Progress in recognizing typeset mathematics

Richard J. Fateman, Taku Tokuyasu

University of California at Berkeley
Computer Science Division
Berkeley, CA 94720-1776

Abstract

Printed mathematics has a number of features which distinguish it from conventional text. These include structure in two dimensions (fractions, exponents, limits), frequent font changes, symbols with variable shape (quotient bars), and substantially differing notational conventions from source to source. When compounded with more generic problems such as noise and merged or broken characters, printed mathematics offers a challenging arena for recognition. Our project was initially driven by the goal of scanning and parsing some 5,000 pages of elaborate mathematics (tables of definite integrals). While our prototype system demonstrates success on translating noise-free typeset equations into Lisp expressions appropriate for further processing, a more semantic top-down approach appears necessary for higher levels of performance. Such an approach may benefit the incorporation of these programs into a more general document processing viewpoint.

We intend to release to the public our somewhat refined prototypes as utility programs in the hope that they will be of general use in the construction of custom OCR packages. These utilities are quite fast even as originally prototyped in Lisp, where they may be of particular interest to those working on "intelligent" optical processing. Some routines have been re-written in C++ as well.

Additional programs providing formula recognition and parsing also form a part of this system. It is important however to realize that distinct conflicting grammars are needed to cover variations in contemporary and historical typesetting, and thus a single simple solution is not possible.

Keywords: optical character recognition, typeset mathematics, utility programs, Lisp.

1 Introduction

What is special and difficult about recognizing mathematical equations?

1. Unusual fonts. Italic, Greek, special symbols are common. While this may be the difference that first comes to mind, it presents only minor difficulties relative to the other points given here.

2. Two-dimensional formatting. In order to recognize subscripts, superscripts, built-up fractions, summations, integral signs, matrices, alternation symbols, etc., spatial relationships in two dimensions have to be understood. Base-line information is both harder to extract and of more limited use. Additionally, super/subscripts may be in typefaces smaller than those ordinarily encountered.

3. Complexity of notation (via substitution, redirection, abbreviation). In-line comments affect the meanings of subsequent equations by giving names to expressions, abbreviations for repeated operators, or other shorthand notations. Common examples include using one equation to stand for two by using notations like $x^2 \pm a^2$, or leaving terms out of a sequence $\{1,\ 4,\ 9,\ \cdots\}$ to be understood by induction.

4. "One of a kind" layouts. Authors attempt to cover unusual conditions with a typographic construction understood by humans because of its context.

5. Extracting "meaning" from the formulas. For instance, a good system would provide output in a form easily transmitted for subsequent processing by computer algebra systems.

To be fair, we should mention a few ways in which mathematics recognition is easier than some other document understanding processes. The most prominent is an expectation of syntactic consistency and formal "correctness." That is, we ordinarily expect parentheses and other bracketing constructions to match. We expect a relatively small number of words (sin, cos, tan) to occur, and that many of the other symbols will be numbers, isolated italic characters, and scientific symbols.

This project [BF94] originated from our interest in scanning and storing thousands of pages of historically interesting tables of integrals. This has allowed us to limit the horizon of possibilities somewhat. Even so, the range of expressions we must deal with appears substantially larger than that usually considered in the literature (see Section 6 for further discussion of this point). We believe this is both a challenging and potentially fruitful area for optical character recognition (OCR) and document analysis research. A description of our overall approach and results so far can be found in [FT96]. In our ongoing efforts to learn what is required to create a robust mathematics recognition engine, we have developed a number of routines that we hope will be of general use to the public for the development of OCR systems. These are described in the following three sections. We turn to a discussion of parsing mathematics in Sections 5 and 6, and close the paper with some timing results and concluding remarks.

2 Progress at Berkeley

Before presenting details, we must point out that we claim no particular originality in the algorithms we have implemented. Good general references include Baird [BBY92] and O'Gorman [OK95], which provide keys to the literature.

We have not been able to utilize commercial OCR software, which appears to be nearly helpless at recognizing glyphs placed within a mathematical context and not aligned on conventional text baselines. We were disappointed in not even finding an application interface that we could use to benefit from commercial modules.

Thus we've developed a set of programs written in Lisp (and in part re-written in C++) to deal with certain parts of the OCR problem generally, and recognition of formulas specifically.

For some readers, the choice of Lisp may seem unusual or controversial, so we answer the question: Why did we program in Lisp? Several reasons:

1. We like Lisp, especially for exploratory programming.

2. We found Lisp's natural support for linked-list data-structures especially convenient for representing mathematics, as well as sparse-encoded graphical material.

