognized as forming part of the outline are marked. This is necessary to avoid reentering the same contour several times in the course of scanning the image but has to be done carefully since some pixels may have to be reused, not only in the same contour (Fig. 8), but even in different ones (Fig. 9). The solution is to mark with a different code (2) points in the left leading edge of the object, thereby making them unavailable as 0–1 transitions in the scanning while retaining them as 0 to non-0 transitions for the contour follower. Note that the introduction of the new code requires that the image use at least two bits per pixel. The algorithm ends when two consecutive points are found to coincide with the first two points of the contour.

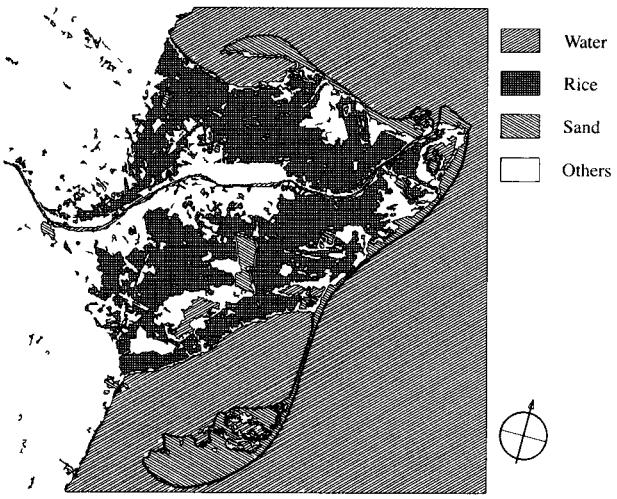


Figure 10 Result of contour extraction on several of the classes in Fig. 2. Map was generated using the Map Manipulation System.

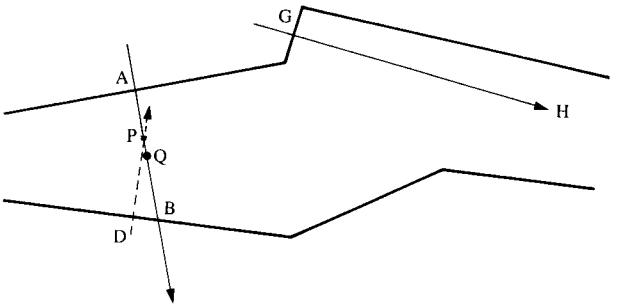


Figure 11 Definition of a point in the skeleton and the approximation used in the thinning algorithm.

Several refinements can be introduced in this basic procedure. As an example, it is possible to mark each contour as an external outline or as an interior "hole" by analyzing the search around the first point (*hole* statement in Fig. 7), thus saving considerable processing in some later operations. It is also possible to do a zero error polygonal approximation while the contour is being generated by keeping only those points in which the direction changes. An example of the use of this algorithm is given in Fig. 10.

Thinning

Contour following is a relatively fast process. Each pixel in the contour needs to be scanned only once, and the total number of operations is proportional only to the length of the final contour. Thinning is a harder problem; classical thinning algorithms [16] work by running a small window (*e.g.*, 3×3) over the image and erasing those pixels that are found to be on the edge, but not on the axis, of the object. The process is repeated until only axial points are left in the image. Since all the pixels in the object have to be checked, the number of operations scales with its area (in fact, with the area times the average thickness of the object) and, since the process is iterative and global, it is difficult to adapt it to cases in which the whole image cannot be held in main memory. Moreover, once the object has been thinned, the result is still in raster form and has to be vectorized.

Recently a new class of thinning algorithms has been proposed that uses only the contour of a region to estimate its skeleton [18, 19]. In fact skeletons were originally defined in terms of the outline only [20]. A commonly accepted definition is the set of all those points interior to the object and equidistant to at least two points of the contour along interior normals. This definition is difficult to implement directly, but a good approximation is easily computed, at least in those cases in which the object is thin (a few pixels wide). Those are, of course, the cases in which the skeleton representation is most interesting. These algorithms present several advantages with respect to the old ones. First, the vectorization is done at the very beginning of the process, and the storage requirements are much reduced during the whole operation. Second, all independent contours are now represented, and processed, as separate entities so that, even if buffering is needed for the image, the whole list of points for a single contour usually fits in memory and the thinning algorithm need not be buffered at all. Lastly, the number of operations scales with the perimeter of the contours. In fact, since we will see that the algorithm can run on polygonalized data, the storage estimates given in the first part of this paper apply to the savings in computer time.

