An Overview of Run-length Encoding of Handwritten Word Images

Petr Slavík and Venu Govindaraju
CEDAR
SUNY Buffalo
{slavik,govind}@cedar.buffalo.edu

August 25, 2000

Abstract

Analysis of handwritten word images is closely tied to the method of representing the images. Different representations have their own sets of advantages and disadvantages. In this paper, we propose a novel method of encoding handwritten images using vertical runs that significantly simplifies the implementation of several image-processing tasks pertaining to handwriting recognition. We demonstrate the advantages of both horizontal and vertical run-length encoding schemes and compare them to other widely used representations like chain-code and bitmap. We illustrate ease of use of horizontal runs for correcting the slant angle, image smoothing, and base-line detection and vertical runs for correcting the skew angle and character segmentation. We believe this paper will serve as a useful tutorial in image representation schemes used in handwriting analysis and recognition.

1 Introduction

Representation of handwritten images is an important issue in handwriting recognition. It impacts on the amount of memory needed to store an image and on the efficiency of image processing. In this tutorial, we discuss run-length representation of images and compare it with other commonly used encoding schemes like bitmap, chain-code, and fax-compressed representations. We guide the comparison of the various image encodings from the perspective of tasks pertaining to handwriting recognition, such as slant and skew angle corrections, noise removal, contour smoothing, image segmentation, etc.

More importantly, we propose a novel *vertical run-length* representation that has not been previously used in the processing of handwritten images. It has particular advantages in skew-angle correction and character segmentation.

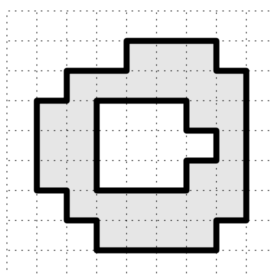
Our experiments with a preliminary version of a word recognizer based on horizontal and vertical runs show that operations such as skew-angle correction can significantly improve recognition rates (Table 1). Some researchers [2, 4] have demonstrated recognition methods that do not use slant correction. However, such methods compensate for slant during the training process where word exemplars of a variety of slants are used or the features are all slant independent.

2 Representation of Handwritten Images

Several methods have been adopted for the representation of binary images to optimize computer resources and processing time. *Line adjacency graph* (LAG) is a technique employing run length encoding to represent objects [13]. Symmetric Axis Transformation (SAT), also called medial axis, is a representation that goes orthogonal to the boundary, while the conventional description traverses the boundary [1]. *Chain code* is a linear structure that results from quantization of the trajectory traced by the centers of adjacent boundary elements in an image array [11]. Each data node in the structure represents one of eight grid nodes that surround the previous data node.

In what follows, we will describe several popular encoding schemes and briefly discuss their pros and cons.

1. **Bitmap Encoding:**
   A *bitmap representation* encodes the entire image as a two-dimensional array where each entry corresponds to one pixel—see Figure 1. Since each pixel is either on (1, black) or off (0, white), each entry in the array requires as little as 1 bit of memory space. This significantly reduces the memory requirements for storing the image and makes the bitmap encoding very popular. On the other hand, any image manipulation is very slow, since each pixel has to be processed individually.



(a)

0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0
0	0	1	1	1	1	1	1	0
0	1	1	0	0	0	1	1	0
0	1	1	0	0	0	0	1	0
0	1	1	0	0	0	1	1	0
0	0	1	1	1	1	1	1	0
0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0

(b)

Figure 1: Original image (a) and its bitmap representation (b)

2. **Chain-code Encoding:**
   Researchers in pattern recognition have recently favored the chain-code representation which is succinct and convenient for image processing. The approach is to store only the information about the pixels on the "boundary" of black regions ([7, 8]). The image is represented as a list of exterior and interior contours. Each contour is essentially an array of consecutive boundary pixels, where each pixel is represented by its x and y coordinates, and some other derivable information such as the length and type of the contour, size of the bounding box, etc. Figure 2 shows a handwritten image and its contour representation.

   Chain-code representation is well-suited for slant-angle detection ([7], [9], [10]). However, any changes in the image appearance like correcting the slant angle or smoothing require special care since changing coordinates of even a single point on the contour can introduce contour breaks or illegal contour points. Segmentation of connected components into subcomponents is even more cumbersome since contours have to be

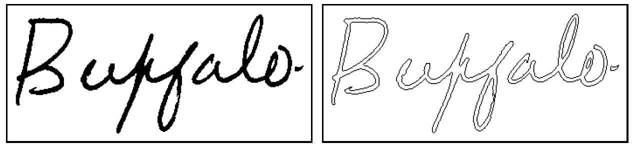


Figure 2: Handwritten image and its contour representation

split into several pieces, interior contours can become exterior contours, and at each segmentation point extra contour elements have to be added. As a result, this requires extra memory for copying parts of contours while adding new contour elements. If linked lists are used with dynamic memory allocation, processing of segment contours becomes even more inefficient as each contour can occupy several distinct memory blocks.

3. **Fax-compressed Images:**
   The fax-compressed representation of images is used for fast transmission of bitonal (black-and-white) images over phone lines. As described in detail in [5], in the basic one-dimensional encoding, the image is scanned row by row from left to right and the lengths of horizontal runs (horizontal line segments of pixels of the same color) of white pixels and the lengths of horizontal runs of black pixels are encoded using a Huffman code [3], [5]. The beginning of each new row is represented by a special new-row symbol. This results in a very compact representation of the entire image.

   The two-dimensional modified READ coding scheme [5] while more succinct is also more complicated. Here, some rows are represented using the one-dimensional encoding while other rows are encoded using only the differences between the current and the previous row.

