Reading Digital Data Embedded in Iconic Text

Dan S. Bloomberg

Xerox Palo Alto Research Center
Palo Alto, CA 94304

ABSTRACT

Methods for embedding arbitrary digital data within an iconic representation of a document page image are summarized. The result of the encoding is a small iconic image containing the iconic data as small rectangular blocks of pixels, along with a mixture of reduced document image components such as graphics, text and images. As a first step in ensuring data recovery, the encoder verifies that the iconic image can contain the entire message, and that it can be decoded correctly from the noiseless pre-printing image.

To retrieve the message, the data must be separated from the other components in the iconic image and decoded. The decoder is assumed to have no prior information about the location of data within the icon, the encoding channels in which it is encoded, or other meta-data about the message, such as the size or the amount of error-correction encoding. There are three major steps in the decoding process: segmentation, to identify and serialize the datablocks in the icon; measurment of encoding parameters, including determination of the encoding channels; and extraction of the message.

Errors can be introduced into the decoding process at a number of places, and it is necessary to provide mechanisms for detecting and correcting them. For the parameters used here, datablocks from icons generated at reductions of up to 7x are robustly decoded, and error-free message decoding is typically achieved for icons derived from arbitrary pages of scanned documents.

Keywords: embedded digital data, machine-readable data, iconic data, iconic, thumbnail, DataIcon, watermarking, image segmentation, document imaging, page segmentation, data synchronization, data encoding.

1 Introduction

Machine-readable data can be placed directly on a document in a variety of ways, which can be classified by their degree of visibility. One- and two-dimensional barcodes are highly visible on inspection. At the other end of the spectrum, data in watermarks is hidden within elements of the document that would appear virtually the same in its absence. Between these extremes, data can be placed within a visible graphic element in such a way that, on casual inspection, the fact that the graphic includes machine-readable data is not obvious. One example is Xerox DataGlyphs,[4] which have the appearance of a uniformly stippled region, particularly when the marks (typically ellipses oriented at ± 45 deg) are separated by less than 0.020 inches. Another example is iconic data,

Encoding and Formatting	*Data embedding*	*Bytes stored*					
		#1	#2	#3	#4	#5	#6
Block-by-block, sequential	3 bits/datablock	391	190	310	182	90	218
Block-by-block, sequential	2 bits/datablock	429	208	338	199	97	241
Block-by-block, line-synch	3 bits/datablock	346	168	285	156	74	203

Table 1: *Data storage of example icons, using various datablock encoding and formatting methods and parameters.*

where the graphic element containing the data resembles a highly-reduced thumbnail of a page image.[3] This paper concerns methods for extracting such iconic data from scanned images of the graphic.

Iconic text, in the form of "paper icons," was first described by Peairs, who generated a reduced *iconic* representation of a page image by replacing the text characters by their filled bounding rectangles.[6] After rescanning the printed icon at high resolution, the number of characters in each word was counted, yielding a signature that could be mapped into an index in a database of documents. Recently, it was shown that *arbitrary digital data* could be written into the text regions of thumbnails without significantly compromising the appearance, when printed at 300 ppi, as that of a reduced page image.[3] We call this object a *DataIcon.* An important property of the DataIcon is that the iconic data can be extracted using conventional scanning equipment, also at 300 ppi.

This paper first reviews the salient features of the methods for embedding arbitrary data within text regions of thumbnails, and then describes in some detail the methods that can be used to extract the data. Some examples of applications of iconic text data have been given previously.[3]

2 Review of Iconic Data Encoding

Data is encoded in the reduced page image within the regions of text that are roughly between 8 and 12 points at full resolution. Rectangular *datablocks* in the DataIcon, corresponding to typical word sizes at reduced resolution, are used.

2.1 Encoding process

In the following, for simplicity, we encode data only in the length of the datablock symbols; however, data can also be encoded in the length of the spaces between datablocks and in the absolute or relative locations of the top and bottom edges of the datablocks. The result is a *quasi iconic representation,* or *quasi* DataIcon, that resembles a thumbnail of the original image, and we refer to it in the following simply as an *icon.* Fig. 1 shows examples of thumbnails and icons, all at 6x reduction. For each line of small text in the thumbnail, a line of datablocks is generated in the icon, with three bits encoded in each datablock. The spaces between datablocks are adjusted to be equal within each line, and to cause the datablocks to span the same line length as in the thumbnail. The byte capacities of these icons are given in the first row of Table 1. When *two* bits are encoded in each datablock, byte capacities are larger by about 8 percent (depending on the textline width), as shown in the second row. However, with only 4 different lengths, the appearance is judged to be somewhat less "text-like."

