# A Suite of Programs for Document Structuring and Image Analysis using Lisp DRAFT DRAFT

Richard J. Fateman*
Taku Tokuyasu
Computer Science Division, EECS Department
University of California at Berkeley

## 1 Introduction and Motivation

As part of an effort to incorporate printed material into a digitally stored library, we were faced with digitizing and (generally) analyzing substantial quantities of printed technical documents. For reasons that will become apparent, it appeared that commercially available optical character recognition (OCR) programs were not adequate to all parts of the task. We have vacillated to some extent in deciding exactly how much of the analysis could be done by commercial software, and have currently settled into a shared arrangement where we attempt to partition documents into sections that can ordinarily be processed by commercial proprietary software, and other sections that need special care.

We begin our processing given two-level images of (mostly) text, and assume that some gross page processing (removing obvious half-tone images) has been done.

Even after such filtering, the text we are concerned with may still be unconventional from the perspective of packaged business-oriented OCR software. Our major interest has been material that includes two-dimensional mathematical notation, with the sprinkling of unusual symbols, that goes with it. We have not been concerned with tables, logos, or diagrams, but at least some of our programs should be applicable to diagrams with non-linear layouts. Prior to our involvement in the digital library project at Berkeley, we had begun a small project on the recognition of mathematical text — specifically tables of integrals — into a semantically useful encoding for automated computer lookup. This task required parsing documents in a domain that can be categorized loosely as "advanced calculus equations". Although the techniques developed for this domain can be carried over into the recognizing the patterns of ordinary documents, (after all, a text word is a special case of a mathematical formula), we expect that the more general nature of our techniques would result in overall less accuracy on ordinary business documents, where words occur fairly reliably on lines in paragraphs (etc.) Our tools would have some advantages in unusual (non-text) circumstances.

*This work was supported in part by NSF Grant number CCR-9214963 and by NSF Infrastructure Grant number CDA-8722788, and NSF Digital Library grant IRI-9411334.

Before discussing the particular attributes of the programs we've developed, we wish to review a few questions relevant to our overall project.

### 1.1 How should scanned text be stored?

The cost of (especially tertiary) storage has dropped to the point where it is economically feasible to base document-handling systems on storage and retrieval of pages as compressed bit-maps. Indeed, some business workflow applications are fundamentally data-base retrieval systems for such pages where externally imposed keys are used for indexing, but the pages themselves are essentially opaque to processing.

On the other hand, for many applications, including ours, it is useful and sometimes critical to be able manipulate the contents of the scanned material as though it were text. To be most useful, this content should be structured or "parsed" so it can be subjected to indexing, search, reformatting, editing (including cutting and pasting), computation, and economical re-transmission. Such a recognized document can be stored as a bitmap plus a structure. These combinations (in the Berkeley digital library project we call them multivalent documents) have become very attractive. They look like especially good solutions for a corpus of widely varying materials subjected to flexible search, retrieval and processing. [8].

### 1.2 Why not use off-the-shelf solutions?

Typical commercial OCR programs are quite properly targeted at their most likely source material: business-letter text. Programs can sometimes succeed on other page recognition task as well. Some can handle columnar data. Some programs are especially successful for forms recognition from pre- "zoned" documents. They are best used for high volumes of essentially similar documents. In some cases considerable effort has been devoted to dealing with lower-quality scanned images in a variety of fonts. The commercial programs are commonly packaged for efficient distribution and ease of use in common cases. Their effectiveness can sometimes be improved if the user specifies particular known qualities of the material (e.g. fixed-width fonts, "noisy" etc.)

The commercial designs appear to substantially preclude using and refining component tools to gain higher levels of recognition on unconventional material (e.g., mathematics).

Along another dimension, most systems appear to take insufficient advantage of certain contextual information that may be available: for example, we can get much higher accuracy on a corpus if we are given material known to be highly structured and "very" stereotypical: for example pages that one can depend upon to be all in a particular known font. Such a specialized recognizer could maintain its relatively high accuracy in the face of much noisier data.

As discussed in more detail below, we also require, for some of our applications, a keyed recognition result where the words refer back to their positions on the original scanned page. Entry-level packages do not provide such results. Typical API "development" packages offer this level of data. Because there are no standards here, the data is provided in some proprietary vendor format (e.g. XDOC for Xerox/Scansoft Textbridge, PDA for Caere).

Our experiments with several commercial OCR programs used on our initial data of *mathematical text* suggested that available monolithic commercial products would just not perform adequately. Accuracy, even in recognizing constituent characters was low. After discussions with the vendors, it was apparent that systems of that time (1994) were not built in a form enabling us to extract "modules" for re-use[1]. Therefore we embarked, reluctantly at first, on a project to design and implement our own OCR programs. The routines described in this document are the early fruits of this effort.

Naturally, not all commercial OCR-related activities are subsumed by the OCR programs available (now) at modest cost "over the counter" or included with scanner hardware. Businesses also provide custom OCR solutions tailored to specific needs. These are, however, expensive and difficult to obtain when the range of documents is not well defined.

## 2 Goals for our design

The modules described here are intended to be portable, reusable, reasonably efficient building blocks for optical character recognition and related document-structuring tasks. They are based on straightforward designs, mostly mirroring what has been shown to be effective in the literature. For the most part, we have deviated from simplicity only when the simplest solution was tried and found inefficient or inadequate. We expect that further development will follow the same route.

The programs need not be used together as an end-to-end package for recognition of bitmaps to ascii text-file. In fact, our first digital library application has been to use one or two modules solely to de-skew bitmaps, without any recognition processing[2]. The de-skewing generally corrects misalignment in the scanner/feeder mechanism, or earlier production problems in the original corpus. We expect that modules for input/output to different formats could be constructed and the modules for recognition and learning of isolated characters easily replaced.

Another important goal for us is to provide, at a reasonably accessible level, the correspondence between words (or equations) and positions on the original page. Thus it should be possible to see that a particular word occupies a specified rectangle on a page image, or that a mathematical expression occupies a particular rectangle[3]. This level of information is used in Berkeley's multivalent document system.

## 3 Non-Goals for our design

While we are willing to see more features added to this suite of programs, we are not attempting to provide all the facilities that have appeared elsewhere. For example, we are *not* concerned at this time with:

- Providing as output, the input formats specific to the many editors available for personal computers.
- Recording the font, style, and point-size of characters, except as it matters for recognition. (In fact, we need to know about italics for mathematics.)
- Recognition of reverse-video, cross-out, script, half-toned or similar modified text.
- Automatic recognition of 90-degree rotated text.

We recognize that such capabilities may be valuable in some contexts, and hope that others may find our tools useful in building such capabilities, should they be necessary.

We also have made some efforts (1999) to integrate other OCR programs (in particular, Scansoft/Xerox's TextBridge) into our system so that in the Windows environment a subsection of a page can be sent to a commercial "intelligent character recognition" engine.

## 4 Outline of OCR tasks

We are grateful to the many researchers in OCR who have explored various options in approaching standard problems. First we briefly outline the currently accepted wisdom on the steps necessary for successful document understanding ([6]), In subsequent sections we show where our pieces fit in.

### 4.1 Pixel-level processing

Faced with an image scanned in to the computer, the first group of processes deal with pixel-level transformations.

- Thresholding is the determination of whether a particular pixel position is to be treated as white or black, given that it is actually perceived as some level of color or gray: We don't address this task in our programs, relying instead on the scanner (perhaps with its low-level software) appropriately adjusted, to come up with the (binary level) image. While such an arrangement can be fooled by inverse-color printing, printing on top of half-tones, etc., we have found that the scanners we've used can generally be adjusted satisfactorily to produce 2-level images from our documents. This decision to leave well enough alone could be re-examined and we could either write our own thresholding program or directly use a gray-scale. This latter approach would seem to be far most costly than our binary bitmap approach. We are willing to re-examine this in the future since it seems especially plausible to trade-off low-resolution gray-scale for high-resolution

[1] Some systems are "open" through application program interfaces, but this is not modular in the sense we were looking for. Five years later, the same situation still prevails.

[2] It may not be obvious, but de-skewing a scanned page, even by a degree or two, helps in the manipulation of the bitmaps "by hand". That is, if one uses the traditional "rubber-band rectangle" aligned to the computer-display X-Y axes to select a "line of text" or some other rectangular section of a page, misalignment causes problem. Straightening the page fixes this problem, and makes the multivalent document processing much easier: selected picture rectangles correspond to text blocks in the OCR format.

[3] In-line math running over two or more lines require a more elaborate shape than a rectangle.

2-level images, and there is considerable evidence that keeping gray-scale information in low-resolution pictures makes them much easier for humans to understand.

- Noise reduction. This can include a host of transformations attempting to modify or filter the shapes represented, including morphological processing and "kFill" filters. Initially we did not do this, but we have added some morphological style processing for half-tone detection, and a kind of smearing operation for segmentation. (Crudely speaking, letters tend to become words if you smear their bounding boxes horizontally). While some kinds of morphological processing are relatively clumsy to execute in our current run-length-encoded representation (e.g. 2-D filtering operations), horizontal smearing is easily accomplished. And if an image is represented as a collection of bounding boxes (as we do for segmentation), then vertical and horizontal smearing is rather fast. We discuss this in a subsequent section.

- Thinning/skeletonization. This is a kind of higher-level morphological concept that can be applied to images which appears especially useful in images that are graphs, maps, etc. Although one could argue that it would be useful also for character recognition, it appears to us that thinning loses some useful information that we could use for recognition. We can easily do naive horizontal thinning by shortening intervals in a scan line. We can do vertical thinning most directly by rotating the RLE encoding, doing horizontal thinning, and reversing the rotation. Since a 90 degree rotation takes a few seconds (in our representation) this is hardly free, but not as expensive as we've seen it in other programs. One would ordinarily try to avoid repeated rotation operations. "Real" thinning is far more sophisticated. We do none of this.

