CS3236 Project Report
An investigation on 3 different linear coding schemes

Xiang Pan
A0092084E
xiangpan.cs@gmail.com

1 Introduction

In this project, we investigate different error-correcting codes and compare their performances across a noisy binary symmetric channel. In particular, we look at the following 3 linear codes:

(i) The (7,4) Hamming Code

(ii) A (9,5) code provided in suggested solutions to the first homework

(iii) The $\mathcal{R}(1,3)$ Reed-Muller code

2 Encoding Schemes

Since all 3 of the investigated error-correcting codes are linear codes, we can represent them using generator matrices. In this section, we show their generator matrices, explain how to decode the transmitted messages and analyze their theoretical performance.

2.1 $(7,4)$ Hamming Code

The $(7,4)$ Hamming Code is a well-studied error-correcting code. It is a simple linear code with many interesting properties. For example, it is a perfect 1-error-correcting code[1], in the sense that any 7-bit binary string is within distance 1 away from exactly one Hamming codeword.

The idea behind the $(7,4)$ Hamming code is to pad on 3 parity bits (r_5, r_6, r_7) to a 4-bit binary string (r_1, r_2, r_3, r_4). Doing so allows us to correct up to 1 error during transmission. The Venn diagram below shows a graphical representation of how the parity bits are chosen - within each circle, we aim to keep an even parity of 1's.

[1]The only other non-trivial perfect codes are Golay codes

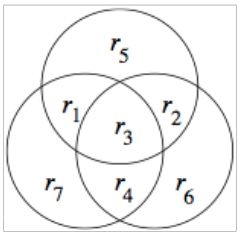


Figure 1: Hamming Code Venn Diagram

While the diagram is useful for visualization, a more efficient way to encode a message is to multiply it with the generator matrix $\mathbf{G}$.

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix} = \left[I_4 | P^T\right]$$

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} = [-P | I_3]$$

To encode a message $\mathbf{x}$ of length 4, we perform $\mathbf{G}^T \cdot \mathbf{x}$. From the transmitted message $\mathbf{t}$, we first compute the syndrome vector $\mathbf{z}$ via $\mathbf{z} = \mathbf{H} \cdot \mathbf{t}$. Based on the results of $\mathbf{z}$, we flip the i^{th} position bit according to the table shown below:

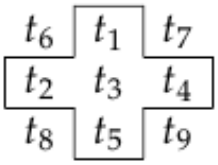
Syndrome $\mathbf{z}$	000	001	010	011	100	101	110	111
Flip this bit	–	r_7	r_6	r_4	r_5	r_1	r_2	r_3

As we send 7 bits for every 4 bits of information, the $(7, 4)$ Hamming code attains a rate of $\frac{4}{7}$. Also, since we are able to correct only up to 1 error during transmission, we have a block error whenever ≥ 2 out of the 7 transmitted bits are corrupted. In a binary symmetric channel with error probability f, that translates to a block error rate of $1 - \Pr(\text{No flipped bits}) - \Pr(1 \text{ flipped bit}) = 1 - (1-f)^7 - \binom{7}{1} f(1-f)^6$.

2.2 $(9, 5)$ Suggested Code

In the suggested solutions to the first homework, we are given the following error-correcting code:

> We encode a string of 5 bits ($k = 5$) $\mathbf{s} = (s_i)_{1 \leq i \leq 5}$ into a 9-bit string ($n = 9$) $\mathbf{t} = (t_i)_{1 \leq i \leq 9}$. The first 5 bits of $\mathbf{t}$ will be identical to those of $\mathbf{s}$, while the other 4 are parity bits. To choose the parity bits, we take inspiration from inscribing the bits as a cross within a square; parity bits are the sum of the three neighboring bits within the square:



In other words,
$t6 = t1 + t2 + t3, t7 = t1 + t3 + t4, t8 = t2 + t3 + t5, t9 = t3 + t4 + t5.$

As per the $(7, 4)$ Hamming code, it is more efficient to encode messages using a generator matrix $\mathbf{G}$ instead of directly making use of the diagram.

