Embedding Meta-Information in Handwriting — Reed-Solomon for Reliable Error Correction

Marcus Liwicki*[†], Seiichi Uchida*, Masakazu Iwamura[‡], Shinichiro Omachi[§] and Koichi Kise[‡]

*Kyushu University, Japan
Email: uchida@ait.kyushu-u.ac.jp

[†]DFKI, Germany
Email: marcus.liwicki@dfki.de

[‡]Osaka Prefecture Univ., Japan

[§]Tohoku Univ., Japan

Abstract

In this paper a more compact and more reliable coding scheme for the data-embedding pen is proposed. The data-embedding pen produces an additional ink-dot sequence along a handwritten pattern during writing. The ink-dot sequence represents, for example, meta-information (such as the writer's name and the date of writing) and thus drastically increases the value of the handwriting on a physical paper. There is no need to get access to any memory on the pen to recover the information, which is especially useful in multi-writer or multi-pen scenarios. In this paper we focus on the compactness of the encoded information. The aim of this paper is to encode as much information as possible in short stroke sequences. In our experiments we show that we can embed more information in shorter strokes than in previous work. In straight lines as short as 5 cm, 32 bits can successfully be embedded. Furthermore, the new encoding scheme also works reliably on more complex patterns.

I. Introduction

Handwriting is an important modality for writing down information, making annotations, or just marking items. Unfortunately, as soon as the ink is on the paper, many information known during writing is already lost. We cannot access meta-information about the handwritten pattern from itself; it is impossible to retrieve who wrote this pattern or when it was written. In other words, a handwritten pattern on a physical paper is just an ink pattern and thus cannot provide any information but its shape.

Digital pens have emerged as a choice to store and retrieve such meta-information — unfortunately, they cannot increase the value of handwriting on paper either. Several digital pens capturing handwriting on normal paper have been developed and those pens can store the stroke sequences on a computer along with meta-information. However, the handwriting left on the paper is still just an ink pattern without any meta-information.

In this paper, we work with a novel pen device which enriches handwriting on physical paper. The pen device, called *data-embedding pen*, can embed arbitrary information (such as meta-information) by an additional ink-dot sequence along the ink stroke of the handwriting. Each ink-dot represents an information bit and thus an ink-dot sequence represents a bit-stream of the information to be embedded.

While we have introduced the data-embedding pen very recently [1], the work presented in this paper adds significant scientific value. In [1] we mainly focused on the feasibility of encoding meta information. There we have investigated the influence of several parameters, like the sampling rate and the length of the code units, to the coding reliability. For error correction, row- and column-wise parity bits were chosen, and finally, about

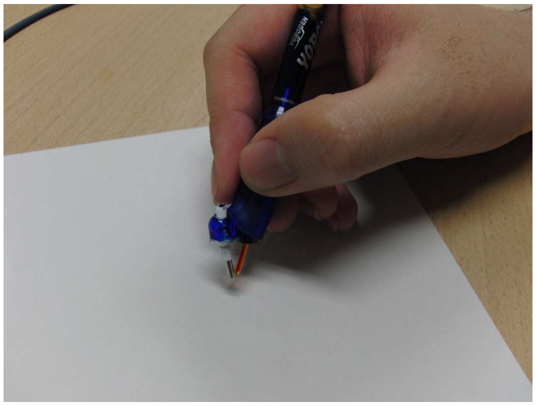

Figure 1. The data-embedding pen.

21 bit of information could be successfully recovered from simple handwritten patterns. In this paper we propose the use of Reed-Solomon error correction and present a way of applying it to the ink-dot sequences. In our experiments we can successfully encode more than 30 bit of information in even shorter sequences. Additionally, the new encoding scheme works successfully on more complex patterns like words and signatures, where the previous approach usually failed.