- Chain coding and vectorization. We do not do either of these *per se*, though run-length encoding on a row-by-row basis serves some of the same needs: it is easier to compute connectivity, and it is potentially far more compact than bitmaps. RLE also makes use of more easily (arithmetically) manipulated chunks of information than bit-strings. Although we hesitate to make a totally language- and machine-independent judgment on this, in our programming environment there is something of a performance penalty for operations which tend to require masking/shifting operations to extract, compare, and count bits from memory, versus doing integer arithmetic.

- Connected components, region detection, feature-level extraction. We provide support for connected component extraction and some heuristic feature detection (e.g. The number of columns or lines of text can be based on finding vertical or horizontal gaps of a given width). Our current design deskews and find connected components first, which seems somewhat at odds with the ordering in O'Gorman [6], but is probably more a consideration of our premise: that we are dealing with text, and prior processing has removed other material.

  If we are uncertain of the nature of the text/half-tone mixture, then it makes sense to try to test for such non-text features and remove them as a preliminary to connected component extraction. Such an operation on a sizable half-tone would be extremely slow.

## 4.2 Region detection

Detection of different text and other regions, combined with reading-order segmentation seem to be important in achieving high quality results in the following sense: Regions detection, by which we typically mean identifying headers, footers, paragraphs, columns, displays, etc. is useful for

- Specializing recognition: Title fonts, mathematics, footnotes may benefit from different approaches. For example, knowing that a section is entirely text means that word/line/character heuristics can be applied for better identification. Knowing that a section is mathematics (or that a text section contains math) means one might expect lower success in looking for words.

- Computing or Indicating Order of Reading: The reading order is just an attempt to impose a consistent ordering for material that appears on the page so that at least the flow of the words of continuous text is given. Given a number of regions, one generally reads them top to bottom. If there are several columns, the left-most column is read first (etc.)

  Beyond that, heuristics begin to play a larger role. If the page is really a table with columns, horizontal lines are more plausibly each "separate records". Some printed material has much fancier layouts. Popular magazines might have large-font quotes pulled from the article and splashed across the page, perhaps crossing columns. Presumably the quotes are to be read first – before *or instead of* reading the page. Sometimes it is not clear how to resume the reading order below such a quote, and human readers are confused. Humans may read captions on figures and tables before the text. Humans probably do not read the page number at all unless there is a suspicion that pages are out of order or missing.

How is this done? There are a variety of techniques that have appeared in the literature, using textures, manipulating of connected components, and hand-correction. We believe that any automatic segmentation is going to be fallible, and therefore to the extent that it must be done correctly, we *require* tools to correct the segmentation. We have designed and implemented a program with a pleasant user interface for zoning (CALZONE) [5]. This allows the user of our document processing system to examine the result of our automatic zoning and correct it by altering general parameters, and to correct particular one-time errors. It also provides a simple mechanism for identifying *by hand* those sections which contain mathematical equations. While display equations can sometimes be identified, it is rather difficult to identify automatically a brief in-line mathematical expression like $x - yz$, and especially tricky to define it as a zone if it is split over two or more lines in a non-rectangular region.

In an attempt to solve some of these problem and bring the manipulation into the Lisp model, in the summer of 1996, Richard Fateman implemented[4] an interactive zoning program, described briefly here. See also `http://http.cs.berkeley.edu/`

[4] We used Allegro Common Lisp and its Common Windows graphics package, a medium-level object-oriented system that depends on light-weight multiprocessing threads in the UNIX Lisp implementation. It seemed to us at the time to be the best compromise between efficiency and utility. In the grand tradition of graphics packages, this has subsequently become an "unsupported standard." Although this

In order to enhance the manipulation of page images, we developed a design for a user interface for panning and zooming in and out[5].

Acrobat scaling seems to be limited to .5 to 8.0 in certain steps. To accommodate a 1-button mouse, it has a slightly different detail in control. The program "Imaging" by Wang Inc (for Windows NT), and distributed with Caere's development system is, in some dimensions, a simpler alternative that allows one to view TIF files by panning and zooming; it however also allows one to edit the image[6].

**Given:**

- A huge picture P off-screen which is far too large to display on a pixel by pixel basis. For example, P might be 5100 by 6600 pixels, corresponding to an 8.5 by 11 inch page scanned at 600dpi.
- A large canvas L on screen, filling much of the display. This might be 800 by 1000 pixels.

**Objectives:**

- To see on the canvas a selected section of the large picture for visualization, editing, selection etc.
- To scroll/pan the selection section around so that any part of the picture can be seen.
- To change the magnification (positive or negative). At one extreme, the whole picture (presumably greatly shrunk) can be seen, displayed on the whole canvas. At the other extreme, a small piece of the picture, much enlarged, even a portion of a single character, is enlarged to the size of the canvas.

The usual tools in a drawing program are scroll-bars horizontal and vertical, and zoom boxes big/small. We found this rather clumsy, requiring considerable "remote" hand motion to get to another location on the page, and also requiring fairly precise pointing within a scroll-bar. Instead we implemented the following alternative tools:

We display a thumbnail image T of the whole page, with a sub-rectangle S, the same proportions as T, superimposed on it. The canvas L is also geometrically congruent to S and T. At any given time, the sub-portion of T covered by S is displayed on the large canvas. By moving S in T, corresponding motions of L on P are accomplished.

Shrinking S in the thumbnail magnifies the view in L.

Through experimentation, we found a variety of features to enhance the user interface, although the primary interface criterion emerged as fast interaction: the speed of redisplay. The simplicity of the controls were important in our own (admittedly *self*-evaluation) (extra features: warping mouse cursors, changing cursor shapes).

These description pale in usefulness to a 15 second demonstration, but we will try to use text, nevertheless. In particular, to shrink/expand S, click/drag inside/outside S with left mouse button. The cursor changes to NE arrow. The mouse is warped to the closest corner of S. Motions of the mouse are tracked with S changing proportionally. Moving out of T terminates the tracking, as does releasing the button. At this point the newly determined size of S covers a portion of the thumbnail picture T which is then re-scaled and displayed to fill canvas L.

To move (pan) the displayed sub-picture, press the left mouse button in the thumbnail window. The mouse cursor warps to the center of the region S, from which the region may be moved until the button is released.

Along with this interactive viewing, we implemented a sequence of operations that guesses (with user input) at zones in a page. The user can set sliders to change parameters that affect this automatic zoning as described below in the section on automatic bottom-up segmentation; if no fully-satisfactory automatic zoning can be attained, the user can use mouse commands to break apart zones that are too large, and join zones that are incorrectly separated. We initially implemented rectangular non-overlapping regions for this purpose; for equations, especially in-line equations, we allow zones that are nearly arbitrary regions. In fact they are collections of small (always rectangular) bounding boxes of connected components from the page.

Of the commercial OCR programs, we have found that heuristics for zoning are not described, although sometimes the user has the opportunity to suggest "one column" or "multi-column", or other basic parameters. In some programs, (e.g. TextBridge as supplied by Wang Imaging Pro), the zoning results are not explicitly displayed; in other programs (e.g. FineWeb3.0 by ABBYY) the zones are available and can be edited. Each of these programs allows the user to define zones that are not to be handled as usual text. These zones may be picture zones or (for FineWeb) table zones[7].

Zoning is not sufficient unto itself, except in the simple case of a single column of text. If there are several columns, or even small decorations like page numbers or headers, it is useful to consider the read-order of zones on the page.

How should this proceed? The simplest mechanism is a top-to-bottom reading of all the material on this page, either treated as one large zone, or a vertical concatenation of zones. When zones are horizontally adjacent, problems appear. As a first step we have implemented an entirely "by hand" specification of read-order; this was dictated by our need to handle 2-column documents correctly. We are thinking about writing an "automatic" zoning program which would take a collection of rectangular (or possibly more general) regions and determine a heuristic read-order based on (a) guessing a model and (b) fitting the model against the found data.

### 4.3 Automatic Bottom-up Segmentation

Given a collection of connected components[8] (ccs) we can try to group them into objects that approximate words, lines, paragraphs, columns.

The basic idea[9] is to vastly reduce complexity overall by change all cc's to empty bounding boxes (bbs).

Let p be a list of boxes of cc's on a page. If smear-x is 10, then any two bbs within 10 pixels in the x-direction of each other are combined to a single bb. This, in general, expands to a superset of the two bb, and thus may make the new box encroach within 10 pixels of additional bbs. This

code was used by 3 or 4 Lisp vendors, in the interests of continued maintainability, portions of the code have been moved to Allego Lisp using Common Graphics (based on Windows graphics), and more of it may be used in the future.

[5] After thinking this through we have discovered an essentially similar interface in Adobe's Acrobat, so any minor originality we might claim is probably of no consequence.

[6] I particularly like the stamp idea: you can stamp "REJECT" or "DRAFT" on an *image*.

[7] Each of these programs makes mistakes on one of our standard examples (hal4.tif) because of either mathematics or footnotes (or both) splitting or joining columns.

[8] we explain this notion in detail in a later section.

