Devanagari Pen-written Character Recognition

Abhiram Ranade* Meghanad Ranade†

June 18, 2001

Abstract

We present a system for recognizing pen-written characters. We developed the system primarily for the Devanagari script, but by changing our database it can be adapted for other scripts. Indeed, our system currently also recognizes (isolated) Roman alphabet.

Our system handles the full range of Devanagari characters including intricacies such as compound characters.

1 Introduction

We have developed a system for recognizing pen (or mouse) written Devanagari characters. This is in the same spirit as the popular Grafitti system for palm top computers and the systems such as NPen++[3] and others, i.e. the input to our system is not a scanned image of the text but a sequence of pen positions sampled as the text is being written. Like the other systems, our system also exploits the dynamic information such as the order in which the lines get drawn.[1] Our system is knowledge based, i.e. no training is required but instead we need a database describing the categories to be recognized. Our system allows substantial variation in the input it classifies – indeed such variation is inevitable because of differences in writing styles of individuals as well as the slurring that commonly occurs while writing rapidly. Our basic classification mechanism is generic – by constructing a suitable knowledge base we expect to recognize other scripts as well. Currently we have developed such knowledge bases for the Roman script (isolated characters) and Devanagari, which is our main focus, of course. Figure 4 shows a screen shot of our program in action.

The Devanagari script is used by many Indian languages including Sanskrit, Marathi and Hindi. Several other languages such as Gujarati, Punjabi and Bengali use scripts which are very similar to Devanagari. Devanagari is more complex than the familiar Roman script in several ways: (a) It has many more basic characters in its alphabet, (b) Vowels are written as modifications of (or

*Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Powai, Mumbai 400076, India. `ranade@cse.iitb.ernet.in`

†D-21, Woodland Appartments, Gandhi Bhavan Road, Kothrud, Pune 411029, India. `mranade@bol.net.in`

[1] Suppose we consider the text obtained by reversing the pen movements and starting at the end. Clearly the end result is the same, but our system, like the others will only recognize the conventional way of writing.

diacritical marks on) the consonant characters, (c) The script allows consonant characters to be combined together to form compound characters. Such compound characters are very common. These complexities pose serious problems for scanned text (handwritten or printed), the most important being how to separate the constituents of a compound character (or even a consonant and a vowel, since the union is often seamless). These difficulties are not serious for Pen based recognition as will be seen. In fact, this may be thought of as a strong motivation for the development of pen based systems.

Pen based systems are attractive for Devanagari for other reasons also. Devanagari typing is harder than Roman typing (for all the reasons mentioned above), and further several keyboard layouts are in use. Thus, a pen-based system is very attractive for a novice (more so than is the case in English). Finally, we would like to observe that pen-based systems are likely to be easier to build for Devanagari than for the Roman script because Devanagari is not written in a cursive manner[2]

1.1 Previous work

We know of no previous work on recognition of pen written Devanagari. However, work is reported for Telugu and Tamil[2]. There is of course, very substantial work for the Roman Alphabet. In fact the focus of the research on the Roman Alphabet seems to have shifted to recognition of cursive writing[3]. Substantial work has been done on optical character recognition of printed Devanagari text, see [1] and the references therein.

We contrast our work with that of Kunte and Samuel[2]. They use a neural network classifier, which is trained substantially (about 144 samples per vowel and 16 samples per consonant). They use Wavelet based features. Our approach is dramatically different if not simpler. We need no training, but need to specify a simple description of each character. The features our classifier considers most are the the pen movement directions. Finally, our classifier is algorithmic (rather than neural network based) and is based on a model of how letters get distorted while writing.

1.2 Overview

For the purposes of writing, it is convenient to think of Devanagari characters as composed of *strokes*. A stroke is simply a shape drawn without lifting the pen. A stroke could be a complete Devanagari character, or a part of a Devanagari character. For example, the characters व म न are usually drawn using a single stroke (without the overhead line). The character ब, for example is drawn using two strokes, first the stroke for व and then a vertical/slanted line overstriking it. Our system has around 70 strokes, these include strokes representing complete vowels (e.g. उ इ), vowel markers (e.g. the stroke atop the क in के), complete consonants (e.g. क च), consonant/vowel parts (e.g. the line needed to turn a व into a ब, or the stroke needed to turn a र into a स), half consonants (e.g. the "half" क in क्या). Certain vowels and consonants (e.g. प) may be written as a single stroke or as multiple strokes (i.e. a half stroke प् followed by ा). Our system allows such differences in styles.

