Line tracking for incremental plotters

R. A. Earnshaw
Centre for Computer Studies, University of Leeds, Leeds LS2 9JT

The characteristics, properties and generative sequences of line tracking algorithms for incremental devices are summarised and the mappings from one into the other are discussed. Extensions to the basic algorithms to exploit a greater variety of possible operations in the basic hardware set of the device and also to cater for various types of curve are summarised. A new method of partitioning based upon derived code sequences and code patterns is presented and is shown to lead to significant code sequence compression (more than 40\%) in the majority of cases, and more importantly a significant overall reduction of between 20\% and 43\% in the total central processor usage. Algorithms for encoding the sequences are presented.

(Received January 1977; revised November 1978)

1. Introduction

The use of an incremental plotter involves the generation of a sequence of movements corresponding to 'best possible straight lines' at some stage in the generation of the picture to be drawn. This arises due to the fact that elemental pen movements can usually only be made in a finite number of directions and therefore pictures that contain lines at orientations other than those of the basic set must represent these by an approximation consisting of a permutation of the appropriate basic movements.

Linear interpolation may be done by hardware or software, and for the latter there are numerous well known algorithms for the generation of these movements in such a way as to minimise the maximum deviation from the straight line join; some of the earliest being those of Stockton (1963), Bresenham (1965), Thompson (1964), and Boothroyd and Hamilton (1970). What is not always so readily appreciated is the relationship between these algorithms or the various ways in which the algorithms may be extended to exploit the facilities of devices containing a greater variety of possible operations in the basic set (Boothroyd, 1974) or even to draw certain types of curve (Pitteway, 1967, 1972). These will be summarised in the following sections.

2. Line generation by sequences

2.1 The Bresenham sequence

Consider the point \((r, q)\) in Fig. 1.

We move to \((r + 1, q)\) axially if this is nearer to the line \(y = (b/a)x\) than is the point \((r + 1, q + 1)\), otherwise we move to \((r + 1, q + 1)\).

Hence we move axially if

\[
(b/a) (r + 1) - q < q + 1 - (b/a) (r + 1)
\]

\[
\text{i.e. } 2(br - qa) + 2b - a < 0 \quad (1)
\]

Let \(S_r = 2(br - qa) + 2b - a\)

If we move axially, then \(r \leftarrow r + 1\) and the new condition is

\[
2(br - qa) + 2b - a + 2b < 0
\]

\[
\text{i.e. } S_r + 2b < 0
\]

Otherwise \(r \leftarrow r + 1, q \leftarrow q + 1\) and the new condition is

\[
S_r + 2(b - a) < 0 \quad \text{which constitute the Bresenham recursive relation (1965).}
\]

(Starting value: \(r = q = 0\), \(S_0 = 2b - a\).)

For example, the Bresenham sequence for the movement from \((0, 0)\) to \((10, 3)\) is as follows:

\[
S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \quad S_7 \quad S_8 \quad S_9
\]

Value of

\[
-4 \quad +2 \quad -12 \quad -6 \quad 0 \quad -14 \quad -8 \quad -2 \quad 4 \quad -10
\]

Movement

a a d a a d a a d a

Where \(a = \text{axial movement}, d = \text{diagonal movement}\)

A useful starting point for the sequence is \((r, q) = (-1, 0)\) then \(S_{-1} = -a\) and the sequence may be generated by

\[
S_i = S_{i-1} + a \quad \text{for } i = 1 \text{ step } 1 \text{ until } a \text{ do}
\]

begin

\[
S_i = S_{i-1} \quad \text{if } S_i < 0 \text{ then } 2 \times b \text{ else } 2 \times (b - a) \text{;}
\]

\[
\text{if } S_i < 0 \text{ then step (axis) else step (diag)}
\]

end

(2)

2.2 The partitioning algorithm

The generation of the minimum sequence may be considered as an algorithm for partitioning a set of elements, where the elements are commands to perform either axial or diagonal incremental movements to produce the optimal straight line sequence.