   While both encoding schemes are very efficient for image transmission, they are not convenient for image processing. This is because of two factors: (a) the use of a Huffman code for encoding the run lengths makes it complicated to (say) change the length of a run, and (b) changing the length of a single run immediately affects the positioning of all the runs to the right (in one-dimensional encoding) and possibly below (in two-dimensional encoding). It turns out, however, that both problems can be avoided by modifying the one-dimensional fax-compressed encoding into the run-length representation.

4. **Horizontal Run-length Encoding:**
   Horizontal run-length encoding [14] is a straightforward modification of the fax representation. One simply keeps track of horizontal runs of black pixels, storing the coordinates of the first pixel and the run length. Figure 3(a) shows an image of a letter "B" together with the corresponding horizontal runs. We have 32 different runs, hence this image can be represented as an array of 32 triples of the form (*row, first-column, run-length*).

   For further image processing, however, we need more that just coordinates of the first pixel and the length of each run. We also need to keep track of neighbor runs, i.e. runs above and below that touch the current run. For example, run 3 in Figure 3(a) has two neighbors above, namely runs 1 and 2, and one neighbor below—run 4. In

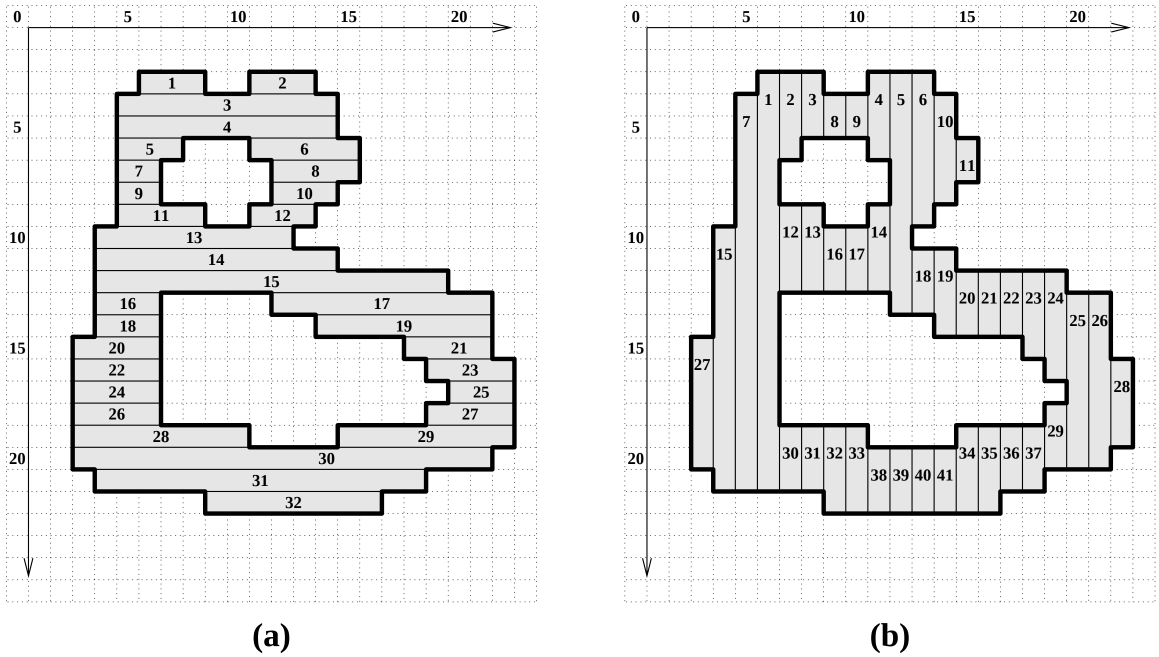


Figure 3: Horizontal and Vertical run-length representation of an image

general, the number of neighbors varies from run to run. It is convenient to represent the neighbor relation using linked lists of pointers (links) to horizontal runs. It is also convenient to keep track of the run on the same row immediately to the right of the current run. This is summarized by the following implementation of horizontal runs (H-runs) and the links (H-links) between them. This implementation is closely related to the Line Adjacency Graph given by Pavlidis in [12].

```
struct Hrun {
  int r,                        /* The row of the horizontal run         */
      c1, c2,                   /* First and last columns of the run     */
      len,                      /* Length of the run                     */
      traced;                   /* Tracing indicator                     */
  struct Hrun *right_run;       /* H-run to the right                    */
  struct Hrun_Link *abv, *bel;  /* Links to touching runs above and below */
};

struct Hrun_Link {
  struct Hrun *hptr;                                /* pointer to H-run    */
  struct Hrun_Link *next;                           /* pointer to next link */
};
```

5. **Block-adjacency Graph (BAG):**
The block-adjacency graph was first introduced in [15]. It is a generalization of the line-adjacency graph where adjacent horizontal runs with the same starting and ending columns are merged into a block that is encoded by the coordinates of its top-left and bottom-right corners. One can also keep track of adjacent blocks above

and below using a structure similar to our horizontal links. Even though BAG offers significant space reduction for images with long vertical edges (see e.g. [6] where BAG is used for encoding pages of printed text) it offers minimum (if any) space reduction for handwritten images due to lack of long vertical edges. BAG would also be inconvenient for the kind of image processing considered in this paper since the size and location of blocks in the processed image might differ significantly from those of the original image.

6. **Vertical Run-length Encoding:**
The *vertical run-length* representation is similar to the horizontal representation. We consider *vertical* runs of black pixels and store the column of the run, the first row of the run, and the run height. We keep track of touching runs in the left and right columns and the two runs in the same column immediately above and below each run. We also determine the "type" of each vertical run. This value depends on the number of left and right neighbors of the V-run. For example, an isolated run (a run with no neighbors) has `type` 0, a run with one left neighbor and no right neighbors has `type` 3, a run with two or more left neighbors and no right neighbor has `type` 6, etc. In general,

$$\texttt{type} = 3 \times \text{number-of-left-neighbors} + \text{number-of-right-neighbors},$$

where more than two neighbors still count as two. This classification significantly speeds up various operations such as traversing of connected components where one has to visit all the touching V-runs. This is because majority of vertical runs (about 90%) are of type 4, i.e. have exactly one left and one right neighbor. Hence for runs of type 4, one does not have to use recursion that would otherwise be required for visiting neighboring runs.