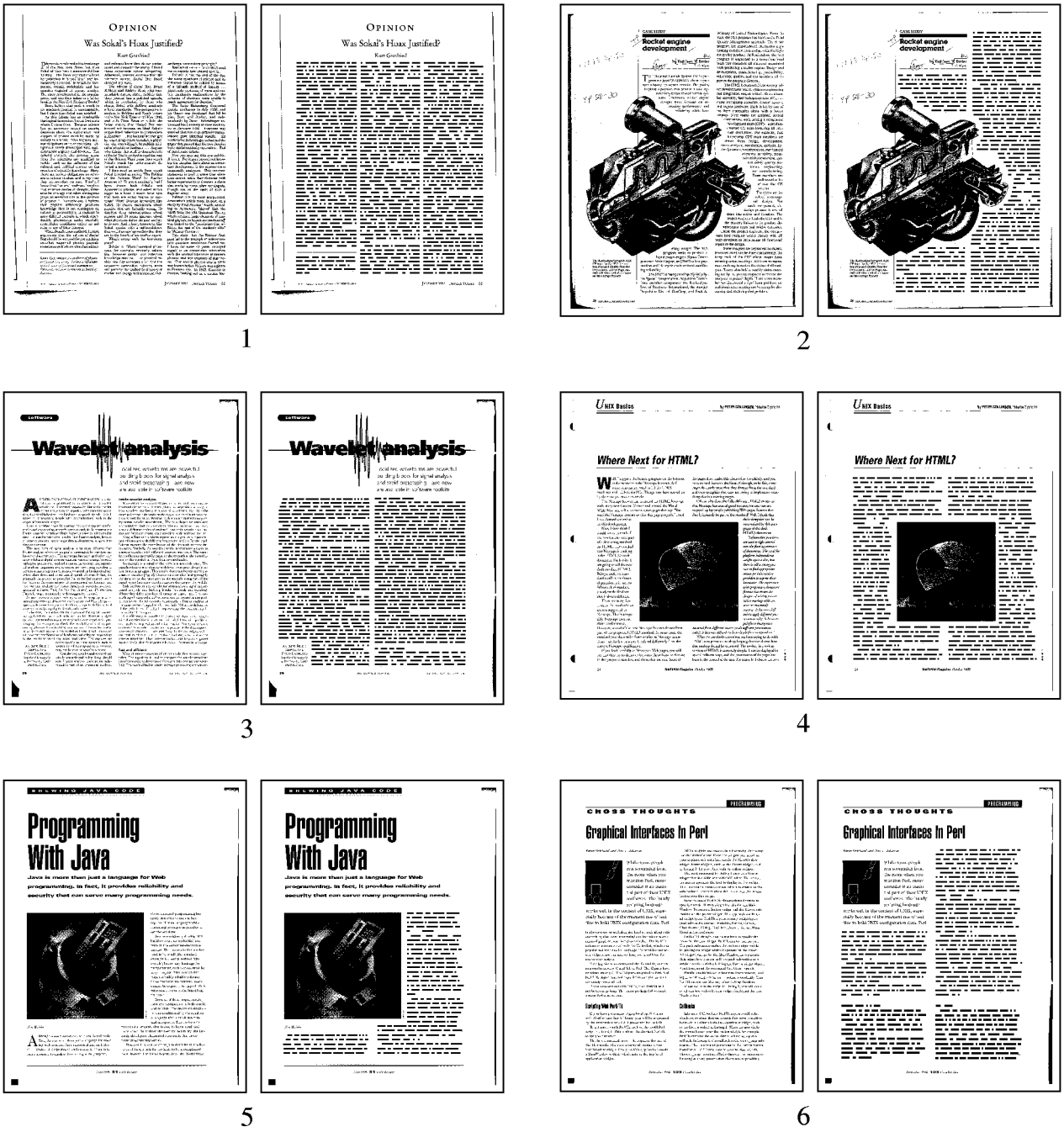

Figure 1: *Six examples of thumbnails and icons, at 6x reduction. The same message is stored in each icon. Three bits of data are stored in each datablock within the icon.*

If a page image does not have sufficient regions of small text to encode the desired message, a *stylistic iconic representation,* or *stylistic* DataIcon, may suffice. These are similar to generic icons used in graphical user interfaces, and represent a *class* of documents rather than the look of a specific one. Stylistic representations are much easier to encode and decode than quasi representations, because the location of the regions containing datablocks is known in advance. They can be used as a default in situations where it is not practical to generate a quasi DataIcon. In the following, they will not be considered.

For quasi DataIcons, the datablocks are placed in lines called *blocklines,* that visually simulate "textlines" in the thumbnail. The blocklines are further grouped into *iconblocks,* which correspond to textblocks. Encoding the message data within an icon proceeds in the following steps:

- *Deskew.* The full resolution image is deskewed.
- *Segmentation.* Regions of median-sized text are located and ordered.
- *Thumbnail.* A reduced image is made at the chosen reduction factor, and the regions where iconic text will be generated are masked out.
- *Message preparation.* Error-correction coding (ECC) bytes are appended to the input message, and a header, consisting of at least a start code, the input message size, and the number of appended parity bytes, is prepended. The appearance of the encoded datablocks can be randomized by XOR encrypting the message. The result is then replicated as many times as necessary to fill the space available in the icon.
- *Datablock generation.* The prepared message is encoded as datablocks in the icon, according to the chosen method.

Segmentation is a subset of the layout analysis used for document image summarization,[2] with different algorithms for textblock sieving and ordering. A linear (or systematic) Reed-Solomon block ECC[7] is used because errors are expected to occur in bursts, due to temporary loss of synchronization during decoding, and with such a code the parity bytes are simply appended to the message bytes. To convert from the message bytes to the sequence of datablocks, two encoding methods have been described. In *block-by-block* encoding, a fixed amount of data is placed in each datablock; the amount of data encoded in a given line length within the icon depends on the data. In *run-length-limited* encoding, analogous to RLL encoding used in storage systems,[5] constraints are placed on the minimum and maximum datablock lengths, and the code rate is nearly constant. (The code rate is weakly data dependent because of the space that must be inserted between datablocks.)

Other details of the encoding process, except for synchronization detection and recovery, are given elsewhere.[3] Once the icon is generated, it should be tested for readability, using the decoder. Because the datablocks in the generated icon are noiseless, successful decoding does not guarantee that the message can be extracted from a printed and rescanned icon. However, for the encoding parameters used in the icons shown here, given in Table 2, we have found that if the noiseless icon can be successfully decoded, then an icon that has been printed and rescanned at 300 ppi is nearly always readable. The reason is that iconic segmentation of noiseless and scanned icons is usually identical.

2.2 Maintaining synchronization

In the decoding process, *synchronization* is the correct numbering of the bytes in the message. Even a small segmentation error can cause loss of synch, which can quickly exhaust the error correcting capability.