[2]A cursive script called *Modi* was in use for Marathi, but is at present nearly forgotten.

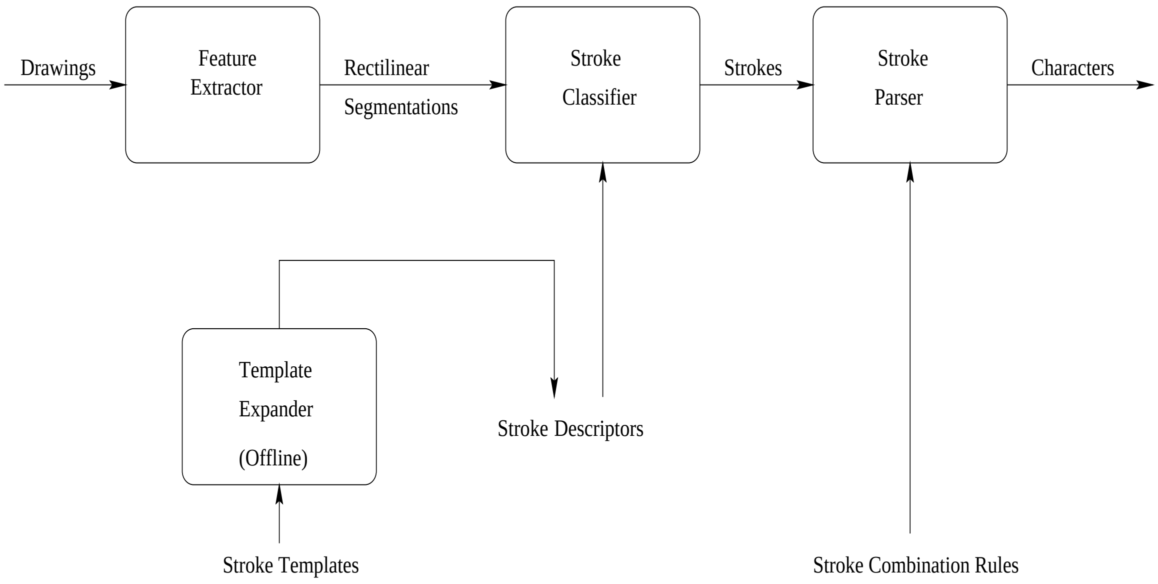

Figure 1: System Architecture

Figure 1 shows the architecture of our system. The input to our system is a sequence of *drawing*s. Each drawing is simply a set of points $p_1, \ldots, p_n$ sampled as the pen (or mouse) is dragged between putting it down and next lifting it up. From each drawing, a set of features, called its *Rectilinear Segmentation* is extracted (Section 2). These are passed on to the Stroke Classifier. The Stroke Classifier (Section 4) attempts to identify the *stroke* that the drawing represents. The seqence of strokes as generated by the Stroke Classifier is fed to the Stroke Parser (Section 5) which combines the strokes into characters constituting the script.

The knowledge base used by our program has two parts. The first is a *stroke template list* (Section 3). For each target stroke, a template must be specified describing the different variations with which the stroke may be drawn. The variations take into account differences in writing styles of different individuals and the slurring that inevitably takes place while writing rapidly. Each template is expanded during offline processing to generate a set of *stroke descriptors* which are used by the on-line Stroke Classifier. The second part of the knowledge base consists of a description of how strokes combine to produces letters.

In Section 6 we discuss some of the filtering operations we perform on the input drawing as preprocessing.

We show our program in action in Section 7 and conclude in Section 8.