Consider now the object in Fig. 11. At point A we draw the interior normal to the contour and extend it until it intersects the opposite side of the outline at B. The midpoint, Q, of AB is taken as a point in the skeleton. It is clear that, according to the definition, the true skeleton would run through P, defined by $AP = PD$, but, for objects which are thin with roughly parallel sides, the error is small. In fact, even in the case in which the angle formed by the segments on both sides of the contour is 45 degrees, the error is less than $1/4$ the thickness of the object and is therefore less than one pixel in most practical cases.

In applying these ideas some precautions must be taken to avoid spurious intersections with distant contours (see GH in Fig. 11). These can be avoided by defining an *a priori* maximum local thickness for the object and discarding those intersections that occur beyond that distance [18]. This thickness can be estimated by inspection or by comparing, for example, the area contained in the contour with its perimeter. In fact, the possibility of defining a characteristic

thickness, far from being a weakness of the algorithm, can be considered a strong point. A thin object is only thin in a relative sense, and some outside criterion has to be used to decide whether a local thickening of the object should be interpreted as a short branch or just as a slightly wider section (see Fig. 12).

The general structure of a thinning algorithm based on these ideas is described now; a more detailed description can be found in [2]. Consider a ribbon-like object whose contour has been digitized and approximated by a polygon (Fig. 13). We assume that the contour contains two parts opposite to each other, which we identify as outer and inner parts. These parts can belong to the same connected contour or to different ones and, in identifying them, the hole labeling facility of the contour follower can be of great help. The algorithm works by casting normals from the midpoints of the segments in the outside contour towards the inner one and then projecting the vertices of both contours parallel to those normals.

We first choose an arbitrary segment in the outer contour (1–2 in Fig. 13) and construct the inwards-facing normal at its midpoint. Note that the direction of this normal is known from the sense in which the contour was followed. If the normal crosses a segment on the inner contour within the prescribed distance, this segment (51–52) is chosen as the inside starting point. Finding this first segment involves an exhaustive search through all the candidates in the inside contour, but once it is found all other searches are sequential and involve only a few segments. The correct identification of this first segment is also very important for the subsequent behavior of the algorithm; if the first pair is not chosen right, the pairing of the following segments can be led completely astray. To avoid this we try to find a segment that is as "well behaved" as possible. Thus, if its mid-normal does not intersect any segment or if there is any doubt as to which intersection to choose, the starting segment chosen in the outer contour is skipped and another one is tested. Since the subsequent algorithm runs over all unused parts of the contour, a segment skipped in this fashion will eventually be found again and completed.

The midpoint normal is the basic direction in which all points are projected while processing an outside segment. Its center point is considered to belong to the skeleton. Then, the two endpoints of the inside segment (51 and 52) are projected back into the outside and, if their projections fall within the outside segment, their center points are also added to the skeleton. The next inner segments (50–51, etc.) are then selected in sequential order, and the projection step is repeated as long as the base of all the projections falls within the segment (1–2). When one of them does not (point 50 in the figure), the endpoint of the *outside* segment being processed is projected along its mid-normal, its center point is added to the skeleton, the next outside segment is selected, and the process is repeated with the new normal. This time, however, it is only necessary to search sequentially in the inside contour starting with the last segment used.

In this way it should be possible to produce a skeleton in which all the inside and outside segments are represented.