Figure 3(b) shows an image of a handwritten letter "B" with vertical runs. Notice that in both Figures 3(a) and (b), the runs are numbered from left to right and from top to bottom. Even though this is not really necessary, it allows some shortcuts in image manipulation. Following is our implementation of vertical runs (V-runs) and the corresponding links (V-links).

```
struct Vrun {
  int col,                            /* The column of the vertical run  */
      r1, r2,                         /* First and last rows of the run  */
      height;                         /* Height of the run               */
      traced;                         /* Tracing indicator               */
  unsigned char segID,                /* Id of run's segment             */
                type;                 /* Type of the run (0-8)           */
  struct Vrun *run_bel, *run_abv;     /* V-runs below and above          */
  struct Vrun_Link *left, *right;     /* Links to touching runs          */
};

struct Vrun_Link {
  struct Vrun *vptr;                             /* pointer to V-run     */
  struct Vrun_Link *next;                        /* pointer to next link */
};
```

The `segID` field is used to keep track of the primitive segment that contains the V-run. This information is useful for extracting features of individual segments and their combinations. Figure 4 shows a diagram of some of the links and runs touching V-run number 1.

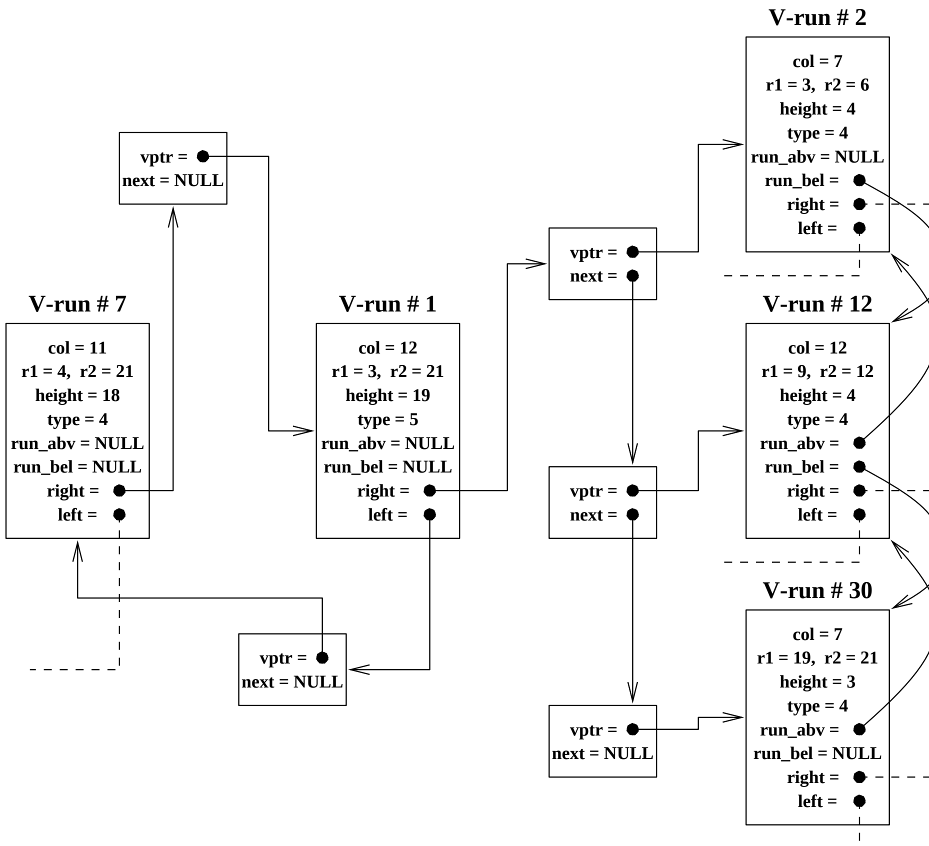


Figure 4: Representation of images using V-runs and V-links.
A snippet of the vertical line adjacency graph of the image from Figure 3(b) shows V-run #1, both its left (V-run # 7) and right neighbors (V-runs # 2, 12, and 30), and the corresponding V-links.

3 Image Processing with Horizontal Runs

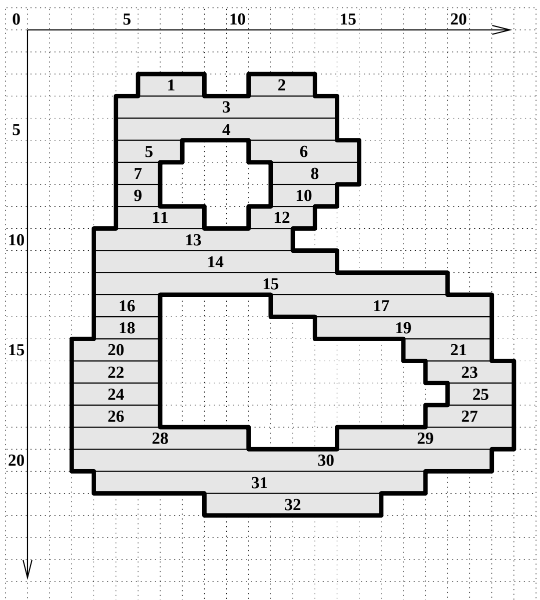
When a handwritten image is encoded in terms of horizontal runs, it is easy to shift entire runs of pixels along the horizontal x-axis. In what follows, we will demonstrate how to efficiently use horizontal runs for *slant-angle detection*, *slant-angle correction*, and subsequent *image enhancement*.

