Textual Image Compression[1]

(Extended Abstract)

Ian H. Witten[2]
Timothy C. Bell[3]
Mary Ellen Harrison[2]
Mark L. James[2]
Alistair Moffat[4]

We describe a method for lossless compression of images that contain predominantly typed or typeset text—we call these *textual images*. These are commonly found in facsimile documents, where a typed page is scanned and transmitted as an image. Another increasingly popular application is document archiving, where documents are scanned by a computer and stored electronically for later retrieval. Our project was motivated by such an application: Trinity College in Dublin, Ireland, are archiving their 1872 printed library catalogues onto disk, and in order to preserve the exact form of the original document, pages are being stored as scanned images rather than being converted to text. Our test images are taken from this catalogue (one is shown in Figure 1). These beautifully typeset documents have a rather old-fashioned look, and contain a wide variety of symbols from several different typefaces—the five test images we used contain text in English, Flemish, Latin and Greek, and include italics and small capitals as well as roman letters. The catalogue also contains Hebrew, Syriac, and Russian text.

The best lossless compression methods for both text and images base their coding on 'contexts'—a symbol is coded with regard to adjacent ones. However, the contexts used for coding text usually extend over significantly more characters than those used in images. In text compression, the best methods make predictions based on up to three or four characters [4, 7], while with black-white images, the most effective contexts tend to have a radius of just a few pixels [6, 8].

One possibility for textual image compression is to perform optical character recognition (OCR) on the text, and only transmit (or store) the ASCII (or equivalent) codes for the characters, along with some information about their position on the page. There are several problems with this. Considerable computing power is required to recognize characters reliably, and even then it is not completely reliable, particularly if unusual fonts, foreign languages or mathematical expressions are being scanned.[5] OCR systems can require "training" to learn a new font, and an operator may have to adjust parameters such as the contrast of the scan to ensure that errors are corrected and small marks are removed from the page. Ironically, although the image may look better, it is actually *inferior*, because it does not faithfully represent the original image. Smudged or badly printed characters are replaced with what the OCR system has interpreted them as, rather than leaving human readers to make their own interpretation. Dirt or ink stains, which may have given valuable clues to a researcher, are lost. Even the typeface may not be reproduced accurately, affecting the look of the document. For typed business letters, this sort of 'noise' may be acceptable, even desirable, but for archives, where the interests of future readers are unknown, there is a strong motivation to record the document as faithfully as possible.

The compression methods investigated here are lossless, so the original document can be reproduced exactly from its compressed form. This is done by attempting to separate the text and noise in the document. The two components are then compressed independently using a method appropriate for each

[1]Please direct all correspondence to [illegible]
[2]Department of Computer Science, University of Calgary, Calgary [illegible], Canada. Telephone +1 403 220 6780
[3]Department of Computer Science, University of Canterbury, Christchurch [illegible], New Zealand. Telephone [illegible]
[4]Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, Australia
[5]One of the authors has spent considerable time editing a [illegible]; the letters 'r' and 'n' were run together and read as an 'm'. The [illegible] was particularly [illegible] [illegible] occurred in the text.

Method of compression

Before proceeding with a detailed description, we briefly summarize the new compression method images with reference to a particular example. Figure 1 shows a test image. The actual library pages are formatted in two columns with a vertical separating line and a full-width header at the top. However, these structural elements are detected in a pre-processing stage and our system is presented with bitmap files representing single columns of text.

First, the image is segmented into individual groups of pixels, called 'symbols.' As each is identified it is checked against a growing library of symbols that have been seen so far. This involves testing the newly-segmented item against every member of the library. If no sufficiently close matches are found, the new symbol is added to the library. The library of symbols that is created from the test image is shown in Figure 3. Symbols occur in order of their appearance in the image.

The textual part of the image is the sequence of symbol numbers that have been encountered in it. Symbols are quite similar to characters; however, characters with disconnected parts like 'i' and 'j' are represented by pairs of library elements, one for the body and the other for the dot on top. The second column of Figure 2 shows the sequence of symbol numbers, and using the library in Figure 3 the symbols can be identified and read off the original picture. The first column gives an approximation to the corresponding library symbols, for easy reference.