II. The Data-Embedding Pen

The data-embedding pen is a device which comprises a usual ballpoint pen and an ink-jet nozzle element. Figure 1 depicts this device. The main casing is used as a channel for the nozzle (right side in Fig. 1) and side-channel is used for the ballpoint pen (left side in Fig. 1). During the writing, the nozzle produces small ink-dots alongside the handwritten stroke. The color of the ink-dots is different from the color of the stroke. In this paper, yellow is used for the ink-dots. (Invisible ink has already been tested as a good alternative.) The number of the ink-dots and their timing are used to encode the desired information.

The nozzle is able to generate up to $2,000$ ink-dots per second. Using this high frequency, we can form a connected line by a sequence of several ink-dots. Hereafter, a line by n sequential ink-dots is called *n-pulse line*. If $n = 1$, the n-pulse line forms a single ink-dot. The line, of course, becomes longer by increasing n.

III. Information Embedding

Our coding scheme is based on the combination of three different n-pulse lines. Specifically, we use $n = 1$ (a dot), 5 (a short line), and 20 (a long line). The ink-dot sequence of Fig. 2 consists of those n-pulse lines. Roughly speaking, the information is converted into a binary (0 and 1) sequence and embedded by using the 1-pulse line as 0 and the 5-pulse line as 1. A short pause is prepared between each bit information (1-pulse or 5-pulse line) like in the Morse code. The 20-pulse line, hereafter called *synchronization blob*, is used as an anchor to make sure that a correct position is extracted (see the leftmost dot in Fig. 2).

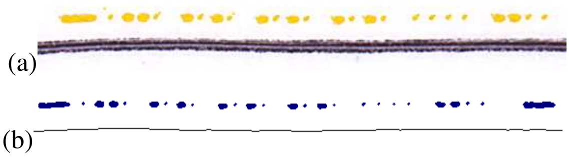

Figure 2. (a) Ink-dots (light) nearby a handwriting stroke (black). (b) After image processing.

Our coding scheme is defined by three units, called *frame*, *block*, and *bit*. (This naming is motivated by the terminology of network protocol design.) The bit is the smallest unit and defined by a 1-pulse line or a 5-pulse line. Several consecutive bits comprises a block and several consecutive blocks comprises a frame. A pause which is longer than the pause between bits is inserted between two consecutive blocks. Each frame begins (or, equivalently, ends) at a synchronization blob.

Figure 2 is an example of a single frame. From left to right, the ink-dot sequence of the frame is comprised of a synchronization blob, 6 blocks, and another synchronization blob. In each block, 4 bits are encoded and thus in the frame 24 bits ($0110 - 1010 - 1010 - 1010 - 0000 - 1100$) are embedded.

The main parameters of the coding scheme are the number of bits per block (bB) and the number of blocks per frame (bF). Accordingly, the number of bits per frame becomes $bF \times bB$. In the example of Fig. 2, $bF = 6$ and $bB = 4$.

Another important parameter, which is the main focus of this paper, is the method for correcting errors which eventually occur during embedding the information. More details about this issue follow in Section V.

IV. Information Recovery

A. Image Processing

Information recovery begins with image processing which extracts ink-dots and black ink strokes from a scanned image. In the this section, four steps of image processing are explained using Fig. 4 (a), which is an intersection part of Figure 3.

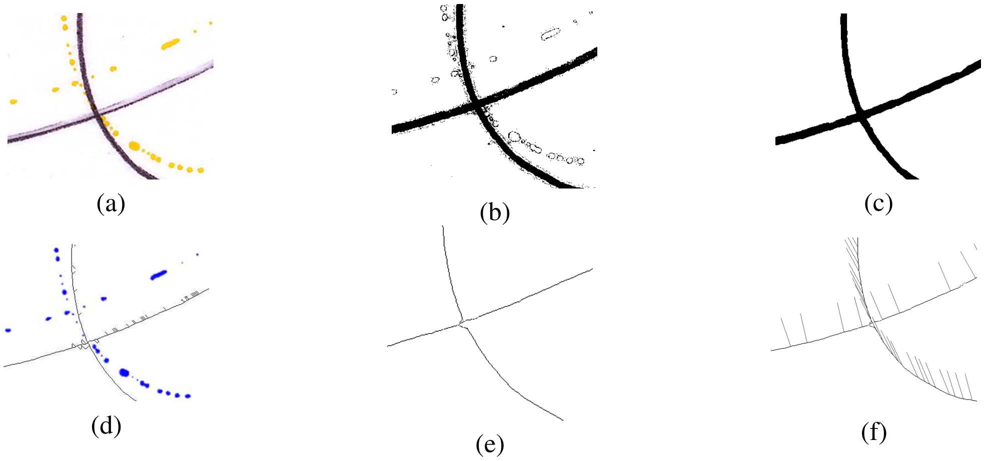

Figure 4. Image processing on an intersection part of Fig. 3. See text for details.