Print resolution	300 ppi
Scan resolution	300 ppi
Icon reduction	6x
Encoding method	block-by-block
Datablock channel embedding	datablock width: 3 bits/datablock
Minimum datablock length	3 pixels
Minimum inter-datablock space	3 pixels
Datablock length quantization increment	3 pixels
Datablock formatting method	sequential
Blockline constraints	none
Data constraints	ascii encoding
Parity bytes	4

Table 2: *Parameters used for encoding icon examples.*

Consequently, the encoding process should include redundancy that allows loss of synch to be identified and synch to be quickly regained. We describe two methods that are distinguished by the low-level formatting of the message into the datablocks. The encoder can set bits in the message header to indicate which of these methods has been used for the icon.

Sequential ascii formatting is a simple method for detecting loss of synch, that has relatively little added redundancy. The binary message is converted to an ascii representation, where the first bit of each byte is a 0. During decoding, the byte framing can be adjusted to minimize the number of insertions or deletions required to keep these bits 0, along with the constraint on the known total message size. A disadvantage of this method is that if there are multiple insertions and/or deletions, it may be difficult to assign correct byte numbering throughout the message. However, if the noiseless icon is found to be decodable, this method should be satisfactory.

Line-synch formatting typically has slightly greater redundancy (overhead), but it is more robust for limiting errors due to loss of synch, and should be used if sequential ascii formatting fails to make a readable icon. In this method, the datablock encoding is constrained to have an integral number of complete bytes on each blockline. Consequently, there is no reason to use ascii data. The extra bits in the blockline, which vary from 0 to 7, can be used to indicate the *byte index* within the message of the first byte in the blockline. For example, if there are $n < 8$ extra bits in the blockline, they are encoded as the message byte index modulo 2^n. (The extra bits can also indicate the blockline index in the message, again modulo 2^n.) In the encoding process, the length in pixels of each blockline is first specified. Data is filled into the blockline, and the last datablock that completes a message byte may contain extra bit(s) that constitute part of the index. It must be verified that this datablock with the correct index bits fits into the blockline. If it does not, one less message byte must be written into the blockline. Also, for the case where the blockline is too short to contain a single message byte, the goal is to encode the index of the first message byte of the previous blockline. If the blockline is not to remain blank, this requires, at a minimum, a single datablock. If the pre-allocated space is too small to hold the correct datablock, the space must be increased. Likewise, if the space is too small to hold the first two datablocks, it can only hold one. Thus, for very short blocklines, the pre-allocated widths must be taken as hints, and the actual blocklines may end up either a little wider or narrower. The third row of Table 1 gives byte capacities of the six icons when using line-synch formatting. The capacity is comparable to that of sequential formatting (first row) if one includes the data expansion that occurs when generating ascii from binary.

3 Iconic Data Decoding

There is a performance asymmetry between the encoding and decoding segmenters. The encoding segmenter has more variable input data, but if it misses some text regions that would have been fair candidates for embedding data, the system doesn't fail—it just becomes less efficient in data storage. On the other hand, the decoding segmenter is searching for the icon datablocks. These are highly regular image components and relatively easy to locate, but it must reliably find most of them.

There are three major steps in the decoding process. The first is to identify the datablocks and their order in the image, and we use two distinct segmentation steps. The second step is to measure the datablocks and spaces to determine how data is encoded and to find the coding parameters. Part of this step is the assignment of coded bits to the datablocks and spaces. The third step is to extract the message, including meta-data that is necessary for decoding.

3.1 Decoding segmentation

As with encoding, the first step in decoding is to deskew the icon. This improves both segmentation and measurement of the datablocks, and is carried out by the standard method of maximizing the variance of differential line pixel count sums,[1] at the full scan resolution of the icon.

Next, the icon is scaled to a standard size, which we take to be exactly 6x reduction at 300 ppi scan resolution. This simplifies segmentation, because the analysis is performed at constant resolution, 50 ppi, regardless of the source of the iconic image. Segmentation then proceeds in two steps: coarse and fine.

Coarse segmentation is a combination of sieving and grouping. Components that are not iconic datablocks are progressively removed (sieved), but at the same time logical groupings of datablocks, blocklines and iconblocks consisting of several blocklines, are identified. Sieving and grouping are done together because they are complementary, each aiding the other. Component removal proceeds conservatively, leaving the decision for removal of ambiguous structures to fine segmentation. In fine segmentation, the microstructure of the pixels composing each identified blockline is examined. Determination is made on a line-by-line basis: if the components of the line are not sufficiently solid, the line is removed. The result of segmentation is to group and order the datablocks into iconblocks, each of which consists of several blocklines.

We show two examples of the segmentation process, taken from icon examples 3 and 4 in Fig. 1. These icons were scanned at 300 ppi, and the resulting noisy icons are shown magnified about 2x in Fig. 2. Segmentation results on examples 3 and 4 are given in Fig. 3 and Fig. 4, respectively. Frame references in the following are to these figures. Frame 1 shows the scanned icon at actual size (6x reduction).

3.1.1 Coarse segmentation

Coarse segmentation takes place in three phases. In Phase 1, image and large graphic parts are removed, leaving iconic text (datablocks) and other components such as reduced text and rules. In Phase 2, the iconblocks are identified by careful merging and some of the noise components are discarded. In Phase 3, operations are done separately on each iconblock, in order to avoid merging them. Small holes are filled, noise is removed, and the squared-up iconblocks are placed in the correct order for reading the datablocks. Binary morphological