[9] are give in file boxsmear.

operation is iterated until it ceases smearing boxes, resulting in some number of new "super" boxes. For example, if there are about 2500 connected components on a page, an appropriate choice of horizontal smearing tolerance (for smear-x), iterated until it stops making changes might settling in on about 500 connected components. These might correspond with substantial, but not perfect accuracy, to the words. Given such a collection of words one can smear them into lines by increasing the smear-x tolerance. One can (vertically) smear them into paragraphs. If we have say, 5-10 paragraphs, perhaps in two or more columns, we can go back to a particular paragraph and repeat the processing to get the line-separation and hence the typical text point-size for that paragraph. This will help if paragraphs on the page are in different fonts or sizes. A paragraph may also consist of a title or a displayed equation, or (if we have not otherwise removed it) a half-tone. These operations are fast because they do not require any consideration of the the bit encoding. Should it be necessary to fill the boxes back with the contents, then `stuff-boxes(bbs, src)` restores the rectangular bitmap encoding into the bounding boxes, given the original source page `src`.

A program for interactively changing the smear parameters and watching the changes in groupings is hooked up to our CALZONE program. Other parameters that can be modified in CALZONE include the sizes (area, height) of items that are too large to be used as boxes (typically these are vertical or horizontal rules, figures, or pictures of page bindings) or items that are too small to be used (typically noise).

A plausible direction to follow is to determine, by the changes in the statistics following the smearing, how many words or lines there are on a page. We have *not* pursued automatic determinations of this information, but an automatic approach might look like this: if we start with 2500 ccs, gently increase the smear-x parameter until the number of distinct ccs drops precipitously. Maybe nudge it up slightly to see how stable that smear-x parameter setting is. If the number of ccs is now plausibly the number of words on the page (say 500), compute some statistics: average or median height of a word might be very useful. Continue nudging that parameter up, and if one gets to a number of ccs that is plausibly the number of lines on the page (say 60), take some comfort in the statistics there, as well. One might even find columns this way.

We can propose to use vertical smearing to get paragraphs in approximately the same way: perhaps first do some horizontal smearing to get words or lines, and then vertical smearing. If there is more space between paragraphs that between ordinary lines, this may help. More subtle recognition to deal with indented or "out-dented" paragraphs may be appropriate.

Software tools: `(smear-x bl tol)` takes all cc's in the list of bounding boxes bl (presumably all in a particular picture p's coordinate system) that are within a horizontal distance (border to border) in the x direction of tol, and which overlap in y coordinates (that is, by the most liberal measure possible, are next to each other), and merges them into a single box. This process entirely ignores the "content" of the pictures, and treats them as empty boxes. To do any operations on the contents, you must go back to the original data in the picture p.

`(smear-y bl tol)` is like smear-x except in the other direction.

### 4.4 Classification of Regions: text, non-text

What is the point in doing this automatically? If we cannot do it perfectly (and we cannot), we must be able to handle the situation of a paragraph being identified as text when it is really mathematics, or vice-versa. The actually options may be more numerous: A map may be identified as a diagram but it will also have text. Some text may look like a half-tone texture, etc.

If every processing component must deal with whatever it is given, even if it is *wrongly* characterized, then the only advantage in a classification is an optimization in time (try this first...), but if (say) the mathematics recognizer is powerful enough to understand both mathematics and Roman text (which it must anyway to deal with embedded words), then one can argue: treat everything as math. The math parser will then chew up text and say "Oh, by the way, you might as well treat this as text in further processing, because there is nothing very math-y about it!"

Unfortunately, the math recognizer is not *yet, anyway* as robust for text as the polished commercial programs, and so we do not really want to shove everything through the math recognizer.

We have not included any *automatic* classification programs, but two indications are easily computed: "average density" of a picture constituting a paragraph: displayed math has a lower density; The distribution of large and small characters (or connected components) is another. In a plot of height vs. width, text has a tendency toward tighter clustering than mathematics. We rely on human recognition in CALZONE to pick out math.

We do not use any automatic read-ordering of zones at this time. It is a plausible topic for research, although it would be necessary to have a convenient interaction "correction."

### 4.5 Text Analysis

We include in this section

- Skew: This is the determination of the "slant" of the page as scanned. For typewritten or typeset pages it is usually possible and beneficial to detect (and sometimes) remove the skew by sliding bits. In some versions of skew removal, only text-lines are straightened: individual characters remain unmodified. Our implementation deskews whole pages, and occasionally introduces a slightly disconcerting single-bit shift in the middle of a character.
- Page layout: detecting lines and paragraphs.
- Character Recognition: At the moment we provide tools for isolated character lookup which is clearly not as good as recognition in context. The result from this stage is a collection of glyphs and locations.
- Interpretation: here we examine the positions of the character to arrange them in text lines, or in one of our primary tests, as mathematical equations. We parse the mathematics into the language of a computer algebra system.

## 5 Data representations for pictures, characters

If one has no *a priori* knowledge of what to expect from an image, the first cut at computing with images is likely to be based on dealing with an array of numbers representing pixel values/colors. The value at a pixel location may be an index into a color map, or some absolute quantity. In the case of a 2-level black and white image, an array of zero and one bits seems to suffice. Such an array can typically 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 400 to 600 dots per inch. Displaying a 600dpi 8.5 by 11 inch picture on a 72 dot/inch workstation screen means that we would need a display lineally about 8.3 times larger than the document size, or about 6 by 7.6 feet. It also means we are using about 4.2 megabytes of RAM for the buffer, at 1 bit per pixel. 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 such full 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. Example:

One experimental data set we have been working with is `6.tif`, a black and white `tiff` (Tagged Image Format) file representing a page approximately 8.2 by 5.7 inches in size, scanned at about 600 dots per inch. (More details of this are given in the appendix.) This image, comprising 3408 by 5064 bits, is delivered by our scanner and associated software as a compressed `tiff` file of 72,592 bytes. In terms of black and white bits, uncompressed, it would be $3408 \cdot 5064 = 17,258,112$ bits, or 2.16 megabytes. Thus the TIF file represents 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.

Yet the raster image as an array is actually not so convenient. A basic operation in character recognition and document processing would seem to require access to successive individual bits: one is routinely trying to find out the extent of a uniformly-colored region. Yet access to the individual adjacent bits in a word, one-by-one, may not be the ideal access. A better result would be one that provided higher-level groupings. Ideally, if we had one grouping of bits per character, much of our processing would be done! In point of fact, we have adopted as our representation a version of a run-length encoding, and have found this to be enormously advantageous in time and probably space.

Consider an image to be a sequence of (say, horizontal) scan-lines. Each scan-line is a sequence of $< s, e >$ pairs, where $s$ is an integer denoting the start of a section of black bits, and $e$ is the corresponding end index. The endpoints of these intervals can be encoded in a modest number of bits: For example, with 13 bits we can encode numbers up to $2^{15}$: assuming 600dpi this limits us to 54.6 inches across, which is far wider than our scanners.

Converted into our picture data as intervals, `6.tif` is 3408 lines of intervals. If we scan this along the long dimension (arguably the less efficient way if we wish to compress horizontal lines efficiently) we find this particular page has a total of 78,873 entries in the scan lines, where, as indicated above, each entry is a pair of numbers. How much storage does this take? Our current design uses a few words for descriptive header information, and then:

1. An array of 3408 words (pointers) to the data structure for each scan-line;

2. A scan-line structure of variable length. The conceptually simplest Lisp[10] structure would use 4 words for each pair: two Lisp CONS cells: one for maintaining the "backbone" of the list, and another for the pair `(s . e)` of start and end indexes. This would look like `((23 . 47) (102 . 131) ...)`.

With this encoding, we would use a total of 315,492 words for the lists of pairs. Considering the array and the pairs together we have a total of: 318,900 words or 1.275 megabytes for the representation of this picture.

This is somewhat wasteful: Since the indexes are numbers less than $2^{15}$ two 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 $2^{29}$ are easily packed into one "fixnum" (an encoded immediate number form). So if we care to pack these integers two-per-word, we now have a linked list of fixnums (not conses) of length 78873. Although the exact details are irrelevant, we actually use two 16-bit fields ("short" in C-language[11] parlance) for storing the two parts. For many computers extracting bits by loading half-words from a register (or shifting) is much faster than following a pointer to memory, so this is a savings in space *and* time. We now have:

1. 3408 words, one for each line
2. 1 Cons cell (= 2 words each) for each of 78873 pairs = 157746 words. for total storage of 645 kilobytes.

This is the representation we use. Another step in storage reduction would be to replace the lists by vectors, reducing the storage to 1 word per pair, and the total storage to 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.

## 6 Discussion of routines

Here we describe in brief the collection of our routines intended to be used by experimenters in OCR. We do not doc-

[10] Why did we program in Lisp? Several reasons:

1. We like Lisp, especially for exploratory programming.
2. Natural data-structures are linked lists.
3. Memory leakages, a common problem in C or C++, are eliminated by automatic memory management.
4. There is a convenient built-in package ("Common Windows") for bit-map display. Interaction with the program for learning and debugging have been substantially assisted by the easy availability of this set of routines providing access to the X-window interface. This 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.
5. (After some work) convenient access to files as formatted from the scanners.
6. Lisp programs tend to be quite portable: Portability to other implementations of Common Lisp would require no alteration except in the interface to libtif, a public domain package we use for input and output of tiff format files. This alteration would consist of a transcription of the foreign-function call mechanism from the Allegro dialect to some other form.

[11] Partly in recognition of the popularity of C and C++, and partly to see how speed would be affected by other language implementations, two versions of the scan-line to encoding program were written, one in Lisp and one in C. Somewhat to our surprise, while the C code was faster by a factor of two in 1996, the 1999 compiler technology and computer hardware has changed the balance: C is not particularly faster now (1999). Furthermore, a cleverer algorithm suggested by Reiner Staszewski easily coded in Lisp has made the Lisp code twice as fast as C!

ument all the detailed algorithms here; considerably more documentation is provided with the program text.

