Optical Character Recognition for Typeset Mathematics

Benjamin P. Berman
Richard J. Fateman*
Computer Science Division, EECS Department
University of California at Berkeley

Abstract

There is a wealth of mathematical knowledge that could be potentially very useful in many computational applications, but is not available in electronic form. This knowledge comes in the form of mechanically typeset books and journals going back more than a hundred years. Besides these older sources, there are a great many current publications, filled with useful mathematical information, which are difficult if not impossible to obtain in electronic form. What we would like to do is extract character information from these documents, which could then be passed to higher-level parsing routines for further extraction of mathematical content (or any other useful 2-dimensional semantic content). Unfortunately, current commercial OCR (optical character recognition) software packages are quite unable to handle mathematical formulas, since their algorithms at all levels use heuristics developed for other document styles[1]. We are concerned with the development of OCR methods that *are* able to handle this specialized task of mathematical expression recognition.

1 Introduction

This work is a part of a larger project [5] that intends to encode, for use by computer algebra systems, integral tables and other documents currently available in hardcopy only. We concentrate here on the surprisingly "unsolved" problems of OCR for mathematical equations.

There are quite impressive (rapid, accurate, versatile, inexpensive) systems of hardware and software for scanning of simple text. Rates of thousands of words per minute, accuracy of 99% or higher, and even automated page-feeding, are available in systems with a retail cost of under \$5,000.00 today. Within this budget one can buy an excellent (hardware) page scanner, a high-speed personal computer (PC or Macintosh), high-capacity memory, disk and tape systems, and one of several competing commercial OCR software programs.

Disappointingly, the vital heuristics that work so well on straight text, multi-column printing, tabular displays, and business graphics fail dramatically on mathematical text. Why? Equations prominently feature variations in font size, changes of base-line, quite different spelling digraph frequencies, and special characters. We naively thought that Greek and Symbol-font characters would be barriers: they are not. In fact if they occurred in straight text, standard OCR would handle them quite well, just as the presence of boldface, italics, variations in font size and typeface, and numerous other complications are not really problematical. That is, as long as the material is relatively straight text as one might get in a business report, newspaper page, or novel, these kinds of systems perform well. Arrange perfectly formed characters in equations, however, and the recognition falls dramatically to 10% or less. Thus we found ourselves forced to write our own OCR software.

2 Objectives

Our objective for this part of the project is to start from a scanned (black/white) binary representation of the mathematical expression, a "bitmap", and end up with a set of pairs: (glyphs,location). A glyph is a representation of a character that includes font information as well as the alphanumeric "value". Later processing [10] deals with the "transcription" process that relates text like $x + y^2$ to an interpretation such as `$x+y^2$` in TeX or `(plus x (power y 2))` in Lisp.

The OCR software generally available focuses primarily on 1-dimensional text. The usual "location" information is merely conveyed through the sequence of the characters in a simple 1-dimensional stream of ASCII characters in a file. In some cases, *some* amount of spacing and columnar information may be included in this type of representation, however. By contrast, for us, a stream will simply not have enough information: both expressions 2^p and $2p$ may look like `2p`. We not only need the 1-dimensional information that p *follows* 2, but also the 2-dimensional information that p *is above* 2.

OCR "omnifont" recognition software usually does not distinguish letters on the basis of fonts sizes or types. That is, an a, an `a`, or an a are considered identical. This kind

---

*This work was supported in part by NSF Grant number CCR-9214963 and by NSF Infrastructure Grant number CDA-8722788. It is an elaboration and extension of a paper accepted for ISSAC94 (Oxford, UK, July, 1994).

[1] They are however, remarkably capable in *other* domains. A useful general reference is the paper by Bokser [1], which discusses in some detail the Calera scheme; other competing commercial systems include Caere and Xerox Imaging Systems.

of of size/font information is useful for our parser to disambiguate mathematical expressions. The size information is used to compute the baseline of a character (in the example of 2^p above, size information would be used to figure out the baseline of the p, which is necessary to know that it is *above* the 2). Without such observations one might be confused. The baselines for p and d are pd are at the same height. If one were to judge by bounding boxes, it almost appears that we have a superscript situation, since the first character is on the face of it, quite a bit lower than the second.

Both size and font information provide other useful hints to the parser, as the use of large letters, small letters, boldface, italics, and different fonts can provide additional semantics. Non-obviously, we have found references (e.g. [4]) using different fonts and sizing for numbers depending upon their use as (for example) page numbers, equation labels, reference numbers, base-line coefficients, exponents, or footnotes[2].

3 Science vs. Engineering

There is a temptation to reduce the set of tasks to neat modules: scanning and segmentation of a page into "zones"; recognition of bits within zones into characters and lines; tokenizing of character-sequences "boxes"; parsing of boxes to mathematics expressed in Lisp. Each of the modules works with neatly specified input–output relations so that the total recognition process is achieved by merely hooking these together.