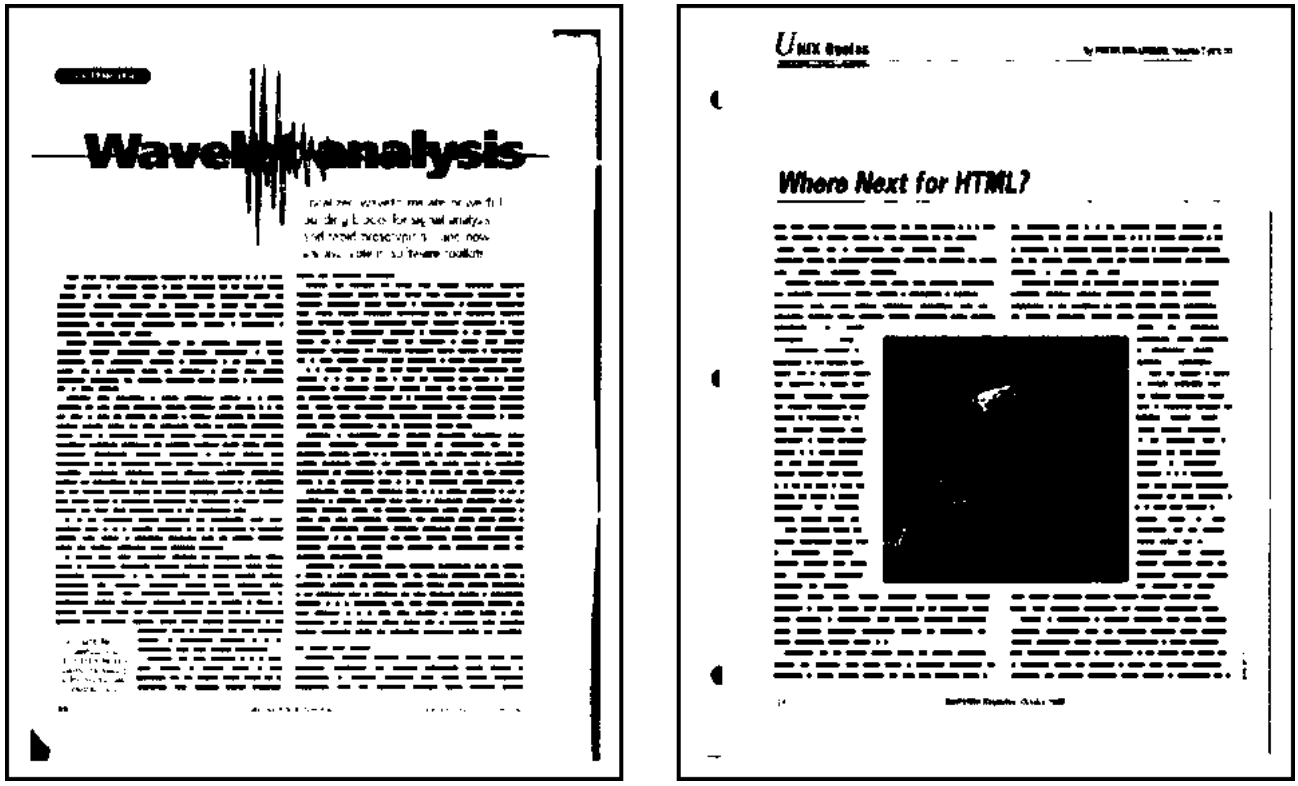

Figure 2: *Scanned icons: these are examples 3 and 4, scanned in from Fig. 1*

operations are used extensively for coarse segmentation. Brick structuring elements are denoted by "width x height"; e.g., open(3x1) is a horizontal opening of width 3.

A vertical whitespace mask is used to maintain separation of laterally-adjacent iconblocks. But a horizontal whitespace mask, which would tend to split iconblocks, cannot be used to avoid merging vertically-adjacent iconblocks. Thus, large vertical closings or dilations must be avoided before the third phase, where iconblocks are handled separately. Generation of the vertical whitespace mask (frame 3) is part of Phase 2, but it is done first for convenience, using the sequence: close(3x1), bit inversion, and open(1x50). The first small closing reduces the ability of the mask to break text blocks, and the large vertical opening extracts the critical inter-column separators. Small additional close/openings can be used to remove foreground and background noise in the mask.

Next is Phase 1, where the image and large graphic parts are removed from the icon (frame 2), by first solidifying frame 1 using a horizontal close/open(3x1), and then removing any components with height of 10 pixels or greater.

Phase 2 continues by conservatively weaking the remaining non-datablock parts, using a sequence of two small openings: open(2x1), open(1x2). This is followed by a close(15x1) to solidify the blocklines, and removal of narrow components (noise) using open(5x1). The horizontal closing was likely to join adjacent iconblocks, so the whitespace mask is subtracted to restore their separation as iconblocks representing adjacent columns of text (frame 4). Components that are too tall (> 8 pixels), too short (< 2 pixels), or too narrow (< 3 pixels) are removed (frame 5). The blocklines are next connected vertically into iconblocks, using close(1x9), and