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} = \left[I_5 | P^T\right]$$

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} = [-P | I_4]$$

To encode a message $\mathbf{x}$ of length 5, we perform $\mathbf{G}^T \cdot \mathbf{x}$. From the transmitted message $\mathbf{t}$, we first compute the syndrome vector $\mathbf{z}$ via $\mathbf{z} = \mathbf{H} \cdot \mathbf{t}$. Based on the results of $\mathbf{z}$, we flip the i^{th} position bit according to the table shown below:

Syndrome $\mathbf{z}$	0000	1100	1010	1111	0101	0011	1000	0100	0010	0001	Others
Flip this bit	–	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	Error

For 'Error', we can choose not to flip anything since we will not be able to correct the block anyway. The analysis for this code is similar to that of the (7,4) Hamming code.

As we send 9 bits for every 5 bits of information, the suggested code attains a rate of $\frac{5}{9}$. Also, since we are able to correct only up to 1 error during transmission, we have a block error whenever ≥ 2 out of the 9 transmitted bits are corrupted. In a binary symmetric channel with error probability f, that translates to a block error rate of $1 - \Pr(\text{No flipped bits}) - \Pr(1 \text{ flipped bit}) = 1 - (1 - f)^9 - \binom{9}{1} f(1 - f)^8$.

2.3 $\mathcal{R}(1, 3)$ Reed-Muller Code

2.3.1 $\mathcal{R}(r, m)$ Reed-Muller Codes

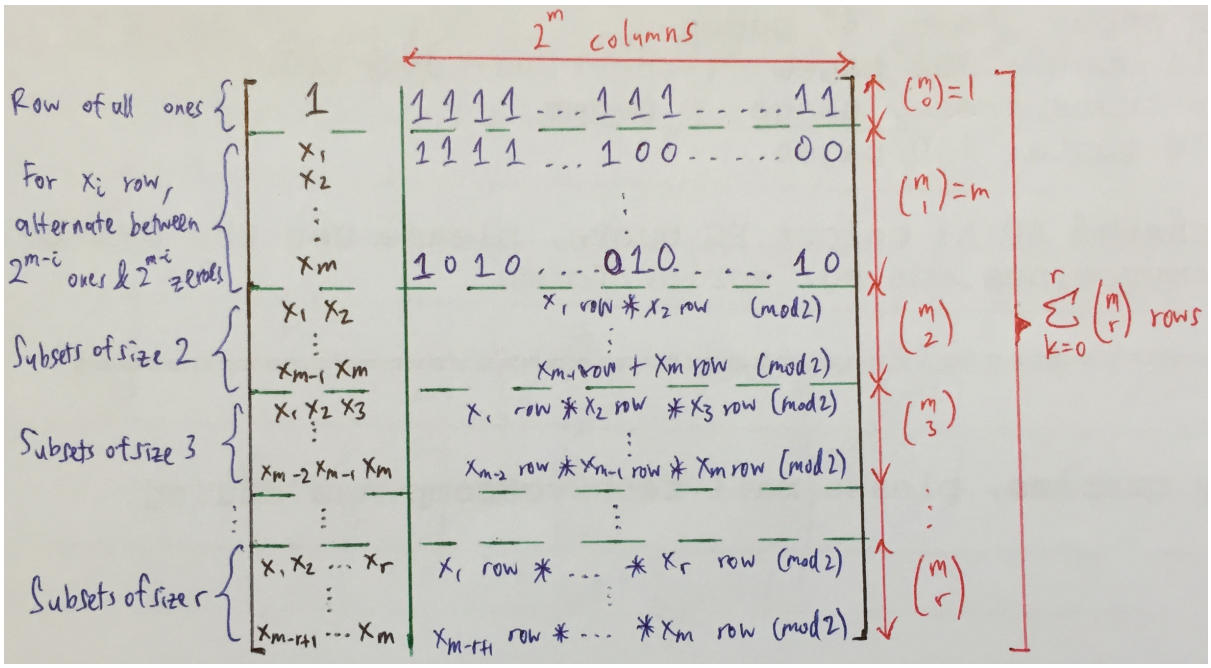
Reed-Muller codes are a family of linear error-correcting codes named after Irving S. Reed and David E. Muller. They are usually listed as $\mathcal{R}(r, m)$, where the r represents the order of the code and m represents the $\log_2$ length of the code (i.e. code length $= 2^m$).