Since bitonal images are often stored in the bitmap format, we have to be able to convert a bit-mapped image into horizontal run-length representation. Fortunately, such conversion is fairly straightforward. We simply scan the bit-mapped image row-by-row, left to right and any time we encounter a black pixel after a run of white pixels, we start a new horizontal run storing the row and the starting column. When we encounter a white pixel after a run of black pixels, we end the horizontal run storing the last column of the run. After scanning pixels in the entire row, we traverse the runs on the current and previous rows from left to right and use their end-points to establish any touching or overlap between a pair of runs, hence determining the neighbor-below and neighbor-above relations. It should be noted that this can be done in time $O(n_1 + n_2)$, where n_1 and n_2 are the numbers of runs in each row.

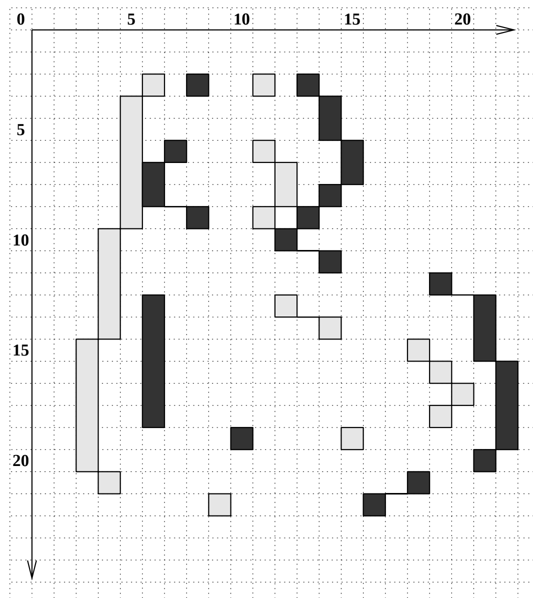
3.1 Slant-angle Detection

In order to describe the algorithm for detecting the slant angle, let us first define some basic terms.

Given an image of a handwritten phrase, we define a *left contour* of the image as a collection of left endpoints of horizontal runs such that the absolute difference between x-coordinates of two consecutive runs is at most 2. One can similarly define a *right contour* using the right endpoints of horizontal runs. Figure 5(a) shows an image of the letter "B" represented by horizontal runs. This image has altogether 7 left and 7 right contours (Figure 5(b)).



(a)



(b)

Figure 5: Horizontal run-length representation and left (in gray) and right (in black) contours

For any (left or right) endpoint of a horizontal run, we define "*slope*" of the endpoint as the sum of (four) differences between the x-coordinates of (left or right) endpoints of five consecutive runs with the current run in the middle. Thus if the current endpoint is on row r, we take the points from the same contour on rows $r-2$, $r-1$, r, $r+1$, and $r+2$, and add the four (absolute) differences between the adjacent points. If there are not sufficiently many endpoints from the same contour above or below, we set the missing differences equal to 2. We say that an endpoint has a *near-vertical slope* if *slope* ≤ 5. A *near-vertical line* is portion of a (left or right) contour consisting of consecutive endpoints with near-vertical slopes.

The *slant angle* of a near-vertical line is the angle α between the line-segment connecting the endpoints of the near-vertical line and the vertical axis, measured in a clockwise direction. Figure 6 shows a contour and the *slope* values for each contour point. Each contour point is near-vertical hence the entire contour is a near-vertical line with slant angle α. Since we measure the slant angle clockwise, α is less than 0.

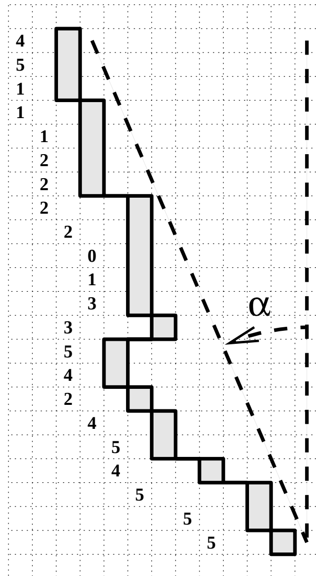


Figure 6: An example of a near-vertical line

The algorithm for computing the average slant angle of near-vertical lines is simple. Just traverse all the left and right contours computing the slope for each contour point and the slant angle for each near-vertical line. Then take the weighted average of slant angles of all near-vertical lines—add all the slant angles multiplied by the height of the corresponding line and divide by the total height of all near-vertical lines.

This algorithm is similar to the algorithm given in [7] where Kim and Govindaraju use contour representation for detecting near-vertical lines. A different, simpler algorithm for slant-angle detection is described in [9] and [10]. Here, the authors use contour representation of the image and simply count the number of vertical transitions (n_v) and the numbers of diagonal transitions in the positive x-direction (n_+) and the negative x-direction (n_-). The average slant angle is given by $\tan\theta = \frac{n_+ - n_-}{n_v + n_+ + n_-}$. While this method is fast, it lacks precision given that it completely disregards all horizontal transitions.

3.2 Slant-angle Correction

The goal of slant-angle correction is to obtain an image for which the average slant angle of near-vertical lines is close to 0. This is achieved by shifting all runs in the horizontal direction by a number of pixels depending on the average angle obtained in step 1 and the row of the run. More precisely, the shift is given by the formula $\mathit{shift} = \mathit{row\#} \times \tan\alpha$, where α is the slant angle. Figure 7 shows a handwritten image before and after slant-angle correction.