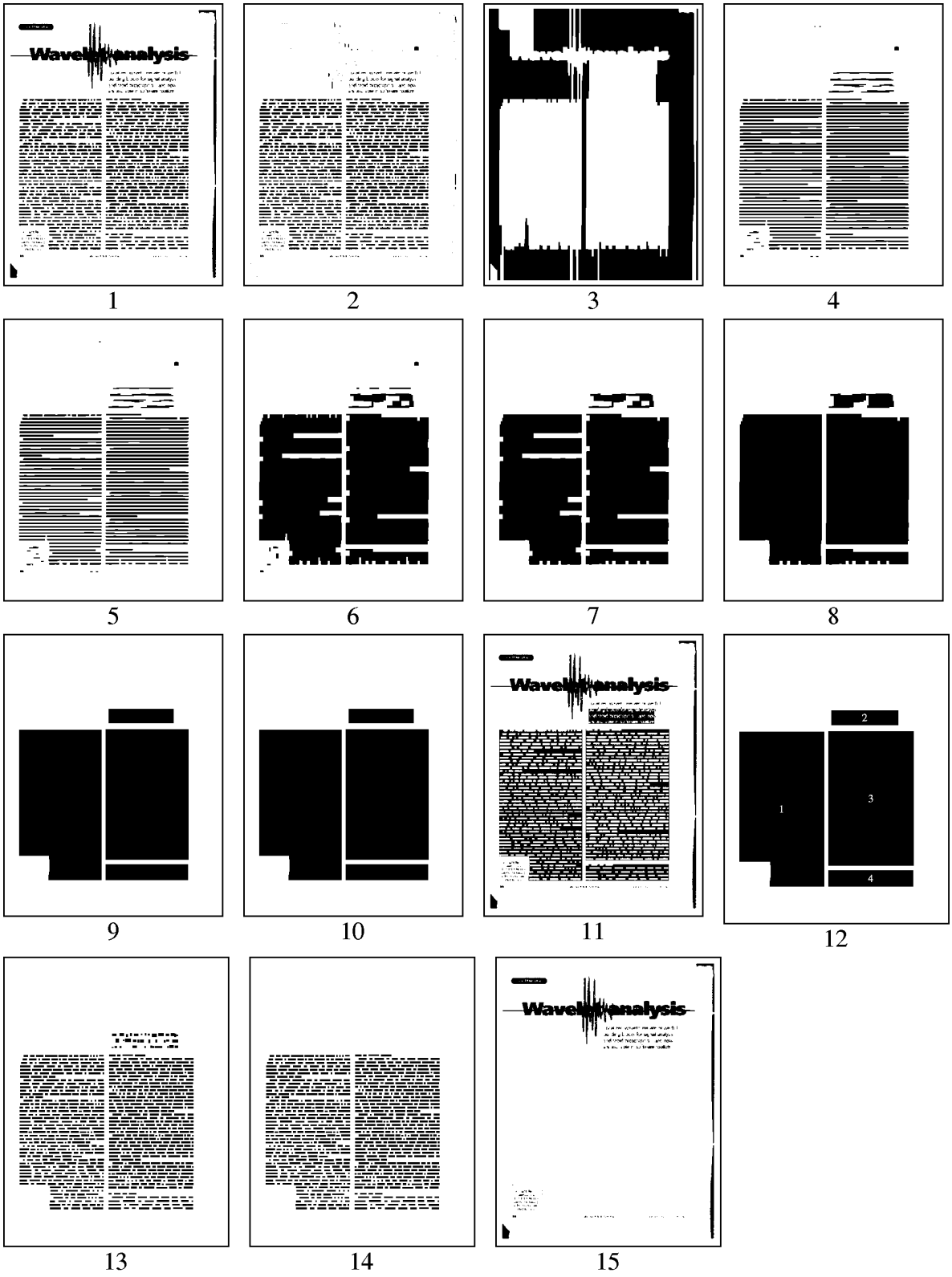

Figure 3: *Segmentation operations for example 3, starting with the scanned image on the left in Fig. 2.*

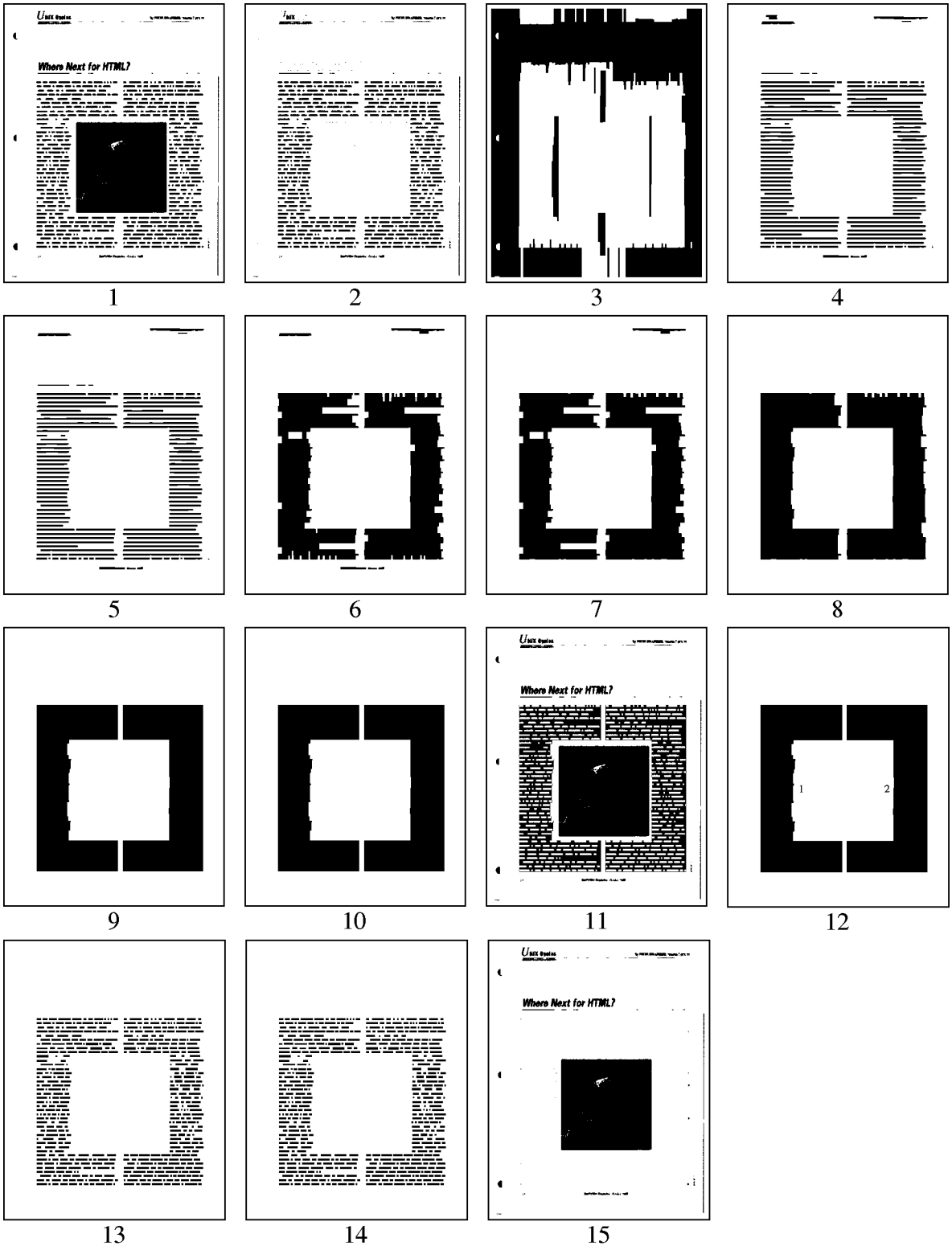

Figure 4: *Segmentation operations for example 4, starting with the scanned image on the right in Fig. 2.*

any remaining thin lines are removed (frame 6) using open(3x3). The connected components are identified, completing Phase 2.