There are several interesting properties about Reed-Muller Codes[2]. For instance, $\mathcal{R}(0, m)$ codes are repetition codes, while $\mathcal{R}(1, m)$ codes are parity check codes. It is also known that a $\mathcal{R}(r, m)$ Reed Muller code has minimum distance $d_{min} = 2^{m-r}$.

Before we explain how a generator matrix for a general $\mathcal{R}(r, m)$ Reed-Muller code is formed, let us first define what it means to be a order r Reed-Muller code, and how does m come into the picture.

In a $\mathcal{R}(r, m)$ Reed-Muller code, we have m variables $x_1, x_2, ..., x_m$. Let $M = \{x_1, x_2, ..., x_m\}$ Then, within the power set $\mathcal{P}(M)$ of M, we consider only the subsets of up to size r. That is to say, out of all the subsets in $\mathcal{P}(M)$, we look at $\sum_{k=0}^{r} \binom{m}{k}$ subsets in total.

Now, when we construct the $[\sum_{k=0}^{r} \binom{m}{k}]x[2^m]$ generator matrix for the $\mathcal{R}(r, m)$ Reed-Muller code as follows:



Note that $\forall a, b \in \{0, 1\}, a * b = 1$ if $a = b = 1$, and $a * b = 0$ otherwise.[3]

For example, when $r = 2, m = 3$, we have x_1, x_2, x_3 and we only consider the subsets $\emptyset, \{x_1\}, \{x_2\}, \{x_3\}, \{x_1, x_2\}, \{x_1, x_3\}, \{x_2, x_3\}$. We obtain the following generator matrix:

$$\left[\begin{array}{c|cccccccc} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ x_1x_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ x_1x_3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ x_2x_3 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right]$$

[2]Source: `http://en.wikipedia.org/wiki/Reed-Muller_code`
[3]One can think of $a * b$ as "a (logical AND) b".

To encode a message $\mathbf{x}$ of length $\sum_{k=0}^{r} \binom{m}{k}$, we perform $\mathbf{G}^T \cdot \mathbf{x}$. While encoding Reed-Muller codes are as easy as encoding Hamming codes, it is less trivial to decode Reed-Muller codes. The high level idea of the decoding process is majority decoding. For the exact details on the general decoding of Reed-Muller codes, we refer readers to the references cited at the end of the report. We will only go through the simple example of decoding our $\mathcal{R}(1,3)$ Reed-Muller code.

2.3.2 $\mathcal{R}(1,3)$ Reed-Muller Code

In this project, we investigate a specific instance of the $\mathcal{R}(1,3)$ Reed-Muller code. Using the construction mentioned above, we obtain the following 4×8 generator matrix $\mathbf{G}$.

$$\mathbf{G} = \left[\begin{array}{c|cccccccc} \mathbf{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ x_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ x_2 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right]$$

To decode, we first calculate the characteristic vectors of each row:

- Characteristic vectors of $CV_{x_3} = \{x_1x_2, x_1\overline{x_2}, \overline{x_1}x_2, \overline{x_1x_2}\}$
- Characteristic vectors of $CV_{x_2} = \{x_1x_3, x_1\overline{x_3}, \overline{x_1}x_3, \overline{x_1x_3}\}$
- Characteristic vectors of $CV_{x_1} = \{x_2x_3, x_2\overline{x_3}, \overline{x_2}x_3, \overline{x_2x_3}\}$

After that, we find the dot product (modulo 2) between the transmitted vector $\mathbf{t}$ with each of these characteristic vectors. This process will give us 4 values in $\{0, 1\}$. We say the cofficient of the corresponding row c_i is '1' if we have more '1's than '0's out of the 4 values, and '0' otherwise.