Unfortunately, such a modular decomposition generally requires a level of pre-cursor perfection: namely, the design is based on the assumption that the inputs to each module are correct. Only then will the output also be correct. Since one normally one tries to produce a robust program—one that can deal with erroneous input, working around this decomposition can be difficult. In particular, it requires considerable effort to polish a "perfect" program (using perfect data) and make it robust. We estimate that under the assumption of perfectly clean input, only 25% of the code we have written to date would be needed; perhaps this would constitute 5% of a more fully bullet-proofed program. Such a program would still make mistakes on some inputs.

Here we describe a "perfect" input scenario as we have constructed it for testing purposes.

3.1 Fake input

A subset of our target domain is easily available. Consider algebraic expressions in some computer algebra system. Here we use the syntax for Macsyma:

```
'INTEGRATE(A*X^2+B,X) = A*X^3/3+B*X
```

Macsyma (which is perfectly capable of computing the right-hand side of that equation) along with most commercial systems can render this expression in TEX form. Macsyma 418 provides this, explicitly:

```
$$ \int {a\,x^{2}+b}{\,dx}={{a\,x^{3}}\over{3}}
       +b\,x $$
```

[2] This font information is really only useful in the context of a *particular* publication, as different fonts/sizing will have different uses in different books. For us, this is still very useful as one target of our interest is large volumes of integral tables. These are large enough that it makes sense to learn a new set of conventions for each new book.

which prints as

$$\int a\,x^{2}+b\,dx=\frac{a\,x^{3}}{3}+b\,x$$

when run through appropriate software. One might then say we can get "perfect" input by scanning this equation, and running it through our programs. While this is not a bad idea, it is not perfect. We can, entirely without paper, produce a much better input. In particular, we can convert the TEX to PostScript, and the PostScript to a bitmap, without ever printing the bitmap. We can then feed the bitmap into our recognition program, being assured that

- There is no noise on the "page" caused by defects in the paper, or errors caused by over/under adhesion of ink (toner, etc).
- The characters are all fully formed and correct to the nearest bit, to whatever detail we have specified (independent of any printer resolution).
- The scanning introduces no noise: In particular, the following problems do not occur: misalignment of paper, dirt on the page or in the scanning mechanism, over-scanning of margins, bindings, etc.
- The fonts are all known, having been produced by the computer system in the first place. Computer scaling of fonts may have been done to produce various sizes, but this has been done systematically.
- The syntax is restricted to that which is known to the computer algebra system in the first place. The generality of mathematics notation cannot be conveniently constructed in the computer systems, nor will their formatting into TEX necessarily echo all that is possible.

3.2 Fake recognition

Carefully typeset input, given sufficient resolution, will not present characters touching (at least not haphazardly). The characters will be aligned perfectly on a grid. While there may be some variation in the mapping of characters to a grid (if the resolution of the character descriptions, the spacing and the bitmap are subject to slight rounding affects), the variations will be quite small. A character can be learned from exactly one example. Indeed, the "learning" of characters can be done entirely by reference to computable tables of bit-maps for the characters. Any *ad hoc* scheme that can accurately identify and distinguish each character from the others that might occur, could be used.

One scheme we found rather accurate was to scale each character to a 5 by 5 grid, and compute the gray-value in each of the 25 sections. We found that this, (along with any identification scheme that includes scaling) to necessarily lose some information: for example, the number 0 and the letters O and o are almost indistinguishable. We have fixed this in part by using some absolute-size data. We have considered eliminating scaling; if we did so, we would recognize as distinct each size of character (thus 9, 10, 11, and 12 point x would all be different and identified by size), at some cost in overall complexity of the program to process this information further.

Running our simple recognizer on perfect data we still had to address some problems: we used different criteria to recognize a horizontal line; and the characters with separate pieces ("i", "=" "!") were recognized as separate parts. This is not entirely comfortable.

Furthermore, the typeset characteristics of some of glyphs varies substantially with historical and aesthetic criteria. One source of material [4] sets the "+" with a descender as low as the tail of a "p". The standard size of font used for TeX mathematics has a smaller "+". Accomodating such differences in one program can be troublesome.

3.3 Fake grouping

Given the perfection of the character recognition, as well as the regularity of the spacing, it becomes easier to group characters into tokens (perhaps using rather more fragile algorithms than would otherwise be advisable). Grouping digits 1, 2, 3 into the number "123", and the letters s, i, n, x into two tokens, "sin" and "x" becomes much easier. Note that a complete list of mathematical symbols (and perhaps their translation into other alphabets) is available on the computer systems. This makes it possible in principle to typeset and re-recognize "Si" as either the product of "S" and "i" or perhaps as the name for a tabulated function (The Sine Integral).

3.4 Fake parsing

Knowing that we are limited to parsing expressions that can be produced by (say) Macsyma means we do not need to make sense of *ad hoc* notations or abbreviations, or even some conventional notations that do not fit easily into the current repertoire of computer algebra systems. Conditional notations, reference markings, footnotes, and "comments" are not produced. We are also guaranteed that there exists (at least) one valid Macsyma expression corresponding to the display we are parsing.