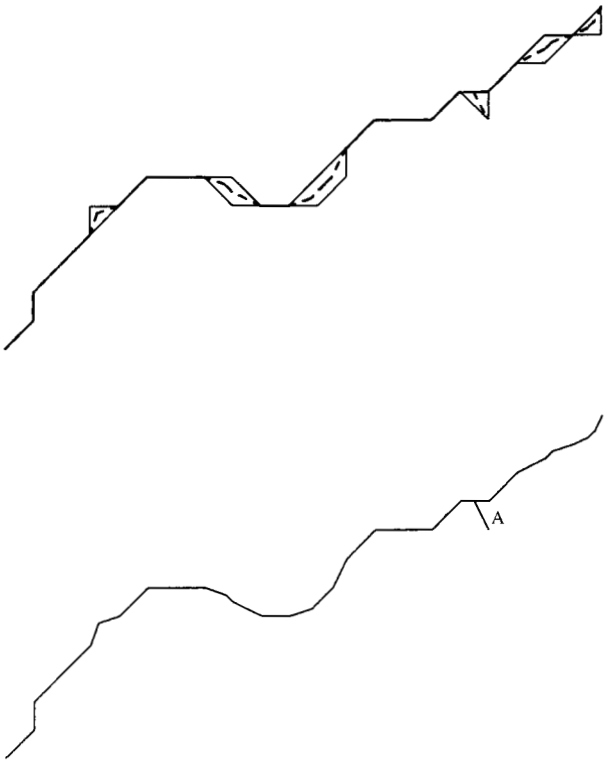


Figure 12 Thin object and skeleton. Branch A can be suppressed by selecting a wider *a priori* thickness for the object.

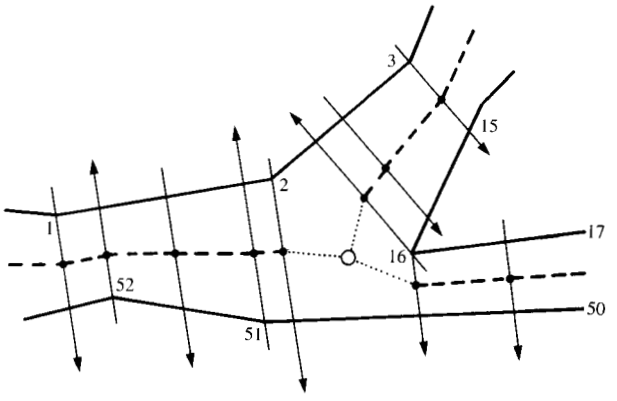


Figure 13 Definition of the thinning algorithm described in the text.

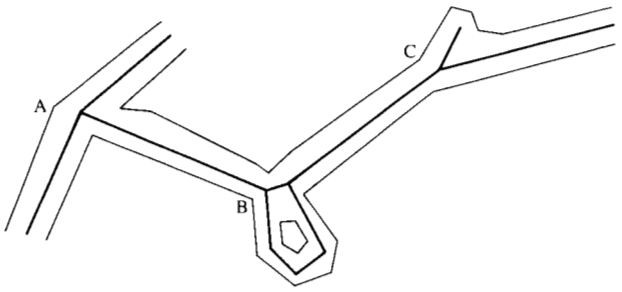


Figure 14 Result of the thinning algorithm on a complex line. Different branch points are treated in different ways after the thinning.

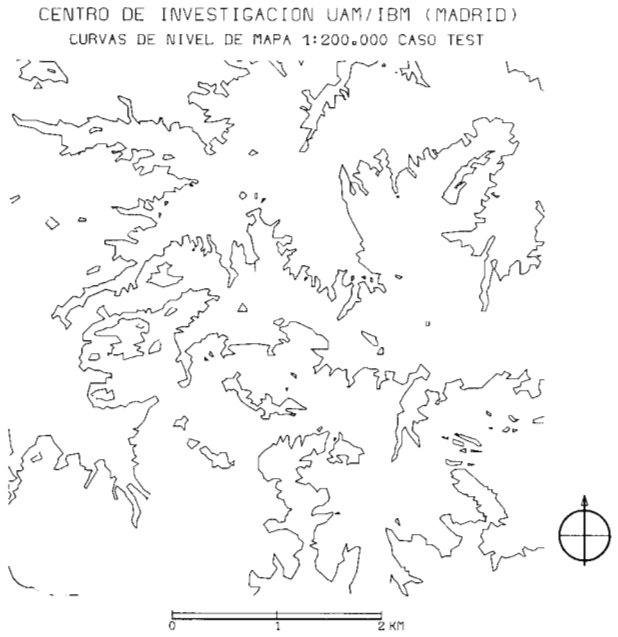


Figure 15 Thinned version of map in Fig. 3.

