Finding Straight Lines and Curves in Engineering Line Drawings

J. Patrick Bixler
Layne T. Watson
J. Patrick Sanford

TR 87-12

FINDING STRAIGHT LINES AND CURVES IN ENGINEERING LINE DRAWINGS

J. Patrick Bixler
Layne T. Watson
J. Patrick Sanford

Department of Computer Science
Virginia Polytechnic Institute & State University
Blacksburg, VA 24061

Abstract. This paper addresses the problem of distinguishing straight lines from curves in noisy gray tone images, and mathematically representing those lines and curves. A method for locating corners is discussed as well as criteria, based on spline representations, for classifying line segments as straight or curved. Results are given for several typical noisy engineering line drawings.

Key Words and Phrases: corner detection, curve recognition, engineering line drawing,

This research was supported by Control Data Corporation Grant 84V101 and by the Center for Innovative Technology Grant INF-85-004.

INTRODUCTION

This paper focuses on the problem of distinguishing straight lines from curves in noisy, gray–tone images of engineering line drawings. The underlying motivation is to develop a fully automatic system that could take a digitized gray–tone image of a line drawing, recognize and extract the lines and shaded regions, and compress the data based on a high–level, mathematical representation. The substantial volume of existing hard–copy drawings would benefit from such a system which would facilitate storage, transmission, and review and modification.

We are not necessarily concerned with the best method of generating arbitrary lines and curves, although many of the same approximation techniques apply. The emphasis here is, first, on accurately locating corners in line drawings, and, second, on developing an effective test for when a set of real data constitutes a straight line. If the data is not a straight line we give a good higher order approximation. We do not attempt to find an optimal approximation satisfying some prescribed fitting criterion. The quality of the fit is judged subjectively based on a comparison of the reconstructed drawing and the original.

The proposed algorithm first uses a simple technique to locate general changes of direction (corner points) in the data, and then determines if each segment between successive corner points is best represented by a straight line or a curve. Two least squares spline approximations are generated, one linear and one cubic, to fit each segment. The quality of these fits then determines which primitive is used to represent that portion of the data. If the linear fit is sufficiently good, the data is represented by a straight line, otherwise, the data is represented by the cubic spline. This method produces good qualitative results, and generates efficient mathematical representations for the primitives.

There is a substantial volume of literature devoted to the topic of fitting scattered data with piecewise polynomial functions and several are listed in the references. Many existing algorithms, however, tend toward the extremes of optimizing efficiency at the expense of quality or allowing substantial processing time to obtain a very close fit. Some algorithms are content to provide a polygonal approximation to all curves and for applications in which most of the components are straight lines, they work well. For images in which many of the components are highly curving, these

techniques are simply not appropriate. Still other algorithms are intended as interactive design tools and require some user intervention.

Our efforts are motivated by the need for an efficient means of archiving typical engineering line drawings. Thus, we have attempted to strike a balance between fitting quality, speed, and final storage requirements. We have also insisted that, beyond specifying a few global parameters, there be no user interaction.

To produce input for this study, any good line finding or contour following algorithm would suffice. It is only necessary that the input consist of a binary image in which each curve has a thickness of one pixel and is represented as a sequence of adjacent pixels. The particular preprocessing applied in this case is as described in [1] and [2]. The data is then in a form that can easily be approximated by a variety of mathematical primitives. See Figure 1.

*** FIGURE 1a and 1b GO HERE ***

FINDING CORNERS

Many of the existing methods for fitting curves begin by first finding a polygonal approximation and then using the vertices of the polygon to help determine the approximating polynomial [3,4]. Sometimes the vertices are dynamically adjusted to better locate corners and to improve the fit. These vertices are then used as breakpoints for a spline approximation. It is possible to determine the placement of the breakpoints that produces an optimal fit, but this is computationally very expensive and the results are usually not worth the effort. That is, a simpler suboptimal placement of the breakpoints can produce an approximation that accurately captures the shape of the underlying data [5]. The real value of the initial polygonal approximation is that it should accurately locate sharp corners in the data, that is, places where two different curves meet. This then allows one to fit each segment between successive corners with a smooth curve requiring only continuity between adjacent segments.

There are numerous techniques for constructing polygonal approximations to arbitrary line data [6,7,8,9]. Two recent algorithms also relevant to our study are reported in [10] and [11]. Both of these are fast, scan-along techniques that essentially try to add data points to the current line until some error threshold is exceeded. In [10], Sklansky uses itersecting cones to to find the longest valid

approximating segment. Circles are first drawn about each data point, then, proceeding from point to point, tangents are drawn to the circles to define sectors in which valid segments must lie. The procedure finds a polygonal approximation to a set of input data points such that the Hausdorff-Euclidean distance between the two is no more than some prespecified ϵ. If ϵ is large, curves can be separated into long straight segments introducing extra vertices in the approximating polygon. A smaller ϵ, on the other hand, can make the algorithm more sensitive to noise which again introduces extra vertices. Additionally, while proceeding around a corner, the algorithm often adds extra data points to the current line segment before terminating. This causes a mislocation of the true corner and results in squares or rectangles appearing as parallelograms or trapezoids. See Figures 2a and 3a.

**** FIGURES 2a AND 2b GO HERE ****

Wall's method [11], also a scan-along polygonal approximation technique, is based on computing an area deviation for each line segment. The longest allowable line segment is determined by merging points, one after another, until the area deviation per unit length of the segment exceeds a prespecified value T. The coordinate system is first translated so that the origin is at the starting data point (x_0, y_0). Successive data points (x_i, y_i), $i = 1, 2, \ldots$, are added and the increments $\triangle x_i = x_i - x_{i-1}$ and $\triangle y_i = y_i - y_{i-1}$ are computed. The accumulated deviation, f_i, is then calculated by:

$$f_i = f_{i-1} + \triangle f_i \quad \text{where} \quad f_0 = 0, \quad \triangle f_i = x_i \times \triangle y_i - y_i \times \triangle x_i.$$

The length L_i of the current line segment from the starting point to (x_i, y_i) is calculated, where

$$L_i = (x_i^2 + y_i^2)^{1/2},$$

and the following test is applied:

$$|f_i| \leq (T \times L_i)\,.$$

If the test is satisfied, the next data point is added and the deviation is recalculated. Otherwise the longest allowable segment has been found, i.e. the segment joining (x_0, y_0) to (x_{i-1}, y_{i-1}). The latter point is taken as the starting point for the next segment and the process is repeated.

The results for various values of T are quite similar to those produced by the Sklansky algorithm. Large values of T fragment the curves and small values of T fragment long straight lines. In some cases the corner degradation is so noticeable that small squares and rectangles are hardly recognizable (see Figure 3b).

**** FIGURES 3a AND 3b GO HERE ****

Another popular method is to first fit the data with a cubic spline and then compute the local extrema of curvature. If curvature is high enough, call the nearest data point a corner point. This method is not only expensive, but can lead to rather unsatisfacory results. Because of noise in the image, the line tracking algorithms do not always produce a "smooth" line. The tracker tends to undulate back and forth as it follows the gray tone intensity ridge, resulting in a sometimes jagged line. These fluctuations in direction appear as local extrema of curvature, and, hence, many spurious corner points may be found.

The goal here is to locate general changes in directions while ignoring the local fluctuations due to noise. A simple technique that does work well is to pass a window over the data and approximate each half of the window by a least squares linear spline. The two linear approximations form an angle at the center of the window. When this angle exceeds some threshold, a potential corner point is found. The exact details of the algorithm are given below.