In Phase 3, all operations are done separately on iconblock components, avoiding further interactions between them. Those components that are very short (< 11 pixels) are removed, and small vertical gaps in the remaining ones are filled (frame 7) using close(5x1). Iconblock components that are too narrow (< 60 pixels) are removed, and the remaining ones are solidifed (frame8) using close(1x20).

Frame 9 shows the iconblock masks after they have been squared up, and after any smaller ones, that are essentially within the *b.b.* (bounding boxes) of larger ones, have been assimilated. The squaring up process identifies the components that constitute the *difference* between the iconblock mask and its bounding box. If either of two tests, using a pre-determined length $L = 50$, succeeds on a component, it is "added" to the iconblock, thus helping to square it up. The first test requires the b.b. area of the component to exceed L^2 and the component to be relatively square. The second test allows the component to be smaller, with area $L^2/4$, but requires that at least 75 percent of the pixels in the b.b. be ON. The highest iconblock in frame 9 of Fig 3 is not derived from iconic data, and will be removed later. An assimilation step is useful to repair iconblocks that may have been accidentally broken in the early stages of segmentation.

Frame 10 shows the iconblocks remaining after those containing less than 3 blocklines have been removed. The result of coarse segmentation is visualized in Frame 11, which shows slightly dilated (2x2) iconblock masks that have been XOR'd with the scanned icon (frame 1).

Finally, the iconblocks are ordered, in strict column-major order, using the same ordering algorithm that was used on the original image for encoding icons. Iconblock precedence is determined by the (x,y) location of the upper-left corner. Iconblocks whose x values are within 8 pixels are sorted based on the y value (e.g., from top to bottom); otherwise by x. The distance of 8 pixels is chosen to group iconblocks from typical icon images transitively into columns. The ordered iconblocks are shown in frame 12.

3.1.2 Fine segmentation

Fine segmentation continues on each iconblock in isolation. Within an iconblock, components that are far too small to be datablocks (e.g., with width < 3 pixels or height < 2 pixels) are removed, and the remaining components are sorted two-dimensionally in row-major order (i.e., into blocklines). Then each line of datablocks is analyzed to determine if it is iconic data. Two criteria are used. First, to remove horizontal rules that have fallen through the coarse sieve, none of the datablocks can exceed a maximum width. Second, to eliminate text regions that were not datablocks in the icon, the pixels within each datablock should, in the aggregate for the line, occupy at least 70 percent of the total datablock b.b. area.

Frame 13 shows the individual datablocks, expanded to their circumscribing rectangles, that passed through the coarse segmentation, and frame 15 shows the remaining image components in the icon. Finally, frame 14 shows those datablocks that qualified as actual data according to the fine segmentation sieving criteria given above.

3.2 Datablock analysis for encoding parameters

The datablocks in each iconblock are analyzed to determine the quantization levels that obtain for datablocks within this iconblock. The first step is to estimate the size of each datablock, and the channel in use here is the width of the datablock. When the datablocks are encoded with length quantization increment less than 5, the b.b. width does not provide sufficient accuracy. Instead, the bitmap of each datablock must be examined. To determine the width of a datablock, the top and bottom rows are discarded, and average location of left and right edges of the remaining rows is found. In computing this average, outliers are discarded. For the left edge, an outlier is a row that starts 2 or more pixels to the right of the leftmost row, and similarly for the right edge. (Likewise, to determine the height, the first and last columns are discarded, and the remaining columns are analyzed to get an average position for the top and bottom of the datablock.)

The datablock widths are converted to integers in units of 0.1 pixel, and a histogram of the widths is constructed, using overlapping buckets without normalization. To find reliable maxima, the histogram is then smoothed with a window width of about 1.0 pixel, considerably smaller than the expected distance between peaks. Individual datablocks are assigned to the set represented by the nearest maximum.

A typical datablock width histogram for an iconblock is shown in Fig. 5. This raw histogram shows that each of the eight datablock sets is composed of a subset set of approximately three peaks representing widths that are separated by 5 units, corresponding to exactly half a pixel. The largest peak in each of the subsets occurs for widths that are exactly factors of 3 pixels apart: 3, 6, 9, ... 24), which are the widths of the datablocks in the ideal icon. The satellite peaks mostly represent larger widths, by up to one pixel, indicating that the process of printing and rescanning tends to increase the datablock widths. The eight subsets are well-separated from each other, so the raw error rate due to placing a datablock in the wrong set should be very low. This is because we used a quantization increment of 3 pixels for encoding datablocks. It is clear that a quantization increment of 2 pixels is too small to be used robustly for binary scanned icons.

The representative width for each of the eight sets is found by making two convolutions on the data. In the first, a histogram of overlapping bins of width 9 units, is constructed from the raw data. The width is chosen to be large enough to contain two of the subpeaks, but not three. This histogram is not normalized, to avoid roundoff error. It is then convolved with a smoothing window, also of width 9 units. The overlapping bin histogram and the smoothed version are shown in Fig. 6.