At the end of the previous step, we would have the coefficients for the rows x_1, x_2, x_3. To calculate the coefficient for the row $\mathbf{1}$, we compute $c_1x_1 + c_2x_2 + c_3x_3 + \mathbf{t}$ (modulo 2). We say the cofficient of c_0 is '1' if we have more '1's than '0's out of the resultant vector, and '0' otherwise.

Lastly, we output the received message as (c_0, c_1, c_2, c_3).

To make things concrete, let us step through the decoding of the following transmitted message $\mathbf{t} = \mathbf{01010110}$.

Recall that:

$$\begin{aligned} \mathbf{1} &= 11111111 \\ x_1 &= 11110000 \\ x_2 &= 11001100 \\ x_3 &= 10101010 \end{aligned}$$

Then,

$$\begin{aligned} CV_{x_3} &= \{x_1x_2, x_1\overline{x_2}, \overline{x_1}x_2, \overline{x_1x_2}\} &= \{11000000, 00110000, 00001100, 00000011\} \\ CV_{x_2} &= \{x_1x_3, x_1\overline{x_3}, \overline{x_1}x_3, \overline{x_1x_3}\} &= \{10100000, 01010000, 00001010, 00000101\} \\ CV_{x_1} &= \{x_2x_3, x_2\overline{x_3}, \overline{x_2}x_3, \overline{x_2x_3}\} &= \{10001000, 01000100, 00100010, 00010001\} \end{aligned}$$

We can check that the dot products with $\mathbf{t}$ give us:

$$\begin{aligned} \mathbf{t} \cdot CV_{x_3} &= \{1,1,1,1\} \\ \mathbf{t} \cdot CV_{x_2} &= \{0,0,1,1\} \\ \mathbf{t} \cdot CV_{x_1} &= \{0,0,1,1\} \end{aligned}$$

So, $c_3 = 1, c_2 = 0, c_1 = 0$. Then $c_1x_1 + c_2x_2 + c_3x_3 + \mathbf{t}$ (modulo 2) = 11111100. So $c_0 = 1$. Hence, the decoded message is **1001**.

Since the minimum distance of the $\mathcal{R}(1,3)$ Reed-Muller code is $d_{min} = 2^{m-r} = 2^{3-1} = 4$, the maximum number of errors that it can correct is $t = \lfloor (d_{min} - 1)/2 \rfloor = \lfloor (4-1)/2 \rfloor = \lfloor 1.5 \rfloor = 1$.

As we send 8 bits for every 4 bits of information, the $\mathcal{R}(1,3)$ Reed-Muller code attains a rate of $\frac{4}{8}$. Also, since we are able to correct only up to 1 error during transmission, we have a block error whenever ≥ 2 out of the 8 transmitted bits are corrupted. In a binary symmetric channel with error probability f, that translates to a block error rate of $1 - \Pr(\text{No flipped bits}) - \Pr(1 \text{ flipped bit}) = 1 - (1-f)^8 - \binom{8}{1} f(1-f)^7$.

3 Implementation and experiments

3.1 Implementation

We used Python and the numpy package in the implementation of our project. This choice was due to the fact that our error-correcting codes were linear codes and we could make use of numpy's various useful matrix methods.

To speed up simulation, we treat the entire file as a single matrix instead of a list of vectors. For a (b, a) error-correcting code[4], we break up the input file of N bits into a $a \times \lceil \frac{N}{a} \rceil$ matrix[5]. After that, we encode the entire message matrix via a single matrix multiplication with $\mathbf{G}$ to obtain a $b \times \lceil \frac{N}{a} \rceil$ encoded matrix. To simulate a binary symmetric channel with error probability f, we simply generate a random noise $\{0,1\}$ matrix (1 with probability f) of size $b \times \lceil \frac{N}{a} \rceil$. Lastly, perform the XOR operation with the noise and encoded matrices to obtain our transmitted matrix, from which we decode the message from.