Given two non-negative integers \(m, n\) (not both zero) then

\[
S_k = a_1 + a_2 + a_3 + \ldots + a_k \quad 1 \leq k \leq m+n
\]

where each \(a_i\) is either \(-m\) or \(+n\) and the sequence \(\{a_i\}\) is so chosen that \(\max|S_k|\) is the least possible (Thompson, 1964).

This is accomplished by choosing \(a_{k+1}\) to be \(+n\) or \(-m\) depending on whether \(2S_k + n - m\) is negative or positive. Given \(S_k\) then

\[
\text{if abs} (S_k + n) < \text{abs} (S_k - m) \text{ then } S_{k+1} = S_k + n \\
\text{else } S_{k+1} = S_k - m
\]

which gives the following algorithm in terms of the variables \(a\) and \(b\) used in 2.1.

\[
S_i = 0; \quad \text{for } i = 1 \text{ step } 1 \text{ until } a \text{ do}
\]

Fig. 1 An incremental step
if \( abs(S + b) < abs(S - (a - b)) \)
then begin \( S = S + b; \) step (axis) end
else begin \( S = S - (a - b); \) step (diag) end

The partitioning sequence for the movement from \((0, 0)\) to \((10, 3)\) is as follows:

\[
S_0 \quad S_1 \quad S_2 \quad S_3 \quad S_4 \quad S_5 \quad S_6 \quad S_7 \quad S_8 \quad S_9 \quad S_{10}
\]

Value of Condition

\[
0 \quad +3 \quad -4 \quad -1 \quad +2 \quad -5 \quad -2 \quad +1 \quad +4 \quad -3 \quad 0
\]

Movement \( a \) \( d \) \( a \) \( a \) \( d \) \( a \) \( a \) \( a \) \( d \) \( a \)

where \( a \) = axial movement, \( d \) = diagonal movement. Thus although the values of \( S \) generated by the partitioning are different to those of the Bresenham sequence, the movements generated are the same. It can now easily be shown that the Thompson sequence can be mapped into the Bresenham sequence by means of a linear relation. For the Thompson sequence the value of \( S(r, q) = (r - q)b - q(a - b) \) since at \((r, q)\) we have performed \( q \) diagonal movements of 'weight' \( - (a - b) \) and \((r - q)\) axial movements of 'weight' \( b \).

Hence \( S(r, q) = (rb - qa) \)

which can be mapped directly into equation (1) for the value of \( S(r, q) \) corresponding to the Bresenham sequence.

Thus if \( S_B \) and \( S_T \) are corresponding elements of the Bresenham and Thompson sequences, \( S_B = 2S_T + 2b - a \) and it may be shown that the condition \( S_B < 0 \) for axial movement in the Bresenham algorithm implies the corresponding axial conditions \( abs(S_T + b) < abs(S_T - (a - b)) \) in the Thompson algorithm. Thus the sequences of movements generated by these different algorithms are the same for the same straight line.

2.3 Extensions to the basic algorithms

Greater variety of basic movements

The spectrum of possible movements in the basic set can be extended by the addition of further intermediate vector movements (e.g. 24 vector mode). A method for exploitation of these additional movements arises from the consideration of the algorithms discussed previously from yet another point of view.

Considering the progress that the basic 8-vector algorithm makes for axial or diagonal movements respectively gives the following result:

Axial steps make progress \( \Delta x = 1 \) unit
\( \Delta y = 0 \) unit

Diagonal steps make progress \( \Delta x = 1 \) unit
\( \Delta y = 1 \) unit

If \( na \) = number of steps in \( x \) direction
and \( nb \) = number of steps in \( y \) direction

Then from \((0, 0)\) to \((x, y)\)

\[
na + nb = x \\
nb = y
\]

Total number of steps = \( na + nb = x = a \)
Number of diagonal steps = \( nb = y = b \)
i.e. \( x = a \) and \( y = b \) in the basic algorithm.

Thus \( a = (na + nb) \) is the total number of elements in the set while \( b \) is the number of elements in the diagonal subset.

Consider the two types of sector shown in Fig. 2.

For Sector 1: Progress in the \( x \) direction by \( na + 2nb = x \)
Progress in the \( y \) direction by \( nb = y \)
(units are those of the 'fine grid').