2 Our feature set

The main feature set we use for classification is what we call a *Rectilinear Segmentation* of a drawing. Let $p_1, \ldots, p_n$ be the points in a drawing. Let s_i denote the slope of the segment (p_i, p_{i+1}). Let $D(p_i)$ denote the slope discretized to the horizontal and vertical directions as follows:

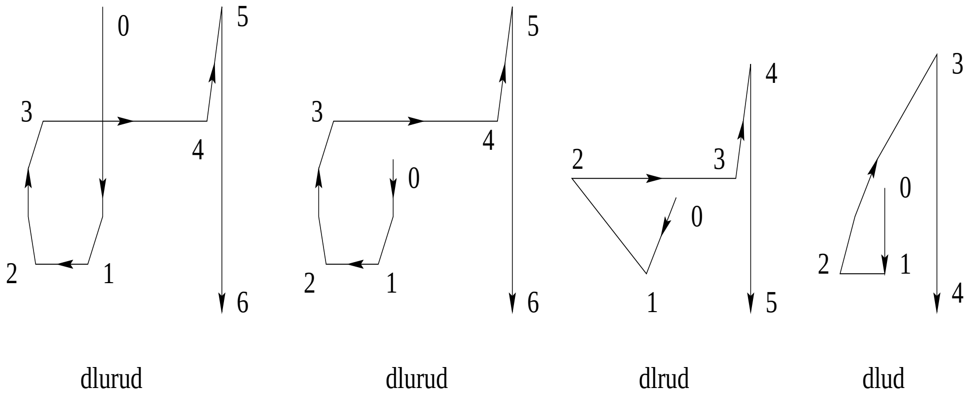

Figure 2: Drawings, segment endpoints, and signatures

$$\begin{aligned} D(p_i) &= r \text{ if } s_i \text{ is between } -45^\circ \text{ and } 45^\circ. && \text{(left)} \\ &= u \text{ if } s_i \text{ is between } 45^\circ \text{ and } 135^\circ. && \text{(up)} \\ &= l \text{ if } s_i \text{ is between } 135^\circ \text{ and } 225^\circ. && \text{(right)} \\ &= d \text{ if } s_i \text{ is between } 225^\circ \text{ and } -45^\circ. && \text{(down)} \end{aligned}$$

Let e_j be the jth point where the discretised slope changes, with $e_0 = p_1$. Let m be the number of slope changes, and define $e_{m+1} = p_n$. We will call the sequence $e_0, e_1, \ldots, e_{m+1}$ the *segment endpoint sequence*, and the sequence $D(e_0), D(e_1), \ldots, D(e_m)$ the *signature* of the input drawing $p_1, \ldots, p_n$. The Rectilinear segmentation for the drawing is simply its segment endpoint sequence together with the signature. Figure 2 shows one drawing of the single stroke letter म and three drawings of the single stroke letter न , and their segment endpoints and signatures. We note that the signature is invariant under scaling and translation.

We use the signature as a coarse classifier. It should be clear that strokes having dramatically different drawings should have different signatures. However, it should also be clear from Figure 2 why the signature itself will not classify completely. The first problem is *synonymy*: different drawings of the same stroke can have different signatures, e.g. the last three drawings of Figure 2. Even so, note, that there is some degree of similarity in these signatures which could be exploited. The second problem is *polysemy*: drawings of different strokes may have the same signature (e.g. first two drawings of Figure 2). We will consider these issues in the next section.

3 Stroke templates

A stroke template is a concise description of a stroke for the purpose of classifying drawings. Each template has 3 parts:

1. **The ideal signature:** Consider a carefully drawn, canonical version of the stroke. Call this its *ideal* drawing. The signature of the ideal drawing is called its *ideal signature*. For example, the first two drawings in Figure 2 could be regarded as the ideal drawings respectively of म and न , and their signatures as shown the ideal signatures.

2. **Signature Matching Rules:** These specify the extent to which the signature of an input drawing *must* match the ideal signature for correct classification. This is meant to overcome the synonymy problem mentioned above.