3.2 Dataset

We tested our error-correcting codes on 2 types of datasets:

- 1000 500KB files of generated using Python's OS.URANDOM
- Some images to have a visual comparison

[4]We send b bits to convey a bits of information.
[5]We pad with 0's if necessary.

3.3 Results

Let us look at the summary of some theoretical properties about each of our 3 error-correcting codes. From the above sections, our **min_dist.py** script, and the equation for maximum error correction $t = \lfloor (d_{min} - 1)/2 \rfloor$, we get the following table[6]:

Code	Rate	d_{min}	t	Block error rate on BSC_f
$(7, 4)$Hamming Code	4/7	3	1	$1 - (1-f)^7 - \binom{7}{1} f(1-f)^6$
$(9, 5)$ Suggested Code	5/9	3	1	$1 - (1-f)^9 - \binom{9}{1} f(1-f)^8$
$\mathcal{R}(1, 3)$ Reed-Muller Code	4/8	4	1	$1 - (1-f)^8 - \binom{8}{1} f(1-f)^7$

Let us consider the block error rates of our codes on a few values of f (rounded to 3 decimal places)[7].

f	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{10}$
$(7, 4)$Hamming Code	0.938	0.737	0.555	0.423	0.33	0.264	0.215	0.178	0.15
$(9, 5)$ Suggested Code	0.98	0.857	0.7	0.564	0.457	0.376	0.313	0.264	0.225
$\mathcal{R}(1, 3)$ Reed-Muller Code	0.965	0.805	0.633	0.497	0.395	0.32	0.264	0.221	0.187

We see that the $(7, 4)$ Hamming Code seems to perform the best while the suggested $(9, 5)$ code is the worst of the 3. We verify this with empirical results. First we generated 1000 500KB files of bytes using Python's OS.URANDOM. For each of the error-correcting schemes, we encode the files, simulate passing through a noisy BSC_f and then decode it. The following table shows the *mean block error rates* over 1000 files.

f	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$	$\frac{1}{8}$	$\frac{1}{9}$	$\frac{1}{10}$
$(7, 4)$Hamming Code	0.938	0.802	0.684	0.590	0.518	0.460	0.414	0.376	0.344
$(9, 5)$ Suggested Code	0.969	0.868	0.763	0.672	0.598	0.537	0.487	0.445	0.410
$\mathcal{R}(1, 3)$ Reed-Muller Code	0.937	0.751	0.575	0.443	0.348	0.279	0.228	0.190	0.160

[6] BSC_f means a binary symmetric channel with bit error probability f

[7] Obtained using our script **block_error.py**.