It is also necessary to code the (x, y) coordinate offsets between one symbol and the next. These inter-symbol gaps are recorded in the last two columns of Figure 2, which show the x- and y-offsets from the right-hand bottom corner of one symbol to the left-hand bottom corner of the next. Occasionally the x-offset will be small and negative (for example, following the body of a 'i' when returning to the dot on top), or large and negative (when returning from the end of one line to the beginning of the next).

Using these symbol numbers, most of the original image can be reconstructed from the library. The result is an approximation to the original image that we call the 'reconstructed text'; it is shown in Figure 4. It differs from the original image in three ways. First, small groups of pixels are rejected by the segmentation process; these correspond to specks in the image and do not appear in the reconstructed text. Second, because matching is approximate, a halo of pixels often appears around the edge of symbols caused by a mismatch between the library element and the actual symbol in the image. Third, symbols that only occur once are not entered into the library and so do not appear in the reconstructed text.

To complete the original picture, it is exclusive-OR'd with the reconstructed text image to form a bitmap called the 'residue,' shown in Figure 5. This is coded using image compression techniques. As can be seen, much of the text can be made out from the residue, which indicates that the text itself should be of considerable help in compressing it. In fact, the reconstructed text is used as part of the context used to code the residue.

Details of the individual steps

Symbol extraction

The first stage in symbol extraction is to identify lines of text so that symbols can be detected in their natural order. This is done in an ad hoc manner—the problem has already been addressed by numerous optical character-recognition programs (though we were unable to locate detailed descriptions of the algorithms they use). The rectangle of pixels that the program identifies as a line of text is called a 'window.' To find the top of the first window we scan down the image seeking a connected group of pixels, in any horizontal position, whose height exceeds some prespecified minimum (currently 15 pixels). The bottom of the window is essentially assumed to be the first point at which a clear white line extends right across the image. However, this may be marred by a slight overlap (say, between subscripts on one line and superscripts on the next) or by a speck of dust. To cope with this, the scan continues until a full-width strip of a certain minimum height (again, 15 pixels) is encountered for which there is no connected black line that extends from top to bottom. This means that there must be a connected white

line extending through this strip right across the image, and this is taken as the bottom of the window.

Next, the window is scanned in transposed raster order from bottom to top, moving one pixel rightward after each scanline. This visits the symbols in left to right order, and in general finds a symbol before any disconnected part that lies above it. For example, the body of an 'i' will be encountered before its dot; the accent on an 'é' will come after the letter itself. As soon as a symbol is found, it is removed from the window. Thus if 'ü' is encountered, the 'u' is found first and removed, then the first dot, and finally the second one. To cope with cases where a high superscript in the next line intrudes into the window, any connected region that extends below the window is left in the image and dealt with as part of the next window.

Symbols are segmented by boundary tracing (we use 8-connectivity), and the result is extracted from the window as a candidate for inclusion in the library. Symbols comprising less than 15 connected pixels are ignored. Other non-typographical symbols—specks of dust, annotations, coffee-stains, etc—are segmented as usual but will be removed from the library unless they occur again.

This method of symbol extraction is not robust and can easily be fooled. We recognize that a full system would have to address this issue but have not yet done so ourselves because, for us, these are not the most interesting research questions.

Template matching

As symbols are extracted, they are matched against those already in the library. With each library member a set is kept of all symbols that match it. If the current symbol matches an existing one, it is added to the set of matches for that library symbol; if not, it is entered into the library as a new symbol.

The template matching procedure is critical to the successful identification of symbols. A number of matching methods have been described in the literature. Holt [4] divides them into two categories depending on whether they use global or local criteria. The former measure the overall mismatch between the new symbol and a library template, while the latter seek local mismatches that comprise just a few pixels. Both work on an *error map* which is the bitwise exclusive-OR between the new symbol and a library member, and for this the two must be registered appropriately. On each symbol in the library, the new one is superimposed by aligning the lower left corners. Nine different registrations are used, corresponding to one-pixel displacements in the eight principal compass directions. In each registration the exclusive-OR between old and new symbols is calculated to yield the error map, and the tests below are repeated nine times. If the template is accepted in any of the registrations, the one with the best fit (i.e. minimum Hamming distance) is chosen.