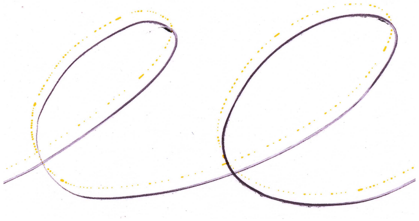

Figure 3. Two loops with ink-dots.

The first step of ink-dot extraction is a simple thresholding operation to extract the black ink stroke and yellow ink-dots. The second step is noise removal because the black ink stroke image extracted includes many noisy pixels, as shown in Fig. 4 (b). Thus, erosion and dilation are applied. Figure 4 (c) shows the result. Similar operations are also applied to the ink-dot image (Fig. 4 (d)). Note that the parameters for those operations can be optimized on a small training set.

The third step is a special treatment of ink-dots occluded by the black ink stroke. Fortunately, those yellow ink-dots are still visible on the stroke (they just appear to be a bit darker). Thus, after extracting the pixels of the black ink stroke, another thresholding operation is performed on those pixels with a lower threshold to recover dark yellow ink-dots. In the following experiments it turned out that about 50% of those dots could be recovered by this approach.

The fourth step is a thinning operation on the black ink stroke. Figure 4 (d) shows the result of an orthodox thinning method. Then, after removing many small loops and short spurious edges by unifying neighboring branches, the final thinning result is obtained as shown in Fig. 4 (e).

B. Aligning Ink-Dots by Stroke Recovery

In order to decode the ink-dots, they should be aligned according to their original temporal order. Since this order is lost in the scanned image, we must estimate it by using the result of stroke recovery. Specifically speaking, after recovering the writing order of the black ink stroke based on the algorithm presented in [2] and establishing the correspondence between the ink-dots and the stroke, we align the ink-dots.

The basic idea of establishing the correspondence as shown in Fig. 4 (f) is to find the closest point on the stroke for each ink-dot. A simple nearest neighbor, however, cannot always provide a correct result because a dot and its corresponding point might be a bit distant due to the pen tilt. Thus, at each ink-dot k, we first calculate the minimum distance $d_{k,\theta}$ to the stroke for each θ of 36 directions (with 10° interval). Then, we select the direction $\overline{\theta}$ with minimum variance, i.e., $\overline{\theta} = \operatorname{argmin}_\theta \operatorname{Var}\{d_{1,\theta}, \ldots, d_{K,\theta}\}$. This direction is the most stable direction and thus represents the pen tilt. Finally, for each ink-dot k, the corresponding point is determined as the closest point in the direction $\overline{\theta}$.

C. Data Decoding

For decoding, the bit information (i.e., 1-pulse and 5-pulse lines and synchronization blob) is first recovered at every ink-dot, just by checking its size. The sequence is separated into frames using the synchronization blobs. Larger gaps are detected within each frame and assumed as the gaps between block.