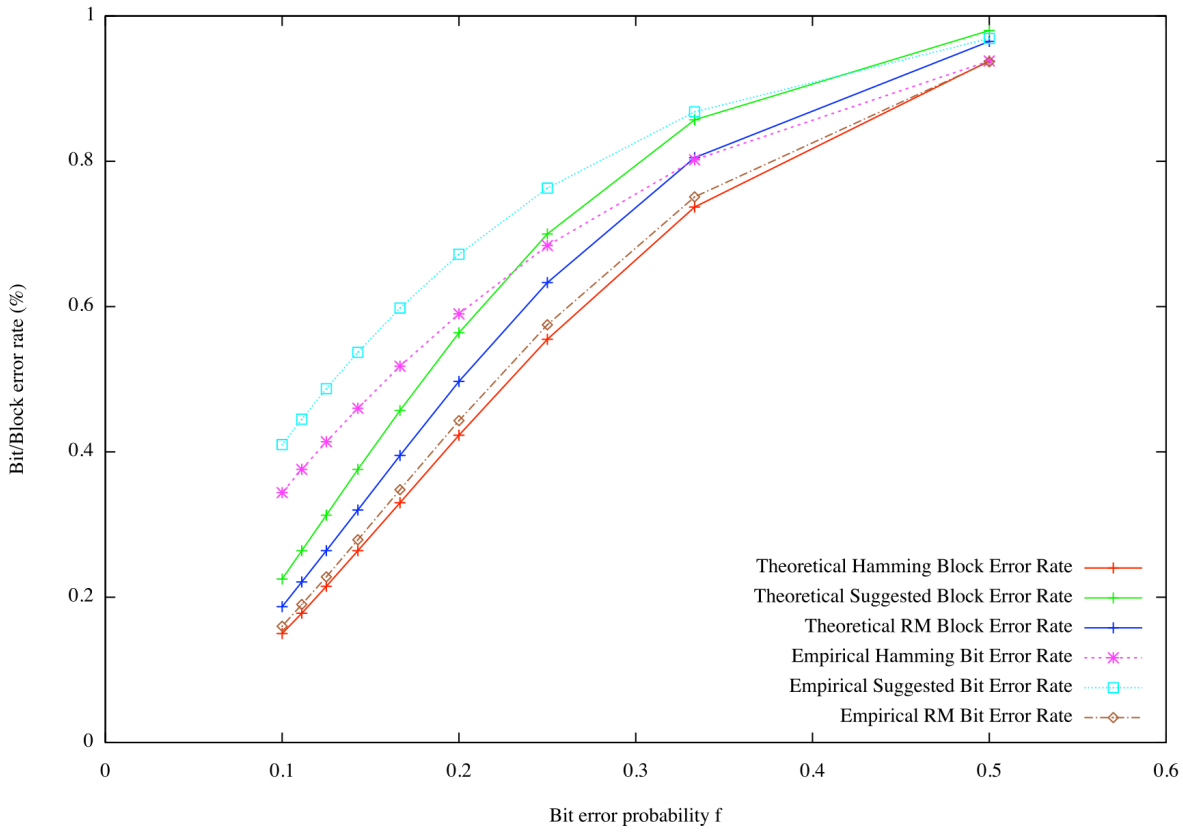


Figure 2: Relationship between f and bit/block error rates, for $f = \{\frac{1}{2}, \frac{1}{3}, ..., \frac{1}{10}\}$

The empirical results support the general trends described by the theoretical analysis. However, it is interesting to observe that the Reed-Muller code performs empirically better than the Hamming code despite the latter having a lower theoretical block error rate.

We conclude the report with image simulations. This is to visualize the difference in performance of the 3 error-correcting codes. We enlist the help of 2 different images - A Pokemon image[8] and a smiley face[9]. The images are converted to .ppm format before the simulation is ran[10]. The image tilings shown below are ordered in the following manner:

Original Image	Hamming Code
Suggested Code	Reed-Muller Code

[8]Source: `http://pt.todoprogramas.com/photos/2009/2/1024.877.273.1a.500.jpg`

[9]Source: `http://www.imgion.com/images/01/Blurred-smiley.gif`

[10]We fix any corrupted .ppm headers using *dd if=<original_image> of=<new_image> bs=15 count=1 conv=notrunc*