Template-matching methods that use global criteria are intended to be size-independent. However, they must be trained on the fonts being used—if not, the results are unreliable. For example, the 'combined symbol matching' method [9] computes a weighted sum of pixels in the error map, where error pixels are weighted more highly if they occur in clusters. A match is rejected if this exceeds a threshold obtained from a training process, which varies depending on the size of the symbol. Holt and Xydeas [3] describe another global method that achieves slightly better performance.

Johnsen *et al.* [5] describe a template-matching method that uses local criteria, rejecting a match if any position in the error map is found to have four or more neighbors set to 1. In order to detect mismatches due to the presence of a thin stroke or gap in one image but not the other, another heuristic is used. Noticing that this method often produced false matches on small characters, Holt [4] used two different criteria to detect thin strokes or gaps—one for large characters and the other for small ones.

We began by simply rejecting the match if any square block of a certain size was found to be set in the error map. A crude normalization for symbol size was performed by rejecting if a 3×3 block was found, unless the height or width were less than 12 pixels in which case we rejected on the basis of a 2×2 block. However, this missed thin strokes—for example, c's were confused with e's. Consequently we implemented Holt's rule [4] to seek thin strokes in the error map. We had to modify this because our test images are digitized at 400 dpi, twice the resolution he used.

After some experimentation we settled on three categories of symbol: large ones (most letters) whose

height and width both exceed 12 pixels; small ones (typically punctuation) where both are less; and the rest (thin lowercase letters such as 'i' and 'l'). For the first category a match is rejected only if two or more 3×3 error blocks are found, while for the last a single 3×3 error suffices. In both cases, a thin line detector is implemented based on [4] and with slightly different criteria for large and intermediate symbols. For small symbols, a match is rejected if an error of 6 pixels is found (with no thin line detection). The reason why one 3×3 error is tolerated in the case of large symbols is that 'identical' letter g's often exhibit a difference of this size—and indeed the results show that the present algorithm still has a tendency to mismatch g's.

Of course, whereas in optical character recognition mismatches are serious errors, in our application they merely cause a small penalty in compression efficiency. The reproduced image will still be a faithful copy of the original.

Constructing the library

The result of template-matching is a provisional library which stores the first-encountered variant of each symbol, along with the set of all symbols that matched it. From this are discarded 'singletons,' that is, symbols that have occurred just once. This removes almost all the noise symbols (and also genuine characters that occur only once). Then each template in the library is replaced by an averaged version in which a pixel is set if it appears in more than half the symbols that matched. This removes the arbitrariness of storing the first-encountered variant, and minimizes the number of bits set in the residue.

Identifying white space

The symbol-extraction process does not recognize white space as a symbol in its own right—any space following a symbol will manifest itself merely as an unusually large x-offset. However, experiments with ordinary text files show that compression performance can deteriorate by up to 5% if space characters are removed. Therefore we experimented with inserting a special code, known in advance to encoder and decoder, into the text between words and at the end of lines. What distinguishes spaces from tabs from newlines is the size of the x (and y) offset associated with the symbol.

Coding the symbol numbers

The symbol numbers illustrated in the second column of Figure 2 are not conventional character codes. For one thing symbols are ordered in the sequence in which they happen to appear in the image, rather than alphabetically; for another, some characters (like 'i') generate more than one symbol. However, this does not affect text compression methods that begin with no preconceptions about the kind of information being coded, and any such 'adaptive' scheme is suitable. We use the well-known PPMC technique [1, 2, 7]. Its performance will depend on the success of the symbol extraction method. If, for example, two variants of a certain character find their way into in the library, (up to) one extra bit will be needed whenever the character occurs to distinguish them.