We take advantage of the fact that the "offset" of all runs on any one row is the same and that runs are generated row-by-row. This allows us to compute the offset only once for each row and save it in a register variable. If one were using a contour representation of

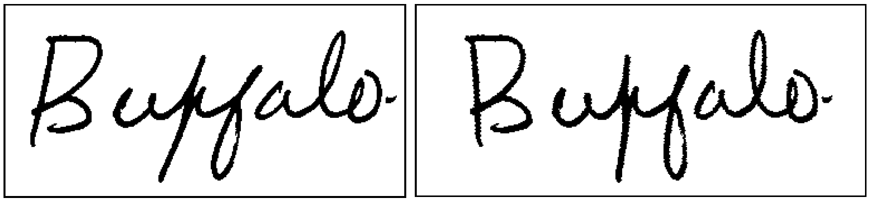


Figure 7: Slant-angle correction

the image, slant-angle correction would require traversing the contours pixel-by-pixel and either recomputing the offset for each pixel or storing the values of the offset for each row in an array and then obtaining the saved values each time the offset is needed.

3.3 Image Enhancement and Smoothing of Contours

The image resulting from the slant-angle correction has in general ragged contours due to rounding of fractional values during slant correction as well as due to the presence of noise in the original image (Figure 7).

To smooth the image, we traverse the runs in the order they were generated (top to bottom and left to right) assigning each left (right) endpoint a new x-coordinate that equals the average of the x-coordinates of the left (right) endpoints of the runs immediately above and below. However, in exceptional cases when the new x-coordinate would change by more than 2, we set the new coordinate to a value that is between the x-coordinates of the endpoints above and below and closest to the original x-coordinate. In case such an adjustment would still be greater than 2 (and the end-point is "sticking out"), we leave the value of the x-coordinate unchanged.

In case the adjustments result in overlapping runs, we simply merge such overlapping runs. We also merge runs that have only one pixel of white space between them. This results in removal of small "holes" that are usually caused by noise. In some cases, however, this may result in filling of important holes, for example the hole in the letter "e".

The algorithm above does not remove imperfections in the vertical direction. In order to smooth the image even further, we remove any runs that have no neighbors above (below) and have exactly one neighbor below (above) at least 4-times longer than the run in question. At this stage, we also remove any small "blobs". A connected component is considered "small" if its area or its width is small compared to the average run length.

Figure 8 shows the first character from Figure 7 before and after the enhancement.

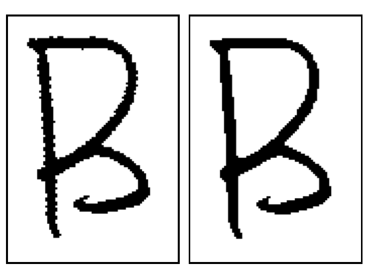


Figure 8: Image enhancement

3.4 Evaluating Preprocessing Results

The process of evaluating the images that result from preprocessing is higly subjective. Any attempt to quantify the quality of an image (by trying to recognize it or assigning it some kind of a score) would be recognizer specific or would depend on the way the score is determined. Thus our criteria throughout this paper was mostly visual appearance, hoping that what appears “nice” to us, would be easy to deal with and recognize by a computer.

3.5 Base-line Detection

A *baseline* of a word approximately aligns with the base of characters. The base of the characters themselves can be approximated by local minima on the lower contour of the word.

A typical way of detecting a baseline is to consider all local minima of the lower exterior contours and then use linear regression to construct a straight line that best fits these points. Clearly, detection of such minima from a bitmap or contour representation is computationally intense. On the other hand, finding such minima from the horizontal run-length representation is trivial. One simply collects all the runs that have no neighbors below. However, a blind use of such approach may result in considering those minimum points that have nothing to do with the base-line, like endpoints of descenders or minima of image fragments.

Keeping the above mentioned factors in mind, a naive algorithm for detecting the base-line could be the following—find the center of mass of the entire image; then the base-line is the straight line that best fits all the H-runs that are below the center of mass and have no neighbors below. There are two problems with such an algorithm:

- Many spurious minima (in particular the endpoints of descenders).
- For long words or phrases with significant skew, a large portion of the word (as much as 1/3) could be written completely above the center of mass, hence some minima would be missed.

To remedy the first problem, one can first detect a horizontal approximation to the baseline using the minima below the center of mass. To find a general baseline approximating the minima, one can use linear regression on only those minima that are within a certain distance from the previous baseline approximation.

Remedying the second problem is more complicated. Instead of the center of mass, one can consider a “center-line”, that is a line that best fits the centers of mass of each column. One can then compute a first approximation of the baseline using all points below the “center-line” and repeat this process considering only those minima that are “close” to the first approximation. The center-line can be detected by using linear regression on all pixel coordinates and this can be done efficiently by traversing all horizontal runs and adding x, y, x^2, y^2, and xy for all pixels in each run. Note that even though we sum over all pixels from each run, only the values `row`, `c1`, and `c2` are needed for any of the sums and the computations do not depend on the length of the run.

There is however one problem with any algorithm that uses the notion of a center-line. On “short” words with several descenders, a center-line could intersect the true baseline resulting in several base-minima being missed causing the approximation to be inaccurate. There does not seem to be any easy way around this so we have decided to use two different algorithms, a conservative algorithm for "short" images, and an aggressive algorithm for "long" images. To classify images as short or long we use the aspect ratio of the image bounding box. Based on experiments on several hundred images extracted from US postal addresses, we have determined the division point to be 2.75. Thus images with aspect ratio less than 2.75 are considered short while images with aspect ratio greater than 2.75 are considered long. To reduce the impact of the word's starting (or ending) strokes, only those runs that are at least 5 pixels away from the vertical sides of the image bounding box are considered.

"Short" images:

1. Find the center of mass.
2. Find all the runs that are below the center of mass and have no neighbors below, store their coordinates, and determine a horizontal approximation of the base-line and the standard deviation σ.
3. Consider only those minima that are within $\sigma + \frac{4}{(\sigma+1)}$ from the horizontal baseline and use linear regression to find the straight line that best fits these minimum points. This is the final base-line approximation.