3. Memory leakages, a common problem in C or C++, are eliminated by Lisp's automatic memory management.

4. There is a convenient built-in package ("Common Windows") for bit-map display. The easy availability of functions providing interactive access to the X-window interface has substantially

assisted in writing programs for learning characters as well as in debugging the OCR routines. This library is not essential for the running of the core routines; however, a realistic model of recognizing text may include a user-interface for spot-checking results, and for quizzing the user about uncertain identification. For these purposes, the Common Windows package should continue to be handy.

5. Lisp programs are quite portable: Portability to other implementations of Common Lisp require no alteration except in the interface to `libtif`, a public domain package we use for input and output of `tiff` (Tagged Image File Format) files. This alteration would consist of a transcription of the foreign-function call mechanism from the Allegro dialect we use to some other brand's form. We have used the same source code on HP and Sun workstations. By contrast, we have found variations in `C++` implementations and interfaces to `C` to be relatively less portable. This is of course a matter of a programmer's experience in a system.

6. We find the speed of our Lisp routines to be quite satisfactory, and preliminary comparison timings of our C++ implementation [HMRT95] suggest we are not far off from the speed attainable in other languages.

.

The tasks for mathematical formula recognition are, at some levels, the same as for other types of document recognition. In particular, designing a compact and efficient representation for input, de-skewing page images, filtering out noise, and the overall separation of text into lines or paragraphs, glyphs, and recognizable symbols are all generally useful. Therefore we have attempted to provide a collection of modules for each step in our processing. The interactive Lisp programming system we are using is especially supportive of a "functional" approach to composing programs, and provides a natural language for writing transformations of documents. By contrast, writing in C or C++ would ordinarily require one to design a script/shell like language for composing operations, or repeatedly editing/compiling/linking a C-language driver to the library.

For example, we have programs for these tasks:

- read a scanned page in `tiff` form into an internal run-length encoded (RLE) picture format,
- write out an RLE picture into a standard `tiff` file,
- detect the skew of a picture that was scanned at a (slight) angle and unskew it,
- rotate a picture by (multiples of) 90 degrees,
- identify connected components of a picture,
- judge the resemblance of a component to a library of letters (that is, recognize characters).

Given such building blocks, it is easy to develop programs to use these facilities. For example, here is a program that unskews and overwrites all the pictures in a directory stored in files named `*.tif`. Although the program below is in Lisp and relies on several routines we define in our package, it illustrates the compactness of composing programs, and the application of a program to a collection of names.

```
(loop for f in (files-in-dir :select "*tif") do (writetiff (unskew (readtiff f)) f))
```

Other functions can be iterated over the files in a directory (or perhaps sub-pictures collected by partitioning a single page) to produce:

- A file containing a character/position map of the scanned image.
- An ascii line-by-line interpretation of the page, positioning the characters in plausible positions, assuming it is text for which such an interpretation makes sense. A multi-column document should have each column set out separately.
- Another line-by-line interpretation of the page which includes font and point-size in some suitable encoding.
- An ascii file that can be interpreted as mathematics in an appropriate computer-algebra interchange form. In our particular case, this is text that can be read into Lisp or into a suitably configured computer algebra system.
- An error log concerning the page recognition task.

In addition to the above low-level tools which we are programming, we have completed a number of modules to explore approaches to higher-level processing. In particular, other tasks that we have addressed:

1. Clustering glyphs or alleged glyphs by some similarity measure for purposes of learning, and reinforcing learned identifications.
2. Learning by simultaneous consideration of a `tiff` image of text and its ground truth interpretation.
3. Interactive editing and prompting during recognition.
4. Spell-checking and other post-processing.

Our overall approach is to break up a page of text into regions, each approximately one equation's worth of material, and to represent these regions as collections of connected components (where a connected component is a set of touching black foreground bits separated from other components by white background bits). As in normal text, some connected components can actually be mapped 1:1 to letters or symbols. Typically, however, some of the actual glyphs will be broken into pieces by noise or design, and others will be stuck together.

The advantage of dealing with connected components is that they are relatively few in number, and a collection of them represents a convenient higher-level construct which provides opportunities for faster operations (at least in our experience) than would the bit-map representation. For example, one can easily find all components that lie, in whole or in part, inside some rectangle. A major benefit is that one ordinarily excludes totally from consideration large areas of a page that constitute margins, gutters, or similar white space. We later discuss (in Section 4.2) some of the difficulties that we (among others) have encountered in using connected components.

3 The Representation

If one has no *a priori* knowledge of what to expect from an image, the first cut at computing with an image is likely to be based on dealing with it as an array of numbers representing pixel values/colors.

In the case of a 2-level black and white image, an array of zero and one bits suffices. Such an array typically can be compressed substantially for secondary storage.