Coding the offsets

The x- and y-offsets are compressed by conditioning them on the symbol with which they are associated, and using adaptive coding. For every symbol, all offset values associated with it so far are stored, along with their frequency counts. If the present value has followed the symbol before, it is coded (using arithmetic coding) according to this frequency distribution. If not, an escape code is sent and the offset is coded according to the frequency distribution of all values that have occurred so far (regardless of what symbol they followed). If that particular offset value has never occurred before, a further escape is sent and the value is coded according to a precomputed, fixed, probability distribution that corresponds to Elias's γ variable-length coding of the integers (see Appendix A of [1]).

Component	Coding method	Single image	Average over five images	Fifth image using model from other 4
library of symbols	2-level coding	1466	2149	134
the symbol sequence	PPMC	760	1336	660
x- and y-offsets	order-1	1972	2536	1601
residue	see text	21798	27452	21297
Total:		25996	33473	23692
original image	2-level coding	34176	43556	33614
original 3900×1120 image	uncompressed	546000	546000	546000

Table 1: Size (in bytes) of each component of the representation for the test images

Coding the library

The library of symbols is coded using normal image compression techniques. It is represented as an image from which members are extracted by the decoder using the same process that the encoder uses to extract symbols from text. Experiments show that the two-level coding scheme devised by Moffat [8], with the 22/10 bit context in his Figure 5, gives the best performance for sample library images.

Coding the residue

The residue is the bitwise exclusive-OR of the original image and the reconstructed text. Because of the success of the character extraction process, far fewer bits are set in it than in the original image. This would seem to indicate that the residue can easily be coded much more efficiently than the original. However, this is not so. When good compression methods are applied, there is little difference in their compressed size. In fact, with the best two-level coding scheme of [8], the compressed residue was slightly larger than the compressed form of the original image!

We found this result disappointing, although with hindsight it is perhaps not surprising. The original image is far more compressible than the residue precisely because most of the black pixels it contains form predictable parts of characters. When the symbols are extracted, what is left is the *noise*—the irregularities around the edges that are caused by deficiencies in the printing and scanning processes—and this is very difficult to compress.

Fortunately, the residue can be coded more efficiently. There is clearly considerable overlap in information content between it and the reconstructed text image—many of the characters can be discerned from the residue alone. Advantage can be taken of this by conditioning the coding of the residue on the reconstructed text image, as well as on that part of the residue coded so far. This is particularly effective because the entire reconstructed text is known before any of the residue is coded. Thus Moffat's 'clairvoyant' templates can be used [8], which assume that all pixel values surrounding the current one are known. We use as context *both* a regular template on the residue coded so far, *and* a clairvoyant template on the reconstructed text image.

Experimental results

Table 1 summarizes the overall result. The five test images from the Trinity College Library catalogue are each 3900×1120 pixels (they actually represent single columns from two-column pages). Using the best conventional compression method we could find, the two-level coding scheme [8] with a 22/10 bit context, the first image is reduced to only 6.2% (34,176 bytes). The new method reduces it to 4.8% (25,996 bytes). The average over the five test images is also shown: Moffat's method compresses them to 8.0% and the new method to 6.1%. The reason why better results are achieved on the first image is

that it is a short column containing a large stretch of white space after the text.

These figures are for the situation where single images are coded in isolation. If several images are processed and then a further one is coded, significant economy is achieved because the overheads are reduced. The extra information needed to code the fifth image (chosen to be the one used for the single-image tests, and that of Figure 1) once the other four have been processed is shown in the final column of Table 1. The total is nearly 10% smaller than that for coding the image in isolation. If a conventional compression technique such as Moffat's is used, the gain from pre-adapting to the other four images is much smaller (1.6%).

We now review the success of the individual processes.

Coding the residue

The residue is by far the largest component of the compressed image. Different combinations of Moffat's regular (Figure 3 of [8]) and clairvoyant (Figure 6 of [8]) templates were evaluated (although in the latter we include the center pixel as well). The 4-bit regular template combined with a 13-bit clairvoyant one works best.