"Long" images:

1. Find the "center-line".
2. Find all the runs with no neighbors below and store those endpoints that are below the center-line. Use linear regression to find a straight line that best fits these points and determine the standard deviation σ.
3. Consider only those minima that are within $\sigma + \frac{4}{(\sigma+1)}$ from the first baseline approximation and once again use linear regression to find the straight line that best fits these minimum points. This is the final base-line approximation.

Combination of these two algorithms produced accurate base-lines even for images with several descenders or large skew angles. Figure 9 shows the process of determining the base-line for both short and long images.

4 Image Processing with Vertical Runs

In the previous section we have demonstrated the use of horizontal run-length representation for manipulating images. In this section our focus will be on *vertical* run-length representation.

4.1 V-run Generation

The algorithm for V-run generation differs from H-run generation, since we cannot efficiently scan the pixels of the image column-by-column. Imagine scanning a bitmap image row-by-row and trying to generate vertical runs. If the current pixel is black and the pixel directly above is white, we are at the top end-point of a vertical run. If the current pixel is white and the pixel directly avove is black, the pixel above is the bottom end-point of a vertical run. Thus one can traverse the image row-by-row, starting a new V-run anytime he encounters

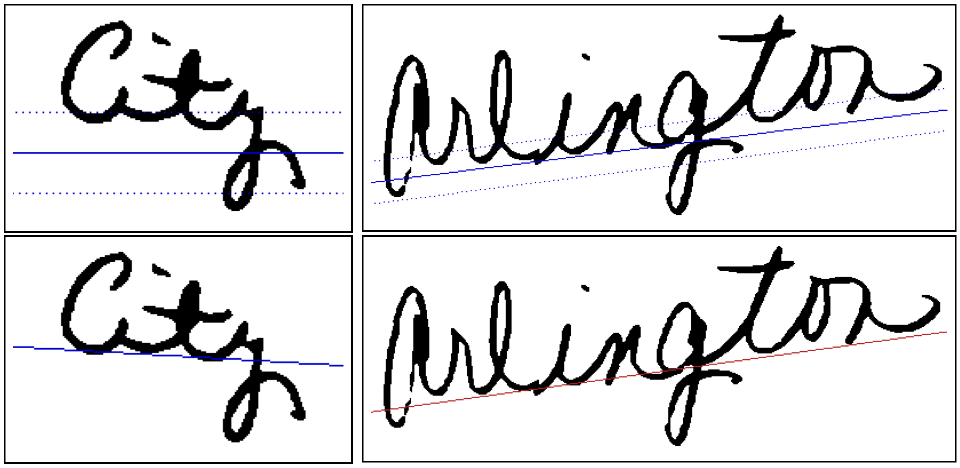


Figure 9: Base-line detection for short and long images
The top picture shows the intermediate steps—horizontal base-line approximation for short images and the first (rough) approximation of the base-line for the long images together with region from which the minima points are taken for the final step. The bottom picture then shows the resulting base-line.

a top end-point and finishing a V-run when he gets to the bottom end-point. This can be efficiently done by traversing the horizontal runs in the order they were generated (row-by-row, left-to-right), and having for each column of the image two variables—one that indicates whether there is a V-run being created in this column; i.e., whether the pixel on the previous row was black, and a second variable that points to the V-run most recently generated or currently in the process of being generated. Thus, at anytime we get to a new H-run, we simply finish all the V-runs corresponding to the white pixels between the current and the previous H-run, and start (or continue) V-runs in columns corresponding to black pixels of the current H-run.

For efficiency reasons, we combine the V-run generation with creating the right-neighbor, left-neighbor, run-below, and run-above links. Each time we create a new V-run, we check whether there are active runs immediately to the left and to the right, and if so, set the values of fields `left` and `right` in the structure `Vrun`. For runs above and below, we simply link the V-run that just started with the run previously generated in that column.

4.2 Skew-angle Correction

Correcting the skew angle (obtained from the base-line) is very similar to correcting the slant angle. However, traversing V-runs in the order they were generated would be inefficient, since V-runs are not organized in column by column fashion and hence the shear-shift value would have to be computed for each V-run. To avoid this excessive computation, we take advantage of the array of pointers to the last V-run in each column that was created during the V-run generation process and the fact that each V-run has a pointer to the run above. Thus the shift has to be computed only once for each column. And we can take this even further! Since the skew angle is usually very small, the shift is identical for several consecutive columns. We call the set of columns with the same value of shear-shift a "plateau". Each plateau has length of k or $k+1$, where $k = \lfloor 1/\tan\omega \rfloor$, with ω being the skew angle. Moreover, the value of the shift for the next plateau is increased (or decreased, depending on the sign of ω) by exactly 1. Thus we can further reduce the number of times the shift is computed by computing it only once for each plateau—to determine whether the plateau is k or $k+1$ pixels long. The shifting itself is straightforward.

4.3 Smoothing after Skew Correction

Unfortunately, correcting the skew angle can result is several undesired side-effects:

1. Two runs that were previously touching (by corners) do not touch anymore.
2. Two runs that were not previously touching are now touching (by corners).
3. Contours do not appear to be smooth anymore.

Clearly, the above problems can happen only at a "break-point", that is at a boundary between two consecutive plateaus. Larger skew angle result in more break-points which renders the image less smooth. Since the image was previously smoothed (after slant correction), we want our smoothing this time to be less aggressive. In fact, if the skew angle is 0, we would prefer not to do any smoothing at all.

In view of the above remarks, it would seem reasonable to apply our smoothing only in the vicinity of the break-points. Thus images with large skew angle would undergo more smoothing than the ones with skew angle close to 0. However, comparing this "close-to-break-point-only" smoothing with "full-image" smoothing shows that even an image that is not corrected for skew benefits from some (conservative) smoothing in the vertical direction. Figure 10 shows an example of processing an image with a large skew angle.