3.5 A more comprehensive theory

Probably the most fully-formed and described theory for the recognition of equations is one that emphasizes the impact of noise, in work by P. Chou [3]. This paper describes a 2-D parser which runs on a bit map produced by eqn and then peppered with noise. Although there is certainly a strong prospect of further achievement along these lines, (using a stochastic context-free grammar, and treating recognition as inversion of a signal processing algorithm in the presence of noise), the apparent cost of the demonstrated algorithm is formidable.

In particular note that the production rules of the grammar must map from the grammar's "start" symbol down to a rectangular array of pixels. In spite of its apparent generality, the algorithm described is actually far more restrictive in some ways than we can afford. For example two factors can be horizontally concatenated only if each has the same point size and the baselines differ by at most one pixel.

We back off from this admirably general long-term approach to explore what we actually must do to get OCR to work on examples of interest in the short term.

4 General method

The two basic phases of building a successful OCR process are the learning phase and the recognition phase. While there may be a need for further adaptation during the recognition phase ("life-long" learning), and even further modification in the transcription phase, it still seems appropriate to consider the two phases (learning, recognition) as generally distinct.

4.1 Learning

In learning, we start with a corpus of pages representative of the book we wish to read. For our integral table task, this should probably also include the "table of notations" to try to provide a complete coverage of sparsely occurring symbols. These are the steps:

1. Collect and cluster objects: Collect all "connected objects" from the learning pages. A "connected" object is any set of ON pixels that are adjacent to one another and separated, by OFF pixels, from all other ON areas. These objects are not ideal forms, but are affected by noise: dirt, ink smears, paper texture, scanner errors etc.

Various heuristics can be used to eliminate from the collection those items that are specks of dirt (too small) or pictures of book-binding creases or page edges (too large). Characters that have several parts (i, j, = %) can be artificially grouped together.

The collection of images is grouped into clusters of similar images, where similarity is based on the same criteria that we will use to determine similarity during the recognition phase; these criteria are discussed in following sections.

2. "Identify" these clusters: This is where human interaction will be the most extensive. Upon being presented with a cluster, a human must:

- Decide that the cluster is a good representation of a particular character (e.g. "These are all italic p's"), and label the group accordingly.
- Decide if certain members of the group are unacceptable representatives and remove or edit them.
- Consider splitting a group into several subgroups and labeling each separately.

3. Generate "model" representations for each of the groups: This is a key function of the learning phase. We generate a kind of average over the group to summarize the qualities of that character. This *model* character for the group will be used to represent the group during the recognition phase. Getting a good representation of the group is of extreme importance, as all comparisons down the line will be dependent on these *models*[3].

Note that we are not requiring all instances of a particular letter to fall within a single cluster. For example, we may have two or more clusters for the number *2*: one which has the loop closed, one which has the loop closed and the center filled in, one that has been only partially collected because of a break in one of the strokes. These clusters should not be considered to be objects grouped according to their symbolic meanings. We may wish to distinguish amongst various kinds of "*2*"s in a given publication, or we may identify each as simply "2".

4.2 Recognition

Given a learned set of models, we are ready to proceed to the recognition phase, where we attempt to resolve each character on the page into the correct category. The general steps involved are outlined below:

1. Segmentation: Just as in the learning phase, we collect a "connected" set of ON pixels.

[3] Keeping all the examples and not averaging them, is an alternative tactic worth consideration.

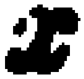

Figure 1: A disconnected x.

Figure 2: An over-connected x^p.

2. Remove obvious noise: Again, heuristically eliminate dirt, etc.

3. Primary Recognition: For each of these connected objects, try to pair it with one of the "models":

- **Heuristic Elimination:** We can eliminate many, if not most, of the possible model choices by very simple (and, consequently, computationally cheap) algorithms. This will vastly cut down the time spent in the primary recognition phase.
- **Comparison to Models:** This is the heart of our system. In this phase, we identify most of the objects by finding the model that is closest to it in appearance. The method we use to make these measurements is described in the next section.

As a result of this phase, we believe most clean characters will be recognized. Characters that are broken or stuck together (and haven't been learned as an assemblage) will be passed through to the next level. It is plausible that some heuristic information will also be available from the lower-level scanning process, if needed. This might include, for example, the slope of horizontal "divide-bar" lines on the page, or the estimation of base-line from characters recently scanned.

4. Secondary Recognition: When we began, we had a full page of objects to recognize. As we recognized or discarded objects, we erased them from this "global" structure. What is left on it now are objects that could not be recognized by the Primary recognition phase. At this point, there are 3 kinds of objects remaining on the page:

1. Disconnected characters: These are characters that have been split up into 2 or more separate objects because they were disconnected on the page. See figure 1 illustrating the 400 dots/inch resolution on an example from [4].
2. "Over-connected" characters: This happens when 2 or more characters are smashed into one object because they were touching at some point. See figure 2.
3. Unrecognizable characters and/or garbage: These are items that can not, for whatever reason, be recognized.