As can be seen from Figure 5, some unidentified symbols—singletons—appear in the residue. In order to determine whether these affect its compression significantly, we removed *all* symbols and re-coded it using the same technique. The size reduced to 20,871 bytes, down by only 4% from the figure in Table 1. It seems that most of the space occupied by the coded residue is inherent in the fact that it is a noisy image—it will be difficult to reduce its size much further.

Symbol extraction and library coding

For the five test images, 359 symbols were initially placed in the library, and this was reduced to 204 by the consolidation process (eliminating singletons). Of these, 123 were valid typographical symbols; 52 were duplicate copies of symbols; five were 'false ligatures,' which are double characters formed by two separate ones that have run together in the printing process (a 'true' ligature is a pair of letters such as 'ff' or 'fi' that are normally printed as a single symbol); 24 were fragments of characters where a printing imperfection has omitted part of a symbol or broken it in two. Of course, all of these imperfections must have occurred twice in order to form part of the final library.

Of the 155 discarded singleton symbols, most were imperfectly formed variants of symbols that had already occurred. Although 52 of these appeared in the library, a further 59 occurred only once and hence were ignored. The next largest category is 55 fragments of characters that were broken up in the printing process; again, some of these appeared two or more times and hence made it into the library. Our connectivity criterion could easily be weakened in an attempt to rejoin such fragments; however, this would inevitably increase the false ligatures—of which there are a lot already.

The final library for the five images together is shown in Figure 6. The reconstructed text is more complete when the library is built from several images. Of the 42 symbols from the first image that were left in the residue (of which Figure 5 shows a part), and are therefore absent from the reconstructed text (of which Figure 4 shows a part), 23 occur again in the remaining four images and so find their way into the library (for example, the 'LOUIS' that can be seen prominently in Figure 5).

With regard to actually coding the library—the image of Figure 3—the best method found was two-level technique with a 22/10 bit context [8]. The average over all five images, coded individually, was 22.1 bytes per symbol in the library.

Coding the symbols and offsets

For the image of Figure 1 the symbol sequence itself, which contains 1800 symbols, is compressed by PPMC into 760 bytes, or 3.38 bit/symbol. Representing the 65-member alphabet directly would require just over 6 bit/symbol, and so appreciable use is made of regularities present in the sequence—even for this rather short stretch of text. The average figure over the five images, coded individually, is 4.28

bit/symbol using PPMC as against 6.60 bit/symbol if the alphabet were represented directly. Extracting the characters in the correct sequence was difficult for some pages that were set with very little leading between the lines of type, and so the efficiency of the text coding might be improved if a better extraction method was used.

For the single test image, the x- and y-offsets are coded in 1972 bytes, or 4.38 bit/offset. The average value over the five images, coded individually, is 4.06 bit/offset.

Identifying white space

The idea behind the identification of white space was to improve the coding of both symbol sequence and character offsets: the former because representing spaces as symbols provides a word-boundary cue that is helpful for predicting character sequences; and the latter by making the set of offsets that follow a particular symbol more consistent, removing the exceptional condition that corresponds to word endings. In fact, this was only marginally successful.

Spaces are inserted fairly consistently, except of course in the above-mentioned case where symbols were extracted in the wrong order, which not surprisingly causes the space-insertion module considerable confusion. Some difficulty was encountered with punctuation, particularly the '.' symbol which serves double duty as a period and as the dot over 'i', 'j', ';', and ':'.

The final result was that the text, after compression, is about 7% to 10% larger after spaces have been inserted than it is without spaces. Considering that over 20% of the characters are spaces, this is not too bad. The offsets are about 2.5% smaller. These take more code space than the text, and the combined result just about breaks even—the space-insertion process causes the total compressed size to grow by about 0.5%. Probably if the characters had all been assigned to their correct lines of text (as they would be if the leading was greater), space-insertion would have proven just beneficial.

System considerations

Our research was motivated by the Trinity College Library catalogue problem, and it is instructive to consider the overall needs of that and similar applications. A typical catalogue will contain approximately 1,000 to 1,000,000 images. Users access it in various different ways: by browsing from page to page, giving a page number, or specifying other information such as a book title or author name (in which case all matching pages must be retrieved). We assume that all access is through bit-mapped display screens.