As usual, parameterized row and column coordinates of an arbitrary curve are considered separately. The independent variable in both cases is the cumulative linear distance between successive data points, where pixel separation is taken to be one unit. Breakpoints and simple knots are placed at the window endpoints and at the center. Here splines are computed as linear combinations of B-splines which implies a distinction between "breakpoint" and "knot" [5]. If the angle made by the resulting linear approximation is less (sharper) than some threshold, the center is recorded as a potential corner point, the window is moved ahead by one, and the process is repeated. Notice that the linear distance between two successive data points is either 1 or $\sqrt{2}$, whereas the change in the corresponding row or column coordinates is at most 1. Since the separate row and column splines are combined to form a single approximation, this discrepancy can result in a skewed fit if the cumulative linear distance is used as the independent variable. The problem is avoided by using a uniform increment of 1 as the independent variable.

A less expensive and slightly less robust method for locating corners is simply to join the data points at the ends of the window to the data point at the center. Although this may do a poor job of approximating the actual data, it usually still detects the sharp corners. Unfortunately, each method detects some corners that the other method does not. The most effective technique seems to be a combination of these two methods. First, set a higher angle threshold (less sharp) and make one pass over the data joining the window endpoints to the center. This detects all points that have a chance of being corners. Then compute the more accurate spline approximation in a small neighborhood of these points and test the angle against the sharper threshold. This hybrid method accurately finds corner points while eliminating the expensive spline computations in cases of very low curvature.

As expected, varying the window size and the angle thresholds produces quite different results. Smaller windows are too sensitive to local noise, resulting in too many corner points and misplacement of some corner points. Large windows, although less sensitive to noise, create general inconsistencies when dealing with shorter line segments, such as those in block letters, small polygonal symbols, etc. If the sides of the figure in the original image are less than half the window size, the approximations can result in recording invalid corner points or in losing valid corner points. As indicated below, very short curves are considered as special cases. In general, a window of size 11 produced the best results for the test data. See Figure 2b.

The angle threshold is intuitively set at 135 degrees. This locates those points at which a line makes more than a 45 degree deviation from its current heading. It also, appropriately, ignores a more gently curving line. Because of the large value for the angle threshold, however, data points surrounding the actual corner may also pass the threshold, thus yielding a string of adjacent corner points. In this case a thinning process is applied: when two corner points are closer than three data points apart, the corner with the sharper angle is retained and the other one is deleted. This ensures that each segment from corner to corner consists of at least five data points.

DISTINGUISHING STRAIGHT LINES FROM CURVES

Having identified the sharp corners in the data, any of a number of methods for fitting the idividual segments could be used. We again take a simple approach. First, generate a cubic spline with simple knots located at equally spaced breakpoints, and breakpoints with triple knots at each of

the corner points. Note that a simple knot ensures that the approximation will be twice continuously differentiable at the corresponding breakpoint, whereas a triple knot guarantees only continuity [5]. Thus, the requirement for smoothness at the corner points is eliminated. At the same time, a linear spline is computed with a breakpoint and simple knot at each corner point.

Errors are then computed to decide whether the data should be represented as a straight line or as a cubic spline. Between successive corner points calculate the RMS and discrete max norm errors for the linear spline, and the RMS error for the cubic spline. If the linear RMS and discrete max norm are below fixed thresholds, the linear fit is acceptable and that interval is taken to be a straight line segment. If either of these errors is greater than its respective threshold, but the cubic RMS error is not significantly better than the linear RMS error, the interval is again classified as a straight line segment. The point is that one might just as well take advantage of the simplicity of the linear representation in the case that a cubic representation is not significantly more accurate. Finally, if both of these tests fail, the segment is taken to be a curve and is represented by the cubic spline approximation.

To clarify the placement of breakpoints and their relationship to the knots, note that the "standard" breakpoints are placed at regular intervals, every 5 units, until a "corner" breakpoint is reached. No standard breakpoint should be placed within 5 units of a corner point, to prevent the data from being simply interpolated. Then, starting with the corner breakpoint, standard breakpoints are again located every 5 units until another corner breakpoint is reached, and so on. Simple knots are located at each standard breakpoint and triple knots at each corner breakpoint. Figure 4 shows the location of the breakpoints and knots for a sample line.

*** FIGURE 4 GOES HERE ***

Because of image noise, it is reasonable to judge shorter line segments differently than longer ones. Very short segments, say, less than five data points, are too short to be approximated by a cubic spline and, hence, they are simply represented by their original data points. On the other hand, as the line segments increase in length, the errors associated with the approximations also increase. For this reason, a bi-level error threshold would seem appropriate. A shorter line, say 5 to 15 data points, would be required to pass a more stringent set of constraints to be classified as a straight line, whereas a longer line would be allowed a bit more error. Most of these shorter lines, however,

are located between valid curved segments. This results in a fragmentation of the curves, which has a negative effect on the appearance of the approximation. For example, the fragmentation is especially noticeable in small polygonal figures, alphanumeric characters, etc. To avoid this problem, all lines of length 5 to 15 are represented by their cubic spline approximations, regardless of their actual shape. For lines longer than 15 points, the classification is determined as above. The precise algorithm follows.

`num_lines` is the number of input lines

FOR `i` := 1, `num_lines` **DO**

`length` is the number of data points in the ith line

`x_coord` is the array of column coordinates for the ith line

`y_coord` is the array of row coordinates for the ith line

STEP 1: Find the corner points

1(a): Find the candidate corner points

1(b): Thin the candidate corner points

STEP 2: Fit the data

`cub_x`(s) is the best cubic spline approximation to the data

(`j`,`x_coord`$_j$), $0 \leq$ `j` $\leq$ `length` $- 1$,with simple and multiple knots as explained above.

`cub_y`(s) is the best cubic spline approximation to the data

(`j`,`y_coord`$_j$), $0 \leq$ `j` $\leq$ `length` $- 1$,with simple and multiple knots as explained above.

`interv` is the number of intervals in the above cubic spline approximations

STEP 3: Separate the lines and curves

FOR `j` := 1, `interv` **DO**

`size` is the number of data points in the jth interval

`lin_x`(s) is the best linear spline approximation to the data

(`k`, `x_coord`$_{\tilde{k}}$), $0 \leq$ `k` $\leq$ `size` $- 1$, where $\tilde{k}$ indexes the points in the jth

interval, with a breakpoint and simple knot placed at the endpoints of the interval

lin_y(s) is the best linear spline approximation to the data (k,y_coord$_{\tilde{k}}$), $0 \le$ k $\le$ size $- 1$, where $\tilde{k}$ indexes the points in the jth interval, with a breakpoint and simple knot placed at the endpoints of the interval

FOR k := 0, size $- 1$ **DO**

$$\texttt{lin\_err}_k := \sqrt{\left(\texttt{lin\_x(k)} - \texttt{x\_coord}_{\tilde{k}}\right)^2 + \left(\texttt{lin\_y(k)} - \texttt{y\_coord}_{\tilde{k}}\right)^2}$$

$$\texttt{cub\_err}_k := \sqrt{\left(\texttt{cub\_x}(\tilde{k}) - \texttt{x\_coord}_{\tilde{k}}\right)^2 + \left(\texttt{cub\_y}(\tilde{k}) - \texttt{y\_coord}_{\tilde{k}}\right)^2}$$

END FOR

$$\texttt{lin\_rms} := \sqrt{\left(\sum_{k=0}^{size-1} \texttt{lin\_err}_k^2\right) / \texttt{size}}$$

$$\texttt{cub\_rms} := \sqrt{\left(\sum_{k=0}^{size-1} \texttt{cub\_err}_k^2\right) / \texttt{size}}$$

$$\texttt{lin\_norm} := \max_{0 \le k \le size-1} |\texttt{lin\_err}_k|$$

$$\texttt{diff} := |\texttt{cub\_rms} - \texttt{lin\_rms}|$$

$$\texttt{lin\_factor} := \texttt{lin\_rms} \times 0.15$$

IF (size ≥ 5) **THEN**

IF (size ≥ 16) **THEN**

IF (((lin_norm ≤ 2.05) **AND** (lin_rms ≤ 0.75))

OR (diff $\le$ lin_factor)) **THEN**

represent the data as a straight line

ELSE

represent the data as a curve

END IF

ELSE

represent the data as a curve

END IF

ELSE

represent the data with the original data points

END IF

END FOR loop on intervals within a line

END FOR loop on lines

If the data within an interval is classified as a straight line segment, the endpoints of the linear approximation are stored to represent the data. Intervals that are not classified as straight lines,but that are longer than 4 data points, are represented by the cubic spline.