Total number of steps = \( na + nb = x - y \)
Number of 'diagonal' steps = \( nb = y \)
Hence \( a = (x - y) \) and \( b = y \) are used in the basic algorithm.

For Sector 2: Progress in the \( x \) direction by \( na + 2nb = x \)
Progress in the \( y \) direction by \( na + nb = y \)
Hence \( a = y \) and \( b = (x - y) \) are used in the basic algorithm.

The above approach was first proposed by Boothroyd (1974) and may be extended to cater for all sectors and all possible combinations of long or short vectors where a given direction has more than one possible movement.

Extensions to curves

Line generation using the methods previously described has two principal advantages over \( ad \) hoc methods. Firstly, the generation of the increments for the line involves only one addition operation and tests at each stage. Secondly, the algorithms can be implemented using economical amounts of storage. With two further addition operations it is possible to generate ellipses or hyperbolas or spirals (Pitteway, 1967; 1972). Two more additions are required for display devices which do not accept incremental commands, and two further tests are required to detect possible changes of sector.

3. Line generation by compression

3.1 Repeated codes

The occurrence of sequences of repeated movements (e.g. see sequence in Section 2.1) in the overall sequence of movements corresponding to a particular line suggests scope for compression of such repetitions into a single entity. This enables the total amount of information to be transmitted to the actual device to be reduced, or alternatively, if backing store is used for intermediate storage of the information, this gives reduction in the total space needed for this purpose. Of course, if the information is to be transmitted to the device directly, it could be argued that complete compaction into the original destination co-ordinates should be the first choice rather than an
intermediate compression and subsequent regeneration, and
delay the line generation process until the output to the actual
device. This of course suggests immediate exploitation for
devices where line generation is done entirely by hardware.
However, the former approach was investigated for the case of
a spooling system on to disc and was found to lead to un-
expected results.

From the observation of graphs alone it is clear that certain
constructs are frequent components of larger pictures and these
constructs consist in the main of repetitive sequences. Ex-
amples of these are border, axes and characters, especially
where the latter are produced by generation of points taken
from an individual raster frame (e.g. 10 x 6).

A series of jobs producing output on the incremental plotter
were analysed in detail to determine the proportion of the
total information containing sequences of repeated identical
codes. Two such analyses are shown in Fig. 3; the proportion
of distinct codes (i.e. those corresponding to a 'repeated
sequence' code count of 1) being also given to enable a com-
parison to be made. Clearly a significant proportion (61% in
picture A; 85% in Picture B) of the total space occupied by the
code information is occupied by sequences of two or more
identical movement codes.

It is apparent that long sequences of identical movement
codes are produced by movements in the X or Y directions
alone, or along any diagonal, whereas shorter sequences are
produced during line tracking in any other direction. The
former can be measured on the resulting picture, but the latter
can only be determined by a detailed analysis of all the move-
ment codes corresponding to the picture. This last was the
method adopted for the results presented here.

One way of implementing the compression is to represent
repeated sequences as shown in Fig. 4. The * indicates that
the following four characters are to be taken as representing
a repeated sequence; each character being, say, six bits. The
particular movement code to be repeated is held in the top
six bits, and the number of times it is to be repeated in the
remaining eighteen bits. The latter will usually be sufficient
to cater for the longest possible sequences. With a represen-
tation such as this, sequences with greater than five identical
codes can be compressed, giving a saving of 20% and 51%
respectively for the two jobs shown in Fig. 3 (the actual space
occupied by sequences with greater than five identical codes
being 28% and 59%, respectively of the total code information).
Wide variation in the results from individual jobs is due
principally to the picture elements constituting the graph.
For example, many pictures include the drawing of axes and a
border, thus the smaller the picture, the greater the proportion
of the code sequence for these items. Picture A corresponds
to a total of 1738539 codes, whereas picture B corresponds
to a total of 74554 codes.

In order to obtain a more accurate estimate of the savings
achieved by compressing identical codes in sequence, a large
range of pictures were analysed. Out of 12308570 codes 38%
of the space occupied by these codes consisted of sequences
containing more than five identical codes. Replacement of
these by the repetition encoding yielded a saving of 26% in the
space occupied by the original codes.