## 6.1 Input, conversion

`tiff2pict ("filename")` Given a string that is the name of a file, `tiff2pict` reads that file in TIF form, presumably a b/w binary scanned file, and returns a `picture` structure: width, height, x-y coordinate of the lower left corner, the collection of rows, the source filename, and some indication of content. For a newly read-in picture, the (x,y) setting is always (0,0). The height of the picture is exactly the number of rows in the collection. The width of the picture is taken as the width of the originating file. The content is the string "a full page". The (x,y) setting is in any case a pair of non-negative integers. If it is, say (10, 100), this means that the 0th row represents line 100 of the page, and that on that row (and every row), the 0 mark is at column 10 of the page.

`pict2tiff (p, "filename.tif")` writes back out to the file system the picture `p` as a `tiff` file with encoded name `filename.tif`. The (x,y) coordinates in `p` are ignored, and assumed to be (0,0).

`pict2bit (p)` converts a picture structure `p` into a bitmap `b`. A `bitmap` is an official data structure used by the Common Windows package in Common Lisp. 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)` transfers (`bitblt` = bit-block-transfer) the image of the picture `p` starting at its lower left, into the window `WIN` at its lower left.

A brief digression on the topic of bitmaps and Common Windows.

You might think that the provision of this structure implies support for other operations. This is true in that it extends to drawing lines on it. Analysis of bitmaps is not so well supported; if it were, we would probably use it more. In the interactive portions of our programs we use it heavily.

Bitmaps are restricted to a total size of 16,777,216 bits so that arrays that are larger than 4096 by 4096 are too big to handle. This is not usually a problem because conventional display systems today handle about 1000 by 1000 pixels, or 1/16 of that bitmap. And there are probably better things you'd like to do with your memory, as well as better ways to represent data off-screen.

While a full page at 8.5 by 11 inches at a typical 300 dpi (dpi = dots per inch) CAN be converted to a bitmap, in our processing experience this has seemed inadvisable. We suggest you scale things down so you can see the whole page using scaledpict2bit, (see below) or perhaps look at a subsection of the page, using smallerpict (see below).

Why?

Simple: Most workstations display about at 72dpi. To fully display every bit on such a bitmap would require a screen about 3 by 3.8 FEET (or 4.8 foot diagonal). You can display about 1/9 of it on your typical 1-million pixel screen.

At 600dpi, which is about the highest resolution for scanning that is used at all frequently, the bitmap is far and away too big for a workstation, which might display 1/33 of the picture. In fact the array is 33 megabits; if we use color for overlays or shading, this is expanded to 1-byte per pixel. For 24 bits of color per pixel for a color display, this means a LOT of memory. Fortunately, for document processing this is much more than we need. Some color image processing can require such expansion, and that's why big machines are recommended for that business.

Pict2bit actually has two sub-cases, the simple case in which the (x,y) coordinates of the lower-left corner are given as (0,0), and the case where some other location is given. In this latter case, the program provides a bitmap that (nevertheless) starts at (0,0), and so may include big areas of blank bitmap. In practice, it seems we generally "renormalize" our picture encodings so that that they have origins of (0,0), and if we want to produce a bitmap of a particular character in (say) the upper-right corner of a page, we should separately remember its (x,y) origin, extract and convert just that little rectangular picture to a bitmap. We can paint it anywhere on the screen, and if we want to put it in the "same" place as it occurred on the picture, we will use its saved origin to position it. A program that assists this is `smaller-pict` described just below.

The final bitmap has the same height and width as the picture.

`smaller-pict (pict x0 x1 y0 y1)` takes a picture `pict` and 4 integers. It extracts from `pict` a picture starting at x-coordinate `x0` and continues up to but not including coordinate `x1`. If `x1` is too far to the right (off the page, so to speak) then the smaller picture will be similarly truncated at the border. The y-coordinate is treated the same way. All the coordinates are re-adjusted so that (x0,y0) is now at location (0,0) in the returned picture.

`bit2pict (b)` takes a bitmap `b`, as might be produced by interactive editing, and returns a picture structure. Since Common Windows programs use the bitmap structure as the underlying support for its drawing canvas, pop-up menus, etc, it is plausible to (for example) write character editing or similar manipulation programs using bitmaps, and then translate the results to a picture form. It takes the lower-left corner of the bitmap as coordinate (0,0).

`pict2tiff (pict, filename)` takes a single picture structure and writes it (fairly rapidly) to a file in a format that is fast to read back in. This assumes the origin of the picture is at (0,0). This primitive routine is probably not of substantial direct use except to rewrite a page that has just been deskewed. Some of the encoding details (e.g. compression, etc.) are set to correspond to the last tiff image that was read in. A careful programmer using this procedure may look at it as a model rather than a complete module for simple use.

`writepicts2file(plist filename)` In this program a list of pictures `plist` is given, rather than a single picture. The output is NOT in tiff, but in a form that is more quickly read and written by lisp. The pictures are written out in the order given in the list. Another file, filename.dic is written to contain some dictionary-style material; a kind of symbol table for the contents of the pictures.

Although we have not tested this extensively for efficiency, assuming that the encoded file is a page of text, a file written out in this way is only slightly larger than a tiff file. It could, however, be much larger if the encoded material consisted of random bits. 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 pages and pages of text where access to individual picture components (e.g., characters or perhaps words) over a whole document are 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. For purposes of multi-page document processing we can read/write pages to/from disk at a fairly rapid clip. (on an HP 9000/715 workstation, .79 sec. to write, 1.1 seconds to read back in a whole page of 3259 connected components. Access to particular character images would be yet faster. A normal lisp text storage format rather than a byte stream, is about 30 times slower. Using tiff as a disk format depends on compression, but seems to take a minimum of 2.9 seconds to read, 4.2 seconds to write, and re-computing connected components takes about 3.2 seconds, and doesn't have the advantage of keeping individual characters in place. The times on a 200 Mhz Pentium Pro computer using Allegro Common Lisp are slightly faster than this, 2.2 and 2.9 seconds, June, 1998.)

## 6.2 Utility Programs

We have made substantial efforts in polishing code to produce a fast deskewing algorithm and a fast connected-components program. We therefore describe these in much greater detail.

### 6.2.1 De-skewing

Pages scanned or printed by a mechanical device are ordinarily read at a slight angle. Humans do not usually have difficulty reading such skewed pages. Indeed, small skew may not not even be noticed. It is advantageous for computer processing to remove skew when possible. The major reason is that regions (lines, paragraphs, zones) of common text, tend to be more easily isolated in properly grid-aligned rectangles. Such rectangles can be described by one origin (x,y), a height and width. An unaligned rectangle typically needs four full pairs of numbers, and the display of un-aligned rectangles tends to show uncomfortable "jaggies". While it is possible to deal with skewed text, and some OCR programs do so quite well, we prefer to deskew when possible.

Three programs are provided:

`deskew-pic(p)` given a picture `p` produces a deskewed version of it. In fact, all this does is find the skew and then unskews the picture by composing the two programs described below.

`find-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 $\pm 10$ degrees. A positive value for `d` means the picture appears to be rotated clockwise by that angle about a point at the lower right. A negative value suggests a skew counter-clockwise around a point at the lower left. Actually as many as 4 additional key-word parameters can be provided to this program: `find-skew(p :min m1 :mid m2 :max m3 :skip s)` The minimum, maximum, and midpoint of the angles expected can be set. By default these are set to (-10, 0, 11). The skip parameter (default 1) indicates how many lines should be skipped in testing skew. Skip=2 uses every other line. This speeds analysis by about a factor of 2. For high-density pages, skip=4 seems to work about as well as skip=1.

`UN-skew(p d)` given a picture and an angle `d` in degrees, produces a new picture that is unskewed.

There are a number of recent papers (refs) written on deskewing algorithms, but from our limited experience it appears that a simple, fast and therefore useful approach can be based on a simple observation. Compute the distribution of black dots on a line-by-line basis in a page of text (or mostly text). If the scan lines are aligned with the text grid, there will be substantial numbers of blank or nearly-blank lines, and substantial numbers of lines with considerable black density. In the case of horizontal lines (say as a table-rule, underline, or mathematical fraction divide-bar), the line may even be a majority of black bits.

On the other hand, if the scan lines are at an angle to the text (think of it as 30 or 45 degrees), then the distribution of white and black bits will be far more uniform: there will be very few purely white or purely black lines.

A statistical measure of the variance of the distribution of bit-counts is easy to compute: we, in effect, do a "ray 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 truly inexpensive, the cost increases, but slowly, as the angle increases. Counting the number of black bits in a full row is a simple operation, and we can compute the variance of `6.tif` at zero degrees, examining 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 $(1/\tan(\pi/180)) = 57.29$. Hoping up one row and skipping to the interval that encloses the appropriate bits can eventually become time-consuming[12].

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 on a few scanned pages, accuracy is relatively good even if 9 out of 10 lines are skipped.

Finding the best angle, that is the one with the highest variance, is done with a conventional numerical search. The function being maximized typically has a global appearance of a single maximum with a rather "flat" top. The detailed local appearance at that top is more fractal in nature, and so finding the precise maximum may be wasteful: an approximation good to 0.1 degrees should be achievable.

### 6.2.2 Correcting Skew

Each of the transformations is a shear: think of a stack of dominos sitting on a table. Pushing it sideways so that each domino is shifted a distance proportional to its height, is a horizontal shear. This is a row-preserving transformation we call a "slant." If each row going upward progressively moves further to the right as the row number increases, we have "italicized" the picture. The second kind of shear is at right angles to the first shear: a column-preserving transformation we refer to as a "tilt". A combination of the two is nearly a rotation for small angles. An actual rotation would require that in each of the two shears, the amount of shifting varies as the row (or the column) changes ([4] p. 829). In particular, their row-preserving transformation is $(x,y) \rightarrow (x\cos(\theta) - y\tan(\theta), y)$ and the column-preserving transformation is $(x,y) \rightarrow (x, x\sin\theta + y\cos\theta)$.