We found this initially appealing approach to be impractical for our objectives, at least given the computing at our disposal, except for rather small images and/or low resolutions. We are scanning at a minimum of 300, but often at 600, dots per inch (dpi). To display a 600dpi 8.5 by 11 inch picture on a 72 dpi workstation screen would require a display about 6 by 7.6 feet in size and use about 4.2 megabytes of RAM. If we are forced to use 8 bits (or more) per pixel, we are at 35 or more megabytes of RAM for one image. We are not saying that one needs to display whole pages at full resolution — just that this is a substantial sized array — and computing with it on a bit-wise basis is not something that one should do casually.

Normally, some substantially compressed data format is used for raster images. For example, one data set we have been working with is a black and white image, comprising 3408 by 5064 bits, which uncompressed would be $3408 \cdot 5064 = 17,258,112$ bits, or 2.16 megabytes in size. This is delivered by our scanner and associated software as a compressed `tiff` file of 72,592 bytes, representing a 30:1 compression.

At first blush it would seem that to do any processing, we would have to unpack the image to bits, and it is certainly possible to proceed on that basis. Our C++ programs treat this as an alternative representation to RLE, yet for LISP, the RLE advantage seems substantial.

Our RLE representation is developed as follows. Consider an image to be a sequence of (say) horizontal scan-lines. Each scan-line is made up of black intervals, or $\langle s, e \rangle$ pairs, where s is an integer denoting the start of a section of black bits, and e is the corresponding end index. These endpoint pairs can be encoded in a modest number of bits. For example, with fifteen bits at 600dpi, we can encode up to a maximum width of 54.6 inches, which is far wider than our scanners.

Using fifteen-bit indices, two of them can easily be stored as "immediate" values in the same space as a 32-bit pointer. In particular, in most Lisps on 32-bit word machines, numbers less that 2^{30} are easily packed into one "fixnum" (an encoded immediate number form). So if we pack these integers two-per-word, we now have a linked list of fixnums (not conses) for the run-length encoding of each line of a picture. The particular dataset we mentioned earlier requires a total list length of 78873 pairs. We thus now have the following storage form for our 3408 line by 5064 column bitmap:

1. 3408 words, one for each line, plus

2. 1 cons cell (= 2 words each) for each of 78873 pairs = 157746 words for a total storage of 645 kilobytes, in memory.

Also associated with each picture, we include data on height, width, and x/y origin of the scan, used in case we have many pictures on a single page.

Although we could reduce storage further by replacing the lists by vectors (reducing the storage to 1 word per pair), the total storage is still about 320 kbytes. But the use of linked lists, with the freedom to add and drop links, is such a convenience that we are not especially eager to trade that off for space.

4 The OCR Operations

Here we give some extracts from our on-going programming and documentation project, outlining some of the more useful and reliable of the programs.

4.1 Input, Output, Conversion

`readtiff ("filename")` Given the name of a b/w `tiff` file, `readtiff` reads it and returns a `picture` structure: width, height, x-y coordinate of the lower left corner (0,0 for a full page), and the collection of rows.

`writetiff (p, "filename.tif")` writes back out to the file system the picture `p` as a `tiff` file with name `filename.tif`.

`pict2bit (p)` converts a picture structure `p` into a bitmap `b`. A `bitmap` is a standard data structure used by Common Windows, and is the key to using the X-window display. If `WIN` is a Common Windows window descriptor for a visible window and `p` is a picture, then `(bitblt (pict2bit p) 0 0 WIN 0 0)` does a bit-block-transfer of the image to the display buffer to show the image of the picture `p` with its lower left corner registered with that of the window "canvas" `WIN`.

`bit2pict (b)` takes a bitmap `b`, as might be produced by interactive editing, and returns a picture structure. Since the Common Windows programs use the bitmap structure, it is plausible to (for example) write character editing or similar manipulation programs using bitmaps, and then translate the results to a picture form.

`writepicts2file (list-of-picts, filename)` takes a list of pictures[1], and writes them (rapidly) to a file in a format that is fast to read back in. Generally it is only slightly larger than a `tiff` file, but it could be much larger if the encoded material was more random. The primary advantage of a file produced in this way is that it can be indexed and accessed in random order if necessary. Therefore it might be a better representation for a large number of pages of text where access to individual picture components (e.g., characters or perhaps words) over a whole document is useful. The inverse operation for this output is `readpicts(filename)`, which returns a Lisp list of pictures from a file written by `writepicts2file`. Our assumption is that this program would be used as a model for a more elaborate program that could (for example) read the pictures non-sequentially.