For the third item above, we ask the user to classify the object for us. For the first two items, we use more sophisticated (and relatively costly) variations of the same algorithms that we used for the Primary Recognition stage.

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

Table 1: A sample bitmap

1	0	1	2
$\sqrt{2}$	1	1	$\sqrt{2}$
2	1	0	1
$\sqrt{5}$	$\sqrt{2}$	1	$\sqrt{2}$

Table 2: The distance transform for Table 1

5 Distance Transforms and the Hausdorff distance

How does one tell if two images are "nearly the same"? There are numerous papers on the topic, and one can easily come up with novel (but perhaps not very reliable) measurements. We experimented with a number of ideas. One can consider using summary statistical measures; e.g. two images are similar if the have the same ratio of black to white pixels in each grid region resulting from an n by m subdivision of the "character space"[4]; morphological reductions: compute the "skeleton" of characters before trying to characterize them by open/closed loops, etc., or compare bitmaps directly as described below.

An emerging winning strategy seemed to be to try a relatively cheap way to deal with the easy cases (this will often succeed), and reserve more expensive techniques for the difficult cases. Fortunately it seems that the difficult cases can often be identified by the cheap technique. That is, the cheap technique identifies these problems as ambiguous characters).

A set of more elaborate techniques boil down to comparing bitmaps directly, and may succeed for us especially if we are willing to learn images at different sizes/fonts separately. We discuss one such set of techniques in substantial detail. In the remainder of this section we are somewhat more formal in describing the distance measurements between objects in 2-dimensional space.

5.1 The Voronoi Surface

The first important concept is the Voronoi surface [7], or its rasterized equivalent, the distance transform [11]. Given a space $\mathcal{A}$ and an image I within that space, the Voronoi surface assigns to every point $a \in \mathcal{A}$ the distance from a to the closest point in I. The distance transform is a discrete rasterized approximation of this concept. Given an array $\mathcal{A}$ and a subset of that set, $\mathcal{A}_d$, which contains the ON pixels within $\mathcal{A}$, the distance transform is an array of the same size as $\mathcal{A}$ with each value being the distance[5] from the nearest point in $\mathcal{A}_d$. So for the bitmap in Table 1, the distance transform array with respect to the ON (=1) pixels is illustrated in Table 2.

5.2 The Hausdorff distance

The methods that we will use for recognition are based upon this idea of the Voronoi surface. In particular, we use a metric known as the *Hausdorff distance* [7]. Whereas the

[4] more on this simple scheme later.

[5] We are using the Euclidean norm, i.e. the distance between points a and b is $\sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$

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

Table 3: Another bitmap

Voronoi surface and distance transform are a measure of distance from an entire *space* to an image, the Hausdorff distance is a measure of the distance from one *image* to another *image*. In particular, the distance from $\mathcal{A}$ to $\mathcal{B}$ as:

$$h(\mathcal{A},\mathcal{B}) = \max_{a\in\mathcal{A}} \min_{b\in\mathcal{B}} \|a-b\| \tag{1}$$

where $\|a-b\|$ refers to some measure of distance (e.g. Euclidean norm) between two points a and b. The intuitive idea is that there is no point in $\mathcal{A}$ that is more than distance $h(\mathcal{A},\mathcal{B})$ away from some point in $\mathcal{B}$. This does not specify the distance from $\mathcal{B}$ to $\mathcal{A}$ ...since there could be points in $\mathcal{B}$ that are indefinitely far from anything in $\mathcal{A}$. Along with Huttenlocher, [7] we will refer to this asymmetric version as the *directed* Hausdorff distance.

If we want a more complete description of the distance between $\mathcal{A}$ and $\mathcal{B}$, we can simply take the greater of the distances from $\mathcal{A}$ to $\mathcal{B}$ and from $\mathcal{B}$ to $\mathcal{A}$. We refer to this measure as the *complete* Hausdorff distance. Formally,

$$H(\mathcal{A},\mathcal{B}) = \max\left(h(a,b), h(b,a)\right). \tag{2}$$

We have found that a useful variant of this measure is to exchange the $\min_{b\in\mathcal{B}}$ term in Equation 1 with something like the kth minimum. This tends to allow for a small bit of tolerance towards slight disparities. When using this method, we are, in effect, asking whether the two images are *mostly* identical, rather than *completely* identical.

The k-complete Hausdorff distance forms the major test for determining if an unidentified character is close to a model. Since this is a rather costly technique, we first need efficient techniques for eliminating consideration of "distant" models without the cost of a full comparison. And then when it seems we must use the Hausdorff computation, a rapid implementation is required.