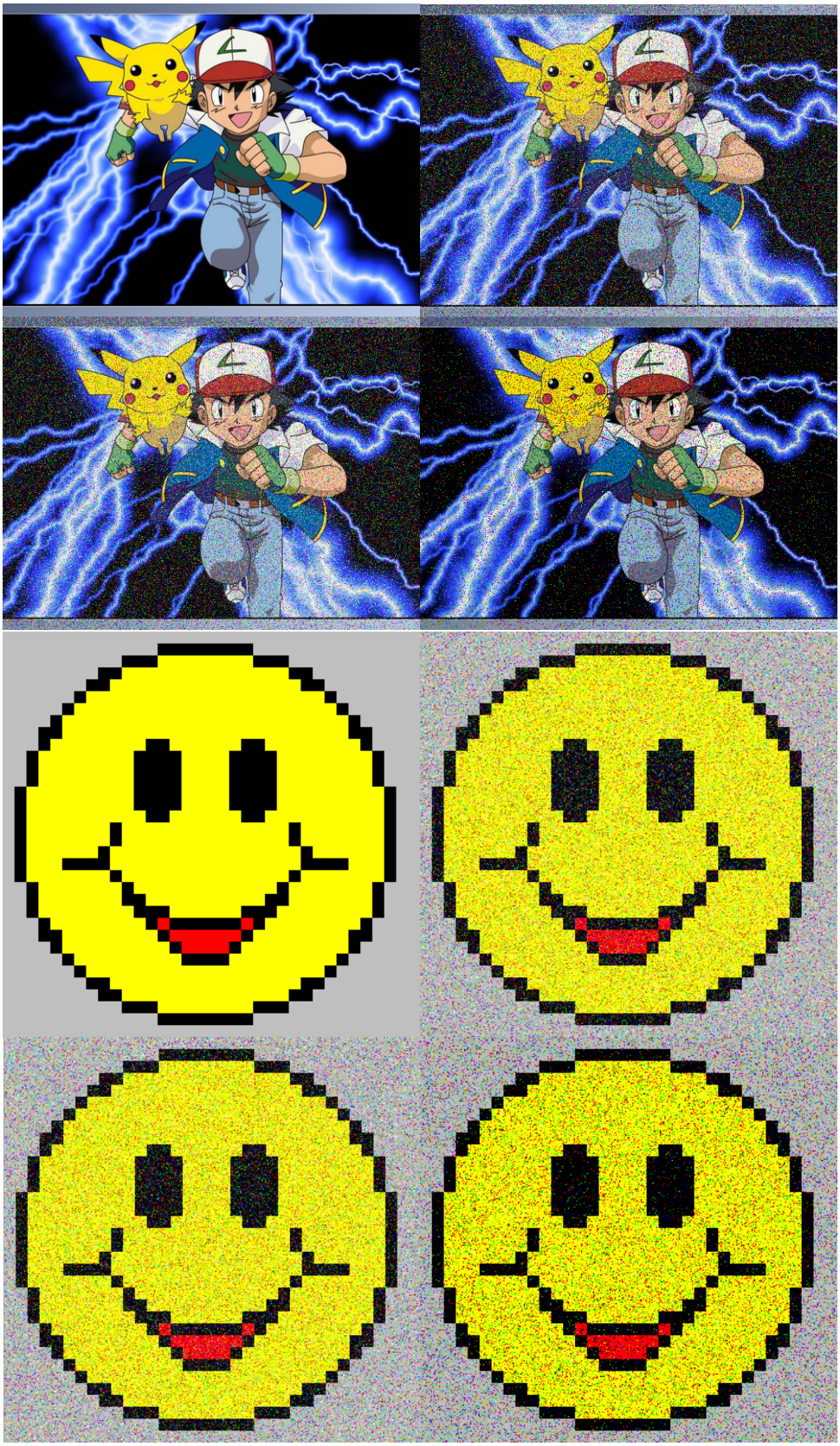


Figure 3: Simulation with $f = \frac{1}{10}$

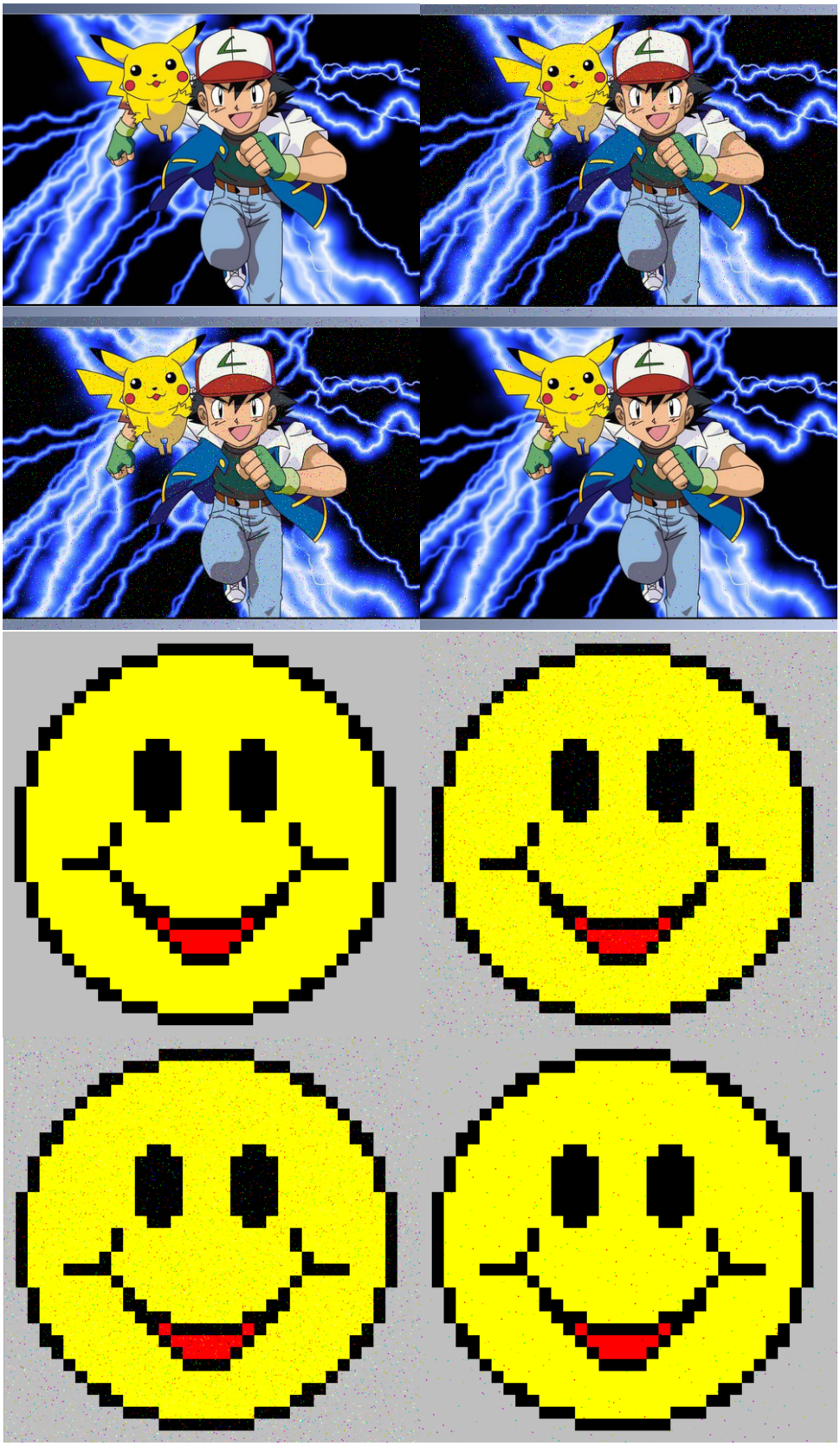


Figure 4: Simulation with $f = \frac{1}{100}$

4 References

Source code is available on Github: `https://github.com/sozos/CodingSchemes`

David J.C. MacKay. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge, UK: Cambridge University Press

Ben Cooke. *Reed-Muller Error Correcting Codes*. MIT Undergraduate Journal of Mathematics. Retrived from `http://www-math.mit.edu/phase2/UJM/vol1/COOKE7FF.PDF`

Irving S. Reed. (1954). A class of multiple-error-correcting codes and the decoding scheme. *IEEE Trans. on Information Theory*, 4:38-49.