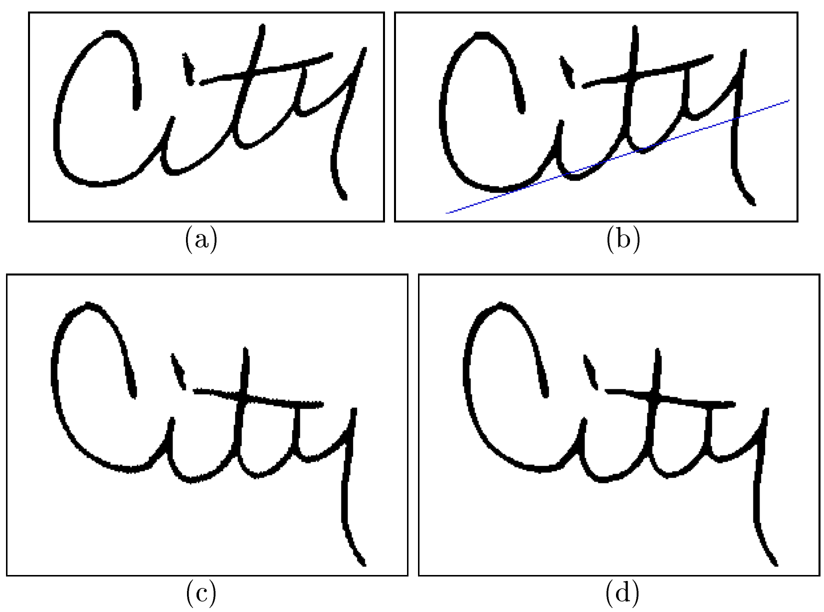


Figure 10: Skew correction and vertical smoothing—(a)original image, (b)image after slant-angle correction and smoothing (with base-line), (c)image after skew-angle correction, (d)image after vertical smoothing.

We would like to point out however, that images with large skew angles are rare and hence images after skew-angle correction (before smoothing) are normally smoother than the one showed in Figure 10(c).

The algorithm for vertical smoothing is somewhat similar to the algorithm for horizontal smoothing. Endpoints of runs are again adjusted based on the average of y-coordinates of two neighboring runs. However, we limit the change of the y-coordinate to at most 1. Also, averages are rounded up (ceiling) or down (floor) based on the endpoint of the run in question, so that endpoints that are within 0.5 of the average are not adjusted. As in the case of smoothing horizontal runs, special care has to be taken of runs of length 1 and runs where smoothing has resulted in touching or even overlapping of runs in the same column. We join runs in the same column that touch or overlap as a result of smoothing as well as runs that are only one pixel apart and have a common left or right neighbor. Figure 11 shows some examples of endpoint adjustments.

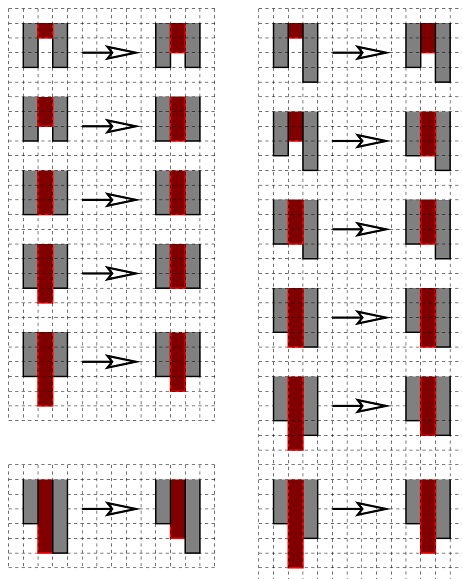


Figure 11: Typical cases for vertical smoothing—run in the middle has its lower endpoint adjusted based on its neighbors

After correcting the skew angle and final smoothing of the image, one can determine the type of each run based on the number of left and right neighbors—see Subsection 6. For this and other simple operations, the runs can be easily visited in the order they were generated by taking advantage of the fact that we keep the runs in an array. This makes such traversals simple and fast.

4.4 Segmentation of Word Images in Vertical Run-length Encoding

Most word recognition systems rely on a segmentation submodule that divides a word into its constituent characters. Typically, a word is segmented into primitive segments that correspond to a single character or its part (in some cases, a primitive segment is allowed to represent a pair of neighboring characters). Primitive segments and their combinations[1]

[1] In what follows, we will use "segment" to refer to both primitive segments and their combinations

are then fed to a module that extracts their features and passes them to the recognition module. Regardless of the features being extracted or the way primitive segments are obtained, one needs an efficient way of determining the part of the image that belongs to a specific segment. Specifications like "all pixels with x coordinates between a and b" clearly work only in simple cases.

In [7], Kim and Govindaraju used chain-code for representing both the original image and the image segments. In such case, obtaining a contour representation of each segment is complicated and error prone. One has to split the original contour, then create additional partial contours through the segmentation points, and then connect all these pieces into one or more contours (stored in some additional storage buffer) that represent one particular segment. Such a contour list is then passed to a feature extractor. Other authors either do not specify their implementation at all or use similar cumbersome techniques like copying the image segment pixel-by-pixel to a memory buffer and then use subimages for feature extraction or further processing.

On the other hand, with vertical run-length encoding, this process is much easier. Splitting an image into segments is as easy as removing left-to-right or right-to-left links between two neighboring runs; joining two (adjacent) segments together is as easy as bringing the links back in. Thus instead of creating a list of contours in an extra memory buffer and then passing such a list to a feature extractor, we set left (right) links of the segmentation runs to NULL (while saving these links for later use) and then pass a list of V-runs, one run for each connected component of the segment, to a feature extractor. We plan to describe a segmentation module that takes a full advantage of vertical run-length encoding in the following paper.