EXPERIMENTAL RESULTS

A sample line drawing used as original input to the entire system is shown in Figure 1. The results of applying the line finding algorithm mentioned in the introduction are shown in Figure 3. This serves as the input to the current algorithm for distinguishing straight lines from curves. A portion of this image, after classifying segments as straight or curved, is shown in Figure 5. There are still some lines that appear straight but were classified as curves by the algorithm. Sometimes this is because there is a short hook or wiggle near the end of the line, not sharp enough to be considered a corner but pronounced enough to exceed the linear error threshold. It may also be because the segment is joined to another straight segment forming an angle of less than 45 degrees. This can be controlled to some extent by adjusting the angle threshold. A third possibility is the case when a straight segment is tangent and connected to a curved segment, as in the side and bottom edges of the cylinder near the top of Figure 5.

Results for two other test images are shown in Figures 6-7. The drawing of Figure 6 consists primarily of straight lines and points out the problems caused by a poorly executed drawing. Although it is obvious to the human observer just which lines are supposed to be straight, a closer look shows that some of the lines have rather noticeable variations in them. Also, some of the corners of the boxes in the keypad area are not well defined. Some of this may be due to the effects of

digitization, perhaps the lines were not drawn uniformly thick or uniformly dense, but some of it is also simply due to the quality of the drawing. The algorithm does a fairly good job of identifying most of the straight lines, but misses some others. The algorithm is reporting, perhaps a bit too faithfully, what is actually in the image, rather that what we see or what we think we see. Such an image might also benefit from from an adaptive thresholding scheme, i.e. relaxing the error criteria for the linear approximations based on the length of the segment.

The final test image (Figure 7) contains mostly straight lines, but also contains substantial noise. In addition to several extraneous marks that are long enough to show up as curves, some of the long (apparently) straight lines are actually broken in places where the original drawing had been folded. Distortion in the vicinity of these creases have caused many of the lines to be represented as curves. Again, adjusting the linear error threshold might help. There are also some problems near the intersections of some of the lines. This may be due to a small blob of ink that gets deposited at the end of a straight line and causes the line tracker to veer off slightly.

All algorithms were implemented and tested on a VAX 11/785 running UNIX. Sample execution times range from 1 to 3 minutes depending on the complexity of the drawing, see Table 1. An improvement of about 20% in performance should be possible by carefully optimizing the code. One possibility for improving the algorithm itself is to make a gross estimate of straightness by first approximating a segment with the line joining its end points. If the error is fairly high, simply go right to the cubic approximation. If it is not, proceed with the algorithm as stated above.

Actual storage requirements, assuming single precision values are used, range from about 18K bytes to 44K bytes for the images tested, see Table 1. Since the standard breakpoints are placed at regular intervals offset from the triple knots, only the sequence of triple knots must be stored for each of the cubic splines. For each segment between corner points storage requirements consist of 4 coefficients for the first cubic and then, because of the continuity conditions at the simple knots, only one additional coefficient for each simple knot. This implies that the actual coefficients must be derived recursively in order to recreate the image from its stored format. The number of polynomials needed is inversely related to the number of data points between knots. We have initially chosen to maintain at least 5 data points between knots to insure that the data is not simply interpolated. Placing the knots at wider intervals would reduce the number of cubics needed and, hence, the

amount of storage, but at the expense of increased smoothing. The straight lines are, of course, stored by their endpoints.

CONCLUSIONS

The two contributions of this work are a means of accurately locating corners in line drawings, and a method of distiguishing straight line segments from curved ones. No attempt has been made to repair or enhance poorly exectued drawings or noisy images, although the algorithms have been designed to tolerate some noise. The algorithms work well on arbitrary line data, but can be somewhat tuned to a given set of drawings. When coupled with a front–end system for extracting line data from real drawings, these techniques form a complete system for archiving existing engineering drawings.

Locally, a change in direction of 10 degrees is not significant, and *should* not be, but the algorithm here would represent two long straight lines meeting at such a 170 degree angle as a cubic spline. A long straight line with a short hook at the end, possibly caused when the ink pen is raised from the paper, is also represented as a cubic spline, because the hook makes the max norm error large. To faithfully reproduce a person's ability to classify straight lines and curves apparently requires a much more sophisticated algorithm, but the basic approach here seems sound.

A possible enhancement to these technique presented here would involve a tier of error thresholds based upon the length of the segment being tested. This would take into account the fact that the RMS and discrete max norm errors generally increase with respect to line length. Other potential refinements to the present algorithm include optimal knot placement and special rules embodying knowledge about engineering line drawings.

REFERENCES

1 **Arvind, K** 'Extraction of lines and regions from grey tone line drawing images' MS Thesis, Computer Science Department, Virginia Polytechnic Institute and State University, Blacksburg, VA, February, 1984.

2 **Bixler, J P and J P Sanford** 'A technique for encoding lines and regions in engineering drawings' Pattern Recognition, 18 (1985) pp 367–377.

3 **Pavlidis, T** 'Curve fitting with conic splines' ACM Trans. on Graphics., 2 (1983) pp 1–31.

4 **Plass, M and M Stone** 'Curve-fitting with piecewise parametric cubics' Xerox PARC Tech. Rep., Xerox Palo Alto Research Center, Palo Alto, Calif. Proc. SIGGRAPH 83 (Detroit, July 1983 25-29), ACM, New York.

5 **de Boor, C** *A practical guide to splines* Springer-Verlag, New York, 1978.

6 **Pavlidis, T** *Algorithms for graphics and image processing* Computer Science Press, Rockville, Md., 1982.

7 **Pavlidis, T** 'Curve fitting as a pattern recognition problem' Proc. 6th Int. Conf. Pattern Recognition (Munich, Oct. 1982), IEEE Computer Society Press, Silver Spring, Md., pp 853–859.

8 **Ramer, U E** 'An iterative procedure for the polygonal approximation of plane curves' Comput. Graphics Image Process. 1 (1972) pp 244–256.

9 **Tomek, I** 'Piecewise linear approximations' IEEE Trans. Computers, C-23 (1974) pp 445–448.

10 **Sklansky, J and V M Gonzalez** 'Fast polygonal approximation of digitized curves' Pattern Recognition, 12 (1980) pp 327–331.

11 **Wall, K and P Danielsson** 'A fast sequential method for polygonal approximation of digitized curves' Comput. Vision Gr. Image Process., 28 (1984) pp 220-227.

12 **Albano, A** 'Representation of digitized contours in terms of conic arcs and straight–line segments' Comput. Gr. Image Process. 3 (1974) 23–33.