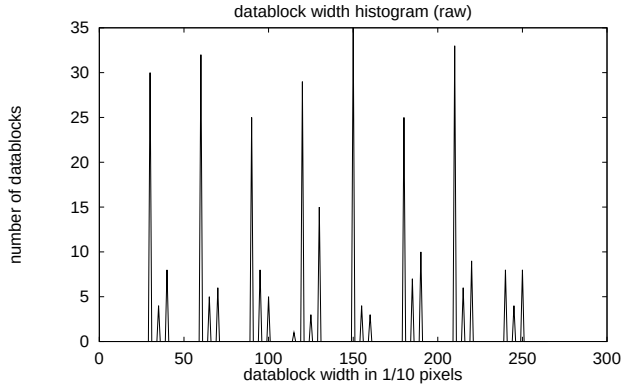

Figure 5: *Histogram of datablock widths for a typical iconblock. Each peak represents a subset of datablocks that were measured to have widths at half-integral pixels (five 0.1 pixel units).*

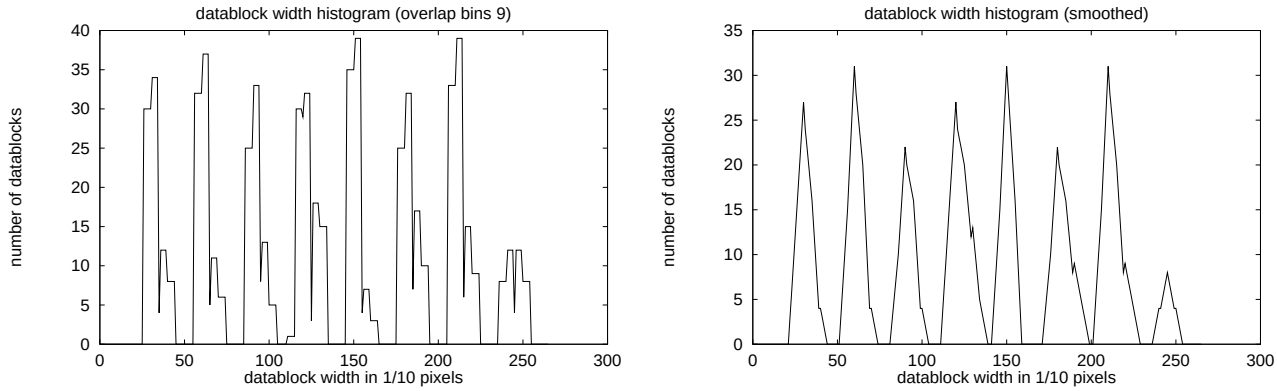

Figure 6: *Unnormalized datablock width histograms. The one on the left is derived from the raw data, using overlapped bins of width 9 units. This is then smoothed using a window of width 9 units to give the histogram on the right, from which the maxima for each set are located.*

3.3 Message extraction

For each iconblock in the ordered sequence, the datablocks are sequentially examined, blockline by blockline, and assigned 3-bit message values. This raw message typically has synch errors due to occasional insertions or deletions of datablocks, and, as we have seen, only rarely to a mis-assignment of the message value for a datablock.

As described in Section 2, the message must be redundantly encoded to identify and correct synch errors. Redundancy is applied in three ways:

1. Within the message, using ECC and optionally ascii encoding.
2. Using message repetitions, with a unique 3-byte start sequence to identify the beginning.
3. Optionally, synching the bytes to the blocklines, and encoding byte indices.

After each 3-byte start sequence, the message size and number of ECC parity bytes is encoded. Appending n bytes of ECC allows correction of *any* $n/2$ bytes, or up to n *erasure* bytes that can be selected near locations in the message where bit synchronization or data is believed to be lost. The two approaches to identifying and correcting synch errors were briefly described in Section 2.2.

The actual message is derived from the decoded message by XOR decryption, followed by conversion, if necessary, from the encoded ascii back to an original binary message. Note that if ascii encoding is used, the XOR key must also be composed of a sequence of ascii bytes.

4 In the Limit: Small DataIcons

The examples given so far have all been of icons at 6x reduction. This is a convenient size, both for allowing a person to read the large non-iconic text and recognize the document, and for robust machine decoding of the iconic data. The question naturally arises: how much smaller can machine-readable quasi icons be made, still assuming print and scan resolutions of 300 ppi? The short answer is: as small as anyone would want. Consider two types of quasi icons:

1. *Line faithful.* All examples to this point are of this type. Each line of text in the thumbnail is geometrically registered with a blockline of iconic text.

2. *Line free.* The textblocks in the thumbnail are loosely mapped to iconblocks in the icon. In particular, encoding datablock parameters are chosen within the iconblocks for decodability, and blocklines are freely constructed without reference to the textlines in the thumbnail.