3. **Geometric Relationships:** These are conditions on the segment endpoints of the rectilinear segmentation of the ideal drawing. A typical example is "the 0th endpoint should be to the left of the 6th endpoint". We require that the endpoints in the input drawing satisfy these conditions as well in order to complete the classification. This is meant to overcome the polysemy problem mentioned above.

3.1 (Approximate) Signature Matching Rules

We have a default approximate matching rule, and the template for each stroke only encodes the exceptions to the default rule as appropriate for the stroke.

The default matching rule: An ideal signature S_I is said to match a signature S_D of an input drawing if S_D can be obtained from S_I by dropping any collection of non-adjacent segments. As an example, suppose that the second drawing of Figure 2 is the ideal drawing of the letter न . Then it will be seen that the signature "dlrud" of the third drawing matches the ideal signature according to our default rule. The signature "dlud" of the fourth drawing, on the other hand, does not match the ideal signature just using the default rule.

The default rule models the intuition that during the writing process some of the segments of the ideal signature could get dropped (or get shortened so much that our sampling process does not detect them), but too many nearby segments will not get dropped as that will alter the drawing beyond recognition even to the writer.

Exceptions to the rule: It is possible, however, that certain segments may be so characteristic of a stroke that they will not be dropped by any writer for reasons of style or writing speed. Likewise, it is also possible that some subsequence of segments in the ideal signature is unimportant so that it may be acceptable to drop the entire subsequence. We provide some simple annotations to describe these exceptions to the default rule on a per stroke basis:

1. If a certain segment is to be declared mandatory, capitalize it in the ideal signature.

2. If we wish to allow consecutive segments s_i, s_{i+1} to be dropped, interpose a "." between them.

Example: If we wish the signature "dlud" of the last drawing of Figure 2 to match the ideal signature "dlurud" of न , then we would need to allow the fourth and fifth segments to be both dropped from the ideal signature. So we specify the signature of न as "Dlur.uD". This says that the first and last segments are compulsary, while the fourth and fifth segments may both be dropped if necessary. The default rule applies at the other positions.

3.2 Geometric Relationships

Let $e_0, \ldots, e_{m+1}$ denote the segment endpoints of the rectilinear segmentation of the ideal drawing of some stroke S, with $e_k = (X_k, Y_k)$. In the template of S we can specify relationships on the relative positions of these endpoints. Each relationship is specified as $i \,\mathrm{op}\, j$, where the operator op could be any of $=_x$, $\geq_x$, $>_x$, $\gg_x$, $=_y$, $\geq_y$, $>_y$, $\gg_y$, with $0 \leq i, j \leq m+1$. The interpretation is as follows, with ϵ a *tolerance* parameter[3].

$$\begin{array}{rcll}
i =_x j & \equiv & |X_i - X_j| \leq \epsilon & \text{(essentially same x)} \\
i \geq_x j & \equiv & X_i + \epsilon > X_j & \text{(essentially same x or } e_i \text{ to the right of } e_j\text{)} \\
i >_x j & \equiv & X_i > X_j & (e_i \text{ to the right of } e_j) \\
i \gg_x j & \equiv & X_i > X_j + \epsilon & (e_i \text{ well to the right of } e_j)
\end{array}$$

The remaining operators are used to specify relationships on the y coordinate in a similar manner.

3.3 Example:

The template for the Devanagari न may be specified as follows:

न Dlur.uD $4 >_y 0$ $0 \gg_x 3$

The first word gives the ideal signature specified with the approximate matching rules. Following this are the relationships on the segment endpoints. The first relationship $4 >_y 0$ asserts that the 4th endpoint be above the 0th endpoint – this is to distinguish न from म . In म e_0 would be above e_4. The second relationship needs some explanation.

Note that in case the input signature matches the ideal signature only approximately, the geometric relationships specified in the template must be reinterpreted suitably. Suppose for example that an input drawing has the signature "durud". Then the input signature matches the ideal signature "dlurud" of न with the second segment ("l") dropped. In this case the relationships "$4 >_y 0$ $0 \gg_x 3$" must now be interpreted as "$3 >_y 0$ $0 \gg_x 2$".