13 **Bookstein, F L** 'Fitting conic sections to scattered data' Comput. Gr. Image Process., 9 (1979) pp 56–71.

14 **Clement, T P** 'The extraction of line-structured data from engineering drawings' Pattern Recognition, 14 (1981) pp 43–52 .

15 **Cox, M G** 'Curve fitting with piecewise polynomials' J. Inst. Math. Applications. 8 (1971) pp 36–52.

16 **Dierckx, P** 'Algorithms for smoothing data with periodic and parametric splines' Comput. Gr. Image Process., 20 (1982) pp 171–184.

17 **Ichida, K and F Yoshimoto** 'Curve fitting by a one-pass Method with a piecewise cubic polynomial' ACM Trans. on Math. Soft., 3 (1977) pp 164–177.

18 **Liao, Y Z** 'A two-stage method of fitting conic arcs and straight line segments to digitized contours' Proc. IEEE Pattern Recognition and Image Processing Conference. (Dallas, 1981) pp 224–229.

19 **Lozover, O and K Preiss** 'Automatic generation of a cubic B-spline representation for a general digitized curve' Eurographics 81, J.L. Encarnacao, Ed., Elsevier North-Holland, New York, 1981 pp 119–126.

20 **Pavlidis, T** 'Optimal piecewise polynomial L_2 approximation of functions of one and two variables' IEEE Trans. Comput., 24 (1975) pp 98–102.

21 **McClure, D E** 'Nonlinear segmented function approximation and analysis of line patterns' Q. Appl. Math., 33 (1975) pp 1–37.

22 **Ramachandran, K** 'A coding method for vector representation of engineering drawings' Proc. IEEE, 68 (1980) pp 813–817.

23 **Shampine, L F and M K Gordon** *Computer solution of ordinary differential equations: The initial value problem* W. H. Freeman, San Fransisco, 1975.

24 **Suetens, P and P Dierckx, R Piessens, A Oosterlinck** 'A semiautomatic digitization method and the use of spline functions in processing line drawings' Comput. Graphics Image Process., 15 (1981) pp 390–400.

25 **Watson, L T and K Arvind, R W Ehrich, R M Haralick** 'Extraction of lines and regions from grey tone line drawing images' Pattern Recognition, 17 (1984) pp 493–507.

26 **Yamaguchi, F** 'A new curve fitting method using a CRT computer display' Comput. Gr. Image Process., 7 (1978) pp 425-437.

Table 1. Results for sample drawings

	data pts	curved lines	straight lines	storage
coil	25091	392	144	44219
keybd	17837	279	240	18562
circuit	17531	397	241	24038

FIGURE CAPTIONS

Fig. 1a. Photograph of the original coil line drawing.

Fig. 1b. Output of line tracker applied to Figure 1a.

Fig. 2a. Mislocation of corner points using scan-along techniques.

Fig. 2b. Corner detection using window method.

Fig. 3a. Results from the Sklansky [21] algorithm with $\epsilon = 0.5$, for a portion of Figure 6a.

Fig. 3b. Results from the Wall [24] algorithm with $T = 2.0$, for a portion of Figure 7a.

Fig. 4. Sample line showing the location of the breakpoints and knots for the cubic spline approximation, where $\bullet$ is a data point, $*$ is a triple knot, and $+$ is a simple knot.

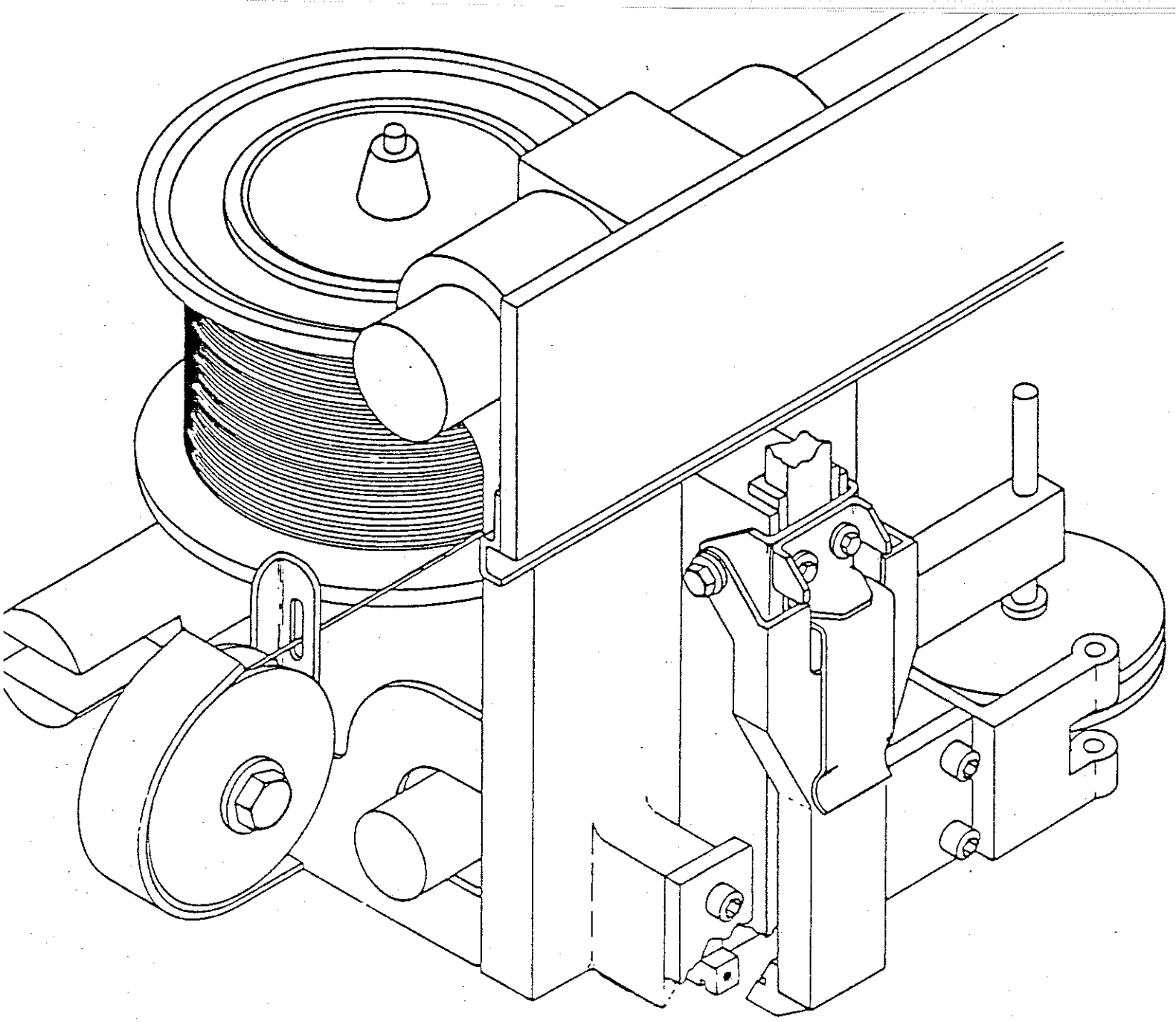
Fig. 5. A portion of Figure 1 following the line classification algorithm (straight lines are dotted and curves are solid).