How far off is the maximum error in using our near-rotation? The approximation we use is based on the fact that $\sin(\theta)\tan(\theta)$ and $\theta$ (in radians) are approximately equal for small $\theta$, and that $\cos(\theta)$ is approximately one. We use $(x,y) \rightarrow (x - y\theta, y)$ $(x,y) \rightarrow (x, x\theta + y)$ For an angle of one

[12] 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 ...).

degree (.01745 radians) and an 8.5 by 11 inch page scanned at 300dpi, or even at 1000dpi the true rotation point given by the formula is off by one pixel at the worst. (A page with such a slant might have been inserted in the copier offset about 0.2 inches out of 11).

At 3 degrees, and 300dpi, the center of the page is displaced $(\Delta x, \Delta y) = (1, 3)$ pixels, about 0.01 inches from the true location, and the worst case, the upper right-hand corner is displaced by (4, 4) pixels, still less than 0.02 inch off the correct position. Since this error is encountered at 3 degrees, how likely is this? In fact a 3 degree error represents a displacement of about 0.58 inches out of 11; this would be a quite noticeable skew in inserting a page into a copier or scanner.

Nevertheless, what about more extreme angles? At angles approaching 10 degrees, or about 2 inches out of 11, the deviation of our formula from a true rotation is substantial, with a (33, 47) erroneous displacement at the upper right-hand corner, nearly 0.2 inches. A sequence of several smaller shears, rather than just two, may be preferable in correcting such a tilt.

Note that deskewing necessarily puts a jag into the data at the point of the corrections. One alternative approach to deskewing would be to maintain all intra-connected-component relationships and only deskew BETWEEN such objects. This keeps the letter shapes unchanged, but levels-out the base-lines of text. For small letters this is fine, but presents a problem for those connected components, the long divide bar in fractions, for example, that we would also like to deskew. For the present, we deskew full pages and suffer the consequences of the occasional letter with a jag in it. An alternative that may be easy to implement in one dimension is to avoid inserting a shift in a foreground color if one can make a correction within a few pixels, in a background color. Note that even a highly accurate deskewing calculation can have local discretization problems, resulting in a re-alignment of the apparently tilted line of dots

..........................................................................

into one that is only nearly-aligned:

..........................................................................

Another approach for de-skewing adopted by some scanning software is to *not* re-represent the page, but to keep the information of the skew angle for following base-lines at an angle across the page.

Occasionally papers are scanned either accidentally or purposely at right angles (or occasionally upside down). Scanning software can be adjusted for this, but sometimes is not. We've 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[13]. On a typical page, it may be better to separate the page into connected components (1.6 seconds) and rotate each of a few thousand small characters: this program requires about another 0.55 seconds. The savings accrue by not having to "rotate the white-space". (This requires specifying the page-width as a second argument to `rot90` because the axis of rotation cannot otherwise be deduced.) We also have written `(rot-90 pic)` for a 1/4 rotation counter-clockwise as well as `(rot180 pic)`. For speed, the last of these changes the data "in place" and hence destroys the data structure. It can be easily recreated by rotation of another 180 degrees.

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

### 6.2.3 Morphology and Half-tone removal heuristics

Morphology transformations on picture forms is easily performed if they are "one dimensional" and correspond to the run-length-encoding direction. That is, one can easily write a program that takes each interval and dilates it. What we programmed were two transformations: `merge-close-intervals(line n)` which merges any two adjacent intervals of foreground color (usually black) if they are separated by fewer than n bits of background (white). We also programmed `remove-narrow-interva n)` which erases any interval of width less than n. These programs can be "mapped" over the lines in any picture by `map-over-pict(pic, function)`. If we wish to run these transformations in the vertical direction, we can rotate the picture by 90 degrees, run the transformation, and then rotate by -90 (or +270) degrees.

We programmed a "blotching" operation on a picture to join, by horizontal or vertical mergings, half-tones images. The objective is to produce a single (or a few) large blob(s) that can be easily dismissed as "not text", and wiped out wholesale. (in fact, it appears that all large objects that do not fit into a horizontal or vertical line model might as well be deleted along with noise.

### 6.2.4 Connected Components

`con-pict(p)` produces a Lisp list of the connected components of the picture `p`. Each one of the components is itself a picture structure of a rectangular element of the original large picture, with an (x,y) origin at the lower left of the bounding box of that component. In many cases these components correspond to the characters on the page image, although they can be both character fragments or artificially merged characters. The components are not necessarily disjoint. Indeed, the bounding boxes of adjacent italic characters can easily overlap. Each component picture includes, however, only those bits that are connected. The current version of this program (including Lisp garbage collection times) finds about 2000 components per second on the file `6.tif` mentioned earlier. This is probably not an entirely typical example since many of the components are small pieces of the binding and page edges.

Another, probably more typical sample: an 8.5 by 11 inch typed page in courier typeface was found to have 2134 connected components in 1.6 seconds, a rate of 1334 chars/sec.

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 [14].

A program to compute the reversal of con-pict is available: `manypict2one(plist)` takes a list `plist` of (perhaps many) pictures, possibly overlapping, and returns one picture structure. This is an especially useful operation if you believe that the connected component breakup of a large picture is wrong, and one should reconnect some of the pieces. In constructing the single resultant picture, an x,y origin is computed that is the minimum `x` and minimum `y` origin of any of the component pieces.

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

a. We must also manage the fairly common situation where a connected component is *not* a complete single

[14] If we were using pixel arrays, we would expect the processing time to decrease by a factor of four. In own representation, while the number of rows would be halved; we would not expect the number of intervals per row to halve as well. Intervals would be lost only when details were lost to the decreased resolution.

character. (We can reassemble them via `manypicts2one`, though).

b. The underlying assumption seems to be 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 as a definition of a character, not only the black bits, but also the space about the bits: 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 in some sense 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 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.

## 6.3 Editing Utilities

Programs for directly editing bitmaps interactively would seem to be more plausible, so we provide a simple substitution program to replace an edited piece into a larger picture: `replace-pict (smallpict bigpict x y)` takes two picture structures and overwrites the bigpict with the small one, starting with the smaller one's lower-left location on location x,y on the larger one, and extending according to its size.

A number of utilities for dealing with individual rows or pairs of rows are available, documented in the source code.

## 6.4 Learning, Clustering and Identification

A simple-minded but, in our experience, largely effective way to approach the decoding of most of a document is to start with two frequently true (but in general, false) assumptions, and proceed to do as much recognition as possible.

First assumption: each character is a single connected component. This assumption fails for the common characters i,j, and punctuation characters such as `=;:?!%`. It also fails for characters with defects that break them into pieces.

Second assumption: each connected component is exactly one character. This fails for the sequences "fi" and "fl" and "ffl" in some fonts, and is often a false statement for realistic scans. That is, the assumption may fail when characters touch by design or through noise.

Nevertheless, we can proceed in this simple-minded way to characterize each of the connected components by some set of properties, and cluster them so that all components that (say) resemble the letter o are together, and those that (say) represent the letter c are together in another cluster.

### 6.4.1 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. Unfortunately this rule does not work very well, at least judging from human perception: characters that *to the human eye are identical shapes*, but are in reality shifted slightly in position relative to each other, are somewhat distant by this metric. Similarly, a human may identify two characters as the same even if they are slightly different in size. A human will not take note of small differences especially if the occurrences are separated in time or location) be

Nevertheless we have programmed a rather efficient `count-bits-in-xo` which takes two bitmaps `p1` and `p2`, the first of which, `p1` is presumed to be a dictionary bitmap of a standard character, and the second of which is a page which one suspects of containing 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.

Any number of better distance metrics can be found. A useful metric must naturally be fast to compute as well as likely to agree with the human reader. We have experimented with using Hausdorff distances; this is appealing but we do not have evidence yet that this is significantly better considering the expense [7].

Another approach, corresponding to a somewhat de-focussed recognizer has some appeal. This technique appears to be used by some commercial programs (Bokser in [6]), and is described below.

### 6.4.2 Computing property vectors

As one classification technique for identifying rectangular regions as potential characters, we divide the area into $n$ by $m$ regions and count the ratio of black vs. white bits in each region. (we currently use 5 by 5, 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 (for the 5 by 5 scale) averages (for page35) to about .51ms, or about 1958 computations per second. (1.09 seconds for the 2134 items on `page35`).

We can compute the (optionally, weighted) distance between property vectors $p$ and $q$ as the Euclidean distance: $\sum_i w_i(q_i - p_i)$. We could use this for finding the nearest neighbor, or find the centroid of some cluster that was closest to a new data point. We found it advantageous to make an "absolute" grouping for complete characters; that is we would 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.

The important of the height-to-width ratio is critical in distinguishing characters that essentially fill a rectangular region. A perfect horizontally oriented rectangle (a line) or a perfect vertically oriented rectangle (a "rule") would both be "all black pixels" in a rectangular field, and therefore could not be distinguished from a square. In fact, a period or dot above an "i" would not be far off from either of these, were it not for the h/w ratio.

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

Another variation would identify each black dot as a kind of Gaussian distribution, and therefore a pixel at the corner of a region would contribute to the pixel density in adjacent regions as well. A simple way of approximating this would be to have overlapping regions for statistics gathering. This would double-count the edge pixels, so a better approximation to reality may be to overlap the regions but then discount edge-pixels by half (and corner pixels by half again). We have not implemented this option.