4.2 Connected Component Extraction

`con-pict(p)` produces a Lisp list of the connected components of the picture `p`. In many cases these components correspond to the characters on the page image, although they can be both character fragments or merged characters. The current version of this program (including Lisp garbage collection times) finds about 1300 to 2000 components per second on a typical page. We believe this program is quite fast, even though this speed could be improved substantially by using lower resolution, and hence smaller, images. Halving the linear resolution should speed the processing by a factor of two[2].

A program to compute the reversal of con-pict is available: `manypict2one(plist)` takes a list `plist` of (perhaps many) pictures and converts it to one picture structure. This is a plausible operation, for example, if one believes that the connected component breakup of a large picture has produced pieces that should be reconnected.

Our initial enthusiasm about the usefulness of `con-pict` has been tempered somewhat by several realizations:

---

[1] Presumably these pictures would have been determined to be characters, graphical objects, or other identifiable components. They could also be lines of text, or perhaps connected components on a single page, as computed by `con-pict`, described below

[2] If we were using pixel arrays, we would expect the processing time to decrease by a factor of four. In our representation, while the number of rows would be halved, the number of intervals per row would decrease only when details were lost due to the decreased resolution.

a. We must also manage the fairly common situation where a connected component is *not* a complete single character (though we can reassemble them via `manypicts2one`).

b. The underlying assumption is that one can identify each component in isolation. By taking a connected component out of its context, we miss the nearby pieces that might be critical to enable us to identify the piece.

c. It is advantageous to take into account not only the black bits, but also the space around those bits, as the definition of a character. Part of the identity of a letter is its spacing relationship with respect to other letters (e.g. in the same word).

Thus unless one is quite sure of an identification, one may have to re-establish the context of "nearby connected components" to see if some re-grouping of pieces makes more sense. If we are attempting to explain a page by some global "best fit" computation that maps explanations (clean characters and positions) to blotches in the bitmap, the connected components may or may not be on the route to such a solution. Nevertheless, we have observed the potentially high usefulness of `con-pict` for a first-cut at recognition based on the following observation. Dealing with a few thousand objects representing connected components, given their coordinates and sizes, still seems to represent a savings, even if the objects do not always represent characters. We can still attempt to identify and separate "lines of text" and then "words" by grouping these object into sorted collections. Recognizing the letters in the context of a word should be easier than isolated connected components.

4.3 De-skewing

Due to imprecise placement, pages are typically scanned or printed by a mechanical device at a slight angle. While humans do not usually have difficulty reading such skewed pages, it is advantageous for computer processing to remove skew when possible. Two programs are provided:

`skew(p)` given a picture `p` computes heuristically a skew angle `d` in degrees that the picture is tilted. Typically the angle will be less than ± 10 degrees.

`unskew(p d)` given a picture and an angle, produces a new picture that is unskewed.

There are a number of recent papers [OK95] written on deskewing algorithms. We use a simple (well-known) approach based on the following observation: the distribution of white and black bits is far more uniform when scanned along a slant instead of parallel to the text lines. This fact is captured by finding the variance of the distribution of bit-counts along lines, which is easy to compute. We do a scan of lines oriented horizontally (very easy and fast), or at slight positive and negative angles (almost as fast). The scanning angle that corresponds most closely to the skew angle will have the highest variance. A short somewhat heuristic search can try to identify the "best" angle.

Given our representation, scanning at a zero angle is trivial, and the cost increases slowly as the angle increases. We can compute the variance of a page of 3400 lines in less than 0.1 second. The cost at one degree is more than one second: to scan a single "row" at one degree, one must scan and sum up the count of black bits from some original row for bit positions 0 to 56 bits, and then hop up one row and scan bit positions 57 to 113, etc. The step-size for one degree is approximately $(1/\tan(\pi/180)) = 57.29$. Hopping up one row and skipping to the interval that encloses the appropriate bits can eventually become time-consuming[3].

[3] Our initial code required a constant step size, but we found this too restrictive. Our current program allows for fractions: a step size of one degree is not exactly 57 to 1, but a sequence: (57 58 57 57 57 57 58 ...).

Fortunately, the statistical computation need not rely on looking at every scan-line, and therefore one can select every kth row (perhaps with a random perturbation), to speed up the calculation. In preliminary tests, we seem to achieve reasonable accuracy even if 9 out of 10 lines are skipped.

Finding the angle with the highest variance is done with a conventional numerical search. The function being maximized typically has the global appearance of a single maximum with a rather "flat" top. The detailed local appearance at the top is more fractal in nature, and so finding the precise maximum may be wasteful. An approximation good to 0.1 degrees is achievable, and by more elaborate curve-fitting, we believe a higher accuracy could be determined, if required.

4.3.1 Correcting skew or orientation

We have programmed two picture-oriented shear transformations that, when combined, can quickly perform a near-rotation of a picture, directly on the run-length-encoded data. This is sufficient for correcting small amounts of skew. Detailed analysis of the errors are given elsewhere [FT].