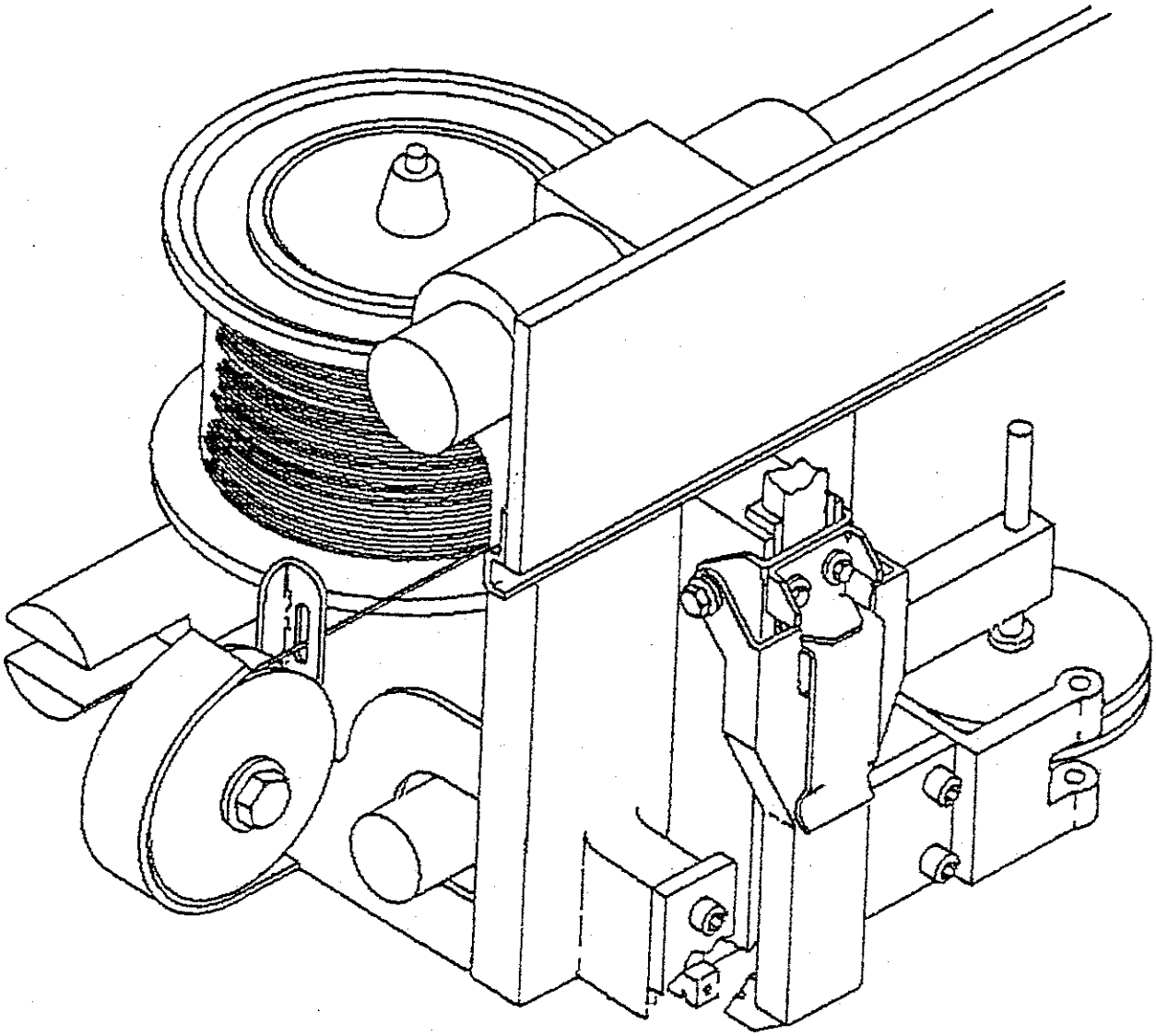
Fig. 6a. Original keyboard drawing after line tracking.

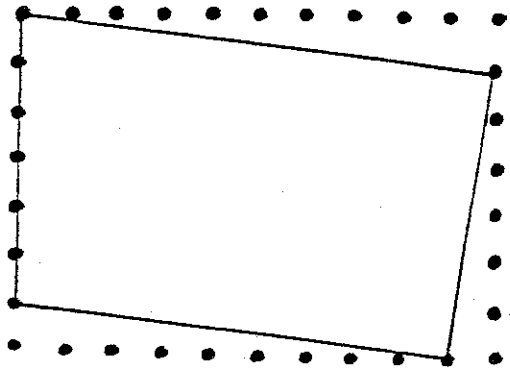
Fig. 6b. A portion of Figure 6a following the line classification algorithm (straight lines are dotted and curves are solid).

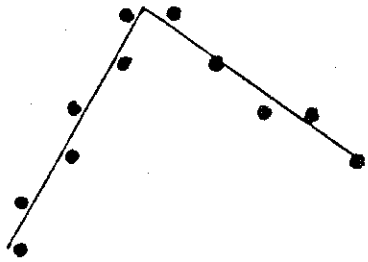
Fig. 7a. Original schematic drawing after the line tracking.

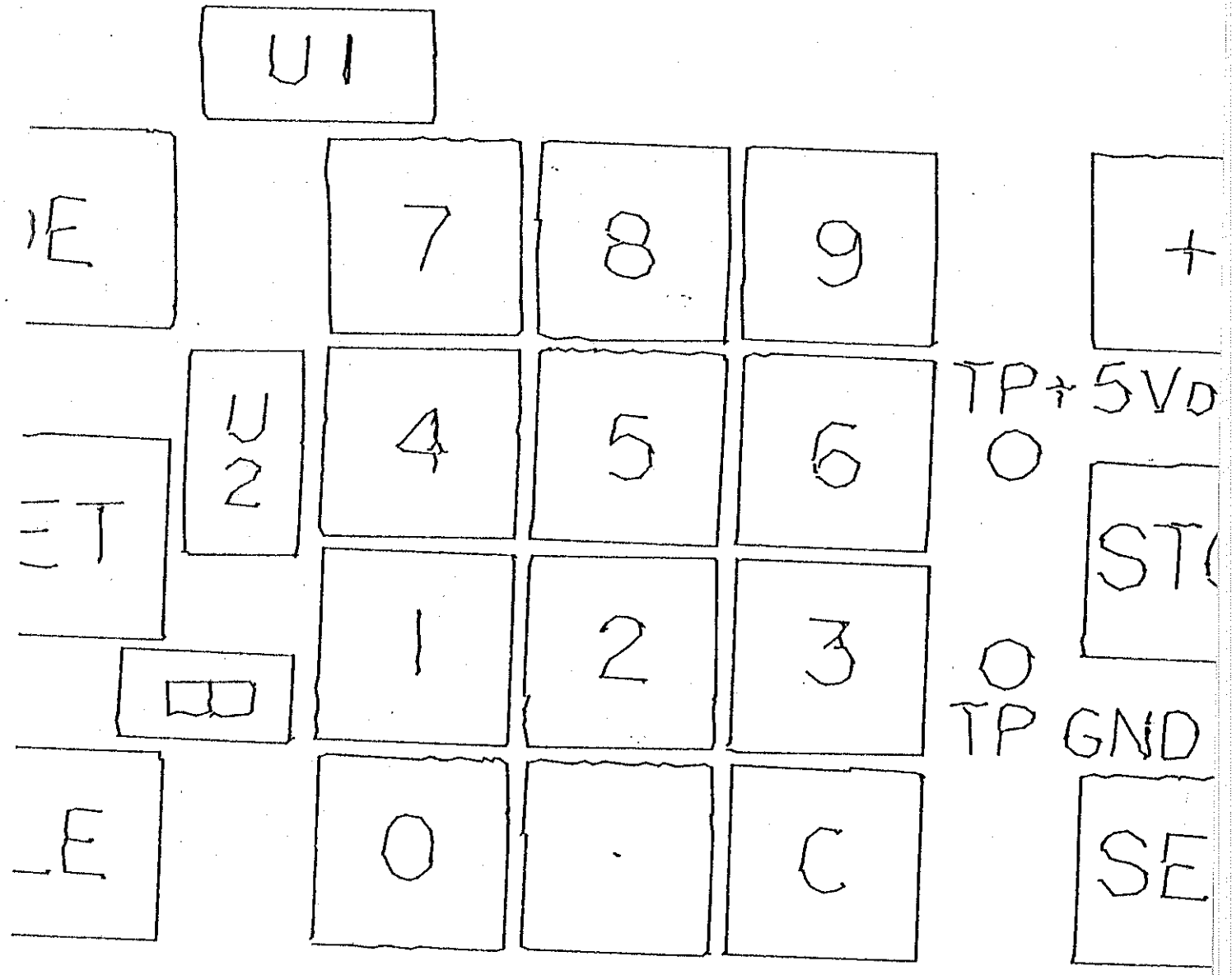
Fig. 7b. A portion of Figure 7a following the line classification algorithm (straight lines are dotted and curves are solid).