Next, a plausibility control is performed on the extracted data. For each block, the number of bits (bB) is confirmed. Sometimes a block has spurious bits,

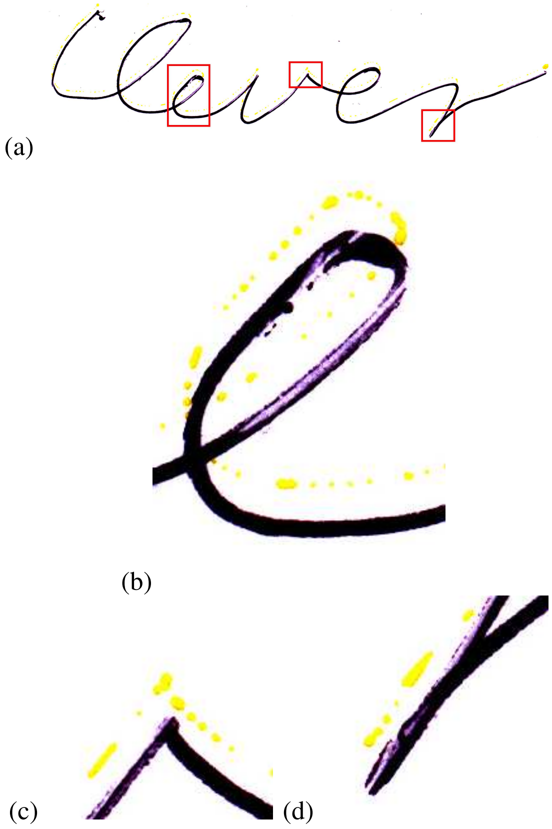

Figure 5. (a) Example of a difficult pattern. (b)–(d) Specific regions of the pattern.

resulting from a wrong mapping or just from noise. In this case, those adjacent bits whose distance deviates too much from the mean distance are deleted. If the number of bits and blocks do not correspond to the values bB and bF, the frame is rejected.

For detecting and correcting errors, the Reed-Solomon error correction is chosen. Details follow in the next section.

V. Error Correction

The process of embedding the ink next to the handwritten stroke is always accompanied with several errors. First, the black ink sometimes overlaps with the information ink (see Fig 5 (b)). Second, several ink-dots might overlap at turning points or stopping points (see Fig 5 (c)). Finally, it is impossible to recover the correct information from double strokes (see Fig 5 (d)), since it is not known which dot belongs to which direction.

In order to recover from the errors, some redundant information has to be added. Simple and intuitive ideas would be to apply repetition and parity check [1]. However, these encodings show some limitations, especially when it comes to more complicated handwritten patterns like signatures or handwritten words with many crossings and double strokes.

In this paper we use Reed-Solomon error correction [3], [4] for reliably recovering from the occurring errors. The idea is to oversample a polynomial $f(x) = a_1 + a_2 * x^1 + \ldots + a_k * x^{(k-1)}$ from the data with more *points* a_j than needed. This makes the polynomial overdetermined. Therefore it is not needed to recover all points correctly as long as enough points are present. The only drawback of this encoding is that the position j of each point a_j needs to be known for a reliable decoding. In this paper we design each frame to be comprised of two blocks, the first block for the position of the point and the second block for its value. While the details of Reed-Solomon codes can be found in [4], in this paper only the important parameters and properties of this encoding scheme are given.

The first parameter of Reed-Solomon encoding is the base m bits for the points. In this paper we have set $m = 4$. This choice of this value is based on the observation that previous experiments have shown that shorter frames have a higher probability of being correctly decoded. Each frame consists of 8 bit; 4 bit for the position and 4 bit for the value.

The next parameter is the length n of the code (including data and error correction bits). Typically this value is set to its maximum value $n = 2^m - 1$ (the values have to be non-zero). This code is divided into k data points (the data to be encoded) and $n - k$ points for error correction. Given the k data points $a_1, \ldots, a_k$ (a message to be encoded), the other values $a_{(k+1)}, \ldots, a_n$ of the polynomial are determined and all n points are encoded (sent).[1]

In the decoding phase, not all n points need to be correctly recovered. Assuming that c points were correctly recovered, s points are missing (erasures) and e points are erroneous, the code can still be correctly decoded if the following equation holds:

$$2e + s \leq n - k \qquad (1)$$

This important property makes the Reed-Solomon codes very useful for applications where burst errors occur. In our case usually the a whole block can be either recognized or not, i.e., it rarely occurs that just one bit is missing (even if only one bit is missing, we do not know the position of the bit).