Reduction of the space to hold the repetition count from,
say 18 bits to 12 bits in order to compress sequences with
greater than four identical codes, with the consequent
inability to cater for the longer but less frequent sequences
resulted in an additional saving of about 1%. In addition,
the extra central processor time needed to compress sequences
of five identical codes implies that such a representation offers
only marginal advantage, and would only be chosen where
store was at a premium.

An indication of the distribution in the repetition sequences
over the various codes was obtained from an analysis of
pictures A and B shown in Table 1; the major proportion of
the possible saving thus arises from the codes corresponding
to movements in X or Y directions only, particularly for the
longer sequences.

| Table 1 | Distribution of code sequences over the various codes |
| Movement code | All sequences (including 1) | Sequences ≥ 2 | Sequences ≥ 1000 |
| Job B | | |
| + Y | 12% | 58% | 17% |
| - Y | 12% | 58% | 17% |
| + X | 17% | 58% | 17% |
| - X | 17% | 58% | 17% |
| + X, + Y | 17% | 58% | 17% |
| - X, - Y | 2% | 1% | 0% |
| + X, - Y | 3% | 1% | 0% |
| - X, - Y | 3% | 1% | 0% |
| up | 2% | 1% | 0% |
| down | 5% | 2% | 0% |
| reset | 0% | 0% | 0% |
| reposition | 0% | 0% | 0% |

| Job A | |
| + Y | 16% | 83% |
| - Y | 18% | 83% |
| + X | 24% | 24% |
| - X | 25% | 24% |
| + X, + Y | 5% | 2% |
| - X, - Y | 7% | 4% |
| + X, - Y | 2% | 1% |
| up | 0.2% | 0.2% |
| down | 0.2% | 0.2% |
| reset | 0% | 0% |
| reposition | 0% | 0% |
The additional central processor time necessary to perform the compression was 25% of that previously required to generate the uncompressed sequence, and there was a small increase by the processing required to perform the regeneration before output to the device.

Although this enhancement catered for repeated codes in sequence, it does not cater for repeated code sequences (i.e. patterns) generated, for example, by tracking movements within areas between principal and diagonal axes, unless the particular pattern contains more than five identical codes in sequence, in which case some economisation is possible using the method previously described. In cases such as these, maximum compression can only be obtained by replacing the code in the representation previously discussed by the particular code sequence (i.e. pattern) which is to be repeated a given number of times.

3.2 Repeated code sequences

As for the Bresenham line tracking algorithm, attention may be initially concentrated on the first quadrant; other directions being effectively catered for by symmetry.

If the direction of the line to be drawn lies along the diagonal (i.e. the number of steps in the x direction \( N_x \) is equal to the number of steps in the y direction \( N_y \)) or along the axis (i.e. \( N_y = 0 \)), then the movement can be encoded directly using the representation described in the previous section.

If the line to be drawn from A to B (Fig. 5) involves a value of \( N_y \) of 1, then clearly the diagonal movement will be made at the mid-point (or within one increment) of the axial distance from A to B (say, \( N_x \) intervals).

That is, the sequence of movements will be:

\[
[(N_x - 1)/2] = N_1 \text{ x movements for } x, y \text{ (i.e. diagonal)}
\]

\[
N_x - (N_1 + 1) = N_2 \text{ x movements}
\]

Thus, if \((N_x - 1)/2\) is an integer, then \( N_1 = N_2 \),

\[
\text{if } (N_x - 1)/2 \text{ is not an integer, } |N_1 - N_2| = 1
\]

Hence, the sequence to be repeated could be based upon the \( x \) movement, this being repeated at least \([(N_x - 1)/2] \) times; and once again this can be encoded directly at the generation stage.

If \( N_x > 1 \) and \((N_x/N_y)\) is an integer, then this reduces to the above, except that the whole sequence is repeated \( N_x \) times. Thus, rather than representing repeated codes, greater savings can be achieved by repeating the pattern corresponding to \((N_x/N_y)\) intervals.

If \( N_x > 1 \) and \((N_y/N_x)\) is not an integer, then the pattern to be repeated needs to be calculated in more detail as it is not necessarily regular.

In this case, consider the point \( P \) in Fig. 6.