5.3 Efficient algorithms for distance transforms and Hausdorff distances

One way to compute the directed Hausdorff distance from an object $\mathcal{A}$ to an object $\mathcal{B}$ is to compute the distance transform of $\mathcal{B}$ (within an area at least as large as $\mathcal{A}$) and take the max of all the points in this space which would be covered by the $\mathcal{A}$ object were the two to be superimposed. If we desired, for example, the directed Hausdorff distance from an image $\mathcal{A}$ shown in Table 3 to the image $\mathcal{B}$ shown in Table 1, for example, we could use the distance transform for $\mathcal{B}$, shown in Table 2. When we superimpose $\mathcal{A}$ and the distance transform for $\mathcal{B}$, we see that the distances from the points of $\mathcal{A}$ to the image $\mathcal{B}$ are $\sqrt{5}$ and 1. The maximum of these is $\sqrt{5}$ and this is our directed Hausdorff distance.

While this is not necessarily the most efficient way to compute the directed Hausdorff distance for a single comparison, it is very efficient if you wish to compare many $\mathcal{A}$s to one $\mathcal{B}$, as you only need to compute the distance transform for $\mathcal{B}$ once and can use it over and over again. This turns out to be very well suited to our application as we generally want to compare one particular unidentified character to each of several plausible model characters.

There are at least a few good ways to compute the distance transform in an efficient manner. The first [2] uses several serial passes of the data to propagate out estimations [6] of distance from ON pixels, where the distance is 0. Another method [11] makes separate vertical and horizontal sweeps of the data to deduce its distances. One reason that we decided upon the latter algorithm is that it is capable of yielding exact Euclidian distance values, whereas the former can only yield approximations. The second reason is that it can take advantage of a *threshold* value.

The threshold value (τ) is the maximum Hausdorff distance of interest to us. Above τ, distances need not be computed because they are so far away that these comparisons can be eliminated. Since we are only interested in examining matches that are very close in nature, we can choose a very low threshold. This low threshold value can greatly increase the efficiency of the algorithm in [11], whereas the other algorithm [2] is not as well suited to this particular constraint.

We refer the reader to the references for details of the algorithms.

6 The Learning Phase: Details

The main goal for the learning phase is to develop a set of model characters for a corpus[7]. While we want to keep this phase as automated as possible, there will necessarily be some human instruction to correctly identify the various symbols.

6.1 Collecting objects from the page

Presumably, the training data will come in the same form as the data to be recognized: pages full of equations and text. The first thing that we need to do is collect separate characters off these pages by scanning for *connected* segments as we have described in previous sections. It is possible (although we do not do so now) to provide additional information such as: estimated "base line" for the character, region of the page (header, footer, margins, body) where the character was collected, or estimate of noise in this region.

6.2 Clustering and labeling the objects

Next we cluster the collections into sets of objects with similar characteristics. Besides using size as a criterion for clustering, we will use many of the same criteria that we use to judge similarity when trying to *recognize* objects. In particular, the Hausdorff distance and its notion of "distance" between two objects will be one such criterion.

At this point, some human help is needed. Since this is the data that the recognizer will use, we want to make sure that it's very clean. For this reason, all the data should be inspected by a human. We have developed tools for the user to label the sets ("This is Italic lower-case p, exponent size"), to clean up (pixel-edit) particular instances, remove mis-classified glyphs from sets, break a set into two new sets, merge data in separate sets, or discard sets.

[6] the accuracy of the estimation is inversely correlated with the speed of the algorithm

[7] Some vendors of commercial OCR systems would like you to believe that there is a magic technique that takes care of all learning problems, namely "neural networks". We may eventually consider some version of neural net for learning, but we view it more as a last resort not a panacea.

After this human phase is finished, we should have clean, labeled sets of characters for the production of model characters and their analysis.

6.3 Producing *model* characters

The idea is this: Take a set of many instances of the same character, and create a representation for them that somehow captures the important characteristics of the entire training set.

If you think of the characters in terms of the area they cover within some space, the important features for that character are areas of that space which are covered by many instances of that character. This is the idea that we will use to produce our model characters.

This idea is realized by compiling the training set into a *pile-of-characters* array (POC). This is a 2-dimensional array of integers with X-Y extent large enough to cover every bitmap in the training set. Every spot in the array is initially zero. For each member of the training set, we go through a process in which we superimpose it over the POC array, incrementing each pixel location in the POC that is covered by a 1-pixel in the new instance of the character. As we get more and more characters "piled" into the POC in this way, "hot spots" begin to emerge to represent the likely positions of 1-pixels in the set of characters as a whole. Setting some percentage threshold gives us a bitmap *model* representative of the training set.