Occasionally papers are scanned either accidentally or purposely at right angles (or upside down). Scanning software can be adjusted for this, but sometimes with unattended operation it is not apparent that a figure or table has been scanned at an angle until later. As a building block for code (yet to be developed) which can automatically correct for page orientation, we have written a program `(rot90 pic)` to rotate a single picture clockwise by 90 degrees. A rotation of a run-length encoded full page at 300dpi takes about 2.8 seconds[4]. On a typical page, it may be better to first separate the page into connected components (1.6 seconds) and rotate each of a few thousand small characters: this rotation requires only another 0.55 seconds. The savings accrue by not having to "rotate the white-space".

4.4 Distances between pictures

How can we tell if one picture (of an alleged character) looks like another picture? The most direct method might be comparing the character bitmaps by an exclusive-or, and counting the unmatched bits "left over." A large number of bits suggests the pictures are of different characters.

We have programmed a rather efficient algorithm `count-bits-in-xor-of-bitmaps` which takes two bitmaps `p1` and `p2`, where `p1` is a bitmap of a character (e.g. from a standard library), and `p2` is a page which might contain instances of the character. Additional arguments to this function should be pre-computed: the number of bits in `p1` alone and the bits in that area in page `p2`. These pre-computed data speed up the computation. Although this is, at first glance, appealing, it also requires a precise line-up of bit maps.

There are quite a few alternative possibilities. A useful metric must be fast to compute as well as likely to coincide with the human judgment. The Hausdorff distance is appealing, but we do not have evidence yet that it is significantly better for recognition, considering the expense [Tok95]. Another approach, which appears to be used by some commercial programs [Bok92], has some appeal and is described below.

We divide the area of a connected component, or some unit separated from the image as a potential character, into n by m regions and count the ratio of black vs. white bits in each region. We currently use a 5 by 5 grid, but more or fewer divisions could be used. For each character we compute the 25 values, each scaled from 0 to 255, for convenience in storage. The cost for this computation averages in one test to about .51ms, or about 1958 computations per second (1.09 seconds for a page).

[4]Times reported in this paper are for a Hewlett-Packard 9000/715 workstation running Allegro Common Lisp 4.2.

We compute the (optionally weighted) distance between property vectors p and q as the Euclidean distance: $\sum_i w_i(q_i - p_i)$. Rather than treating all characters uniformly, we found it advantageous to make an "absolute" grouping for complete characters. That is, we only bother to compare p and q if they were within 10% in absolute height and their height-to-width ratio was within 10%. This characterization assumes (in general this is an optimistic assumption) that all the parts of the character are connected, or have, by means of heuristics, been put into a single picture form.

Note that there are some unusually tall or wide characters in our domain of interest: divide bars for fractions, tall parentheses or integral signs, etc.

An undergraduate team writing similar programs in C++ [HMRT95] also found that separately recognizing figures in categories based on absolute size improved accuracy, and that including two additional regions to the 5 by 5 grid, the top 10 percent and the bottom 10 percent of the figure, helped as well.

5 The Mathematics Operations

Our initial approach was to try ordinary commercial OCR to get a page of mathematics into a page of text, and then run a somewhat conventional (although 2-D) parser, over the result. This is not even close to being viable. The commercial OCR programs generally fail to even recognize the symbols adequately, partly because they are not on appropriate baselines, are unusual in typeface and size, and do not conform to expectations of English, etc. Even if this were to succeed, formulas as typeset follow peculiar conventions that are *not* those formulated by programming languages specialists. Much as one would like to believe that FORTRAN translates formulas into computer language, it is hardly the universal solvent. Yes, FORTRAN would easily parse `sin(a*x)*cos(b*x)`, but this is not how such an expression would appear in a reference book! Perhaps one would see the un-FORTRANish $\sin(ax)\cos(bx)$. We are yet more likely to be faced with forms that require more devious parsers. Consider the expression $\sin ax \cdot \cos bx$, taken from an integral table [dH39]. Observe that there are two multiplication "symbols" in use. The center-dot $\cdot$ is a low-precedence multiplication (lower binding than cos), but the "empty space" between b and x is also a multiplication, but of higher binding precedence than cos. Note also that "empty space" is used to designate the application of the function cos to its arguments. A realistic parser must take this, and much more, into account. Even so, there are enough one-of-a-kind formulas, even in a collection of integrals, that some after-the-fact human editing is likely to be required.