Since we encode the positions of the points in the frame, the positions of the missing points are known

[1] The determination of the values is based on the primitive element of the finite field α and finding a function $f(x)$ for which holds $f(\alpha^{(i-1)}) = a_i$, for $i = 1, \ldots, k$ and then applying $f(x)$ to the remaining $\alpha^i, i = k, \ldots, n-1$.

before decoding. In the extreme case, up to $n-k$ points can be missing and still it would be possible to decode the information correctly. In the other extreme case, i.e., if there is no missing point, up to $(n-k)/2$ errors are allowed to occur, which means for each erroneous point, one more correct point should be at hand.

VI. Experiments and Results

A. Data

Two sets of data-embedded handwriting were collected using the current pen prototype. The first set (Set1) contains 50 horizontal straight lines with a length of 20 cm. All lines have been drawn with approximately the same velocity. (Experiments with varying velocity appear in [1].) The second set (Set2) contains patterns which might appear in a real world scenario, i.e., 12 "@" symbols, 12 checkmarks, 12 simulated signatures, and 12 instances of the handwritten word "Clever". The former two symbols have sizes of 3×3 cm at maximum, the latter symbols have a size of 4×3 cm.

B. Reed-Solomon Encoding

For the Reed-Solomon encoding, the Shifra Open Source error correcting code library was used[2]. We used a Galois field polynomial of the order 4. The code length was fixed to 15 points ($2^4 - 1$), each point being a hexadecimal number (4 bit).

The aim of the experiments is to find a suitable value k for the number of data points. Therefore it was varied from 1 to 15. This was achieved by applying the following strategy. First, we set $k = 1$ with $a_1 = 1$ and computed the other values a_i for this setting. The resulting code word is $1, 9, 13, 15, 14, 7, 10, 5, 11, 12, 6, 3, 8, 4, 2$. If we now set $k = 2$ and $a_2 = 9$, the same values would be estimated, and so on. This means that only the encoding of this code is needed and during decoding we can choose the actual value for k. This makes the full use of all collected data, i.e., it is not needed to write down new patterns for each value of k. Note that using this strategy also eliminates side-effects like more noise in some patterns, because always the same patterns are used for the evaluation.

The code was always repeated in the data, i.e., after reaching f_{15}, we began with f_1 again.

Table I. Percentage (%) of correctly recovered information for Set1 (varying line lengths)

# data points (k)	# bits	5 cm	10 cm
1	4	100	100
8	32	100	100
9	36	94	100
10	40	72	100
11	44	56	100
12	48	20	100
13	52	0	100
15	60	0	92
Coding scheme of [1]*			
	28	50	100
	56	22	46

* Note that for the coding scheme of [1] the amount of decoded information is presented, e.g. 46 % means that often half of the information could be recovered, while never all 56 bits were correctly recovered.

C. Results for Set1

In the experiments on Set1 we wanted to find out how much information can be embedded in straight lines. In this task only rarely some decoding errors occur on the frame level, since there are no crossings. (Figure 6 provides an example for the extraction result of a 5 cm long part of a straight line where no errors occurred.) The only problem were some overlapping ink-dots if there was a slow pen-movement. This happened in about 10 % of the frames. Note that these frames were rejected during the frame decoding step presented in Section IV-C, resulting in missing points for the Reed-Solomon error correction.

As stated above, the straight lines had a length of 20 cm. Since the code was repeated, no errors occurred on these long lines. We decided to measure the results on shorter lines. Therefore we cut the line first into 10 cm parts and finally into 5 cm parts.

The results of the experiments on Set1 appear in Table VI-C. This table shows the percentage of samples where the information could be correctly recovered by using the Reed-Solomon error correction. Up to a number of $k = 8$ data points the codeword was always correctly recovered even for straight lines as short as 5 cm. For larger k value, the performance decreases, because only a limited number of frames appear in a 5 cm line. (In Fig. 6, for example, 10 points (frames) appear.) For the length 10 cm there were only problems if no error correction point appears, i.e., in 8 cases there was a missing point which could not be recovered.

