Error correcting codes from scratch

Oleg Mazurov

November 26, 2017

1 Reed-Solomon codes

1.1 A very simple code

I have a collection of k numbers I want to store: $y_0, y_1, ..., y_{k-1}$. I put the numbers into numbered boxes: y_0 goes into box 0, y_1 - into box 1, etc.

My first concern is what happens if one of those boxes is lost. A simple solution to that problem is to create a new number, y_k, so that the sum of all numbers including the new one is 0 and put that number into a new box k:

$$\sum_{i=0}^{k} y_i = 0 \tag{1}$$

$$y_k = -\sum_{i=0}^{k-1} y_i \tag{2}$$

Now, if any one box is lost, I can easily restore the number from that box using (1):

$$y_j = -\sum_{i \neq j} y_i \tag{3}$$

I notice that (3) is the same as (2) as if when encoding y_k I was actually restoring it.

1.2 One step up

Naturally, I now want even more security and think of devising a scheme that would allow me to restore any two lost boxes. Knowing that I was able to solve the problem for one lost box with only one extra box, I hope to do the same here with only two extra boxes.

Box numbers come to the rescue. Now I add two more numbers so that the total sum of all numbers is zero and a new sum of stored numbers multiplied

by their box number is also zero:

$$\begin{aligned} \sum_{i=0}^{k+1} y_i &= 0, \\ \sum_{i=0}^{k+1} i * y_i &= 0 \end{aligned} \tag{4}$$

The two new numbers can be computed from the two equations above:

$$\begin{aligned} y_k + y_{k+1} &= -\sum_{i=0}^{k-1} y_i \\ k * y_k + (k+1) * y_{k+1} &= -\sum_{i=0}^{k-1} i * y_i \end{aligned} \tag{5}$$

or

$$\begin{aligned} y_k &= -(k+1) * \sum_{i=0}^{k-1} y_i + \sum_{i=0}^{k-1} i * y_i = -\sum_{i=0}^{k-1} (k+1-i) * y_i \\ y_{k+1} &= -\sum_{i=0}^{k-1} i * y_i + k * \sum_{i=0}^{k-1} y_i = -\sum_{i=0}^{k-1} (i-k) * y_i \end{aligned} \tag{6}$$

I notice again that there is nothing special about y_k and y_{k+1} in (4) and I can solve the equations for any two unknown y_{j_0} and y_{j_1}:

$$\begin{aligned} y_{j_0} &= -\sum_{i \neq j_0, i \neq j_1} \frac{(i-j_1)}{(j_0-j_1)} y_i \\ y_{j_1} &= -\sum_{i \neq j_0, i \neq j_1} \frac{(i-j_0)}{(j_1-j_0)} y_i \end{aligned} \tag{7}$$

Equations (6) are thus generalized to (7), which look symmetrical, as one would expect. Division is now an essential part of the lost number reconstruction process - something that was not explicit in (6).

1.3 The whole ladder

Encouraged rather than frightened by (7) I now want to restore an arbitrary number m of lost boxes by adding only m extra boxes. That indeed becomes

possible if I extend (4) to the following m sums:

$$\begin{aligned}
&\sum_{i=0}^{k+m-1} y_i = 0,\\
&\sum_{i=0}^{k+m-1} i * y_i = 0,\\
&\sum_{i=0}^{k+m-1} i^2 * y_i = 0,\\
&...\\
&\sum_{i=0}^{k+m-1} i^{m-1} * y_i = 0
\end{aligned} \tag{8}$$

How to solve it for any m unknown $y_{j_0}, y_{j_1}...y_{j_{m-1}}$?

If I multiply equations (8) by arbitrary numbers $p_0, p_1, ...p_{m-1}$ and add them together, I get:

$$\begin{aligned}
&p_0 * \sum_{i=0}^{k+m-1} y_i + p_1 * \sum_{i=0}^{k+m-1} i * y_i + ... + p_{m-1} * \sum_{i=0}^{k+m-1} i^{m-1} * y_i = 0,\\
&\sum_{i=0}^{k+m-1} \left[p_0 * y_i + p_1 * i * y_i + ... + p_{m-1} * i^{m-1} * y_i\right] = 0,\\
&\sum_{i=0}^{k+m-1} y_i * \left[p_0 + p_1 * i + ... + p_{m-1} * i^{m-1}\right] = 0,\\
&\sum_{i=0}^{k+m-1} y_i * P(i) = 0,
\end{aligned} \tag{9}$$

where $P(i) = \sum_{l=0}^{m-1} p_l * i^l$ is an arbitrary polynomial of degree $m-1$. This is the key to solving (8).