Stepping back from this naive approach, we try this instead: We first collect objects (connected components) on a page. We then divide the page into buckets separated by large gaps of white space. We analyze the contents of the buckets by a top-down/bottom-up analysis in four phases, some of which we may later re-program in an interspersed manner. The first phase handles sorting and segmentation, and next we recognize tokens like tan or numbers. Following this we order the 2-D collection into a structure that is linear, and finally we use a nearly-conventional parser on this structure. As programmed, the process provides two results:

- an explanation for the components that "make sense" to its rules for the construction of mathematical expressions, and
- a list of those pieces it was unable to explain.

If the second item is empty, or is a list of components that can be dismissed as noise, we are done.

A more energetic approach would search more effectively for components that are needed to complete explanations of constructions in the phases below, *as they are being proposed.* In the segmentation phase, failing to find an integral sign or an $=$ suggests that this is a continuation line and should be joined to the line above it. In the lexical analysis phase, uncertainty as to whether a character is a digit or a letter could be resolved depending on what is expected. In the linearization phase, failing to find a matching right bracket (appropriately sized parenthesis, square-bracket, brace) in an expression should trigger a reconsideration of the left-bracket character, or renew consideration of possibilities for the right-bracket character in the region.

In more detail, we describe the phases outlined above.

Segmentation This phase separates equations from each other and from the page decorations. We may have to use local clues such as recognizing that each equation has an integral sign on the left and possibly a continuation line or lines. For continuation lines we must "attach" the continuation to the right side of the previous line, eliding repeated operators. Distinct equations are generally separable by noticing more vertical white-space than between continuation lines. At this stage it may be plausible to try to remove citation, reference and other textual information, although we currently pass this through to later stages.

Lexical analysis Within each segment, we collect adjacent digits and names to form numbers and symbols such as sin, cos, tan, log, exp. Normally, printed formulas do not contain multiple-character names other than those in the usual list of functions, and those normally in Roman font, so other adjacent letters or numbers such as $2p$ or dx are *not* joined in the lexical stage.

We can distinguish horizontal lines in five categories: quotient bar, minus sign, top of square (or nth) root, and part of $=$, $\geq$, $\leq$. We also collect comma plus dot as ";", etc., if the OCR phase has not already done so. Other items may be single-letter names, if alphabetic. Other symbols such as parentheses are their own categories.

The bounding boxes for collected objects are adjusted as needed. For example the token for sin now looks like the "spatial sum" of the characters in its body, suitably bulked out to a rectangular form.

To distinguish a minus sign from a quotient bar, we construct a square bisected by the horizontal line. If there is a character in a box intersecting the upper half, we check to see if it is also a horizontal line (an $=$ sign), a $>$ (a $\geq$), etc. If none of these situations hold and there is also a character in the lower half, then it is classified as a quotient bar, or possibly a disconnected square-root. In practice we must shrink these boxes slightly because italic characters' bounding boxes tend to overlap. That is, in $-p$. the tail of the p may in fact be literally below the $-$. After identifying the quotient bar, we then extend it slightly to the left to make sure it is to the left of any character in the numerator or denominator. A square-root bar (when detached) is identified by the presence of the "check-mark"-like piece at its lower left.

In some of our documents, the typesetting of long lines seems somewhat haphazard with frequent continuity problems, and it may be judicious to try to fill in some missing pixels at this time (although we don't currently do this).

Continuing on in our processing, if there are no characters above or below the horizontal line, it is a minus sign. There are other cases involving letters that are underlined or overlined, but deciphering

$$\frac{\underline{a}}{b} \equiv \frac{a}{\overline{b}}$$

which in TEX is `$ {{ \underline a} \over { b} } \equiv  {a \over {\overline b} }$` requires more global processing than we are willing to use. We do not expect to be able to parse everything that can be perversely constructed. Nevertheless, these methods correctly recognize all the quotient and minus signs in numerous test cases [FT96] as well as the equal sign.

Linearization In this phase we build horizontal and vertical hierarchical structures that encode the adjacency detected among the tokens in a non-numerical format.

Heuristics for what constitutes "adjacent" and what constitutes "superscript" are somewhat corpus-dependent and must be, in part, based on baseline measurements for characters. Notice that in the form x^p the exponent is nearly adjacent to the base, while in the form p^x, the bottom of the exponent is far above the base. The disparity is minimized if we consider the the base-lines of the characters rather than their bounding boxes.

The structure we produce here consists of encoders such as `vbox`, a box encoding vertical information, `expbox`, encoding the relationship of exponents to bases, and `hbox`, a catch-all for horizontal encoding. Our full grammar may eventually need more precise encoders such as right-vertical box (`rvbox`) and centered-vertical box (`cvbox`) to account for the two ways of typesetting an integral's limits, but we may do not yet need these subtleties. The output of this phase is a recursively nested list of structures that is quite typical of a Lisp expression, with individual characters or strings (the lexical content of one of these boxes) at the end.