In the bottom of Table VI-C the decoding performance for the coding scheme of [1] (parity bits for each frame) is given. The Reed-Solomon error correction obviously allows a more compact code. For a length of 5 cm, the recovery rate of 28 bits (even 32 bits) is twice as high than with the parity bits. The

[2] Available at (2010): http://www.schifra.com/

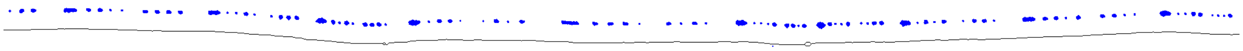

Figure 6. Extraction result (after thinning) of a 5 cm long line of Set1 (enlarged).

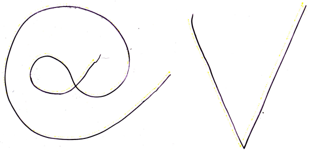

Figure 7. Example images of Set2.

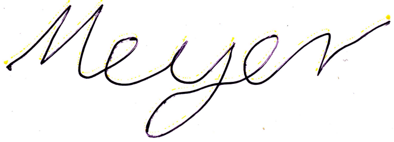

Figure 8. Example of the signature Meyer.

Table II. Percentage (%) of correctly recovered information for Set2

k	# bits	hook	@	Meyer	Clever
8	32	100	100	100	100
9	36	83	100	75	100
10	40	75	92	67	100
11	44	58	83	33	100
12	48	50	67	0	83
13	52	16	7	0	42
14	56	0	0	0	0

only drawback is that Reed-Solomon codes lose all information if not enough data points are present, while with the previous method still parts of the data could be correctly recovered. However, in practical applications it is often important to recover all information.

D. Results for Set2

As stated above, Set2 consists of patterns which might occur in practice. Examples for these patterns are shown in Figs. 5, 7, and 8.

Table II presents the results of the experiments on Set2. On all patterns codes of length 32 could be correctly recovered. It is a very interesting result that even on the more complicated patterns the correct information could be decoded. The main reason for unsuccessful decoding are either missing points (for short sequences like the hook) or some errors, e.g., a 1-bit was interpreted as a 0-bit if it was partially occluded by black ink (first frame of Fig 5 (b)).

VII. Conclusions

In this paper we have presented recent results of the research on the data-embedding pen. This pen makes it possible to augment handwritten patterns with meta-infomation like the time of writing, the writer ID, and other application-dependent data. The main idea is to encode the desired information in an ink-dot sequence plotted nearby the writing strokes. The hardware design as well as the methods for embedding and recovering information have been also described.

We proposed the use of the Reed-Solomon error correction scheme for successfully encoding and recovering the meta-information. The Reed-Solomon correction scheme uses an overdetermined polygon for encoding the data. During decoding only as many correct points are needed as the number of data points, disregarding their position. The other points might be missing. For each erroneous point one more correct points is needed to recover from the error.

In our experiments we have shown that the Reed-Solomon error correction scheme is very useful if applied as proposed in this paper. In a first set of experiments, the length of the handwritten stroke could be reduced to the half, compared to previous encoding schemes, still keeping the amount of information. That is, using a stroke length of just 5cm, 32 bits of information could be successfully embedded and recovered from straight lines.

In the second set of experiments we have used more complex patterns, ranging from symbols to handwritten words. Even in this setup we could always recover 32-bit of information. Note that 32 bit is enough to distinguish 2^{32} people. This implies that if a company uses this tiny marks for showing that a certain employee has checked a document, it is possible to identify which employee has checked the document. Also note that small read/write RFID-cards usually allow to store the same amount of information (32 bit).

References

[1] M. Liwicki, S. Uchida, M. Iwamura, S. Omachi, and K. Kise, “Data-embedding pen — augmenting ink strokes with meta-information,” in *9th Int. Workshop on Document Analysis Systems*, 2010.

[2] Y. Kato and M. Yasuhara, “Recovery of drawing order from single-stroke handwriting images,” *IEEE Trans. Pat. Anal. Mach. Intell.*, vol. 22, no. 9, pp. 938–949, 2000.

[3] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Code*. New York: North-Holland Publishing Company, 1977.

[4] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.