![Fig. 5 An example of line tracking in the first quadrant](image)

![Fig. 6 Location of repeated sequences](image)

\[
PR = n \times \frac{N_y}{N_x} \quad \text{where } n = \text{ the number of axial movements performed to reach } P.
\]

\[
PR = PQ + QR
\]

and \( QR = \) number of diagonal movements performed so far.

\[
\therefore \text{ an axial movement should be performed in a given interval if:}
\]

\[
n \times \frac{N_y}{N_p} \leq \text{ number of diagonal } + 0.5
\]

(5)

\[
\text{for } n = 1, 2, 3 \ldots N_x
\]

otherwise a diagonal movement is performed.

This is an alternative formulation for the algorithms given in (2) and (3) and can be thought of as distributing the necessary \( N_x \) diagonal movements evenly over the interval 0–\( N_x \). It may be expressed as an algorithm as follows:

1. \( \text{diagsofar} = 0; \)
2. \( \text{for } n = 1 \text{ step 1 until } N_x \text{ do} \)
3. \( \text{if } n \times N_y/N_x \leq \text{diagsofar} + 0.5 \)
4. \( \text{then step (axis)} \)
5. \( \text{else begin step (diag)}; \)
6. \( \text{diagsofar} = \text{diagsofar} + 1 \)
7. \( \text{end} \)

Testing this on the case \((0, 0) \text{ to } (10, 3)\), for which \( N_x/N_y = 3.3\), yields the following:

\[
n \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10
\]

\[
n \times N_y/N_x \quad 0.3 \quad 0.6 \quad 0.9 \quad 1.2 \quad 1.5 \quad 1.8 \quad 2.1 \quad 2.4 \quad 2.7 \quad 3.0
\]

\[
\text{diagsofar} + 0.5 \quad 0.5 \quad 0.5 \quad 1.5 \quad 1.5 \quad 1.5 \quad 2.5 \quad 2.5 \quad 2.5 \quad 3.5
\]

Movement \( a \quad d \quad a \quad d \quad a \quad d \quad a \quad d \quad a \quad d \quad a \quad d \quad a \)

Changing the condition to

\[
n \times N_y/N_x \leq \text{diagsofar} + 0.5
\]

yields the following sequence of movements

\[
a \quad d \quad a \quad d \quad a \quad d \quad a \quad d \quad a \quad d \quad a
\]

which is just as good and corresponds to the same sequence as generated by the Bresenham (2) and Thompson (3) algorithms. If \((N_x/N_y)\) is reduced to its prime factor decomposition where

\[
N_x = p_1^{i_1} p_2^{i_2} p_3^{i_3} \ldots p_n^{i_n} = \Pi p_i^{i_j}
\]

and

\[
N_y = p_1^{j_1} p_2^{j_2} p_3^{j_3} \ldots p_m^{j_m} = \Pi p_j^{j_k}
\]

where \( p_1, p_2, \ldots \) etc. are primes and \( i, j \) are non-negative integers, then \((N_x/N_y)\) can be reduced to its simplest form by obtaining the divisors of \( N_x \) and \( N_y \) which cancel out. The product of these divisors gives the number of times the basic pattern (given by the simplest form of \((N_x/N_y)\)) is to be repeated. This is the greatest common divisor, given by
which may be obtained without factoring $N_x$ and $N_y$ (which is a relatively slow process) by using either Euclid's method (Knuth, 1969; Dijkstra, 1976) or a more efficient variant (Stein, 1967) which does not use the division operation. The efficiency of this part of the algorithm is clearly crucial to the overall efficiency of the method; the larger the greatest common divisor, the greater the extent of the repetition.

As an example of the way in which a given sequence can repeat, consider the case of $N_y = 22$ and $N_x = 60$. This reduces to the pattern for the interval $N_x = 30$ ($N_y = 11$) which can be repeated again for the interval $N_x = 30$ to 60. Fig. 7 shows the line tracking sequence for the interval, $N_x$, 0 to 30. The sequence of movements is thus

$$\prod_{p} \min (i_p, j_p)$$

(where $a$ represents an axial movement and $d$ represents a diagonal movement) which (apart from the end) involves a repetition of $aad$ or $ad$, not unexpected in view of the (gradient)$^{-1}$ of the original line ($N_x/N_y = 2.727$).