Actually, there is another complication: we cannot just superimpose each new character into the POC. Our collected representations of these objects are not perfectly aligned with each other by (say) placing their lower-left corners on top of one another. There may actually be a better match if you shift them over a few pixels in one direction or another. For this reason, we must search to find the **right** place to superimpose them onto the POC, else we might deform our POC and decrease our chances to extract a good representation for the character. We find the **right** place to do the superposition by again using the Hausdorff distance. As we compile the individual instances into the POC, we keep an *in progress* version of the model character being created. This *in progress* version is initially just the first character in the training set. Next, we try to find the best place to overlay the next member of the training set with respect to this new on-line model. We do this by translating the training character within some range around the model. At each translation, we take the Hausdorff distance between the model and the translated image. We choose the smallest of these Hausdorff measurements as the best place to lay the model. After doing this for each new item in the training set, the characters are, in effect, piled into a "neat" stack and the emerging model character is perfected. We then trim it back to an appropriate height and width.

There are a variety of heuristically chosen tolerances required for complete formalization of this algorithm.

7 The Recognition Phase: Details

The main goal of the recognition phase is to match each object on the page to one of the model characters created in the learning phase, or to reject it as being unrecognizable. The different stages of this procedure are given below.

7.1 Segmentation

Segmentation is done in much the same way as it was in the learning phase. We progressively scan through the page collecting all of the *connected* objects. Not all are characters: Some of them may be random pieces of dirt, along with portions of characters that are disconnected or multiple characters that are linked together. We refer to any such collected object as an *image*, which we compare to our standard *models*.

7.2 Elimination

This is the initial stage of classifying each of the connected objects. The goal here is to eliminate any prospective matches to models that are particularly unlikely using simple, almost instantaneous, methods.

1. **Throw away dirt specks, especially in unexpected areas.** Just as we have done in all other segmentation processes, we can eliminate dirt and other interference by simply throwing out any image that is radically smaller (dirt) or radically larger (sometimes caused by the bindings of books) than *any* of the model characters. In our experience, punctuation, the dots on letters i and j, primes, and similar letter pieces are larger than dirt and therefore won't get discarded here.

2. **Elimination based on size.** Now we will go through each of the model characters, trying to eliminate as many as possible from further consideration. The first basis upon which we will do this elimination is size mismatch between a prospective model and the image. As mentioned previously, we want to classify different font sizes as different symbols. We can, therefore, eliminate a model if it is not within a certain size range around the image. This, the simplest of all the methods used, can eliminate a great many possibilities for classification with almost no computational effort.

3. **Low resolution pass.** Once we have eliminated on the basis of size, we can optionally go through a kind of low-resolution Hausdorff calculation. This means that instead of taking the maximum of all the a points in the equation $h(\mathcal{A},\mathcal{B}) = \max_{a\in\mathcal{A}} \min_{b\in\mathcal{B}} \|a-b\|$, we take only the maximum over some small fraction of the a points (This fraction could be every fifth pixel in the image, for example. Then only 1/25th of the work is needed.) The efficacy of this approximation remains to be studied.

7.3 Primary Recognition

At this point, we have some subset of the model characters which resemble the image. The goal of this stage is to, if possible, classify the images as one of these models by a k-complete (two-way) Hausdorff measurement to each prospective classification. If one, and only one, of these classifications is within some threshold of what we consider to be a *correct* match, we have successfully classified the image and may move on to the next image.

There are a few ways in which we optimize this process to improve its efficiency a bit. On a very low level, the algorithm we use to compute the directed Hausdorff distance will return immediately (with a "too distant to beat threshold" failure) as soon as this is detected. Since the complete Hausdorff distance is computed in two stages, failure in the first direction (too many points too great a distance from the model to the image) naturally causes a failure to match without requiring an examination of the reverse direction.

In a more "global" sense, there are a variety of other optimizations. The principal one is this: If we decide that a particular model is not a good match for a particular image (the Hausdorff distance is above some threshold), it is possible to eliminate other models that are close to the rejected model. Let's suppose, for example, that we are trying to match the image against the character "d" and the Hausdorff distance between the model "d" and the model "c" is 3. Let's also suppose that the threshold distance value is 2, i.e. anything with k-complete Hausdorff distance above 2 can be safely rejected. If the Hausdorff distance between the *image* and "d" is 10, we can eliminate not only "d" as a possibility, but, by a kind of triangle inequality, conclude that the image differs from "c" by at least 7 and eliminate it, as well (because 7 is greater than 2).

7.4 Secondary Recognition

Now that we have picked off the page all recognizable characters and stray pieces of dirt, we must deal with other noise-induced phenomena, especially with old, poorly printed, or photocopied material. What is left on the page after elimination and primary recognition (aside from objects that we haven't learned and thus *can't* recognize) are glyphs that have been broken apart or joined together with other images.

The problem of interpreting these kinds of objects fits in nicely with the notion of the "directed" Hausdorff distance. Remember that this metric gives you a "one-way" measurement of distance between two objects. Consider the directed Hausdorff distance from the graphical figure $\ominus$ to the figure $\oplus$. This distance is 0 because points of $\ominus$ are all 0 distance from $\oplus$. In the other direction, from $\oplus$ to $\ominus$ there are points on $\oplus$ that are some distance away from $\ominus$ (specifically the points on the vertical bar). Thus this has a non-zero distance.