For most queries an approximate image is sufficient, and for this our proposed compression regime is particularly useful. To display a page in full, the symbol library must be decoded, then the list of symbols and (x, y) offsets, and finally the residue. The residue is an order of magnitude larger than the other components, and, although necessary to obtain a faithful reproduction, is not generally needed for an overview of the page. In the reconstructed image of Figure 4, the original text can be read without difficulty, certainly enough for a casual browser to decide whether it is of interest, and in most cases enough for a user to find the shelf location of a book being sought. Only by express command, or a request for a faithful printed copy, would the residual bitmap be retrieved and decompressed.

Although access will normally be to a single catalogue page, compression can be improved by building the symbol library for several pages and coding the symbols as a longer sequence. Decoding the residual image is the most intensive part of accessing the database, in terms of both bandwidth and processor time. Batching of the remainder of the information into blocks of pages has little impact upon retrieval performance, but improves the quality of the reconstructed text image (by gathering a more complete library) and yields a small compression saving (Table 1). The separation allows different media to be used for the different components. The symbol library and symbol list might be retained on fast magnetic disk, while the residual image could be held on a bulk storage device such as an optical disk or jukebox. This two-level structure caters particularly well to casual browsers, who tend to flick rapidly from one page to the next after a scan of just a few seconds.

We believe our compression techniques to be applicable to more general tasks of document archiving. Despite the phenomenal capacity of WORM optical disks, the huge space requirements of raw images make compression even more essential than for textual databases. Compared to decompression, the compression phase is relatively time consuming, which is unfortunate because this is the most frequent operation in a document archiving environment. We suggest that compression should be carried out as a background process, with new documents spooled to some temporary holding area until they can be added to the permanent collection.

Conclusion

We have described a mechanism for compressing textual images, based upon identifying any repeated symbols appearing in the image, encoding these symbols and their locations, and encoding a residual bitmap to allow faithful reproduction of the original image. Previous image compression algorithms were general purpose, in that they applied to any binary images; here, by exploiting knowledge of the contents, improved compression has been obtained. Experiments with pages of the Trinity College library catalogue show the technique to be effective, particularly when the two-stage progressive nature of the decompression process is taken into account.

The techniques we have described might also be applied to other families of source documents. For example, printed sheet music could be parsed into sequences of notes, which, together with the location of the staves, would give a crude representation of the image. Detailed information would be again coded as a residue image, so that an exact reproduction of any particular page would be obtainable.

Acknowledgements

We are most grateful to David Abrahamson of Trinity College, Dublin, for telling us about this problem and kindly supplying us with test images. This work is supported by the Natural Sciences and Engineering Council of Canada and the Australian Research Council.

References

[1] Bell, T.C., Cleary, J.G. and Witten, I.H. (1990) *Text compression*. Prentice Hall, Englewood Cliffs, NJ.

[2] Cleary, J.G. and Witten, I.H. (1984) 'Data compression using adaptive coding and partial string matching,' *IEEE Trans Communications* COM-32(4): 396–402; April.

[3] Holt, M.J.J. and Xydeas, C.S. (1986) 'Recent developments in image data compression for digital facsimile,' *ICL Technical Journal*: 123–146; May.

[4] Holt, M.J. (1988) 'A fast binary template matching algorithm for document image data compression,' in *Pattern Recognition*, J. Kittler (ed.) (Proc. Int. Conf., Cambridge). Springer Verlag, Berlin.

[5] Johnsen, O., Segen, J. and Cash, G.L. (1983) 'Coding of two-level pictures by pattern matching and substitution,' *Bell System Technical J* 62(8): 2513–2545; May.

[6] Langdon, G.G. and Rissanen, J. (1981) 'Compression of black-white images with arithmetic coding,' *IEEE Trans Communications* COM-29(6): 858–867; June.

[7] Moffat, A. (1990) 'Implementing the PPM data compression scheme,' *IEEE Trans Communications* COM-38(11): 1917–1921; November.