To decide which movement (or sequence of movements) to take next in any given position, it was not possible to determine this (for a case such as the above) in terms of the patterns so far completed alone, although clearly it would have been advantageous to be able to do so. A calculation of the form of that in equation (5) must be performed.

To cater for compression of this information a different form of encoding to that used for repeated codes must be used. One possible form is shown in Fig. 8.

To investigate line generation by means of repeated patterns, numerous pictures (three of which are shown in Figs. 9, 10 and 11) using only straight lines were used. Figs. 9 and 11 contain a good selection of lines of different lengths and orientations.

The information corresponding to the straight lines was encoded using the two methods previously outlined. The encoding for repeated patterns was extended to include an additional six bits for the repetition count to use the same output routine as the encoding for repeated codes (and also to enable repeated codes to be encoded as a 'pattern' of one code). To determine the repeated patterns, an algorithm for determining the greatest common divisor of $N_x$ and $N_y$ was used. Both high and low level versions were derived, the latter being coded without the use of division or multiplication operations. Repeated identical codes (e.g. for lines in axial or diagonal directions) were catered for as a 'pattern' of one code, although in these cases the encoding was one character greater than that for the same line by encoding as a repeated code (no terminating symbol being used in the latter). Table 2 shows the three jobs run under the various systems; the time for the generation and encoding of the lines was added to that for the subsequent decoding (typically < 1 second) to give an overall figure corresponding to line generation. All increases or decreases are with respect to the standard system.
using the unmodified Bresenham algorithm (1965). It is clear that the encoding of repeated patterns gives even greater improvement in the compression of the total information for the graph than the encoding of repeated codes, and more importantly, gives a significant decrease (20–40%) in the overall central processor time needed to generate the lines for the graphs. The variation in this figure for the individual graphs examined (Table 2) is clearly due to the extent to which the lines constituting the graph correspond to repeated patterns. For these three graphs the mean compression for repeated codes was 40% and for repeated patterns was 54%. Although the compression for graph B was higher for repeated codes than for repeated patterns, this is not particularly significant. This graph used axial or diagonal codes almost exclusively for the lines that were drawn (Fig. 10), although this was not always the case for lines with the pen raised (i.e. movements to a new position on the diagram). Thus the difference in the length of the encoding accounts for the reduction in the savings by compression for the repeated pattern algorithm (the graph consisting largely of sequences of repeated codes rather than code patterns).

The lines in this graph are therefore perhaps not typical of the line orientations that will be encountered generally; this being more effectively demonstrated by Figs. 9 and 11. For these two graphs the mean compression for repeated codes was 23% and for repeated patterns was 49%, indicating the significant additional compression by encoding repeated patterns.

Combining the two encodings for even greater compression than that obtained for repeated patterns alone was considered as a further possibility. The presence of a repeated pattern does not necessarily imply that a sequence of greater than five consecutive identical codes cannot occur. The basic pattern appearing in the encoding of a repeated pattern may well contain such sequences. Although catering for repeated codes within this encoding (and also outside it, for those cases where no repetition of basic patterns occurs) involves additional central processor overheads of the same order as those for compression of repeated codes alone; the additional savings that would be made in code compression were investigated.

The information corresponding to the graphs in Figs. 9, 10, and 11 was encoded so that both repeated patterns and repeated codes were catered for. This gave additional savings of 11%, 31%, and 5%, respectively, in the space required for the total information, compared to that required for the encoding of repeated patterns only. This reduction therefore reflects the additional saving brought about by encoding repeated codes. The saving for Fig. 10 is perhaps atypical to a certain extent, for the reason previously outlined. Thus for the information corresponding to Figs. 9 and 11 the additional reductions are relatively marginal, and considerations about these marginal advantages must also include the additional central processor overheads (typically 17%; this being measured for the line routines only) that are required to perform the encoding of the repeated codes. Assuming that such an additional saving is not required (or the additional central processor overhead is not desired, and backing store is not at a premium), then the advantages of line tracking by compression of patterns (only) can be exploited to the full.