It will now be seen that the second relationship in the template prevents a ज from being classified as a न . A drawing of ज could also have the signature "durud" – but in this case the 0th endpoint would be to the left of the second endpoint, and hence would not satisfy the (reinterpreted) second relationship of न .

3.4 Completeness

A template allows us to define a stroke as a signature and a conjunction (i.e. an AND) of the geometric relationships. What if we need to express a disjunction (OR) of relationships $c_1, c_2, \ldots, c_k$?

We can achieve this effect since our system allows multiple templates to be specified for a single stroke. So we simply specify k templates, mentioning the relationship c_i in the ith template.

[3] We have fixed ϵ to be one sixth of the longer side of the axis parallel bounding box of the stroke.

It should be clear that in this manner, we can effectively use arbitrary logical combinations of the geometric relationships while defining a stroke to our classifier.

4 Stroke Classification Algorithm

The basic ideas behind stroke classification have already been discussed: in order for a drawing to be identified as a certain stroke, its signature must first match the ideal signature of the stroke (as per Section 3.1). In addition, the segment endpoints of the drawing must satisfy the geometric relationships specified in the template (with reinterpretation as described in Section 3.3).

It is possible that a given drawing matches templates of two strokes, i.e. its signature matches both the ideal signatures, and the endpoints satisfy both sets of geometric relationships. In such case we consider the *match distance*, i.e. the difference in the lengths between the ideal signature and the drawing signature. Whichever stroke has the smaller match distance is declared as the matching stroke.

4.1 Template Expansion (Offline):

Since the stroke templates are fixed during any session, we can expand them offline to reduce matching time. Specifically, given a template such as

न Dlur.uD $4 >_y 0$ $0 \gg_x 3$

we generate all possible signatures consistent with the matching rules as specified in Section 3.1. Further, we also reinterprete the geometric relationships to account for the dropped segments. So for the above template we would generate the following signatures and the associated relationships:

न	DURUD	$3 >_y 0$	$0 \gg_x 2$
न	DLRUD	$3 >_y 0$	$0 \gg_x 2$
	...		
	...		

The above constitutes the *stroke descriptor* set for न . The stroke descriptors are computed offline. For Devanagari, our template expansion process generates about 6500 stroke descriptors. The Stroke Classifier does not use the templates but directly the stroke descriptors themselves. With this, the Stroke Classifier does not need to do any approximate matching – it only checks if the signature of the input drawing *exactly* matches any of the signatures in the stroke descriptors as well as the geometric relationships therein. So the signature matching step can be done in constant time essentially by hashing on the signature.

4.2 Positional cues

In addition to the ideas described above, our classification algorithm also makes use of the *relative position* of different strokes. This is extremely necessary in some cases. For example, consider the stroke representing the half त् , (i.e. the stroke before या in त्या), the stroke commonly called a *rafar* (i.e. the stroke above थ in अर्थ) and the stroke that signals the vowel *ri*, (i.e. the

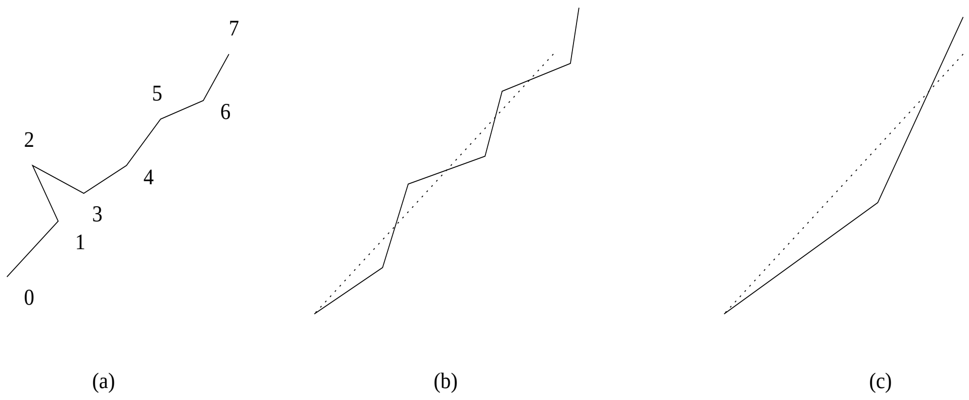

Figure 3: Filtering input noise

stroke under the क in कृपा) These have the same basic shape. But they can be very easily distinguished by noting that the rafar can occur only *above* another stroke, the ri only *below* another stroke, and the half त neither below nor above the preceding stroke.