Another routine, `count-bits-in-xor-of-bitmaps`, computes the count of the bits in the exclusive-or of two bitmaps A and B. This is done rather faster than computing the x-or followed by a bit-count: First note that we can get a quick upper and lower bound on $K$ =count(xor(A,B)): assume we pre-compute count(A) and count(B). Then let $M =$ max(count(A),count(B)), $N$ =min(count(A),count(B)). $M - N <= K <= M + N$.

Also if we compute $L$ =count(and(A,B)), then $K = M + N - 2L$.

The big hazard here is that to make this a good detector of similarity, the bitmaps for A and B must be aligned accurately. To achieve this with minimal cost, it seems appropriate to pre-compute for each template, a number of shifted images. This tends to proliferate templates dramatically: In addition to shifting North/South and East/West, one can shift NW (etc.).

(In the past we have used other measures such as the Hausdorff distance; this is less sensitive to minor shifts or rotations. We are unconvinced that the complexity of this computation is justified.)

There are, of course, other possible property vectors that can be computed. A set of 17 properties of letters was suggested by F. W. Frey and D. J. Slate Letter Recognition Using Holland-style Adaptive Classifiers, Machine Learning vol 6 no 2 March 1991, These include 17 small integers: horizontal and vertical positions of the enclosing box relative to the imputed character position, height, width, total number of pixels, average x position and variance, average y position and variance, mean x-y correlation, mean $x^2y$ mean $xy^2$, and various edge statistics. We have experimented with the 20,000 characters are offered in this benchmark, and find that we can partition the data points automatically into about 850 clusters, many of which consist of just a few of the 26 letters. Given our algorithm, the properties seem inadequate to reliably distinguish Y from V etc.)

REWRITE HERE

After some experimentation and reading, the set of properties we chose was as follows. Consider copying the alleged character into a bitmap w by h. Divide the bitmap into 5 x 5 regions. In each region compute a number from 0 to 255 to represent the gray scale in that region. 255 represents "all black (foreground)" and 0 represents "all background". This 5 x 5 grid gives us 25 "property dimensions" for each number. We have some misgivings about using exactly these numbers, using the same number of division, and using non-overlapping regions. (We speculate that overlapping the regions is a good idea.)

We add two more: the height-to-width ratio normalized to be between 0 and 255 (where 64 is square, 128= high, 32= wide, 0 is very wide, and 255 is $> 4 : 1$), and the absolute height in pixels (although if the height exceeds 255, we use 255).

The similarity /distance between two characters is computed in a somewhat *ad hoc* way: if the height to width ratio differs by more than 20/255, or if the characters differ in height by more than 20 pixels, the characters are different (large distance between them). Otherwise the difference is the sum of squares of differences of the corresponding gray levels of the two characters. We tried some other techniques and the benefits of more expensive techniques (e.g. Hausdorff distance) seemed minor [1].

It is not our intention to defend our metric for several reasons:

1. We provide the programs, and you are free to change the property vectors, as well as the distance metric in any way you wish.
2. Alternative techniques for selection of characteristics and for training, (most notably using neural networks) are popular and can be found described in the literature.

We chose this technique as among the simplest, and one that seemed to do a reasonable job on our test cases[7].

We wrote several clustering programs on the simple principle that we would cluster all connected components into the same bucket as long as they were within some modest distance of the (running) average of the components in that bucket. Our training program actually has two tolerances: `really-close` (e.g. for our normalization, this meant about 10,000 units) which indicates to us that this certainly belongs in the given bucket, visually. A larger number (e.g. 400,000) is used as some outside tolerance `toler`: If the distance from the closest cluster to a new component is between `really-close` and `toler`, then the (human) assistant is asked if this letter is indeed different distinct from its closest cluster or not. If it is actually not distinct, it is merged into the cluster, adjusting the cluster's running average. (This is how the cluster "learns"). If the new component is distinct, then a new cluster is formed around this form[15].

Our experience has been that reading a page with "suitable" values of `really-close` and `toler` indeed breaks the contents of a page into clusters, mostly without human intervention. In our tests some clusters are not truly useful – in our current program run on `page35`, (see appendix) there are several clusters consisting of various size dots: the dots from the letters i and j, the dots from colons, semicolons, or periods. Several clusters may be scraps of noise, broken letters, etc. Occasionally a cluster will be a single object corresponding to an unusual joined letter. It is in fact quite possible to have clusters corresponding to letter combinations that are routinely joined, such as the kerned ff or fl combinations, or letter combinations that just happen to touch accidentally.

After a collection of clusters has been formed, it is possible for a human to "teach" the computer the conventional

[15] For example, a human might be asked if a (perhaps noisy) e is a member of the cluster to which it is closest. In a noisy document in which no "e" cluster has been previously formed, the closest cluster might be mostly letter c's (and perhaps some broken o's). If such a noisy example is viewed in isolation, a human may not in fact be able to tell that it is an e — it is only after some enclosing context is displayed that a human's high level of accuracy is achieved. Given a choice of placing the noisy e into the closest cluster or starting a separate one, it may be preferable to start a separate cluster, even if there is another cluster of non-noisy (but more distant) e's.

ascii character for each cluster. In the case of multi-part letters like i or j, we associate the identity with the "non-dot" part. These identities can then be used in a translation of the page into ascii. How can one tell between the dot over the i and a period? The location relative to characters on the line seems to distinguish them fairly well, though not perfectly.

Once a cluster/identification matrix is set up, additional clean pages from the same document can be processed with (we believe) high accuracy.

It becomes clear after some experimentation that an adequate setting of tolerances for one document may differ substantially from the setting that must be used on another. Various alternative formulations can be proposed (and some may be implemented soon) based on a better way of determining "distance from a cluster" in a multidimensional space. [2]. Conceptually, if instances of the letter A occupy a very narrow cluster in space, but instances of B are more spread out, then a letter whose properties place it "halfway" between A and B is more likely to be a B: In a sense, the standard deviation of the B cluster is larger, and thus the range of acceptable alternatives for B may be larger than that for A.

*********

## 6.5 Other Bottom-up tools

```
Local Morphological transformations
Parsing of rectangular regions / Math
Other kinds of parsing
```

THE FOLLOWING UNORDERED PIECES ARE GATHERED HERE FOR REFERENCE...

IMPORTANT: DOCUMENT STRUCTURE

Maybe some morphology? Trim very thin connections, nubbins; join close matches? (All risky, and maybe too early to do so.)

Finding perfectly clear lines to separate lines of text is not entirely adequate. They must be clumped heuristically.

a sequence "xxx iiii jjj jim" has a horizontal break between the dots of the i and the character bodies.

Technology used: All of the library `libtiff` is available in the lisp system. The specifications are mapped from the usual C-interface into Lisp, roughly as follows. We have prepared a special version of the Allegro Common Lisp system by loaded it with `libtiff`, and then dumping it back out. Thus subsequent users do not have to specify the library, nor wait for the loading. Next, any functions that are needed are mapped on to lisp names: for example, the declarations for `TIFFOpen` and `TIFFReadScanline` look like this:

```
(defforeign 'tiffopen
  :entry-point   (convert-to-lang "TIFFOpen")
  :arguments '(string string)
  :return-type :integer)

(defforeign 'tiffreadscanline

    :entry-point (convert-to-lang "TIFFReadScanline")
    :arguments '(t array integer integer)
    )
```

Any conversion of Lisp's data types (string, integer, etc) to the C conventions is done automatically, with the exception of return-values by reference. In such cases, the Lisp programmer passes a fixnum array `A` of length 1 as the argument to the C program, and on return, the value of `A[0]` (or in Lisp, `(aref a 0)` is set).

We wrote one program in C, called solely by `tiff2pict` to convert the tiff scanlines into interval endpoint data, and continue to use it, although it provides only about a factor of 2 speed improvement.

*Implementation notes:* tiff2pict calls a C-language routine defined in the file cfuns10.c with two arguments. one is a bit array that has just been filled in by `TIFFreadscanline`, and the other is a scratch-array of fixnums. The vector of bits represents background and foreground colors in the picture. Unfortunately, the order of the bits is backwards in bytes, so we have to read them out (this is what the C routine does) in order 7-6-5-4-3-2-1-0-15-14-13-12-11-10-9-8 etc. When the color changes from background to foreground, we record the index or position of the start of a foreground interval in the next 16 bits of the fixnum array,. When the color changes to background, we record in the next 16 bits of the fixnum array, the end. The memory for this working array has been provided by the Lisp program. Now the array looks like

| $s_0$ | $e_0$ | $s_1$ | $e_1$ | $\cdots$ |
|---|---|---|---|---|

(each segment is 16 bits). on returning to Lisp this object as an array of 32-bit fixnum quantities. Because we are going to be handling many such vectors, of varying lengths, and in a manner that may require us to add or delete intervals from the middle, a linked list representation of this would seem to be a good representation. Thus we can represent a scanline from the file (a row in the picture) by a list like ((s0 . e0) (s1 . e1) ....) or we could have a list of pairs of 16-bit numbers packed into 32-bit INOBs "Immediate Number-Like Objects". That is, n0 = s0e0 packed together, n1 = s1e1 packed together, etc. Thus the row is now a list (n0 n1 ...). Since INOBs, as long as we do not use ALL 32 bits, but only 30, are packed nicely by lisp into the same space as a pointer, very little excess storage allocation is needed.

There is nothing essential in the computing requiring the C-language program, and in fact we have a version entirely in Lisp: This also calls the tiff library, but does the fancy footwork for decoding, directly in lisp. This is called `tiff3pict`. In fact, the Lisp system's handling of bit vectors seems sufficiently fast that we left the program `pict2tiff` without a C-language helper.

The major operator of the C program is to count how many bits in a row are the same color, where "in a row" involves counting backwards in a byte. (Presumably this is caused of byte-order differences between our scanner's native mode and our machine's native mode.)