We have programmed the process of parsing by using "recursive descent" where each parse of a subexpression is confined to a region of the surface. For example, presuming that the leftmost character of the region (x_0, y_0) = bottom-left to (x_1, y_1) = top-right is an integral sign, one can divide the remaining region into three: above, below and to the right of the $\int$. If there is an upper limit, then it must begin in the rectangle above the $\int$ and below the upper limit y_1. The limit can continue to the right of its start-region, and similarly for the lower limit. After considering these regions to produce limits, the whole region to the right of the $\int$ must be parsed as the integrand, dx, $=$, and the answer.

The case of a fraction is guided by the horizontal quotient bar, which is the leftmost character. Our program also limits the horizontal extent of the numerator and the denominator by the size of the quotient bar. For those symbols where a superscript is plausible[5] we key off the baseline of the main expression. Other expressions like $x + a$ or $-x$ are treated by proceeding from left to right (approximately) along the baseline.

The key in each case is that given a prospective rectangle for an expression, the leftmost glyph governs the meaning of the expression. In the few exceptions to this rule, we have tried, by manipulating the expression glyphs to expand the key operator to the left to assert this truth. For example, we consider extending $\int$ by a "virtual" bar extending to its left.

Final phase Linearization is followed by an essentially conventional parser. The encoding of the previous phase has removed from consideration the absolute location information of characters, and has linearly-encoded up/down information through `vboxes`. This phase handles the more sophisticated interpretation necessary to determine when adjacency implies multiplication, when it implies function application and when it implies some kind of operator (like d followed by x). For our approach to be successful generally, this phase must be easily modifiable to provide parsing capability for specific

[5]We do not expect, and therefore do not look for, superscripts everywhere. Certainly not on a +, for example.

constructions encountered on a corpus-by-corpus basis. We have not pursued this in detail, but a rewrite of this parser using syntax-generated techniques keyed to a 2-D grammar specification is plausible.

6 How hard is parsing an integral formula?

It is easy to underestimate the difficulty of the mathematics parsing task unless one has actually looked at some formulas. The following two integral formulas were selected as representative of many similar formulas in a standard reference, [PBM83]. These particular forms appear in section 2.16.1, page 336.

18.
$$\int_0^\infty x^{\alpha-1} e^{\pm bx^r} \mathrm{Ei}(\mp bx^r) K_\nu(cx)\,dx = -\frac{\pi}{2}\hat{U}_\nu(0,0)$$

$$[r, \mathrm{Re}\ b, \mathrm{Re}\ c > 0;\ |\mathrm{Re}\ \nu| < \mathrm{Re}\ \alpha;\ \hat{U}_\nu(\gamma,\epsilon) \text{ defined by 2.13.1.13}].$$

19.
$$\int_0^\infty x^{\alpha-1} e^{\pm bx^r} \mathrm{Ei}(-bx^r) K_\nu(cx)\,dx = X_\nu^\pm$$

$$[r, \mathrm{Re}\ c, \mathrm{Re}\ \beta > 0;\ \mathrm{Re}\ \alpha > |\mathrm{Re}\ \nu|].$$

$$X_\nu^\pm = \frac{b^{-\alpha/r}}{2} \sum_{k=0}^{\infty} \frac{1}{k!}[B_k(\nu) + B_k(-\nu)] \left(\pm\frac{2c}{b^{1/r}}\right)^k,$$

$$B_k(\nu) = \left(\frac{2b^{1/r}}{c}\right)^\nu \frac{\Gamma(\nu)(1/2-\nu)_k}{(-\nu-\alpha-k)(1-2\nu)_k}\Gamma\left(\frac{\alpha+k-\nu}{r}\right)$$

$$[r > 1;\ \ 2\nu \neq \ldots,\ -2,\ -1,\ 0,\ 1,\ 2,\ldots].$$

Note the following peculiarities: even if the character recognizer is able to recognize, separate, and correctly position all the characters, you have just begun[6]. First of all, the $\pm$ notation is not context-free, because it requires a reference to a (possible) previous or next instance. The variables α, b, c and r are used "free" in the formulas for X and B (and $\hat{U}$ defined on page 254 but not shown here, not only refers to those free variables, it implicitly refers to the choice of $\mp$). The restriction that ν is not a positive or negative integer, or a half integer requires the interpretation of ellipses. The superscript $\pm$ can be expected to break many parsers as well, since this symbol would ordinarily be a unary prefix or binary infix operator, not a stand-alone token or marker like a prime.