For any m unknown $y_{j_0}, y_{j_1}...y_{j_{m-1}}$ I can construct a special polynomial $P_0(i) = (i-j_1)(i-j_2)...(i-j_{m-1})$, which evaluates to 0 at all box numbers $j_1, j_2, ...j_{m-1}$ but not j_0. Using that polynomial in (9) gives:

$$\begin{aligned}
&\sum_{i=0}^{k+m-1} y_i * P_0(i) = 0,\\
&y_{j_0} * P_0(j_0) + \sum_{i \neq j_s} y_i * P_0(i) = 0,\\
&y_{j_0} = -\sum_{i \neq j_s} y_i * \frac{P_0(i)}{P_0(j_0)},\\
&y_{j_0} = -\sum_{i \neq j_s} y_i * \frac{(i-j_1)(i-j_2)...(i-j_{m-1})}{(j_0-j_1)(j_0-j_2)...(j_0-j_{m-1})}
\end{aligned} \tag{10}$$

To recover other unknown y_{j_l} I construct similar polynomials $P_l(i)$:

$$\begin{aligned} P_l(i) &= (i-j_0)...(i-j_{l-1})(i-j_{l+1})...(i-j_{m-1}) \\ y_{j_l} &= -\sum_{i \neq j_s} y_i * \frac{P_l(i)}{P_l(j_l)} = -\sum_{i \neq j_s} y_i * \prod_{s \neq l} \frac{(i-j_s)}{(j_l-j_s)} \end{aligned} \tag{11}$$

Every unknown number is a linear combination of all the known numbers.

What I got here is a Reed-Solomon code with a procedure to restore erasures - missed values with known locations (box numbers). As the defining equations (8) represent a Vandermonde matrix, I'll call this a Vandermondian approach to defining Reed-Solomon codes.

1.4 Generalizing indexes

So far I have been using non-negative whole numbers to mark my boxes. From (11), it should be obvious that any distinct numbers $x_0, x_1, ..., x_{k+m-1}$ would serve as well. Introducing the total number of boxes $n = k + m$, I can write:

$$y_{j_l} = -\sum_{i=0, i \neq j_s}^{n-1} y_i * \prod_{s \neq l} \frac{(x_i - x_{j_s})}{(x_{j_l} - x_{j_s})} \tag{12}$$

1.5 Dealing with errors

Now that lost, or erased, numbers got their ultimate treatment, my next concern becomes what if some of my stored numbers get corrupted over time.

The simplest code from 1.1 can detect an error as long as the sum of all stored numbers is not 0 but is helpless in determining which number, even if only one, is wrong. That, in fact, is exactly my goal here: to determine which numbers are wrong. Once I know that, I can treat them as lost numbers and reconstruct them using the previously described algorithm.

Is my code with two extra boxes from 1.2 any better in that regard? It is. Computing sums in (4) will reveal that my code has been violated:

$$\begin{aligned} \sum_{i=0}^{k+1} y_i &= S_0 \neq 0, \\ \sum_{i=0}^{k+1} i * y_i &= S_1 \neq 0 \end{aligned} \tag{13}$$

However, if only one number is corrupted, say, y_{j_0} turns into $y'_{j_0} = y_{j_0} + e_{j_0}$, then $S_0 = e_{j_0}$ and $S_1 = j_0 * e_{j_0}$, from which I can easily figure out:

$$\begin{aligned} j_0 &= \frac{S_1}{S_0}, \\ e_{j_0} &= S_0 \end{aligned} \tag{14}$$

Both the box number and the error by which the stored value should be adjusted are algebraically computed from the code syndromes, as S_0 and S_1 are called.

1.6 More errors

How do I recover from more errors in the general case? Say, I have computed m code syndromes:

$$\begin{aligned}
&\sum_{i=0}^{n-1} y_i = S_0, \\
&\sum_{i=0}^{n-1} i * y_i = S_1, \\
&\sum_{i=0}^{n-1} i^2 * y_i = S_2, \\
&... \\
&\sum_{i=0}^{n-1} i^{m-1} * y_i = S_{m-1}
\end{aligned} \tag{15}$$

Remembering how I could use (9) to get rid of unwanted values in the sums, I now see how to reapply the trick to exclude all values with errors. If e is the number of errors and $P_e(i) = (i - j_0)(i - j_1)...(i - j_{e-1})$ is the error location polynomial (another name is 'the error locator polynomial'), then

$$\sum_{i=0}^{n-1} y_i * P_e(i) = 0, \tag{16}$$

still holds as all errors at locations j_s are neutralized by $P_e(j_s) = 0$. By rewriting the error location polynomial as $P_e(i) = \sum_{l=0}^{e} p_l * i^l$, I get

$$\sum_{i=0}^{n-1} y_i * \sum_{l=0}^{e} p_l * i^l = \sum_{l=0}^{e} p_l \sum_{i=0}^{n-1} y_i * i^l = \sum_{l=0}^{e} p_l * S_l = 0, \tag{17}$$