## 7 Acknowledgments

Thanks to undergraduates Alex Bui, Katherine Marsden, Paul Tero for prototypes of some of the programs. Continuing discussions with Gary Kopec, Jon Hull, Dan Halpern, Ted Einwohner, and other participants in the document-structure analysis seminar at UC Berkeley were also useful in the design, implementation, and testing of these modules.

## 8 Appendix 1: Descriptions of Sample Texts

`page35` is page 35 of "HAKMEM" (MIT Artificial Intelligence Lab memo 239), February 1972. Hakmem is a collection of random programs, data, problems, and "hacks". Most of this memo, and all of page 35 appears to have been printed with a Courier typeface Selectric (tm) type-ball, with a few math and superscript symbols from a Symbol type-ball. `Page35` suffers from being reproduced by offset and then copied xerographically; it also has minor defects like staple holes. This page was scanned at 600 dpi and then reduced to 300 dpi in order to try commercial OCR programs on it. It has a skew of about -0.22 degrees. The 300 dpi version which is 2548 by 3300 pixels. Commercial OCR recognition of this page is fairly successful.

`6.tif` is from a double-page (pages 336–337) of a table of integrals by Prudnikov, Brichkov, and Marichev: Integrals and Series, volume 1. This was printed in 1983 by the USSR government printing office in Moscow on low-quality paper. It was copied on a xerographic copier once to make it easier for us to scan mechanically. The right-half part (page 337) is skewed by about 1.5 degrees. We did our experiments on the left side, after cropping out some edge defects. This document was scanned at 600dpi. It is 3408 by 5604 bits. Commercial OCR of this page results in essentially no useful information.

`form001.tif` is from a double-page (pages 254–255) of a table of integrals by Gradshteyn and Rhyzik, specifically entries 3.161.3-6 on page 255. Although this was published by Academic Press, we believe it was produced from plates that were mechanically copied from the original Russian version. Again, it was copied on a xerographic copier once to make it easier for us to scan mechanically. The right-half part (page 255) was deskewed by our software. We did our experiments This document was scanned at 600dpi. It is 1831 by 1524 pixels. Commercial OCR of this page results in no useful information.

`hal4.tif` is a full page from a journal (The American Economic Review) of an article by Hal Varian It contains a modest number of displayed equations and some in-line mathematics with substantial text. It was cleanly scanned, and has only a few joined characters. It is 4192 by 5696 pixels. Commercial OCR by Xerox Scanworx of this page provides nearly perfect recognition of the text words. The display mathematics embedded in the text is rendered as either gibberish or nothing. These failures have the additional consequence of upsetting the page-layout deduction and hindering the zoning. A perfect separation would yield a headline, a page number, two columns, and a footnote area.

### 8.1 Appendix: A scenario

What might one do with this package?

0. start up a lisp and load in the package `:ld int (in-package :tiff)`

1. read in a page, say the page 35 hakmem image.

```
(setf p35 (tiff2pict "page35.tif"))
```

This particular page has width 2548, length 3300, is in ccittfax4 format.

1.1. Review the page. First initialize the graphics if not already done. Then scale the picture say, down to 1/8 size, and display it.

```
(init-test)
(scaledpicts (list p35) 1/8)
```

2. deskew the page, calling the result (`s35`

```
(setf s35 (deskew-pic p35))
```

The deskewed page has width 2562 length 3311.

3. find the horizontal breaks, namely those locations where there is a perfect cut running through the page. The program `horiz-break` takes an array of run-length encoded lines and returns the beginning and ending of the non-blank lines as a single list of start-end encodings. These represent pairs, in order from top to bottom, of those regions of rows that are non-nil. (Yes, this is somewhat disrespecting the abstraction, but it saved us some programming)

```
(setf breaks (horiz-break (picture-rows s35)))
```

There are 59 breaks

4. Produce a list of the heights of each of those rows.

```
(setf heights (mapcar #'(lambda(x)(- (end x)(start x))) breaks))
```

5. Filter out the too-short lines as noise.

```
(setf lineheights (delete-if #'(lambda (h)(<= h 3)) heights))
```

There are 47 remaining lines.

6. Find the median lineheight `(defun median (l) (elt (sort l #'>) (truncate (length l) 2)))`

```
(median lineheights).
```

This results in the number 43 (pixels) for line height.

This information can be used for guesses at character size and similar data that can be used for segmentation purposes.

7. The basis of our character recognition is the assumption that we can correctly segment text into characters by looking for connected components. As a rough approximation, it works reasonably well for most of cleanly typeset mathematics. More generally, in the case of full text and especially noisy text, it is demonstrably false, and any attempt to get high accuracy this way will have to be modified substantially. To be more specific about the kinds of information we necessarily forego, and which would undoubtedly improve accuracy:

- font size estimation, which allows us to split merged characters or join fragmented ones with some confidence, including dots over i's etc.
- baseline computations (classifying letters jgyp as descenders, other letters or punctuation as "short" or "tall" etc.).
- letter-pair frequency identifications ("th").
- references to dictionaries.

There are other heuristics possible, of course, but unless they are also applicable to mathematics, we did not implement them. (See, however, OCRchie [5], a student project we supervised.)

In any case, here is how we can find connected components.

```
(setf cc (con-pict s35))
```

Now `cc` is a lisp list of a few thousand little pictures. For this page, 2100. Many are letters, but some are fragments of letters like the dot over the i, or broken pieces of E, etc. If you want to see them, try (scaledboxpicts cc 1/8). If you want to see them and the letters both, and larger, consider

```
(clear *w*)
(scaledboxpicts cc 1/3)
(scaledpicts cc 1/3)
```

8. If we want to scrap all parts of cc that have less than some area, say 16 pixels inside, we can do this: `(setf cc (filter-out-noise 16 cc))`

Why did we pick 16? By trying various different area values we found that the number of connected components of area $> 2$ was 2075. Of area $> 16$ was 2058. Of area $> 30$ was 2052. Of area $> 40$, 1926. The precipitous drop between

30 and 40 suggests we are starting to wipe out important components that may even be letters.

9. Suppose we wish to recognize the connected components and treat them as characters, but we have no idea what characters there are on that page. One approach, and one for which we provide programming support, is to look at each putative character and compute a property vector for it, group these in clusters according to some "close-enough" criterion, and present each cluster to the human user for examination, identification as to "this is an italic e" etc.

The program `cluster` has this characteristic:
*** ***

There are several possibilities at this point.

a. The clusters may be just right, and all and only the letter e's are in one cluster, the letter c's are in another, etc. This is highly unlikely, but even if we had perfect clustering, we would still have separate pieces for characters like `i,j,:;="%`.

b. The clusters may be too large. For example, the e's and the c's would be in the same cluster because they look nearly the same.

c. The clusters may be too small. For example, some of the e's may be in one cluster and some in another. This is not mutually exclusive with the previous defect. We could have two or more clusters, each with some e's and some c's.

How to deal with them.. . one can interactively edit these cluster

For example

We can do more here, like cluster the characters into groups that resemble each other, paste the dots on the i's and semi-colons. Identify half-tones, figures, italics, mathematics zones. ... more...
*************

## 8.2 Appendix on Editing

A fun student project, might be to implement "Image EMACS" (ref. CACM 1994 article by Kopec et al)

given a page of pictures as connected components, then grouped as words, and text lines, provide the following operations:

```
 cursor character motions: F B P N (forward back previous-line next-line)
 word motions F B
 line motions A E
delete character:  D rubout
group connect components as character
learn (characters)
delete word
delete line
search
fill-paragraph

display in artificial font
copy
yank
insert typed characters?
OCR
```

```
More anecdotal info..  form001.tif, a file produced by scanning pages
254-255 of Gradshteyn at 400 dpi produced a picture of width 4384,
height 3392.  Time to read in, 2.01CPU+.2GC; Time to compute connected
components 1.5CPU+3GC; 3332 found.  After filtering out spots of area
$<4$, the number of components is 2403.

Time to convert and display (bitblt to window) a 1/4 scale version
of the 2403 characters: 8.4CPU+.44GC seconds.
```

```
This double-page spread was copied by a xerographic copier, and
then run through the scanner.  The two pages look (visually)
like the left one (254) is approximately straight, but the
right one is skewed.
Presumably the right way to deskew is to separate the two piece
deskew them
thus..

Separating out the left part and right part:
(setf left (con-pict pict1 :width 2150)) ;from x= 0 only up to
(setf right (con-pict pict1 :left 2151))  ; start at middle
(setf left1 (dfilter-out-noise 5 right)) ;for example, to clear
(setf right1 (dfilter-out-noise 5 right))
```

If necessary one can reassemble the connected components, after filtering, into a single bitmap instead of a collection. Here's how: `(setf left1t (manypicts21 left1))`

The left part appears to be skewed at -0.34 degrees. The right part appears to be skewed at 1.27 degrees

## 9 Appendix: Characteristics of pictures

If we have identified a pile of connected components, how do we tell what they are? One way to start is to see what the most common size of an enclosing box is: If we find that (statistical) mode of the width of a box is a reliable statistic, we can use that as an estimate of the width of a character. For hakmem `page35`, the mode of the width is 22, the height is 23. This suggests that connected components less than this width may be incomplete.

For the noisy `6.tif`, the mode is 1 bit wide, though after filtering noise out, it becomes 52 bits wide.

## 10 Comments on re-use of code

In general, software developed in an academic research project is usually not re-usable outside the host institution, if it is even re-usable in the same department. Re-use in the same research group is sometimes a problem! While there are exceptions, they are rare.