By reference to the source text, one can learn that the definition of $\hat{U}$ requires most of a further page of mathematics, and involves evaluation of infinite summations each term of which involves non-trivial notations. Also note that a separate formula is given for $X_n^\pm$ for integer n.

7 Timing

Times reported in this section, and in passing previously in the text are for a Hewlett-Packard 9000/715 workstation running Allegro Common Lisp 4.2.

[6]The original scanned text is far less clear than the freshly typeset notation above.

On a typical page, one can read in a tiff file in about 2 seconds. Deskew can take a few seconds (depending on sample size and actual angle—small angles are faster). We can separate the page into connected components in 1.6 seconds. If one decides that the page was actually scanned sideways, we can most rapidly rotate the page in a representation of a collection of components. Assuming a standard page of a few thousand small characters, this program requires about another 0.55 seconds. We are much less certain about the character identification times, since this will likely grow as identification of noisier pages are attempted and our programs become more ambitious.

Parsing of those mathematical formulas on which we have total success takes negligible time compared to the prior processing steps. For example, in tests on a 54-character formula [FT96], heuristics to find plausible explanations and disambiguations of the horizontal lines (minus signs, divide bars, equal signs) took about 0.38 seconds, dividing up the space into super/subscript areas, matching parenthesis, and similar tasks took 0.15 seconds, and the final parsing an additional 0.01 seconds.

Since our initial implementation of some of these programs was reported in [BF94], we have reprogrammed components and achieved substantial speedups as indicated above.

A recently completed (12/95) [HMRT95] student project writing code to a similar design but in C++ rather than Lisp has resulted in other timing data points for some, but not all components. We do not have any direct comparison to other implementations of the exact same capability; however, this speed seems more than adequate for our needs.

8 Conclusion

Mathematical equation OCR remains a challenge. Tools must continue to be developed for a satisfactory resolution of problems we have encountered. Many of our algorithms so far have been low-level routines appropriate for a bottom-up character recognition approach. Together with the parsing scheme given above, this is quite sufficient for the recognition of many relatively simple perfectly typeset images of mathematical formulas [FT96]. Our experiments have also shown, however, that a purely bottom-up approach is of limited use in the face of noisy data (such as, in our case, integral tables printed in 1939 [dH39]). We plan to incorporate routines for higher-level structure analysis, borrowing for example from ideas for page segmentation and classification [PZ92]. These tools can, we believe, be used in general document structure analysis, and so have a general payoff in development beyond the immediate objectives of mathematics recognition.

9 Acknowledgments

This work was supported in part by Grants numbers CCR-9214963 and IRI-9411334, and by NSF Infrastructure Grant number CDA-8722788. Conversations with students and colleagues have assisted us with numerous issues from conceptual to implementation. Thanks particularly to Ben Berman, Ted Einwohner, Dan Halpern, Jon Hull, Gary Kopec, Kathey Marsden and participants in the UCB document structure analysis seminar.

For access to the code, contact either of the authors, `fateman` or `tokuyasu@cs.berkeley.edu`.

References

[BBY92] H.S. Baird, H. Bunke, and K. Yamamoto, editors. *Structured document image analysis*, Berlin, 1992. Springer-Verlag.

[BF94] B. P. Berman and R. J. Fateman. Optical character recognition for typset mathematics. In *ACM Proc. ISSAC '94, Oxford, UK, July, 1994*, pages 348–353, 1994.

[Bok92] M. Bokser. Omnidocument technologies. *Proceedings of the IEEE*, 80(7), July 1992.

[dH39] D. Bierens de Haan. *Nouvelles Tables d'Intégrales Définies.* G. E. Stechert & Co., NY, 1867 corrected edition, October 1939. with an English Translation of the Introduction by J.F. Ritt.

[FT] R. Fateman and T. Tokuyasu. A suite of lisp programs for optical character recognition. manuscript in progress.

[FT96] R. Fateman and T. Tokuyasu. Optical character recognition and parsing of typeset mathematics. *J. Vis. Commun. Image Represent.*, 1996. to appear.

[HMRT95] James Hopkin, Kathey Marsden, Archie Russell, and Cynthia Si Tian. Ocrchie: Optical character recognition library. CS169 term project, U.C. Berkeley, 1995.

[OK95] Lawrence O'Gorman and Rangachar Kasturi, editors. *Document Image Analysis*, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[PBM83] Prudnikov, Brichkov, and Marichev. *Integrals and Series, (volume 1).* Moscow gov't printing office, 1983.

[PZ92] Theo Pavlidis and Jiangying Zhou. Page segmentation and classification. *CVGIP: Graphical Models and Image Processing*, 54(6), November 1992.

[Tok95] Taku A. Tokuyasu. Optical character recognition of typeset mathematics. Master's thesis, U.C. Berkeley, May 1995.