For the problem of *disconnected* images, we can use the directed Hausdorff *from* the disconnected fragment *to* the model. Let's say that the symbol "$\bowtie$" becomes disconnected into the fragments "$\triangleright$" and "$\triangleleft$". When we get to the fragment "$\triangleright$" on the page, we check the directed Hausdorff from "$\triangleright$" to "$\bowtie$" and find that it is a perfect match. Since we have found a *portion* of the object "$\bowtie$", we hypothesize this symbol as the correct one and proceed to check if the other fragments exist. We check other nearby images on the page that would be covered by this new hypothesized model. Since "$\triangleleft$" falls within this range, we take the one way Hausdorff from "$\triangleleft$" to "$\bowtie$". Again, a perfect match. We do this with as many fragments as can be found resting under the hypothesized symbol in this way. A non-obvious aspect to this process is that we must try translating the prospective model all around the object on the page, because the disconnected fragment could have come from any area of the model.

The solution to the problem of *over-connected* images works in much the same way. We use the reverse Hausdorff direction, however, from the model to the over-connected images. Much like we did in the opposite case, we must translate the model all around the image as it could appear in any location *within* the over-connected image. Let's say, for example, that the two characters "o" and "e" were smashed together on the page to form the over-connected object "œ". Eventually, we will check the directed Hausdorff distance from "e" to "œ" and we will realize that the result is a distance of 0. At this point, we can confidently recognize this particular part of the object as an "e". After going through some more models we will encounter the same situation when we check the directed Hausdorff distance from "o" to "œ" and find out that we can recognize that area as an "o". We simply go through our entire list of models finding out which ones exist as part of one of these over-connected objects.

7.5 Human Intervention

Anything that we can't get in any other way will need attention from a human assistant. When the recognizer fails in this way it can trigger a temporary return to the learning phase.

7.6 Are Typeset Results Correct?

Correctness, and other questions having to do with the content of the integral tables themselves, are discussed at length in a draft document [6]. Since a referee raised the particular issue of correctness, we should point out that we do not take for granted that the formulas in the tables of integrals are error-free. Quite the contrary. Substitution of numbers for parameters and numerical integration can form a partial confirmation, [8] but this too may be inadequate—the prospect for errors with regard to validity of domains is also substantial, and a particular weak point with computer algebra systems today. Only with *in*definite integral formulas is there a prospect of checking by differentiating the result. Even this tends to present difficulties because results may appear to be drastically different. Testing to see if the difference of two expressions is zero, is sometimes challenging. Even so, confirmed "equivalent"results may not have the same domains, and thereby be non-equivalent.

8 Conclusion and further work

We have outlined our current directions in optical character recognition. To see if our current design and implementation can achieve satisfactorily high speeds and sufficient accuracy remains to be demonstrated. We speculate that, even with all of the programming being done on a Sparc 1+ workstation in (compiled) Lisp, that adequate rates of scanning for our purposes (a few characters per second) should be achievable. We do not require the high speeds of commercial text scanning, but we do require more specialized recognition. Other specialized tasks (other than equations) may yield to the same general approach. Joined with some graph-type recognition, certain types of chemical and physical diagrams could be tackled as well.

Since beginning this work in the Spring of 1993, we have found on-going related activity at Xerox's Palo Alto Research Center by Dennis Arnon, Phil Chou, and Gary Kopec [3], [9]. The theoretical framework provided by this work, viewing OCR as a signal processing (decoding) task, identifies equation recognition as one special case among a number of OCR tasks. Rather than segmenting the recognition from the parsing as we have, the Xerox PARC group is looking at these in a more tightly connected fashion. We look forward to working with this group to translate theory into practice, and developing highly accurate and efficient OCR equation-parsing programs.

At the time we are revising this paper (June, 1994), we have largely set aside the Hausdorff algorithms in this paper, using instead a much simpler recognizer initially written by

$$\int \frac{x^p - x^q}{x-1} \frac{dx}{x+r} = \frac{\pi}{1+r}\left(\frac{r^p - Cos\, p\pi}{Sin\, p\pi} - \frac{r^q - Cos\, q\pi}{Sin\, q\pi}\right)$$

Figure 3: A moderately complicated equation.

Richard Fateman in the summer of 1993. It was improved and generalized somewhat based on ideas in Mindy Bokser's paper [1] in May, 1993. Principally, she suggests that a recognizer based on simple properties (a 16 by 16 grid of gray-scale values), and closeness based on neural network training, would work fine. Since we are not dealing with "omnifont" documents, we chose, somewhat arbitrarily, a 5 by 5 grid, and used for clustering, simply Euclidean distance between vectors of gray-scales.