Moreover, if $e < m-1$, I can write (16) for polynomials $i*P_e(i)$, $i^2*P_e(i)$,...,$i^{m-e-1}*P_e(i)$ and get a set of linear equations for p_l similar to (17):

$$\begin{aligned}
&\sum_{l=0}^{e} p_l * S_l = 0, \\
&\sum_{l=0}^{e} p_l * S_{l+1} = 0, \\
&... \\
&\sum_{l=0}^{e} p_l * S_{l+m-e-1} = 0
\end{aligned} \tag{18}$$

If the number of equations is greater or equal to the number of unknown p_l, I should be able to solve it and find $P_e(i)$. As $p_e = 1$, I have e unknowns and $m-e$

equations. The condition $e \leq m-e$, or $e \leq m/2$, thus determines the solvability of (18). While more sophisticated algorithms for finding roots of $P_e(i)$ exist, I may simply compute the polynomial for each $0..n-1$ (or for each generalized index $x_0...x_{n-1}$) and write down those for which the computed value is zero. If I get all e distinct roots, I have successfully reconstructed all error locations. Now I can apply the erasure decoding algorithm as in (11) and be done with error decoding.

1.7 Solving the linear equations

One question about solving the linear equations (18) is how should I go about constructing the matrix and the result vector as I don't know how many errors there are. Trying out different values of e until the solution is found would work but I think I could do better than that. The main idea is that I can start Gauss elimination assuming the maximum $e = \lfloor m/2 \rfloor$ and, when there is no possibility to continue due to a smaller number of errors, have the matrix ready for evaluation of the unknowns. For example, if $m = 6$ the full matrix describing (18) is

$$\begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ S_1 & S_2 & S_3 & S_4 \\ S_2 & S_3 & S_4 & S_5 \end{pmatrix}$$

If there is only one error, after I eliminate elements below the diagonal in the first column I end up with matrix

$$\begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} => \begin{pmatrix} S_0 & S_1 \end{pmatrix}$$

And with two errors, I'll stop after having cleared the second column:

$$\begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ 0 & s_{22} & s_{23} & s_{24} \\ 0 & 0 & 0 & 0 \end{pmatrix} => \begin{pmatrix} S_0 & S_1 & S2 \\ 0 & s_{22} & s_{23} \end{pmatrix}$$

1.8 Optimization

Three steps in my error decoding algorithm are $O(nm)$ time-wise: computing syndromes, finding roots of the error location polynomial $P_e(i)$, and erasure decoding. While optimizing some steps won't change the asymptotic behavior of the algorithm, it may still help with the constant in $O(nm)$ and improve overall performance. Here, I'll try to avoid the full scan of the codeword in the last step, erasure decoding.

Taking another look at (16) and noticing its resemblance to (10), I figure that if I exclude error location j_0 from $P_e(i)$, i.e. if I use $P_0(i) = P_e(i)/(i - j_0)$ in (16) the sum won't evaluate to zero only because y_{j_0} is not correct. In other words, if e_{j_0} is the error value at location j_0, then:

$$e_{j_0} * P_0(j_0) = \sum_{i=0}^{n-1} y_i * P_0(i),$$

$$e_{j_0} * P_0(j_0) = \sum_{i=0}^{n-1} y_i * \sum_{l=0}^{e} p_{0l} * i^l = \sum_{l=0}^{e} p_{0l} \sum_{i=0}^{n-1} y_i * i^l = \sum_{l=0}^{e} p_{0l} * S_l, \qquad (19)$$

$$e_{j_0} = \frac{1}{P_0(j_0)} \sum_{l=0}^{e} p_{0l} * S_l = \frac{\sum_{l=0}^{e} p_{0l} * S_l}{\sum_{l=0}^{e} p_{0l} * j_0^l}$$

Computing all error values is quadratic in e but for $e \leq m/2 < n/2$ it's better than $n * m$.

How to compute coefficients of $P_0(i)$? One way is to multiply $(i - j_l), l = 1..e - 1$. Another one is to divide $P_e(i)$ by $(i - j_0)$. The remainder must be zero if j_0 is a root of $P_e(i)$. It now occurs to me that the two steps of my error decoding algorithm, root finding and erasure decoding, can actually be combined into one: for each index $j = 0..n - 1$ (or generalized index $x_0..x_{n-1}$) try to divide $P_e(i)$ by $(i-j)$; if successful, use the quotient to compute the error value according to (19).

1.9 A practical perspective

1.10 Another arithmetic

1.11 Yet another arithmetic

1.12 A different approach

2 Going beyond the $m/2$ boundary

3 Multiple dimensions