When a gap occurs, as with segment (2–3) in Fig. 13, a branch is initiated and the current arc of the skeleton is considered complete. The process of initiating a branch is equivalent to the one of starting a new skeleton, although "used" segments are, of course, no longer available. The construction of an arc ends either when a branch point is reached, as above, or in a tip; a tip is a point in which the inside contour runs into the outer one. When all the segments in the contour have been used, the only operation remaining is to find common points to join branches together. The solution given in Fig. 13 (the center of gravity of all endpoints) is probably as good as any other.

The result of the thinning is now a set of polygons connected into a network like the one in Fig. 14. Some branching points (A) are intrinsic to the problem, and their treatment depends on the specific application, but others (B or C) arise from noise in the image and should be corrected. There are a variety of ways in which this can be done. Perhaps the simplest is to consider the network as an algebraic graph and to use a path-tracing method to optimize the result. The little branch in C is easily discarded by noting that it is too small and isolated. The loop in B can be fixed by taking the longest available path. The result of thinning the line image of Fig. 3 is shown in Fig. 15.

The Map Manipulation System

An example of an integrated system using these algorithms is the Map Manipulation System, which is a program package under development at the IBM Madrid Scientific Center with the purpose of exploring the interrelations between raster and vector graphic information. Its nucleus is a series of vectorization and polygon manipulation algorithms that allow the generation of graphic structures either from images or by integration from outside sources. The package is interfaced with a set of interactive graphic display routines, which work either on a color raster display terminal (IBM 7350 or RAMTEK 9351) or on a black and white calligraphic screen (IBM 3277 Graphic Attachment), and with a small data base in which vector graphics, some alphanumeric information, and raster images are held in the form of standard CMS files.

The whole system is driven by a monitor in which functions are chosen from a set of hierarchically organized menus; a batch version of the graphic programs also exists. The following is a short list of the functions available in the system:

- Load and create external files containing graphic, alphanumeric, or raster image data.
- Display and interactively edit images if the appropriate terminal is available.
- Manipulate images to prepare them for vectorization.
- Perform contour following, thinning, and polygonal approximation. These are the operations discussed in this paper.
- Create and update an internal graphic data base using this information.
- Edit this internal data base [21]. Editing functions include

  Display and graphic interrogation functions. In a raster terminal, vector information can be overlaid on images.
  Deletion, merging, and renaming of graphic objects.
  Geometric modification of objects.
  Introduction of alphanumeric labels associated with specific points.

The system has been developed as an experimental tool and has grown mainly as a result of the requirements of

different users. The primary interest was in geographic work, and most of the work done has been in this area. Usage to date includes the direct digitization of pre-existing maps, the generation of land use maps from Landsat and aircraft images, and the integration of both types of information. Parts of the system have been used in other contexts; the polygonal approximation and contour following have been used in quantifying fracture patterns in rocks [22]. A somewhat unexpected application has been as a graphic aid in the display of purely raster information; the contour follower has been adapted as a contour line generating algorithm for general images [23], and the whole system has been used to generate three-dimensional views of a turbulent flow from a series of parallel sections in a high-speed film [24].

Conclusions

We have shown that binary images can be vectorized efficiently and economically. The resulting representation can generally be stored in much less space than would be required for the raster image. The exceptions are very complex or "busy" images. Both the amount of work needed to vectorize an image and the space needed to store the result scale roughly with the length of resulting lines. We show in the first part of the paper that this length is only linear in the size of the image for relatively smooth cases, but that for very complex cases it grows almost as fast as the area. It might not be worthwhile to code these images in vector form. On the other hand, differentiable contours can be coded as polygons with a number of vertices that grows only with the square root of their size.