5 Stroke Parser

Many Devanagari characters are written using combinations of strokes. As mentioned earlier, व overwritten with a short line segment stroke becomes a ब . Similarly, प overwritten with a short line segment stroke becomes ष . This information is encoded in the Stroke Combination Rules. These basically have the form:

ब = व + overstriking line
ष = प + overstriking line
...
...

The Stroke Parser takes as input the stream of strokes constructed by the Stroke Classifier, and merges strokes to form characters as per the stroke combination rules illustrated above. The output of the Stroke Parser is a stream of characters.[4]

6 Filtering

Point Noise: Because of mechanical imperfections in the mouse/pen or because of a shake of the hand, a drawing might contain isolated points that are clearly

[4] It is not entirely clear what is meant by a Devanagari character. Different font/encoding systems in use today appear to have slightly different definitions. For example, in the UNICODE encoding, the short इ matra appears after the consonant, while in many other encodings it appears before. Our program generates a stream of UNICODE that can be printed. A few changes would be needed if we were generate a stream for another encoding. The issues in this are well understood, but they add more complexity to the implementation.

outliers, as point 2 in Figure 3(a). Such points need to be filtered out before the rectilinear segmentation is computed. Our strategy for this is as follows. We declare a point p_i to be potential noise point if the drawing direction abruptly changes because of p_i, and the length of $p_{i-1}p_i$ is small, i.e. smaller than a constant δ which is determined by considering the quality of the mouse/pen etc.[5] When a point in the drawing is found to be potential noise, we simply ignore it. This is fine if the point is actually noise; if the point in fact signals genuine change of direction, then subsequent points will lie in the new direction, and will be further than δ from p_{i-1}, and thus the new direction will get recorded. We note that the "simpler" idea of discarding a point if it is too close to the previous point runs the danger of shaving off corners in the drawing. Hence the need for our more complex condition.

Slope Discretization Problems: If a writer is attempting to draw a line with slope θ, it can only be drawn within a few degrees of θ. This is serious problem when $\theta = 45°$ or odd multiples of $45°$. Thus, for the segment in Figure 3(b) our algorithm would interprete the small changes of slope around $45°$ and declare the signature to be "rururu". It is much better to have the simpler signature "ru" which corresponds to the similar looking drawing of Figure 3(c). Our filtering procedure in fact does this. The key question, of course, is how to ensure that drawings with genuine slope changes (e.g. if a true staircase pattern were being drawn) are not assigned simplified signatures. Our procedure does this by examining the actual slopes of the adjacent displacements to determine if they fall in a small enough angular interval before simplifying the signature.

7 Experience

We have implemented our system in JAVA. The handwritten input text is provided using a standard mouse, or a touchpad, or a *pen mouse* (also called a stylus or digitizer tablet). There are 2 programs, the offline template expander, and the on-line recognition program (which essentially consists of the 3 modules shown in the upper half of Figure 1).

Figure 4 shows a snapshot of the recognition program in action. The handwritten text is in the lowermost panel, the text as recognized by the program is shown in the panel above that. As can be seen, many characters (e.g. भ म र) have been recognized in spite of being drawn in different ways. The program works fast – the recognition is instantaneous.

We have substantially tested our system with a corpus containing isolated letter sequences as well as full sentences, written by many writers. Sometimes the writers need some amount of training in order to get used to the manner in which our system expects letters to be drawn. We observed this period to be fairly small.

[5] If δ is too small, we will not filter out much noise. If it is too large, then we will filter out genuine direction changes. Our implementation uses a statically configured delta which can be tuned by the user to suit the hardware platform, especially the mouse.