To get an idea of the time this all takes, let us assume that prior to our looking at an equation, the page processing has identified a "zone" containing a single equation from [4] See figure 3. This was scanned at 200 dpi, and slightly cleaned up so that our naive recognition would work. Cleaning constituted breaking connected glyphs in *Sin* and *Cos*; we also removed the dots above the i's but that could have been handled other ways, too. Finally, we removed the notation for the two condition that $p^2 < 1$ and and $q^2 < 1$. These caused no problems for the character recognition, but the arrangement (one above the other inside large square brackets) was not one that our parser can handle right now.

The time to changle this bitmap into a Lisp expression

```
 (= ((Integral NIL NIL)
 (* (* (* (^ x p) (- (^ x q)))
       (^ (* x (- 1)) -1))
    (* 1 (^ (+ x r) -1)))
 x)
(* (* pi (^ (+ 1 r) -1))
   (* (* (* (^ r p) (- (Cos (* p pi))))
         (^ (Sin (* p pi)) -1))
      (- (* (* (^ r q) (- (Cos (* q pi))))
   (^ (Sin (* q pi)) -1))))))
```

was about 10 seconds, using a Sun Sparc 1+ computer.

Reading from a file (produced by a PC and scanning device) into a Lisp array of bits took about .25 seconds. This produced a 663 by 77 bitmap.

The optical character recognition took another about 9.2 seconds for the 54 distinct characters. The character database had about 50 characters in it at this time. This rate of about 5.8 characters per second is heavily dependent on the time taken in extracting single bits from the bitmap. The rate is slower on large characters. In our current scheme, very little time is taken to compare the character's property vector to those that have been learned, and so we have not tried to optimize this into (say) a k-d tree. Using a cruder bitmap (say 200 dpi instead of 400 dpi) appears to be quite feasible for all the processing except for the higher probability of missing the separations between characters. At 200 dpi the scanning would proceed at about 4 times the speed compared to the 400dpi document (23 characters/second), and the storage/retrieval of bitmaps would be correspondingly faster. For at least some documents, 200dpi seems adequate.

The next step consisted of checking for possible lexical groupings like adjacent digits constituting a number, or S i n constituting a reserved word, and disambiguating the horizontal lines (minus, divide-bar, parts of the equal-sign, etc.).[8]

9 Acknowledgments

We acknowledge useful discussions on this and related topics with J. Daniel Halpern, Nicholas Mitchell (principal author of the first version of the parser), and Brian L. Evans. Taku Tokuyasu has begun quantifying measures of success in OCR on a character basis, and we hope to see a better assessment of our simple techniques plus a sturdier implementation of the Hausdorff techniques, soon.

References

[1] M. Bokser. "Omnidocument technologies." *Proceedings of the IEEE,* July 1992, vol.80, (no.7) 1066—78.

[2] G. Borgefors. "Distance transforms in digital images." *IEEE Trans. Pat. Anal. an Mach. Intel.,* 34:344–271, 1986.

[3] P. Chou, "Recognition of equations using a two-dimensional stochastic context-free grammar", *SPIE Conf. on Visual Communications and Image Processing,* Philadelphia, PA, Nov. 1989.

[4] D. Bierens de Haan. *Nouvelles Tables d'Inteǵrales Defínies* edition of 1867 corrected, with an English Translation of the Introduction by J.F. Ritt. G. E. Stechert & Co. NY. 1939.

[5] Richard J. Fateman. "Recognition and Parsing of Typeset Mathematics," UCB CS Division internal report, January 1994.

[6] Richard J. Fateman and Theodore H. Einwohner. "Automated Integral Tables" UCB CS Division internal report, January 1994.

[7] D.P. Huttenlocher, Gregory A. Klandermand, William J. Rucklidge. "Comparing Images Using the Hausdorff Distance," Cornell Univ. CS Dept. Tech. Rpt.

[8] Melvin Klerer and Fred Grossman. "Error Rates in Tables of Indefinite Integrals." *Industrial Math. 18* (1968) 31–62. See also, by the same authors, *A new table of indefinite integrals; computer processed* Dover, New York, 1971.

[9] G. Kopec and P. Chou, "Automatic generation of custom document image decoders", *Proc. Second Intl. Conf. on Doc. Anal. and Recog.*, Tsukuba Science City, Japan, Oct. 20–22, 1993.

[10] Nicholas Mitchell. "A Parser for Two-Dimensional OCR of Mathematics," UCB progress report 1994.

[11] D.W. Paglieroni. "Distance transforms: Properties and machine vision applications," *Computer Vision, Graphics and Image Proc.: Graphical Models and Image Processing,* 54(1):56-74, 1992.

[8] As a heuristic to aid further processing, we extend the divide-bars so that they will be more likely to span the width of the numerator and denominator. This accomodates our parser's assumption that the left-most character of a quotient is the divide bar. This phase took about .38 seconds.

Arranging the tokens into boxes to provide information for the parser regarding super/subscript information took .15 seconds.

The final parsing step, which, for example, must decide ax is (* a x) but $\sin x$ is (sin x), took negligible time: .01 seconds.