Making software re-usable is a challenge, especially when the programs themselves are still under development, or require special environments to run successfully. Experimental programs are allowed to be only partial solutions: that is, the easy or fun parts can be solved; the difficult or laborious parts can be ignored. The programs can be too slow, or might work only on small problems. In our experience here, some neat programs worked only on small bitmaps!

The additional effort necessary to perfect the programs is considerable, and generally not part of the research project. Occasionally software is picked up via some technology transfer to commercial enterprises.

More particularly for OCR, we've looked at material from NIST and CalPolySLO; these did not seem to be reuseable for our purposes, although some components, especially of the NIST code, may ultimately be useful in future development.

We cannot say how others will feel about the programs we've developed: certainly using Lisp appears unusual in the OCR community. (other than our own work we are aware only of G. Kopec's Xerox PARC developments were prototyped in Lisp but recently (1996) rewritten in C.) We believe the choice of Lisp has had some very positive consequences in flexibility, modularity, and interactivity. Interfacing with text or document manipulating artificial intelligence programs, where the use of Lisp is traditional, will presumably be easier than if other cross-language interfaces are needed. Time will tell if the code is efficient enough yet general enough for others to use. We hope others will at our source code, refine and augment the facilities, and provide feedback to the design.

## 11 Match-up strategies: Collecting Characters

### 11.1 Connected Components don't work well enough

One annoying barrier to the collection of characters on a page is that the major heuristic we use is *guaranteed to fail* on perfect scans of some common characters, and may also fail on noisy images of other characters. This major heuristic is that a character has a 1:1 correspondence with a connected component of the "graph" of the image. This heuristic is fine for a connected well-formed version of a character like S. The heuristic clearly fails for i, j, = and punctuation including :";!? Additionally, some typeset characters are disconnected by accident; e.g. it is common to see a separated a leg on an n as a consequence of a thin curve.

Accidental connection are also possible, in which case two or more letters, or fragments of them, are touching. A common join is found in the sequence rn, which is connected to look rather like an m.

### 11.2 How to fix this up?

First, consider a fixed width font such as Courier (Normal) If we draw boundary boxes around the characters' connected components, we see that they are in fact not all the same width. For example, I is narrower than M. What *is* fixed, is the spacing from the origin of one character to its neighbors' origins to the left and right. Thus there is an imaginary rectangle that dictates the character width and a height. This rectangle includes enough space for the tallest character as well as the one with the deepest descender (e.g. p, y, or for that matter, parentheses, brackets and other characters.) In some fonts the characters do not strictly adhere to the rules: some characters actually intrude into neighboring boxes just a little. Italic fonts do this quite routinely. If we reliably knew the (in general, variable) offset from character to character (including spaces), as well as the glyph size, we would be have a solution to a major problem. This would vastly simplifying the next task of identifying each character. If we could recognize that all the text is in a fixed width font, there would be simple strategies to help recognition.

These notes below are much less effective and address variations on our current techniques.

#### 11.2.1 Strategy 1

To join the dot to the i, and the pieces of broken letters together. or more generally, to find a box around each character, each a character width wide, and as high as is necessary.

1. Divide the page up into horizontal "swaths". These generally correspond to lines, but in the case of tightly spaced lines, we may find no breaks, and end up with whole paragraphs. We believe these swaths can be found rapidly by deskewing and looking for large cuts. Our method should not be so sensitive to failure that it matters if we end up with two or more lines in one swath.
2. Next, Determine heuristically, the size of the characters. For a line typeset in all one size, or mostly one size, the median left-edge to left-edge distance of adjacent characters provides the character width. This need not be exact. The height can be determined by the maximum height of characters, after eliminating some percentage of outliers: merged characters from several lines, vertical rules, integral signs. One can also use as a guideline, the character width and height of the previous line(s), as well as some heuristic ratio of known width to height characteristics for fonts.
3. We scan left to right in the regions, picking out
   - connected components that are so completely clearly a letter that they are easily recognized as such. Remove them.
   - scraps that are not recognized. For these, we take their bounding box, with (x,y) origin at lower left. extend a character-width box to the right, and a 50% character-height box up and down. Collect all pieces that are entirely *or partly* within this box. (This allows us to connect together a W whose upper-left leg is the first encountered piece separated from the rest of the body, as well as an A whose lower-left leg is separated.) An alternative is to determine the baseline heuristically, and then use the x-coordinate of the scrap plus the presumed character position.
   - Next, create a minimum bounding box around this new object. Compare this object (picture, bitmap) to those characters with approximately similar heuristically determined character sizes: if this is too large, consider chopping it to the right size. If it passes muster, treat it as a connected component for purposes of removal from the page image, clustering, recognition, etc.

#### Extension to variable-width fonts

In this case we repeat the exercise looking up first the widest possible explanation of the character-width, and then narrowing down so that as many pieces are explained as possible, with the fewest explanations.

This will not work well if narrow character pieces are first happily explained away, e.g. a letter I is recognized and removed; afterward a scrap that looks like H with its left leg missing must be explained, when the real explanation is that it should have been joined to the "I".

This will not work well for characters whose left-most piece and/or right-most piece just missing. A W with a missing left leg, resembling perhaps an italic N, will not be aligned correctly. And if both left and right pieces are missing, for example a W which would resemble an A without a cross-bar, we will also have problems. I think we're going to just miss these characters, at least on a first pass.

#### 11.2.2 Strategy 2

Consider the more simple-minded but computationally expensive approach of Sliding matchups. Potentially very slow, but subject to many heuristics. The general idea would be to try to "explain" as many bits as possible on a page, perhaps by first looking for base-lines then using a relaxation/ best fit method to try to cover the bits with favorite text characters. The plausible locations for characters are limited considerably if we can produce believable implied base-lines for text. The digram frequencies can limit the search substantially, if you are really sure you have identified a letter t, the probability that the next letter is an h is substantial.

further discussion of deskew.. page35 was identified as having a skew of about -0.22 degrees.

If we look for breaks between horizontal swaths in the page 35 as scanned, we find there are 51 swaths: 14 consist solely of noise, 6 consist of double-line-height swaths, and 31 accurate lines.

If we look for breaks in the deskewed version, we find 49 swaths, 10 are 44 pixels high, 15 are 45 pixels high, and all but 6 are between 35 and 47 pixels high. There are three tall lines which have superscripts, and 6 small swaths that are noise, including the image of the staple holes in the corner. All 43 lines are correctly isolated.

Looking for vertical breaks (between characters) is plausible on this page only because it is set in a fixed-width font. In the deskewed version, we can easily separate about 49 character spaces: each "column" is a vertical swath that is between 24 and 27 pixels, plus a few spaces that are marginal noise, and a few spaces that are multiple characters wide: two, three, or four.

On a page set with a variable-width font we would not expect much success in aligning characters in columns anyway, so we would not make much use of this inter-character spacing, in general. (The vertical break detection would still be useful to separate columns in a 2-column page.)

Without deskewing, but treated just as scanned, the vertical breaks between lines are obscured.

## 12 Appendix: Pre-loading a Lisp with LIBTIF

It is handy to have a Common Lisp with the file-accessing library for TIFF (Tagged Image Format) pre-loaded. It is faster to start up and may have additional sharing of code if there are several processes using the software.

We[16] made a new version of Allegro Common Lisp 4.2 linked up with appropriate entry points from the tiff library. The linkage is done by putting together a dummy C language program (we call it `dummy.c` here) that mentions appropriate entry points. One then builds a new Allegro Common Lisp in a standard way using its script called `config`, as defined in its `build` directory, using this dummy program and the tiff library. (Anyone who has installed a copy of this lisp as distributed will have addressed the issue of this `config` file already.) Other Lisps have very similar mechanisms for "foreign function" loading.

Our particular script looks like this:

```
  cc -c dummy.c
  sh config temp=/usr/tmp dummy.o /usr/sww/lib/libtiff.a
```

What should `dummy.c` contain?

In principle one could access any or all of the entry points in the library by mentioning them in `dummy.c` but for conciseness[17] we have eliminated all that are unused in the current project.

```
int dummyRoutinesToForceLoadOfLibTiff()  (* dummy.c *)
{ TIFFClose();
  TIFFSetField();
  TIFFGetField();
  TIFFReadScanline();
  TIFFOpen();
  TIFFWriteScanline();
}
```

[16] Actually, David Glowacki.

[17] We have a perl program courtesy of David Glowacki, that automatically constructs dummy.c from the symbol table of all entries in the tiff library.

## References

[1] Benjamin Berman and Richard Fateman. "Optical Character Recognition for Typeset Mathematics," *Proc. of Int'l Symp. on Symbolic and Algebraic Computation*, (ACM Press) (ISSAC-94) Oxford, UK. July, 1994. 348-353.

[2] Richard O. Duda and Peter E.Hart. *Pattern classification and scene analysis,* Wiley, 1973.

[3] Richard Fateman, Taku Tokuyasu, Benjamin Berman, Nicholas Mitchell. "Optical Character Recognition and Parsing of Typeset Mathematics," *Journal of Visual Communication and Image Representation vol 7* no. 1 (March 1996), 2—15.

[4] James Foley, Andries van Dam, Steven Feiner, John Hughes. *Computer Graphics, Principles and Practice*, 2nd ed. Addison Wesley, 1990.

[5] Katherine Marsden, OCRchie. Sr. Honors Project, May, 1996 Univ. Calif. Berkeley, Computer Science Division, EECS. `http://www.cs.berkeley/~fateman/kathey`

[6] Lawrence O'Gorman and Rangachar Kasturi: *Document Image Analysis*, IEEE Computer Society Press, 1995.

[7] Taku Tokuyasu. "Optical Character Recognition of Typeset Mathematics," MS project, Univ. Calif., Berkeley, 1995.

[8] Wilensky: http://cs.berkeley.edu/elib/