So as to investigate the representative nature of the graphs chosen (Table 2) further jobs containing straight lines (only) were run under these systems. Values of the same order as those for Figs. 9 and 11 were obtained for central processor time and file compression.

One further area examined was that of curve drawing.
Clearly, line generation by exploiting repeated patterns is applied at the line generation stage rather than the output stage (although this inhibits the straightforward implementation of drawing in broken format). For curves, no simple parallel formulation at the generation stage would appear to be possible, other than that similar to the implementation at the output stage (i.e. corresponding to encoding repeated codes for a graph containing curves).

The decision whether or not to cater for curves by compression depends strongly on the expected benefits, the latter being related to the proportion of code information corresponding to curves and also the extent to which such information can be compressed in general. The analysis of the code information for a graph containing curves (Fig. 12) showed that (excluding the drawing of the border) the compression of repeated codes (> 5 identical codes in sequence) gave a saving of 19% in the total space occupied by the code information (17894 codes). This corresponds principally to the drawing of the curves. A more detailed analysis of 6058552 codes corresponding to curve drawing information only, gave a saving of 17% by the compression of more than five repeated codes in sequence. The results of the analysis of the coding for Fig. 12 are therefore not unrepresentative of the general overall savings that can be made for curves.

It is possible to implement compression of curve drawing codes by duplication of the output routine; one copy then being used specifically for curves. The modifications for the compression of repeated codes can then be inserted into this output routine only, and will not apply to all other output which is routed through the usual (unchanged) output routine. In this way, the additional central processor overheads necessary for the compression will only apply to the time spent in generating curves rather than the time to generate the complete graph. As the former is 10% (on average) of the total central processor time spent in code generation, the overheads introduced thus apply only to this 10%. The overall increase in the central processor time for this modification is therefore of the order of 2%.

4. Conclusions

Line tracking by compression of repeated patterns gives a significant compression of the information required for the graph (~49%) for those cases using straight lines only, and more importantly, gives a significant decrease (~20–40%) in the overall central processor time needed to generate the lines.

These algorithms have been incorporated into a general purpose graphics system (Earnshaw, 1976).

The algorithms presented in this paper can readily be extended to cater for incremental plotters with more than eight vector modes.

5. Acknowledgements

The author would like to thank John Boothroyd and David Holdsworth for helpful discussion and the referee for constructive suggestions.

References


Book review


(Input Two-Nine, £6.50)

This book is ‘an introduction to system description in the programming language SIMULA’ according to its preface. It is not very clear to what audience the book is directed—the preface states that it is based on material developed from SIMULA courses given at the Norwegian Computing Centre. Clearly, it is of not much use on its own, but requires a context of a fairly solid course in software engineering.

SIMULA is rather a strange language: its origins are ALGOL 60 (indeed when the reviewer was participating in the revision of ALGOL 60, SIMULA was proposed as a potential standard). It has incorporated ideas from ‘ALGOL-W’, but placed side-by-side with languages such as ALGOL-68 and Pascal, it seems to be very incomplete. As its name implies, SIMULA is mainly used as a simulation language, but this book is not a book on SIMULA as a simulation language, but, again, its intention is unclear.

The would-be reviewer always faces the problem of trying to understand the way in which the book is presented to its putative audience; as a rule, the reviewer is not in this audience, and a certain amount of imagination has to be employed. The reviewer found it very difficult in this case: presuming himself to be in the situation of giving a course based on this book, he would have given up in despair! The contents of the book, taken as a whole, are very useful but the sequence is confusing, and at times, misleading. The reader is introduced to SIMULA concepts in an erratic way: odds and ends of SIMULA are thrown into the text in a completely unstructured way and the SIMULA class concept and co-routines only emerge late in the book.

However, the examples provided in Chapter 7 onwards are excellent and, indeed it might be a good idea to start reading the book at this point (page 209, more than halfway through). There are many exercises, with over 30 pages of solutions, though the commentary is highly variable.

The book is attractively presented with ample inner margins. The examples are both in rather broadly spaced printing with stop-words in italics (difficult to read) and computer printed output (with rather erratic tabulating conventions).

R. M. De Morgan (Reading)