5 Experiments

Using 3000 handwritten word images we have conducted experiments to illustrate the impact of preprocessing operations such as smoothing, skew correction, and slant correction, on the performance of a run-length based recognizer similar to the one described in [7]. Table 1 shows the improvement in performance of about 5% for lexicons of size 10 and about 10% for lexicons of size 100. Most of the improvement comes from the correction of slant and skew. It is to be noted that the impact of these preprocessing operations also depends on the types of features used by the word recognizer and the methodology of segmentation.

Lexicon size	10		
Preprocessing	Smoothing, Skew and Slant correction	Only Smoothing	None
% 1st choice correct	95.70	91.69	91.48
% 1st OR 2nd correct	98.40	96.16	95.76
Lexicon size	100		
Preprocessing	Smoothing, Skew and Slant correction	Only Smoothing	None
% 1st choice correct	87.22	75.68	76.28
% 1st OR 2nd correct	92.36	84.35	84.23

Table 1: The recognition rate of a handwritten word recognizer benefits from the inclusion of preprocessing operations such as smoothing, skew correction, and slant correction as described in this paper using horizontal and vertical runs.

We have also compared the execution times of our slant-angle detection, correction and subsequent smoothing using horizontal runs to a preprocessing module with identical functionality [7] that uses chain-code for representing handwritten images. The timing results on Sun Ultra 5 workstation are given below.

	chain-code	H-runs
Slant-angle detection:	0.15	0.43
Slant-angle correction:	0.97	0.03
Smoothing:	0.62	0.27
Total:	1.74	0.73

Table 2: Average processing times (in ms) for chain-code and horizontal run-length representations.

Notice the enormous difference in average execution time for the slant-angle correction. This is caused by computing the shift for each chain-code element separately. Also, smoothing is slower for the chain-code representation due to the necessity of copying all contours since extra chain-code elements may have to be inserted into contour brakes caused by angle correction. On the other hand, the slant-angle detection using H-runs is not that fast since runs have to be traversed using neighbor links and that is of course much slower than traversing in the order the runs were generated.

6 Conclusion

We have illustrated the advantages of both horizontal and vertical run-length encodings by describing methods that correct both the slant and skew angles and smooth out the resulting images. We have shown how efficient it is to use horizontal run-length encoding for slant-angle correction and image smoothing. We have introduced a novel vertical run-length encoding that makes skew-angle correction particularly straightforward. The true power of vertical runs lies however in their use for segmentation of words into (sub)characters. This will be part of our future work. We are currently developing a segmentation module that relies solely on vertical run-length representation of a handwritten word. The V-runs make it easy to split and/or merge segments and avoid laborious contour cutting and/or gluing (as would be required in chain-code representation) or splitting or joining of horizontal runs (as in H-run representation).

Acknowledgments

We would like to thank our colleagues at CEDAR for many fruitful discussions. In particular, the experience and insights of Evie Kleinberg were very helpful.

References

[1] H. Blum and R. Nagel. Shape description using weighted symmetric axis features. *Pattern Recognition*, 10:167–180, 1978.

[2] M. Brown and S. Ganapathy. Preprocessing techniques for cursive word recognition. *Pattern Recognition*, 16(5):447–458, 1983.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[4] M. Cote, E. Lecolinet, M. Cheriet, and C. Suen. Automatic reading of cursive scripts using a reading model and perceptual concepts, the percepto system. *International Journal on Document Analysis and Recognition*, 1(1):3–17, 1998.

[5] R. Hunter and A. H. Robinson. International digital facsimile coding standards. *Proceedings of the IEEE*, 68(7):854–867, July 1980.

[6] A. K. Jain and B. Yu. Page segmentation using document model. In *Proceeding of the Fourth International Conference on Document Analysis and Recognition*, volume 1, pages 34–38, Ulm, Germany, Aug. 1997.

[7] G. Kim and V. Govindaraju. Efficient chain code based image manipulation for handwritten word recognition. In *SPIE Symposium on Electronic Imaging Science and Technology*, pages 262–272, San Jose, CA, 1996.

[8] F. Kimura, M. Shridhar, and N. Narasimhamurthi. Lexicon driven segmentation - recognition procedure for unconstrained handwritten words. In *International Workshop on Frontiers in Handwriting Recognition (IWFHR-3)*, pages 122–131, Buffalo, NY, May 1993.

[9] F. Kimura, M. Sridhar, and Z. Chen. Improvements of a lexicon directed algorithm for recognition of unconstrained handwritten words. In *Proceedings of the Second International Conference on Document Analysis and Recognition*, pages 18–22, Tsukuba Science City, Japan, 1993.

[10] S. Knerr, V. Anisimov, O. Baret, N. Gorski, D. Price, and J. C. Simon. The A2iA intercheque system: Courtesy amount and legal amount recognition for french checks. In S. Impedovo, P. S. P. Wang, and H. Bunke, editors, *Automatic Bankcheck Processing*, volume 28 of *Series in Machine Perception and Artificial Intelligence*, pages 43–86. World Scientific, 1997.

[11] C. Lu and J. G. Dunham. Highly efficient coding schemes for contour lines based on chain code representations. *IEEE Transactions on Communications*, 39(10):1511–1514, 1991.

[12] T. Pavlidis. A minimum storage boundary tracing algorithm and its application to automatic inspection. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8(1), Jan. 1978.

[13] T. Pavlidis. *Algorithms for Graphics and Image Processing.* Computer Science Press, Rockville, MD, 1982.

[14] A. Rosenfeld and A. C. Kak. *Digital Picture Processing.* Academic Press, New York, second edition, 1982.

[15] B. Yu and A. K. Jain. A generic system for form dropout. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18:1127–1134, 1996.