At this point some reference must be made to alternative representation schemes for digital images. In recent years, a lot of attention has been paid to the statistical compression of images by a variety of methods that rely on the fact that images are not completely random [25]. In fact, this is the same reason that polygonal contour approximations need less storage space than the original images. The compression achieved by these statistical methods can be very important and, in some cases, on the same order as the ones presented here. In [26] a comparison is made between some particular polygonal and statistical representation schemes. The results are similar to the one expressed above; relatively simple images are handled best by polygonal approximations, while complex ones are represented better by other schemes. Statistical compression is not limited to two-dimensional images; [27] studies the best statistical representation of chain codes, showing that only two bits per pixel (instead of three) are needed for them.

In coding a particular class of images all these methods have to be considered, but it would be a mistake to judge them only in terms of the final storage requirements. In most statistical representations the image has to be decoded before it is useful for some applications. As an example, in computing the intersection of two regions in raster form, or in counting the number of independent regions of some kind, the image has to be decoded into full two-dimensional representation. This can be inconvenient when not enough main memory is available to hold the entire image and the operation does not adapt easily to buffering. On the other hand, most geometrical operations can be performed directly on polygons, and in this sense polygonal approximations have the advantage of being in a "final" form.

Before full advantage of the vector representation can be realized, a complete set of polygon manipulation algorithms has to be developed. This subject constitutes the area of computational geometry [3] and is, at present, one of the most active areas of research in computer graphics. An interesting consequence of all this is that the traditionally separate fields of digital image processing and computer graphics are becoming closer. This is due in part to the availability of efficient interfacing algorithms, such as the ones described here, but also to the appearance on the market of relatively cheap raster graphic terminals capable of displaying both types of information. As this trend develops, it is to be expected that the present distinction between the two fields may come to be considered no more than a historical accident.

Acknowledgment

Javier Jimenez wishes to acknowledge the hospitality extended to him by the Department of Applied Mathematics of the California Institute of Technology during much of the preparation of this paper.

References

1. M. Chock, A. F. Cardenas, and A. Klinger, "Manipulating Data Structures in Pictorial Information Systems," *Computer* **14,** 43–50 (1981).
2. J. L. Navalon and J. Jimenez, "The Map Manipulation System," *Report No. SCR-03.82,* IBM Scientific Center, Madrid, Spain, 1982.
3. G. Nagy and S. Wagle, "Geographical Data Processing," *Computing Surv.* **11,** No. 2, 139–181 (1979).
4. H. Freeman, "Computer Processing of Line Drawing Images," *Computing Surv.* **6,** No. 1, 57–97 (1974).
5. L. D. Landau and E. M. Lifshitz, *Fluid Mechanics,* Pergamon Press, Inc., Elmsford, NY, 1959, pp. 120–123.
6. J. R. Rice, *The Approximation of Functions, Vol. II,* Addison-Wesley Publishing Co., Reading, MA, 1969.
7. D. E. McClure, "Nonlinear Segmented Approximation and Analysis of Line Patterns," *Quarterly App. Math.* **33,** 1–37 (1975).
8. T. Pavlidis, "Waveform Segmentation Through Functional Approximation," *IEEE Trans. Computers* **C-22,** 689–697 (1973).
9. B. Mandelbrot, *Fractals: Form, Chance and Dimension,* W. H. Freeman and Co., San Francisco, 1977.
10. C. deBoor and J. R. Rice, "Least Squares Cubic Spline Approximation, II: Variable Knots," *Tech. Report 21,* Computer Science Dept., Purdue University, Lafayette, IN, 1968.