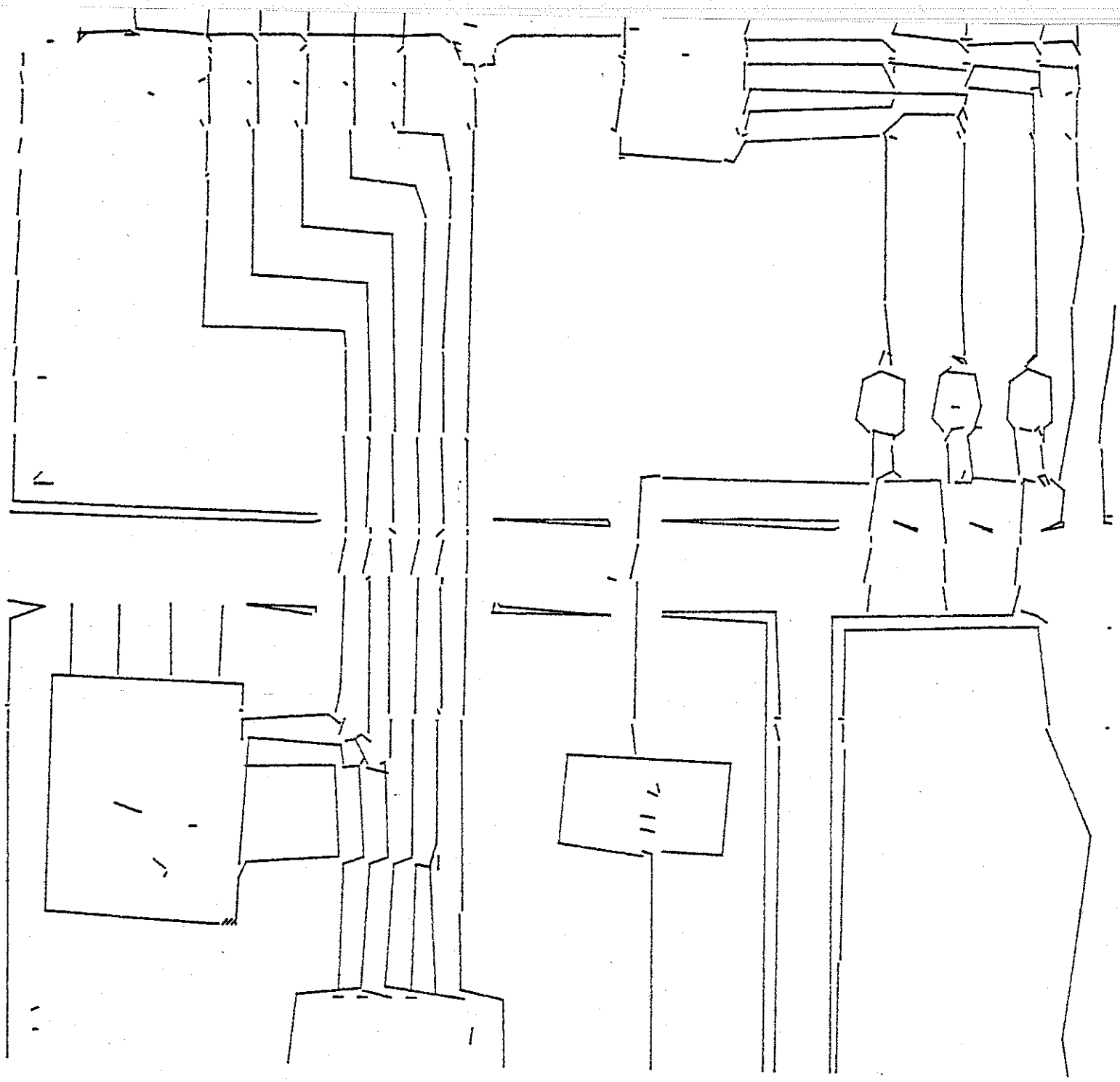


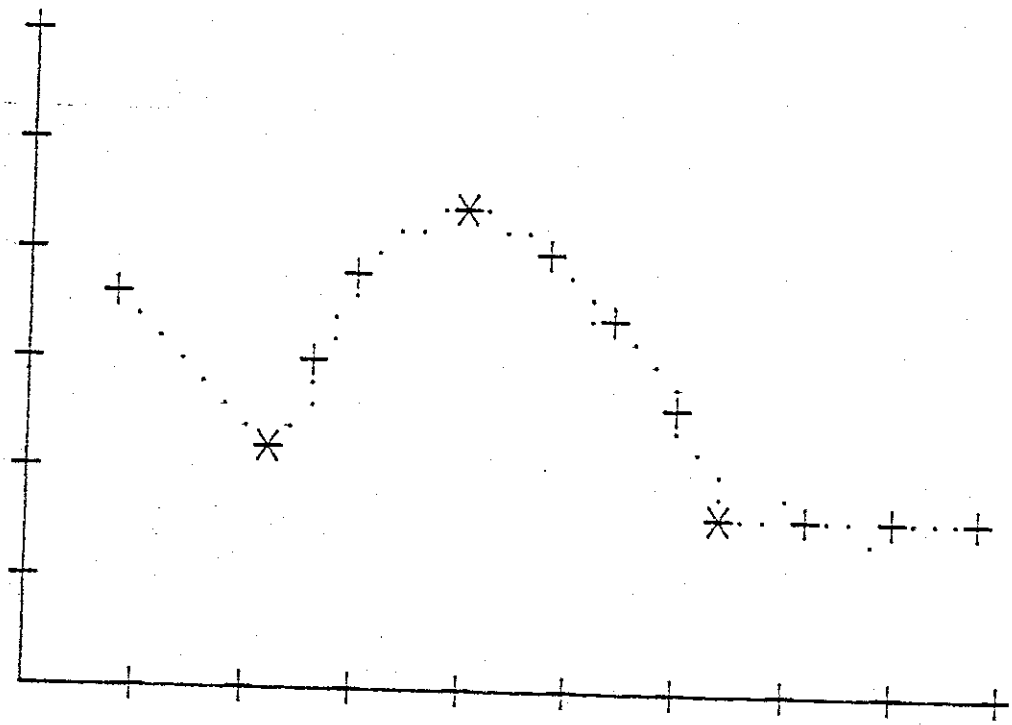


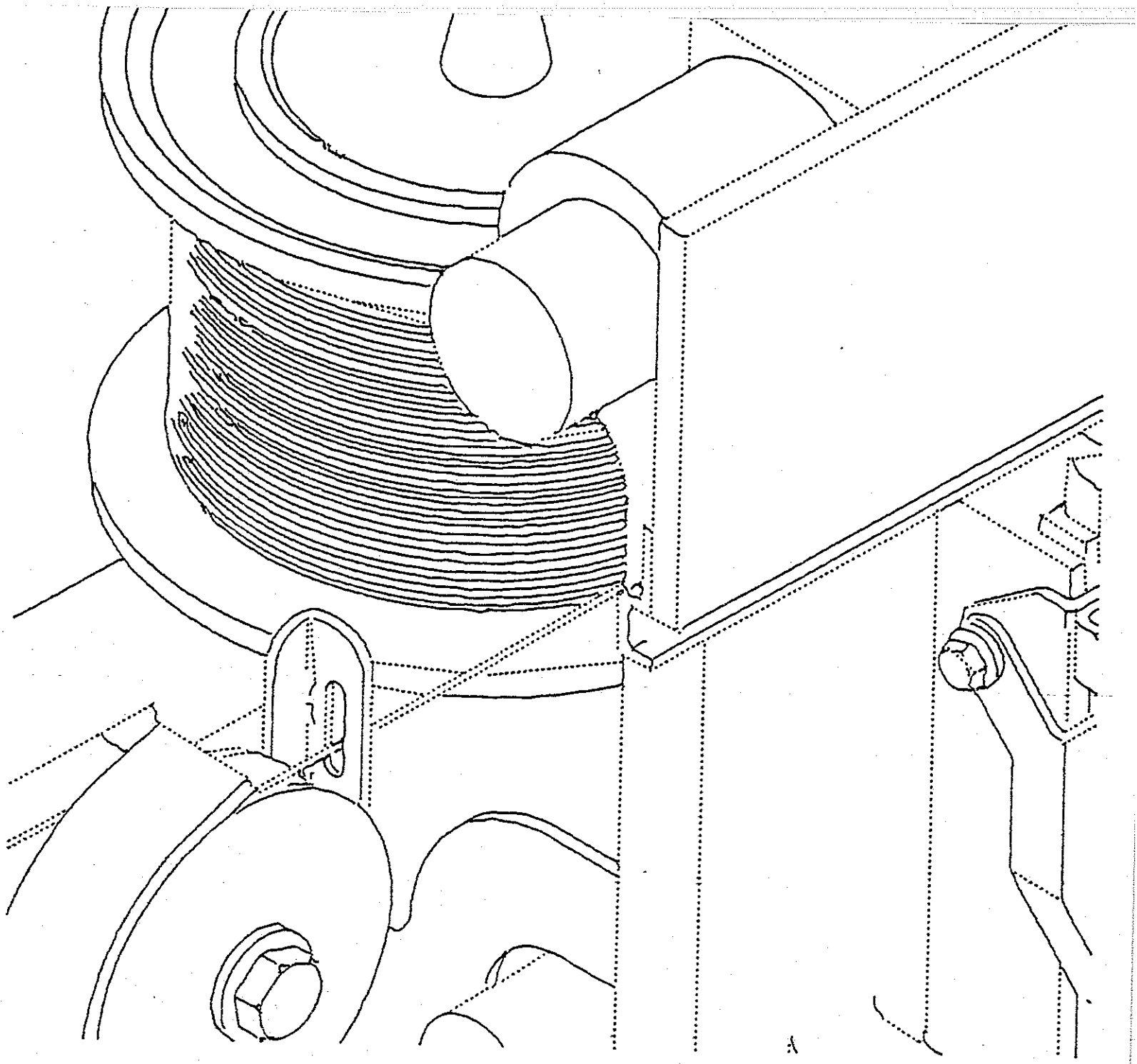


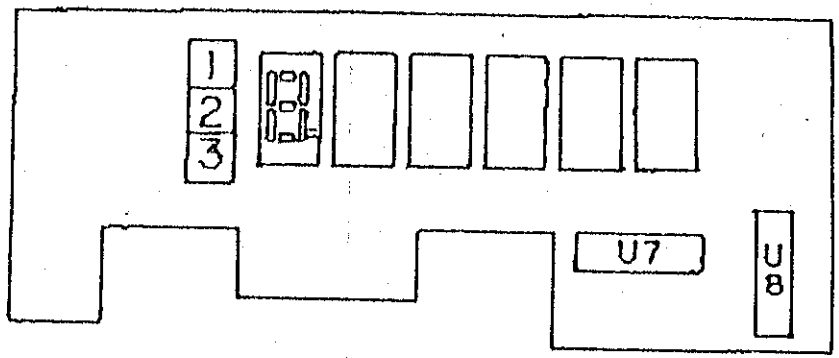