[8] Moffat, A. (1991) 'Two level context based compression of binary images,' in *Proc. DCC'91*, J.A. Storer and J.H. Reif (eds.), pp. 382–391. IEEE Computer Society Press, Los Alamitos, CA.

[9] Pratt, W.K., Capitant, P.J., Chen, W.H., Hamilton, E.R. and Wallis, R.H. (1980) 'Combined symbol matching facsimile data compression system,' *Proc IEEE* 68(7): 786–796; July.

— Resolutie van de staten generael der Vereenighde Nederlanden, dienende tot antwoort op de memorie by de ambassadeurs van sijne majesteyt van Vranckrijck.
's Graven-hage, 1678. 4°. Fag. H. 2. 80. N°. 20. Fag. H. 2. 85. N°. 17. Fag. H. 3. 42. N°. 4.

— Tractaet van vrede gemaeckt tot Nimwegen op den 10 Augusty, 1678, tusschen de ambassadeurs van [LOUIS XIV.] ende de ambassadeurs vande staten generael der Vereenighde Nederlanden.
Fag. H. 2. 85. N°. 21.

— Nederlantsche absolutie op de Fransche belydenis.
Amsterdam, 1684. 4°. Fag. H. 2. 50. N°. 22.

— Redenen dienende om aan te wijsen dat haar ho. mog. [niet] konnen verhindert werden een vredige afkomst te maken op de conditien by memorien van den grave d' Avaux van de 5 en 7 Juny, 1684, aangeboden.
[*s. l.*] 1684. 4°. Fag. H. 2. 86. N°. 3. Fag. H. 2. 96. N°. 8. Fag. H. 3. 44. N°. 52.

— Redenen om aan te wijsen dat de bewuste werving van 16000 man niet kan gesustineert werden te zullen hebben konnen strekken tot het bevorderen van een accommodement tusschen Vrankrijk en Spaigne.
[*s. l.*] 1684. 4°. Fag. H. 2. 86. N°. 4. Fag. H. 2. 96. N°. 2.

— D' oude mode van den nieuwen staat van oorlogh.
[*s. l.* 1684]. 4°. Fag. H. 2. 86. N°. 12. Fag. H. 2. 96. N°. 3.

— Aenmerkingen over de althans swevende verschillen onder de leden van den staat van ons vaderlant.
[*s. l.*] 1684. 4°. Fag. H. 2. 92. N°. 1. Fag. H. 2. 98. N°. 16. Fag. H. 3. 1. N°. 18.

— Missive van de staten generael der Vereenighde Nederlanden, . . . 14 Maert, 1684.
's Graven-hage, 1684. 4°. Fag. H. 2. 92. N°. 10.

— Missive van de staaten generael der Vereenigde Nederlanden, . . . 11 July, 1684.
[*sin. tit.* 1684]. 4°. Fag. H. 2. 96. N°. 13. Fag. H. 3. 44. N°. 69.

— Resolutie vande staten generael der Vereenighde Nederlanden, . . . 2 Maart, 1684.
's Gravenhage, 1684. 4°. Fag. H. 2. 92. N°. 11. Fag. H. 3. 44. N°. 9.

— Extract uyt de resolutien van de staten generael, . . . 31 Maert, 1684.
[*s. l.*] 1684. 4°. Fag. H. 2. 92. N°. 13. Fag. H. 2. 96. N°. 25. Fag. H. 3. 44. N°. 11. Fag. H. 3. 44. N°. 15.

— Antwoort van de staten generael der Vereenighde Nederlanden op de propositie van wegen sijne churf. doorl. van Ceulen, Maert 23, 1684, gedaen.
's Gravenhage, 1684. 4°. Fag. H. 2. 92. N°. 12.