11. J. R. Rice, "General Purpose Curve Fitting," *Approximation Theory,* A. Talbot, Ed., Academic Press, Inc., New York, 1970, pp. 191–204.
12. D. E. Knuth, *The Art of Computer Programming, Vol. 3,* Addison-Wesley Publishing Co., Reading, MA, 1973, Ch. 6.
13. R. G. Bland, "The Allocation of Resources by Linear Programming," *Scientific American* **244,** No. 6, 126–144 (1981).
14. F. Orti, "A Center-Covariance Adaptive Clustering Algorithm," *Report No. SCR-02.78,* IBM Scientific Center, Madrid, Spain, 1978.
15. W. K. Pratt, *Digital Image Processing,* John Wiley & Sons, Inc., New York, 1978.
16. T. Pavlidis, *Structural Pattern Recognition,* Springer-Verlag, Berlin, 1977.
17. S. A. Dudani, "Region Extraction Using Boundary Following," *Pattern Recognition and Artificial Intelligence,* C. H. Chen, Ed., Academic Press, Inc., New York, 1976, pp. 216–232.
18. B. B. Chauduri, "A Simple Method of Thinning," *J. Instn. Electron. & Telecom. Engrs.* **24,** 264–265 (1978).
19. B. Shapiro, J. Pisa, and J. Sklansky, "Skeleton Generation from x, y Boundary Sequences," *Computer Graph. & Image Process.* **15,** 136–153 (1981).
20. H. Blum, "Biological Shape and Visual Science, I," *J. Theor. Biol.* **38,** 205–287 (1973).
21. J. Jimenez and J. L. Navalon, "The Structure of Queries on Geometric Data," *Data Base Techniques for Pictorial Applications,* A. Blaser, Ed., Springer-Verlag, Berlin, 1979.
22. Luis Montoto, "Digital Multi-Image Analysis: Application to the Quantification of Rock Microfractography," *IBM J. Res. Develop.* **26,** 735–745 (1982, this issue).
23. N. Garcia, A. Santisteban, and J. L. Carrascosa, "Digital Processing of Biological Images," *Proc. IEEE Vigo Workshop in Signal Processing,* Vigo, Spain, July 1981.
24. J. Jimenez, "Fluid Mechanics, a Test Case for Computer Imagery," *Proc. Int. Symp. on the Future of Computing,* Caracas, Venezuela, June 1982.
25. Special issue on digital encoding of graphics, *Proc. IEEE* **68,** 755–929 (1980).
26. T. Pavlidis, "Optimal Compaction of Pictures and Maps," *Computer Graph. & Image Process.* **3,** 215–224 (1974).
27. T. H. Morrin II, "Chain-Link Compression of Arbitrary Black-White Images," *Computer Graph. & Image Process.* **5,** 172–189 (1976).

Received March 3, 1982; revised June 14, 1982

Javier Jimenez *IBM Spain, Paseo de la Castellana, 4, Madrid 1, Spain.* Dr. Jimenez is a research scientist at the Madrid Scientific Center, where he joined IBM in 1975. His research activities include geographic data processing, computer graphics, digital image processing (in particular, its application to treatment of experimental data in fluid mechanics), and theoretical and experimental study of turbulent flow. Dr. Jimenez has just completed a year at the California Institute of Technology working on large-scale computing problems in fluid mechanics. He received his aeronautical engineer degree in 1969 from the Universidad Politécnica Madrid, his M.S. in aeronautics in 1970, and his Ph.D. in applied mathematics in 1973 from the California Institute of Technology. From 1973 to 1974 he was a research fellow at the California Institute of Technology working on numerical analysis in fluid mechanics. At present he is a part-time lecturer on fluid mechanics and applied mathematics at the School of Aeronautical Engineering at the Universidad Politécnica Madrid. Dr. Jimenez is a member of the American Mathematical Society.

Jose L. Navalon *IBM Spain, Paseo de la Castellana, 4, Madrid 1, Spain.* Mr. Navalon is a research scientist at the Madrid Scientific Center, working in the areas of image processing, pattern recognition, statistics, geographic data processing, vectorial information treatment, automatic vectorization, and computer graphics. He received his degree in civil engineering in 1973 from the Universidad Politécnica Valencia, Spain. From 1973 to 1975 he had an Autonomous University of Madrid and IBM Fellowship to work in pattern recognition and statistics. He joined IBM permanently in 1975.