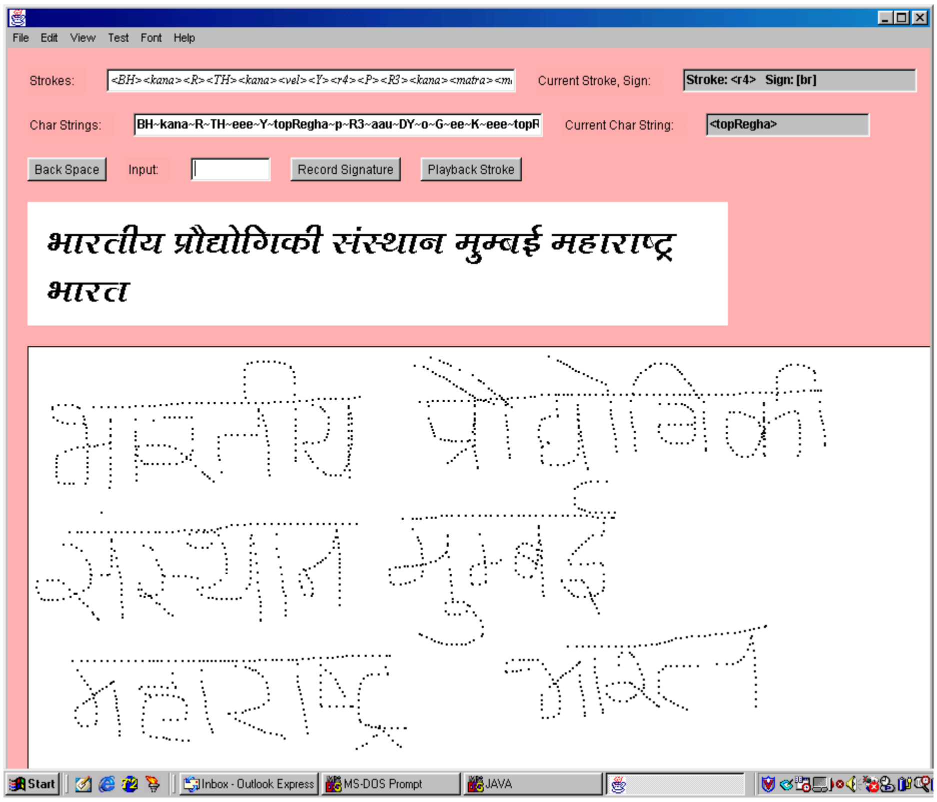

Figure 4: Our Program in Action

8 Concluding Remarks

How to accomodate distortion/variation that happens due to stylistic variation or during rapid writing, is the central question that any recognition program must address. We presented a model for this distortion process (Section 3.1); and a notation to describe how a Devanagari stroke can get distorted during writing. Because of the underlying model, each stroke (i.e. its *template*), can be described very concisely. Other approaches, e.g. those based on neural networks, do not attempt to understand this distortion process. This we feel is a major strength of our work.

The idea of using pen direction as a classification feature is not new. However, we find it interesting that it suffices to consider only the horizontal and vertical directions, i.e. discretize the direction to just 4 possibilities (left, right, up, down). A possible explanation of this success is that Devanagari strokes have very strong movements in this direction.[6]

Finally, we would like to mention as noteworthy our precomputation procedure. Because of this, we can solve a reasonably complex string matching problem in essentially constant time.

References

[1] V. Bansal and R. M. K. Sinha. Partitioning and searching dictionary for correction of optically read devanagari character strings. In *Fifth International Conference on Document Analysis and Recognition*, 1999.

[2] R. S. R. Kunte and R. D. S. Samuel. Writing Pad for Indian Languages-On-Line Character Recognition System for Handwritten Characters/Script with Bilingual Facility Employing Neural Classifiers and Wavelet Features. In *Knowledge Based Computer Systems*, pages 1–12, 2000.

[3] S. Manke, M. Finke, and A. Waibel. NPen++: A Writer Independent Large Vocabulary On-Line Cursive Handwriting Recognition System. In *Proceedings of the International Conference on Document Analysis and Recognition*, 1995.

[6] If a script has strong movements in other directions, perhaps we would need to choose those directions.