Figure 1 A test image

symbol	symbol number	x-offset	y-offset
start-of-page	0	19	62
—	1	23	7
R	2	1	-1
e	3	2	0
s	4	3	-1
o	5	1	0
l	6	2	0
u	7	1	0
t	8	2	0
l	9	-9	-25
.	10	5	25
e	3	15	0
v	11	2	-1
a	12	2	0
n	13	16	0
d	14	2	0
e	3	24	-1
s	4	3	0
t	8	2	0
a	12	1	0
t	8	2	0
e	3	2	-1
n	13	22	10
g	15	2	-10
e	3	2	-1
n	13	2	1
e	3	2	-1
r	16	3	0
a	12	2	0
e	3	1	-1
l	6	17	0
d	14	2	0
e	3	3	-1
r	16	17	0
V	17	1	0
e	3	1	-1
r	16	2	0
e	3	2	1
e	3	2	-1
n	13	2	-1
i	9	-8	-24
.	10	4	35
g	15	1	-11
h	18	2	0
d	14	2	0
e	3	-1012	53
N	19	1	-1
e	3	3	0
d	14	1	0
	•	•	•

Figure 2 Symbols and x- and y-offsets created from the test image

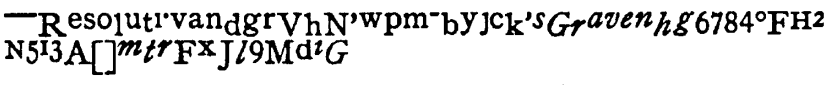

Figure 3 Library of symbols created from the test image

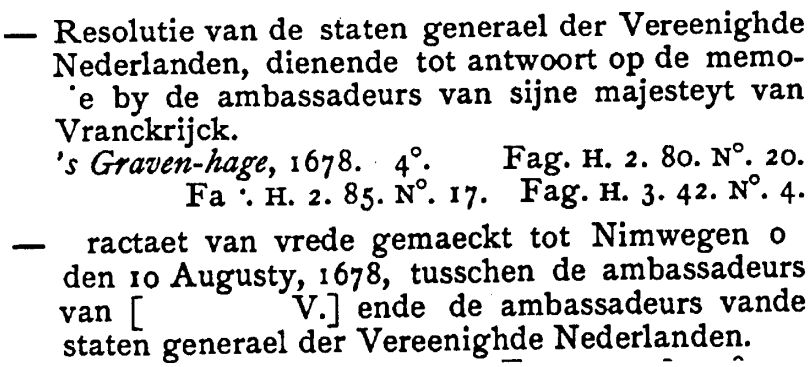

Figure 4 The 'reconstructed text' image, recreated from the information in Figures 2 and 3

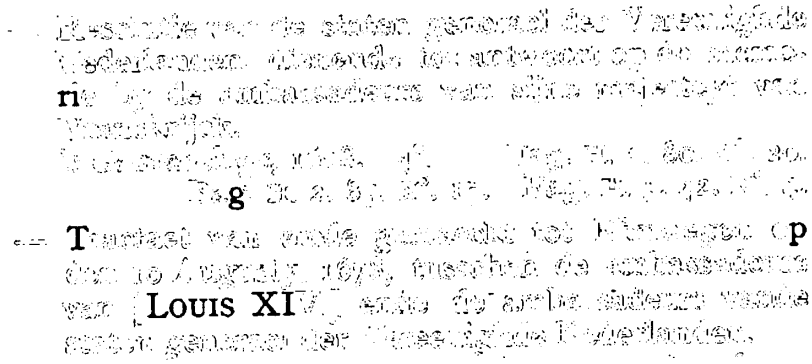

Figure 5 The 'residue' image, which is the difference between the original and reconstructed text

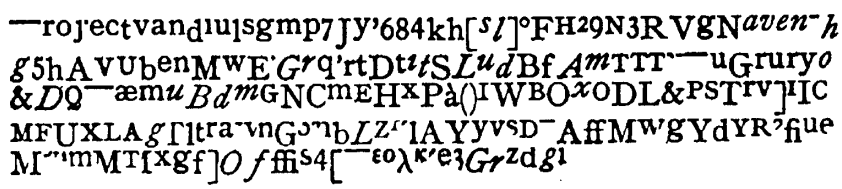

Figure 6 Library of symbols created from the 5 test images together