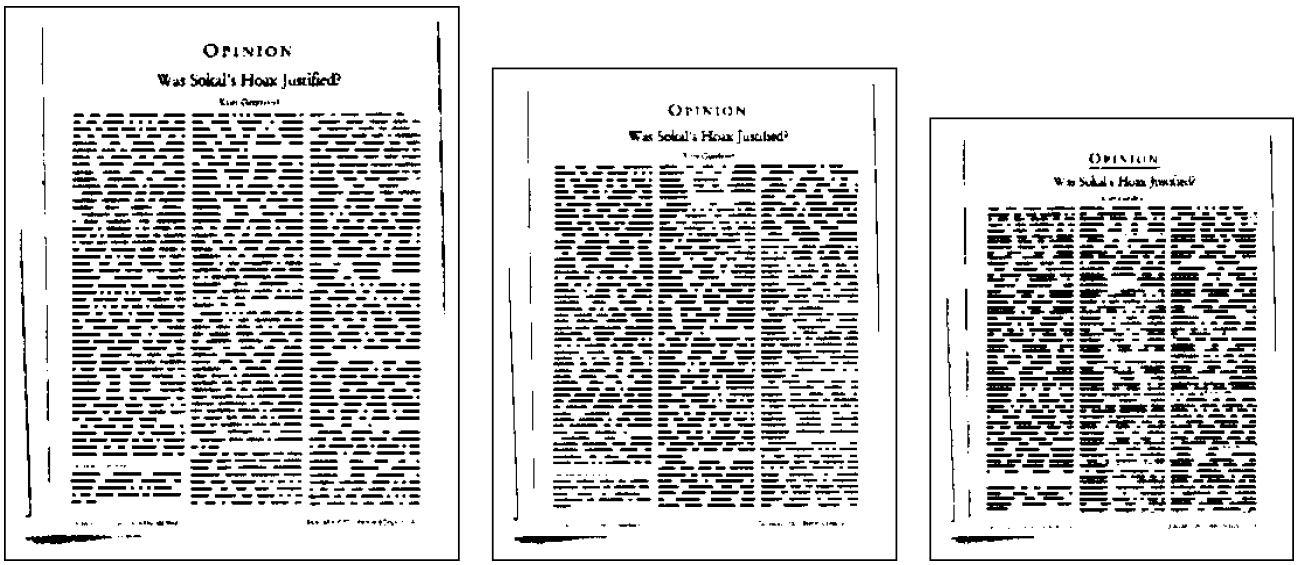

Figure 7: *Scanned icons from example 1 of Fig. 1. These were generated at reductions of 7x, 8x and 9x, and are shown magnified approximately 2x.*

Fig. 7 shows line faithful 7x, 8x and 9x icons after rescanning, at about 2x magnification. They are derived from the document of example 1 of Fig. 1. The amount of raw data stored in these three icons is 325, 281 and 243 bytes, respectively. The stored data decreases approximately *linearly* with the reduction factor, because the width quantization increments and minimum datablock width are at their minimim allowed size of 3 pixels. The 7x and 8x icons are readily decodable, but the 9x icon is difficult because a large number of the datablocks have merged vertically. Decoding this icon requires different methods for segmentation, parameter estimation and datablock decoding. In particular, no assumption can be made that the datablocks are separate connected components.

Fortunately, it is not necessary to solve this hard problem. Instead, icons should be designed so that no datablock merging occurs after scanning. A reasonable criterion is that 3 pixels of white space are typically maintained between blocklines (with some noise pixels allowed in the gap). This is obtained for the 6x icons, where the blockline center-to-center spacing is typically between 7 and 8 pixels. The xerographic response of the

printer needs to be considered, because write-black printers thicken lines whereas write-white printers are either neutral or act to thin lines. The 8x icon in Fig. 7 has a blockline center-to-center spacing of between 5 and 6 pixels, which happens to work for this icon but is too small for safety. For an icon reduction of 8x or greater, line free quasi icons can be used, where the datablock height and inter-blockline spacing are constants that are chosen according to a conservative criterion such as that given above. Then the amount of stored data decreases approximately as area of the iconblocks; i.e., as the *square* of the reduction factor.

5 Summary

Methods for embedding arbitrary digital data within a DataIcon, an iconic representation of a document page image, have been briefly reviewed. To ensure data recovery, the encoder must first verify that the encoded iconic image contains the entire message and that it can be decoded correctly from its ideal pre-printing image. If, additionally, the datablocks remain separated after scanning, the data should be readable by the decoder.

To retrieve the message from a scanned icon, the data must be separated from the other components in the iconic image and decoded. The decoder is assumed to have no *a priori* information about the datablock encoding parameters or the layout of the datablocks within the icon. The retrieval process has three major steps. The first is segmentation, to identify and order the datablocks in the icon. The second is to measure the retrieved datablocks and spaces, in order to determine in which channels the data is encoded, and to find the encoding parameters. The third step is to extract the message, using error correction and meta-information, such as message size and synchronization indicators, as needed.

Errors can be introduced into the decoding process at a number of places, and it is necessary to provide mechanisms for detecting and correcting them. During segmentation, regions that are not data blocks are conservatively removed, iconblocks are generated by carefully merging datablocks and are consolidated independently, and a final verification is performed by analyzing the shape of the datablocks in each blockline remaining in the icon. Uniform encoding of datablocks and spaces allows extraction of robust parameters. The serial ordering of extracted iconblocks must be robust to small perturbations in their location, because exact spatial alignment of such regions between the original image and the scanned (and possibly copied) image is not possible. A small number of byte errors can be removed by adding ECC coding to the message. Byte alignment within the message must be maintained throughout, and two formatting methods that require redundant encoding (byte alignment on blocklines and ascii encoding) have been described. To compensate for uncorrectable loss of synchronization, multiple encodings with unique start-of-message markers are used. Data can be reliably retrieved from icons of any desired size if the datablock encoding parameters are chosen to be compatible with both the printer characteristics and the scan resolution.

6 REFERENCES

[1] D. S. Bloomberg, G. E. Kopec and L. Dasari, "Measuring document image skew and orientation," *SPIE Conf. 2422, Document Recognition II*, pp. 302-316, San Jose, CA, Feb 6-7, 1995.

[2] D. S. Bloomberg and F. R. Chen, "Extraction of text-related features for condensing image documents," *SPIE Conf. 2660, Document Recognition III*, pp. 72-88, San Jose, CA, Jan 29-30, 1996.

[3] D. S. Bloomberg, "Embedding digital data on paper in iconic text," *SPIE Conf. 3027, Document Recognition*

IV, pp. 67-81, San Jose, CA, Feb. 12-13, 1997.

[4] D. S. Bloomberg, "Binary image processing for decoding self-clocking glyph shaped codes," U.S. Patent 5,168,147, Dec. 1, 1992.

[5] P. A. Franaszek, "Sequence-state methods for run-length-limited coding," *IBM J. Res. Dev.* **14**, pp. 376-383, July 1970.

[6] M. Peairs, "Iconic paper," *ICDAR '95*, pp. 1174-1179, Montreal, Quebec, Aug. 14-16, 1995.

[7] W. W. Peterson and E. J. Weldon, *Error Correcting Codes*, 2nd ed, Ch. 5, MIT Press, Cambridge MA, 1972.