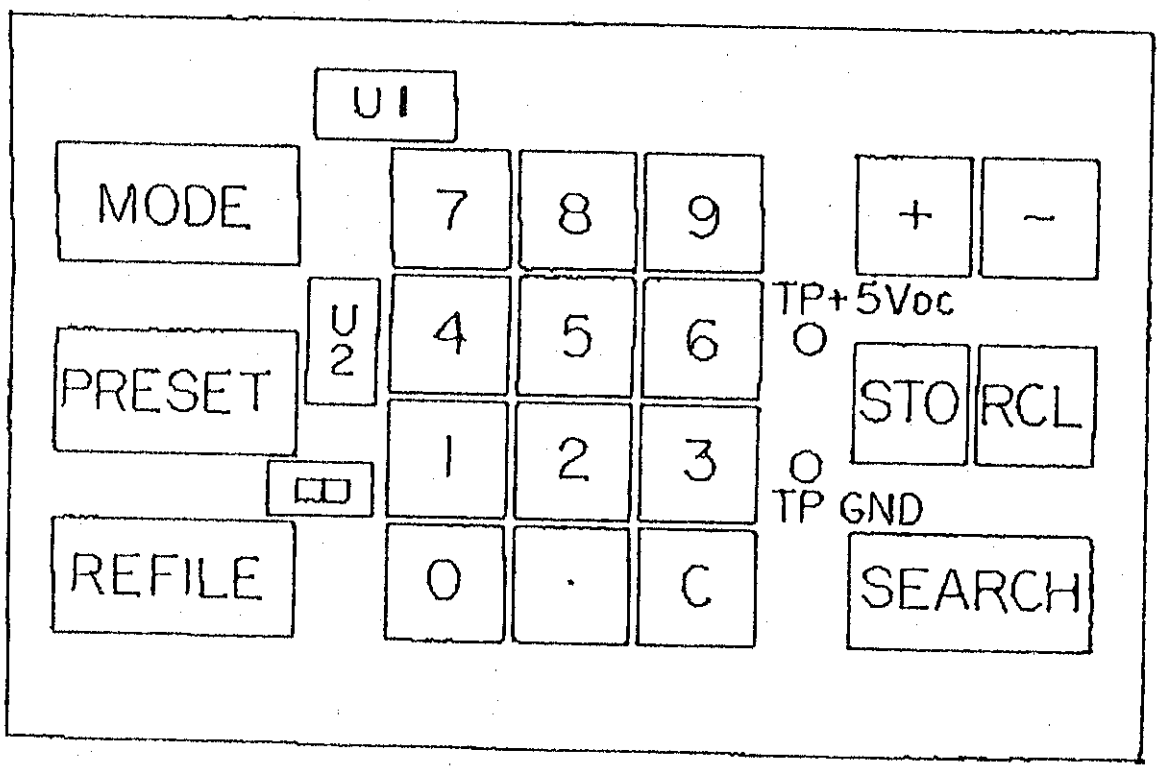

U1
7
8
9
+
TP+5VD
U2
4
5
6
1
2
3
TP GND
0
.
C

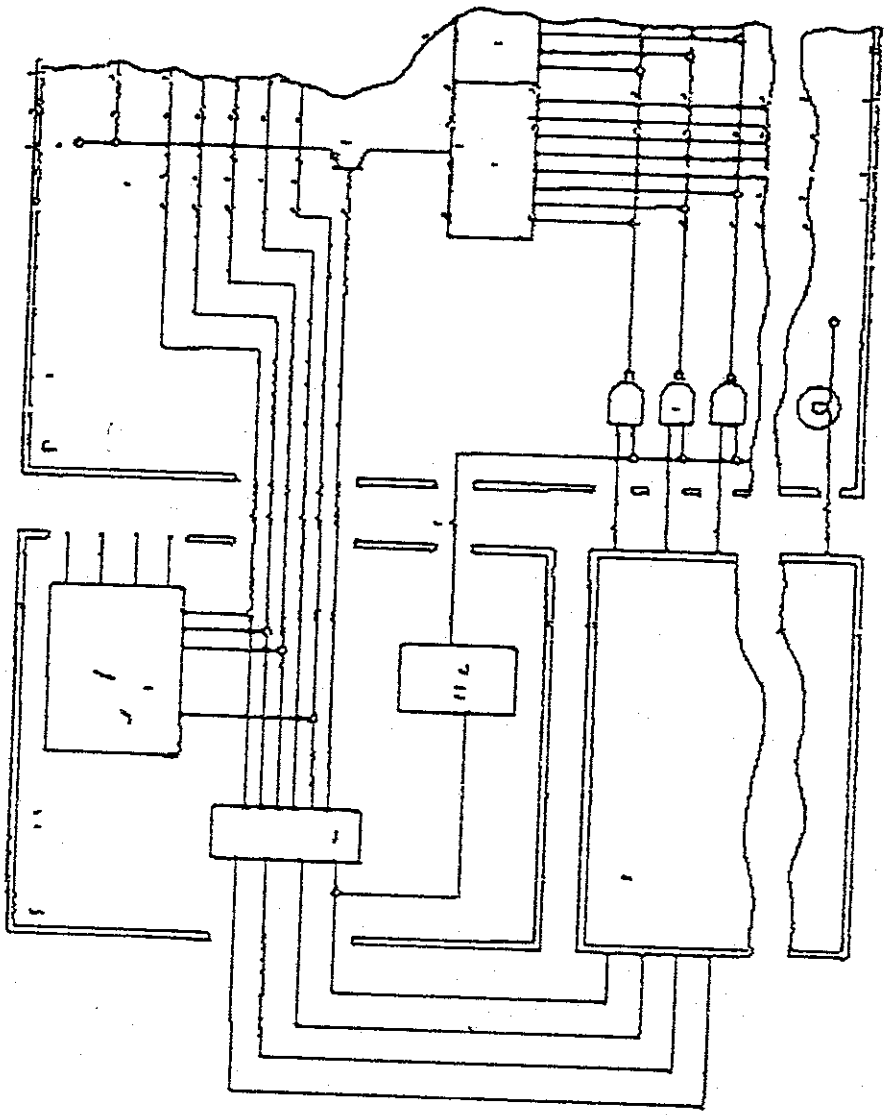


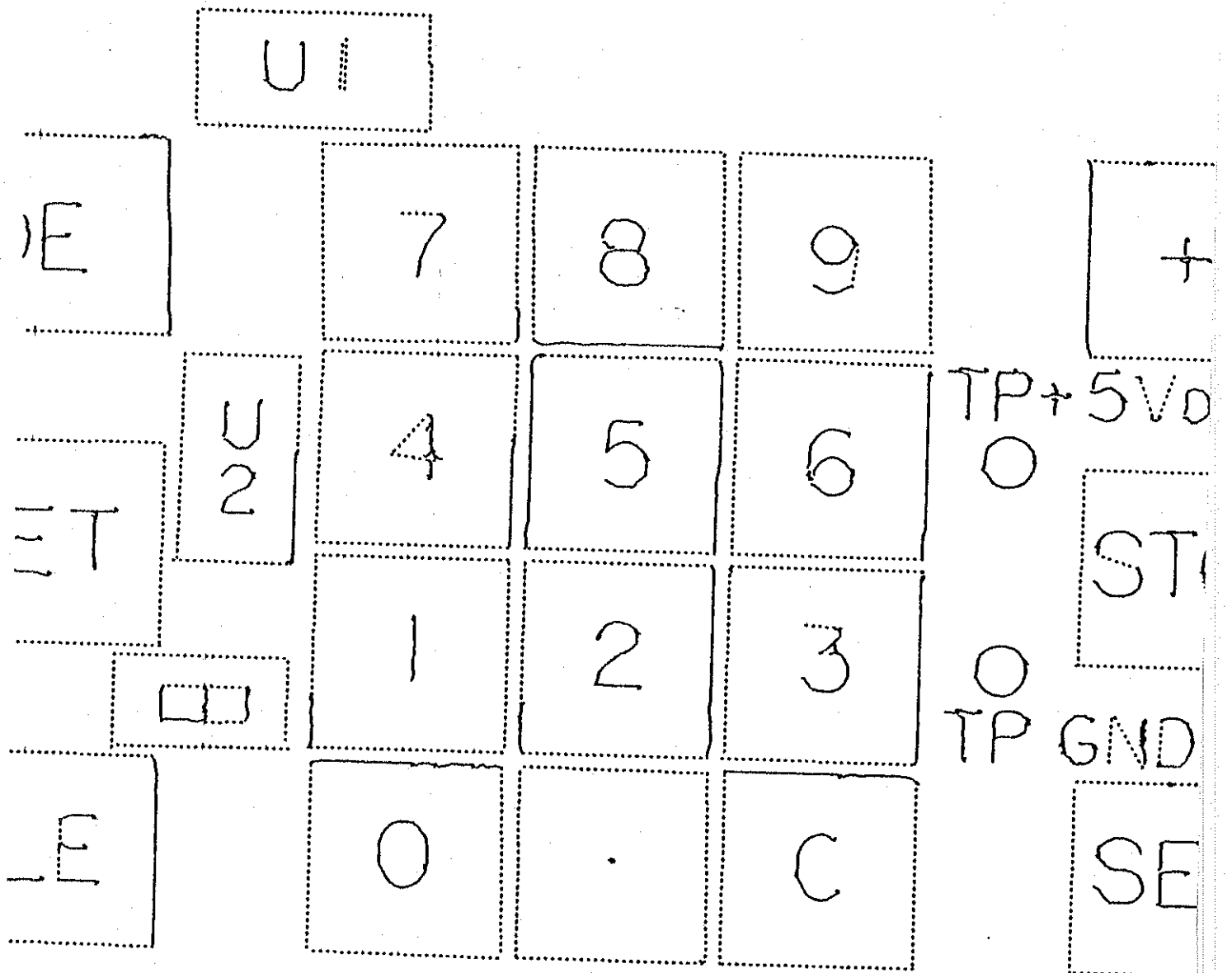





1
2
3
U7
U8


U1
MODE
7
8
9
+
−
U2
4
5
6
TP+5Voc
PRESET
STO
RCL
1
2
3
TP GND
REFILE
0
.
C
SEARCH




U1
7
8
9
4
5
6
1
2
3
0
C
U2
TP+5VD
TP GND